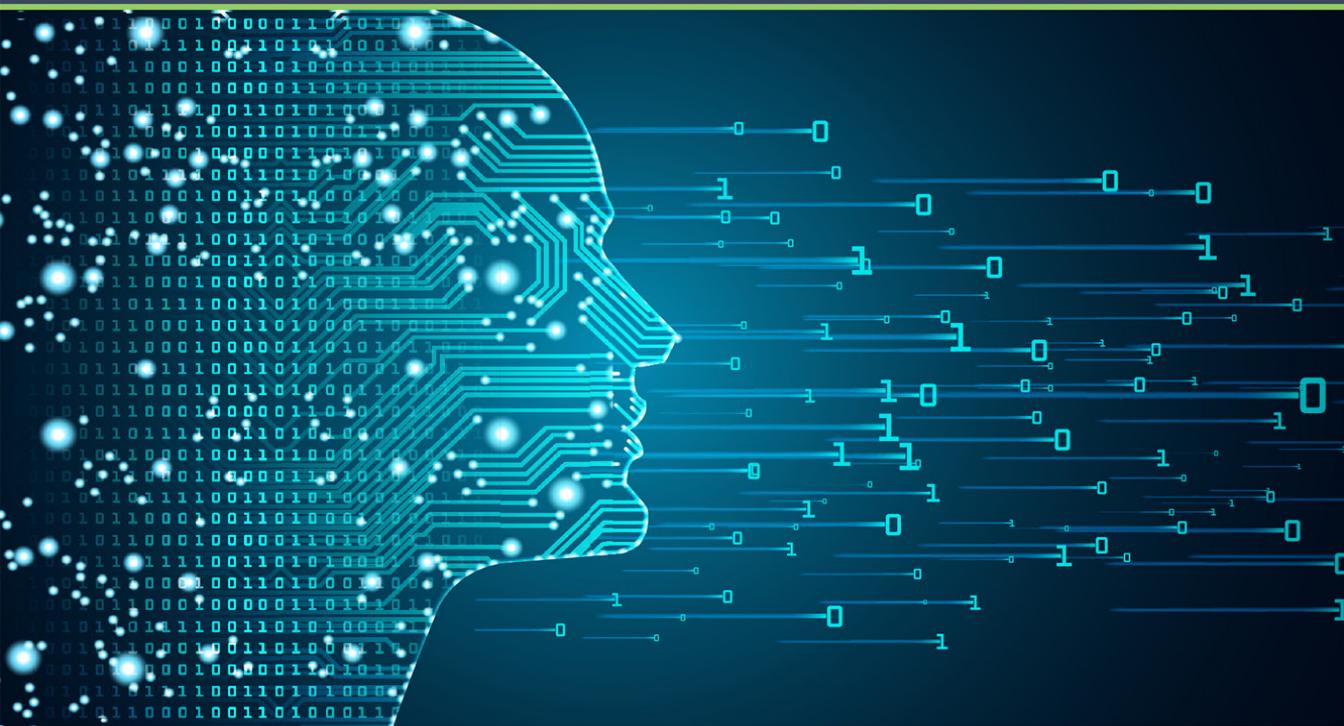


INTRODUCTION TO PYTHON FOR DIGITAL HUMANISTS



WILLIAM J. B. MATTINGLY



CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Introduction to Python for Humanists

This book will introduce digital humanists at all levels of education to Python. It provides background and guidance on learning the Python computer programming language, and as it presumes no knowledge on the part of the reader about computers or coding concepts allows the reader to gradually learn the more complex tasks that are currently popular in the field of digital humanities. This book will be aimed at undergraduates, graduates, and faculty who are interested in learning how to use Python as a tool within their workflow. An Introduction to Python for Digital Humanists will act as a primer for students who wish to use Python, allowing them to engage with more advanced textbooks. This book fills a real need, as it is first Python introduction to be aimed squarely at humanities students, as other books currently available do not approach Python from a humanities perspective. It will be designed so that those experienced in Python can teach from it, in addition to allowing those who are interested in being self-taught can use it for that purpose.

Key Features:

- Data analysis
- Data science
- Computational humanities
- Digital humanities
- Python
- Natural language processing
- Social network analysis
- App development

Chapman & Hall/CRC

The Python Series

About the Series

Python has been ranked as the most popular programming language, and it is widely used in education and industry. This book series will offer a wide range of books on Python for students and professionals. Titles in the series will help users learn the language at an introductory and advanced level, and explore its many applications in data science, AI, and machine learning. Series titles can also be supplemented with Jupyter notebooks.

Image Processing and Acquisition using Python, Second Edition

Ravishankar Chityala, Sridevi Pudipeddi

Python Packages

Tomas Beuzen and Tiffany-Anne Timbers

Statistics and Data Visualisation with Python

Jesús Rogel-Salazar

Introduction to Python for Humanists

William J.B. Mattingly

For more information about this series please visit: <https://www.routledge.com/Chapman--HallCRC-The-Python-Series/book-series/PYTH>

Introduction to Python for Humanists

William J.B. Mattingly



CRC Press

Taylor & Francis Group

Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Front cover image: Ryzhi/Shutterstock

First edition published 2023
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

© 2023 William Mattingly

Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, access www.copyright.com or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. For works that are not available on CCC please contact mpkbookspermissions@tandf.co.uk

Trademark notice: Product or corporate names may be trademarks or registered trademarks and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Names: Mattingly, William J. B., author.

Title: Introduction to Python for humanists / William J.B. Mattingly.

Description: First edition. | Boca Raton : CRC Press, [2023] | Series: Chapman & Hall/CRC the Python series | Includes bibliographical references and index. | Identifiers: LCCN 2022056310 (print) | LCCN 2022056311 (ebook) | ISBN 9781032377902 (hbk) | ISBN 9781032378374 (pbk) | ISBN 9781003342175 (ebk)

Subjects: LCSH: Python (Computer program language) | Engineering--Data processing. | Science--Data processing. | Computer programming.

Classification: LCC QA76.73.P98 M383 2023 (print) | LCC QA76.73.P98 (ebook) | DDC 005.13/3--dc23/eng/20221202

LC record available at <https://lccn.loc.gov/2022056310>

LC ebook record available at <https://lccn.loc.gov/2022056311>

ISBN: 978-1-032-37790-2 (hbk)

ISBN: 978-1-032-37837-4 (pbk)

ISBN: 978-1-003-34217-5 (ebk)

DOI: 10.1201/9781003342175

Typeset in Palatino LT Std
by Deanta Global Publishing Services, Chennai, India

Contents

Preface	xv
Acknowledgments	xix
About the Author	xxi
I The Basics of Python	1
1 Introduction to Python	3
1.1 Introduction to Python	3
1.1.1 Why Should Humanists Learn to Code?	3
1.1.1.1 The Timeline for Starting a Large Digital History Project	3
1.1.1.2 The Self-Reliant Digital Humanist	4
1.1.1.3 Other Benefits	5
1.1.2 What Is Python?	6
1.1.3 Why Python?	7
1.2 Installing Python	7
1.2.1 Trinket	7
1.2.2 Using Google Colab	7
1.2.3 Using Binder from JupyterBook	9
1.2.4 Using Jupyter Notebooks Online	11
1.2.5 Installing Python Locally	13
1.2.5.1 Download Anaconda Navigator	14
1.2.5.2 Using Anaconda Navigator	14
1.2.5.3 Installing JupyterLab	14
1.2.6 Conclusion	15
1.3 Coding Basics	16
1.3.1 The Print Function	16
1.3.2 Objects	17
1.3.3 Variables	17
1.3.4 Case Sensitivity	18
1.3.5 Reserved Words	18
1.3.6 Built-in Types	19
1.3.7 Type Function	21
1.3.8 Bugs	21
2 Data and Data Structures	23
2.1 Introduction to Data	23
2.1.1 What Is Data?	23
2.1.2 Strings	24
2.1.3 Working with Strings as Data	25
2.1.3.1 Upper Method	26

2.1.3.2	Lower Method	26
2.1.3.3	Capitalize Method	26
2.1.3.4	Replace Method	26
2.1.3.5	Split Method	27
2.1.4	Numbers (Integers and Floats)	28
2.1.5	Working with Numbers as Data	29
2.1.6	Booleans	30
2.1.7	Conclusion	30
2.2	Introduction to Data Structures	30
2.2.1	Data Structures	30
2.2.2	Lists	31
2.2.2.1	Indexing a List	31
2.2.3	Tuples	32
2.2.4	Mutability vs Immutability	33
2.2.5	Sets (Bonus Data Structure)	33
2.2.6	Dictionaries	34
2.2.6.1	Indexing Dictionaries	34
3	Loops and Logic	37
3.1	Introduction to Loops	37
3.1.1	What Are Loops?	37
3.1.2	For Loops	37
3.1.3	List Comprehension	39
3.1.4	Indexing a List with and without Enumerate	40
3.1.5	Operators	41
3.1.6	While Loops	42
3.2	Conditionals	43
3.2.1	If Statement	43
3.2.2	Else Statement	44
3.2.3	Elif and the 'in' Operator with Strings	44
3.2.4	Conditionals and Lists with 'in' and 'not in'	45
3.2.5	Conclusion	46
4	Formal Coding: Functions, Classes, and Libraries	47
4.1	Functions	47
4.1.1	Introduction	47
4.1.2	Functions in Action	47
4.1.3	Docstrings	49
4.1.4	Functions with Multiple Arguments	49
4.1.5	Keyword Argument	50
4.1.6	Keyword Arbitrary Arguments	50
4.1.7	Conclusion	51
4.1.8	Answer for Result	51
4.2	Classes	51
4.2.1	Introduction	51
4.2.2	Creating a Class	52
4.2.3	Adding Functions to a Class	53
4.3	Libraries in Python	54
4.3.1	Introduction	54
4.3.2	How to Install Python Libraries	54

4.3.3	How to Import a Library	55
4.3.4	Conclusion	56
5	Working with External Data	57
5.1	Working with Textual Data	57
5.1.1	Introduction	57
5.1.2	The “With” Statement	57
5.1.3	How to Open a Text File	57
5.1.4	How to Write Data to a Text File	58
5.2	Working with JSON Data	59
5.2.1	Introduction	59
5.2.2	Writing JSON Data with <code>json.dump()</code>	59
5.2.3	Reading JSON data with <code>json.load()</code>	60
5.3	Working with Multiple Files	60
5.3.1	Introduction	60
5.3.2	Working with Glob	60
5.3.3	Grabbing Multiple Nested Directories	61
5.3.4	Walking a Directory	61
5.3.5	Conclusion	63
6	Working with Data on the Web	65
6.1	Introduction to HTML	65
6.1.1	Introduction	65
6.1.2	Diving into HTML	65
6.1.3	Understanding Attributes	66
6.1.4	Parsing HTML with BeautifulSoup	66
6.1.5	How to Find a Website’s HTML	69
6.2	Scraping Web Pages with Requests and BeautifulSoup	70
6.2.1	Introduction	70
6.2.2	Requests	70
6.2.3	BeautifulSoup	71
II	Data Analysis with Pandas	75
7	Introduction to Pandas	77
7.1	Introduction to Pandas	77
7.1.1	What Is Pandas?	77
7.1.2	Why Use Pandas?	77
7.1.3	How to Install Pandas	78
7.1.4	How to Import Pandas	78
7.2	The Basics of Pandas	78
7.2.1	How to Create a DataFrame from a Dictionary	78
7.2.2	How to Display a DataFrame	79
7.2.3	How to Save DataFrame to CSV	80
7.2.4	How to Read DataFrame from CSV	80
7.2.5	How to Save DataFrame to JSON	81
7.2.6	How to Add a Column to the DataFrame	81
7.2.7	How to Grab a Specific Column	82
7.2.8	How to Convert a Column to a List	83
7.2.9	Isolating Unique Values in a Column	83
7.2.10	How to Grab a Specific Row of a DataFrame with <code>iloc</code>	83

7.2.11 Iterating over a DataFrame with df.iterrows()	84
7.2.12 Conclusion	84
8 Working with Data in Pandas	85
8.1 Finding Data in DataFrame	85
8.1.1 About the Titanic Dataset	85
8.1.2 How to Find Column Data	85
8.1.3 How to Get a Quick Sense of the Dataset with df.head()	86
8.1.4 How to Grab a Specific Range of Rows with df.iloc[]	88
8.1.5 How to Get a Quick Quantitative Understanding of the Dataset with describe()	89
8.1.6 How to Find Specific Information in the Dataset with df.loc	89
8.1.7 How to Query with “OR” () on a DataFrame	93
8.2 Organizing the DataFrame	95
8.2.1 How to Sort Data By Single Column	95
8.2.2 How to Reverse Sort Data by Single Column	97
8.2.3 How to Sort Data by Multiple Columns	98
8.2.4 How to Sort Data by Multiple Columns with Different Values Organized Differently	100
8.3 Cleaning the DataFrame	101
8.3.1 How to Drop a Column in Pandas DataFrame	101
8.3.2 How to Remove Rows That Have NaN in Any Column	104
8.3.3 How to Remove Rows That Have NaN in a Specific Column	104
8.3.4 How to Convert DataFrame Data Types (from Float to Int)	105
8.3.5 Conclusion	107
9 Searching for Data	109
9.1 Advanced Searching on Strings	109
9.1.1 Finding Features within a String	109
9.1.2 Finding Strings That Don’t Contain Feature	110
9.1.3 Using RegEx with Pandas	110
9.2 Filter and Querying	112
9.2.1 Introduction	112
9.2.2 The Filter Function	113
9.2.3 The Query Function	116
9.3 Grouping with groupby()	121
9.3.1 Introduction	121
9.3.2 groupby()	121
9.3.3 Quantitative Analysis with .count() and .sum()	122
9.3.4 Working with Multiple Groups	123
9.3.5 Groupings with Many Subsets	124
10 Advanced Pandas	125
10.1 Plotting Data with Pandas	125
10.1.1 Importing the DataFrame	125
10.1.2 Bar and Barh Charts with Pandas	126
10.1.3 Pie Charts with Pandas	128
10.1.4 Scatter Plots with Pandas	130
10.2 Graphing Network Data with Pandas	138
10.2.1 Getting the Data from Pandas to NetworkX	138
10.2.2 Graphing the Data	139

10.2.3	Customize the Graph	139
10.3	Time Series Data	141
10.3.1	What Is Time Series Data	141
10.3.2	About the Dataset	141
10.3.3	Cleaning the Data from Float to Int	143
10.3.4	Convert to Time Series DateTime in Pandas	149
III	Natural Language Processing with spaCy	155
11	Introduction to Spacy	157
11.1	The Basics of spaCy	157
11.1.1	What Is spaCy?	157
11.1.2	How to Install spaCy	158
11.1.3	Containers	158
11.2	Getting Started with spaCy and Its Linguistic Annotations	159
11.2.1	Importing spaCy and Loading Data	159
11.2.2	Creating a Doc Container	160
11.2.3	Sentence Boundary Detection (SBD)	162
11.2.4	Token Attributes	163
11.2.4.1	Text	164
11.2.4.2	Head	164
11.2.4.3	Left Edge	164
11.2.4.4	Right Edge	164
11.2.4.5	Entity Type	164
11.2.4.6	Ent IOB	165
11.2.4.7	Lemma	165
11.2.4.8	Morph	165
11.2.4.9	Part of Speech	165
11.2.4.10	Syntactic Dependency	166
11.2.4.11	Language	166
11.2.5	Part-of-Speech (POS) Tagging	166
11.2.6	Named Entity Recognition (NER)	167
11.2.7	Conclusion	168
11.3	spaCy's Pipelines	168
11.3.1	Standard Pipes (Components and Factories) Available from spaCy	168
11.3.1.1	Attribute Rulers	169
11.3.1.2	Matchers	170
11.3.2	How to Add Pipes	170
11.3.3	Examining a Pipeline	171
11.3.4	Conclusion	172
12	Rules-Based spaCy	173
12.1	The EntityRuler	173
12.1.1	Introduction to spaCy's EntityRuler	173
12.1.2	Demonstration of EntityRuler in Action	173
12.1.3	Introducing Complex Rules and Variance to the EntityRuler (Advanced)	177
12.2	The Matcher	177
12.2.1	Introduction	177
12.2.2	A Basic Example	178
12.2.3	Attributes Taken by Matcher	179

12.2.4	Applied Matcher	180
12.2.4.1	Grabbing all Proper Nouns	181
12.2.4.2	Improving it with Multi-Word Tokens	181
12.2.4.3	Greedy Keyword Argument	181
12.2.4.4	Adding in Sequences	182
12.3	The PhraseMatcher	182
12.3.1	Introduction	182
12.3.2	Setting a Custom Attribute	184
12.3.3	Adding a Function with <code>on_match</code>	185
12.4	Using RegEx with spaCy	186
12.4.1	What Is Regular Expressions (RegEx)?	186
12.4.2	The Strengths of RegEx	186
12.4.3	The Weaknesses of RegEx	186
12.4.4	How to Use RegEx in Python	186
12.4.5	How to Use RegEx in spaCy	188
12.5	Working with Multi-Word Token Entities and RegEx in spaCy 3x	191
12.5.1	Key Concepts in This Notebook	191
12.5.2	Problems with Multi-Word Tokens in spaCy as Entities	191
12.5.3	Extract Multi-Word Tokens	191
12.5.4	Reconstruct Spans	192
12.5.5	Inject the Spans into the doc.ents	192
12.5.6	Give Priority to Longer Spans	193
13	Solving a Domain-Specific Problem: A Case Study with Holocaust NER	195
13.1	Cultivating Good Datasets for Entities	195
13.1.1	Introduction to Datasets	195
13.1.2	Acquiring the Data	195
13.1.3	United States Holocaust Memorial Museum	195
13.1.4	Normalizing Data	196
13.2	The Challenges of Holocaust NER	197
13.2.1	An Overview of the Problems	197
13.2.2	Ethical	197
13.2.3	Linguistic	197
13.2.4	Toponyms	198
13.3	Creating a Rules-Based Pipeline for Holocaust Documents	198
13.3.1	Creating a Blank spaCy Model	199
13.3.2	Creating EntityRulers	199
13.3.3	Creating Function for Matching RegEx	200
13.3.4	Add Pipe for Finding Streets	202
13.3.5	Creating a Pipe for Finding Ships	203
13.3.6	Create Pipe for Identifying a Military Personnel	205
13.3.7	Create Pipe for Identifying Spouses	205
13.3.8	Creating a Pipe for Finding Ghettos	206
13.3.9	Creating a Geography Pipe	206
13.3.10	Seeing the Pipes at Work	207
14	Topic Modeling: Concepts and Theory	211
14.1	What Is Topic Modeling?	211
14.1.1	Rules-Based Methods	212
14.1.2	Machine Learning-Based Methods	212

14.1.3	Why Use Topic Modeling?	213
14.2	Topics and Clusters	213
14.2.1	What Are Topics?	213
14.2.2	What Are Clusters?	214
14.3	Bigrams and Trigrams	214
14.3.1	Textual Ambiguity	214
14.3.2	Bigrams	214
14.3.3	Trigrams	215
14.3.4	Why Are These Important?	215
14.4	LDA Topic Modeling	215
14.4.1	Process of Topic Modeling	215
14.4.2	Knowing the Total Number of Topics	217
14.4.3	Applying a Topic Model	218
14.4.4	Summary of Key Issues with LDA Topic Modeling	218
14.5	Creating LDA in Python	218
14.5.1	Importing the Required Libraries and Data	218
14.5.2	Cleaning Documents	220
14.5.3	Create ID-Word Index	220
14.5.4	Creating LDA Topic Model	222
14.5.5	Analyze a Document	222
14.5.6	Analyze the Topic Model	224
14.6	Transformer Models	225
14.6.1	Importing Libraries and Gathering Data	225
14.6.2	Embedding the Documents	226
14.6.3	Flattening the Data	227
14.6.4	Isolating Clusters with HDBScan	227
14.6.5	Analyzing the Labels	227
14.6.6	Outliers (Noise)	229
14.7	Top2Vec in Python	230
14.7.1	Creating a Top2Vec Model	231
14.7.2	Analyzing Our Topic Model	232
14.7.3	Working with Bigrams and Trigrams	235
14.7.4	Saving and Loading a Top2Vec Model	236
15	Text Analysis with BookNLP	239
15.1	Introduction to BookNLP	239
15.1.1	What Is BookNLP?	239
15.1.2	Why Books and Larger Documents?	240
15.1.3	How to Install BookNLP	241
15.2	Getting Started With BookNLP	241
15.2.1	Importing BookNLP and Creating a Pipeline	241
15.2.2	Setting up the File and Directories	242
15.2.3	Running the Pipeline	243
15.3	The Output Files	243
15.3.1	The .tokens File	244
15.3.2	The .entities File	245
15.3.3	The .quotes File	247
15.3.4	The .supersense File	248
15.3.5	The .book File	248
15.3.6	The .book.html File	251

15.4	Character Analysis	252
15.4.1	Analyzing the Characters (From BookNLP Repo)	252
15.4.2	Parsing Verb Usage	254
15.5	Events Analysis	256
15.5.1	Exploring the Tokens File	256
15.5.2	Grabbing the Events	257
15.5.3	Analyzing Events Words and Lemmas	258
15.5.4	Grabbing Event Sentences	259
15.5.5	Bringing Everything Together	260
15.5.6	Creating an <code>.events</code> File	261
15.5.7	Conclusion	261
16	Social Network Analysis	263
16.1	The Basic Concepts of Social Network Analysis	263
16.1.1	Basic Terminology	263
16.1.2	SNA Libraries in Python	264
16.2	Introduction to NetworkX	264
16.2.1	Adding All Edges at Once	265
16.2.2	Asymmetrical Networks	266
16.2.3	Calculating Shortest Distance	267
16.2.4	Calculating Connections	268
16.2.5	Identifying Major Actors in a Network	268
16.2.6	Limitations of Matplotlib and NetworkX	268
16.3	Producing Dynamic Graphs with PyVis	269
16.3.1	The Basics of PyVis	269
16.4	NetworkX and PyVis	270
16.5	Adding Color to Nodes	271
16.6	SNA on Humanities Data: Structuring the Data	273
16.6.1	Examining the Data	273
16.7	SNA on Humanities Data: Creating the Graph	276
16.7.1	Building the Network	277
16.7.2	Visualizing the Network	277
16.7.3	Adding Menus	279
16.7.4	Conclusion	282
IV	Designing an Application with Streamlit	283
17	Introduction to Streamlit	285
17.1	Creating Our First App	285
17.1.1	Options for Application Development in Python	285
17.1.2	Installing Streamlit	286
17.1.3	Creating a Home Page	286
17.2	Displaying Data in Streamlit	287
17.2.1	Displaying Text to Users	287
17.2.2	Displaying Python Data Structures	289
17.2.2.1	Data Structures with <code>st.write()</code>	289
17.2.2.2	Data Structures with <code>st.json()</code>	289
17.2.3	Displaying Tabular Data	289
17.2.3.1	Tabular Data with <code>st.write()</code>	290
17.2.3.2	Tabular Data with <code>st.dataframe()</code>	290
17.2.3.3	Tabular Data with <code>st.table()</code>	291

17.2.3.4	Tabular Data with <code>st.markdown()</code>	291
17.2.4	Displaying Multimedia in Streamlit	291
17.2.4.1	Images	292
17.2.4.2	Audio	292
17.2.4.3	Video	292
17.3	Streamlit Input Widgets	292
17.3.1	Text Input Widgets	293
17.3.1.1	<code>st.text_input()</code>	293
17.3.1.2	<code>st.text_area()</code>	293
17.3.2	Numerical Input Widgets	294
17.3.2.1	<code>st.number_input()</code>	294
17.3.2.2	<code>st.slider()</code>	294
17.3.3	Date and Time Input Widgets	295
17.3.3.1	<code>st.date_input()</code>	295
17.3.3.2	<code>st.time_input()</code>	296
17.3.4	Boolean Input Widgets	296
17.3.4.1	<code>st.checkbox()</code>	297
17.3.4.2	<code>st.button()</code>	297
17.3.5	Selection Widgets	298
17.3.5.1	<code>st.radio()</code>	298
17.3.5.2	<code>st.selectbox()</code>	298
17.3.5.3	<code>st.multiselect()</code>	299
18	Advanced Streamlit Features	301
18.1	Data Visualization	301
18.1.1	Metrics	301
18.1.2	Plotting Basic Graphs with Streamlit	302
18.1.2.1	Line Charts with <code>st.line_chart()</code>	303
18.1.2.2	Bar Charts with <code>st.bar_chart()</code>	303
18.1.2.3	Area Charts with <code>st.area_chart()</code>	303
18.1.3	Map Charts	304
18.1.3.1	Creating Maps with <code>st.map()</code>	305
18.1.3.2	Third-Party Maps – An Example with PyDeck	305
18.2	Layout Design	307
18.2.1	Layout Widgets	307
18.2.1.1	Sidebar	307
18.2.1.2	Columns	307
18.2.1.3	Expander	307
18.2.1.4	Container	309
18.2.1.5	Tabs	309
18.2.1.6	Empty	310
18.3	Streamlit Cache and Session States	310
18.3.1	Caching Data with <code>@st.cache_data</code>	311
18.4	Storing Data with <code>st.session_state</code>	311
18.5	Custom HTML	312
18.6	Multi-Page Applications	313
19	Building a Database Query Application	315
19.1	Building a Database Query Application	315
19.1.1	Importing the Libraries	317

19.1.2	Caching Data	317
19.1.3	Creating Our App Layout	318
19.1.4	Using User Inputs to Produce a New DataFrame	319
19.2	Deploying an App in the Cloud with Streamlit Share	321
19.2.1	Create a GitHub Account	321
19.2.2	Upload Application to GitHub	321
19.2.3	Connect Streamlit Share to your GitHub	322
19.2.4	Create a New App	322
19.2.5	Set Custom Subdomain	322
V	Conclusion	327
20	Conclusion	329
Index		331

Preface

I designed this textbook to serve two functions. First, it will function as a primer to Python for humanists (or, more generally, those without coding experience or a background in computer science). In this regard, readers will acquire a basic understanding of necessary background information, such as data and data structures, as well as the basics of Python.

Second, this textbook is designed to not only provide the reader with a basic understanding of Python and how to use it, but how to apply it specifically to humanities-based problems. The book particularly explores applying Python in data analysis with Pandas, natural language processing (NLP) with spaCy, topic modeling with Gensim (for LDA topic modeling) and Top2Vec (for modern topic modeling), and social network analysis with NetworkX, Matplotlib, and PyVis. It also teaches readers how to design quick applications with Streamlit and deploy them in the cloud with Streamlit Share.

All code throughout this textbook is designed to be as reproducible as possible. The goal is for the reader to only need to replace the default data with their own data (or with minimal effort get their data into a structured format) and have similar results. This should allow the reader to begin applying the methods discussed throughout this textbook with relative ease to their own areas of expertise.

Part I of the textbook introduces the reader to the basics of Python. Here, we will learn about the basics of coding (Chapter 1) and the essentials about data and data structures and how to work with them via Python (Chapter 2). These chapters will provide the necessary foundation for exploring key programming basics, such as loops and conditionals (Chapter 3), functions, classes, and libraries (Chapter 4), and working with external data, such as text files and JSON (Chapter 5). The final chapter of Part I will introduce the reader to the basics of web scraping and working with data found on the web.

After Part I, the reader will have a basic understanding of Python, its syntax, and be able to begin working with data to design projects. The remainder of the textbook is designed to reinforce all of the skills acquired in Part I. Each of the following parts of the textbook will also introduce the reader to the key libraries associated with their respective subjects.

In Part II, we will take a deep dive into data analysis. In Python, the essential library for working with data, specifically tabular data, is Pandas. We will learn the basics of Pandas while working with the open-source Titanic dataset. By the end of Part II, the reader will have an understanding of Pandas and be able to leverage it in their own projects.

Part III shifts focus to text analysis. Here, we will learn about natural language processing (NLP) and how to use Python and the library spaCy to engage in NLP. This part of the textbook presumes no knowledge on the part of the reader about NLP or linguistics. It will, therefore, provide all the basic information needed to begin working with texts in more robust ways. The reader will learn about two different approaches to NLP, specifically rules-based (heuristics) and machine learning-based. Both serve different functions and should be used in different situations. By the end of this part, the reader will have a basic understanding of each and know when to use them. While the machine learning-based approaches will be rooted in using off-the-shelf spaCy models, the reader will learn how to use NLP rules to create custom solutions. The final chapter of this part of the textbook

will look at a real-world problem, creating a rules-based heuristic pipeline to identify and extract specific types of entities from texts.

Part IV shifts to other applications of Python to humanities-based problems. In Chapter 14, we will learn how to do topic modeling, specifically Latent Dirichlet Allocation (LDA) topic modeling so that they will have an understanding of the basic concepts and the history of the field. After this, we will learn about more recent approaches to topic modeling using machine learning with the library Top2Vec. Chapter 15 will look at performing text analysis on larger documents with BookNLP. In Chapter 16, we will look at Social Network Analysis with NetworkX and Matplotlib to produce static maps. We will also learn how to create dynamic JavaScript and HTML network maps with the Python library PyVis. These chapters are all designed to give you the essential background knowledge, terminology, and Python code to get started applying these libraries and methods on your own dataset.

Part V of the textbook will introduce the reader to app development with the Streamlit library. Readers will gain an understanding of the basics of Streamlit and how to leverage its components to create custom apps within just a few hours that can be hosted in the cloud. The purpose of this part is to help the reader take an idea from concept to reality in as short of time as possible.

After completing this textbook, you will have a strong enough command of Python to begin leveraging it in your own projects. You will also have a broad exposure to different ways that Python can be applied to humanities-based problems. Finally, you will have the resources necessary for continuing your education.

Limitations of This Textbook

While this book will provide a cursory overview of Python, it will not provide you with all aspects of the language or how to use it. This book is designed to get you up and running with Python as quickly as possible, giving you the essential tools you need to read and write in the language to solve tasks quickly and effectively. This textbook is not designed for computer scientists who wish to explore the depths of the programming language, rather humanists who need Python to automate certain tasks in their workflow. Explanation of certain aspects of the language are, therefore, kept to a minimum.

It is important to note that this book is entirely designed in Jupyter Notebooks (discussed in Part I, Chapter 1). This means that you will not receive exposure to the command line or receive proper training in writing a Python script (.py) file. These are useful skills to have, but not necessary to begin working with data. Despite these limitations of the textbook, this book will give you the tools necessary to begin learning on your own.

Online Version

The print version of this textbook also has a free online component compiled as a Jupyter-Book. As the libraries and methods discussed in this textbook advance, the print version of this book will not be easily updated without new editions; the online version, however, will be updated and maintained. If a section of code quits working because something has changed with Python or one of the libraries used in this textbook, the online version will be

corrected. If you notice that there is a problem with code, you can also formally submit an issue or suggest an edit on GitHub so that it can be updated. These can be minor issues from typographical errors, the need for greater explanation in a specific area, or problems with code not working. To submit a GitHub issue, you can use the GitHub icon in the top-right corner of the online version of this book.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

Acknowledgments

This Python Textbook was created during my postdoctoral fellowship at the Smithsonian Institution's Data Science Lab with collaboration at the United States Holocaust Memorial Museum. It would not have been possible without the help of Rebecca Dikow, Mike Trizna, and those in the Data Science Lab who listened to, aided, and advised me while creating these notebooks. I would also like to thank the content experts at the United States Holocaust Memorial Museum (USHMM), specifically Michael Haley Goldman, Michael Levy, Robert Ehrenreich, and Silvina Fernandez-Duque.

I would also like to dedicate this book to my wife, Stephanie Mattingly. She has been a constant source of encouragement while I wrote this book in all its stages. She also always encouraged me to pursue data science as a career path. Without her constant guidance, this book would not have been possible.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

About the Author

William Mattingly is a 2022 Harry Frank Guggenheim Distinguished Scholar and a 2022-2023 ACLS Grantee for his work as co-principal investigator and lead developer for the *Bitter Aloe Project* which examines testimonies of violence from South Africa's Truth and Reconciliation Commission. He is currently the Postdoctoral Fellow for the Analysis of Historical Documents at the Smithsonian Institution's Data Science Lab. Mattingly currently works on two projects at the Smithsonian. The first is based at the United States Holocaust Memorial Museum (USHMM), where he is developing a robust pipeline of machine learning image classification and natural language processing (NLP) models to automate the cataloging of millions of images. At the Smithsonian, he is working on a project connected to the American Women's History Initiative. Here, he is developing machine learning and heuristic pipelines with spaCy, a Python NLP library. This pipeline will identify women in Smithsonian documents and automatically extract knowledge about them so that we can better understand the influential role women played at the Smithsonian.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

Part I

The Basics of Python



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

1

Introduction to Python

1.1 Introduction to Python

1.1.1 Why Should Humanists Learn to Code?

Before we begin this book, let's begin with a simple question. *Why should humanists learn to code?* To answer it, let's consider the ideal digital humanities project.

In an ideal digital humanities project, a humanist (or team of humanists) will leverage their **domain knowledge**, or area of expertise. As a domain expert, this project leader, better known as principal investigator (PI), will convey their idea to their team which will consist of others in their field and usually a technical lead. This technical lead will (depending on funding) either handle all technical aspects of the project (web development, coding, data management, etc.) or lead a team to handle all technical aspects of it. This is the ideal scenario. Each person on the team does what they do best.

Such a project requires two things that are in high commodity to most researchers: funding and time. The self-reliant humanist who can leverage their domain knowledge while also being able to function in a technical capacity on a project drastically reduces both of these issues in several ways. To understand how, we must first understand how digital humanities projects receive funding.

1.1.1.1 The Timeline for Starting a Large Digital History Project

Large digital projects often take years to begin. These projects often start with a researcher (or researchers) wishing to explore a question or create a digital platform for an aspect of their research. These scholars will then reach out to others in their field to gauge interest. This process can take several months to a year, especially if the researchers wait to present their idea at the next conference in their field.

If there is an interest in such a project, the project PI will find a technical expert who can tell them if the project is possible and receive an estimate for its cost. If the PI does not have a technical expert and cannot find one associated with their university, this can be challenging and time-consuming. As we will see, the relationship between PI and technical lead is a unique bond that requires good communication from both parties. Selecting the right technical lead is important because their pay may be one of the larger budget items. It is also the person who makes the PI's vision a digital reality.

Next, the PI will create a list of funding opportunities for the project and then begin applying to them. Each application can take several months to complete. Once complete, the PI will then usually submit the application to the granting association through their university. If there are multiple PIs on the project at different universities, this process can be more complicated as all institutions usually need to communicate with each other.

The university will usually review all materials. They will be especially interested in the budget. This process can take a few weeks while the university ensures all numbers are

accurate. After all, most universities will be taking a cut of the grant money (sometimes up to 30%), so they want to make sure they see the numbers first.

After this, the university will submit the grant. Depending on the granting agency, the wait time can be anywhere from a few months to nearly a year. During this time, the digital project will may not move forward much, if at all. Rather, it may sit in a holding period while the PI waits to hear back.

When we look at this timeline, it means that from initial idea to the start of the project, even in the ideal scenario, it can be realistically 2–3 years before the project begins. All of this comes with a bit of a gamble as well. The PI believes in the idea, they have also gauged the community to know that there is interest in it, and they have found a technical lead who can do the job, but a lot can happen in 2–3 years. In that time, new developments in the domain expert’s field may have altered the project trajectory (or, worst-case-scenario, made it redundant). Further, new digital approaches/methods may mean that an entirely new approach is necessary. This is especially common today if the project involves machine learning, a field in which new advances are happening every week. In addition to this, the technical lead may have taken a job elsewhere. (This does happen as they are not contracted until the grant is received). Finally, there is no guarantee that funding agencies value the idea. This means that the time spent designing the project conceptually and preparing to get started could be entirely in vain.

By learning to code, you can often negate this entire process by testing your idea within days or weeks. This means that you will be able to test the validity of the idea and potentially even create a **minimum viable product (MVP)** that can be used to assist in obtaining funding. An MVP is a working subset of a project to demonstrate its validity. For a digital humanities project, this may mean a simple front-end application or a subset of the data prepared a specific way. MVPs are quite common today in places such as Silicon Valley, where MVPs are used to present an idea to investors. This helps give a real sense of how the project will work. While not all mechanics will be available in an MVP, it is a good proof-of-concept. It also demonstrates that the project is possible, the team can do the work they are promising, and that the team already has a good handle of the potential issues that may surface. These are all things that granting agencies like to see. Throughout this textbook, you will develop the skills necessary to create an MVP that you can present to a granting agency. This is especially true with the final chapter, where we learn how to build cloud-based Python applications with Streamlit.

1.1.1.2 The Self-Reliant Digital Humanist

Ultimately, the ability to code turns the humanist into an entirely self-reliant researcher. By learning to code, you can do far more than test out ideas and create MVPs. You can significantly reduce the cost of a digital humanities project. As a domain expert, you know your field well. As a coder, you will know what is technically possible. If you end up gaining an expertise in data science or machine learning (something that is quite realistic today as both fields are more accessible than ever), you can leverage some of the most powerful methods in technical fields within a digital humanities context. You can also eliminate many issues that may surface during a digital humanities project.

One problem is communication. The project PI needs the ability to communicate their ideas to the technical lead. This communication often occurs in a hybrid language that changes from team to team. The PI over time can begin to use technical terms and the technical lead will begin to use some domain-specific terms. This relationship and communication takes time to create. By have a strong command of the technical aspects of a project, the PI can either eliminate the need for a technical lead entirely or communicate their ideas to the technical side of the project in technical language.

Another issue that vanishes is the lag time from idea to execution. Once a funded project begins, the PI will guide their vision and convey each step to the technical lead whose job it is to make that vision a reality. There is a lag that exists here. Often, meetings on digital projects occur on a regular basis, sometimes weekly or bi-weekly. During that time, issues may surface that are placed on hold until the next meeting. This can put a digital project behind schedule. If the humanist is self-reliant, however, they can develop their idea independently of a technical lead. This means that when an issue surfaces, the PI can immediately rectify it and communicate the solution to the technical lead.

Third, technical leads on projects usually absorb a lot of funding. Hiring a programmer can be costly, but if your project requires data science and machine learning, experts in these areas can costs hundreds of dollars per hour. If the PI has the technical expertise, they can perform technical aspects of a project until enough money is received to hire a technical lead.

1.1.1.3 Other Benefits

Learning to code gives the humanist tools beyond simply the ability to code. It fundamentally alters how the humanist views their field and the questions they can explore. Things that seemed impossible before, will suddenly appear quite simple to solve.

One of the greatest advantages of learning to code is **automation**, or the process by which we write rules for a computer to perform a repetitive task. As humanists, we do a lot of things repetitively. Many of the tasks we need to perform as humanists are repetitive and we, as humans, are prone to make mistakes. Being able to automate these tasks can radically reduce the amount of time we spend performing repetitive tasks, from hours, weeks, or even years, to seconds, minutes, or hours. Imagine having to grab data from an archive website. Imagine that the information you need is found on 2,000 different pages. How long would it take you to go to each of those pages and copy and paste the text into a Word document? In Python, that task can be coded in minutes and left to run for an hour. This allows you, the researcher to go off and do some other task more essential; or, perhaps, enjoy a nice tea break in a hammock so that when you return to analyze the documents, you will be well-rested.

Coding does not just allow us to automate tasks like this. It also allows us to systematically clean data. Imagine you wanted to search across PDF scans of medieval Latin. This problem presents many key issues that make such a task impossible or unreliable. First, medieval Latin did not have any firm spelling convention. This means that some scans may have variant spellings of words. Second, Latin is a highly inflected language, meaning word order is not important, rather the ending of a given word is. When combined with variant spellings, this means that each word can be represented sometimes hundreds of different ways. In addition to this, your texts are scans. Are those scans even searchable? If they are, was the OCR, or Optical Character Recognition, accurate? If it was done prior to 2015, it likely is not. If after, then the scans may be in a bad enough state where the OCR is not accurate. In addition to these problems, any OCR system will retain line breaks, meaning if the key word that you want to search for is hyphenated because the word is broken up between two lines, you need to account for that in your search. Next, we need to take into account editor notations, which often are noted in brackets, parentheses, and carrots. While this example is certainly a complex one, it is perfectly common. And while I am using Latin to demonstrate a greater issue with inflected languages, these same issues, especially those around OCR, surface with English texts as well. Coding allows us to address each and every one of these issues, some more easily than others.

The issue of searching is further complicated when this needs to be done for many documents simultaneously. Imagine if these issues surfaced in 5,000 different PDFs that you needed to analyze. Could you realistically run all these searches across 5,000 documents?

If you could, would your results be good? To answer the former, yes, if you are willing to spend months doing it; to answer the latter, likely not. Programming allows for you to develop programs that perform all these tasks across all 5,000 documents. When you run a search, you will not simply hit `ctrl+f`. You will code your own search method so that your simple search can return very complex results that accounts for variance in the text.

Python makes all of this possible.

1.1.2 What Is Python?

Python is a programming language. Programming languages are ways that we, as humans, can write commands that will then be executed by a computer. There are many different programming languages available for humanists to choose from:

1. C
2. Python
3. JavaScript
4. R
5. HTML (this is debatable)

Not all programming languages are created equal. Some are best used for developing software, such as C; others are best suited for web development, such as HTML and JavaScript. And others are great at statistical analysis, R (and Python). Where does Python excel? Python excels in many areas. One thing that sets it apart from other programming languages for new coders is that it is easy to read and easy to write.

It is relatively easy to write compared to other programming languages because the syntax, or way in which you write tasks in code, is straightforward. It is easy to read because Python uses forced indentation. This means that blocks of code that can be difficult to read in other programming languages are easily spotted in Python.

Since its creation in the early 1990s, Python has soared in popularity which has, in turn, resulted in a large community of programmers and a large number of libraries available. We will learn about libraries later in this textbook. For now, think of libraries as large quantities of code that have already been written for you so that you can write one line of code to perform a complex task that may take hundreds or thousands of lines of code to write.

Today, Python is one of the most widely used programming languages and is considered the essential language for text analysis, machine learning, data analysis (alongside R), web scrapping, and much more.

Python (as of this writing) is currently in version 3.11.0. Let's break each of these numbers down. The 3 refers to the main Python language. Python 2 still exists (in fact it comes standard on all Macs), but it is no longer supported and is slowly being replaced by Python 3. You should not invest time in learning Python 2 as it is only used to support legacy software and has a high number of security issues due to its depreciated status. This is important to note because certain things are coded differently between the two programming languages. If you were to look for help on a coding forum, such as StackOverflow, it is important to know about these distinctions.

The 11.0 in Python 3.11.0, tells us specifically what version of Python 3. Not all code from certain libraries is backwards compatible, meaning some libraries require a specific version of Python, so understanding this now as a concept will save confusion later. Each time a new version comes out, new features are available, so it is important to stay up-to-date with current versions, but it is not always essential.

1.1.3 Why Python?

For all the above reasons, I encourage humanists to learn Python as their first programming language. It is one of the easiest languages to learn, straightforward to write, and can solve most of the programmatic problems a humanist may face. The large quantities of libraries and tutorials mean that there are few tasks that will prove impossible to do.

If you are sold on Python, then continue reading this textbook as we learn how to install it and use it as humanists.

1.2 Installing Python

The most challenging thing about working with Python (or any programming language) is learning how to install it correctly. Why is this such a challenge? Because installing Python differs dramatically by operating system and version of operating system. I present to you five different ways you can start programming Python from the easiest to the more challenging.

1. Using the Trinket applications embedded throughout the digital version of the textbook
2. Using Google Colab for each individual section of the textbook
3. Using the free built-in Binder feature of this textbook
4. Using a free online Jupyter Notebook
5. Installing Python locally via Anaconda Navigator and using JupyterLab

If you are entirely new to programming, I recommend ignoring options 2–4 and simply using option 1 during the first part of this textbook. In my experience, it is important early in your programming journey to have as few roadblocks as possible. If you are interested in jumping in with Python as quickly as possible, then use the freely available Trinket widgets in the online version of this textbook and return to the installation of Python at a later time.

1.2.1 Trinket

Throughout the digital version of the textbook, you will see Trinket applications embedded within each page. Trinket allows you to practice your coding skills without ever installing Python. If you are new to coding and want to get started immediately without installing Python, I recommend working with Trinket inside the digital version of this textbook.

These will allow for you to practice the lesson right inside the digital textbook.

You will be able to write your code inside this Trinket application and then click the play button at the top. This will execute your code and populate the results on the right hand output.

1.2.2 Using Google Colab

During the first part of the textbook, nearly all code can be executed with Trinket. Even the section on Pandas can be done inside of Trinket. However, the later sections will be

increasingly difficult as they require libraries that are not supported by Trinket. At this time, you will want to consider another option. Again, to delay the installation of Pyonn on your machine, we can rely on other cloud-based alternatives.

There are two options for running this textbook's code in the cloud outside of the textbook: Google Colab and Binder (addressed below). In the top right corner of each page is a rocket ship. If you hold your mouse over this rocket ship, you will have two options to choose from, Binder or Colab. Click Colab.



2.2. Introduction to Data Structures

2.2.1. Data Structures

In the last section, we met strings, integers, floats, and booleans. Each of these were types of data. Strings, for example, allowed us to work with text and numbers allowed us to work with integers and floats. In this chapter, we will begin working with data structures. **Data structures** are ways of storing

FIGURE 1.1

Picture of Rocket Ship in Top-Right Corner of Online JupyterBook.

Once you click the Colab button you will be taken to Google Colab and see a Notebook inside the Google Colab system that should look like this:

```

01_02-03_data_structures.ipynb
File Edit View Insert Runtime Tools Help
+ Code + Text Copy to Drive
Introduction to Data Structures
Data Structures

In the last section, we met strings, integers, floats, and booleans. Each of these were types of data. Strings, for example, allowed us to work with text and numbers allowed us to work with integers and floats. In this chapter, we will begin working with data structures. Data structures are ways of storing multiple kinds of data in a systematic way. In Python, these are created as objects that can be stored in memory and called later in a script. They are divided into two categories: mutable and immutable. We encountered these terms in the last chapter, but we will explore what they mean in more depth below.

Throughout this section, we will learn about some of the key types of data structures, how they are different, and how they can be used. We will only cover these in a cursory manner. Throughout this textbook, we will use these data structures as we write code and perform data cleaning and data analysis tasks. To keep things simple for now, we will focus on four types of data structures: lists, tuples, sets and dictionaries. There are other types of data structures in Python, but these are the core four that you will use most frequently.

Lists

The first data structure we will work with is known as a list. Lists are precisely what they sound like, a list of data. As we will see below, there are multiple ways of storing information in a list-like manner in Python, such as with tuples and sets, but the way we create lists and the way we interact with lists is distinct.

Lists and tuples are identical with one major exception: lists are mutable. This means that you can create a list object and then alter it in memory. This allows for you to do very powerful things to lists that you cannot do to tuples. And these are going to be one of the key data structures you use in all digital humanities projects. The reason? We often need to adjust data while working with it.

As with data, we can create a list object in memory by creating a variable followed by an equal sign. To tell Python that the specific type of object we are creating is a list, we use an open and a close bracket. Each item in the list will be separated by a comma. Lists can store any type of data. To see this in action, let's create our first list.

[ ] first_list = [1, 1.0, "one"]
print(first_list)

[1, 1.0, 'one']

Indexing a List

```

FIGURE 1.2

Demo of Google Colab Opened.

Once here, you can run the notebook.

Colab and Binder have their advantages. Google Colab will be far quicker but will require you to install the requisite libraries for each page. For most of the first part of the textbook, this will not be necessary as we will not be working with libraries. You will learn how to install libraries locally and in a Jupyter Notebook (such as those used on Google Colab) later in this part of the textbook. Once the libraries are installed, the entire notebook can be run as if you were running the notebook locally. A downside to Google Colab is its inability to leverage the data found within this textbook. This means that in some cases, you will need to manually upload the data into Google Colab for some notebooks to run as expected.

Another benefit of Google Colab is that it is connected to your Google account. This means any changes that you make to the notebook can be saved onto your Google Drive and used at a later date. Each time you open the notebook, however, a unique instance of Python is created which means you will need to reinstall your libraries. Fortunately, this process is quite quick.

1.2.3 Using Binder from JupyterBook

Another cloud-based solution is with Binder. Unlike Google Colab, Binder will load up an environment for the entire textbook and install all libraries using the `requirements.txt` file found in this textbook's repository on GitHub. This means that Binder will take substantially longer to load, but will contain the entire textbook. It will save the instance for a period of time as well so that if you return to the same link in a short period of time, there is a chance it will not have to reinstall all the libraries again. In my experience, Binder can run into some issues that can be difficult to diagnose, especially if you are new to programming. For this reason, I would recommend Google Colab over Binder to new programmers.

As with Google Colab, to begin, put your mouse over the rocket ship icon in the top-right portion of the page.



2.

Installing Python

Dr. W.J.B. Mattingly

Smithsonian Data Science Lab and United States Holocaust Memorial Museum

January 2022

2.1. Covered in this Chapter

FIGURE 1.3

Demonstration of Rocket Ship in Top-Right Corner.

A button should emerge that says “Binder”. Click it.



FIGURE 1.4

Selecting the Binder Button.

At this stage, your screen should look like this:

If it does, then sit back and relax. It may take a few minutes as Binder builds the environment. Once Binder is finished, click on “File” in the top left corner of the screen and then select “New Notebook”.

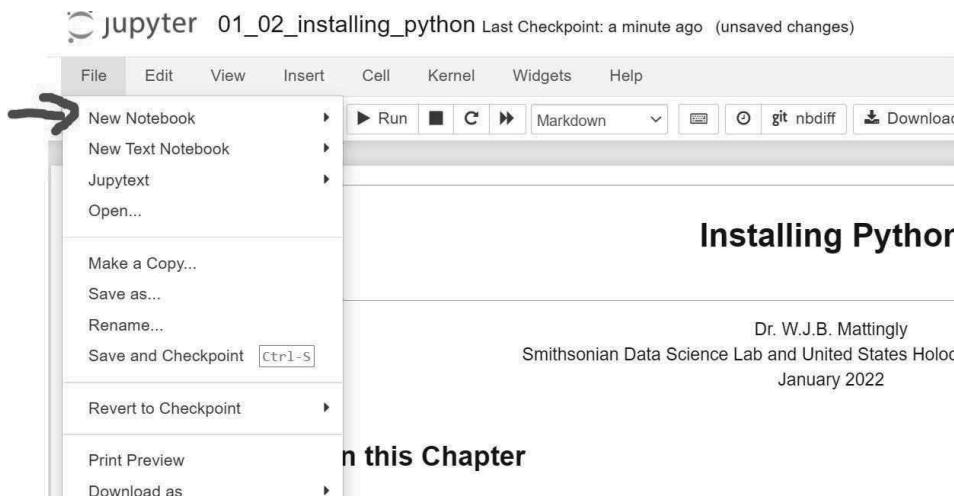


FIGURE 1.5

Creating a New Notebook in Binder.

After this, select “Python 3 (ipykernel)”. This will open a fresh, new notebook.

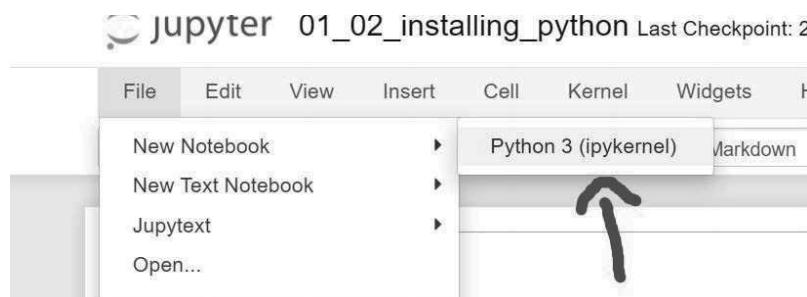


FIGURE 1.6

Selecting the Python 3 Kernal for the New Notebook.

Now you should have a new notebook that looks like this:

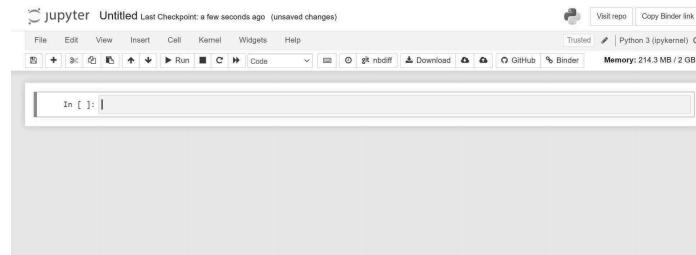


FIGURE 1.7
Example of a New Notebook Page.

And now, you can fully follow along with this textbook.

1.2.4 Using Jupyter Notebooks Online

If you want to be a bit more independent and learn how to code in Python online without Binder and this textbook, you can! You do not need to download and install Python on your local computer either. Instead, you can use free online compilers that allow you follow along with nearly all of this textbook without issue.

If this is your view at the moment, then click this link: <https://jupyter.org/try>
Once on the page click the picture that says Jupyter Notebook.

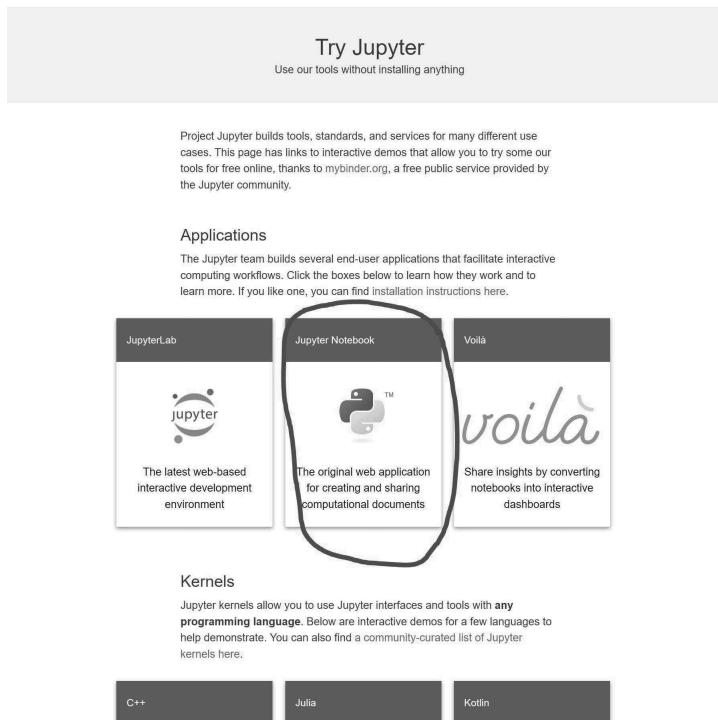


FIGURE 1.8
Creating a Jupyter Notebook Online.

After that, you will see a screen that looks like this:

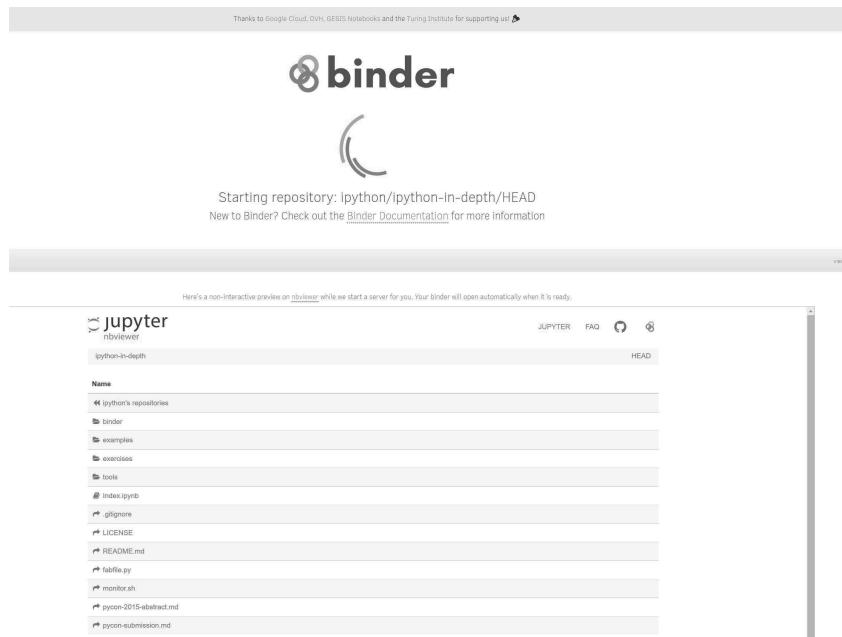


FIGURE 1.9

Binder Creating a New Environment for the Notebook.

After a few minutes, the environment will load and you should see a notebook. Click “File” in the top left corner, New Notebook, and select Python (Pyodide).

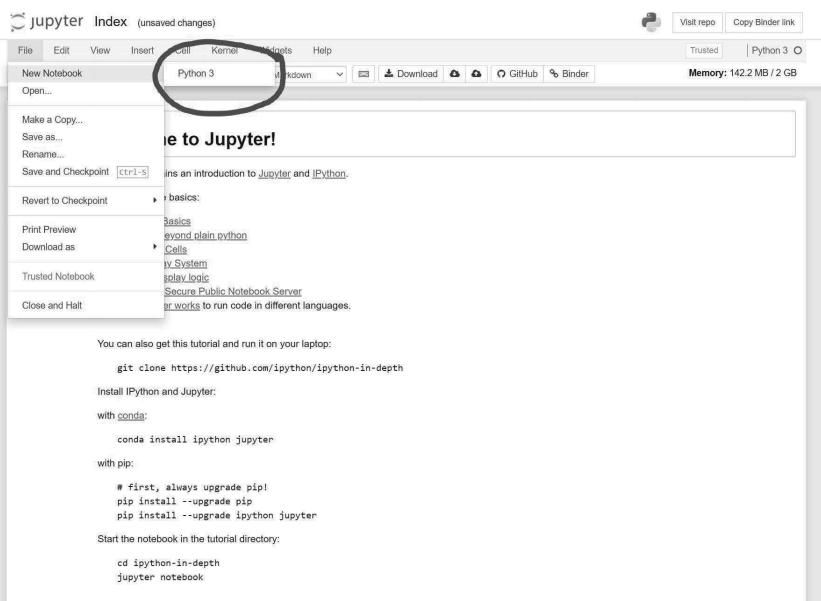


FIGURE 1.10

Creating a New Notebook.

Once loaded, you can follow along with this textbook in the new notebook.

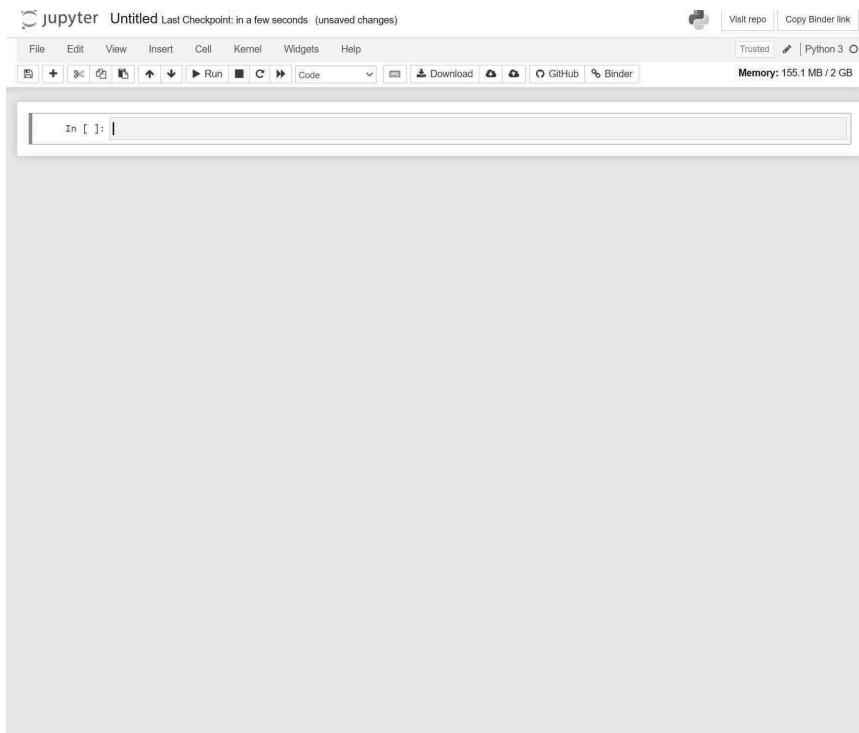


FIGURE 1.11
Example of a New Notebook.

1.2.5 Installing Python Locally

If you wish to install Python locally and it is your first time, there are many problems that can surface. For this reason, I am now recommending all students install via Anaconda Navigator. It adds extra steps into the installation process, but it eliminates the potential for mistakes to occur.

Anaconda Navigator is a user-friendly interface that handles the installation process for you. It also allows for you to create environments, which are small areas on your computer that have a unique version of Python and libraries installed. We will learn more about this in Part I, Chapter 4 when we explore libraries.

In this chapter, I will walk you through the steps of installing Anaconda Navigator on your machine, regardless of operating system.

When we write code in Python, we do so a few different ways, depending on that code's use. Because this is a textbook and the code I am writing is for presentation purposes, I am using a Jupyter Notebook. Other times, you may write a program in an IDE, or an Integrated Development Environment. Some of these include PyCharm, VS Code, etc. In other cases, you will use Python in the terminal to perform quick tasks on data in a directory.

Anaconda Navigator removes the need for you to learn how to do all of this because it allows for you to easily install JupyterLab which functions like an IDE but is a bit more forgiving. In addition, you can call terminal sessions. I know these terms do not make sense right now, but as your understanding of programming expands, this paragraph will make

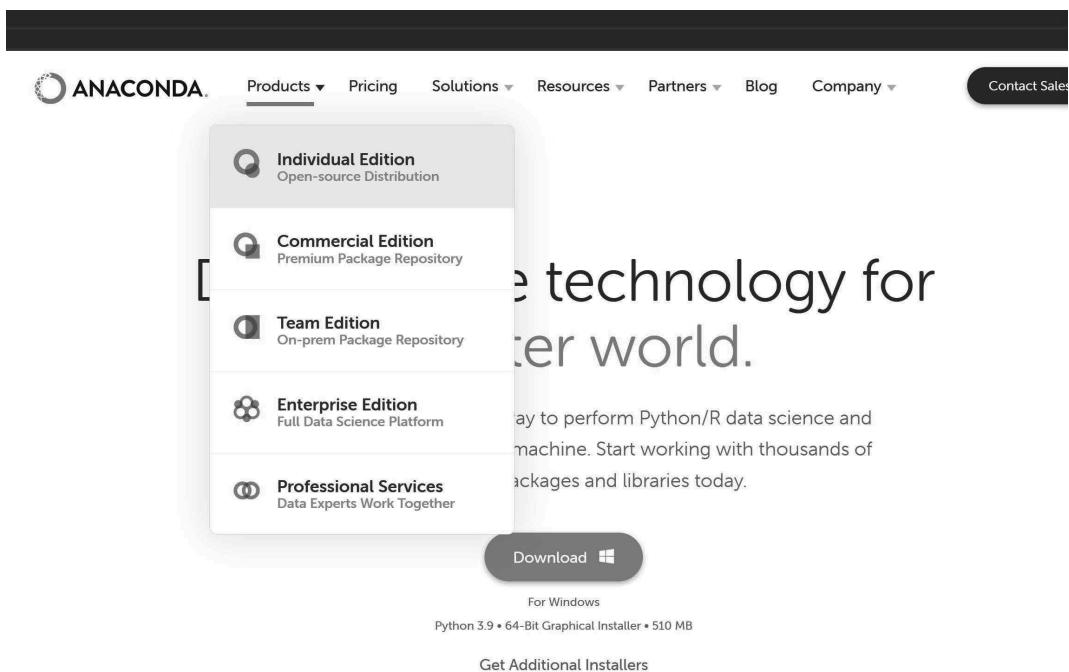


FIGURE 1.12
Anaconda’s Homepage.

more sense. For now, simply understand that Anaconda Navigator and JupyterLab (both of which we install in this chapter), make your start to learning to code much, much simpler.

1.2.5.1 *Download Anaconda Navigator*

Now, click on the “Download” button as seen in Figure 1.13 and walk through the installation process on your computer. Once complete, you will have Anaconda Navigator installed.

1.2.5.2 *Using Anaconda Navigator*

Now that Anaconda Navigator is installed, you can use it by opening it up on your computer. When you first see Anaconda Navigator, you will see several options on the left. In environments, you will only see one environment: base. Don’t worry about environments this early in your Python career. For now, we will simply click the “Home” button on the left.

1.2.5.3 *Installing JupyterLab*

In order to start using Python, there are a few different options available to you. You could use a terminal and follow along with the textbook, but that can be quite difficult if you are just starting out. Since this textbook was designed in JupyterLab, I think it makes the most sense to install that software. Go ahead and click “Install” under JupyterLab. Once installed, you will be able to “Launch” it and follow along with the textbook.

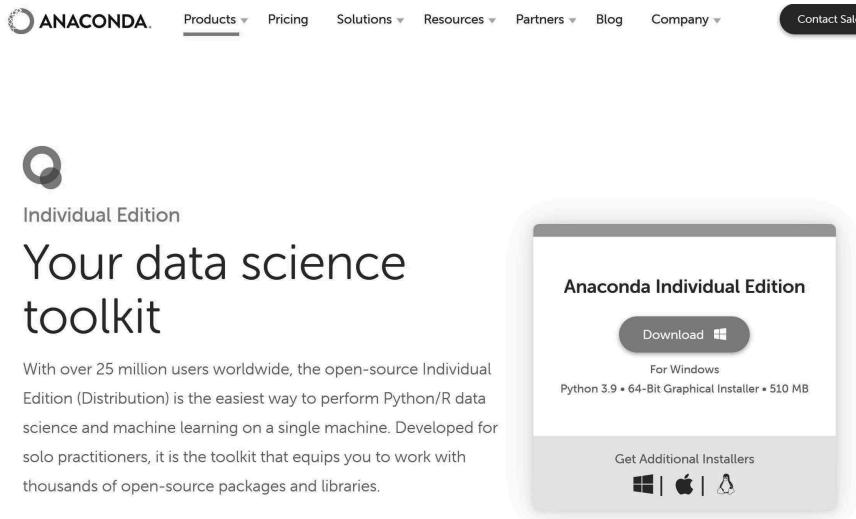


FIGURE 1.13
Download Individual Edition.

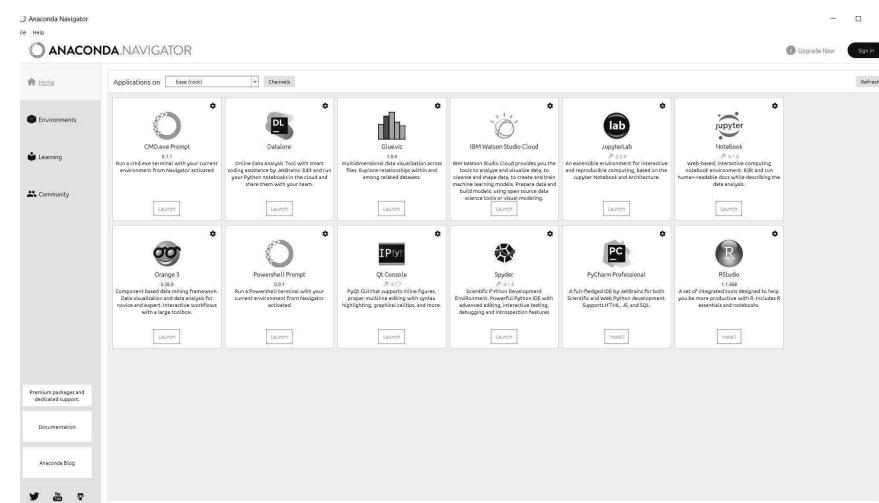
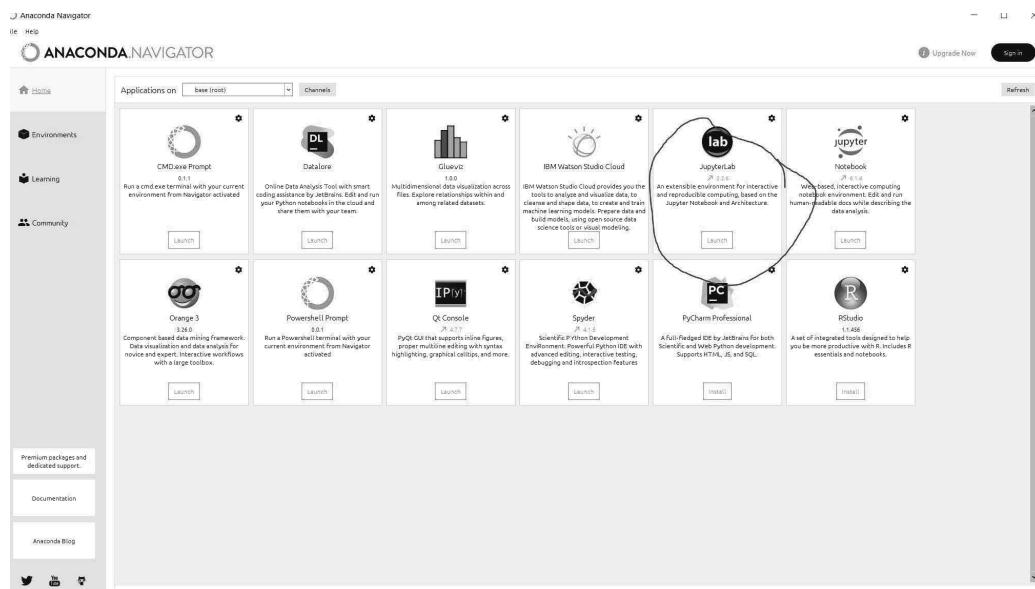


FIGURE 1.14
The Anaconda Navigator Main User Interface.

1.2.6 Conclusion

If at any point in the installation process you get frustrated, do not get discouraged. You are not alone and it is not a simple or easy process. Those who have been programming for years can make these steps look simple, but troubleshooting issues on your own is challenging.

**FIGURE 1.15**

How to Install Jupyter Lab.

It is important to remember that programming in general is challenging. Try to view the challenges that surface as a fun problems that need solving. Try to look at your mistakes through a lens of Bob Ross and view them as happy accidents.

1.3 Coding Basics

Before we jump into the textbook, we should cover a few basic aspects of programming that the reader may not know. These are some of the most essential terms and concepts that you will see used throughout this textbook. Remember, you can always come back to this section if you forget one of these terms.

1.3.1 The Print Function

The very first thing that every programmer learns is how to print something off using the relevant programming language. In most tutorials, you will see the phrase “Hello, World!” used. In this textbook, let’s try something a bit different. Let’s say I wanted to print off “Hello, William”. We can do this in Python by using the **print function**. The print function lets us print off some piece of data. In our case, that piece of data will be a piece of text, known as a string (see below). Our text is “Hello, William!”. The thing that we want to print off must be located between an open parentheses and a close parentheses. Let’s try to execute the print command in the cell below.

```
print("Hello, William!")
```

```
Hello, William!
```

As we can see, the code that we typed in the Jupyter cell outputted beneath it. Now, it is your turn. Try to print off something. In order to print off text, you will need to use an open and a close quotation mark. We will learn more about this as we meet strings in the next chapter.

```
<IPython.lib.display.IFrame at 0x7f0a6c7b3af0>
```

This has worked wonderfully, but what if our Python notebook needs to be more dynamic, meaning it needs to adjust based on some user input. Perhaps, we need to use a name other than William based on some user-defined input. To do this, we would need to store a piece of data in memory. We can do this in Python by creating a variable that will point to an object. Before we get into how this is done, let's first get to know objects and variables.

1.3.2 Objects

When we import or create data within Python, we are essentially creating an object in memory with a variable. These two words, object and variable mean slightly different things, but are often used interchangeably. We will not get into the complexities of their differences and why they exist in this textbook, but for now, view an object as something that is created by a Python script and stored in your computer's memory so that it can be used later in a program.

Think of your computer's memory rather like your own brain. Imagine if you needed to remember what the word for "hello" is in German. You may use your memory rather like a flashcard, where "hello" in English equates to "hallo" in German. In Python, we create objects in a similar way. The object would be the dictionary entry of "hello: hallo".

1.3.3 Variables

In order to reference this dictionary entry in memory, we need a way to reference it. We do this with a variable. The variable is simply the name of the item in our script that will point to that object in memory. Variables make it so that we can have an easy-to-remember name to reference, call, and change that object in memory.

Variables can be created by typing a unique word, followed by an "=" sign, followed by the specific data. As we will learn throughout this chapter, there are many types of data that are created differently. Let's create our first object before we begin. This will be a string, or a piece of text. (We will learn about these in more detail below.) In my case, I want to create the object author. I want author to be associated with my name in memory. In the cell, or block of code, below, let's do this.

```
author = "William Mattingly"
```

Excellent! We have created our first object. Now, it is time to use that object. Below, we will learn about ways we can manipulate strings, but for now, let's simply see if that object exists in memory. We can do this with the print function.

The print function will become your best friend in Python. It is, perhaps, the function I use most commonly. The reason for this is because the print function allows for you to easily debug, or identify problems and fix them, within your code. It allows us to print off objects that are stored in memory.

To use the print function, we type the word print followed by an open parentheses. After the open parentheses, we place the object or that or piece of data that we want to print. After

that, we close the function with the close parentheses. Let's try to print off our new object author to make sure it is in memory.

```
print(author)
```

```
William Mattingly
```

Notice that when I execute the cell above, I see an output that relates to the object we created above. What would happen if I tried to print off that object, but I used a capital letter, rather than a lowercase one at the beginning, so Author, rather than author?

1.3.4 Case Sensitivity

```
print(Author)
```

```
-----
NameError                                 Traceback (most recent call last)
Input In [3], in <cell line: 1>()
----> 1 print (Author)

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

The scary looking block of text above indicates that we have produced an error in Python. This mistake teaches us two things. First, Python is case sensitive. This means that if any object (or string) will need to be matched in not only letters, but also the case of those letters. Second, this mistake teaches us that we can only call objects that have been created and stored in memory.

Now that we have the variable pointing to a specific piece of data, we can make our print function above a bit more dynamic. Let's try and print off the same statement as before, but with the new full author name. I don't expect you to understand the specifics of the code below, rather simply understand that we will frequently need to store variables in memory so that we can use them later in our programs.

```
print(f"Hello, {author}!")
```

```
Hello, William Mattingly!
```

1.3.5 Reserved Words

When working with Python, there are a number of words known as **reserve words**. These are words that cannot be used as variable names. As of Python version 3.6, there are a total of 33 reserve words. It can sometimes be difficult to remember all of these reserve words, so Python has a nice built in function, "help". If we execute the following command, we will see an entire list.

```
help("keywords")
```

```
Here is a list of the Python keywords. Enter any keyword to get more help.
```

False	class	from	or
None	continue	global	pass
True	def	if	raise

(continues on next page)

(continued from previous page)

and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

These are words that you cannot use as a variable name.

1.3.6 Built-in Types

In addition to reserve words, there are also built-in types in Python. These are words that you can use (as we will see) to convert one type of data into another. There are 94 of these in total. Unlike reserve words, you *can* use a built-in type as a variable name. It is, however, strongly discouraged to do so because it will overwrite the intended use of these variable names in your script.

```
import builtins  
[getattr(builtins, d) for d in dir(builtins) if isinstance(getattr(builtins, d), type)]
```

```
[ArithError,  
AssertionError,  
AttributeError,  
BaseException,  
BlockingIOError,  
BrokenPipeError,  
BufferError,  
BytesWarning,  
ChildProcessError,  
ConnectionAbortedError,  
ConnectionError,  
ConnectionRefusedError,  
ConnectionResetError,  
DeprecationWarning,  
EOFError,  
OSError,  
Exception,  
FileExistsError,  
FileNotFoundException,  
FloatingPointError,  
FutureWarning,  
GeneratorExit,  
OSError,  
ImportError,  
ImportWarning,  
IndentationError,  
IndexError,  
InterruptedError,  
IsADirectoryError,  
KeyError,  
KeyboardInterrupt,  
LookupError,  
MemoryError,  
ModuleNotFoundError,  
NameError,
```

(continues on next page)

(continued from previous page)

```
NotADirectoryError,  
NotImplementedError,  
OSError,  
OverflowError,  
PendingDeprecationWarning,  
PermissionError,  
ProcessLookupError,  
RecursionError,  
ReferenceError,  
ResourceWarning,  
RuntimeError,  
RuntimeWarning,  
StopAsyncIteration,  
StopIteration,  
SyntaxError,  
SyntaxWarning,  
SystemError,  
SystemExit,  
TabError,  
TimeoutError,  
TypeError,  
UnboundLocalError,  
UnicodeDecodeError,  
UnicodeEncodeError,  
UnicodeError,  
UnicodeTranslateError,  
UnicodeWarning,  
UserWarning,  
ValueError,  
Warning,  
OSError,  
ZeroDivisionError,  
_frozen_importlib.BuiltinImporter,  
bool,  
bytearray,  
bytes,  
classmethod,  
complex,  
dict,  
enumerate,  
filter,  
float,  
frozenset,  
int,  
list,  
map,  
memoryview,  
object,  
property,  
range,  
reversed,  
set,  
slice,  
staticmethod,  
str,
```

(continues on next page)

(continued from previous page)

```
super,  
tuple,  
type,  
zip]
```

Of this long list, I recommend paying particular attention to the ones that you are more likely to write naturally: bool, dict, float, int, list, map, object, property, range, reversed, set, slice, str, super, tuple, type, and zip. You are more likely to use these as variable names by accident than, say, ZeroDivisionError; and this shorter list is a lot easier to memorize.

1.3.7 Type Function

It is frequently necessary to check to see what type of data a variable is. To identify this, you can use the built-in function **type** in Python. To use type, we use the command below with the data we want to analyze placed between the two parentheses.

```
type("this is a string...")
```

```
str
```

1.3.8 Bugs

Throughout your programming career, you will often read or hear about bugs. A bug is a problem in your code that either returns an error or an unintended result in the output. Tracing down bugs and fixing them is known as debugging. Aside from figuring out how to code solutions to problems, debugging can be one of the more time-consuming aspects of writing a program, especially if it is quite complex. Throughout this textbook, we will encounter common errors so that you can see them in this controlled space. We will also walk through what the error means and how to resolve it. That said, you are likely to create many other bugs as you try to apply the code in this textbook to your own data. That is expected. Always remember to read each line of your Python carefully and, if you have an error message, identify where the error is coming from and what it is.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

2

Data and Data Structures

In this chapter, we will be learning about data and data structures. As with any programming language, it is important to have a basic understanding in these two concepts as they are the building blocks for most things you will do in Python. Whenever you write a program or do something with code, you are essentially writing commands to load, save, interact with, and manipulate data in some way. Understanding the different types of data and how to structure and store them is, therefore, essential.

2.1 Introduction to Data

2.1.1 What Is Data?

In Python there are seven key pieces of data and data structures which we will be working with:

- Strings (text)
- Integers (whole numbers)
- Floats (decimal numbers)
- Booleans (True or False)
- lists
- tuples (lists that cannot be changed in memory)
- dictionaries (key : value)

In this chapter, we will explore each of these. While this section focuses on data: strings, integers, floats, and Booleans; the next section will focus on data structures: lists, tuples, and dictionaries.

Data are pieces of information (the singular is datum) i.e., integers, floats, and strings. Data structures are objects that make data relational, i.e., lists, tuples, and dictionaries. Start to train your brain to recognize the Python syntax for these pieces of data and data structures discussed below.

For your convenience, here is a cheatsheet for helping you refer back to these types of data.

Cheatsheet for Data Types and Data Structures in Python

Name	Type	Category	Example	Trick	Mutable
string	str	text	"William"	quotes	no
integer	int	number	1	no decimal	no
float	float	number	1.1	with decimal	no
Boolean	bool	boolean	True	True or False	no
list	list	sequence	[1, 1.1, "one"]	[]	yes
tuple	tuple	sequence	(1, 1.1, "one")	()	no
set	set	sequence	{1, 1.1, "one"}	{ } - unique	yes
dictionary	dict	mapping	{"name": "tom"}	{key:value}	yes

2.1.2 Strings

Strings are a sequence of characters. A good way to think about a string is as text. We can create a string in python by designating a unique variable name and setting = to something within quotation marks, i.e., "" or ''.

The opening of a quotation mark indicates to Python that a string has begun and the closing of the same style of a quotation mark indicates the close of a string. It is important to use the same style of quotation mark for a string, either a double or a single.

Let's take a look at a few examples of strings.

Examples of Strings

In our first example of a string, we will name our first string object with the variable name `first_string`. Notice that we are using an underscore (`_`) to indicate a separation of two words in our variable name. This is common practice in Python. We cannot use a space in a variable name. Another naming convention could be `firstString`, where the first word is lower cased while the start of each sequential word is capitalized. This naming convention, while acceptable, is more common in other programming languages, such as JavaScript.

```
first_string = "This is a string."
```

Now that we have created our first object, let's try and print it off. We can do this using the `print` function.

```
print(first_string)
```

```
This is a string.
```

Let's take a look at another example. This time, however, we will use two ' '.

```
second_string = 'This is a string too.'
print(second_string)
```

```
This is a string too.
```

Let's now take a look at a bad example of a string. In the example below, we will intentionally try to create an error message by combining two different types of quotation marks.

```
bad_string = "This is a bad example of a string'
```

```
Input In [6]
bad_string = "This is a bad example of a string'
^
SyntaxError: EOL while scanning string literal
```

This error message tells us that we have triggered a “`SyntaxError`”. A `SyntaxError` is a Python error in which we have used improper syntax, or coding language. This means that when our computer tries to execute the above code, it does not know how to interpret the line. You will frequently encounter errors like this in your journey as a programmer.

It is always important to read the error message as it will inform you about the specifics of the problem. For example, I can see that my error was triggered because of something in the input cell 6. If you were executing this same command within a Python script, you would see the specific line that triggered the error. This allows you to begin to **debug** or identify the source of the error and fix it. If, for some reason, you cannot understand the cause of the error, it may help to Google the error message and search for answers on forums, such as StackOverflow.

Sometimes, it will be necessary to have a very long string that spans multiple lines within a Python script. In these instances, you can use three of the same quotation mark style (single or double) consistently to create a string over multiple lines.

```
long_string = '''
This is a verrry long string.
'''
```

```
print(long_string)
```

```
This is a verrry long string.
```

2.1.3 Working with Strings as Data

Often when working within Python, you will not be simply creating data, you will be manipulating it and changing it. Because humanists frequently work with strings, I recommend spending a good deal of time practicing and memorizing the basic ways we work with strings in Python via the built-in methods.

In order to interact with strings as pieces of data, we use methods and functions. The chief functions for interacting with strings on a basic level come standard with Python. This means that you do not need to install third-party libraries. Later in this textbook we will do more advanced things with strings using third-party libraries, such as Regex, but for now, we will simply work with the basic methods.

Let's learn to manipulate strings now through code, but first we need to create a string. Let's call it sentence.

```
sentence = "I am going to learn how to program in Python!"
```

2.1.3.1 Upper Method

It is not a very clever name, but it will work for our purposes. Now, let's try to convert the entire string into all uppercase letters. We can do this with the method `.upper()`. Notice that the `.upper()` is coming after the string and within the `()` are no arguments. This is a way you can easily identify a method (as opposed to a function). We will learn more about these distinctions in the chapters on functions and classes.

```
print(sentence.upper())
```

```
I AM GOING TO LEARN HOW TO PROGRAM IN PYTHON!
```

2.1.3.2 Lower Method

Notice that our string is now all uppercase. We can do the same thing with the `.lower()` method, but this method will make everything in the string lowercase.

```
print(sentence.lower())
```

```
i am going to learn how to program in python!
```

2.1.3.3 Capitalize Method

On the surface, these methods may appear to only be useful in niche circumstances. While these methods are useful for making strings look the way you want them to look, they have far greater utility. Imagine if you wanted to search for a name, "William", in a string. What if the data you are examining is from emails, text messages, etc. William may be capitalized or not. This means that you would have to run two searches for William across a string. If, however, you lowercase the string before you search, you can simply search for "william" and you will find all hits. This is one of the things that happens on the back-end of most search engines to ensure that your search is not strictly case-sensitive. In Python, however, it is important to do this step of data cleaning before running searches over strings.

Let's explore another method, `.capitalize()`. This method will allow you to capitalize a string.

```
first_name = "william"
```

```
print(first_name.capitalize())
```

```
William
```

I will use this in niche circumstances, particularly when I am performing data cleaning and need to ensure that all names or proper nouns in a dataset are cleaned and well-structured.

2.1.3.4 Replace Method

Perhaps the most useful string method is `.replace()`. Notice in the cells below, replace takes a mandatory of two arguments, or things passed between the parentheses. Each is separated by a comma. The first argument is the substring or piece of the string that you want to replace and the second argument is what you want to replace it with. Why is this so useful? If you are using Python to analyze texts, those texts will, I promise, never be well-cleaned. They may have bad encoding, characters that will throw off searches, bad OCR,

multiple line breaks, hyphenated characters, the list goes on. The `.replace()` method allows you to quickly and effectively clean textual data so that it can be standardized.

Unlike the above methods, for `.replace()`, we need to put two things within the parentheses. These are known as arguments. This method requires two and they must be in order. The first is the thing that you want to replace and the second is the thing that you want to replace it with. These will be separated by a parentheses and both must be strings. It should, therefore, look something like this:

```
.replace("the thing to replace", "the thing you want to replace it with")
```

In the example below, let's try and replace the period at the end of "Mattingly."

```
introduction = "My name is William Mattingly."
```

```
print(introduction.replace(".", ""))
```

```
My name is William Mattingly
```

Excellent! Now, let's try and reprint off `introduction_sentence` and see what happens.

```
print(introduction)
```

```
My name is William Mattingly.
```

Uh oh! Something is not right. Nothing has changed! Indeed, this is because strings are immutable objects. Immutable objects are objects that cannot be changed in memory. As we will see in the next chapter with lists, the other type of object is one that can be changed in memory. These are known as mutable objects. In order to change a string, therefore, you must recreate it in memory or create a new string object from it. Let's try and do that below.

```
new_introduction = introduction.replace(".", "")
```

```
print(new_introduction)
```

```
My name is William Mattingly
```

2.1.3.5 Split Method

Strings have a lot of other useful methods that we will be learning about throughout this textbook, such as the `split()` method which returns a list of substrings, or smaller strings, that are split by the delimiter, which is the argument of the method. The delimiter tells Python how to split the string. By default, `split()` will split your string at the whitespace. Let's try and split the following string.

```
book_name = "Harry Potter and the Chamber of Secrets"
```

As with `replace`, `split()` is a method and, therefore, we use it with a `"."` after the variable name.

```
print(book_name.split())
```

```
[ 'Harry', 'Potter', 'and', 'the', 'Chamber', 'of', 'Secrets' ]
```

Our book name is now split into individual words thanks to the `split` function's default delimiter which is the white space. Notice, however, that the delimiter vanishes from our list of substrings. In the next chapter, we will learn about data structures, such as lists. In that section, we will learn how to grab specific indices, or sections, of the list.

`Split` can also take an argument that identifies where to split a string, e.g. something other than a whitespace. Let's say, I were interested in grabbing the subtitle of the book name. I could split the string at the "and".

```
print(book_name.split(" and "))
```

```
[ 'Harry Potter', 'the Chamber of Secrets' ]
```

As we can see, we have successfully separated the title from the subtitle. We will be working with strings a lot more as we progress through this textbook. You should at this point be familiar with what strings are, how to create them, and how to interact with them through some basic methods, such as `upper()`, `lower()`, `capitalize()`, `replace()`, and `split()`.

2.1.4 Numbers (Integers and Floats)

Numbers are represented in programming languages in several ways. The two we will deal with are integers and floats.

An integer is a digit that does not contain a decimal place, e.g. 1 or 2 or 3. This can be a number of any size, such as 100,001,200. A float, on the other hand, is a digit with a decimal place. So, while 1 is an integer, 1.0 is a float. Floats, like integers, can be of any size, but they necessarily have a decimal place, e.g. 200.0020002938. In Python, you do not need any special characters to create an integer or float object. You simply need an equal sign. In the example below, we have two objects which are created with a single equal sign. These objects are titled `an_integer` and `a_float` with the former being an object that corresponds to the integer 1 and the latter being an object that corresponds to the float 1.1.

Examples of Numbers

```
int1 = 1
```

```
print(int1)
```

```
1
```

```
float1 = 1.1
```

```
print (float1)
```

```
1.1
```

2.1.5 Working with Numbers as Data

Now that you understand how strings work, let's begin exploring another type of data: numbers. Numbers in Python exist in two chief forms:

- integers
- floats

As noted above, integers are numbers without a decimal point, whereas floats are numbers with a decimal point. This is an important distinction that you should remember, especially when working with data imported from and exported to Excel.

As digital humanists, you might be thinking to yourself, "I just work with text, why should I care so much about numbers?" The answer? Numbers allow us to perform quantitative analysis. What if you wished to know the times a specific author wrote to a colleague or to which places he wrote most frequently, as was the case with the Republic of Letters project at Stanford?

To perform that kind of analysis, you need a command of how numbers work in Python, how to perform basic mathematical functions on those numbers, and how to interact with them. Further, numbers are essential to understand in order to perform more advanced functions in Python, such as loops, explored in Chapter 3.

Throughout your digital humanities project, you will very likely need to manipulate numbers through mathematical operations. Here is a cheatsheet of the common operations.

Cheatsheet for Mathematical Operations

operation	code	description	example	result
addition	+	adds numbers together	1+1	2
subtraction	-	subtracts one number from another	1-1	0
multiplication	*	multiplies two numbers	1*2	2
exponential multiplication	**	performs exponential multiplication	2**2	4
division	/	divides one number from another	2/2	1
modulo	%	remainder	2%7	1
floor	//	occurrences	2//7	3

The way in which you create a number object in Python is the to create an object name, use the equal sign and type the number. If your number has a decimal, Python will automatically consider it a float. If it does not, it will automatically consider it an integer. Let's create an integer and a float.

```
an_integer = 1
print(an_integer)
```

1

```
a_float = 1.1
print(a_float)
```

1.1

2.1.6 Booleans

The term boolean comes from Boolean algebra, which is a type of mathematics that works in binary logic. Binary is the basis for all computers, save for the more nascent quantum computers. Binary is 0 or 1; off or on; true or false. A boolean object in programming languages is either `True` or `False`. `True` is 1, while `False` is 0. In Python we can express these concepts with capitalized T or F for True or False. Let's make one such object now.

Examples of Booleans

```
bool1 = True
```

```
print(bool1)
```

```
True
```

2.1.7 Conclusion

This section has introduced you to some of the essential types of data: strings, integers, floats, and booleans. It has also introduced you to some of the key methods and operations that you can perform on strings and numbers. Before moving onto the next chapter, I recommend spending some time and testing out these methods on your own data. Try and manipulate an input text to locate and retrieve specific information.

2.2 Introduction to Data Structures

2.2.1 Data Structures

In the last section, we met strings, integers, floats, and booleans. Each of these were types of data. Strings, for example, allowed us to work with text and numbers allowed us to work with integers and floats. In this chapter, we will begin working with **data structures**. Data structures are ways of storing multiple kinds of data in a systematic way. In Python, these are created as objects that can be stored in memory and called later in a script. They are divided into two categories: mutable and immutable. We encountered these terms in the last section, but we will explore what they mean in more depth below.

Throughout this section, we will learn about some of the key types of data structures, how they are different, and how they can be used. We will only cover these in a cursory manner. Throughout this textbook, we will use these data structures as we write code and perform data cleaning and data analysis tasks. To keep things simple for now, we will focus on four types of data structures: lists, tuples, sets, and dictionaries. There are other types of data structures in Python, but these are the core four that you will use most frequently.

2.2.2 Lists

The first data structure we will work with is known as a **list**. Lists are precisely what they sound like, a list of data. As we will see below, there are multiple ways of storing information in a list-like manner in Python, such as with tuples and sets, but the way we create lists and way we interact with lists is distinct.

Lists and tuples are identical with one major exception: lists are mutable. This means that you can create a list object and then alter it in memory. This allows for you to do very powerful things to lists that you cannot do to tuples. And these are going to be one of the key data structures you use in all digital humanities projects. The reason? We often need to adjust data while working with it.

As with data, we can create a list object in memory by creating a variable followed by an equal sign. To tell Python that the specific type of object we are creating is a list, we use an open and a close bracket. Each item in the list will be separated by a comma. Lists can store any type of data. To see this in action, let's create our first list.

```
first_list = [1, 1.0, "one"]
print(first_list)
```

```
[1, 1.0, 'one']
```

2.2.2.1 Indexing a List

In Python, we will frequently need to access a piece of data within a list or some other data structure. This is known as **indexing**. The way in which we index a list is with an open and a close bracket within which we place the position at which the data sits that we want to access. It is important to note that Python is a **zero-index language**. This means that we always begin with the number 0 and then count upward, so the item that sits in the first position in our list is index 0.

Let's grab the item at index 0 in `first_list`.

```
print(first_list[0])
```

```
1
```

In the cell below try to grab the string “one” from `first_list`.

```
print(first_list)
```

```
[1, 1.0, 'one']
```

Notice that we have printed off successfully the number 1. Often times, though, it is important to index multiple items in a list. If we want to do this, we use “[]” again. Within the brackets we will have a start position and an end position. The end position will be

the point after we want to grab. These will be separated by a “:`”. In code, it would look something like this:

```
index_item[start:end]
```

Let's say, we wanted to grab the first three items from the list, we would want to do something like this.

```
print(first_list[0:2])
```

```
[1, 1.0]
```

We can also work backwards with indexing. We can, for example, use a -1 to grab the final item in the list.

```
print(first_list[-1])
```

```
one
```

We can also use range indexing to grab the final three items. In Python, if you index a list with no end point, it will grab everything up to the end of that list. We can see this in the two examples below.

```
print(first_list[-2:])
```

```
[1.0, 'one']
```

We can likewise do the same in reverse by grabbing all indices up to the first index. In other words, the item in index 0.

```
print(first_list[:1])
```

```
[1]
```

2.2.3 Tuples

Tuples are lists of data that cannot be changed. When we look at lists above, we will see that lists are the exact same thing as tuples, except they can be changed. We can distinguish tuples from lists by the way in which they are formed. While lists use square brackets, tuples use parentheses. We create a tuple, like the example below. Our tuple object is `a_tuple` and the tuple consist of three items: an integer 1, a float 1.0, and a string of “one”. Lists and tuples can contain all three of these types of data. The way in which we separate items in a tuple is with a comma.

```
first_tuple = (1, 1.0, "one")
```

```
print(first_tuple)
```

```
(1, 1.0, 'one')
```

In the Tricket application above, try to create a tuple and index it.

2.2.4 Mutability vs Immutability

As noted above, tuples are immutable which means they cannot be changed. Let's see precisely what this means in practice. Say, we wanted to add to a list. We can do this with the `.append()` method. This will take one argument, or piece of information placed between the parentheses. You will learn about arguments later when we discuss functions and methods in greater depth. For now, understand that the information passed between the parentheses tells the method or function what is needed to perform the function. In this case, `.append()` allows us to append, or add, something to a list. The argument that we pass, "one", tells what we want to append. In this case, the string, "one".

```
first_list.append("one")
```

```
print(first_list)
```

```
[1, 1.0, 'one', 'one']
```

Notice that we do not have an error. This is because our list is mutable, or changeable. This means that we can add to it, delete items from it, and other operations that allow us to change how it is stored in memory. Tuples, on the other hand, are immutable, or unchangeable. Let's try and perform the same method on the tuple and see what happens.

```
first_tuple.append("one")
```

Notice that we get an `AttributeError`. This means that a tuple does not have the ability to use the `append` method. This does not exist for tuples because they are immutable or unchangeable. The only way to alter the object name, `first_tuple`, is to entirely replace it in memory.

2.2.5 Sets (Bonus Data Structure)

There is one other data structure similar to lists and tuples and I include it here as a bonus data structure. This is the set. A set is identical to a list. It is mutable, meaning we can update it, but unlike a list, it cannot contain duplicates. This is useful in niche circumstances, such as when you need to remove all duplicates from a list. I include it here just so that you are aware that other types of data structures do exist.

```
first_set = {1, 1.1, "one", "one"}
print(first_set)
```

```
{1, 'one', 1.1}
```

2.2.6 Dictionaries

Like tuples and lists, dictionaries are a data structure in Python. Like lists, dictionaries are mutable, meaning they can be changed in memory. Unlike tuples and lists, dictionaries are not lists of data. Instead, they have two components: keys and values. These two components are separated by a colon. All of this is contained within squiggly brackets. In the example below, we have a dictionary, `a_dict`, with a key of “name” and a value of “William”.

```
names = {
    "first_name": "William",
    "last_name": "Mattingly"
}
```

```
print(names)
```

```
{'first_name': 'William', 'last_name': 'Mattingly'}
```

In digital humanities projects, dictionaries are particularly useful for structuring complex data that you may have in Excel with each key being an Excel column and each value being its corresponding value. The dictionary name could be the name of the individual to whom the row corresponds. Like lists and tuples, you can embed data structures within a Python dictionary.

While we could realistically store our data in a list such as the one below (`name_list`), we would need to be consistent and always place the first name in index 0 and last name in index 1. This introduces potential issues later in a project. Imagine if one programmer left a project. Without good documentation, there is nothing inherent in this list that equates index 0 to first name and index 1 to last name. It is entirely up to the reader of the data to make sense of this data structure.

Remember, in programming it is always best to be explicit and produce readable code that others can understand. The dictionary allows us to create keys that indicate with greater specificity about the type of data with which we are working. We know from the `names` dictionary above that “William” is a first name and “Mattingly” is a last name without having to think about which index each string resides. We can do this because the keys of the dictionary are explicit.

```
name_list = ["William", "Mattingly"]
print(name_list)
```

```
['William', 'Mattingly']
```

2.2.6.1 Indexing Dictionaries

In Python, we will frequently need to index a dictionary. Dictionaries, remember, are a bit different from lists and tuples. Rather than being a sequence of items in a list, a dictionary is a collection of keys and corresponding values. To index a dictionary, therefore, we need

to work a bit differently. Rather than indexing at a specific point, we index dictionaries at a specific key.

To understand this, it is best to see it in practice, so let's go ahead and try to grab the first name in our dictionary names.

```
print (names["first_name"])
```

```
William
```



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

3

Loops and Logic

3.1 Introduction to Loops

3.1.1 What Are Loops?

Loops are a fundamental concept in all programming languages. They are essential in nearly every piece of code that I write and my experience is far from unique. Loops are essential because they allow you to systematically iterate over data. Iteration is the process by which you move across a piece of data or collection of data. As we will learn in this section, there are two types of loops in Python:

1. For loops
2. While loops

While most tasks can be achieved with either loop, the way in which you use them is different. By the end of this section, you will understand not only how to construct a loop in Python, but have a general idea about when to use which loop. As you code more and more, you will learn that coders tend to favor one type of loop over the other. For me (pun intended), I prefer the for loop. If you asked me why, I could not give you an answer. Maybe by the end of this section, you will have a favorite as well!

3.1.2 For Loops

A good way to think about a for loop is to first imagine in your mind a list of names:

Tom, Nancy, Drew, Steph.

Notice that in our list above, we have four names. Each name is separated by a comma. The for loop allows us to move across this list where each comma occurs. Still to this day, when I write a for loop, I say the following in my mind:

“For each item in this list, do something.”

Let’s imagine that we wanted to use Python to print off each of the names. I would say in my mind:

“For each item in this list, print off the name.”

I have not written a single bit of code, rather expressed a type of logic, or series of operations in regular English. This is known as **pseudo-code**. Pseudo-code is a great way to thinking about code because it is language-agnostic. By this I mean that the same pseudo-code can be applied to Python and C alike. It is useful to think about psuedo-code because it focuses on the logic of the statements, not the code. As you write more complex programs, you will find that it is not always the programming language that is difficult and produces errors, rather the logic behind how you have rendered that language. Pseduo-code lets you first figure out the logic of what you want to do, then you can focus on the execution of that logic.

The above statement in English (“For each item in this list, print off the name.”) will not work if I write it in a Python file, but it allows for me to get the logic of what I want to do down on paper. Next, I would need to write out this logic into normal Python syntax. Let’s try and do that now.

First, let’s make a list of names by creating an object named `name_list`. You may also want to call this variable `nameList`. Either convention is fine.

```
name_list = ["Tom", "Nancy", "Drew", "Steph"]
```

Great! Now that we have our `name_list` variable, let’s try and iterate over it, or pass over each item in that list. To do this in Python, we would write something like this:

```
for name in name_list:
    print(name)
```

```
Tom
Nancy
Drew
Steph
```

Let’s break down a series of things that are happening here. We start off by writing “`for`”. This tells Python that we are about to enter a for loop and that it should check for proper syntax. Next, we have the word “`name`”. This is a variable that will point to an object created in memory. It will be changed each time the loop iterates over the next item in the expression, “`name_list`”. This word “`name_list`” corresponds to the object name that I want to iterate over. Finally there is a “`:`”. In Python, this is the way that we note a nested portion of code. We will use these in nearly every script we write because we frequently need to nest items in code. Once Python sees a “`:`”, it will expect the next line to be indented.

On the next line, we have an indent followed by `print`. If you remember correctly, the `print` function allows us to print something off in our output. Next, we have what we want to be printed, the variable “`name`”. In other words, we want each name that is temporarily created in memory to be printed off.

Notice in the block of code below, I have changed the object “`name`” to “`item`”. The name here does not matter. It is simply a word that will indicate the name of the object that you are creating. It is considered Pythonic, or good form, to select a name for the object that makes sense. Because this is a `name_list`, it makes sense to use the word “`name`” for that object name so that others who read your code will understand it better.

```
for item in name_list:
    print(item)
```

```
Tom
Nancy
Drew
Steph
```

While being able to print things off in a for loop is useful, especially when debugging, usually for loops are used to modify data in some capacity. In the cell below, I want to be able to create a new list based on the list of names in `namelist`. However, I know that all individuals in that list have the last name “`Mattingly`”. I could manually add the last name `Mattingly` to each individual person, but that would be tedious and, if I had a list of thousands of names, impractical.

Within the for loop, therefore, I want to modify the existing string and add it to the new list, “`new_names`”. We can modify a string in several ways. For now, we will use the simpler

approach of simply adding a plus sign (+) between the object name and the new piece of information I want to add to it, " Mattingly"

```
new_names = []
for name in name_list:
    print (name+" Mattingly")
    new_names.append(name+" Mattingly")
```

```
Tom Mattingly
Nancy Mattingly
Drew Mattingly
Steph Mattingly
```

Now that we have executed the cell above, let's see how our new list looks. Let's print it off.

```
print(new_names)
```

```
['Tom Mattingly', 'Nancy Mattingly', 'Drew Mattingly', 'Steph Mattingly']
```

Viola! Like magic, we have brought two different pieces of data together in a for loop.

3.1.3 List Comprehension

Sometimes in someone's code you will see this same idea expressed in a single line of code. It will look like this:

```
new_names_comprehension = [name+" Mattingly" for name in name_list]
print(new_names_comprehension)
```

```
['Tom Mattingly', 'Nancy Mattingly', 'Drew Mattingly', 'Steph Mattingly']
```

This is known as **list comprehension**. List comprehension is a way of creating a list on a single line with a loop appearing within the open and close brackets. This single line does precisely the same thing as the four lines of code above, but allows a programmer to express it in **tighter code**. Tighter code is a phrase often used to describe code that is more concise. This is often what polished, final code should look like and it is a style in which computer scientists in particular try to write. While tighter code is often considered more polished, it can be more difficult for humanists or novice programmers to understand. Like writing in a spoken language, the purpose of writing in code is also to be readable. You should know your audience. If your audience are those who are not computer scientists, perhaps writing more verbose code is better.

Let's parse out the components of this single line. First, we have the object that we want to create with the variable name "new_names_comprehension" we set this equal to an open bracket. Within this open bracket we state what we want to do to the temporary variable created across the loop. In this case, that variable is again "name". Again, we specify that we want to add " Mattingly" to that variable name. Next, we specify the loop, e.g. "for name in name_list". Finally, we use a closed bracket to indicate the end of that statement.

For now, focus on being more verbose and writing out your code as seen in the first example. This will help you get the logic down and debug issues as they surface more easily as your bugs will appear on a specific line that caused the error. When you get more comfortable with Python, then try to use list comprehension.

I am including list comprehension here not so that you use it, but so that you will see it here first in a controlled setting. You will likely see this in GitHub repositories and on

StackOverflow with little explanation. Now that you have seen what list comprehension looks like and understand its components, you will be able to parse that code a bit more easily.

3.1.4 Indexing a List with and without Enumerate

Often when we create a loop, we need to understand where we are in a specific index of a list. As you will find with programming, there are multiple ways to do this.

Let's explore a simple approach first so that you can understand the logic behind the approach. Next, we will see a cleaner, but more complex example.

```
i = 0
for name in name_list:
    print(i)
    print(name)
    i=i+1
```

```
0
Tom
1
Nancy
2
Drew
3
Steph
```

In the above example, everything is nearly identical to our initial for loop with one clear addition, the variable “i”. The name “i” here represents a counter that we are storing outside of our loop. We initially set this variable to 0. As we iterate over our loop, we conclude each pass with the code “`i=i+1`”. This command says to Python take the object “i” and whatever that number is add 1 to it. This means that each time we pass over our `name_list`, we know precisely where we are in the index.

We can write this exact same code, however, by using the built-in function “`enumerate`”. `enumerate` automatically creates a variable for us during our loop and ticks it up each time we pass over an item in the list. We can use `enumerate` to write cleaner and tighter code.

```
for i, name in enumerate(name_list):
    print(i)
    print(name)
```

```
0
Tom
1
Nancy
2
Drew
3
Steph
```

Notice that we have the precise same result as above. Let's break down precisely what is happening in this example.

The only thing that has changed is the following line:

```
for i, name in enumerate(name_list):
```

Notice the addition of “i, name”. This means that we have two variables that we will be creating each time we loop. When using enumerate, the very first variable should always be “i” or something that points to the integer that will count up. Next, we have a comma. This separates the two variables out in the loop. Next, we have the item in the list that we create as we iterate over our data. This is the same variable name that we used before, “name”. Finally, we have “enumerate(name_list):” This tells Python to use enumerate on the name_list. The enumerate function is what creates the “i” for us in memory.

In the area below, try to create your own list and iterate over it. Next, try to use enumerate and print off the index of each item in the list.

```
<IPython.lib.display.IFrame at 0x7fa632fabe50>
```

3.1.5 Operators

We will very often in loops need to identify Comparison Operators (equal to, less than, etc). Here is a list of them:

1. Equal to (==)
2. Greater than (>)
3. Less than (<)
4. Less than or equal to (<=)
5. Greater than or equal to (>=)
6. Not equal to (!=)

Operators allow us to return a Boolean (True or False) about the question we pose with them. Say, for example, we wanted to know if 1 was less than 2. This is True, but in Python we could structure it like so:

```
print(1 < 2)
```

```
True
```

We could also structure the false statement of if 1 is greater than 2.

```
print(1 > 2)
```

```
False
```

Or even if 1 is equal to 2. Notice the double = here. We have to use two = because one = in Python sets up a new variable.

```
print(1 == 2)
```

```
False
```

In math, we use an equal sign with a slash through it to state not equal to, but in code that does not work so cleanly because it is not a character on the keyboard. Instead, we use

`!=`. Believe it or not, you will get really good about writing `!=` over time. If we wanted to see if 1 is not equal to 2, we would write this:

```
print (1 != 2)
```

```
True
```

On the surface, it may seem like you would never need to use comparison operators. You know that 1 is not equal to 2. In fact, you will use them all the time because comparison operators allow for you to leverage Boolean logic, or True–False logic. This is particularly useful in loops. A good way to demonstrate this is with a while loop.

3.1.6 While Loops

These operators allow us to structure complex conditions within our loop. So, we can say that while something is equal to something else, Python should do x. We will see this same concept in the following section as we explore conditionals. I think the best way to learn about this concept is to jump in and explore it.

In the for loop, the loop iterated over a set of data. A while loop is a bit different. In a while loop, the loop will run continuously while something is true. Like the for loop, we create the loop with a set of commands beginning with its name, while. Next we state the condition to be met that will result in breaking, or stopping the loop. Let's say we wanted to count from 0 to 10. We would create an object named x and set that equal to our start position. Then we would state so long as x `!=` (not equal to) 10, print off x. In order to ensure that x ticks up, we need to make sure that we change the object from x to x+1 so that it moves up by 1 number each time.

```
x=0
while x != 10:
    print (x)
    x=x+1
```

```
0
1
2
3
4
5
6
7
8
9
```

I know I have stated this before, but it is worth mentioning again. In coding, there is usually never one way to do a task. Notice in the block of code below what is different? Why don't you take a look at it and see if you can figure out precisely what is happening and why it succeeds in performing the same task.

```
x=0
while x < 10:
    print (x)
    x=x+1
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

3.2 Conditionals

Like loops, conditionals are an essential component of all programming languages. Conditionals are a type of logic in programming that allow you to control if something happens based on condition that something is true or false. Conditionals are always binary. I always find it useful to think about conditionals in plain English first.

If something is true, then do this. If that something is not true, then you can specify what should happen if that is the case. In pseudo-code, it would look something like this:

```
if something is true:  
    do this  
if not:  
    do this instead.
```

Notice again the indentation. As we progress throughout this chapter, you will learn three essential components of conditional statements.

Cheatsheet for Conditionals

code	description
if	Functions as the start of a condition
else	Functions as "if not"
elif	Functions as "or if"

3.2.1 If Statement

Let's begin with the `if` statement by first creating an object called `x` and make it equal to 0. Now, we can say `if x is equal to 0`, then print it off. Notice in the code below the `:` and the indentation afterwards. Just as we saw with loops, the `:` indicates that the next line should be indented. Whenever you have a line of code that uses a conditional statement, a `:` must always be at the end of the line to indicate to Python where that condition ends. The next line also, must always be indented.

```
x=0  
if x==0:  
    print (x)
```

```
0
```

We have successfully printed off the `x` variable because the condition we stated is `True`. Let's try and state the opposite of this. Let's set our condition to `x == 1`.

```
if x==1:
    print (x)
```

Notice that nothing happens. This is because our condition is False.

3.2.2 Else Statement

If we want something to occur if that condition is not met, we can use the `else` statement. Think of `else` as “if not” in English. In pseudo-code, `else` would look something like this:

```
if this condition is True:
    do this
if not (else):
    do something else
```

Let's write that out in real code.

```
if x==1:
    print (x)
else:
    print ("X is not 1")
```

```
x is not 1
```

Notice that `else` has the same indentation placement as the `if` and it too has a “`:`” followed by indented code. As noted above, all conditional statement lines must conclude with a colon and the proceeding line be indented.

The `else` statement is important because it allows a programmer to create more complex logic.

3.2.3 Elif and the ‘in’ Operator with Strings

I debated including `elif` in this section for fear that it may introduce too many new things at once, but I think it is import to at least be familiar with its existence, even if you will not be using it right away. In English syntax, `elif` functions rather like “or if”.

```
if condition 1 is True:
    do this
or if (elif) condition 2 is True:
    do this instead
```

Within this paradigm, it functions a little differently than `else`. In an `else` statement, we perform a specific action if the preceding condition was False. With `elif`, we state that another condition must be true.

On the surface, this may seem to function as two sequential `if` statements, but it functions a little differently than a regular `if` statement. To understand how, it is best to see the `elif` statement in code.

In the code below, you will see within the conditional the word `in`, known as an operator. In Python, `in` functions rather like the way it functions in English. It is used to test if something is inside a piece of data or a data structure. So, if we want to know if some string is within another string, we can use `in` to see if that is the case.

In the example below, we want to know if a substring, or a smaller string, appears within the `text` variable.

```
text = "I know two people. Marge and Susan."  
if "Marge" in text:  
    print ("Marge Found")  
elif "Susan" in text:  
    print ("Susan Found")
```

```
Marge Found
```

This output may appear to be surprising. Both Marge and Susan are in the string, “text”. Why, then, do we not see “Susan Found” printed off? The reason is because `elif` only is triggered if one of the preceding `if` or other `elif` statements has not been triggered. If I wanted to check if both conditions were met, I would, instead, use two `if` statements, like so:

```
if "Marge" in text:  
    print ("Marge Found")  
if "Susan" in text:  
    print ("Susan Found")
```

```
Marge Found  
Susan Found
```

The `elif` statement allows for more robust logic and sequential conditionals to occur. When you first start programming, you may not use `elif` frequently, but it is important, even early in your programming career, to be familiar with it.

3.2.4 Conditionals and Lists with ‘in’ and ‘not in’

Conditionals are frequently necessary when checking to see if something is in a list. Remember, lists are data structures that contain a list of data. The data contained within a list can be strings, integers, floats, or even other data structures, such as lists, tuples, and dictionaries. You will frequently need to understand what a list contains. To do this, we can use conditionals with the same `in` operator that we saw above.

Let’s begin by first making a list called `names`.

```
names = ["Terry", "Marge", "Joanne"]
```

Now that we have a list of names, let’s use the `in` operator to see if the name “Marge” is in `names`.

Warning: Remember: Python is case-sensitive.

In pseudo-code, we could write this as:

```
if Marge is in names:  
    then print off True
```

Because Python resembles English syntax so closely, we only have to make a few modifications to make this work in actual code.

```
if "Marge" in names:  
    print (True)
```

True

This same thing can also be done in reverse. What if we wanted to know if another person was not in names. We could write that as pseudo-code that looks something like this:

```
if Tom is not in names:  
    then print off Tom is not on the list
```

Python makes this construction simple because the `in` operator has a negative version to, `not in`. This operator functions precisely the same way, but it is the opposite of the `in` operator. Let's see it in action.

```
if "Tom" not in names:  
    print ("Tom is not in the list.")
```

Tom is not in the list.

Notice that we have successfully created a conditional statement that checked to see if a string is not in `names`. Both `in` and `not in` also work with checking if a substring appears in a string. If we wanted to see if a a substring `Jerry` appears in the text `Tom and Jerry`, we could create a conditional that looks something like this:

```
if "Tom" in "Tom and Jerry":  
    print ("Tom found.")
```

Tom found.

3.2.5 Conclusion

This section has introduced the three essential components of conditional statements: `if`, `else`, and `elif`. I would recommend practicing with these on your own data. The best way to learn about conditionals is to use them. Remember, as you write them in Python, speak out loud in English. I still speak aloud (or in my mind) pseudo code as I write in proper Python syntax.

4

Formal Coding: Functions, Classes, and Libraries

4.1 Functions

4.1.1 Introduction

In this section, you will learn all about functions. You have already been exposed to several functions in this textbook, such as the `print()` function. My goal in this section is to teach you how to create your own functions.

Functions exist in all programming languages. They allow you to write a set of code that can be used later in your program. They are essential because they allow you to not rewrite the same block of code over and over again. Good code is code that does not repeat. In other words, if your program has a task that it needs to do repetitively, then you should develop that code into a clean function (or class which we will meet in the next section).

A function, as we will see has five parts:

1. The name of the function
2. The arguments passed to the function (optional)
3. A docstring that explains the function (optional)
4. The code of the function
5. The returned data (optional)

By the end of this section, you will understand each of these components.

4.1.2 Functions in Action

I think the best way to learn about functions is to see them in action before breaking down what is happening. Let's take a look at the code below.

```
def adding_one(number):  
    x = number+1  
    return x
```

Let's break down the code above by starting with line one.

```
def adding_one(number):
```

Here, we use the word `def`. In Python, `def` stands for define because we are defining a function. When Python sees `def` it is expecting that what follows will follow the syntax of a function.

The next component in line one is `adding_one`. This is the name of the function. The `print` function's name, as you may expect, is `print`. This is what you will use when you want to call your function later in your script.

Next, we see `(number)`. A Python function name must be followed by parentheses. Inside your parentheses, you can place one or multiple parameters. These are optional. Parameters allow you to pass arguments to a function. These arguments can be used by the function to perform some sort of action by either manipulating that parameter or using it in some capacity to do some task. In the case of our function, the only parameter is `number`. If you are creating multiple parameters, these should be separated by a comma, as we will see below. Notice that parameters do not have quotation marks around them. This is because they are not strings, rather parameters that will function like variables inside the code of the function.

Finally, in this first line, we see “`:`”. This tells Python two things. First, the syntax for defining the function is complete and that the next line will be indented within which the code for your function will sit. This “`:`” and the indentation are mandatory.

The next line of code is:

```
x = number+1
```

This is the code of the function. Here we are creating a variable within the function called `x` which will be the result of the `number` passed to the function and `1`.

The final portion of the cell is:

```
return x
```

This will return for the user `x` that we created temporarily in the function.

Let's now call this function in the cell bellow to see how it works.

```
result = adding_one(1)
```

In the code above, we have created a new object, `result`. This will be the result of our function. Like all objects, it can be named whatever you like, save for the forbidden object names in Python.

We then call the function, `adding_one` and pass a single argument to it, an integer. In this case, `1`.

I want you to take a moment and try and guess what `result` will be. Go ahead and try and recreate this code in your own notebook, or, if you are using this textbook with the built-in binder environment, create a new cell below and print off “`result`”.

```
print(result)
```

Did you guess correctly? If so, great. If not, that's okay. Why do you think your answer was wrong? Do you understand why? These are some of the core questions you should be asking yourself at this moment.

If we give the function the number `1`, then our output will be `2`. Likewise, if we do this with `3`, we will get `4`. But what would happen if we tried to pass the string “`one`”?

```
bad_result = adding_one("one")
```

```
-----
TypeError                                 Traceback (most recent call last)
Input In [8], in <cell line: 1>()
----> 1 bad_result = adding_one("one")

Input In [2], in adding_one(number)
      1 def adding_one(number):
----> 2     x = number+1
      3     return x

TypeError: can only concatenate str (not "int") to str
```

We receive an error. This error message allows us to debug the problem. It is a `TypeError`, meaning we tried to use the wrong object type. We know that the error occurs in line two of our cell, `x = number+1`. Why do you think we received this error?

If you said because "one" is a string and you cannot add a string by the integer 1, then you would be right. If we were writing this program for ourselves, we would know this and never try to pass a string to the function, but what if someone else is trying to use our code and does not know precisely what it is supposed to do? In Python, we can pass in some key information to help our users.

4.1.3 Docstrings

To help users, we can provide them with a long string at the start of the function that explains what the function does. This is known as a `docstring`. Let's create a new function called `adding_two` with a `docstring`. This function will do the same as our earlier function, but add two to `number` and return that result.

```
def adding_two(number):
    """
    This function expects an integer and will return that integer plus 2.
    """
    x = x+2
    return x
```

Notice that the `docstring` is wrapped around three " at the beginning and end of the string. I can access this `docstring` by calling the function and using `.__doc__`.

```
print(adding_two.__doc__)
```

This function expects an integer and will return that integer plus 2.

4.1.4 Functions with Multiple Arguments

In the above section, we saw a simple function that had a single argument. In Python, you can assign as many arguments you want to a function. Let's try to make a slightly different function that adds two numbers together, each supplied by the user.

```
def my_function2(number1, number2):
    x = number1+number2
    return x
```

Notice that `my_function2` is the precise same as the function above except we have two arguments, "number1" and "number2". Also, in the function `x` is the result of `number1` plus `number2`.

Now, let's try and use this function by passing two numbers to it: 1 and 3.

```
result2 = my_function2(1,3)
print (result2)
```

4

Yay! We got the result we desired. These are known as positional arguments because we are relying on the position of the argument in the function's () section. Let's modify this function slightly again so that you can see what I mean.

```
def my_function3(number1, number2):
    print ("Number 1 is ", number1)
    print ("Number 2 is ", number2)
```

```
my_function3(1, 3)
```

```
Number 1 is 1
Number 2 is 3
```

Notice that I have deleted the return line. This is because this function does not need to return anything to the user. Instead, it's sole purpose is to simply print off what the two arguments are. Let's reverse the order of our arguments now.

```
my_function3(3, 1)
```

```
Number 1 is 3
Number 2 is 1
```

Because these are positional arguments, we are dependent upon the position of the arguments to assign them correctly. We can get around this by specifying which argument we want to assign things to, thus avoiding the reliance on the order in which we pass the arguments to the function. Check out the code below to see this in action.

```
my_function3(number2=3, number1=1)
```

```
Number 1 is 1
Number 2 is 3
```

4.1.5 Keyword Argument

Sometimes when we create a function, we want to make it optional for the user to pass an argument. In these instances, we will create what is known as a keyword argument, something that is set to a default. In the function below, we want to give the user the option to pass a last name to the function. Notice, though, it defaults to "Mattingly".

```
def add_surname(first_name, last_name = "Mattingly"):
    print (first_name, last_name)
```

```
add_surname("William")
```

```
William Mattingly
```

That worked well, but a user still has the ability to change the `last_name` object.

```
add_surname("William", "Smith")
```

```
William Smith
```

4.1.6 Keyword Arbitrary Arguments

In rare instances, you will not know precisely how many arguments a user will need to pass to your function, so you want to give them the option to pass as many as they wish. In these

instances, you will use what are known as arbitrary arguments. You assign these with an `**` before the argument name.

```
def print_names(*names):
    for name in names:
        print (name)
```

```
print_names("William", "Marge", "Sally", "Alex")
```

```
William
Marge
Sally
Alex
```

Notice that the function has allowed the user to pass as many arguments as they wish to the function. In my entire time coding, I have only used an arbitrary argument a handful of times, but it is important to keep it in the back of your mind in case you are ever in that situation.

4.1.7 Conclusion

Hopefully, you now have a basic understanding about what functions are and how they work in practice. I recommend spending a few hours playing around with some basic functions to do some tasks you need to perform on your own personal data. If you get stuck, try Googling your question. You would be surprised how many responses are available. Pay particular attention to StackOverflow responses.

4.1.8 Answer for Result

```
print (result)
```

```
2
```

4.2 Classes

4.2.1 Introduction

In this section, we will meet classes. **Classes** are rather like data structures, but they differ in one significant way. They can have functions attached to them. When you create a class in Python, you are essentially creating a special kind of data object that can have functions. Functions embedded within classes are known as **methods**. You have actually already met methods. When we first learned about strings, you may recall that when we altered the data, we used a `.` after the string object name followed by the function (method) that we wanted to call, e.g. `str1.replace(something, something_else)`. This is what separates a function from a method syntactically in Python. While a function is called by itself, a method must be called from a class object. While there is no easy way to explain this distinction, I hope that this explanation will be aided by working with classes and methods at a closer level below.

4.2.2 Creating a Class

Let's try creating a basic class. Our class will store data specifically related to emperors, so let's go ahead and give it the name `Emperor`. We expect each emperor in our dataset to have four attributes: name, birth, coronation, and death.

```
class Emperor:
    def __init__(self, name, birth, coronation, death):
        self.name = name
        self.birth = birth
        self.coronation = coronation
        self.death = death
```

This is the way this basic class will look. It can be a bit difficult to parse what is happening here when you see it for the first time, so let's break it down a bit.

In line one, we state:

```
class Emperor:
```

The first word we see is the keyword `class`. Python will see this and expect the syntax for a `class` object to follow. This is rather like the function's `def` which starts to define a new function.

The next thing we see is the class name. In our case, this is `Emperor`. After the class name, we have “`()`”. This is followed by “`:`” which concludes our first line. This also means that Python will expect the next line to be indented.

The next line reads:

```
def __init__(self, name, birth, coronation, death)
```

This is the structure you will use for most classes. The `def __init__()` is a special method that loads when the class instance is created. Next, we see the parameters the function will take within the “`()`”. There are five parameters. The first is `self` which points to the instance of the class itself. You should always write `self` here first. The next four parameters are `name`, `birth`, `coronation`, and `death`.

The `self` parameter also allows each of the attributes: `name`, `birth`, `coronation`, and `death` to be associated with a unique instance of class. This will become more clear as we proceed.

The next four lines are indented and read:

```
self.name = name
self.birth = birth
self.coronation = coronation
self.death = death
```

This function will attach these attributes and bind them to the instance of the class, by stating `self.name = name` and so forth. With this, our class is created. Now, let's try and make an object that is associated with this class.

As a Carolingianist, I studied Charlemagne, one of the most popular medieval kings who was the second Carolingian king. He was born in 742, coronated as Roman Emperor in 800 and died in 814. These are the four essential pieces of data we need for our class.

We can create our special class object `Emperor` for Charlemagne by writing the following line of code.

```
charlemagne = Emperor("Charlemagne", 742, 800, 814)
```

Notice that we create the class object by using the class name, `Emperor()` and passing four arguments associated with the four attributes: name, birth, coronation, death.

Let's now try and print off that class object.

```
print (charlemagne)
```

```
<__main__.Emperor object at 0x7fdd6c16fc70>
```

This is new and likely not what you expected. This indicates that the object is a special class object. We can get the data from this class object by using the `vars()` command.

```
print (vars(charlemagne))
```

```
{'name': 'Charlemagne', 'birth': 742, 'coronation': 800, 'death': 814}
```

We can access specific pieces of data in our class by calling our class object and then using `.name_of_attribute`. If we wanted to access Charlemagne's birth year, for example, we could use the following command.

```
print (charlemagne.birth)
```

```
742
```

At this point, you may be thinking to yourself: "âŁ! what's the big deal? I've just made a dictionary."

And, at this point, you would be right. Our class, the way it is structured is really nothing more than a dictionary of data stored in a special way. This really is not a good usecase for a class. Remember, what makes classes unique is the ability to attach functions to them. Let's learn how to do that now!

4.2.3 Adding Functions to a Class

To create a function within a class, we create a function within it. Let's make a simple function that will print off a string that states who the emperor is and when that emperor was born. To do that, we will alter a class as so:

```
class Emperor():
    def __init__(self, name, birth, coron, death):
        self.name = name
        self.birth = birth
        self.coron = coron
        self.death = death
    def birth_date(self):
        print (f"{self.name} was born in {self.birth}")
```

Notice that we have now added a function within our class called `birth_date()`. This will receive `self`, or all the data associated with that particular class. This function will now strictly print off a string that will state when a specific emperor was born.

Now, we just need to create a new class for Charlemagne.

```
charlemagne = Emperor("Charlemagne", 742, 800, 814)
```

Within this class, we can access the method `birth_date()`, like so:

```
charlemagne.birth_date()
```

```
Charlemagne was born in 742
```

And viola! You have now created your first class that has a specific function associated with it. Although we covered the basics of classes here, it should be noted that we necessarily did not cover everything. Classes have far more complexity and flexibility than presented here; nevertheless, the information provided above should give you enough of a sense about classes to keep moving forward.

Classes and functions are the two building blocks of any project. They allow for you to have tight, neater code that will be easier to read and look polished and professional. They prevent writing duplicate code. While it may appear unnecessary at times to view your code through functions and classes, I highly encourage you to start looking at it that way. It will make you a better programmer and you are less likely to have mistakes in your code.

4.3 Libraries in Python

4.3.1 Introduction

Libraries are common across all programming languages. They allow you to import large amounts of code that contain functions and classes that you can leverage in your own code. A good way to think about a library is as the old expression: “don’t reinvent the wheel”. Use what others have done! Open-source is open for a reason! People who make these libraries do it so that you don’t have to solve certain problems. They’ve done it for you.

In this section, we will learn how to install and import libraries. The reason we are doing this now is because the remainder of this textbook will require external libraries, specifically `pandas`, `requests`, and `BeautifulSoup`.

4.3.2 How to Install Python Libraries

Python comes preinstalled with pip. Pip is a package manager. It downloads, installs, and manages different versions of software on your machine. If you are coming to this textbook from windows, this concept might be a bit foreign. Think of pip as something that will manage all your libraries for you. If you have installed Python via Anaconda, as I recommended, then pip will be automatically put into your system’s path. Path on Windows is a way that your computer can understand commands for .exe files that you have on your system. If you open the terminal and you type “`pip-version`”. On Windows, the terminal is command prompt.

Because I am using JupyterNotebook for this textbook, I can make terminal commands with the “!” before a block of code. When you type “`pip-version`”, you should see something like the following output.

```
!pip --version
```

```
pip 21.2.4 from /home/wjbmattingly/anaconda3/lib/python3.9/site-packages/pip
(python 3.9)
```

If pip is working, then it means we can install the libraries that we will need for the remainder of the textbook. We will require three libraries:

- Pandas
- Requests
- BeautifulSoup

To install a library, you use the pip command “`pip install library_name`”, like so:
`!pip install pandas`

You should see an output similar to this. If you do, then pandas has installed correctly. Let’s now do the same thing for requests and BeautifulSoup.

```
!pip install requests
```

```
Requirement already satisfied: requests in /home/wjbmattingly/anaconda3/lib/
→python3.9/site-packages (2.27.1)
Requirement already satisfied: idna<4,>=2.5 in /home/wjbmattingly/anaconda3/
→lib/python3.9/site-packages (from requests) (3.3)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /home/wjbmattingly/
→anaconda3/lib/python3.9/site-packages (from requests) (1.26.8)
Requirement already satisfied: charset-normalizer~=2.0.0 in /home/
→wjbmattingly/anaconda3/lib/python3.9/site-packages (from requests) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /home/wjbmattingly/
→anaconda3/lib/python3.9/site-packages (from requests) (2021.10.8)
```

If you have already installed a library, the output will look something like the above.

For BeautifulSoup, we need to say “`pip install beautifulsoup4`” (I will explain why later in this textbook).

```
!pip install beautifulsoup4
```

```
Collecting beautifulsoup4
  Downloading beautifulsoup4-4.10.0-py3-none-any.whl (97 kB)
  |||||||..... 97 kB 2.5 MB/s eta 0:00:011
?25hCollecting soupsieve>1.2
  Downloading soupsieve-2.3.1-py3-none-any.whl (37 kB)
Installing collected packages: soupsieve, beautifulsoup4
Successfully installed beautifulsoup4-4.10.0 soupsieve-2.3.1
```

And that’s it! You have installed three new Python libraries! We need to learn one last thing, though, before we conclude this section. We need to learn how to import libraries within a Python script.

4.3.3 How to Import a Library

To import a library, we use the command `import library_name`, like so:

```
import requests
```

Sometimes, when we work with libraries, there is a certain Pythonic way to import the library. A classic case is pandas. With pandas, we import it “`as pd`”. The “`as pd`” means that the name in the script will not be “`pandas`”, rather “`pd`”.

```
import pandas as pd
```

4.3.4 Conclusion

That's all we need to cover in this chapter about libraries. You should feel a bit more comfortable about what libraries are, why they are useful, how to install them, and how to import them. As we continue through the final parts of this textbook, you will become more comfortable with libraries.

5

Working with External Data

5.1 Working with Textual Data

5.1.1 Introduction

Up until now, we have strictly worked with data that we have produced with our own code. Rarely will you ever copy-and-paste data into a Python script. Instead, you will need to interact with external data. Further, you will often need to save data in some way. This section is the first of two that walk you through how to interact with external data. We will be working with textual data in this section, while in the following section we will work with JSON, or JavaScript Object Notation data.

5.1.2 The “With” Statement

The “with” statement in Python is an essential command that allows for you to do something in memory for so long as the statement is open. This is really important when opening text files. There are several ways to open text files in Python, but I am only showing you this method because it is the most Pythonic. Other methods require you to close the text file in code. If you do not, it will remain open in memory. If you are working with 10,000 text files (I have done this before), and forget to add a close command in your code while iterating across all 10,000 files, your computer will run out of memory and crash. The “with” statement avoids this problem entirely.

5.1.3 How to Open a Text File

Let’s see what it looks like in action and then we will break down what is happening.

```
with open ('../data/sample.txt', "r") as f:  
    data = f.read()
```

The first line in our code is

```
with open (file, "r") as f:
```

We can see that we start clearly with the “with” statement. Next, we specify what we are going to do with the with statement. In this case, we will be using the “open” command. Open tells Python to open a file. Within the parentheses, we pass two arguments. The first argument is a string that corresponds to where the file is located. Here, the file is in our data subfolder and is called “sample.txt”. The final component of this line is “as f”. This tells Python to open that file up temporarily as the object name “f”. The final colon, which we have seen before, indicates that we are about to do something indented.

The next line is indented because this is a nested bit of code that is being done within the `with` statement. The line reads:

```
data = f.read()
```

The name `data` is the object name that we are assigning to the contents of the file. Next, we see “`f.read()`”. This command tells Python to take the “`f`” object, in this case the temporary file, open it and read its contents.

Now, let’s take a look at those contents! We can print them off with the `print` command.

```
print(data)
```

```
This is a sample of a text
This is another line of the same sample text
This is a third line.
```

We can see that the text file contains three lines. It is frequently necessary to split up input data by linebreaks. Sometimes this is necessary when the text file is a list of data with each piece of data stored on a new line. Other times, you need to clean the data so that a paragraph of a scanned book is a continuous string of text. One of the easiest ways to do this is to use the built-in method, `splitlines()`.

```
print(data.splitlines())
```

```
['This is a sample of a text', 'This is another line of the same sample text',
 →'This is a third line.]
```

As you can see, `splitlines()` allows us to convert our string into a list of strings that are separated by linebreaks.

5.1.4 How to Write Data to a Text File

Just as it is necessary to access external data via Python, you will find yourself frequently needing to store data outside of a Python script. Let’s take a look at how to do this in the following block of code.

```
new_string = "This is a new string."
with open("../data/sample2.txt", "w") as f:
    f.write(new_string)
```

This block of code looks very similar to the one above, but with a few exceptions. First, we created a string called `new_string`. Next, we need to drop that new string into a text file. In this case, we will drop it into `data/sample2.txt`. Notice, that we have replaced the “`r`” with “`w`”. This is because we are telling Python that we want to write to the file, not read it.

In the indented bit of code, we use `f.write()`, rather than `f.read()`. This allows us to write to the “`f`” object, rather than read from it. It will take one argument, the string that we want to write to the file. In our case, it is the string object, `new_string`.

5.2 Working with JSON Data

5.2.1 Introduction

In this section, you will meet JSON data. JSON is a data format that is frequently used online. It stands for JavaScript Object Notation. It is the principal way in which data is stored for websites and called by JavaScript. There are two reasons to learn JSON data structures early in your coding career:

1. JSON is universally recognized by all browsers.
2. JSON allows you to structure hierarchical data.

What is hierarchical data? This is data that may be nested within other data. This type of data is sometimes difficult to represent cleanly in CSV format. A good way to think about this is with the following data structure. Imagine you wanted to store a series of texts in Excel. In one column, you would have a text and then in the next column, you would store a list of speakers. Now, how do you represent the list of names? You could do it like this:

Text	Names
John and Kathy both know each other. Marge and Francois do not.	John, Kathy, Marge, Francois

But this can get messy really quickly because lists are difficult to store as a data structure within a cell. There are ways to get around this, but when you start working with this type of CSV data computationally, it can get complicated the deeper the hierarchies go. Imagine if for each person, we had an age and a role. How would we store that data? Perhaps like this:

Text	Names	Age
John and Kathy both know each other. Marge and Francois do not.	John, Kathy, Marge, Francois	20, 25, 30, 35

If I want to update the data, I need to make sure that the list of ages corresponds to the list of names. Again, this can lead to issues down the road. The solution is to stop using CSV or Excel to store this type of data and use a format that is more flexible and able to handle things like lists and nested hierarchical data. In Python, the easiest solution is JSON. To use JSON, you do not need to install any special libraries. It comes prebuilt with a special library called JSON. The only methods you need to know to use JSON: `json.dump()` and `json.load()`. But first, let's import JSON.

```
import json
```

Let's create the data above as a data structure within Python.

```
data = {"text": "John and Kathy both know each other. Marge and Francois do not.",  
       "names": ["John", "Kathy", "Marge", "Francois"]}  
data
```

```
{'text': 'John and Kathy both know each other. Marge and Francois do not.',  
 'names': ['John', 'Kathy', 'Marge', 'Francois']}
```

5.2.2 Writing JSON Data with `json.dump()`

Now, let's try and store that data outside of Python as a JSON file. To do that, we will use the `with open` operator we learned about in the last chapter. Instead of naming this a `.txt`

file, however, we will name is .json. Next, we will execute the command `json.dump()`. This will take 2 essential arguments: the data that you want to dump to the file and the object in which you want to dump the data, in this case “`f`”. The other keyword argument here is `indent`. I always like to use this because it makes the JSON file easier to read. It indents the data 4 spaces each time it goes deeper into the hierarchy.

```
with open ("data/sample_json.json", "w") as f:
    json.dump(data, f, indent=4)
```

5.2.3 Reading JSON data with `json.load()`

Now that we have dumped the data into a file, let’s try and load it back up. Here, we will open the same JSON file, but this time as readable. We will create a new object, `new_data` and use `json.load()`. This will take one argument, the file object that you want to load from. So long as your JSON file is not corrupted, the data will load successfully.

```
with open ("data/sample_json.json", "r") as f:
    new_data = json.load(f)
new_data
```

```
{'text': 'John and Kathy both know each other. Marge and Francois do not.',
 'names': ['John', 'Kathy', 'Marge', 'Francois']}
```

These are the only two commands that you need to know to start working with JSON data in Python. I highly encourage you to spend a few minutes playing with these commands and trying to store data with `json.dump()` and load data with `json.load()`.

5.3 Working with Multiple Files

5.3.1 Introduction

Often times, it is necessary to open multiple files in a Python script. There are multiple ways to do this, but unlike most Python textbooks. I recommend that beginners use the library called `glob`. `Glob` comes standard with Python which means you do not have to install it. A good way to think about `glob` is as a library that allows you to look into a directory and find all potential files based on certain parameters.

5.3.2 Working with Glob

Working with `glob` can be a little confusing at first, but let’s break it down. First, we need to import `glob`.

```
import glob
```

Next, we need to use the `glob` class. This will take one argument, the string of files that you want to find. Let’s use the `data` subfolder as an example. In the example below, we pass one string to this class. This string will be the subfolder in which the data lies followed by a slash followed by an asterisk. This asterisk is known as a wild card. In our case, it looks for all files within this directory.

```
files = glob.glob("data/asterisk")
```

Let's print off the files now to see what all is inside the folder.

```
print (files)
```

```
['data/other2', 'data/sample_json.json', 'data/other', 'data/sample2.txt', 'data/sample.txt']
```

Notice that we have grabbed all files! Most of the time, however, you will only want to grab files that are a specific type, e.g. .txt or .json. To achieve this we can add a .txt after the *. This will grab all the .txt files.

```
files2 = glob.glob("data/*.txt")
print (files2)
```

```
['data/sample2.txt', 'data/sample.txt']
```

5.3.3 Grabbing Multiple Nested Directories

If you look in the subdirectory data, you will notice two nested directories called "other" and "other2". We can grab all files in each directory with the same wildcard *.

```
files3 = glob.glob("data/**/.txt")
print (files3)
```

```
['data/other2/sample4.txt', 'data/other/sample3.txt']
```

5.3.4 Walking a Directory

While glob is easy-to-use, it has certain limitations. One of these is when you need to walk through a directory. Imagine if we needed to grab all the .txt files in data, data/other, and data/other2. In order to grab all of these, we need to walk through all the subdirectories of data and collect all potential files. This is not possible to do with glob. In these rare circumstances, you should be familiar with the os library.

The os library allows us to do a lot of more advanced things in Python that I will not cover in this introductory textbook. One of the main things you will use os for is to navigate directories and create directories. For Linux users, a lot of the syntax will be familiar, but for Windows users it can feel a bit foreign. For now, let's simply import os.

```
import os
```

Once we have imported os, we can use the os.walk command. This will take one string. This should correspond to the directory that you want to start walking. Imagine that this is our directory:

- data
 - other
 - other2

We want to get all the text files in data, other, and other2.

```
walking = os.walk("data/")
print (walking)
```

```
<generator object _walk at 0x7fcf42690eb0>
```

This above output tells us what what we are looking at is a generator. Generators are a bit beyond this textbook, but think of them as a special kind of object that exists for a single moment in memory. Whenever you see generators, you can usually convert them to a list to work with them. Let's convert it by using the `list()` function in Python

```
walking = list(walking)
print (walking)
```

```
[('data/', ['other2', 'other'], ['sample_json.json', 'sample2.txt', 'sample.txt']),
 ('data/other2', [], ['sample4.txt']),
 ('data/other', [], ['sample3.txt'])]
```

This can be a bit difficult to parse, so let's iterate over each item in "walking".

```
for item in walking:
    print (item)
```

```
('data/', ['other2', 'other'], ['sample_json.json', 'sample2.txt', 'sample.txt'])
('data/other2', [], ['sample4.txt'])
('data/other', [], ['sample3.txt'])
```

As we can see, each item is a tuple with three parts:

- root directory
- subfolders
- files

For our purposes, we only care about the root directory and the file itself. We can use these two pieces of data. Let's now modify our loop to grab these two pieces of data and print them off.

```
final_files = []
for item in walking:
    root = item[0]
    files = item[2]
    print ("This is the Root")
    print (root)
    print ("These are the Files")
    print (files)
    print ()
```

```
This is the Root
data/
These are the Files
['sample_json.json', 'sample2.txt', 'sample.txt']
```

```
This is the Root
data/other2
```

(continues on next page)

(continued from previous page)

```
These are the Files
['sample4.txt']
```

```
This is the Root
data/other
These are the Files
['sample3.txt']
```

As we can see, the files are a list. We can then iterate over the files to recombine the root with the files to cultivate a list. With os, we can do this with `os.path.join()`. This will take two arguments, the root directory and the file. Let's again modify our loop. We will print off the results.

```
for item in walking:
    root = item[0]
    files = item[2]
    for file in files:
        if file.endswith(".txt"):
            print(os.path.join(root, file))
```

```
data/sample2.txt
data/sample.txt
data/other2/sample4.txt
data/other/sample3.txt
```

Excellent! Now, we can use this exact same loop to append the file names to an empty list called `final_files`.

```
final_files = []
for item in walking:
    root = item[0]
    files = item[2]
    for file in files:
        if file.endswith(".txt"):
            final_files.append(os.path.join(root, file))
print (final_files)
```

```
['data/sample2.txt', 'data/sample.txt', 'data/other2/sample4.txt', 'data/other/
→sample3.txt']
```

Notice that we now have all the .txt files in the main directory and all subdirectories. This will be very useful when your files are in multiple subdirectories within multiple subdirectories.

5.3.5 Conclusion

I would recommend playing around with the code provided to you in this chapter. I cannot emphasize enough how frequently you will need to work with multiple files in a Python project. It is perhaps one of the things you should work hard to commit to memory. Over time, it will become more instinctual.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

6

Working with Data on the Web

6.1 Introduction to HTML

6.1.1 Introduction

In this section, we will learn about web scraping, one of the more vital skills of a digital humanist. **Web scraping** is a process by which we automate the calling of a server (which hosts a website) and parsing that request which is an HTML file. **HTML** stands for HyperText Markup Language. It is the way in which websites are structured. When we scrape a website, we write rules for extracting pieces of information from it based on how that data is structured within the HTML. To be competent at web scraping, therefore, one must be able to understand and parse HTML.

In this section, we will break down HTML and you will learn the most common tags, such as “div”, “p”, “strong”, and “span”. You will also learn about attributes within these tags, such as “href”, “class”, and “id”. It is vital that you understand this before moving onto the next section in which we learn how to web scrape with Python.

6.1.2 Diving into HTML

So why is HTML useful? HTML, like other markup languages, such as **XML**, or eXstensible Markup Language, allows users to structure data within data. This is achieved by what are known as tags. I think It is best to see what this looks like in practice. Let's examine a simple HTML file.

```
<div>
  <p>This is a paragraph</p>
</div>
```

Above, we see a very simple HTML file structure. In the first line of this HTML, we see `<div>`. Note the use of `<` and `>`. The opening `<` indicates the start of a tag in HTML. A **tag** is a way of denoting structure within an HTML file. It is a way of saying that what comes after this nested bit of code is this type of data. After the `<`, we see the word `div`. This word denotes the type of tag that we are using. In this case, we are creating a `div` tag. This is one of the most common types of tags in HTML. After the tag's name, we see `>`. This identifies the end of the tag creation.

In line two, we see a nested, or indented bit of HTML. In HTML, unlike Python, indentation is optional. It is, however, good practice to use line breaks and indentation in HTML to make the document easier for humans to parse. Line two begins with a `<p>` tag. The `<p>` tag in HTML is used to denote the start of a paragraph.

After the creation of the `<p>` tag, we see `This is a paragraph`. In HTML, text that lies outside of the tags is text that appears on the web page. In this case, the HTML file would display the text `This is a paragraph`. Immediately after this bit of text we encounter

our first close tag. A **close tag** in HTML indicates that this nested bit of structure is over. In our case, the first close tag is `</p>`. We know that it is a close tag because of the `</`, as opposed to `<`.

In our final line, we see a close `div` tag.

6.1.3 Understanding Attributes

Let's take a look at another block of HTML. This time, we will make one small change. Can you spot it?

```
<div class="content">
    <p>This is a paragraph</p>
</div>
```

If you said the `class="content"` portion of the open `div` tag, then you would be right. This bit is nested within the tag and is known as an **attribute**. In our case, the special attribute used is a `class` attribute (a very common attribute type). This attribute has a value of `content`.

When scraping websites, you can use these attributes to specify which `div` tag to grab. There are several common attributes, specifically `class` and `id`.

6.1.4 Parsing HTML with BeautifulSoup

Now that we understand a bit about HTML, let's start trying to parse it in Python. When we **parse** HTML, we attempt to automate the identification of HTML's structure and systematically interpret it. This is the basis for web scraping. To begin, let's create a simple HTML file in memory.

```
html = """
<html>
    <body>
        <div class="content">
            <p>This is our first content</p>
        </div>
        <div class="other">
            <p>This is another piece of content</p>
        </div>
    </body>
</html>
"""
```

There are many Python libraries available to you for parsing HTML data. For more robust problems, Selenium¹ is the industry standard. While Selenium is powerful, it has a steep learning curve and can be challenging for those new to Python, especially for those programming on Windows. Additionally, most web scraping problems can be solved without Selenium.

For those reasons, we will not be using Selenium in this textbook, rather BeautifulSoup². BeautifulSoup is a light-weight Python library for parsing HTML. It is quick and effective. The most challenging thing about BeautifulSoup is remembering how to install it and import it in Python.

¹<https://selenium-python.readthedocs.io>

²<https://pypi.org/project/beautifulsoup4/>

To install BeautifulSoup, you must run the following command in your terminal:

```
pip install beautifulsoup4
```

Once it is installed, you can then import BeautifulSoup in the following way:

```
from bs4 import BeautifulSoup
```

With BeautifulSoup imported, we can use the BeautifulSoup class. This class allows us to parse HTML. As we will see throughout this section, you will rarely have HTML as a string within your Python script, but for now, since we are starting, let's try to parse the above html string by passing it to the BeautifulSoup class. It is Pythonic to name your BeautifulSoup variable `soup`. If you have a more complex script that is parsing `soup` objects from multiple websites, you may want to be more original in your naming conventions, but for our purposes, `soup` will serve us well.

```
soup = BeautifulSoup(html)
```

With our `soup` object created in memory, we can now begin to examine it. If we print it off, we won't notice anything special about it. It appears as a regular string.

```
print(soup)
```

```
<html>
<body>
<div class="content">
<p>This is our first content</p>
</div>
<div class="other">
<p>This is another piece of content</p>
</div>
</body>
</html>
```

While it may look like a string, it is not. We can observe this by asking Python what type of object it is with the `type` function

```
type(soup)
```

```
bs4.BeautifulSoup
```

As we can see, this is a `bs4.BeautifulSoup` class. That means that while it may look like a string, it actually contains more data that can be accessed. For example, We can use the `.find` method to find specific the first occurrence of a specific tag. The `.find` method has one mandatory argument, a string that will be the tag name that you wish to extract. Let's grab the first `div` tag.

```
first_div = soup.find("div")
print(first_div)
```

```
<div class="content">
<p>This is our first content</p>
</div>
```

As you can see, we were able to grab the first `div` tag with `.find`, but we know that there are multiple `div` tags in our HTML string. What if we wanted to grab all of them? For

that, we can use the `.find_all` method. Like `.find`, `.find_all` takes a single mandatory argument. Again, it is the string of the tag name you wish to extract.

```
all_divs = soup.find_all("div")
print(all_divs)
```

```
[<div class="content">
<p>This is our first content</p>
</div>, <div class="other">
<p>This is another piece of content</p>
</div>]
```

Unlike `.find` which returns a single item to us, the `.find_all` method returns a list of tags. What if we did not want all tags? What if we only wanted to grab the `div` tags with a specific class attribute. BeautifulSoup allows us to do this by passing a second argument to `.find` or `.find_all`. This argument will be a dictionary whose keys will be the attributes and whose values will be the attribute names of the tags you wish to extract.

```
div_other = soup.find_all("div", {"class": "other"})
print(div_other)
```

```
[<div class="other">
<p>This is another piece of content</p>
</div>]
```

Once we have obtained the specific tag that we want to extract from the HTML, we can then access its nested components. Each `bs4.BeautifulSoup` object functions the same way as the original `soup`. It contains all children (nested) tags. We can, therefore, grab the `p` tag embedded within the `div` tag that has a `class` attribute of `other` by using `.find("p")`.

```
paragraph = div_other[0].find("p")
print(paragraph)
```

```
<p>This is another piece of content</p>
```

If we are working with textual data, though, we rarely want to retain the HTML, rather we want to extract the text within the HTML. For this, we can access the raw text that falls within the HTML with `.text`.

```
print(paragraph.text)
```

```
This is another piece of content
```

Being able to do this programatically means that we can automate the scraping of HTML files via Python. We can, for example, extract all the text from each `div` tag in our file via the following two lines of code.

```
for div in all_divs:
    print(div.text)
```

```
This is our first content
```

```
This is another piece of content
```

6.1.5 How to Find a Website's HTML

Now that you are familiar generally with HTML and how it works, let's dive in and take a look at some real-world HTML from a real website. In the next section, we will web scrape Wikipedia, so let's go ahead and focus on Wikipedia here as well. If you are using a web browser that supports it (Chrome and Firefox), you can enter developer mode. Each operating system and browser has a different set of hotkeys to do this, but on all you can right click the webpage and click "inspect". This will open up developer mode. At this point, I highly recommend switching over to the video as it will be a bit easier to follow along.

For this section (and the next), we will be working specifically with this page: https://en.wikipedia.org/wiki/List_of_French_monarchs.

Go ahead and open it up on another screen or in a new tab. This Wikipedia article contains some text, but primarily it hosts a list of all French monarchs, from the Carolingians up through the mid-19th century with Napoleon III.

When you inspect this page, you will see all the nested HTML within it. Spend some time and go through these tags. In the next section, we will learn how to scrape this page, but for now take a look at what we can do with a few basic commands in Python.

```
import requests
from bs4 import BeautifulSoup

url = "https://en.wikipedia.org/wiki/List_of_French_monarchs"
s = requests.get(url)

soup = BeautifulSoup(s.content)
body = soup.find("div", {"id": "mw-content-text"})
for paragraph in body.find_all("p")[:5]:
    if paragraph.text.strip() != "":
        print (paragraph.text)
```

The monarchs of the Kingdom of France ruled from the establishment of the Kingdom of the West Franks in 843 until the fall of the Second French Empire in 1870, with several interruptions. Between the period from King Charles the Bald in 843 to King Louis XVI in 1792, France had 45 kings. Adding the 7 emperors and kings after the French Revolution, this comes to a total of 52 monarchs of France.

In August 843 the Treaty of Verdun divided the Frankish realm into three kingdoms, one of which (Middle Francia) was short-lived; the other two evolved into France (West Francia) and, eventually, Germany (East Francia). By this time, the eastern and western parts of the land had already developed different languages and cultures.

Initially, the kingdom was ruled primarily by two dynasties, the Carolingians and the Robertians, who alternated rule from 843 until 987, when Hugh Capet, the progenitor of the Capetian dynasty, took the throne. The kings use the title "King of the Franks" until the late twelfth century; the first to adopt the title of "King of France" was Philip II (r. 1180–1223). The Capetians ruled continuously from 987 to 1792 and again from 1814 to 1848. The branches of the dynasty which ruled after 1328, however, are generally given the specific branch names of Valois (until 1589), Bourbon (from 1589 until 1792 and from 1814 until 1830), and the Orléans (from 1830 until 1848).

In the cell above, we used two libraries, `requests` and `BeautifulSoup` to call the Wikipedia server that hosts that particular page. We then searched for the "div" tag that contains the main body of the page. In this case, it was a "div" tag whose attribute "id" corresponded to "mw-content-text". I then searched for all of the "p" tags, or paragraphs within that body and printed off the the text if the text was not blank. By the end of the next section, you will

not only understand the code above, but you will be able to write it and code it out yourself. For now, inspect that page and see if you can find where the “div” tag whose id corresponds to “mw-content-text” is located in the HTML. It is okay if this is hard! It is not something that naturally you can do quickly. It takes practice. A trick to help get you started, however, is to right click the area that you want to scrape and then click inspect.

Once you feel comfortable with this, feel free to move on to the next section to start learning how to web scrape!

6.2 Scraping Web Pages with Requests and BeautifulSoup

6.2.1 Introduction

In the last section, we learned about the basics of HTML and the BeautifulSoup library. We were not, however, working with data found on the web, rather our HTML was stored locally. In this section, we will learn how to make calls to a remote server with the requests library and then parse that data with BeautifulSoup.

6.2.2 Requests

The **requests** library allows us to send a signal via Python to server. A good way to think about requests is as an invisible browser that opens in the background of your computer. Requests does the precise thing your browser does. It sends a signal over the internet to connect to a specific server address. While all servers have a unique IP address. Often the internet links a specific and unique address that can be used as a way to connect to a server without having to type out an IP address. Unlike your browser, however, requests does not will up the results for you to see. Instead, it receives the return signal and simply stores the HTML data in memory.

To begin learning how requests works, let’s first import the requests library:

```
import requests
```

Now that we have imported requests, let’s go ahead and create a string object that will be the website we want to scrape. I always call this string “url” in my code.

```
url = "https://en.wikipedia.org/wiki/List_of_French_monarchs"
```

Excellent! Now we can use the requests library to make a call to this particular page. We will do this via the get function in the requests library. The get function has one mandatory argument: the website that you want to request. In our case, this will be our “url” string.

```
s = requests.get(url)
```

Now that we have created a request object, let’s take a look at what this looks like.

```
print (s)
```

```
<Response [200]>
```

On the surface, this looks like we may have failed. What is this odd “Response [200]” and what does it mean? This particular response means that our attempt to connect to a server was successful. There are many types of responses, but [200] is the one we want to see. If you ever see a response that is not [200], you can Google the particular server response

and you will find out what is happening. Sometimes, a response indicates that your request was blocked. This may be because the website has anti-web scraping measures in place. In other cases, the page may be protected, meaning that it lies behind a login. There are too many potential errors that may surface that I cannot detail them all in this basic introduction section. I will, however, give you a solution to a very common problem that can allow you to get around a common 403 response. For that solution, check out the final section of this chapter.

6.2.3 BeautifulSoup

Now that we have learned how to make a call to a server and stored the response (the HTML) in memory, we need a way to parse that data. Buried within the “s” request object is the HTML content. We can access that data by accessing the content method of the response object class. Let’s do that and check out the first 100 characters.

```
print (s.content[:100])
```

```
b'<!DOCTYPE html>\n<html class="client-nojs" lang="en" dir="ltr">\n<head>\n<meta_\n→charset="UTF-8"/>\n<title'
```

Notice that this data is HTML. At this stage, however, we do not have an easy way to take this string and process it as structured data. This is where BeautifulSoup comes into play. BeautifulSoup allows us to convert s.content into structured data that we can then parse. To do this, we first need to import BeautifulSoup. Unlike most libraries, BeautifulSoup installs as bs4 (BeautifulSoup4). Because of this we need to import the BeautifulSoup class from bs4. The command below does this for us.

```
from bs4 import BeautifulSoup
```

Next, we need to create a new soup object.

```
soup = BeautifulSoup(s.content)
```

If we don’t see an error, then it means we have successfully created a soup object. Let’s print it off to see what it looks like:

```
print (str(soup)[:200])
```

```
<!DOCTYPE html>
<html class="client-nojs" dir="ltr" lang="en">
<head>
<meta charset="utf-8"/>
<title>List of French monarchs - Wikipedia</title>
<script>document.documentElement.className="client-js";
```

While the soup object looks precisely the same as the s.content, it is entirely different. It retains the structure of the HTML because BeautifulSoup has parsed it for us. This means that we can use special methods. In this section of the textbook, we will use `find` and `find_all`.

- `find` – this will allow us to find the first instance of a tag being used and grab that tag and all its nested components.
- `find_all` – this will return a list of all tags and their nested components that align to this specific tag.

Let's take a look at a basic example of the `find` method.

```
div = soup.find("div")
```

Here, we grabbed the first “div” tag on the page. “div” tags, however, are quite common because they are one of the essential building blocks of HTML. Let’s take a look at how many there are on the entire page. We can do this with the `find_all` method and then count the list size with the `len` function.

```
all_divs = soup.find_all("div")
print (len(all_divs))
```

97

So, if we want to grab a specific div with only `find` and `find_all`, we would have to count all the “div” tags and find the right index and grab it. This would not work at scale because this would vary from page to page, even if the HTML data structure were similar across all pages on a site. We need a better solution. This is where our second optional argument comes in. The `find` function can also take a dictionary that allows us to pass some specificity to our parsing of the `soup` object.

Let’s say I want to grab the main body of the Wikipedia article. If I inspect the page, I will notice that one particular “div” tag contains all the data corresponding to the body of the Wikipedia article. This “div” tag has a special unique id attribute. This id attribute’s name is “`mw-content-text`”. This means that I can pass as a second argument a dictionary where `id` is the key and the corresponding id name is the value. This will tell BeautifulSoup to find the first “div” tag that has an `id` attribute that matches `mw-content-text`. Let’s take a look at this in code.

```
body = soup.find("div", {"id": "mw-content-text"})
```

Now that we have grabbed this body portion of the article, we can print it off with the `text` method.

```
print (body.text[:500])
```

This article is about French monarchs. For Frankish kings, see List of Frankish_kings.

Division of the Frankish Empire at the Treaty of Verdun in 843
The monarchs of the Kingdom of France ruled from the establishment of the Kingdom_of the West Franks in 843 until the fall of the Second French Empire in 1870,_
with several interruptions. Between the period from King Charles the Bald in
843 to King Louis XVI in 1792, France had 45 kings. Adding the 7 emperors and_
kings after the French Revolut

This is fantastic! But, what if we wanted to grab the text and maintain the structure of the paragraphs. We can do this by searching the `soup` object at the body level. The `body` object that we created is still a `soup` object that retains all that HTML data, but it only contains the data for the data found under that particular div. We can use `find_all` now to grab all the paragraphs from within the body. We can use `find_all` on the `body` object to find all the paragraphs like so:

```
paragraphs = body.find_all("p")
```

We can now iterate over the paragraphs. Let's do that now over the first five paragraphs and print off the text.

```
for paragraph in paragraphs[:5]:  
    print (paragraph.text)
```

The monarchs of the Kingdom of France ruled from the establishment of the Kingdom of the West Franks in 843 until the fall of the Second French Empire in 1870, with several interruptions. Between the period from King Charles the Bald in 843 to King Louis XVI in 1792, France had 45 kings. Adding the 7 emperors and kings after the French Revolution, this comes to a total of 52 monarchs of France.

In August 843 the Treaty of Verdun divided the Frankish realm into three kingdoms, one of which (Middle Francia) was short-lived; the other two evolved into France (West Francia) and, eventually, Germany (East Francia). By this time, the eastern and western parts of the land had already developed different languages and cultures.

Initially, the kingdom was ruled primarily by two dynasties, the Carolingians and the Robertians, who alternated rule from 843 until 987, when Hugh Capet, the progenitor of the Capetian dynasty, took the throne. The kings use the title "King of the Franks" until the late twelfth century; the first to adopt the title of "King of France" was Philip II (r. 1180–1223). The Capetians ruled continuously from 987 to 1792 and again from 1814 to 1848. The branches of the dynasty which ruled after 1328, however, are generally given the specific branch names of Valois (until 1589), Bourbon (from 1589 until 1792 and from 1814 until 1830), and the Orléans (from 1830 until 1848).

Viola! You have now officially webscraped your first page in Python and grabbed some relevant data. Scraping data from the web is never a copy-and-paste task because every website structures its HTML a bit differently. Nevertheless, the mechanics are the same. The basic methods you learned in this chapter should allow you to scrape the majority of static websites.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

Part II

Data Analysis with Pandas



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

7

Introduction to Pandas

In this chapter, we will begin learning about Pandas, a powerful Python library used for working with tabular data. Throughout this chapter, you will learn about the basics of Pandas, why its useful, how to install it, and how to use it to perform basic tasks.

7.1 Introduction to Pandas

7.1.1 What Is Pandas?

Pandas is a Python library in Python that allows for you to easily work with tabular data that is often stored in Excel files or .csv files. It is often considered one of the most vital Python libraries for data analysis because of its robust capabilities. There are other Python libraries that allow you to work with Excel data, such as XLSX and XLRD, which are leveraged by Pandas on the back end to interact with Excel spreadsheets. However, most tabular data is stored in .csv files, or Comma Separated Values files. There are other variations of this same structure (such as .tsv), but they all work the same. They use a special character to denote change in the table to the next column and/or row.

7.1.2 Why Use Pandas?

If you work with data at all or plan to in the future, becoming comfortable with Pandas early on will make your life a lot easier. There are other Python libraries that allow you to work with .csv files, such as the CSV library, but these are not the same as Pandas. Pandas has one large advantage over the CSV library. It not only allows you to input .csv files into Python, it allows you to easily load them as DataFrames.

DataFrames are special data structures that contain not only the raw data in a table, but preserve the structure and hierarchy of that table. By loading .csv files as a DataFrame, Pandas not only allows you easy access to your data, but a powerful way to analyze it within a script. In addition to that, Pandas also has robust built-in features that we will explore throughout this part of the textbook.

Finally, many resources for data analysis are built on top of Pandas. Being able to understand tutorials and documentation of these resources, therefore, often requires a basic understanding of Pandas.

So, why use Pandas? Because it is the best library for importing and working with tabular data. It allows you to easily read files as DataFrames. And, it is a required library for most data analysis resources.

7.1.3 How to Install Pandas

Installing pandas is as easy as installing any other Python library. If you are working within a Jupyter notebook like this one, you can execute the following command within a cell:

```
!pip install pandas
```

In Jupyter notebooks the “!” indicates that you want to perform a command in the terminal. We then specify what command we want to run. In this case, `pip install`. Finally, we specify the library we want to install, `pandas`. If you are not working within a Jupyter notebook, you can do the same thing by opening up your terminal, such as command prompt on Windows, and executing the same command.

7.1.4 How to Import Pandas

Once you have installed Pandas, it is time to import it. It is Pythonic, or good Python practice, to use `import pandas as pd`. By importing a library as something, you give it that specific variable as a name. This has a few benefits. First, it makes Pandas easier to call in your script, because you can call the library with “pd” rather than “pandas”. Second, all Pandas tutorials and posts on Stackoverflow will use “pd”. This means that your script will conform to traditional convention.

```
import pandas as pd
```

After executing the above command, you will have successfully imported Pandas into your Python script. In the next notebook, we will start working with Pandas.

7.2 The Basics of Pandas

7.2.1 How to Create a DataFrame from a Dictionary

As we noted in the last section, we need to first import Pandas. To do that we will use `import pandas as pd`.

```
import pandas as pd
```

With Pandas now loaded correctly, we can begin to work with the library. Normally, you will create a Pandas DataFrame from a CSV or some external data file. We will see examples of that below. To begin, though, let’s start with the basics. Below we have a dictionary. A good way to think of this dictionary is as an Excel Spreadsheet.

Each key in the dictionary is a column and its value is a list which contains each cell in that column. We will see an example of a two-column dataset below, but for now let’s work with the single column dataset, “names”. In this column, we have a list of six names.

```
names_dict = {"names": [
    "Tom",
    "Mary",
    "Jeff",
    "Rose",
    "Stephanie",
    "Rodger"
]}
```

	A	B	C
1	names		
2	Tom		
3	Mary		
4	Jeff		
5	Rose		
6	Stephanie		
7	Rodger		

FIGURE 7.1

Example of an Excel Spreadsheet.

Normally in Python, we would work with this data as a dictionary. I could do something like the following to get the value of names:

```
print(names_dict["names"])
```

```
['Tom', 'Mary', 'Jeff', 'Rose', 'Stephanie', 'Rodger']
```

But this section is on Pandas and DataFrames. We want to do more! We want to work strictly with our data as a DataFrame. To do that, we can use the line of code below. The `pd.DataFrame()` class can take numerous arguments. We won't get into all of them right now. For now, understand that there is one essential argument that you must pass: the data that you wish to convert into a DataFrame. In our case, we will be converting the single-column dictionary into a DataFrame, so we pass that object as the only argument. We can see this in the code below.

```
df = pd.DataFrame(names_dict)
```

7.2.2 How to Display a DataFrame

Loading the data as DataFrame is not the end of our work. It is often times essential to view that data in a Jupyter notebook or terminal. To see what it looks like, you can use the following command.

```
df
```

```
      names
0      Tom
1     Mary
2     Jeff
3     Rose
4  Stephanie
5    Rodger
```

Note that we are not printing off the data with `print()`. This is because we are working within a Jupyter notebook. Were we working within an IDE, such as Atom, we would need to use the following command:

```
print(df)
```

	names
0	Tom
1	Mary
2	Jeff
3	Rose
4	Stephanie
5	Rodger

Notice, however, that the formatting of the data in the output is a bit different. When we print off a DataFrame, we do not see the nice formatting, such as the row highlighting and column header emboldening. It is for these reasons, that I recommend using the command “`df`” rather than `print(df)`.

7.2.3 How to Save DataFrame to CSV

Often times when you convert your data into a DataFrame, you will process it and then ultimately save it to disk. To do this, we have a few different options, such as CSV and JSON. We will meet this process with JSON a bit later. For now, let’s focus on one file type: CSV, or comma separated value. To save your DataFrame to a CSV file, you can write the following command

```
df.to_csv("../data/names.csv")
```

As we will see a little later, there are different arguments that you can pass here (and should!) For now, let’s focus on that single argument that we used: a string. This string should correspond to the file that you want to create. In this case, we are putting it into the data subfolder under the name “`names.csv`”.

7.2.4 How to Read DataFrame from CSV

Now that we have the data saved to a CSV file, let’s create a new DataFrame, `df2`, and read that data. We can do this with the command `pd.read_csv()`. As with `to_csv`, we can pass multiple arguments here, but for now, we will stick with the one mandatory one, a string of the file that we wish to open. In this case, it is the same file we just created. Let’s open it and print it off.

```
df2 = pd.read_csv("../data/names.csv")
```

```
df2
```

	Unnamed: 0	names
0	0	Tom
1	1	Mary
2	2	Jeff
3	3	Rose
4	4	Stephanie
5	5	Rodger

This doesn't look right. Notice that this DataFrame looks a bit off from what we saved to disk. Why is that? It is because of *how* we saved the file. If we don't specify an index, Pandas will automatically create one for us. In order to correctly save our file, we need to pass an extra keyword argument, specifically `index=False`. Let's try and save this file again under a different name: `names_no_index.csv`.

```
df.to_csv("../data/names_no_index.csv", index=False)
```

Let's create a new DataFrame, `df3`, and reopen and print off the data.

```
df3 = pd.read_csv("../data/names_no_index.csv")
```

```
df3
```

```
      names
0      Tom
1     Mary
2     Jeff
3    Rose
4  Stephanie
5   Rodger
```

Like magic, now we have a DataFrame that represents our original data.

7.2.5 How to Save DataFrame to JSON

In Pandas, we are not limited to CSV files, we can do this same process with JSON files, which are JavaScript Object Notation files. These are a bit different from CSV files and are used to store more complex data, specifically data that is used on the web because all browsers can interpret JSON data off-the-shelf. To save our data to a JSON file, we can use the `to_json()`. Note that we are not passing the `index` argument here. When our data is stored as a JSON file this is not necessary.

```
df3.to_json("../data/names.json")
```

Now, let's open that same data as a new DataFrame, `df4`. We can do the same thing as we did with the CSV file, except use `read_json()` and then print it off.

```
df4 = pd.read_json("../data/names.json")
```

```
df4
```

```
      names
0      Tom
1     Mary
2     Jeff
3    Rose
4  Stephanie
5   Rodger
```

7.2.6 How to Add a Column to the DataFrame

When working with DataFrames, you will almost always need to manipulate the data in some way. This means adding columns, deleting columns, performing permutations on data

in columns, etc. We are going to cover all these things throughout this textbook. For now, let's start with the basics. Imagine we got the names of individuals from one source and their ages from another. We now need to add those ages into our DataFrame. We can approach the DataFrame as something like a dictionary here. We can add a column by creating it with `df4["ages"]`. This allows us to make that equal to the new data. The command below essentially adds a column to our DataFrame. Let's execute the command and print it off.

```
df4["ages"] = [20, 26, 20, 18, 52, 40]
```

```
df4
```

	names	ages
0	Tom	20
1	Mary	26
2	Jeff	20
3	Rose	18
4	Stephanie	52
5	Rodger	40

Notice that we now have our second column. I want to stress right now, that to do this we needed data that matched the length of the names. If we tried to do this same act, but with five ages, rather than six, we would have received an error.

7.2.7 How to Grab a Specific Column

When working with a DataFrame, you will often need to grab a single column of data. To do that, we can navigate the column data rather like a Python Class. Let's create a new object, `names`, and grab just the names column by stating `df4.names`. This command tells Pandas to grab the "names" column. Note, this is case sensitive. After we grab it, let's print it off too.

```
names = df4.names
```

```
print(names)
```

0	Tom
1	Mary
2	Jeff
3	Rose
4	Stephanie
5	Rodger

Name: names, dtype: object

Notice that we have a lot of data here about our names. We have their index (left column of integers). At the bottom, we have the type of data that it is. Don't worry about this extra data at the bottom for now. Let's try and grab the ages column now.

```
ages = df4.ages
```

```
ages
```

0	20
1	26
2	20

(continues on next page)

(continued from previous page)

```
3    18
4    52
5    40
Name: ages, dtype: int64
```

7.2.8 How to Convert a Column to a List

When you are working with a DataFrame, you will often times need to work with that data not as a column, rather as a list. To do this, you will want to convert that data into a list. You can do this, by calling the method `.tolist()` after the data in question. Let's try it with `ages` and print off a list of ages.

```
ages_list = df4.ages.tolist()
```

```
print(ages_list)
```

```
[20, 26, 20, 18, 52, 40]
```

While this may not seem necessary on the surface, it is. One of the main reasons that this is essential is for processing large quantities of data. It is often times computationally less expensive to work with that data as a list or, better yet, as a NumPy array.

7.2.9 Isolating Unique Values in a Column

We can also isolate the unique values in a column with `.unique()`.

```
unique_ages = df4.ages.unique()
print(unique_ages)
```

```
[20 26 18 52 40]
```

7.2.10 How to Grab a Specific Row of a DataFrame with `iloc`

It will often times be necessary to grab a DataFrame by row, not column. We have a lot of different ways to grab a row, but for now let's imagine we want to just grab a specific row by index. (We can grab all rows that have a certain value in a certain column also, but we will see that a bit later.) To grab a row by index, you can use the `iloc` command. This stands for index location. Index location can be indexed rather like a list, as in the code below. The index you choose should correspond to the row index (starting with 0).

```
row1 = df4.iloc[1]
```

```
row1
```

```
names    Mary
ages     26
Name: 1, dtype: object
```

7.2.11 Iterating over a DataFrame with df.iterrows()

Often, we will need to iterate over an entire DataFrame. We can do that with `df.iterrows()` in a `for` loop. It is important to note that we will need two variables to do this, the index (`idx`) and the row (`row`). This will iterate down a DataFrame row by row.

```
for idx, row in df4.iterrows():
    print(row)
```

```
names      Tom
ages       20
Name: 0, dtype: object
names      Mary
ages       26
Name: 1, dtype: object
names     Jeff
ages       20
Name: 2, dtype: object
names     Rose
ages       18
Name: 3, dtype: object
names  Stephanie
ages       52
Name: 4, dtype: object
names    Rodger
ages       40
Name: 5, dtype: object
```

We can isolate the values of each row by indexing the row at each column.

```
for idx, row in df4.iterrows():
    print(row["names"], row["ages"])
```

```
Tom 20
Mary 26
Jeff 20
Rose 18
Stephanie 52
Rodger 40
```

This allows you to iterate over an entire DataFrame and isolate each row individually in a loop. As we will learn throughout this part of the textbook, we do not always need to iterate over each row to isolate specific data from a DataFrame.

7.2.12 Conclusion

You should now feel comfortable with creating, reading, and saving DataFrames. You should also be comfortable with adding columns and indexing by rows. In the following chapters, we will dive into more complex features of DataFrames.

8

Working with Data in Pandas

8.1 Finding Data in DataFrame

8.1.1 About the Titanic Dataset

```
import pandas as pd
```

Throughout this chapter, we will be using the same dataset, the infamous *Titanic Survivor* dataset that was put out by Kaggle. It has since become a staple dataset in machine learning and data science communities because of the breadth of data that it offers on the Titanic passengers. The goal of most machine learning challenges is to use this dataset to train a model that can accurately predict if a passenger would have survived given certain gender, economic, and class conditions.

In this section, we will be using a cleaned version of the dataset available on the GitHub page below. We will be using it because of its breadth of its qualitative and quantitative data.

In this notebook, we are strictly interested in finding key data in the DataFrame. By the end of the notebook, you should have a good understanding of why working with CSV data in Python via Pandas is far more powerful and easier than using Excel.

```
#data obtained from => https://github.com/datasciencedojo/datasets  
df = pd.read_csv("../data/titanic.csv")
```

8.1.2 How to Find Column Data

Let's presume that we just obtained this dataset. Let's also presume we know nothing about it. Our first job is to get a sense of the data. Maybe we want to know all the columns that the dataset contains. To find this information, we can use `df.columns`.

```
df.columns
```

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',  
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],  
      dtype='object')
```

With our knowledge from the last chapter, we now know how to convert this into a list!

```
cols = df.columns.tolist()
```

```
cols
```

```
[ 'PassengerId',
  'Survived',
  'Pclass',
  'Name',
  'Sex',
  'Age',
  'SibSp',
  'Parch',
  'Ticket',
  'Fare',
  'Cabin',
  'Embarked' ]
```

8.1.3 How to Get a Quick Sense of the Dataset with df.head()

Another way to get a quick sense of the data is to use `df.head()`. This allows us to return to top pieces of information from the dataset. By default, head returns the top five rows.

```
df.head()
```

	PassengerId	Survived	Pclass	\
0	1	0	3	
1	2	1	1	
2	3	1	3	
3	4	1	1	
4	5	0	3	

	Name	Sex	Age	SibSp	\
0	Braund, Mr. Owen Harris	male	22.0	1	
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	
2	Heikkinen, Miss. Laina	female	26.0	0	
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	
4	Allen, Mr. William Henry	male	35.0	0	

	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
1	0	PC 17599	71.2833	C85	C
2	0	STON/O2. 3101282	7.9250	NaN	S
3	0	113803	53.1000	C123	S
4	0	373450	8.0500	NaN	S

We can pass in an argument to return more than five. Let's try and display 20.

```
df.head(20)
```

	PassengerId	Survived	Pclass	\
0	1	0	3	
1	2	1	1	
2	3	1	3	
3	4	1	1	
4	5	0	3	
5	6	0	3	
6	7	0	1	
7	8	0	3	
8	9	1	3	

(continues on next page)

(continued from previous page)

9	10	1	2					
10	11	1	3					
11	12	1	1					
12	13	0	3					
13	14	0	3					
14	15	0	3					
15	16	1	2					
16	17	0	3					
17	18	1	2					
18	19	0	3					
19	20	1	3					
0		Name		Sex	Age	SibSp	\	
1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	Braund, Mr. Owen Harris Heikkinen, Miss. Laina		male	22.0	1		
2				female	38.0	1		
3	Futrelle, Mrs. Jacques Heath (Lily May Peel) Allen, Mr. William Henry			female	26.0	0		
4				male	35.0	1		
5		Moran, Mr. James		male	NaN	0		
6		McCarthy, Mr. Timothy J		male	54.0	0		
7		Palsson, Master. Gosta Leonard		male	2.0	3		
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)			female	27.0	0		
9	Nasser, Mrs. Nicholas (Adele Achem)			female	14.0	1		
10	Sandstrom, Miss. Marguerite Rut			female	4.0	1		
11	Bonnell, Miss. Elizabeth			female	58.0	0		
12	Saundercock, Mr. William Henry			male	20.0	0		
13	Andersson, Mr. Anders Johan			male	39.0	1		
14	Vestrom, Miss. Hulda Amanda Adolfina			female	14.0	0		
15	Hewlett, Mrs. (Mary D Kingcome)			female	55.0	0		
16	Rice, Master. Eugene			male	2.0	4		
17	Williams, Mr. Charles Eugene			male	NaN	0		
18	Vander Planke, Mrs. Julius (Emelia Maria Vande... Masselmani, Mrs. Fatima			female	31.0	1		
19				female	NaN	0		
0	Parch	Ticket	Fare	Cabin	Embarked			
1	0	A/5 21171	7.2500	NaN	S			
2	0	PC 17599	71.2833	C85	C			
3	0	STON/O2. 3101282	7.9250	NaN	S			
4	0	113803	53.1000	C123	S			
5	0	373450	8.0500	NaN	S			
6	0	330877	8.4583	NaN	Q			
7	0	17463	51.8625	E46	S			
8	1	349909	21.0750	NaN	S			
9	2	347742	11.1333	NaN	S			
10	0	237736	30.0708	NaN	C			
11	1	PP 9549	16.7000	G6	S			
12	0	113783	26.5500	C103	S			
13	0	A/5. 2151	8.0500	NaN	S			
14	5	347082	31.2750	NaN	S			
15	0	350406	7.8542	NaN	S			
16	0	248706	16.0000	NaN	S			
17	1	382652	29.1250	NaN	Q			
18	0	244373	13.0000	NaN	S			
19	0	345763	18.0000	NaN	S			
		2649	7.2250	NaN	C			

8.1.4 How to Grab a Specific Range of Rows with `df.iloc[]`

Sometimes, however, you need to grab a specific range of rows. Let's say I am interested in rows 5–20. To grab this data, we can use the `df.iloc[]` command that we met in the last notebook. This will allow us to pass a specific index range like a list.

```
df.iloc[5:20]
```

	PassengerId	Survived	Pclass	\
5	6	0	3	
6	7	0	1	
7	8	0	3	
8	9	1	3	
9	10	1	2	
10	11	1	3	
11	12	1	1	
12	13	0	3	
13	14	0	3	
14	15	0	3	
15	16	1	2	
16	17	0	3	
17	18	1	2	
18	19	0	3	
19	20	1	3	
				Name
5				Moran, Mr. James
6				McCarthy, Mr. Timothy J
7				Palsson, Master. Gosta Leonard
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)			
9				Nasser, Mrs. Nicholas (Adele Achem)
10				Sandstrom, Miss. Marguerite Rut
11				Bonnell, Miss. Elizabeth
12				Saundercock, Mr. William Henry
13				Andersson, Mr. Anders Johan
14				Vestrom, Miss. Hulda Amanda Adolfina
15				Hewlett, Mrs. (Mary D Kingcome)
16				Rice, Master. Eugene
17				Williams, Mr. Charles Eugene
18	Vander Planke, Mrs. Julius (Emelia Maria Vande... Masselmani, Mrs. Fatima			
19				
	Parch	Ticket	Fare	Cabin Embarked
5	0	330877	8.4583	NaN Q
6	0	17463	51.8625	E46 S
7	1	349909	21.0750	NaN S
8	2	347742	11.1333	NaN S
9	0	237736	30.0708	NaN C
10	1	PP 9549	16.7000	G6 S
11	0	113783	26.5500	C103 S
12	0	A/5. 2151	8.0500	NaN S
13	5	347082	31.2750	NaN S
14	0	350406	7.8542	NaN S
15	0	248706	16.0000	NaN S
16	1	382652	29.1250	NaN Q
17	0	244373	13.0000	NaN S
18	0	345763	18.0000	NaN S
19	0	2649	7.2250	NaN C

8.1.5 How to Get a Quick Quantitative Understanding of the Dataset with `describe()`

When working with numerical, quantitative data, we can automatically grab all numerical rows and learn a lot about the data with `df.describe()`. This will return the count, mean, standard deviation, minimum, maximum, etc. of our quantitative data. Sometimes, this data is useful to compute in such a way, such as the column “Survived”. Here, we can see that roughly “.38” or 38% of the passengers (in this dataset) survived. Other numerical data does not really lend itself well to this kind of analysis, e.g. “PassengerId” which has a unique numerical number corresponding to each passenger.

```
df.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	\
count	891.000000	891.000000	891.000000	714.000000	891.000000	
mean	446.000000	0.383838	2.308642	29.699118	0.523008	
std	257.353842	0.486592	0.836071	14.526497	1.102743	
min	1.000000	0.000000	1.000000	0.420000	0.000000	
25%	223.500000	0.000000	2.000000	20.125000	0.000000	
50%	446.000000	0.000000	3.000000	28.000000	0.000000	
75%	668.500000	1.000000	3.000000	38.000000	1.000000	
max	891.000000	1.000000	3.000000	80.000000	8.000000	
	Parch	Fare				
count	891.000000	891.000000				
mean	0.381594	32.204208				
std	0.806057	49.693429				
min	0.000000	0.000000				
25%	0.000000	7.910400				
50%	0.000000	14.454200				
75%	0.000000	31.000000				
max	6.000000	512.329200				

8.1.6 How to Find Specific Information in the Dataset with `df.loc`

As researchers, however, we often need to find targeted information in our dataset. Let's say I am researching female passengers on the Titanic. I can achieve this fairly easily in Excel and in Python with Pandas. To do this in Pandas, you can use `df.loc[]` to pass a specific argument. In the example below, we are stating that we are looking for the columns in the DataFrame of “Sex” that match the string “female”. Try to recreate this and find all males in the dataset on your own. Note at the bottom of the DataFrame 314×12. This is the dimensions of the DataFrame. We can see that we have 314 results.

```
df.loc[df["Sex"] == "female"]
```

	PassengerId	Survived	Pclass	\
1	2	1	1	
2	3	1	3	
3	4	1	1	
8	9	1	3	
9	10	1	2	
..	
880	881	1	2	
882	883	0	3	
885	886	0	3	
887	888	1	1	
888	889	0	3	

(continues on next page)

(continued from previous page)

		Name	Sex	Age	SibSp	\
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1		
2	Heikkinen, Miss. Laina	female	26.0	0		
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1		
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0	0		
9	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0	1		
..	
880	Shelley, Mrs. William (Imanita Parrish Hall)	female	25.0	0		
882	Dahlberg, Miss. Gerda Ulrika	female	22.0	0		
885	Rice, Mrs. William (Margaret Norton)	female	39.0	0		
887	Graham, Miss. Margaret Edith	female	19.0	0		
888	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1		
	Parch	Ticket	Fare	Cabin	Embarked	
1	0	PC 17599	71.2833	C85	C	
2	0	STON/O2. 3101282	7.9250	NaN	S	
3	0	113803	53.1000	C123	S	
8	2	347742	11.1333	NaN	S	
9	0	237736	30.0708	NaN	C	
..	
880	1	230433	26.0000	NaN	S	
882	0	7552	10.5167	NaN	S	
885	5	382652	29.1250	NaN	Q	
887	0	112053	30.0000	B42	S	
888	2	W./C. 6607	23.4500	NaN	S	

[314 rows x 12 columns]

But that's not all `df.loc[]` can do. We can add multiple arguments to our search query here, rather like SQL. In the example below, note the addition of the parentheses around `df["Sex"] == "female"` this denotes one parameter of our search. We then use the "&" to add a second parameter. In this case, we are interested in not only returning those that are female, but also those who are in "Pclass", or passenger class 1, i.e. First Class. When we execute the command below, we see that we have found 94 results.

```
df.loc[(df["Sex"] == "female") & (df["Pclass"] == 1)]
```

	PassengerId	Survived	Pclass	\	Name	Sex	Age	SibSp	\
1	2	1	1		Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	
3	4	1	1		Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	
11	12	1	1		Bonnell, Miss. Elizabeth	female	58.0	0	
31	32	1	1		Spencer, Mrs. William Augustus (Marie Eugenie)	female	NaN	1	
52	53	1	1		Harper, Mrs. Henry Sleeper (Myra Haxtun)	female	49.0	1	
..	
856	857	1	1						
862	863	1	1						
871	872	1	1						
879	880	1	1						
887	888	1	1						

(continues on next page)

(continued from previous page)

856		Wick, Mrs. George Dennick (Mary Hitchcock)	female	45.0	1
862		Swift, Mrs. Frederick Joel (Margaret Welles Ba...)	female	48.0	0
871		Beckwith, Mrs. Richard Leonard (Sallie Monypeny)	female	47.0	1
879		Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0	0
887		Graham, Miss. Margaret Edith	female	19.0	0
	Parch	Ticket	Fare	Cabin	Embarked
1	0	PC 17599	71.2833	C85	C
3	0	113803	53.1000	C123	S
11	0	113783	26.5500	C103	S
31	0	PC 17569	146.5208	B78	C
52	0	PC 17572	76.7292	D33	C
...
856	1	36928	164.8667	NaN	S
862	0	17466	25.9292	D17	S
871	1	11751	52.5542	D35	S
879	1	11767	83.1583	C50	C
887	0	112053	30.0000	B42	S

[94 rows x 12 columns]

Let's say now we are interested in only finding those that did not survive of the 94. To do that, we would simply add a third argument to our query. We ask Pandas to only show those whose "Survived" value is 0, or False. In other words, we are seeking to find those who did not survive.

```
df.loc[(df["Sex"] == "female") & (df["Pclass"] == 1) & (df["Survived"] == 0)]
```

	PassengerId	Survived	Pclass	\
177	178	0	1	
297	298	0	1	
498	499	0	1	
			Name	Sex
177		Isham, Miss. Ann Elizabeth	female	Age
297		Allison, Miss. Helen Loraine	female	SibSp
498	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)		female	
	Parch	Ticket	Fare	Cabin Embarked
177	0	PC 17595	28.7125	C49 C
297	2	113781	151.5500	C22 C26 S
498	2	113781	151.5500	C22 C26 S

With this data, I not only have qualitative information about each passenger, I also have some significant quantitative information as well. I know that of the 314 people in the dataset with the sex of female, 94 were in first class and of those 94, all survived except 3. Let's now pose a question and I think you may already know the answer. Did class in any way play a role in the chance off survival on the Titanic. We don't need a fancy machine learning algorithm to help us answer this question. We can simply examine the results from our query. Let's find the total number of passengers that are identified as female, but not in first class, and then find the number of those who did not survive.

```
df.loc[(df["Sex"] == "female") & (df["Pclass"] > 1)]
```

	PassengerId	Survived	Pclass	\
2	3	1	3	
8	9	1	3	
9	10	1	2	
10	11	1	3	
14	15	0	3	
..	
875	876	1	3	
880	881	1	2	
882	883	0	3	
885	886	0	3	
888	889	0	3	
				Name
2				Heikkinen, Miss. Laina
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)			female 26.0 0
9	Nasser, Mrs. Nicholas (Adele Achem)			female 27.0 0
10	Sandstrom, Miss. Marguerite Rut			female 14.0 1
14	Vestrom, Miss. Hulda Amanda Adolfina			female 4.0 1
..				...
875	Najib, Miss. Adele Kiamie "Jane"			female 15.0 0
880	Shelley, Mrs. William (Imanita Parrish Hall)			female 25.0 0
882	Dahlberg, Miss. Gerda Ulrika			female 22.0 0
885	Rice, Mrs. William (Margaret Norton)			female 39.0 0
888	Johnston, Miss. Catherine Helen "Carrie"			female NaN 1
	Parch	Ticket	Fare	Cabin Embarked
2	0	STON/O2. 3101282	7.9250	NaN S
8	2	347742	11.1333	NaN S
9	0	237736	30.0708	NaN C
10	1	PP 9549	16.7000	G6 S
14	0	350406	7.8542	NaN S
..
875	0	2667	7.2250	NaN C
880	1	230433	26.0000	NaN S
882	0	7552	10.5167	NaN S
885	5	382652	29.1250	NaN Q
888	2	W./C. 6607	23.4500	NaN S

[220 rows x 12 columns]

```
df.loc[(df["Sex"] == "female") & (df["Pclass"] > 1) & (df["Survived"] == 0)]
```

	PassengerId	Survived	Pclass	\
14	15	0	3	
18	19	0	3	
24	25	0	3	
38	39	0	3	
40	41	0	3	
..	
854	855	0	2	
863	864	0	3	
882	883	0	3	
885	886	0	3	
888	889	0	3	
				Name
14	Vestrom, Miss. Hulda Amanda Adolfina			female 14.0 0

(continues on next page)

(continued from previous page)

18	Vander Planke, Mrs. Julius (Emelia Maria Vande...	female	31.0	1
24	Palsson, Miss. Torborg Danira	female	8.0	3
38	Vander Planke, Miss. Augusta Maria	female	18.0	2
40	Ahlin, Mrs. Johan (Johanna Persdotter Larsson)	female	40.0	1
..
854	Carter, Mrs. Ernest Courtenay (Lilian Hughes)	female	44.0	1
863	Sage, Miss. Dorothy Edith "Dolly"	female	NaN	8
882	Dahlberg, Miss. Gerda Ulrika	female	22.0	0
885	Rice, Mrs. William (Margaret Norton)	female	39.0	0
888	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1
Parch	Ticket	Fare	Cabin	Embarked
14	0	350406	7.8542	NaN
18	0	345763	18.0000	NaN
24	1	349909	21.0750	NaN
38	0	345764	18.0000	NaN
40	0	7546	9.4750	NaN
..
854	0	244252	26.0000	NaN
863	2	CA. 2343	69.5500	NaN
882	0	7552	10.5167	NaN
885	5	382652	29.1250	NaN
888	2	W./C. 6607	23.4500	NaN

[78 rows x 12 columns]

As we can see, we have 220 total people identified as female with 78 not surviving. We can perform some quick calculations to understand better our results.

```
x = 3/94
x
```

```
0.031914893617021274
```

```
y = 78/220
y
```

```
0.35454545454545455
```

In this case, `x` is the death rate for a first class female, whereas `y` is the death rate for those not in first class and female. The result is clear. The chance for survival was significantly greater for those identified as female if they were in first class, roughly 32.5% difference. Were there other factors at play, possibly. But that is beyond the point here. I am simply trying to illustrate how to obtain data to begin framing the research question at hand.

Although we can achieve this same analysis in Excel, doing this in Python allows us to leverage the power of an entire programming language with the data at hand. This is a simple example of the power of Pandas over Excel. As we move forward, you will see more examples of this.

8.1.7 How to Query with “OR” (`|`) on a DataFrame

Above we saw how to use `df.loc` to structure logical arguments for finding two specific conditions with “`&`”. This indicated that both conditions must be true to be returned as a result. In case you don’t remember it, it looked like this:

```
df.loc[(df["Sex"] == "female") & (df["Pclass"] > 1)]
```

	PassengerId	Survived	Pclass	\		
2		3	1	3		
8		9	1	3		
9		10	1	2		
10		11	1	3		
14		15	0	3		
..		
875	876		1	3		
880	881		1	2		
882	883		0	3		
885	886		0	3		
888	889		0	3		
		Name	Sex	Age	SibSp	\
2		Heikkinen, Miss. Laina	female	26.0	0	
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)		female	27.0	0	
9	Nasser, Mrs. Nicholas (Adele Achem)		female	14.0	1	
10	Sandstrom, Miss. Marguerite Rut		female	4.0	1	
14	Vestrom, Miss. Hulda Amanda Adolfina		female	14.0	0	
..	
875	Najib, Miss. Adele Kiamie "Jane"		female	15.0	0	
880	Shelley, Mrs. William (Imanita Parrish Hall)		female	25.0	0	
882	Dahlberg, Miss. Gerda Ulrika		female	22.0	0	
885	Rice, Mrs. William (Margaret Norton)		female	39.0	0	
888	Johnston, Miss. Catherine Helen "Carrie"		female	NaN	1	
	Parch	Ticket	Fare	Cabin	Embarked	
2	0	STON/O2. 3101282	7.9250	NaN	S	
8	2	347742	11.1333	NaN	S	
9	0	237736	30.0708	NaN	C	
10	1	PP 9549	16.7000	G6	S	
14	0	350406	7.8542	NaN	S	
..
875	0	2667	7.2250	NaN	C	
880	1	230433	26.0000	NaN	S	
882	0	7552	10.5167	NaN	S	
885	5	382652	29.1250	NaN	Q	
888	2	W./C. 6607	23.4500	NaN	S	

[220 rows x 12 columns]

We can do the same conditional approach, but rather than using and, we can also tell Pandas “or” with the |; this is known as a **pipe**. Let’s try and grab all first and second class passengers. We can specify if the “Pclass” is either 1 OR 2.

```
df.loc[(df["Pclass"] == 1) | (df["Pclass"] == 2)]
```

	PassengerId	Survived	Pclass	\
1		2	1	1
3		4	1	1
6		7	0	1
9		10	1	2
11		12	1	1
..
880	881		1	2

(continues on next page)

(continued from previous page)

					Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
883	884	0	2										
886	887	0	2										
887	888	1	1										
889	890	1	1										
1	Cumings, Mrs. John Bradley (Florence Briggs Th... 3					female	38.0						1
6	Futrelle, Mrs. Jacques Heath (Lily May Peel) 9				McCarthy, Mr. Timothy J Nasser, Mrs. Nicholas (Adele Achem)	female	35.0						1
11					Bonnell, Miss. Elizabeth	male	54.0						0
..						female	14.0						1
880	Shelley, Mrs. William (Imanita Parrish Hall) 883				Banfield, Mr. Frederick James	female	58.0						0
886					Montvila, Rev. Juozas	male	25.0						0
887					Graham, Miss. Margaret Edith	male	28.0						0
889					Behr, Mr. Karl Howell	female	27.0						0
						male	19.0						0
						male	26.0						0
1	Parch				Ticket	Fare	Cabin	Embarked					
3	0				PC 17599	71.2833	C85	C					
6	0				113803	53.1000	C123	S					
9	0				17463	51.8625	E46	S					
11	0				237736	30.0708	NaN	C					
..	..				113783	26.5500	C103	S					
880	1				230433	26.0000	NaN	S					
883	0	C.A./SOTON	34068		10.5000	NaN	NaN	S					
886	0				211536	13.0000	NaN	S					
887	0				112053	30.0000	B42	S					
889	0				111369	30.0000	C148	C					

By learning to use Pandas in this way and structuring Pandas-based queries, we can generate the data necessary to begin answering research questions about our data. Pandas, however, can do more, it can also let us organize our data.

8.2 Organizing the DataFrame

8.2.1 How to Sort Data By Single Column

```
import pandas as pd  
df = pd.read_csv("../data/titanic.csv")
```

Now that we've imported Pandas and created our DataFrame, let's see what it looks like again.

df

	PassengerId	Survived	Pclass	
0	1	0	3	
1	2	1	1	

(continues on next page)

(continued from previous page)

2	3	1	3								
3	4	1	1								
4	5	0	3								
..								
886	887	0	2								
887	888	1	1								
888	889	0	3								
889	890	1	1								
890	891	0	3								
0	Name									Sex	Age
1	Braund, Mr. Owen Harris									male	22.0
2	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina									female	38.0
3	Futrelle, Mrs. Jacques Heath (Lily May Peel) Allen, Mr. William Henry									female	26.0
4	Graham, Miss. Margaret Edith Johnston, Miss. Catherine Helen "Carrie" Behr, Mr. Karl Howell Dooley, Mr. Patrick									male	35.0
..
886	Montvila, Rev. Juozas Graham, Miss. Margaret Edith									male	27.0
887	Johnston, Miss. Catherine Helen "Carrie"									female	19.0
888	Behr, Mr. Karl Howell Dooley, Mr. Patrick									female	NaN
889	...									male	26.0
890	...									male	32.0
0	Parch	Ticket	Fare	Cabin	Embarked						
1	0	A/5 21171	7.2500	NaN	S						
2	0	PC 17599	71.2833	C85	C						
3	0	STON/O2. 3101282	7.9250	NaN	S						
4	0	113803	53.1000	C123	S						
..						
886	0	211536	13.0000	NaN	S						
887	0	112053	30.0000	B42	S						
888	2	W./C. 6607	23.4500	NaN	S						
889	0	111369	30.0000	C148	C						
890	0	370376	7.7500	NaN	Q						
[891 rows x 12 columns]											

In this scenario, I am interested in sorting the data (rather like Excel). Rather than using the sort feature in Excel, we can use the `df.sort_values()` method in Python. This will take one argument, specifically the column that you want to organize by. By default, this will be ascending. Let's do this by class. In other words, sort the DataFrame so that those who were in first class appear first and those in third class appear last. We will do this by passing the argument "Pclass", the column name corresponding to passenger class.

```
df.sort_values("Pclass")
```

445	PassengerId	Survived	Pclass	\
310	446	1	1	
309	311	1	1	
307	310	1	1	
306	308	1	1	
..	
379	307	1	1	
381	380	0	3	
382	381	1	3	

(continues on next page)

(continued from previous page)

382	383	0	3		Name	Sex	Age	SibSp	\
371	372	0	3		Dodge, Master.	Washington	male	4.0	0
890	891	0	3		Hays, Miss.	Margaret Bechstein	female	24.0	0
					Francatelli, Miss.	Laura Mabel	female	30.0	0
307	Penasco y Castellana, Mrs.	Victor de Satode (M...			female	17.0		1	
306	Fleming, Miss.	Margaret			female	NaN		0	
..					
379	Gustafsson, Mr.	Karl Gideon			male	19.0		0	
381	Nakid, Miss.	Maria ("Mary")			female	1.0		0	
382	Tikkanen, Mr.	Juho			male	32.0		0	
371	Wiklund, Mr.	Jakob Alfred			male	18.0		1	
890	Dooley, Mr.	Patrick			male	32.0		0	
	Parch	Ticket	Fare	Cabin	Embarked				
445	2	33638	81.8583	A34	S				
310	0	11767	83.1583	C54	C				
309	0	PC 17485	56.9292	E36	C				
307	0	PC 17758	108.9000	C65	C				
306	0	17421	110.8833	NaN	C				
..				
379	0	347069	7.7750	NaN	S				
381	2	2653	15.7417	NaN	C				
382	0	STON/O 2.	3101293	7.9250	NaN				
371	0	3101267	6.4958	NaN	S				
890	0	370376	7.7500	NaN	Q				
[891 rows x 12 columns]									

8.2.2 How to Reverse Sort Data by Single Column

As we can see, our data is now appearing as expected. We can pass additional keyword arguments to sort the data in the opposite direction, or descending by setting `ascending` parameter to `False`. See the example below.

```
df.sort_values("Pclass", ascending=False)
```

	PassengerId	Survived	Pclass	\
0	1	0	3	
511	512	0	3	
500	501	0	3	
501	502	0	3	
502	503	0	3	
..	
102	103	0	1	
710	711	1	1	
711	712	0	1	
712	713	1	1	
445	446	1	1	
				Name
0				Braund, Mr. Owen Harris
511				Webber, Mr. James
				Sex
				male
				22.0
				1
				male
				NaN
				0

(continues on next page)

(continued from previous page)

500		Calic, Mr. Petar	male	17.0	0
501		Canavan, Miss. Mary	female	21.0	0
502		O'Sullivan, Miss. Bridget Mary	female	NaN	0
..		
102		White, Mr. Richard Frasar	male	21.0	0
710	Mayne, Mlle. Berthe Antonine ("Mrs de Villiers")		female	24.0	0
711		Klager, Mr. Herman	male	NaN	0
712		Taylor, Mr. Elmer Zebley	male	48.0	1
445		Dodge, Master. Washington	male	4.0	0
	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
511	0	SOTON/OQ 3101316	8.0500	NaN	S
500	0		8.6625	NaN	S
501	0		7.7500	NaN	Q
502	0		7.6292	NaN	Q
..
102	1	35281	77.2875	D26	S
710	0	PC 17482	49.5042	C90	C
711	0	113028	26.5500	C124	S
712	0	19996	52.0000	C126	S
445	2	33638	81.8583	A34	S

[891 rows x 12 columns]

8.2.3 How to Sort Data by Multiple Columns

Again, we can see the power of Pandas over Excel by the simplicity of altering our command to include multiple columns. Let's say that we want to sort all the data by "Pclass", then we want that data organized again by sex, so that all male and female passengers appear in order. We can do this by passing the argument of what we want organized as a list. Note the order of the list as well. The columns that appear earlier in the list correspond to those that receive primacy in the ascending. In other words, we organize by passenger class first, then sex. In this case, the method head, is simply showing the top 100 rows.

```
df.sort_values(["Pclass", "Sex"]).head(100)
```

	PassengerId	Survived	Pclass	\
1	2	1	1	
3	4	1	1	
11	12	1	1	
31	32	1	1	
52	53	1	1	
..	
23	24	1	1	
27	28	0	1	
30	31	0	1	
34	35	0	1	
35	36	0	1	
				Name
1	Cumings, Mrs. John Bradley (Florence Briggs Th...			female
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)			female
11				female
31	Bonnell, Miss. Elizabeth			58.0
	Spencer, Mrs. William Augustus (Marie Eugenie)			NaN

(continues on next page)

(continued from previous page)

52		Harper, Mrs. Henry Sleeper (Myra Haxtun)	female	49.0	1
..		
23		Sloper, Mr. William Thompson	male	28.0	0
27		Fortune, Mr. Charles Alexander	male	19.0	3
30		Uruchurtu, Don. Manuel E	male	40.0	0
34		Meyer, Mr. Edgar Joseph	male	28.0	1
35		Holverson, Mr. Alexander Oskar	male	42.0	1
Parch	Ticket	Fare	Cabin	Embarked	
1	0 PC 17599	71.2833	C85	C	
3	0 113803	53.1000	C123	S	
11	0 113783	26.5500	C103	S	
31	0 PC 17569	146.5208	B78	C	
52	0 PC 17572	76.7292	D33	C	
..	
23	0 113788	35.5000	A6	S	
27	2 19950	263.0000	C23 C25 C27	S	
30	0 PC 17601	27.7208	NaN	C	
34	0 PC 17604	82.1708	NaN	C	
35	0 113789	52.0000	NaN	S	

[100 rows x 12 columns]

As with before, we can control how the data is sorted, either ascending or descending. If we set ascending to “False”, we organize all items in the list by this method. We can do this with the sample code below.

```
df.sort_values(["Pclass", "Sex"], ascending=False)
```

	PassengerId	Survived	Pclass	\
0	1	0	3	
4	5	0	3	
5	6	0	3	
7	8	0	3	
12	13	0	3	
..	
856	857	1	1	
862	863	1	1	
871	872	1	1	
879	880	1	1	
887	888	1	1	
		Name	Sex	Age
0		Braund, Mr. Owen Harris	male	22.0
4		Allen, Mr. William Henry	male	35.0
5		Moran, Mr. James	male	NaN
7		Palsson, Master. Gosta Leonard	male	2.0
12		Saundercock, Mr. William Henry	male	20.0
..	
856		Wick, Mrs. George Dennick (Mary Hitchcock)	female	45.0
862		Swift, Mrs. Frederick Joel (Margaret Welles Ba...	female	48.0
871		Beckwith, Mrs. Richard Leonard (Sallie Monypeny)	female	47.0
879		Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0
887		Graham, Miss. Margaret Edith	female	19.0
	Parch	Ticket	Fare	Cabin
0	0 A/5 21171	7.2500	NaN	S
4	0 373450	8.0500	NaN	S
5	0 330877	8.4583	NaN	Q

(continues on next page)

(continued from previous page)

7	1	349909	21.0750	NaN	S
12	0	A/5. 2151	8.0500	NaN	S
..
856	1	36928	164.8667	NaN	S
862	0	17466	25.9292	D17	S
871	1	11751	52.5542	D35	S
879	1	11767	83.1583	C50	C
887	0	112053	30.0000	B42	S

[891 rows x 12 columns]

8.2.4 How to Sort Data by Multiple Columns with Different Values Organized Differently

What if we want to organize the data differently. By this I mean, we want for all the data to be organized by passenger class first and for that data to be ascending (1, 2, 3), but we want the sex of the passengers to be organized descending (male, female, rather than female, male). To achieve this, we can pass a list to ascending with 0s and 1s. 0 is “False” and 1 is “True”.

```
df.sort_values(["Pclass", "Sex"], ascending=[1, 0])
```

	PassengerId	Survived	Pclass	Name \				
6	7	0	1	McCarthy, Mr. Timothy J				
23	24	1	1	Sloper, Mr. William Thompson				
27	28	0	1	Fortune, Mr. Charles Alexander				
30	31	0	1	Uruchurtu, Don. Manuel E				
34	35	0	1	Meyer, Mr. Edgar Joseph				
..
863	864	0	3	Sage, Miss. Dorothy Edith "Dolly"				
875	876	1	3	Najib, Miss. Adele Kiamie "Jane"				
882	883	0	3	Dahlberg, Miss. Gerda Ulrika				
885	886	0	3	Rice, Mrs. William (Margaret Norton)				
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"				
	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
6	male	54.0	0	0	17463	51.8625	E46	S
23	male	28.0	0	0	113788	35.5000	A6	S
27	male	19.0	3	2	19950	263.0000	C23 C25 C27	S
30	male	40.0	0	0	PC 17601	27.7208	NaN	C
34	male	28.0	1	0	PC 17604	82.1708	NaN	C
..
863	female	NaN	8	2	CA. 2343	69.5500	NaN	S
875	female	15.0	0	0	2667	7.2250	NaN	C
882	female	22.0	0	0	7552	10.5167	NaN	S
885	female	39.0	0	5	382652	29.1250	NaN	Q
888	female	NaN	1	2	W./C. 6607	23.4500	NaN	S

[891 rows x 12 columns]

What is particularly nice about Pandas over Excel is that this operation scales nicely. If we want to add more methods of sorting, we can do that too by simple increasing the indices of our lists. Always make sure that the length of your lists match, however. In other words, do not have 3 attributes to sort by and 2 items in your ascending list. In this case, we want to organize by passenger class, sex, and age with passenger class ascending, sex descending, and age ascending. Let’s see what that would look like.

```
df.sort_values(["Pclass", "Sex", "Age"], ascending=[1, 0, 1])
```

	PassengerId	Survived	Pclass	\				
305	306	1	1					
445	446	1	1					
802	803	1	1					
550	551	1	1					
505	506	0	1					
..					
697	698	1	3					
727	728	1	3					
792	793	0	3					
863	864	0	3					
888	889	0	3					
			Name	Sex	Age	SibSp	Parch	\
305		Allison, Master.	Hudson Trevor	male	0.92	1	2	
445		Dodge, Master.	Washington	male	4.00	0	2	
802		Carter, Master.	William Thornton II	male	11.00	1	2	
550		Thayer, Mr.	John Borland Jr	male	17.00	0	2	
505		Penasco y Castellana,	Mr. Victor de Satode	male	18.00	1	0	
..			
697		Mullens, Miss.	Katherine "Katie"	female	NaN	0	0	
727			Mannion, Miss.	Margareth	female	NaN	0	0
792			Sage, Miss.	Stella Anna	female	NaN	8	2
863			Sage, Miss.	Dorothy Edith "Dolly"	female	NaN	8	2
888			Johnston, Miss.	Catherine Helen "Carrie"	female	NaN	1	2
	Ticket	Fare	Cabin	Embarked				
305	113781	151.5500	C22 C26	S				
445	33638	81.8583	A34	S				
802	113760	120.0000	B96 B98	S				
550	17421	110.8833	C70	C				
505	PC 17758	108.9000	C65	C				
..				
697	35852	7.7333	NaN	Q				
727	36866	7.7375	NaN	Q				
792	CA. 2343	69.5500	NaN	S				
863	CA. 2343	69.5500	NaN	S				
888	W./C. 6607	23.4500	NaN	S				

[891 rows x 12 columns]

As we move forward, we will explore more robust ways to sort and organize our data. For now, you should feel comfortable with how to use `sort_values()` to do fairly robust tasks quickly.

8.3 Cleaning the DataFrame

8.3.1 How to Drop a Column in Pandas DataFrame

```
import pandas as pd
df = pd.read_csv("../data/titanic.csv")
df
```

	PassengerId	Survived	Pclass	\		
0		1	0	3		
1		2	1	1		
2		3	1	3		
3		4	1	1		
4		5	0	3		
..		
886	887	887	0	2		
887	888	888	1	1		
888	889	889	0	3		
889	890	890	1	1		
890	891	891	0	3		
		Name	Sex	Age	SibSp	\
0		Braund, Mr. Owen Harris	male	22.0	1	
1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina		female	38.0	1	
2			female	26.0	0	
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)		female	35.0	1	
4		Allen, Mr. William Henry	male	35.0	0	
..
886		Montvila, Rev. Juozas	male	27.0	0	
887		Graham, Miss. Margaret Edith	female	19.0	0	
888	Johnston, Miss. Catherine Helen "Carrie"		female	NaN	1	
889		Behr, Mr. Karl Howell	male	26.0	0	
890		Dooley, Mr. Patrick	male	32.0	0	
	Parch	Ticket	Fare	Cabin	Embarked	
0	0	A/5 21171	7.2500	NaN	S	
1	0	PC 17599	71.2833	C85	C	
2	0	STON/O2. 3101282	7.9250	NaN	S	
3	0	113803	53.1000	C123	S	
4	0	373450	8.0500	NaN	S	
..	
886	0	211536	13.0000	NaN	S	
887	0	112053	30.0000	B42	S	
888	2	W./C. 6607	23.4500	NaN	S	
889	0	111369	30.0000	C148	C	
890	0	370376	7.7500	NaN	Q	

[891 rows x 12 columns]

Imagine that we have a large DataFrame, but we are not interested in a couple columns. This is especially import when your DataFrame has 10s or 100s of columns. In these instances, you need to examine the DataFrame without the useless data. Imagine that we wanted to study the Titanic data but knew that “Parch” and “Ticket” were categories that we did not need. We can use `df.drop()` to pass an argument to remove those specific columns.

```
df.drop(columns=["Parch", "Ticket"])
```

	PassengerId	Survived	Pclass	\
0		1	0	3
1		2	1	1
2		3	1	3
3		4	1	1
4		5	0	3
..

(continues on next page)

(continued from previous page)

886	887	0	2						
887	888	1	1						
888	889	0	3						
889	890	1	1						
890	891	0	3						
0				Name	Sex	Age	SibSp	\	
1	Cumings, Mrs. John Bradley (Florence Briggs Th...			Braund, Mr. Owen Harris	male	22.0	1		
2				Heikkinen, Miss. Laina	female	38.0	1		
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)				female	26.0	0		
4				Allen, Mr. William Henry	female	35.0	1		
..				
886				Montvila, Rev. Juozas	male	27.0	0		
887				Graham, Miss. Margaret Edith	female	19.0	0		
888				Johnston, Miss. Catherine Helen "Carrie"	female	Nan	1		
889				Behr, Mr. Karl Howell	male	26.0	0		
890				Dooley, Mr. Patrick	male	32.0	0		
0		Fare	Cabin	Embarked					
1	7.2500		NaN	S					
2	71.2833		C85	C					
3	7.9250		NaN	S					
4	53.1000		C123	S					
..					
886	13.0000		NaN	S					
887	30.0000		B42	S					
888	23.4500		NaN	S					
889	30.0000		C148	C					
890	7.7500		NaN	Q					
 [891 rows x 10 columns]									

Likewise, we can do the opposite. Rather than dropping certain columns, we can keep certain columns with the example code below.

```
df[["Survived", "Name"]]
```

	Survived			Name					
0	0			Braund, Mr. Owen Harris					
1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...							
2	1			Heikkinen, Miss. Laina					
3	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)							
4	0			Allen, Mr. William Henry					
..				
886	0			Montvila, Rev. Juozas					
887	1			Graham, Miss. Margaret Edith					
888	0			Johnston, Miss. Catherine Helen "Carrie"					
889	1			Behr, Mr. Karl Howell					
890	0			Dooley, Mr. Patrick					
 [891 rows x 2 columns]									

Note the use of double brackets here, [[]].

8.3.2 How to Remove Rows That Have NaN in Any Column

One of the biggest problems in datasets is the absence of data. If you are training a machine learning model or just performing quantitative analysis, rows that have missing values, or NaN, can radically alter your results. It is often good practice to ignore that data or alter it in some way. Let's presume that we want to simply remove it from our dataset. To do that, we can use `df.dropna()` which will remove all rows that have any instance of NaN in any column.

```
df.dropna()
```

	PassengerId	Survived	Pclass	\	Name	Sex	Age	SibSp	\
1	2	1	1		Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	
3	4	1	1		Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	
6	7	0	1		McCarthy, Mr. Timothy J	male	54.0	0	
10	11	1	3		Sandstrom, Miss. Marguerite Rut	female	4.0	1	
11	12	1	1		Bonnell, Miss. Elizabeth	female	58.0	0	
..	
871	872	1	1		Beckwith, Mrs. Richard Leonard (Sallie Monypeny)	female	47.0	1	
872	873	0	1		Carlsson, Mr. Frans Olof	male	33.0	0	
879	880	1	1		Potter, Mrs. Thomas Jr (Lily Alexenia Wilson)	female	56.0	0	
887	888	1	1		Graham, Miss. Margaret Edith	female	19.0	0	
889	890	1	1		Behr, Mr. Karl Howell	male	26.0	0	
	Parch	Ticket	Fare	Cabin	Embarked				
1	0	PC 17599	71.2833	C85	C				
3	0	113803	53.1000	C123	S				
6	0	17463	51.8625	E46	S				
10	1	PP 9549	16.7000	G6	S				
11	0	113783	26.5500	C103	S				
..	
871	1	11751	52.5542	D35	S				
872	0	695	5.0000	B51 B53 B55	S				
879	1	11767	83.1583	C50	C				
887	0	112053	30.0000	B42	S				
889	0	111369	30.0000	C148	C				
[183 rows x 12 columns]									

8.3.3 How to Remove Rows That Have NaN in a Specific Column

In some instances, though, we don't want to remove an entire row just because of NaN in one column. Maybe that column is not as important for quantitative analysis and we are not planning to include it in our analysis, but we still want to see it. A good example of

this is the column “Cabin” which is a string or “Age” which is a float (we’ll get to that in a moment). Let’s say we want to remove all rows that have NaN in the “Age” column, we can use the command below.

```
df2 = df[df["Age"].notna()]
```

```
df2
```

	PassengerId	Survived	Pclass	\			
0		1	0	3			
1		2	1	1			
2		3	1	3			
3		4	1	1			
4		5	0	3			
..			
885	886	886	0	3			
886	887	887	0	2			
887	888	888	1	1			
889	890	890	1	1			
890	891	891	0	3			
			Name	Sex	Age	SibSp	\
0			Braund, Mr. Owen Harris	male	22.0	1	
1			Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	female	38.0	1	
2			Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	26.0	0	
3			Allen, Mr. William Henry	male	35.0	1	
4							
..			
885			Rice, Mrs. William (Margaret Norton)	female	39.0	0	
886			Montvila, Rev. Juozas	male	27.0	0	
887			Graham, Miss. Margaret Edith	female	19.0	0	
889			Behr, Mr. Karl Howell	male	26.0	0	
890			Dooley, Mr. Patrick	male	32.0	0	
			Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	Nan		S	
1	0	PC 17599	71.2833	C85		C	
2	0	STON/O2. 3101282	7.9250	Nan		S	
3	0		113803	53.1000	C123		S
4	0		373450	8.0500	Nan		S
..
885	5	382652	29.1250	Nan		Q	
886	0	211536	13.0000	Nan		S	
887	0	112053	30.0000	B42		S	
889	0	111369	30.0000	C148		C	
890	0	370376	7.7500	Nan		Q	

[714 rows x 12 columns]

As we can see, the size of our DataFrame dropped from 891 rows to 714.

8.3.4 How to Convert DataFrame Data Types (from Float to Int)

In other instances, it may be important not to simply remove a column, but alter it into a different type of data. In this dataset, “Age” is a float. This is to account for infants who were below the age of one on the Titanic. Let’s presume that we want to convert all these floats to integers. To do that we can use the `.astype()` method on a specific row.

```
df2.Age = df2.Age.astype(int)
```

```
/tmp/ipykernel_708450/3009824114.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
→stable/user_guide/indexing.html#returning-a-view-versus-a-copy
df2.Age = df2.Age.astype(int)
```

```
df2
```

	PassengerId	Survived	Pclass	\			
0		1	0	3			
1		2	1	1			
2		3	1	3			
3		4	1	1			
4		5	0	3			
..			
885	886		0	3			
886	887		0	2			
887	888		1	1			
889	890		1	1			
890	891		0	3			
			Name	Sex	Age	SibSp	\
0			Braund, Mr. Owen Harris	male	22	1	
1		Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38	1		
2		Heikkinen, Miss. Laina	female	26	0		
3		Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1		
4		Allen, Mr. William Henry	male	35	0		
..
885		Rice, Mrs. William (Margaret Norton)	female	39	0		
886		Montvila, Rev. Juozas	male	27	0		
887		Graham, Miss. Margaret Edith	female	19	0		
889		Behr, Mr. Karl Howell	male	26	0		
890		Dooley, Mr. Patrick	male	32	0		
	Parch	Ticket	Fare	Cabin	Embarked		
0	0	A/5 21171	7.2500	NaN	S		
1	0	PC 17599	71.2833	C85	C		
2	0	STON/O2. 3101282	7.9250	NaN	S		
3	0		53.1000	C123	S		
4	0		8.0500	NaN	S		
..		
885	5	382652	29.1250	NaN	Q		
886	0	211536	13.0000	NaN	S		
887	0	112053	30.0000	B42	S		
889	0	111369	30.0000	C148	C		
890	0	370376	7.7500	NaN	Q		

[714 rows x 12 columns]

Now our “Age” column is no longer a float, rather an integer.

8.3.5 Conclusion

Sorting, cleaning, and organizing data in Pandas can require practice. Even after years of using Pandas, you will still find yourself looking things up on Stackoverflow or other resources online. The power of Pandas comes at the cost of making it difficult to master quickly. With regular practice, however, Pandas does get easier to use over time. Once you have a command of Pandas, you can do quick data cleaning and analysis all in Python. In the next chapter, we will look at how leverage Pandas to do more advanced searching methods.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

9

Searching for Data

9.1 Advanced Searching on Strings

9.1.1 Finding Features within a String

```
import pandas as pd
df = pd.read_csv("../data/titanic.csv")
df = df[["Name"]]
df
```

```
Name
0 Braund, Mr. Owen Harris
1 Cumings, Mrs. John Bradley (Florence Briggs Th...
2 Heikkinen, Miss. Laina
3 Futrelle, Mrs. Jacques Heath (Lily May Peel)
4 Allen, Mr. William Henry
...
886 Montvila, Rev. Juozas
887 Graham, Miss. Margaret Edith
888 Johnston, Miss. Catherine Helen "Carrie"
889 Behr, Mr. Karl Howell
890 Dooley, Mr. Patrick

[891 rows x 1 columns]
```

When I am looking at the “df”, I notice that there is a “Rev.” in index 886. As a historian, I find this fascinating. Now, I start to wonder, how many reverends were there on the Titanic? Is this individual unique? If I wanted to ask this question outside of Pandas, I could do the following:

```
names = df.Name.tolist()
revs = []
for name in names:
    if "Rev." in name:
        revs.append(name)
print (revs)
```

```
['Byles, Rev. Thomas Roussel Davids', 'Bateman, Rev. Robert James', 'Carter,
→ Rev. Ernest Courtenay', 'Kirkland, Rev. Charles Leonard', 'Harper, Rev.-
→John', 'Montvila, Rev. Juozas']
```

Sure, that works, but I don’t have any of the other data associated with each of these reverends. I would have to then do some manual searching in the DataFrame to find their corresponding data, or save the data as a dictionary and then run look ups. But why do all

of that, when we can do it in a single line of code using Pandas' built-in function. We can use `.str.contains()` which takes an argument of what we want to return.

```
df.loc[df["Name"].str.contains("Rev\.")]
```

	Name
149	Byles, Rev. Thomas Roussel Davids
150	Bateman, Rev. Robert James
249	Carter, Rev. Ernest Courtenay
626	Kirkland, Rev. Charles Leonard
848	Harper, Rev. John
886	Montvila, Rev. Juozas

We can, therefore, see not only the reverends, but also their corresponding data.

9.1.2 Finding Strings That Don't Contain Feature

What if we wanted to eliminate all names that do not contain "Rev."? We can introduce "`~`" prior to "df" to specify that the "Name" column should not have whatever condition we express.

```
df.loc[~df["Name"].str.contains("Rev\.")]
```

	Name
0	Braund, Mr. Owen Harris
1	Cumings, Mrs. John Bradley (Florence Briggs Th...)
2	Heikkinen, Miss. Laina
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)
4	Allen, Mr. William Henry
..	...
885	Rice, Mrs. William (Margaret Norton)
887	Graham, Miss. Margaret Edith
888	Johnston, Miss. Catherine Helen "Carrie"
889	Behr, Mr. Karl Howell
890	Dooley, Mr. Patrick

[885 rows x 1 columns]

9.1.3 Using RegEx with Pandas

Out of the box, Pandas supports RegEx. RegEx stands for Regular Expressions. It is a powerful way of performing complex string matching. If we were interested in finding any instance of "Rev." or "Mr.", we would have to write something like this without RegEx:

```
df.loc[(df["Name"].str.contains("Rev\.")) | (df["Name"].str.contains("Mr\."))]
```

	Name
0	Braund, Mr. Owen Harris
4	Allen, Mr. William Henry
5	Moran, Mr. James
6	McCarthy, Mr. Timothy J
12	Saundercock, Mr. William Henry
..	...
883	Banfield, Mr. Frederick James
884	Sutewall, Mr. Henry Jr

(continues on next page)

(continued from previous page)

```

886      Montvila, Rev. Juozas
889      Behr, Mr. Karl Howell
890      Dooley, Mr. Patrick

[523 rows x 1 columns]

```

While this works, imagine if we had 20 or 30 different conditions. That would be a very long piece of code to write and while it would work, it is always best practice to write shorter, tighter code. So, let's do the same thing, but with RegEx. We can add the `or` condition into the `str.contains()` argument. This is a RegEx command. To ensure that RegEx is registered, it may be necessary to pass it as an argument.

```
df.loc[df["Name"].str.contains("Rev\..|Mr\.", regex=True)]
```

	Name
0	Braund, Mr. Owen Harris
4	Allen, Mr. William Henry
5	Moran, Mr. James
6	McCarthy, Mr. Timothy J
12	Saundercock, Mr. William Henry
..	...
883	Banfield, Mr. Frederick James
884	Sutewall, Mr. Henry Jr
886	Montvila, Rev. Juozas
889	Behr, Mr. Karl Howell
890	Dooley, Mr. Patrick

```
[523 rows x 1 columns]
```

In some instances, we may have uncleaned data and the use of "Rev." may be lowercase in one instance. To ensure that we grab both upper and lowercase forms of this sequence, let's ignore the case by using the `case` keyword and setting it to "False".

```
import re
df.loc[df["Name"].str.contains("Rev\..|Mr\.", case=False, regex=True)]
```

	Name
0	Braund, Mr. Owen Harris
4	Allen, Mr. William Henry
5	Moran, Mr. James
6	McCarthy, Mr. Timothy J
12	Saundercock, Mr. William Henry
..	...
883	Banfield, Mr. Frederick James
884	Sutewall, Mr. Henry Jr
886	Montvila, Rev. Juozas
889	Behr, Mr. Karl Howell
890	Dooley, Mr. Patrick

```
[523 rows x 1 columns]
```

9.2 Filter and Querying

9.2.1 Introduction

In this section, we will meet two advanced ways of filtering or searching (querying) our data. These are aptly named `filter()` and `query()` functions. These two functions allow us to do some fairly advanced things within a narrow scope. By narrow scope, I mean the questions that we want to pose. Whenever you are manipulating or probing data, it is always best to think about the task as simply asking a question. In essence, this is precisely what you are doing. You are asking the database a question. In order to ask the question correctly, as is the case with any language, you need to know the correct syntax and when that particular question is the right one to ask. In this section, we explore how to frame specific questions with `filter()` and `query()`.

Each function is used in particular circumstances. `filter()` is useful for getting a large data down to a smaller size, based on the questions you want to ask. `query()`, on the other hand, is useful for phrasing questions that use comparison operators (less than, equal to, greater than, etc.). Let's explore each in turn, but first, let's import our Titanic dataset.

```
import pandas as pd
df = pd.read_csv("../data/titanic.csv")
df
```

	PassengerId	Survived	Pclass	\
0		1	0	3
1		2	1	1
2		3	1	3
3		4	1	1
4		5	0	3
..
886	887	888	889	890
887	888	889	889	890
888	889	890	890	891
889	890	891	891	891
890	891	891	891	891

	Name	Sex	Age	SibS	\
0	Braund, Mr. Owen Harris	male	22.0	1	
1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	female	38.0	1	
2	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	26.0	0	
3	Allen, Mr. William Henry	male	35.0	0	
4					
..
886	Montvila, Rev. Juozas	male	27.0	0	
887	Graham, Miss. Margaret Edith	female	19.0	0	
888	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	
889	Behr, Mr. Karl Howell	male	26.0	0	
890	Dooley, Mr. Patrick	male	32.0	0	

	Parch	Ticket	Fare	Cabin	Embarked
0	0	A/5 21171	7.2500	NaN	S
1	0	PC 17599	71.2833	C85	C
2	0	STON/O2. 3101282	7.9250	NaN	S
3	0	113803	53.1000	C123	S
4	0	373450	8.0500	NaN	S

(continues on next page)

(continued from previous page)

```

...     ...
886      0        211536  13.0000  NaN    S
887      0        112053  30.0000  B42    S
888      2        W./C. 6607  23.4500  NaN    S
889      0        111369  30.0000  C148   C
890      0        370376  7.7500  NaN    Q

```

[891 rows x 12 columns]

9.2.2 The Filter Function

The filter function is a great way to grab only the relevant columns. The syntax of filter is a bit easier to use. It takes a single argument, a list of strings. These strings correspond to the columns.

When to use filter(): Use filter() when you want to get a quick sense of your dataset or, as we shall see, create a new DataFrame based on the columns you want. It is particularly useful if your dataset has many columns. You can also use it to reorder your columns in a more desired way.

Let's say that we are interested in just studying the names of the passengers of the Titanic. It does not make sense to work with the entire DataFrame. We can, therefore, use filter to just grab the names column, like so:

```
df.filter(["Name"])
```

	Name
0	Braund, Mr. Owen Harris
1	Cumings, Mrs. John Bradley (Florence Briggs Th...
2	Heikkinen, Miss. Laina
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)
4	Allen, Mr. William Henry
..	...
886	Montvila, Rev. Juozas
887	Graham, Miss. Margaret Edith
888	Johnston, Miss. Catherine Helen "Carrie"
889	Behr, Mr. Karl Howell
890	Dooley, Mr. Patrick

[891 rows x 1 columns]

This is great, but what if I also want to see the ages of these passengers. No problem. I can add an additional column to the list.

```
df.filter(["Name", "Age"])
```

	Name	Age
0	Braund, Mr. Owen Harris	22.0
1	Cumings, Mrs. John Bradley (Florence Briggs Th...	38.0
2	Heikkinen, Miss. Laina	26.0
3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	35.0
4	Allen, Mr. William Henry	35.0
..
886	Montvila, Rev. Juozas	27.0
887	Graham, Miss. Margaret Edith	19.0

(continues on next page)

(continued from previous page)

```
888      Johnston, Miss. Catherine Helen "Carrie"    NaN
889                               Behr, Mr. Karl Howell  26.0
890                               Dooley, Mr. Patrick  32.0
```

[891 rows x 2 columns]

What if I want age to come before name? I can rearrange the order.

```
df.filter(["Age", "Name"])
```

	Age	Name
0	22.0	Braund, Mr. Owen Harris
1	38.0	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina
2	26.0	
3	35.0	Futrelle, Mrs. Jacques Heath (Lily May Peel) Allen, Mr. William Henry
4	35.0	
..
886	27.0	Montvila, Rev. Juozas
887	19.0	Graham, Miss. Margaret Edith
888	NaN	Johnston, Miss. Catherine Helen "Carrie"
889	26.0	Behr, Mr. Karl Howell
890	32.0	Dooley, Mr. Patrick

[891 rows x 2 columns]

Note that we are not bound to the order of the DataFrame. This is particularly useful if we want to make a new DataFrame.

```
new_df = df.filter(["Age", "Name"])
new_df
```

	Age	Name
0	22.0	Braund, Mr. Owen Harris
1	38.0	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina
2	26.0	
3	35.0	Futrelle, Mrs. Jacques Heath (Lily May Peel) Allen, Mr. William Henry
4	35.0	
..
886	27.0	Montvila, Rev. Juozas
887	19.0	Graham, Miss. Margaret Edith
888	NaN	Johnston, Miss. Catherine Helen "Carrie"
889	26.0	Behr, Mr. Karl Howell
890	32.0	Dooley, Mr. Patrick

[891 rows x 2 columns]

Now, we can only examine the data that we actually need. This will make our code faster and require examining fewer data for each row. Filter's big limitation is in the fact that it cannot filter the data further. We cannot, for example, add an extra argument to `filter()` that would only return the names with "Miss.", but we can tack on additional arguments to `filter`, such as those that we saw in the previous chapter on searching strings.

Let's try that now.

```
df.filter(["Age", "Name"]).Name.str.contains("Miss\.)")
```

```

0      False
1      False
2      True
3     False
4     False
...
886    False
887    True
888    True
889   False
890   False
Name: Name, Length: 891, dtype: bool

```

Note that we now have a list of “True” or “False” statements. These tell us if the word “Miss.” is in the column “Name”. If it is there, we see “True”. If it is not, we see “False”. Let’s say we want to know how many passengers have the title “Miss.”, we stack `.value_counts()` into the chain. Note the plural of counts.

```
df.filter(["Age", "Name"]).Name.str.contains("Miss\.").value_counts()
```

```

False    709
True    182
Name: Name, dtype: int64

```

Now, let’s say we were interested in ONLY seeing the rows that contain “Miss.” in them. We need to structure that filtering as a list. But note that if we wrap the whole thing in a list, we don’t filter out just the “Age” and “Name” columns. Instead, we get the entire DataFrame.

```
df[df.filter(["Age", "Name"]).Name.str.contains("Miss\.") == True]
```

	PassengerId	Survived	Pclass	Name \				
2	3	1	3	Heikkinen, Miss. Laina				
10	11	1	3	Sandstrom, Miss. Marguerite Rut				
11	12	1	1	Bonnell, Miss. Elizabeth				
14	15	0	3	Vestrom, Miss. Hulda Amanda Adolfina				
22	23	1	3	McGowan, Miss. Anna "Annie"				
..				
866	867	1	2	Duran y More, Miss. Asuncion				
875	876	1	3	Najib, Miss. Adele Kiamie "Jane"				
882	883	0	3	Dahlberg, Miss. Gerda Ulrika				
887	888	1	1	Graham, Miss. Margaret Edith				
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"				
	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
2	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
10	female	4.0	1	1	PP 9549	16.7000	G6	S
11	female	58.0	0	0	113783	26.5500	C103	S
14	female	14.0	0	0	350406	7.8542	NaN	S
22	female	15.0	0	0	330923	8.0292	NaN	Q
..
866	female	27.0	1	0	SC/PARIS 2149	13.8583	NaN	C
875	female	15.0	0	0	2667	7.2250	NaN	C
882	female	22.0	0	0	7552	10.5167	NaN	S
887	female	19.0	0	0	112053	30.0000	B42	S
888	female	NaN	1	2	W./C. 6607	23.4500	NaN	S

[182 rows x 12 columns]

This is because of where our filter occurs in the chain of commands. Note that filter occurs within the brackets where we are setting up our parameters. This means that we are filtering under the conditions of how the list is created, but that once a row is processed, the DataFrame is unfiltered. If we want our filter to work, we need to place it after the conditions have been sorted. Notice that the filter is outside of our brackets now.

```
df[df.Name.str.contains("Miss\."). == True].filter(["Age", "Name"])
```

	Age	Name
2	26.0	Heikkinen, Miss. Laina
10	4.0	Sandstrom, Miss. Marguerite Rut
11	58.0	Bonnell, Miss. Elizabeth
14	14.0	Vestrom, Miss. Hulda Amanda Adolfina
22	15.0	McGowan, Miss. Anna "Annie"
..
866	27.0	Duran y More, Miss. Asuncion
875	15.0	Najib, Miss. Adele Kiamie "Jane"
882	22.0	Dahlberg, Miss. Gerda Ulrika
887	19.0	Graham, Miss. Margaret Edith
888	NaN	Johnston, Miss. Catherine Helen "Carrie"

[182 rows x 2 columns]

9.2.3 The Query Function

The Pandas `query()` method is a fantastic way to filter and query data. Unlike other Pandas methods, it uses a string argument that functions rather similar to SQL syntax.

When to use query: You should only use `query()` when your question (query) can be posed as greater than, less than, equal to, or not equal to (or some combination of these). Let me demonstrate. If we wanted to filter out all the rows where the “Pclass” was equal to 3, we could use the following string:

```
df.query("Pclass == 3")
```

	PassengerId	Survived	Pclass	Name \
0		1	0	Braund, Mr. Owen Harris
2		3	1	Heikkinen, Miss. Laina
4		5	0	Allen, Mr. William Henry
5		6	0	Moran, Mr. James
7		8	0	Palsson, Master. Gosta Leonard
..
882	883	0	3	Dahlberg, Miss. Gerda Ulrika
884	885	0	3	Suttehall, Mr. Henry Jr
885	886	0	3	Rice, Mrs. William (Margaret Norton)
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"
890	891	0	3	Dooley, Mr. Patrick

	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	male	35.0	0	0	373450	8.0500	NaN	S
5	male	NaN	0	0	330877	8.4583	NaN	Q
7	male	2.0	3	1	349909	21.0750	NaN	S
..
882	female	22.0	0	0	7552	10.5167	NaN	S

(continues on next page)

(continued from previous page)

```

884    male  25.0      0      0    SOTON/OQ 392076   7.0500  NaN      S
885  female  39.0      0      5          382652 29.1250  NaN      Q
888  female    NaN      1      2        W./C. 6607  23.4500  NaN      S
890    male  32.0      0      0          370376  7.7500  NaN      Q

```

[491 rows x 12 columns]

Query can also look for if a column contains any item in a list. We can do this by setting "Pclass == [list]".

```
df.query("Pclass == [1,3]")
```

	PassengerId	Survived	Pclass	\		
0		1	0	3		
1		2	1	1		
2		3	1	3		
3		4	1	1		
4		5	0	3		
..			
885	886	886	0	3		
887	888	888	1	1		
888	889	889	0	3		
889	890	890	1	1		
890	891	891	0	3		
		Name	Sex	Age	SibSp	\
0		Braund, Mr. Owen Harris	male	22.0		1
1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina		female	38.0		1 0
2			female	26.0		
3	Futrelle, Mrs. Jacques Heath (Lily May Peel) Allen, Mr. William Henry		female	35.0		1 0
4			male	35.0		
..		
885	Rice, Mrs. William (Margaret Norton) Graham, Miss. Margaret Edith		female	39.0		0
887			female	19.0		0
888	Johnston, Miss. Catherine Helen "Carrie"		female	NaN		1
889			male	26.0		0
890	Behr, Mr. Karl Howell Dooley, Mr. Patrick		male	32.0		0
	Parch	Ticket	Fare	Cabin	Embarked	
0	0	A/5 21171	7.2500	NaN	S	
1	0	PC 17599	71.2833	C85	C	
2	0	STON/O2. 3101282	7.9250	NaN	S	
3	0		53.1000	C123	S	
4	0		8.0500	NaN	S	
..	
885	5	382652	29.1250	NaN	Q	
887	0	112053	30.0000	B42	S	
888	2	W./C. 6607	23.4500	NaN	S	
889	0	111369	30.0000	C148	C	
890	0	370376	7.7500	NaN	Q	

[707 rows x 12 columns]

We can even stack questions together within this string. Let's say, I am interested in all who were in "Pclass 3" and "Survived". I could write the following string argument:

```
df.query("Pclass == 3 & Survived == 1")
```

	PassengerId	Survived	Pclass	\		
2	3	1	3			
8	9	1	3			
10	11	1	3			
19	20	1	3			
22	23	1	3			
..			
838	839	1	3			
855	856	1	3			
858	859	1	3			
869	870	1	3			
875	876	1	3			
		Name	Sex	Age	SibSp	\
2		Heikkinen, Miss. Laina	female	26.0	0	
8	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)		female	27.0	0	
10		Sandstrom, Miss. Marguerite Rut	female	4.0	1	
19		Masselmanni, Mrs. Fatima	female	NaN	0	
22		McGowan, Miss. Anna "Annie"	female	15.0	0	
..		
838		Chip, Mr. Chang	male	32.0	0	
855		Aks, Mrs. Sam (Leah Rosen)	female	18.0	0	
858		Baclini, Mrs. Solomon (Latifa Qurban)	female	24.0	0	
869		Johnson, Master. Harold Theodor	male	4.0	1	
875		Najib, Miss. Adele Kiamie "Jane"	female	15.0	0	
		
	Parch	Ticket	Fare	Cabin	Embarked	
2	0	STON/O2. 3101282	7.9250	NaN	S	
8	2	347742	11.1333	NaN	S	
10	1	PP 9549	16.7000	G6	S	
19	0	2649	7.2250	NaN	C	
22	0	330923	8.0292	NaN	Q	
..	
838	0	1601	56.4958	NaN	S	
855	1	392091	9.3500	NaN	S	
858	3	2666	19.2583	NaN	C	
869	1	347742	11.1333	NaN	S	
875	0	2667	7.2250	NaN	C	
		
[119 rows x 12 columns]						

Let's make the question even more complex. We want to now find the number of these individuals who were over the age of 40.

```
df.query("Pclass == 3 & Survived == 1 & Age > 40")
```

	PassengerId	Survived	Pclass		Name	Sex	Age	\
338	339	1	3	Dahl, Mr. Karl Edwardt	male	45.0		
414	415	1	3	Sundman, Mr. Johan Julian	male	44.0		
483	484	1	3	Turkula, Mrs. (Hedwig)	female	63.0		
	SibSp	Parch		Ticket	Fare	Cabin	Embarked	
338	0	0		7598	8.0500	NaN	S	
414	0	0	STON/O 2.	3101269	7.9250	NaN	S	
483	0	0			4134	9.5875	NaN	S

I now have a list of three individuals who met all criteria. I can also use my “or” operator (|), rather than “&”. Let's see if we can achieve what we want. (Note this is an intentional mistake. Look below for why.)

```
df.query("Pclass == 3 & Survived == 1 & Age > 40 | Age < 10")
```

	PassengerId	Survived	Pclass	Name \
7	8	0	3	Palsson, Master. Gosta Leonard
10	11	1	3	Sandstrom, Miss. Marguerite Rut
16	17	0	3	Rice, Master. Eugene
24	25	0	3	Palsson, Miss. Torborg Danira
43	44	1	2	Laroche, Miss. Simonne Marie Anne Andree
..
827	828	1	2	Mallet, Master. Andre
831	832	1	2	Richards, Master. George Sibley
850	851	0	3	Andersson, Master. Sigvard Harald Elias
852	853	0	3	Boulos, Miss. Nourelain
869	870	1	3	Johnson, Master. Harold Theodor
	Sex	Age	SibSp	Parch
7	male	2.00	3	1
10	female	4.00	1	1
16	male	2.00	4	1
24	female	8.00	3	1
43	female	3.00	1	2
			Ticket	Fare Cabin Embarked
7	male	2.00	349909	21.0750 NaN S
10	female	4.00	PP 9549	16.7000 G6 S
16	male	2.00	382652	29.1250 NaN Q
24	female	8.00	349909	21.0750 NaN S
43	female	3.00	SC/Paris 2123	41.5792 NaN C
..
827	male	1.00	0	S.C./PARIS 2079 37.0042 NaN C
831	male	0.83	1	29106 18.7500 NaN S
850	male	4.00	4	347082 31.2750 NaN S
852	female	9.00	1	2678 15.2458 NaN C
869	male	4.00	1	347742 11.1333 NaN S

[65 rows x 12 columns]

Woops! Something has gone seriously wrong here. We have all different kinds of “Pclasses”, not just “3s”. We have people who survived and did not. And, most problematically, we have people only under the age of 10. What has gone wrong here!? The answer lies in a perhaps forgotten part of math from when we were children, the order of operations. If you recall from those lessons, the order of operations determines the way in which you process the problem. $4 + 7 \times 2$ is very different from $(4 + 7) \times 2$. The former is 18 and the latter is 22 because the latter has parentheses which tell the reader to do that operation first. Because programming sits on top of mathematics (especially Boolean algebra), the syntax of mathematics is often embedded in programming.

Let's use the order of operations correctly and rephrase our query. Note the parentheses now before Age > 40 and after Age < 10. Note also that the “&” is before the parentheses.

```
df.query("Pclass == 3 & Survived == 1 & (Age > 40 | Age < 10)")
```

	PassengerId	Survived	Pclass	\
10	11	1	3	
165	166	1	3	
172	173	1	3	
184	185	1	3	
233	234	1	3	
261	262	1	3	
338	339	1	3	
348	349	1	3	
381	382	1	3	
414	415	1	3	
448	449	1	3	
469	470	1	3	
479	480	1	3	

(continues on next page)

(continued from previous page)

483	484	1	3					
489	490	1	3					
644	645	1	3					
691	692	1	3					
751	752	1	3					
777	778	1	3					
788	789	1	3					
803	804	1	3					
869	870	1	3					
10		Name	Sex	Age	SibSp	\		
165	Goldsmith, Master.	Sandstrom, Miss. Marguerite Rut	female	4.00		1		
172		Frank John William "Frankie"	male	9.00		0		
184		Johnson, Miss. Eleanor Ileen	female	1.00		1		
233		Kink-Heilmann, Miss. Luise Gretchen	female	4.00		0		
261		Asplund, Miss. Lillian Gertrud	female	5.00		4		
338		Asplund, Master. Edvin Rojj Felix	male	3.00		4		
348	Coutts, Master.	Dahl, Mr. Karl Edward	male	45.00		0		
381		William Loch "William"	male	3.00		1		
414		Nakid, Miss. Maria ("Mary")	female	1.00		0		
448		Sundman, Mr. Johan Julian	male	44.00		0		
469		Baclini, Miss. Marie Catherine	female	5.00		2		
479		Baclini, Miss. Helene Barbara	female	0.75		2		
483		Hirvonen, Miss. Hildur E	female	2.00		0		
489	Coutts, Master.	Turkula, Mrs. (Hedwig)	female	63.00		0		
644		Eden Leslie "Neville"	male	9.00		1		
691		Baclini, Miss. Eugenie	female	0.75		2		
751		Karun, Miss. Manca	female	4.00		0		
777		Moor, Master. Meier	male	6.00		0		
788		Emanuel, Miss. Virginia Ethel	female	5.00		0		
803		Dean, Master. Bertram Vere	male	1.00		1		
869	Thomas, Master.	Assad Alexander	male	0.42		0		
	Johnson, Master.	Harold Theodor	male	4.00		1		
Parch		Ticket	Fare	Cabin	Embarked			
10	1	PP 9549	16.7000	G6	S			
165	2	363291	20.5250	NaN	S			
172	1	347742	11.1333	NaN	S			
184	2	315153	22.0250	NaN	S			
233	2	347077	31.3875	NaN	S			
261	2	347077	31.3875	NaN	S			
338	0	7598	8.0500	NaN	S			
348	1	C.A. 37671	15.9000	NaN	S			
381	2	2653	15.7417	NaN	C			
414	0	STON/O 2.	3101269	7.9250	NaN	S		
448	1		2666	19.2583	NaN	C		
469	1		2666	19.2583	NaN	C		
479	1		3101298	12.2875	NaN	S		
483	0		4134	9.5875	NaN	S		
489	1	C.A. 37671	15.9000	NaN	S			
644	1		2666	19.2583	NaN	C		
691	1		349256	13.4167	NaN	C		
751	1		392096	12.4750	E121	S		
777	0		364516	12.4750	NaN	S		
788	2	C.A. 2315	20.5750	NaN	S			
803	1		2625	8.5167	NaN	C		
869	1		347742	11.1333	NaN	S		

This is why query is such a powerful function in Pandas. You can do a lot with a single string. There are other ways to achieve this same result, but if your question can be entirely phrased as a series of comparison operators (equal to, less than, etc.), then query is likely the best option.

9.3 Grouping with groupby()

9.3.1 Introduction

When working with large quantities of data, it can sometimes be a bit difficult to understand broad patterns within your data. Often, you will need to group your data into small subsections based on some parameter, such as age, name, or some other feature. You can do this in Pandas using `groupby()`, which will be the main subject of this section. **Groupby** is a feature of Pandas that returns a special groupby object. This object can be called to perform different types of analyses on data, especially when leveraging the built-in quantitative features of Pandas, such as `count()` and `sum()`. In this section, we will explore these features and see how they can be used on a real-world dataset, the Titanic dataset.

```
import pandas as pd
df = pd.read_csv("../data/titanic.csv")
df = df[["Name", "Sex", "Age", "Pclass", "Fare"]]
df
```

```
Name      Sex   Age  Pclass \
0    Braund, Mr. Owen Harris    male  22.0      3
1    Cumings, Mrs. John Bradley (Florence Briggs Th... female  38.0      1
2          Heikkinen, Miss. Laina  female  26.0      3
3    Futrelle, Mrs. Jacques Heath (Lily May Peel) female  35.0      1
4        Allen, Mr. William Henry    male  35.0      3
..          ...
886       Montvila, Rev. Juozas    male  27.0      2
887       Graham, Miss. Margaret Edith female  19.0      1
888  Johnston, Miss. Catherine Helen "Carrie" female   NaN      3
889           Behr, Mr. Karl Howell    male  26.0      1
890         Dooley, Mr. Patrick    male  32.0      3

Fare
0    7.2500
1    71.2833
2    7.9250
3    53.1000
4    8.0500
..    ...
886  13.0000
887  30.0000
888  23.4500
889  30.0000
890  7.7500

[891 rows x 5 columns]
```

9.3.2 groupby()

The `groupby()` function allows us to easily group our data in the DataFrame. Once your data are grouped, there are a lot of quantitative questions you can begin to ask. Let's start simple. Let's group our DataFrame by "Sex".

```
df.groupby("Sex")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002754F3CFB80>
```

This output may not be quite what you expect. This is an object to which we can now pose targeted questions. Let's try and see a DataFrame that only has "male" in the "Sex" column. We can do that by using `get_group("male")`.

```
df.groupby("Sex").get_group("male")
```

		Name	Sex	Age	Pclass	Fare
0	Braund, Mr. Owen Harris		male	22.0	3	7.2500
4	Allen, Mr. William Henry		male	35.0	3	8.0500
5	Moran, Mr. James		male	Nan	3	8.4583
6	McCarthy, Mr. Timothy J		male	54.0	1	51.8625
7	Palsson, Master. Gosta Leonard		male	2.0	3	21.0750
..
883	Banfield, Mr. Frederick James		male	28.0	2	10.5000
884	Suthehall, Mr. Henry Jr		male	25.0	3	7.0500
886	Montvila, Rev. Juozas		male	27.0	2	13.0000
889	Behr, Mr. Karl Howell		male	26.0	1	30.0000
890	Dooley, Mr. Patrick		male	32.0	3	7.7500

[577 rows x 5 columns]

This argument does not have to be a string. Let's say, we want to just get all the people who are aged 20. We can do the same thing by grouping the dataset by "Age" and then getting the group of 20 year olds.

```
df.groupby("Age").get_group(20)
```

		Name	Sex	Age	Pclass	Fare
12	Saundercock, Mr. William Henry		male	20.0	3	8.0500
91	Andreasson, Mr. Paul Edvin		male	20.0	3	7.8542
113	Jussila, Miss. Katriina		female	20.0	3	9.8250
131	Coelho, Mr. Domingos Fernandeo		male	20.0	3	7.0500
378	Betros, Mr. Tannous		male	20.0	3	4.0125
404	Oreskovic, Miss. Marija		female	20.0	3	8.6625
441	Hampe, Mr. Leon		male	20.0	3	9.5000
622	Nakid, Mr. Sahid		male	20.0	3	15.7417
640	Jensen, Mr. Hans Peder		male	20.0	3	7.8542
664	Lindqvist, Mr. Eino William		male	20.0	3	7.9250
682	Olsvigen, Mr. Thor Anderson		male	20.0	3	9.2250
725	Oreskovic, Mr. Luka		male	20.0	3	8.6625
762	Barah, Mr. Hanna Assi		male	20.0	3	7.2292
840	Alhomaki, Mr. Ilmari Rudolf		male	20.0	3	7.9250
876	Gustafsson, Mr. Alfred Ossian		male	20.0	3	9.8458

9.3.3 Quantitative Analysis with `.count()` and `.sum()`

This is typically not how you would use the `groupby` function. It is far more powerful and often used for quantitative analysis on subsets of your data. Let's say that I want to examine my dataset by sex and I am interested in known the quantity of column based solely on the metric of sex. I could use `groupby()` and `.count()`. When chained together, our question then becomes, how many "PassengerId", "Survived", "Pclass", "Name", etc., do we see for each column based on sex. While this question is particularly useful for the qualitative rows (such as "Name") or numerical strings (such as "PassengerId") because they display

the total number of passengers because each person has a unique “PassengerId” and “Name”.

```
df.groupby("Sex").count()
```

	Name	Age	Pclass	Fare
Sex				
female	314	261	314	314
male	577	453	577	577

For the quantitative rows, we can use `sum()` function. This will tell us the sum of all the columns that have floats or integers. Note that this is not a really good question to pose for the “Age” column. It is, however, very useful for the “Fare” column and the “Survived” column. Remember, if a person survived, they have a 1; if they did not, they have a 0. We can use the sum to know how many male vs. female survivors there were.

```
df.groupby("Sex").sum()
```

	Age	Pclass	Fare
Sex			
female	7286.00	678	13966.6628
male	13919.17	1379	14727.2865

Let’s say, though, that we are only interested in the “Fare” column. Before we add `sum` to our chain, we can specify that we want specifically the “Fare” column.

```
df.groupby("Sex").Fare.sum()
```

Sex	
female	13966.6628
male	14727.2865
Name: Fare, dtype:	float64

9.3.4 Working with Multiple Groups

Now, we have just the data on a single column. We can see that the combined fare of male passengers was greater than the combined sum of female passengers. Let’s say though that we are interested in how these sums divide over “Pclass”. We can pass a list to `groupby`, rather than just a string. This list will be a list of strings that correspond to columns.

```
df.groupby(["Sex", "Pclass"]).Fare.sum()
```

	Sex	Pclass	Fare
female	1		9975.8250
	2		1669.7292
	3		2321.1086
male	1		8201.5875
	2		2132.1125
	3		4393.5865
Name: Fare, dtype:	float64		

The result of this new question is more nuanced. We are not looking at the sum of all fares, rather the sum of fares divided on a “Pclass-by-Pclass” basis. This means that we can now understand that these sums varied by class. For example, while the total fare for male

passengers was greater, the total fare for first class female passengers was greater than their first class male counterparts. The male fare, however, is greater for both the 2nd Class and 3rd Class groups.

9.3.5 Groupings with Many Subsets

What if we were interested in something that would have more than just six neat subsections, such as three classes per sex. What if we also wanted to add another aspect to the groups, such as age. If we try and do that, our results are cutoff. We can try and use `pd.set_option()`.

```
df.groupby(["Sex", "Pclass", "Age"]).Fare.sum()
```

Sex	Pclass	Age	Fare
female	1	2.0	151.5500
		14.0	120.0000
		15.0	211.3375
		16.0	183.8792
		17.0	165.9000
male	3	59.0	7.2500
		61.0	6.2375
		65.0	7.7500
		70.5	7.7500
		74.0	7.7750

What if we wanted to make this look a bit nicer, as a Pandas DataFrame? We can pass all our data back into a new DataFrame object.

```
df = pd.DataFrame(df.groupby(["Sex", "Pclass", "Age"]).Fare.sum())
df
```

Fare			
Sex	Pclass	Age	
female	1	2.0	151.5500
		14.0	120.0000
		15.0	211.3375
		16.0	183.8792
		17.0	165.9000
male	3	59.0	7.2500
		61.0	6.2375
		65.0	7.7500
		70.5	7.7500
		74.0	7.7750

[283 rows x 1 columns]

This is now a bit easier to read. You should now have a fairly good understanding of how to group data in Pandas using `groupby()` and some of the more powerful ways you can use `groupby()` to manipulate quantitative data.

10

Advanced Pandas

10.1 Plotting Data with Pandas

10.1.1 Importing the DataFrame

Pandas sits on top of Matplotlib, one of the standard libraries used by data scientists for plotting data. As we will see in the next notebooks, you can also leverage other, more robust, graphing libraries through Pandas. For now, though, let's start with the basics. In this notebook, we will explore how to create three types of graphs: bar (and barh), pie, and scatter. I will also introduce you to some of the more recent features of Pandas 1.3.0, that allow you to control the graph a bit more.

Before we do any of that, however, let's import pandas and our data.

```
import pandas as pd
```

```
df = pd.read_csv("../data/titanic.csv")
```

```
df
```

	PassengerId	Survived	Pclass	\
0		1	0	3
1		2	1	1
2		3	1	3
3		4	1	1
4		5	0	3
..
886	887	0	2	
887	888	1	1	
888	889	0	3	
889	890	1	1	
890	891	0	3	
		Name	Sex	Age
0	Braund, Mr. Owen Harris	male	22.0	1
1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	female	38.0	1 0
2	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	26.0	
3	Allen, Mr. William Henry	male	35.0	1
4	35.0	0
..
886	Montvila, Rev. Juozas	male	27.0	0
887	Graham, Miss. Margaret Edith	female	19.0	0
888	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1
889	Behr, Mr. Karl Howell	male	26.0	0
890	Dooley, Mr. Patrick	male	32.0	0

(continues on next page)

(continued from previous page)

```
Parch      Ticket      Fare Cabin Embarked
0         0    A/5 21171  7.2500   NaN     S
1         0      PC 17599  71.2833  C85     C
2         0  STON/O2. 3101282  7.9250   NaN     S
3         0        113803  53.1000  C123     S
4         0       373450  8.0500   NaN     S
..        ...
886        0       211536 13.0000  NaN     S
887        0       112053 30.0000  B42     S
888        2      W./C. 6607 23.4500  NaN     S
889        0       111369 30.0000  C148     C
890        0       370376  7.7500  NaN     Q
[891 rows x 12 columns]
```

10.1.2 Bar and Barh Charts with Pandas

With our data imported successfully, let's jump right in with bar charts. Bar charts a great way to visualize qualitative data quantitatively. To demonstrate what I mean by this, let's consider if we wanted to know how many male passengers were on the Titanic relative to female passengers. I could grab all the value counts and look at the numbers by calling `.value_counts()`, as in the example below.

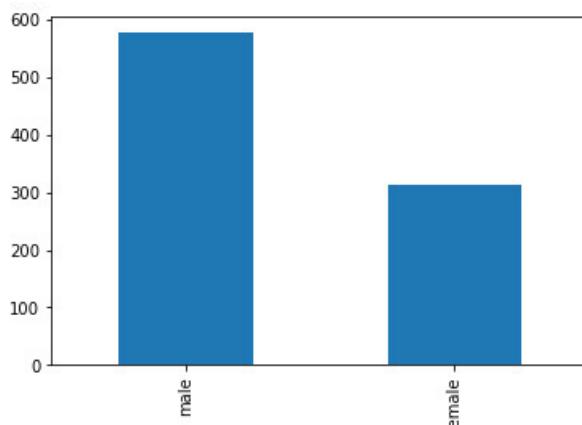
```
df['Sex'].value_counts()
```

```
male      577
female    314
Name: Sex, dtype: int64
```

This kind of raw numerical data is useful, but it is often difficult to present visually to audiences. For this reason, it is quite common to have the raw numerical data available, but to give the audience a quick sense of the numbers visually. We can take that initial code we see above and using the `.plot.bar()` function we get the following result.

```
df['Sex'].value_counts().plot.bar()
```

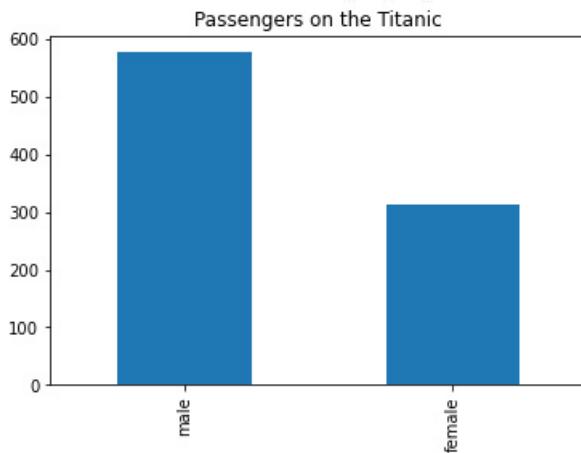
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b2877751c0>
```



Not bad, but this chart is quite staid. For one thing, it doesn't even have a title! Let's fix that. We can pass a keyword argument of title. This will take a string.

```
df['Sex'].value_counts().plot.bar(title="Passengers on the Titanic")
```

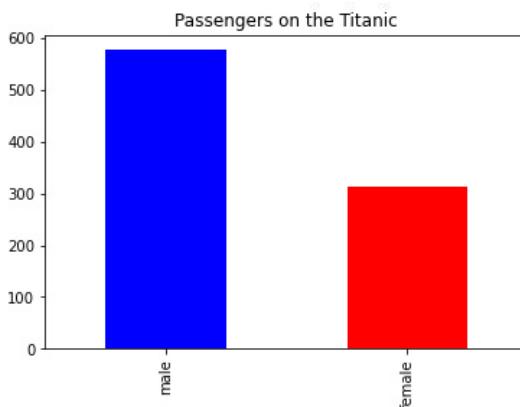
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b2e64cdf10>
```



We have another serious issue, though. Both types of gender are represented with the same color. This can be difficult for audiences to decipher in some instances, so let's change that. We can pass the keyword argument of color which will take a list of colors.

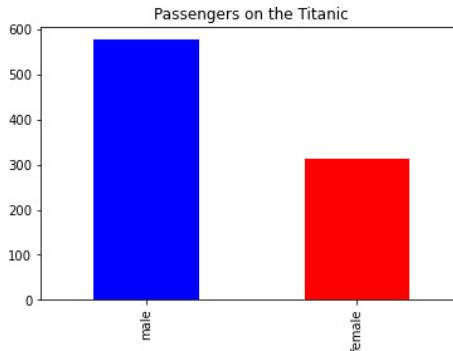
```
df['Sex'].value_counts().plot.bar(title="Passengers on the Titanic", color=["blue", "red"])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b289804040>
```



```
df['Sex'].value_counts().plot.bar(title="Passengers on the Titanic", color=["blue", "red"])
```

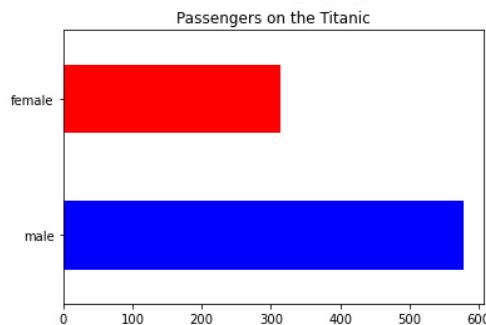
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28a8d52e0>
```



We can do the same thing with a barh graph, or a bar-horizontal graph.

```
df['Sex'].value_counts().plot.barh(title="Passengers on the Titanic", color=["blue", "red"])
```

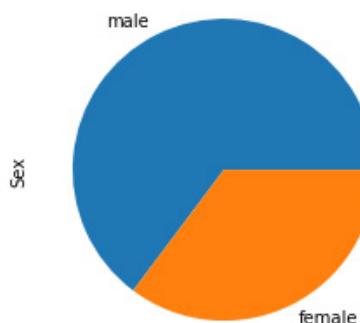
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28a93d460>
```



10.1.3 Pie Charts with Pandas

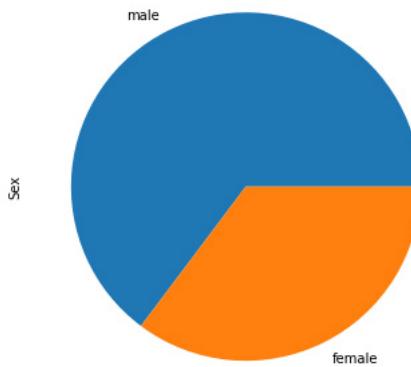
```
df['Sex'].value_counts().plot.pie()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28a990250>
```



```
df['Sex'].value_counts().plot.pie(figsize=(6, 6))
```

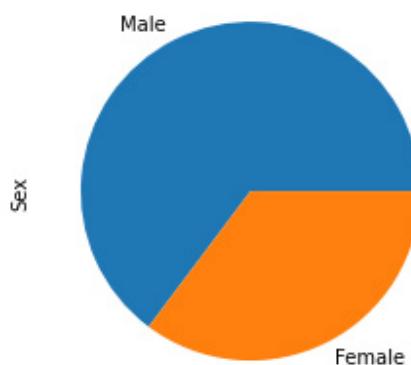
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28a9db490>
```



Let's say I was interested in the title of the genders not being lowercase. I can add in some custom labels to the data as a keyword argument, labels, which takes a list.

```
df['Sex'].value_counts().plot.pie(labels=["Male", "Female"])
```

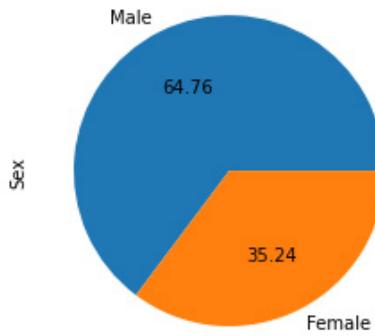
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28aa1b100>
```



Now that we have our labels as we want them, let's give the audience a bit of a better experience. Let's allow them to easily see the percentage of each gender, not just visually, but quantitatively. To do this, we can pass the keyword argument, autopct, which will take a string. In this case, we can pass in the argument "% .2f" which is a formatted string. This argument will convert our data into a percentage.

```
df['Sex'].value_counts().plot.pie(labels=["Male", "Female"], autopct="% .2f")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28aa0ecd0>
```



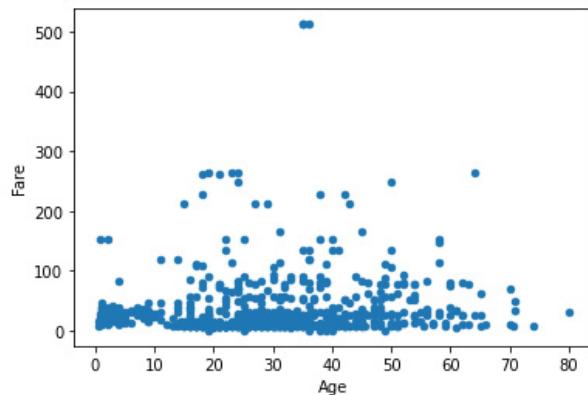
10.1.4 Scatter Plots with Pandas

Scatter plots allow us to plot qualitative data quantitatively in relation to two numerical attributes. Let's imagine that we are interested in exploring all passengers, something qualitative. Now, we want to know how each passenger relates to other passengers on two numerical, or quantitative attributes, e.g. the age of the passenger and the fare that they paid. Both of these are quantitative. We can therefore represent each person as a point on the scatter plot and plot them in relation to their fare (vertical, or y axis) and age (horizontal, or x axis) on the graph.

In Pandas we can do this by passing two keyword arguments, `x` and `y` and set them both equal to the DataFrame column we want, e.g. "Age" and "Fare".

```
df.plot.scatter(x="Age", y="Fare")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28a99f9a0>
```

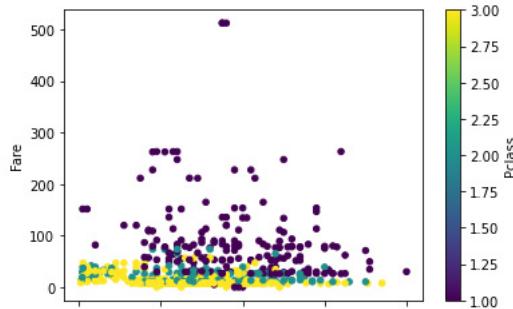


That looks good, but we can do better. Let's try to color coordinate this data. Let's say we are interested in seeing not only the passenger's age and fare, but we're also interested in color-coordinating the graph so that their "Pclass" effects the color of each plot. We can do this by passing a few new keyword arguments.

1. `c="Pclass"` => `c` will be the column that affects the color
2. `cmap="virdis"` => will be the color map we want to use (these are built into Pandas)

```
df.plot.scatter(x="Age", y="Fare", c="Pclass", cmap="viridis")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28aaaf2310>
```



This is starting to look a lot better now. But let's say we didn't want to represent our data as a series of marginally changing numbers. When we pass a DataFrame column to "c" as a set of numbers, Pandas presumes that that number corresponds to a gradient change in color. But passenger class is not a gradient change, it is an integral change, meaning no one will be Pclass 1.2. They will be 1, 2, or 3. In order to fix this graph, we can make a few changes. First, we can use `df.loc` that we met in a previous notebook to grab all classes. Now, we know there are three. We can convert these from numerical representations of the class into string representations, e.g. First, Second, and Third.

Next, we can convert that entire column from a string column into a Pandas Categorical Class.

```
df.loc[(df.Pclass == 1), 'Pclass']="First"
df.loc[(df.Pclass == 2), 'Pclass']="Second"
df.loc[(df.Pclass == 3), 'Pclass']="Third"
```

We can now see that our data has now been altered in the "Pclass" column.

```
df
```

	PassengerId	Survived	Pclass	\
0	1	0	Third	
1	2	1	First	
2	3	1	Third	
3	4	1	First	
4	5	0	Third	
..	
886	887	0	Second	
887	888	1	First	
888	889	0	Third	
889	890	1	First	
890	891	0	Third	
0	Braund, Mr. Owen Harris	male	22.0	1
1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	female	38.0	1 0
2	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1
3	Allen, Mr. William Henry	male	35.0	0
4

(continues on next page)

(continued from previous page)

```

886                         Montvila, Rev. Juozas    male  27.0    0
887                         Graham, Miss. Margaret Edith female 19.0    0
888                 Johnston, Miss. Catherine Helen "Carrie" female   NaN    1
889                         Behr, Mr. Karl Howell    male  26.0    0
890                         Dooley, Mr. Patrick    male  32.0    0

      Parch            Ticket       Fare Cabin Embarked
0         0          A/5 21171    7.2500   NaN      S
1         0           PC 17599   71.2833   C85      C
2         0      STON/O2. 3101282    7.9250   NaN      S
3         0           113803  53.1000  C123      S
4         0           373450   8.0500   NaN      S
...       ...
886        0          211536  13.0000   NaN      S
887        0          112053  30.0000  B42      S
888        2          W./C. 6607  23.4500   NaN      S
889        0          111369  30.0000  C148      C
890        0          370376   7.7500   NaN      Q

[891 rows x 12 columns]

```

Now that our data is successfully converted into a string, you might be thinking that we can run the same code as before and we should see the data divided between strings, rather than a gradient shift between floats. If we execute the cell below, however, we get a rather large and scary looking error. (Scroll down to see the solution.)

```
df.plot.scatter(x="Age", y="Fare", c="Pclass", cmap="viridis", s=50)
```

```

-----
ValueError                                Traceback (most recent call last)
File ~\anaconda3\lib\site-packages\matplotlib\axes\_axes.py:4239, in Axes._parse_
scatter_color_args(c, edgecolors, kwargs, xsize, get_next_color_func)
4238     try: # Is 'c' acceptable as PathCollection facecolors?
-> 4239         colors = mcolors.to_rgba_array(c)
4240     except ValueError:

File ~\anaconda3\lib\site-packages\matplotlib\colors.py:340, in to_rgba_array(c,_
alpha)
339     else:
--> 340         return np.array([to_rgba(cc, alpha) for cc in c])

File ~\anaconda3\lib\site-packages\matplotlib\colors.py:340, in <listcomp>(.0)
339     else:
--> 340         return np.array([to_rgba(cc, alpha) for cc in c])

File ~\anaconda3\lib\site-packages\matplotlib\colors.py:185, in to_rgba(c, alpha)
184     if rgba is None: # Suppress exception chaining of cache lookup failure.
--> 185         rgba = _to_rgba_no_colorcycle(c, alpha)
186     try:

File ~\anaconda3\lib\site-packages\matplotlib\colors.py:261, in _to_rgba_no_-
colorcycle(c, alpha)
260         return c, c, c, alpha if alpha is not None else 1.
--> 261     raise ValueError(f"Invalid RGBA argument: {orig_c!r}")
262 # tuple color.

ValueError: Invalid RGBA argument: 'Third'

```

(continues on next page)

(continued from previous page)

During handling of the above exception, another exception occurred:

```
ValueError                                     Traceback (most recent call last)
Input In [18], in <cell line: 1>()
----> 1 df.plot.scatter(x="Age", y="Fare", c="Pclass", cmap="viridis", s=50)

File ~\AppData\Roaming\Python\Python38\site-packages\pandas\plotting\_core.
→py:1636, in PlotAccessor.scatter(self, x, y, s, c, **kwargs)
    1553     def scatter(self, x, y, s=None, c=None, **kwargs):
    1554         """
    1555         Create a scatter plot with varying marker point size and color.
    1556
    (...)

    1634         ...
    1635         """
    1636     return self(kind="scatter", x=x, y=y, s=s, c=c, **kwargs)

File ~\AppData\Roaming\Python\Python38\site-packages\pandas\plotting\_core.
→py:917, in PlotAccessor.__call__(self, *args, **kwargs)
    915     if kind in self._dataframe_kinds:
    916         if isinstance(data, ABCDataFrame):
----> 917             return plot_backend.plot(data, x=x, y=y, kind=kind, **kwargs)
    918     else:
    919         raise ValueError(f"plot kind {kind} can only be used for data_
frames")

File ~\AppData\Roaming\Python\Python38\site-packages\pandas\plotting\_matplotlib\
→__init__.py:71, in plot(data, kind, **kwargs)
    69         kwargs["ax"] = getattr(ax, "left_ax", ax)
    70 plot_obj = PLOT_CLASSES[kind](data, **kwargs)
----> 71 plot_obj.generate()
    72 plot_obj.draw()
    73 return plot_obj.result

File ~\AppData\Roaming\Python\Python38\site-packages\pandas\plotting\_matplotlib\
→core.py:288, in MPLPlot.generate(self)
    286     self._compute_plot_data()
    287     self._setup_subplots()
----> 288     self._make_plot()
    289     self._add_table()
    290     self._make_legend()

File ~\AppData\Roaming\Python\Python38\site-packages\pandas\plotting\_matplotlib\
→core.py:1070, in ScatterPlot._make_plot(self)
    1068 else:
    1069     label = None
-> 1070     scatter = ax.scatter(
    1071         data[x].values,
    1072         data[y].values,
    1073         c=c_values,
    1074         label=label,
    1075         cmap=cmap,
    1076         norm=norm,
    1077         **self.kwds,
    1078     )
    1079 if cb:
    1080     cbar_label = c if c_is_column else ""
```

(continues on next page)

(continued from previous page)

```

File ~\anaconda3\lib\site-packages\matplotlib\__init__.py:1565, in _preprocess_
→data.<locals>.inner(ax, data, *args, **kwargs)
    1562 @functools.wraps(func)
    1563 def inner(ax, *args, data=None, **kwargs):
    1564     if data is None:
-> 1565         return func(ax, *map(sanitize_sequence, args), **kwargs)
    1567     bound = new_sig.bind(ax, *args, **kwargs)
    1568     auto_label = (bound.arguments.get(label_namer)
    1569                   or bound.kwargs.get(label_namer))

File ~\anaconda3\lib\site-packages\matplotlib\cbook\deprecation.py:358, in _  

→delete_parameter.<locals>.wrapper(*args, **kwargs)
    352 if name in arguments and arguments[name] != _deprecated_parameter:
    353     warn_deprecated(
    354         since, message=f"The {name!r} parameter of {func.__name__}() "
    355         f"is deprecated since Matplotlib {since} and will be removed "
    356         f"%({removal})s. If any parameter follows {name!r}, they "
    357         f"should be pass as keyword, not positionally.")
---> 358 return func(*args, **kwargs)

File ~\anaconda3\lib\site-packages\matplotlib\axes\_axes.py:4401, in Axes.  

→scatter(self, x, y, s, c, marker, cmap, norm, vmin, vmax, alpha, linewidths,  

→verts, edgecolors, plotnonfinite, **kwargs)
    4397 if len(s) not in (1, x.size):
    4398     raise ValueError("s must be a scalar, or the same size as x and y")
    4400 c, colors, edgecolors = \
-> 4401     self._parse_scatter_color_args(
    4402         c, edgecolors, kwargs, x.size,
    4403         get_next_color_func=self._get_patches_for_fill.get_next_color)
    4405 if plotnonfinite and colors is None:
    4406     c = np.ma.masked_invalid(c)

File ~\anaconda3\lib\site-packages\matplotlib\axes\_axes.py:4245, in Axes._parse_
→scatter_color_args(c, edgecolors, kwargs, xsize, get_next_color_func)
    4242     raise invalid_shape_exception(c.size, xsize)
    4243     # Both the mapping *and* the RGBA conversion failed: pretty
    4244     # severe failure => one may appreciate a verbose feedback.
-> 4245     raise ValueError(
    4246         f"'c' argument must be a color, a sequence of colors, or "
    4247         f"a sequence of numbers, not {c}")
    4248 else:
    4249     if len(colors) not in (0, 1, xsize):
    4250         # NB: remember that a single color is also acceptable.
    4251         # Besides *colors* will be an empty array if c == 'none'.

ValueError: 'c' argument must be a color, a sequence of colors, or a sequence of_  

→numbers, not ['Third' 'First' 'Third' 'First' 'Third' 'Third' 'First' 'Third'  

→'Third'
    'Second' 'Third' 'First' 'Third' 'Third' 'Second' 'Third'
    'Second' 'Third' 'Third' 'Second' 'Second' 'Third' 'First' 'Third'
    'Third' 'Third' 'First' 'Third' 'Third' 'First' 'First' 'Third'
    'First' 'First' 'Third' 'Third' 'Third' 'Third' 'Second' 'Third'
    'Second' 'Third' 'Third' 'Third' 'Third' 'Third' 'Third' 'Third'
    'Second' 'Third' 'Third' 'Third' 'Third' 'Third' 'Second' 'Third'
    'First' 'Second' 'First' 'First' 'Second' 'Third' 'Second' 'Third'
    'Third' 'First' 'First' 'Third' 'First' 'Third' 'Second' 'Third'
    'Third' 'Second' 'Third' 'Second' 'Third' 'Third' 'Third' 'Third'
    'Second' 'Third' 'Third' 'Third' 'First' 'Second' 'Third' 'Third'

(continues on next page)

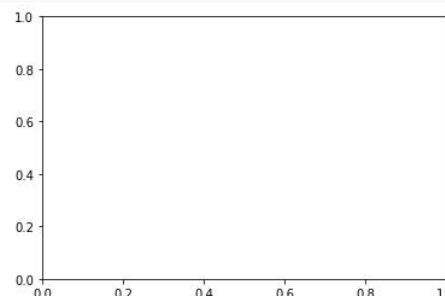
```

(continued from previous page)

(continues on next page)

(continued from previous page)

```
'Third' 'Third' 'Third' 'First' 'Third' 'Second' 'Third' 'Second'
'Third' 'Third' 'First' 'Second' 'Third' 'First' 'Third' 'First' 'Third'
'Third' 'First' 'Second' 'First' 'Third' 'Third' 'Third' 'Third' 'Third'
'Second' 'Third' 'Third' 'Second' 'Second' 'Third' 'First' 'Third'
'Third' 'Third' 'First' 'Second' 'First' 'Third' 'Third' 'First' 'Third'
'First' 'First' 'Third' 'Second' 'Third' 'Second' 'Third' 'Third' 'Third'
'First' 'Third' 'Third' 'Third' 'First' 'Third' 'First' 'Third' 'Third'
'Third' 'Second' 'Third' 'Third' 'Third' 'Second' 'Third' 'First' 'Third'
'Second' 'First' 'First' 'Third' 'First' 'Third' 'Third' 'Second'
'Second' 'Third' 'Third' 'First' 'Second' 'First' 'Second' 'Second'
'Second' 'Third' 'Third' 'Third' 'Third' 'First' 'Third' 'First' 'Third'
'Third' 'Second' 'Second' 'Third' 'Third' 'Third' 'First' 'First' 'Third'
'Third' 'Third' 'First' 'Second' 'Third' 'Third' 'First' 'Third' 'First'
'First' 'Third' 'Third' 'Second' 'First' 'Second' 'First' 'First' 'Third'
'First' 'First' 'First' 'Third' 'Second' 'Third' 'First' 'First' 'Second'
'Third' 'Second' 'Third' 'Second' 'Second' 'First' 'Third' 'Second'
'Third' 'Second' 'Third' 'First' 'Third' 'Second' 'Second' 'Second'
'Third' 'Third' 'First' 'Third' 'First' 'First' 'Third' 'Third' 'Third'
'Third' 'Third' 'Second' 'Third' 'Second' 'Third' 'Third' 'First' 'Third'
'First' 'Third' 'First' 'First' 'Third' 'First' 'First' 'Third' 'Third'
'Third' 'First' 'Third' 'Second' 'First' 'Second' 'First' 'Second' 'First'
'Third' 'First' 'Third' 'Second' 'Second' 'First' 'Second' 'First' 'First'
'Third' 'First' 'Third' 'Second' 'Second' 'First' 'First' 'Second' 'First'
'Third' 'First' 'Third' 'Second' 'Second' 'First' 'First' 'First' 'Second'
'First' 'Third' 'Second' 'First' 'Third' 'Third' 'Second' 'Third' 'Third'
'First' 'Third' 'Second' 'Third' 'Third' 'First' 'Third' 'First' 'Third'
'Third' 'First' 'Third' 'Second' 'Second' 'First' 'Second' 'First' 'Third'
'Third' 'First' 'Third' 'Second' 'Second' 'First' 'First' 'Second' 'First'
'Third' 'Second' 'First' 'Third' 'Second' 'Second' 'Second' 'Second' 'First'
'Third' 'Third' 'First' 'First' 'Third' 'Second' 'Third' 'Second' 'Third'
'Third' 'Third' 'First' 'Second' 'Third' 'First' 'Second' 'Third' 'First'
'Second' 'First' 'Third' 'First' 'First' 'Third']
```

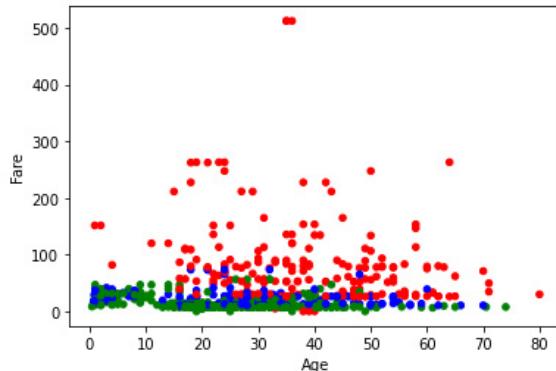


Keeping this massive error in the textbook is essential, despite its size being rather annoying. It tells us a lot of information about the problem. When we try and pass a keyword argument of “c”, Pandas is expecting a series of numbers (which will correspond to gradient shifts in the cmap), a list of colors, or a Pandas Categorical column. To change our data to a list of colors, let’s convert our data into three different colors.

```
df.loc[(df.Pclass == "First"), 'Pclass']="red"
df.loc[(df.Pclass == "Second"), 'Pclass']="blue"
df.loc[(df.Pclass == "Third"), 'Pclass']="green"
```

```
df.plot.scatter(x="Age", y="Fare", c="Pclass")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b286ac0c10>
```



Now, our plots are all color coordinated. But I don't like this. It doesn't have a nice ledger to read. Instead, we should convert this data into a Categorical Column. To do this, let's first get our data back into First, Second, and Third class format.

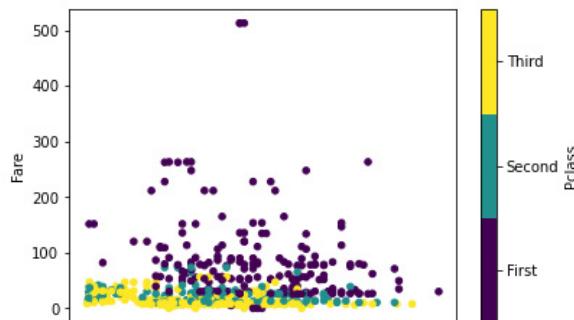
```
df.loc[(df.Pclass == "red"), 'Pclass']="First"
df.loc[(df.Pclass == "blue"), 'Pclass']="Second"
df.loc[(df.Pclass == "green"), 'Pclass']="Third"
```

Now, let's try this again by first converting Pclass into a Categorical type.

```
df['Pclass'] = df.Pclass.astype('category')
```

```
df.plot.scatter(x="Age", y="Fare", c="Pclass", cmap="viridis")
```

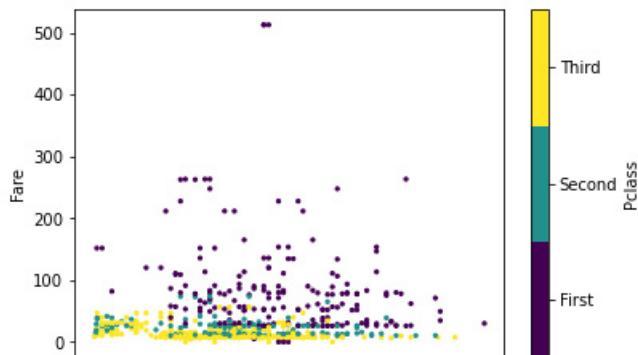
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28e20b4f0>
```



Now, like magic, we have precisely what we want to see. But we can do even better! Let's say we don't like the size of the nodes (points) on the graph. We want to see smaller nodes to distinguish better between the points. We can pass another keyword argument, "s", which stands for size. This expects an integer.

```
df.plot.scatter(x="Age", y="Fare", c="Pclass", cmap="viridis", s=5)
```

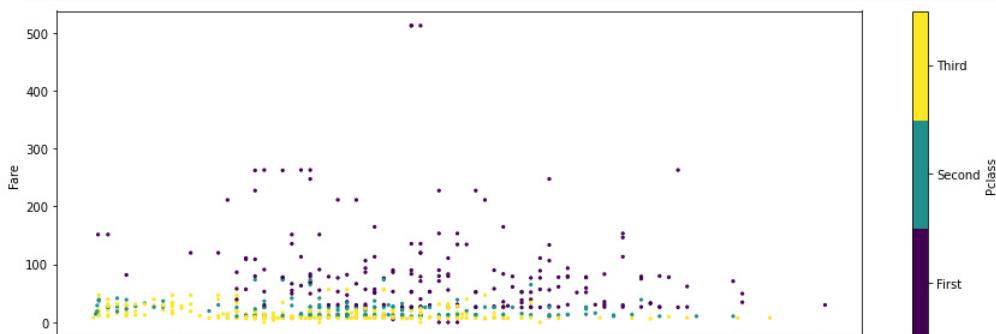
```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28f28f100>
```



To make it a bit easier to read, let's also adjust the size a bit. We can do this by passing the keyword argument, "figsize", that we saw above with pie charts.

```
df.plot.scatter(x="Age", y="Fare", c="Pclass", cmap="viridis", s=5, figsize=(15,5))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1b28f308340>
```



By now, you should have a good sense of how to create simple bar, pie, and scatter charts. In the next few notebooks, we will be looking at other ways of leveraging Pandas to produce visualizations, such as using plotly and social networks with NetworkX.

10.2 Graphing Network Data with Pandas

10.2.1 Getting the Data from Pandas to NetworkX

Pandas on its own cannot plot out network data. Instead, we must rely on two other libraries, NetworkX and Matplotlib. NetworkX is the standard Python library for working with networks. I have a forthcoming textbook, like this one, that walks users through NetworkX. Matplotlib is one of the standard plotting libraries. The purpose of this brief section, is to provide the code necessary for making Pandas work with NetworkX and for Matplotlib to take networks stored in a Pandas DataFrame and transform the relationships into graphs. We will address social networks in greater detail in Part IV of this textbook.

```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
```

Let's now load our data and see what it looks like.

```
df = pd.read_csv("data/network.csv")
```

```
df
```

```
   source  target
0      Tom     Rose
1     Rose   Rosita
2    Jerry     Jeff
3     Jeff    Larry
4  Carmen   Carmen
5  Rosita   Rosita
6   Larry   Carmen
7   Larry     Jerry
```

This is a pretty standard format for networks. We have two columns of data, a source, and a target. Imagine drawing a line to demonstrate networks, the source is where you start drawing the line and the target is where that line ends. This is known as force in network theory and is important for understanding the relationship between nodes, or individual points, in a network graph.

We can use NetworkX's built in function `from_pandas_edgelist()` and get that data straight into an edgelist.

```
G= nx.from_pandas_edgelist(df, "source", "target")
```

10.2.2 Graphing the Data

And with just two more lines of code we can plot that data out.

```
nx.draw(G)
plt.show()
```



10.2.3 Customize the Graph

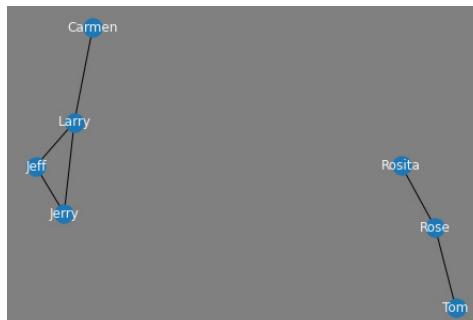
We have a problem with the image above, however, it is difficult to understand who the nodes represent. Let's give them some labels.

```
nx.draw(G, with_labels=True)
plt.show()
```



Now that we have labels, we need to make them a bit easier to read. We can do this by changing the font color to “whitesmoke” and setting the background to gray. To achieve this we first need to create a fig object to which we will append a few attributes. Next, we draw the network graph and give it a `font_color` of our desire. Finally, we set the `facecolor` to gray and plot it.

```
fig = plt.figure()
nx.draw(G, with_labels=True, font_color="whitesmoke")
fig.set_facecolor('gray')
plt.show()
```



What if I wanted each node in our network to have an individual color? We can do that too by setting up a color map.

```
val = []
for i in range(len(G.nodes)):
    val.append(i)
nx.set_node_attributes(G, val, 'val')
fig = plt.figure()
nx.draw(G, with_labels=True, node_color=val, font_color="whitesmoke")
fig.set_facecolor('gray')
plt.show()
```



10.3 Time Series Data

10.3.1 What Is Time Series Data

Time series data is data that reflects either time or dates. In Pandas this type of data is known as datetime. If you are working with time series data, as we shall see, there are significant reasons to ensure that Pandas understands that the data at hand is a date or a time. It allows for easily manipulation and cleaning of inconsistent data formatting. Let us consider a simple example. Imagine we were given dates from one source as 01/02/2002 and another as 01.02.2002. Both are valid date formats, but they are structured entirely differently. Imagine now you had a third dataset that organized the data as 2 January 2002. Your task is to merge all these datasets together.

If you wanted to do that, you could write out some Python and script them into alignment, but Pandas offers the ability to do that automatically. In order to leverage that ability, however, you must tell Pandas that the data at hand is datetime data. Exactly how you do that, we will learn in this section.

Time series data is important for many different aspects of industry and academia. In the financial sector, time series data allows for one to understand the past performance of a stock. This is particularly useful in machine learning predictions which need to understand the past to predict accurately the future. More importantly, they need to understand the past a sequence of data. In the humanities, time series data is important for understand historical context and, as we shall see, plotting data temporally. Understanding how to work with time series data, therefore, in Pandas is absolutely essential.

10.3.2 About the Dataset

In this section, we will be working with an early version of a dataset I helped cultivate at the *Bitter Aloe Project*, a digital humanities project that explores apartheid violence in South Africa during the 20th century. This dataset comes from Vol. 7 of the Truth and Reconciliation Commission's *Final Report*. I am using not our final, well-cleaned version of this dataset, rather an earlier version for one key reason. It contains problematic cells and structure. This is more reflective of real-world data, which will often times come from multiple sources and need to be cleaned and structured. As such, it is good practice in this section to try and address some of the common problems that you will encounter with time series data.

```
import pandas as pd
df = pd.read_csv("../data/trc.csv")
df
```

	ObjectId	Last	First	\
0	1	AARON	Thabo Simon	
1	2	ABBOTT	Montaigne	
2	3	ABRAHAM	Nzaliseko Christopher	
3	4	ABRAHAMS	Achmat Fardiel	
4	5	ABRAHAMS	Annalene Mildred	
...
20829	20888	XUZA	Mandla	
20830	20889	YAKA	Mbangomuni	
20831	20890	YALI	Khayalethu	
20832	20891	YALO	Bikiwe	

(continues on next page)

(continued from previous page)

			Geoffrey Yali				
			Description		Place	\	
0		An ANCYL member who was shot and severely inju...		Bethulie			
1		A member of the SADF who was severely injured ...		Messina			
2		A COSAS supporter who was kicked and beaten wi...		Mdantsane			
3		Was shot and blinded in one eye by members of ...		Athlone			
4		Was shot and injured by members of the SAP in ...		Robertson			
...					
20829		Was severely injured when he was stoned by a f...		Carletonville			
20830		An IFP supporter and acting induna who was sho...		Mvutshini			
20831		Was shot by members of the SAP in Lingelihle, ...		Cradock			
20832		An IFP supporter whose house and possessions w...		Port Shepstone			
20833		An ANC supporter and youth activist who was to...		George			
	Yr	Homeland	Province	Long	Lat	\	
0	1991.0	NaN	Orange Free State	25.97552	-30.503290		
1	1987.0	NaN	Transvaal	30.039597	-22.351308		
2	1985.0	Ciskei	Cape of Good Hope	27.6708791	-32.958623		
3	1985.0	NaN	Cape of Good Hope	18.50214	-33.967220		
4	1990.0	NaN	Cape of Good Hope	19.883611	-33.802220		
...	
20829	1991.0	NaN	Transvaal	27.397673	-26.360943		
20830	1993.0	KwaZulu	Natal	30.28172	-30.868900		
20831	1986.0	NaN	Cape of Good Hope	25.619176	-32.164221		
20832	1994.0	NaN	Natal	30.4297304	-30.752126		
20833	1986.0	NaN	Cape of Good Hope	22.459722	-33.964440		
		HRV		ORG			
0		shoot injure	ANC ANCYL Police SAP				
1		injure		SADF			
2		beat		COSAS Police			
3		shoot blind		SAP			
4		shoot injure		Police SAP			
...				
20829		injure stone		ANC			
20830		shoot		NaN			
20831		shoot		SAP			
20832		destroy		ANC			
20833		torture detain torture		ANC SAP			

[20834 rows x 12 columns]

As we can see, we have a few different columns which are relatively straight forward. In this notebook, however, I want to focus on "Yr", which is a column that contains a single year referenced within the description. This corresponds to the year in which the violence described occurred. Notice, however, that we have a problem. Year is being recognized as a float (a number with a decimal place), or floating number. To confirm our suspicion, let's take a look at the data types by using the following command.

```
display(df.dtypes)
```

ObjectId	int64
Last	object
First	object
Description	object

(continues on next page)

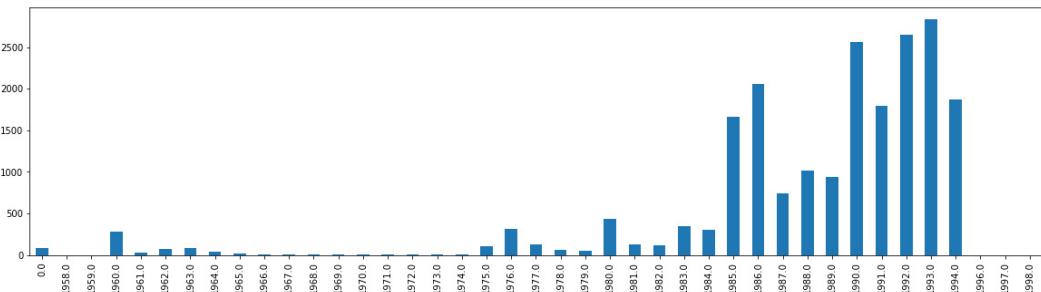
(continued from previous page)

```
Place          object
Yr            float64
Homeland       object
Province       object
Long           object
Lat            float64
HRV            object
ORG            object
dtype: object
```

Here, we can see all the different columns and their corresponding data types. Notice that "Yr" has "float64". This confirms our suspicion. Why is this a problem? Well, if we were to try and plot the data by year (see the bar graph below), we would have floating numbers in that graph. This does not look clean. We could manually adjust these years to have no decimal place, but that requires effort on a case-by-case basis. Instead, it is best practice to convert these floats either to integers or to datetime data. Both have their advantages, but if your end goal is larger data analysis on time series data (not just plotting the years), I would opt for the latter. In order to do either, however, we must clean the data to get it into the correct format.

```
df['Yr'].value_counts().sort_index().plot.bar(figsize=(20,5))
```

<AxesSubplot :>



10.3.3 Cleaning the Data from Float to Int

Let's first try and convert our float column into an integer column. If we execute the command below which would normally achieve this task, we get the following error.

```
df['Yr'] = df['Yr'].astype(int)
```

```
-----
IntCastingNaNError                                     Traceback (most recent call last)
<ipython-input-4-dc1db5c67903> in <module>
----> 1 df['Yr'] = df['Yr'].astype(int)

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    \core\generic.py in astype(self, dtype, copy, errors)
        5804         else:
        5805             # else, only a single dtype is given
        5806             new_data = self._mgr.astype(dtype=dtype, copy=copy,_
    <errors=errors>
        5807             return self._constructor(new_data).__finalize__(self, method=
    <continues on next page>
```

(continued from previous page)

```

5808

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\internals\managers.py in astype(self, dtype, copy, errors)
        412
        413     def astype(self: T, dtype, copy: bool = False, errors: str = "raise"
    ↪") -> T:
    --> 414         return self.apply("astype", dtype=dtype, copy=copy,-
    ↪errors=errors)
        415
        416     def convert(


c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\internals\managers.py in apply(self, f, align_keys, ignore_failures,-
    ↪**kwargs)
        325             applied = b.apply(f, **kwargs)
        326         else:
    --> 327             applied = getattr(b, f)(**kwargs)
        328         except (TypeError, NotImplementedError):
        329             if not ignore_failures:

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\internals\blocks.py in astype(self, dtype, copy, errors)
        590         values = self.values
        591
    --> 592         new_values = astype_array_safe(values, dtype, copy=copy,-
    ↪errors=errors)
        593
        594         new_values = maybe_coerce_values(new_values)

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\dtypes\cast.py in astype_array_safe(values, dtype, copy, errors)
        1307
        1308     try:
    -> 1309         new_values = astype_array(values, dtype, copy=copy)
        1310     except (ValueError, TypeError):
        1311         # e.g. astype_nansafe can fail on object-dtype of strings

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\dtypes\cast.py in astype_array(values, dtype, copy)
        1255
        1256     else:
    -> 1257         values = astype_nansafe(values, dtype, copy=copy)
        1258
        1259     # in pandas we don't store numpy str dtypes, so convert to object

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\dtypes\cast.py in astype_nansafe(arr, dtype, copy, skipna)
        1166
        1167     elif np.issubdtype(arr.dtype, np.floating) and np.issubdtype(dtype,-
    ↪np.integer):
    -> 1168         return astype_float_to_int_nansafe(arr, dtype, copy)
        1169
        1170     elif is_object_dtype(arr):


c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    ↪core\dtypes\cast.py in astype_float_to_int_nansafe(values, dtype, copy)
        1211     """

```

(continues on next page)

(continued from previous page)

```
1212     if not np.isfinite(values).all():
-> 1213         raise IntCastingNaNErr(
1214             "Cannot convert non-finite values (NA or inf) to integer"
1215         )
```

IntCastingNaNErr: Cannot convert non-finite values (NA or inf) to integer

At the very bottom, we see why the error was returned. “IntCastingNaNErr: Cannot convert non-finite values (NA or inf) to integer”. This means that somewhere in our data, there are a few blank cells in the “Year” column. We need to fill in these blank cells. To do that, we can use the `fillna()` function.

```
df = df.fillna(0)
```

If we try and rerun our same command as above, you will notice we have no errors.

```
df['Yr'] = df['Yr'].astype(int)
```

Now, let's see if it worked by displaying the data types again.

```
display(df.dtypes)
```

```
ObjectId      int64
Last          object
First          object
Description   object
Place          object
Yr            int32
Homeland       object
Province       object
Long           object
Lat            float64
HRV            object
ORG            object
dtype: object
```

Notice that “Yr” is now “int32”. Success! Now that we have the data in the correct format, let's plot it out. We can plot out the frequency of violence based on year by using value counts. This will go through the entire “Yr” column and count all the values identified and store them as a dictionary of frequencies.

```
df['Yr'].value_counts()
```

1993	2835
1992	2648
1990	2556
1986	2056
1994	1867
1991	1793
1985	1665
1988	1015
1989	935
1987	744
1980	438
1983	352
1976	319

(continues on next page)

(continued from previous page)

```

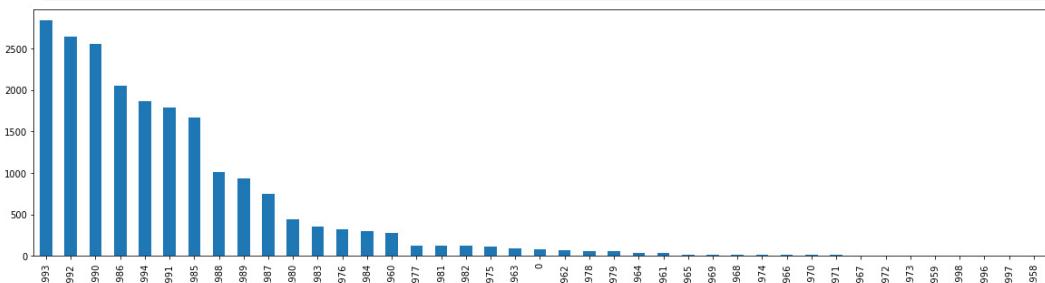
1984      301
1960      280
1977      128
1981      124
1982      123
1975      111
1963       88
0          84
1962       69
1978       60
1979       53
1964       37
1961       32
1965       19
1969       14
1968       14
1974       12
1966       11
1970       10
1971       10
1967        8
1972        6
1973        5
1959        3
1998        3
1996        3
1997        2
1958        1
Name: Yr, dtype: int64

```

This looks great, but let's try and plot it.

```
df['Yr'].value_counts().plot.bar(figsize=(20,5))
```

<AxesSubplot :>



What do you notice that is horribly wrong about our bar graph? If you noticed that it is not chronological, you'd be right. It would be quite odd to present our data in this format. When we are examining time series data, we need to visualize that data chronologically (usually). We can fix this, by adding `sort_index()`.

```
df['Yr'].value_counts().sort_index()
```

Year	Count
0	84
1958	1

(continues on next page)

(continued from previous page)

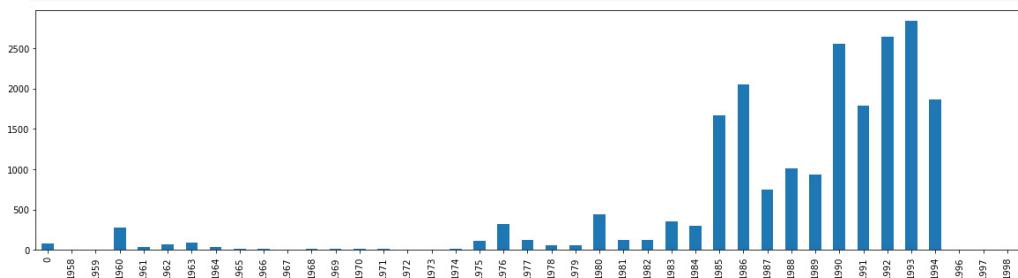
1959	3
1960	280
1961	32
1962	69
1963	88
1964	37
1965	19
1966	11
1967	8
1968	14
1969	14
1970	10
1971	10
1972	6
1973	5
1974	12
1975	111
1976	319
1977	128
1978	60
1979	53
1980	438
1981	124
1982	123
1983	352
1984	301
1985	1665
1986	2056
1987	744
1988	1015
1989	935
1990	2556
1991	1793
1992	2648
1993	2835
1994	1867
1996	3
1997	2
1998	3

Name: Yr, dtype: int64

Notice that we have now preserved the value counts, but organized them in their correct order. We can now try plotting that data.

```
df['Yr'].value_counts().sort_index().plot.bar(figsize=(20,5))
```

<AxesSubplot:>



We have a potential issue, however. That first row, 0, is throwing off our bar graph. What if I didn't want to represent 0, or no date, in the graph. I can solve this problem a few different ways. Let's first create a new dataframe called `val_year`.

```
val_year = df["Yr"].value_counts().sort_index()
val_year
```

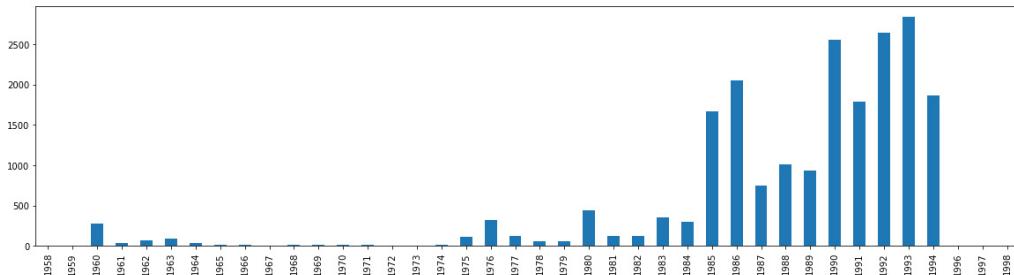
0	84
1958	1
1959	3
1960	280
1961	32
1962	69
1963	88
1964	37
1965	19
1966	11
1967	8
1968	14
1969	14
1970	10
1971	10
1972	6
1973	5
1974	12
1975	111
1976	319
1977	128
1978	60
1979	53
1980	438
1981	124
1982	123
1983	352
1984	301
1985	1665
1986	2056
1987	744
1988	1015
1989	935
1990	2556
1991	1793
1992	2648
1993	2835
1994	1867
1996	3
1997	2
1998	3

Name: Yr, dtype: int64

With this new dataframe, I can simply start at index 1 and then graph the data. Notice that the 0 value is now gone.

```
val_year.iloc[1:].plot.bar(figsize=(20,5))
```

<AxesSubplot:>



Although we have been able to now plot our time series data chronologically, Pandas has not seen this as a datetime type. Instead, it has viewed these years solely as integers. In order to work with the years as time series data formally, we need to convert the integers into datetime format.

10.3.4 Convert to Time Series DateTime in Pandas

Our goal here will be to create a new column that will store "Yr" as a datetime type. One might think that we could easily just convert everything to datetime. Normally the following command would work, but instead we get this error.

```
df['Dates'] = pd.to_datetime(df['Yr'], format='%Y')
```

```
-----
TypeError                                         Traceback (most recent call last)
c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    \core\tools\datetimes.py in _to_datetime_with_format(arg, orig_arg, name, tz,
    \fmt, exact, errors, infer_datetime_format)
      508     try:
--> 509         values, tz = conversion.datetime_to_datetime64(arg)
      510         dta = DatetimeArray(values, dtype=tz_to_dtype(tz))

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    \libs\tslibs\conversion.pyx in pandas._libs.tslibs.conversion.datetime_to_
    \datetime64()

TypeError: Unrecognized value type: <class 'int'>

During handling of the above exception, another exception occurred:

ValueError                                         Traceback (most recent call last)
<ipython-input-14-cc2c68a810bf> in <module>
----> 1 df['Dates'] = pd.to_datetime(df['Yr'], format='%Y')

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    \core\tools\datetimes.py in to_datetime(arg, errors, dayfirst, yearfirst, utc,
    \format, exact, unit, infer_datetime_format, origin, cache)
     881     result = result.tz_localize(tz) # type: ignore[call-arg]
     882     elif isinstance(arg, ABCSeries):
--> 883         cache_array = _maybe_cache(arg, format, cache, convert_listlike)
     884         if not cache_array.empty:
     885             result = arg.map(cache_array)

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
    \core\tools\datetimes.py in _maybe_cache(arg, format, cache, convert_listlike)
   193     unique_dates = unique(arg)
```

(continues on next page)

(continued from previous page)

```

194         if len(unique_dates) < len(arg):
--> 195             cache_dates = convert_listlike(unique_dates, format)
196             cache_array = Series(cache_dates, index=unique_dates)
197             # GH#39882 and GH#35888 in case of None and NaT we get_
→duplicates

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
→core\tools\datetimes.py in _convert_listlike_datetimes(arg, format, name, tz,
→unit, errors, infer_datetime_format, dayfirst, yearfirst, exact)
391
392     if format is not None:
--> 393         res = _to_datetime_with_format(
394             arg, orig_arg, name, tz, format, exact, errors, infer_
→datetime_format
395         )

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
→core\tools\datetimes.py in _to_datetime_with_format(arg, orig_arg, name, tz,
→fmt, exact, errors, infer_datetime_format)
511         return DatetimeIndex._simple_new(dta, name=name)
512     except (ValueError, TypeError):
--> 513         raise err
514
515

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
→core\tools\datetimes.py in _to_datetime_with_format(arg, orig_arg, name, tz,
→fmt, exact, errors, infer_datetime_format)
498
499     # fallback
--> 500     res = _array_strptime_with_fallback(
501         arg, name, tz, fmt, exact, errors, infer_datetime_format
502     )

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
→core\tools\datetimes.py in _array_strptime_with_fallback(arg, name, tz, fmt,
→exact, errors, infer_datetime_format)
434
435     try:
--> 436         result, timezones = array_strptime(arg, fmt, exact=exact,
→errors=errors)
437         if "%Z" in fmt or "%z" in fmt:
438             return _return_parsed_timezone_results(result, timezones, tz,
→name)

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\pandas\
→libs\tslibs\strptime.pyx in pandas._libs.tslibs.strptime.array_strptime()

ValueError: time data '0' does not match format '%Y' (match)

```

Just as the NaN cells plagued us before, so too has the 0s that we filled them with. Fortunately, we can fix this issue by passing the keyword argument `errors="coerce"`.

```
df['Dates'] = pd.to_datetime(df['Yr'], format='%Y', errors="coerce")
```

```
display(df.dtypes)
```

```

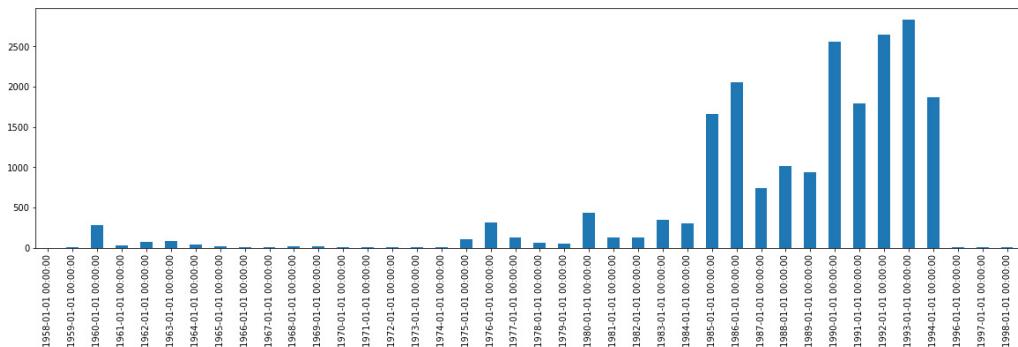
ObjectId          int64
Last             object
First            object
Description      object
Place            object
Yr               int32
Homeland         object
Province         object
Long              object
Lat              float64
HRV              object
ORG              object
Dates           datetime64[ns]
dtype: object

```

And like magic, we have not only created a new column, but notice that it is in `datetime64 [ns]` format. We should also understand the keyword argument passed here, `format`. Format takes a formatted string that will tell Pandas how to interpret the data being passed to it. Because our integer referred to a single year, we use “%Y”. Let’s try and plot this data now to see how it looks.

```
df['Dates'].value_counts().sort_index().plot.bar(figsize=(20,5))
```

`<AxesSubplot:>`



While this data is now plotted as Pandas-structured time series data, it does not look good. Our dates are rendered in the long, full format that has both the date (in its entirety) and the time. Let’s fix this by first, extracting the relevant data. In this case, the year and the counts.

```
new_df = df['Dates'].value_counts().sort_index()
new_df
```

1958-01-01	1
1959-01-01	3
1960-01-01	280
1961-01-01	32
1962-01-01	69
1963-01-01	88
1964-01-01	37
1965-01-01	19
1966-01-01	11
1967-01-01	8

(continues on next page)

(continued from previous page)

```

1968-01-01      14
1969-01-01      14
1970-01-01      10
1971-01-01      10
1972-01-01       6
1973-01-01       5
1974-01-01      12
1975-01-01     111
1976-01-01     319
1977-01-01     128
1978-01-01      60
1979-01-01      53
1980-01-01    438
1981-01-01    124
1982-01-01    123
1983-01-01   352
1984-01-01   301
1985-01-01  1665
1986-01-01  2056
1987-01-01   744
1988-01-01  1015
1989-01-01   935
1990-01-01  2556
1991-01-01  1793
1992-01-01  2648
1993-01-01  2835
1994-01-01  1867
1996-01-01      3
1997-01-01      2
1998-01-01      3
Name: Dates, dtype: int64

```

Next, we need to convert that data into a new DataFrame.

```

new_df = pd.DataFrame(new_df)
new_df.head()

```

```

          Dates
1958-01-01      1
1959-01-01      3
1960-01-01    280
1961-01-01     32
1962-01-01     69

```

Now that we have that new DataFrame created, let's fix our column name and change Dates to "ViolentActs".

```

new_df = new_df.rename(columns={"Dates": "ViolentActs"})
new_df.head()

```

```

          ViolentActs
1958-01-01      1
1959-01-01      3
1960-01-01    280
1961-01-01     32
1962-01-01     69

```

With the new DataFrame, we can also fix the index so that it is strictly the year. Because Pandas knows that the index is a datetime type, then we can use the extra method, `year`, to grab just the year.

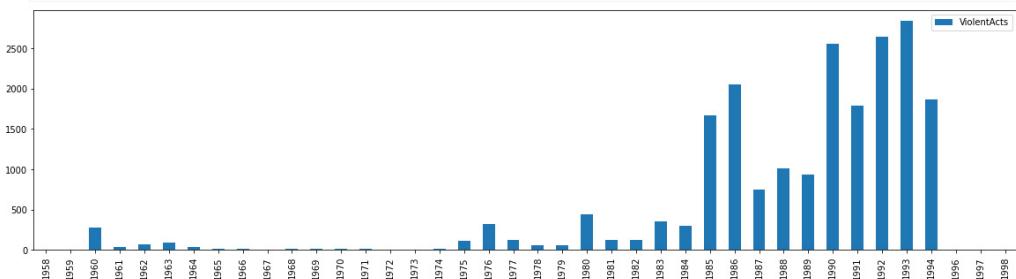
```
new_df.index = new_df.index.year  
new_df.head()
```

```
ViolentActs  
1958      1  
1959      3  
1960    280  
1961     32  
1962     69
```

Notice that our data is now just the year, the only piece of data in the time series data that matters to us. With that new DataFrame in the correct format, we can now plot it.

```
new_df.plot.bar(figsize=(20, 5))
```

```
<AxesSubplot :>
```



And thus we have successfully plotted our datetime data after properly formatting it in Pandas. While working with time series data in Pandas as a datetime can be a bit more complex in the beginning, it allows for you to more advanced things, such as we saw above by calling the year with `.year`. As we will see in the next few chapters, there are other advantages as well.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

Part III

Natural Language Processing with spaCy



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

11

Introduction to Spacy

In this section, we will begin exploring natural language processing, or NLP, and the Python library spaCy. spaCy is a framework for performing NLP via Python. Throughout this section, you will learn the basics of spaCy, from the key terminology and concepts, to processing texts via spaCy.

11.1 The Basics of spaCy

In this part of the textbook, we will learn how to do natural language processing (NLP) with spaCy. This section will approach spaCy top-down. In this way, we will gain an understanding of NLP generally and spaCy's mechanics specifically. By learning these concepts first, it will make it easier to approach spaCy in code.

11.1.1 What Is spaCy?

spaCy (yes, spelled with a lowercase "s" and uppercase "C") is a natural language processing framework. **Natural language processing**, or NLP, is a branch of linguistics that seeks to parse human language in a computer system. This field is generally referred to as computational linguistics, though it has far reaching applications beyond academic linguistic research.

NLP is used in every sector of industry, from academics who leverage it to aid in research to financial analysts who try and predict the stock market. Lawyers use NLP to help analyze thousands of legal documents in seconds to target their research and medical doctors use it to parse patient charts. NLP has been around for decades, but with the increasing developments in deep learning, a subfield of machine learning, NLP has rapidly expanded. This is because, as we shall learn all too well throughout this book, language is inherently ambiguous and complex. By this, I mean that language does not always make perfect sense. In some cases, it is entirely illogical. The double-negative in English is a good example of this. In some contexts, it can be a positive, as in, "I don't not like pasta." In other cases, the double negative can be an emphatic negative, as in, "I ain't not doing that!"

As humans, especially native speakers of a language, we can parse these complex illogical statements with ease, especially with enough context. For computers, this is not always easy.

Because NLP is such a complex problem for computers, it requires a complex solution. The answer has been found in artificial neural networks, or ANNs (neural nets for short). These are the primary areas of research for deep learning practitioners. As the field of deep learning (and machine learning in general) expand and advance, so too does NLP. New methods for training, such as transformer models, push the field further.

11.1.2 How to Install spaCy

In order to install spaCy, I recommend visiting their website: <https://spacy.io/usage>. They have a nice user-friendly interface. Input your device settings, e.g. Mac, Windows, or Linux, and your language, e.g. English, French, or German. The web-app will automatically populate the commands that you need to execute to get started. Since this is a JupyterBook, we can install these with a “!” in a cell to indicate that we want to run a terminal command. I will be installing spaCy with the small English model, `en_core_web_sm`.

```
!pip install spacy
```

```
!python -m spacy download en_core_web_sm
```

Now that we've installed spaCy let's import it to make sure we installed it correctly.

```
import spacy
```

Great! Now, let's make sure we downloaded the model successfully with the command below.

```
nlp = spacy.load("en_core_web_sm")
```

Excellent! spaCy is now installed correctly and we have successfully downloaded the small English model. We will pick up here with the code in the next notebook. For now, I want to focus on big-picture items, specifically spaCy “containers”.

11.1.3 Containers

Containers are spaCy objects that contain a large quantity of data about a text. When we analyze texts with the spaCy framework, we create different container objects to do that. Here is a full list of all spaCy containers. We will be focusing on three (emboldened): Doc, Span, and Token.

- Doc
- DocBin
- Example
- Language
- Lexeme
- Span
- SpanGroup
- Token

I created the image below to show how I visualize spaCy containers in my mind. At the top, we have a Doc container. This is the basis for all spaCy. It is the main object that we create. Within the Doc container are many different attributes and subcontainers. One attribute is the `Doc.sents`, which contains all the sentences in the Doc container. The doc container (and each sentence generator) is made up of a set of token containers. These are things like words, punctuation, etc.

Span containers are kind of like a token, in that they are a piece of a Doc container. Spans have one thing that makes them unique. They can cross multiple tokens.

We can give spans a bit more specificity by classifying them into different groups. These are known as SpanGroup containers.

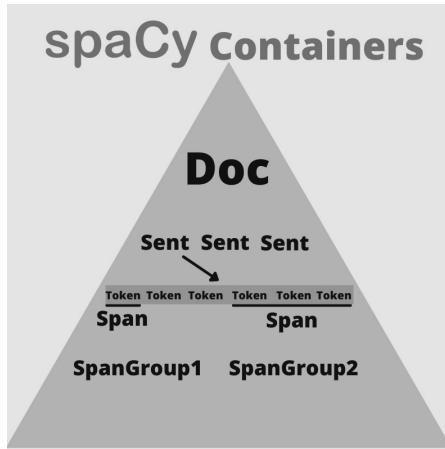


FIGURE 11.1
spaCy Container Structure.

If you do not fully understand this dynamic, do not worry. You will get a much better sense of this pyramid as we move forward throughout this section. For now, I recommend keeping this image handy so you can refer back to it as we progress through this section in which we explore the basics of spaCy. In the next section, we will start applying these concepts in code by creating a doc object and learning about the different attributes containers have as well as how to find the linguistic annotations.

11.2 Getting Started with spaCy and Its Linguistic Annotations

The goal of this section are twofold. First, it is my hope that you understand the basic spaCy syntax for creating a Doc container and how to call specific attributes of that container. Second, it is my hope that you leave this section with a basic understanding of the vast linguistic annotations available in spaCy. While we will not explore all attributes, we will deal with many of the most important ones, such as lemmas, parts-of-speech, and named entities. By the time you are finished with this section, you should have enough of a basic understanding of spaCy to begin applying it to your own texts.

11.2.1 Importing spaCy and Loading Data

As with all Python libraries, the first thing we need to do is import spaCy. In the last notebook, I walked you through how to install it and download the small English model. If you have followed those steps, you should be able to import it like so:

```
import spacy
```

With spaCy imported, we can now create our `nlp` object. This is the standard Pythonic way to create your model in a Python script. Unless you are working with multiple models

in a script, try to always name your model, `nlp`. It will make your script much easier to read. To do this, we will use `spacy.load()`. This command tells spaCy to load up a model. In order to know which model to load, it needs a string argument that corresponds to the model name. Since we will be working with the small English model, we will use “`en_core_web_sm`”. This function can take keyword arguments to identify which parts of the model you want to load, but we will get to that later. For now, we want to import the whole thing.

```
nlp = spacy.load("en_core_web_sm")
```

Great! With the model loaded, let’s go ahead and import our text. For this section, we will be working with the opening description from the Wikipedia article on the United States. In this repository, it is found in the subfolder `data` and is entitled `wiki_us.txt`.

```
with open("../data/wiki_us.txt", "r") as f:
    text = f.read()
```

Now, let’s see what this text looks like. It can be a bit difficult to read in a JupyterBook, but notice the horizontal slider below. You don’t need to read this in its entirety.

```
print(text)
```

The United States of America (U.S.A. or USA), commonly known as the United States (U.S. or US) or America, is a country located in North America. It consists of 50 states, a federal district, five major unincorporated territories, nine Minor Outlying Islands, and 326 Indian reservations. It is the third-largest country by both land and total area. The United States shares land borders with Canada to its north and with Mexico to its south. It has maritime borders with the Bahamas, Cuba, Russia, and other nations. With a population of over 331 million, it is the third most populous country in the world. The national capital is Washington, D.C., and the most populous city and financial center is New York City.

11.2.2 Creating a Doc Container

With the data loaded in, it’s time to make our first Doc container. Unless you are working with multiple Doc containers, it is best practice to always call this object “`doc`”, all lowercase. To create a Doc container, we will usually just call our `nlp` object and pass our text to it as a single argument.

```
doc = nlp(text)
```

Great! Let’s see what this looks like.

```
print(doc)
```

The United States of America (U.S.A. or USA), commonly known as the United States (U.S. or US) or America, is a country located in North America. It consists of 50 states, a federal district, five major unincorporated territories, nine Minor Outlying Islands, and 326 Indian reservations. It is the third-largest country by both land and total area. The United States shares land borders with Canada to its north and with Mexico to its south. It has maritime borders with the Bahamas, Cuba, Russia, and other nations. With a population of over 331 million, it is the third most populous country in the world. The national capital is Washington, D.C., and the most populous city and financial center is New York City.

If you are trying to spot the difference between this and the text above, good luck. You will not see a difference when printing off the Doc container. But I promise you, it is quite different behind the scenes. The Doc container, unlike the text object, contains a lot of valuable metadata, or attributes, hidden behind it. To prove this, let's examine the length of the Doc object and the text object.

```
print (len(doc))
print (len(text))
```

```
146
716
```

What's going on here? It is the same text, but different length. Why does this occur? To answer that, let's explore it more deeply and try and print off each item in each object.

```
for token in text[:10]:
    print (token)
```

```
T
h
e

U
n
i
t
e
d
```

As we would expect. We have printed off each character, including white spaces. Let's try and do the same with the Doc container.

```
for token in doc[:10]:
    print (token)
```

```
The
United
States
of
America
(
U.S.A.
or
USA
)
```

And now we see the magical difference. While on the surface it may seem that the Doc container's length is dependent on the quantity of words, look more closely. You should notice that the open and close parentheses are also considered an item in the container. These are all known as tokens. **Tokens** are a fundamental building block of spaCy or any NLP framework. They can be words or punctuation marks. Tokens are something that has syntactic purpose in a sentence and is self-contained. A good example of this is the contraction "don't" in English. When tokenized, or the process of converting the text into tokens, we will have two tokens. "do" and "n't" because the contraction represents two words, "do" and "not".

On the surface, this may not seem exceptional. But it is. You may be thinking to yourself that you could easily use the split method in Python to split by whitespace and have the same result. But you'd be wrong. Let's see why.

```
for token in text.split()[:10]:
    print(token)
```

```
The
United
States
of
America
(U.S.A.
or
USA),
commonly
known
```

Notice that the parentheses are not removed or handled individually. To see this more clearly, let's print off all tokens from index 5 to 8 in both the text and Doc objects.

```
words = text.split()[:10]
```

```
i=5
for token in doc[i:8]:
    print(f"SpaCy Token {i}:\n{token}\nWord Split {i}:\n{words[i]}\n\n")
    i=i+1
```

```
SpaCy Token 5:
(
Word Split 5:
(U.S.A.

SpaCy Token 6:
U.S.A.
Word Split 6:
or

SpaCy Token 7:
or
Word Split 7:
USA),
```

We can see clearly now how the spaCy Doc container does much more with its tokenization than a simple split method. We could, surely, write complex rules for a language to achieve the same results, but why bother? spaCy does it exceptionally well for all languages. In my entire time using spaCy, I have never seen the tokenizer make a mistake. I am sure that mistakes may occur, but these are probably rare exceptions.

Let's see what else this Doc container holds.

11.2.3 Sentence Boundary Detection (SBD)

In NLP, sentence boundary detection, or SBD, is the identification of sentences in a text. Again, this may seem fairly easy to do with rules. One could use `split("."),` but in

English we use the period to also denote abbreviation. You could, again, write rules to look for periods not proceeded by a lowercase word, but again, I ask the question, “why bother?”. We can use spaCy and in seconds have all sentences fully separated through SBD.

To access the sentences in the Doc container, we can use the attribute `sents`, like so:

```
for sent in doc.sents:
    print (sent)
```

The United States of America (U.S.A. or USA), commonly known as the United States (U.S. or US) or America, is a country located in North America. It consists of 50 states, a federal district, five major unincorporated territories, nine Minor Outlying Islands, and 326 Indian reservations. It is the third-largest country by both land and total area. The United States shares land borders with Canada to its north and with Mexico to its south. It has maritime borders with the Bahamas, Cuba, Russia, and other nations. With a population of over 331 million, it is the third most populous country in the world. The national capital is Washington, D.C., and the most populous city and financial center is New York City.

We got an error. That is because the `sents` attribute is a `Span` container that is stored as a generator. Generators are beyond the scope of this textbook, but in short generators will yield results from functions rather than return results. While you can iterate over a generator like a list, they are not the same thing. Instead of populating all results in memory, a generator lets us only load one result at a time. We can use `next` to grab the first sentence.

```
sentence1 = next(doc.sents)
print(sentence1)
```

The United States of America (U.S.A. or USA), commonly known as the United States (U.S. or US) or America, is a country located in North America.

Now we have the first sentence. Now that we have a smaller text, let's explore spaCy's other building block, the token.

11.2.4 Token Attributes

The token object contains a lot of different attributes that are *vital* to performing NLP in spaCy. We will be working with a few of them, such as:

Name	Description
<code>sent</code>	the sentence to which the span belongs
<code>text</code>	the raw text of the token
<code>head</code>	the parent of the token
<code>left_edge</code>	the left edge of the token's descendants
<code>right_edge</code>	the right edge of the token's descendants
<code>ent_type_</code>	the label of the token, if it is an entity
<code>lemma_</code>	the lemmatized form of the token
<code>morph</code>	provides the morphology of the token
<code>pos_</code>	part of speech
<code>dep_</code>	dependency relation
<code>lang_</code>	the language of the parent document

I will briefly describe these here and show you how to grab each one and what they look like. We will be exploring each of these attributes more deeply in this section and future sections. To demonstrate each of these attributes, we will use one token, “States” which is part of a sequence of tokens that make up “The United States of America”

```
token2 = sentence1[2]
print (token2)
```

States

11.2.4.1 Text

Verbatim text content. -spaCy docs

```
token2.text
```

'States'

11.2.4.2 Head

The syntactic parent, or “governor”, of this token. -spaCy docs

```
token2.head
```

is

This tells which word it is governed by, in this case, the primary verb, “is”, as it is part of the noun subject.

11.2.4.3 Left Edge

The leftmost token of this token’s syntactic descendants. -spaCy docs

```
token2.left_edge
```

The

If part of a sequence of tokens that are collectively meaningful, known as **multi-word tokens**, this will tell us where the multi-word token begins.

11.2.4.4 Right Edge

The rightmost token of this token’s syntactic descendants. -spaCy docs

```
token2.right_edge
```

,

This will tell us where the multi-word token ends.

11.2.4.5 Entity Type

Named entity type. -spaCy docs

```
token2.ent_type
```

```
384
```

Note the absence of the “_” at the end of the attribute. This will return an integer that corresponds to an entity type, where “_” will give you the string equivalent, as below.

```
token2.ent_type_
```

```
'GPE'
```

We will learn all about types of entities in the chapter on Named Entity Recognition, or NER. For now, simply understand that GPE is geopolitical entity and is correct.

11.2.4.6 Ent IOB

IOB code of named entity tag. “B” means the token begins an entity, “I” means it is inside an entity, “O” means it is outside an entity, and “” means no entity tag is set.

```
token2.ent_iob
```

```
'I'
```

IOB is a method of annotating a text. In this case, we see “I” because states is inside an entity, that is to say that it is part of the United States of America.

11.2.4.7 Lemma

Base form of the token, with no inflectional suffixes. -spaCy docs

```
token2.lemma
```

```
'States'
```

```
sentence1[12].lemma
```

```
'know'
```

11.2.4.8 Morph

Morphological analysis -spaCy docs

```
sentence1[12].morph
```

```
Aspect=Perf|Tense=Past|VerbForm=Part
```

11.2.4.9 Part of Speech

Coarse-grained part-of-speech from the Universal POS tag set. -spaCy docs

```
token2.pos_
```

```
'PROPN'
```

11.2.4.10 Syntactic Dependency

Syntactic dependency relation. -spaCy docs

```
token2.dep_
```

```
'nsubj'
```

11.2.4.11 Language

Language of the parent document's vocabulary. -spaCy docs

```
token2.lang_
```

```
'en'
```

11.2.5 Part-of-Speech (POS) Tagging

In the field of computational linguistics, understanding parts-of-speech is essential. spaCy offers an easy way to parse a text and identify its parts of speech. Below, we will iterate across each token (word or punctuation) in the text and identify its part of speech.

```
for token in sentence1:  
    print (token.text, token.pos_, token.dep_)
```

```
The DET det  
United PROPN compound  
States PROPN nsubj  
of ADP prep  
America PROPN pobj  
( PUNCT punct  
U.S.A. PROPN appos  
or CCONJ cc  
USA PROPN conj  
) PUNCT punct  
, PUNCT punct  
commonly ADV advmod  
known VERB acl  
as ADP prep  
the DET det  
United PROPN compound  
States PROPN pobj  
( PUNCT punct  
U.S. PROPN appos  
or CCONJ cc  
US PROPN conj  
) PUNCT punct  
or CCONJ cc  
America PROPN conj
```

(continues on next page)

(continued from previous page)

```
, PUNCT punct
is AUX ROOT
a DET det
country NOUN attr
located VERB acl
in ADP prep
North PROPN compound
America PROPN pobj
. PUNCT punct
```

Here, we can see two vital pieces of information: the string and the corresponding part-of-speech (pos). For a complete list of the pos labels, see the spaCy documentation (<https://spacy.io/api/annotation#pos-tagging>). Most of these, however, should be apparent, i.e. PROPN is proper noun, AUX is an auxiliary verb, ADJ, is adjective, etc. We can visualize this sentence with a diagram through spaCy's displaCy Notebook feature. To make this easier to read in a book, we will be using a different, smaller sentence.

```
from spacy import displacy
sample_doc = nlp("The dog ran over the fence.")
displacy.render(sample_doc, style="dep", options={"compact":True})
```

```
<IPython.core.display.HTML object>
```

11.2.6 Named Entity Recognition (NER)

Another essential task of NLP, is named entity recognition, or NER. I spoke about NER in the last notebook. Here, I'd like to demonstrate how to perform basic NER via spaCy. Again, we will iterate over the Doc object as we did above, but instead of iterating over `doc.sents`, we will iterate over `doc.ents`. For our purposes right now, I simply want to print off each entity's text (the string itself) and its corresponding label (note the `_` after label). I will be explaining this process in much greater detail in the next two notebooks.

```
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
The United States of America GPE
U.S.A. GPE
USA GPE
the United States GPE
U.S. GPE
US GPE
America GPE
North America LOC
50 CARDINAL
five CARDINAL
nine CARDINAL
Minor Outlying Islands PERSON
326 CARDINAL
Indian NORP
third ORDINAL
The United States GPE
Canada GPE
Mexico GPE
```

(continues on next page)

(continued from previous page)

```
Bahamas GPE
Cuba GPE
Russia GPE
over 331 million MONEY
third ORDINAL
Washington GPE
D.C. GPE
New York City GPE
```

Sometimes it can be difficult to read this output as raw data. In this case, we can again leverage spaCy's displaCy feature. Notice that this time we are altering the keyword argument, style, with the string "ent". This tells displaCy to display the text as NER annotations.

```
displacy.render(doc, style="ent")
```

```
<IPython.core.display.HTML object>
```

11.2.7 Conclusion

I recommend spending a little bit of time going through this notebook a few times. The information covered throughout this notebook will be reinforced as we explore each of these areas in more depth with real-world examples of how to implement them to tackle different problems.

11.3 spaCy's Pipelines

Now that we understand the basics of spaCy's linguistic features, let's explore pipelines in spaCy. As we have seen, spaCy offers both heuristic (rules-based) and machine learning natural language processing solutions. These solutions are activated by pipes. Here, we will explore pipes and pipelines generally and the ones offered by spaCy specifically. In the next section, we will explore how you can create custom pipes and add them to a spaCy pipeline. Before we jump in, let's import spaCy.

```
import spacy
```

11.3.1 Standard Pipes (Components and Factories) Available from spaCy

spaCy is much more than an NLP framework. It is also a way of designing and implementing complex pipelines.

A **pipe** is a component of a **pipeline**. A pipeline's purpose is to take input data, perform some sort of operation on that input data, and then output those operations either as a new data or extracted metadata. A pipe is an individual component of a pipeline. In the case of spaCy, there are a few different pipes that perform different tasks. The tokenizer, tokenizes the text into individual tokens; the parser, parses the text, and the NER identifies entities and labels them accordingly. All of this data is stored in the Doc object.

It is important to remember that pipelines are sequential. This means that components earlier in a pipeline affect what later components receive. Sometimes this sequence is

essential, meaning later pipes depend on earlier pipes. At other times, this sequence is not essential, meaning later pipes can function without earlier pipes. It is important to keep this in mind as you create custom spaCy models (or any pipeline for that matter).

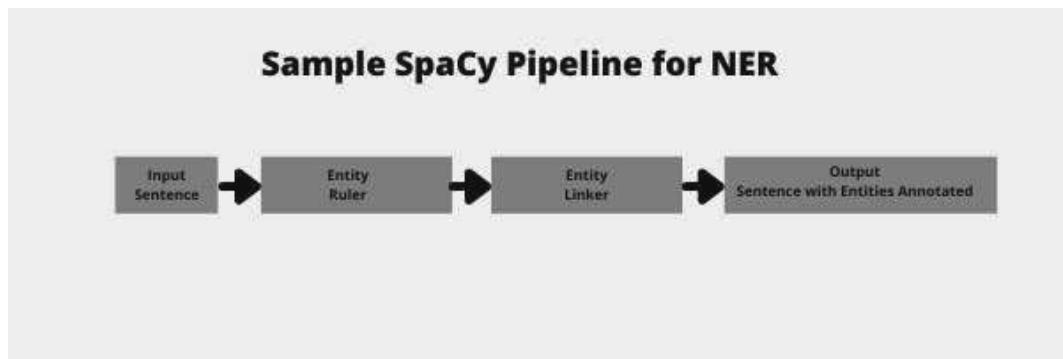


FIGURE 11.2

Example of an NER Pipeline.

Here, we see an input, in this case a sentence, enter the pipeline from the left. Two pipes are activated on this, a rules-based named entity recognizer known as an EntityRuler which finds entities and an EntityLinker pipe that identifies what entity that is to perform toponym resolution. The sentence is then outputted with the sentence and the entities annotated. At this point, we could use the `doc.ents` feature to find the entities in our sentence. In spaCy, you will often use pipelines that are more sophisticated than this. You will specifically use a Tok2Vec input layer to vectorize your input sentence. This will allow machine learning pipes to make predictions.

Below is a complete list of the AttributeRuler pipes available to you from spaCy and the Matchers.

11.3.1.1 Attribute Rulers

- Dependency Parser
- EntityLinker
- EntityRecognizer
- EntityRuler
- Lemmatizer
- Morpholog
- SentenceRecognizer
- Sentencizer
- SpanCategorizer
- Tagger
- TextCategorizer
- Tok2Vec

- Tokenizer
- TrainablePipe
- Transformer

11.3.1.2 Matchers

- DependencyMatcher
- Matcher
- PhraseMatcher

11.3.2 How to Add Pipes

In most cases, you will use an off-the-shelf spaCy model. In some cases, however, an off-the-shelf model will not fill your needs or will perform a specific task very slowly. A good example of this is sentence tokenization. Imagine if you had a document that was around 1 million sentences long. Even if you used the small English model, your model would take a long time to process those 1 million sentences and separate them. In this instance, you would want to make a blank English model and simply add the Sentencizer to it. The reason is because each pipe in a pipeline will be activated (unless specified) and that means that each pipe from Dependency Parser to named entity recognition will be performed on your data. This is a serious waste of computational resources and time. The small model may take hours to achieve this task. By creating a blank model and simply adding a Sentencizer to it, you can reduce this time to merely minutes.

To demonstrate this process, let's first create a blank model.

```
nlp = spacy.blank("en")
```

Here, notice that we have used `spacy.blank`, rather than `spacy.load`. When we create a blank model, we simply pass the two letter combination for a language, in this case, "en" for English. Now, let's use the `add_pipe()` command to add a new pipe to it. We will simply add a Sentencizer.

```
nlp.add_pipe("sentencizer")
```

```
<spacy.pipeline.sentencizer.Sentencizer at 0x7fd926ca9840>
```

```
import requests
from bs4 import BeautifulSoup
s = requests.get("https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.
˓→shakespeare.txt")
soup = BeautifulSoup(s.content).text.replace("\n", "").replace("\n", "
˓→")[:100000]
```

```
%%time
doc = nlp(soup)
print(len(list(doc.sents)))
```

```
500
CPU times: user 155 ms, sys: 3.84 ms, total: 159 ms
Wall time: 157 ms
```

```
nlp2 = spacy.load("en_core_web_sm")

%%time
doc = nlp2(soup)
print (len(list(doc.sents)))

781
CPU times: user 2.99 s, sys: 79.5 ms, total: 3.07 s
Wall time: 3.07 s
```

The difference in time here is remarkable. Our text string was 100,000 characters. The blank model with just the Sentencizer completed its task in 159 milliseconds and found 781 sentences. The small English model, the most efficient one offered by spaCy, did the same task in 3.07 seconds and found 781 sentences. The small English model, in other words, took much longer. This time increases significantly as your texts get larger. The reason for this large difference in time is the other components active in the spaCy pipeline, such as the POS Tagger and NER pipes.

Often times you need to find sentences quickly, not necessarily accurately. In these instances, it makes sense to know tricks like the one above.

11.3.3 Examining a Pipeline

In spaCy, we have a few different ways to study a pipeline. If we want to do this in a script, we can do the following command:

```
nlp2.analyze_pipes()

{'summary': {'tok2vec': {'assigns': ['doc.tensor'],
    'requires': [],
    'scores': [],
    'retokenizes': False},
    'tagger': {'assigns': ['token.tag'],
    'requires': [],
    'scores': ['tag_acc'],
    'retokenizes': False},
    'parser': {'assigns': ['token.dep',
        'token.head',
        'token.is_sent_start',
        'doc.sents'],
    'requires': [],
    'scores': ['dep_uas',
        'dep_las',
        'dep_las_per_type',
        'sents_p',
        'sents_r',
        'sents_f'],
    'retokenizes': False},
    'attribute_ruler': {'assigns': [],
    'requires': [],
    'scores': [],
    'retokenizes': False},
    'lemmatizer': {'assigns': ['token.lemma'],
    'requires': [],
    'scores': ['lemma_acc'],
    'retokenizes': False}},
```

(continues on next page)

(continued from previous page)

```
'ner': {'assigns': ['doc.ents', 'token.ent_iob', 'token.ent_type'],
'requires': [],
'scores': ['ents_f', 'ents_p', 'ents_r', 'ents_per_type'],
'retokenizes': False}},
'problems': {'tok2vec': []},
'tagger': [],
'parser': [],
'attribute_ruler': [],
'lemmatizer': [],
'ner': []},
'attrs': {'token.lemma': {'assigns': ['lemmatizer'], 'requires': []},
'doc.sents': {'assigns': ['parser'], 'requires': []},
'token.is_sent_start': {'assigns': ['parser'], 'requires': []},
'token.dep': {'assigns': ['parser'], 'requires': []},
'token.tag': {'assigns': ['tagger'], 'requires': []},
'doc.ents': {'assigns': ['ner'], 'requires': []},
'token.ent_iob': {'assigns': ['ner'], 'requires': []},
'token.head': {'assigns': ['parser'], 'requires': []},
'doc.tensor': {'assigns': ['tok2vec'], 'requires': []},
'token.ent_type': {'assigns': ['ner'], 'requires': []}}}
```

Note the dictionary structure. This tells us not only what is inside the pipeline, but its order. Each key after “summary” is a pipe. The value is a dictionary. This dictionary tells us a few different things. All of these value dictionaries state: “assigns” which corresponds to a value of what that particular pipe assigns to the token and doc as it passes through the pipeline. In some cases, there will be a key of “scores” in the dictionary. This indicates how the machine learning model was evaluated.

11.3.4 Conclusion

This section has given you an umbrella overview of spaCy. Over the next two sections, we will deep dive into specific areas and use spaCy to solve general and domain-specific problems in the digital humanities.

12

Rules-Based spaCy

In this chapter, we will build upon our previous knowledge of spaCy by exploring some of its rules-based pipes, specifically the EntityRuler, Matcher, PhraseMatcher, and custom RegEx-based pipes. By the end of this chapter, you will understand how to apply each of these standard spaCy pipes to specific problems.

12.1 The EntityRuler

12.1.1 Introduction to spaCy's EntityRuler

The Python library that spaCy comes with offers a few different methods for performing rules-based NER. One such method is via its EntityRuler.

The **EntityRuler** is a spaCy factory that allows one to create a set of patterns with corresponding labels. A **factory** in spaCy is a set of classes and functions preloaded in spaCy that perform set tasks. In the case of the EntityRuler, the factory at hand allows the user to create an EntityRuler, give it a set of instructions, and then use this instructions to find and label entities.

Once the user has created the EntityRuler and given it a set of instructions, the user can then add it to the spaCy pipeline as a new pipe. I have spoken in the past notebooks briefly about pipes, but perhaps it is good to address them in more detail here.

In this notebook, we will be looking closely at the EntityRuler as a component of a spaCy model's pipeline. Off-the-shelf spaCy models come preloaded with an NER model; they do not, however, come with an EntityRuler. In order to incorporate an EntityRuler into a spaCy model, it must be created as a new pipe, given instructions, and then added to the model. Once this is complete, the user can save that new model with the EntityRuler to the disk.

The full documentation of spaCy EntityRuler can be found here: <https://spacy.io/api/entityruler>.

This notebook will synthesize this documentation for non-specialists and provide some examples of it in action.

12.1.2 Demonstration of EntityRuler in Action

In the code below, we will introduce a new pipe into spaCy's off-the-shelf small English model. The purpose of this EntityRuler will be to identify small villages in Poland correctly.

```
# Import the requisite library
import spacy

# Build upon the spaCy Small Model
```

(continues on next page)

(continued from previous page)

```
nlp = spacy.load("en_core_web_sm")

#Sample text
text = "The village of Treblinka is in Poland. Treblinka was also an_
→extermination camp."

#create the Doc object
doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Treblinka GPE
Poland GPE
```

Depending on the version of model you are using, some results may vary.

The output from the code above demonstrates spaCy's small model's to identify Treblinka, which is a small village in Poland. As the sample text indicates, it was also an extermination camp during WWII. In the first sentence, the spaCy model tagged Treblinka as an LOC (location) and in the second it was missed entirely. Both are either imprecise or wrong. I would have accepted ORG for the second sentence, as spaCy's model does not know how to classify an extermination camp, but what these results demonstrate is the model's failure to generalize on data. The reason? There are a few, but I suspect the model never encountered the word Treblinka.

This is a common problem in NLP for specific domains. Often times the domains in which we wish to deploy models, off-the-shelf models will fail because they have not been trained on domain-specific texts. We can resolve this, however, either via spaCy's EntityRuler or via training a new model. As we will see over the next few notebooks, we can use spaCy's EntityRuler to easily achieve both.

For now, let's first remedy the issue by giving the model instructions for correctly identifying Treblinka. For simplicity, we will use spaCy's GPE label. In a later notebook, we will teach a model to correctly identify Treblinka in the latter context as a concentration camp.

```
#Import the requisite library
import spacy

#Build upon the spaCy Small Model
nlp = spacy.load("en_core_web_sm")

#Sample text
text = "The village of Treblinka is in Poland. Treblinka was also an_
→extermination camp."

#create the EntityRuler
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns
patterns = [
    {"label": "GPE", "pattern": "Treblinka"}
]

ruler.add_patterns(patterns)
```

(continues on next page)

(continued from previous page)

```
doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Treblinka LOC
Poland GPE
Treblinka GPE
```

If you executed the code above and found that you had the same output, then you did everything correctly. This method has failed. Why? The answer comes back to the concept of pipelines. We created and added the EntityRuler to the spaCy model's pipeline, but by default, spaCy adds a new pipe to the end of the pipeline. In order to visualize the pipeline, let's use spaCy's `analyze_pipes()`.

```
nlp.analyze_pipes()
```

```
{'summary': {'tok2vec': {'assigns': ['doc.tensor'],
    'requires': [],
    'scores': [],
    'retokenizes': False},
    'tagger': {'assigns': ['token.tag'],
        'requires': [],
        'scores': ['tag_acc'],
        'retokenizes': False},
    'parser': {'assigns': ['token.dep',
        'token.head',
        'token.is_sent_start',
        'doc.sents'],
        'requires': [],
        'scores': ['dep_uas',
            'dep_las',
            'dep_las_per_type',
            'sents_p',
            'sents_r',
            'sents_f'],
        'retokenizes': False},
    'attribute_ruler': {'assigns': [],
        'requires': [],
        'scores': [],
        'retokenizes': False},
    'lemmatizer': {'assigns': ['token.lemma'],
        'requires': [],
        'scores': ['lemma_acc'],
        'retokenizes': False},
    'ner': {'assigns': ['doc.ents', 'token.ent_iob', 'token.ent_type'],
        'requires': [],
        'scores': ['ents_f', 'ents_p', 'ents_r', 'ents_per_type'],
        'retokenizes': False},
    'entity_ruler': {'assigns': ['doc.ents', 'token.ent_type', 'token.ent_iob'],
        'requires': [],
        'scores': ['ents_f', 'ents_p', 'ents_r', 'ents_per_type'],
        'retokenizes': False}}},
```

(continues on next page)

(continued from previous page)

```
'problems': {'tok2vec': [],
  'tagger': [],
  'parser': [],
  'attribute_ruler': [],
  'lemmatizer': [],
  'ner': [],
  'entity_ruler': []},
'attrs': {'token.ent_iob': {'assigns': ['ner', 'entity_ruler'],
  'requires': []},
  'token.tag': {'assigns': ['tagger'], 'requires': []},
  'token.dep': {'assigns': ['parser'], 'requires': []},
  'token.is_sent_start': {'assigns': ['parser'], 'requires': []},
  'token.lemma': {'assigns': ['lemmatizer'], 'requires': []},
  'token.ent_type': {'assigns': ['ner', 'entity_ruler'], 'requires': []},
  'doc.sents': {'assigns': ['parser'], 'requires': []},
  'token.head': {'assigns': ['parser'], 'requires': []},
  'doc.tensor': {'assigns': ['tok2vec'], 'requires': []},
  'doc.ents': {'assigns': ['ner', 'entity_ruler'], 'requires': []}}}
```

This can be a bit difficult to read at first, but what it shows us is the order in which our pipes are set up and a few other key pieces of information about each pipe. If we locate “ner”, we notice that “entity_ruler” sits behind it.

In order for our EntityRuler to have primacy, we have to assign it to after the “ner” pipe, as the example below shows in this line:

```
ruler = nlp.add_pipe("entity_ruler", after="ner")
```

```
#Build upon the spaCy Small Model
nlp = spacy.load("en_core_web_sm")

#Sample text
text = "The village of Treblinka is in Poland. Treblinka was also an_
→extermination camp."

#Create the EntityRuler
ruler = nlp.add_pipe("entity_ruler", after="ner")

#List of Entities and Patterns
patterns = [
    {"label": "GPE", "pattern": "Treblinka"}
]

ruler.add_patterns(patterns)

doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Treblinka GPE
Poland GPE
Treblinka GPE
```

Notice now that our EntityRuler is functioning before the “ner” pipe and is, therefore, prefinding entities and labeling them before the NER gets to them. Because it comes earlier in the pipeline, its metadata holds primacy over the later “ner” pipe.

12.1.3 Introducing Complex Rules and Variance to the EntityRuler (Advanced)

In some instances, labels may have a set type of variance that follow a distinct pattern or sets of patterns. One such example (included in the spaCy documentation) is phone numbers. In the United States, phone numbers have a few forms. The standard formal method is (xxx)-xxx-xxxx, but it is not uncommon to see xxx-xxx-xxxx or xxxxxxxxx. If the owner of the phone number is giving that same number to someone outside the US, then +1(xxx)-xxx-xxxx.

If you are working within a United States domain, you can pass RegEx formulas to the pattern matcher to grab all of these instances.

The spaCy EntityRuler also allows the user to introduce a variety of complex rules and variances (via, among other things, RegEx) by passing the rules to the pattern. There are many arguments that one can pass to the patterns. For a complete list, see: <https://spacy.io/usage/rule-based-matching>. To experiment with how these work, I recommend using the spaCy Matcher demo: <https://explosion.ai/demos/matcher>.

In the example below we work with one example from the spaCy documentation in which we extract a phone number from a text. This same task can be done via RegEx as well.

```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number (555) 555-5555."

#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-
#matching)
patterns = [
    {"label": "PHONE_NUMBER", "pattern": [{"ORTH": "(", "SHAPE": "ddd"}, {"ORTH": ")", "SHAPE": "ddd"}, {"ORTH": "-", "OP": "??"}, {"ORTH": "ddd"}]}
]
#add patterns to ruler
ruler.add_patterns(patterns)

#create the doc
doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

(555) 555-5555 PHONE_NUMBER

12.2 The Matcher

12.2.1 Introduction

SpaCy's built in Matcher component is another way a user can leverage spaCy to leverage linguistic annotations to find and extract structured data from unstructured text. Unlike the

EntityRuler that we met in the previous section, the Matcher does not assign the identified patterns into an extension, such as `doc.ents`. Instead, the Matcher is designed to be leveraged outside of a spaCy pipeline. Also unlike the EntityRuler, the Matcher will not sit inside a spaCy pipeline, rather it will run over a spaCy Doc container.

12.2.2 A Basic Example

To understand how this works, let's first look at a very basic example of the spaCy Matcher in practice. We will be working with the small spaCy English pipeline in this example, so let's go ahead and import the `Matcher` class and load up the pipeline.

```
import spacy
from spacy.matcher import Matcher
```

```
nlp = spacy.load("en_core_web_sm")
```

Now that we have our pipeline loaded up into memory, let's go ahead and start working with the Matcher component. To load up the Matcher, we will use the `Matcher` class that we imported above. The `Matcher` class will take a single argument, the vocabulary of the `nlp` pipeline. We can access the `nlp.vocab` by using the command `nlp.vocab`.

```
matcher = Matcher(nlp.vocab)
```

With our Matcher now loaded into memory, it is time to create a basic pattern. The pattern that we will want to find and extract from our texts is an email. As we will learn below, there are many token attributes you can leverage to create powerful rules-based pattern matchers in spaCy, rather like the EntityRuler we learned about in the previous section. One attribute we can look for is a Boolean (True or False) that looks to see if a token looks like an e-mail address. This attribute is `LIKE_EMAIL`.

```
pattern = [
    {"LIKE_EMAIL": True}
]
```

As with the EntityRuler, we can add these patterns into the Matcher. Unlike our EntityRuler, we will use the `.add` method, rather than `.add_patterns`. This method will take two arguments. First, the label that you want to assign to the matched token(s) and second the patterns themselves. Note that just like the EntityRuler, our patterns must be nested in a list.

```
matcher.add("EMAIL_ADDRESS", [pattern])
```

With the Matcher now loaded in memory with patterns, we can run it over some text. Because the spaCy Matcher is a spaCy component, that text must first be converted into a spaCy Doc container. We can create that Doc container just as we normally would.

```
doc = nlp("This is an email address: wmattingly@aol.com")
```

We can then pass that Doc container to the Matcher as a single argument.

```
matches = matcher(doc)
```

Let's go ahead and print off the `matches` object now.

```
print(matches)
```

```
[ (16571425990740197027, 6, 7) ]
```

This may not be what you were expecting, but let's dive in and explore what is going on with this output. The output is a list with a single index. Each index will be a tuple. This tuple will consist of three parts: `lexeme`, `start token`, and `end token`. The `lexeme` is a numerical representation of our label in the `nlp.vocab`. We can access the label name by indexing the `nlp.vocab` at the `lexeme` index and then accessing the raw text with `.text`.

Let's do each of these steps in turn. First, we will create an object called `first_match`. This will be our first hit with the Matcher.

```
first_match = matches[0]
print(first_match)
```

```
(16571425990740197027, 6, 7)
```

Next, we will grab `lexeme`, which is the first index in our `first_match`.

```
lexeme = first_match[0]
print(lexeme)
```

```
16571425990740197027
```

Finally, we will print off the raw text of the label by indexing the `nlp.vocab` at that specific `lexeme` numerical value.

```
print(nlp.vocab[lexeme].text)
```

```
EMAIL_ADDRESS
```

This is the basic idea behind the Matcher. It is a way of finding structured text using a pattern-based matching technique. As with the EntityRuler, there are a lot of token attributes we can leverage.

12.2.3 Attributes Taken by Matcher

- ORTH – The exact verbatim of a token (str)
- TEXT – The exact verbatim of a token (str)
- LOWER – The lowercase form of the token text (str)
- LENGTH – The length of the token text (int)
- IS_ALPHA
- IS_ASCII
- IS_DIGIT
- IS_LOWER
- IS_UPPER

- IS_TITLE
- IS_PUNCT
- IS_SPACE
- IS_STOP
- IS_SENT_START
- LIKE_NUM
- LIKE_URL
- LIKE_EMAIL
- SPACY
- POS
- TAG
- MORPH
- DEP
- LEMMA
- SHAPE
- ENT_TYPE
- _ – Custom extension attributes (Dict[str, Any])
- OP

12.2.4 Applied Matcher

Let's now take a look at a practical application of the Matcher. Say, we had a piece of text. In our case, it will be the following text:

```
text = """
Harry Potter was the main character in the book.
Harry was a normal boy who discovered he was a wizard.
Ultimately, Potter goes to Hogwarts.
He is also known as the Boy who Lived.
The Boy who Lived has an enemy named Voldemort who is known as He who Must not-
→be Named.
"""
```

Our goal in this exercise is to isolate all proper nouns and extract them. Ideally, we would like to extract proper nouns that are bigrams or trigrams and keep them intact.

To do this, we will need to load up the spaCy small English pipeline.

```
nlp = spacy.load("en_core_web_sm")
```

12.2.4.1 Grabbing all Proper Nouns

Our initial pattern will be quite simple. We are simply going to grab all patterns where a single token has a part-of-speech (POS) is “PROPN”, or proper noun.

```
matcher = Matcher(nlp.vocab)
pattern = [{"POS": "PROPN"}]
matcher.add("PROPER_NOUNS", [pattern])
doc = nlp(text)
matches = matcher(doc)
print (len(matches))
for match in matches[:10]:
    print (match, doc[match[1]:match[2]])
```

```
6
(3232560085755078826, 1, 2) Harry
(3232560085755078826, 2, 3) Potter
(3232560085755078826, 12, 13) Harry
(3232560085755078826, 27, 28) Potter
(3232560085755078826, 30, 31) Hogwarts
(3232560085755078826, 52, 53) Voldemort
```

12.2.4.2 Improving it with Multi-Word Tokens

While the above output is good, it does not capture multi-word-tokens, such as Harry Potter. Ideally, we would like to find and extract these instances. We can do this by passing in a second argument in our pattern after the POS: OP and set it to “+”. This will look for any sequence of proper nouns.

```
matcher = Matcher(nlp.vocab)
pattern = [{"POS": "PROPN", "OP": "+"}]
matcher.add("PROPER_NOUNS", [pattern])
doc = nlp(text)
matches = matcher(doc)
print (len(matches))
for match in matches[:10]:
    print (match, doc[match[1]:match[2]])
```

```
7
(3232560085755078826, 1, 2) Harry
(3232560085755078826, 1, 3) Harry Potter
(3232560085755078826, 2, 3) Potter
(3232560085755078826, 12, 13) Harry
(3232560085755078826, 27, 28) Potter
(3232560085755078826, 30, 31) Hogwarts
(3232560085755078826, 52, 53) Voldemort
```

12.2.4.3 Greedy Keyword Argument

As we can see, this has worked pretty well, but we have a key issue. Harry is grabbed as Harry Potter as is Potter. These are all three instances of a single multi-word token: Harry Potter.

The Matcher can also take another keyword argument when we add patterns: greedy. This will take one of two possible arguments: FIRST or LONGEST. They each define how spaCy will function when it encounters two matches that have overlapping spans. FIRST

will extract the first hit with overlapping spans, while `LONGEST` will extract only the longest. Let's consider the same example above.

```
matcher = Matcher(nlp.vocab)
pattern = [{"POS": "PROPN", "OP": "+"}]
matcher.add("PROPER_NOUNS", [pattern], greedy='LONGEST')
doc = nlp(text)
matches = matcher(doc)
print (len(matches))
for match in matches[:10]:
    print (match, doc[match[1]:match[2]])
```

```
5
(3232560085755078826, 1, 3) Harry Potter
(3232560085755078826, 12, 13) Harry
(3232560085755078826, 27, 28) Potter
(3232560085755078826, 30, 31) Hogwarts
(3232560085755078826, 52, 53) Voldemort
```

12.2.4.4 Adding in Sequences

Let's say I not only wanted to extract these type of instances, but I wanted to look for more robust patterns. What if I wanted to look for places in the text where Harry Potter is followed by a verb of action (not a verb of to be). We can do this by passing a third component to our pattern: POS is VERB.

```
matcher = Matcher(nlp.vocab)
pattern = [{"POS": "PROPN", "OP": "+"}, {"POS": "VERB"}]
matcher.add("PROPER_NOUNS", [pattern], greedy='LONGEST')
doc = nlp(text)
matches = matcher(doc)
matches.sort(key = lambda x: x[1])
print (len(matches))
for match in matches[:10]:
    print (match, doc[match[1]:match[2]])
```

```
1
(3232560085755078826, 27, 29) Potter goes
```

And just like this, we were able to extract an instance of Harry Potter's name followed by a verb of action. In our case, `goes`. While this example is quite simple, the Matcher allows for robust pattern matching with spaCy containers. The patterns that we looked at here can also be used and applied to the EntityRuler.

12.3 The PhraseMatcher

12.3.1 Introduction

Another rules-based component built into spaCy is the PhraseMatcher. Like the Matcher, the PhraseMatcher does not sit inside a spaCy pipeline. It does not, therefore, align with a spaCy extension, such as `doc.ents`, like the EntityRuler does. Instead, it is meant to run over a Doc container, just like the Matcher. Unlike the Matcher, however, the PhraseMatcher does not function a sequence of linguistic features at the token level, rather it is focused on matching at the phrase level.

In practice, you would use the Matcher when you need to rely on a sequence of linguistic features at the token level to extract data. This is powerful, but can sometimes be difficult to write robust patterns to match all instances of a the patterns you wish to match. The PhraseMatcher, on the other hand, should be used when you know relatively well how the data will appear in a text. It is easier to use the PhraseMatcher, but it is not as dynamic as the Matcher.

Basic Example

As with the Matcher, it is best to see the PhraseMatcher in action with a basic example. First, let's import the `PhraseMatcher` class and load up the default small English pipeline.

```
import spacy
from spacy.matcher import PhraseMatcher
```

```
nlp = spacy.load("en_core_web_sm")
```

Now, let's consider a basic example. Let's consider the text below. Here, we wish to find and extract the instances where Harry Potter appears in the text. Harry appears in four different ways in the text: 1) Harry Potter, 2) Harry, 3) Potter, and 4) The Boy who Lived.

```
text = """
Harry Potter was the main character in the book.
Harry was a normal boy who discovered he was a wizard.
Ultimately, Potter goes to Hogwarts.
He is also known as the Boy who Lived.
The Boy who Lived has an enemy named Voldemort who is known as He who Must not...
→be Named.
"""
```

```
matcher = PhraseMatcher(nlp.vocab)
```

```
matcher.add("HARRY_POTTER", [nlp("Harry Potter"), nlp("Harry"), nlp("Potter"),
→nlp("the Boy who Lived")])
```

```
doc = nlp(text)
```

Again, we will create our matches.

```
matches = matcher(doc)
```

Let's iterate over our matches.

```
for match in matches:
    print(match)
```

```
(12243270181114079557, 1, 2)
(12243270181114079557, 1, 3)
(12243270181114079557, 2, 3)
(12243270181114079557, 12, 13)
(12243270181114079557, 27, 28)
(12243270181114079557, 38, 42)
```

And now let's iterate over our matches and grab a bit more data, including the token spans and the sentence in which a match was found.

```
for match in matches:
    lexeme, start, end = match
    print(nlp.vocab[lexeme].text, doc[start:end])
    print(f"Sentence: {doc[start].sent}")
```

```
HARRY_POTTER Harry
Sentence: Harry Potter was the main character in the book.

HARRY_POTTER Harry Potter
Sentence: Harry Potter was the main character in the book.

HARRY_POTTER Potter
Sentence: Harry Potter was the main character in the book.

HARRY_POTTER Harry
Sentence: Harry was a normal boy who discovered he was a wizard.

HARRY_POTTER Potter
Sentence: Ultimately, Potter goes to Hogwarts.

HARRY_POTTER the Boy who Lived
Sentence: He is also known as the Boy who Lived.
```

As we can tel, the results are good, but we are missing one case. The second example of The Boy who Lived was not grabbed because The was not capitalized. We can account for this by changing the main attribute of the PhraseMatcher.

12.3.2 Setting a Custom Attribute

Unlike the Matcher, the PhraseMatcher does not let us control how it reads each individual token in the pattern. The way the PhraseMatcher parses the phrase is as the sequence level. By default, the PhraseMatcher reads the entire pattern as ORTH, or raw text. In other words, it must be a precise match in order to be flagged and extracted. In some instances, however, it may be important for a pattern to be not just raw text, but also in all forms, both uppercase and lowercase. This is particularly important for phrases, like the boy who lived, where the word the may be capitalized if it is at the start of a sentence. In these instances, we can change the main way the PhraseMatcher works by using the attr argument. By using attr="LOWER", we can make our PhraseMatcher pattern case-agnostic.

```
matcher = PhraseMatcher(nlp.vocab, attr="LOWER")
```

```
matcher.add("HARRY_POTTER", [nlp("Harry Potter"), nlp("Harry"), nlp("Potter"), nlp("the Boy who Lived
↪")])
```

```
doc = nlp(text)
```

```
matches = matcher(doc)
```

```
for match in matches:
    lexeme, start, end = match
    print(nlp.vocab[lexeme].text, doc[start:end])
    print(f"Sentence: {doc[start].sent}")
```

```
HARRY_POTTER Harry
Sentence: Harry Potter was the main character in the book.

HARRY_POTTER Harry Potter
Sentence: Harry Potter was the main character in the book.

HARRY_POTTER Potter
Sentence: Harry Potter was the main character in the book.

HARRY_POTTER Harry
Sentence: Harry was a normal boy who discovered he was a wizard.

HARRY_POTTER Potter
Sentence: Ultimately, Potter goes to Hogwarts.

HARRY_POTTER the Boy who Lived
Sentence: He is also known as the Boy who Lived.

HARRY_POTTER The Boy who Lived
Sentence: The Boy who Lived has an enemy named Voldemort who is known as He_
→who Must not be Named.
```

Notice that now, we have grabbed all ways the phrase the boy who lived is expressed in our text.

12.3.3 Adding a Function with `on_match`

In production, it can sometimes be difficult to deploy a spaCy-based solution that requires pasting a for loop each time you want to iterate over the results. Usually, you want to automate certain tasks so that when a match is found, some event occurs in your code. The PhraseMatcher allows you to pass an extra argument to your patterns: `on_match`. This keyword argument will take a function which will receive four arguments from the PhraseMatcher: `matcher` (the PhraseMatcher), `doc` (the doc container that the PhraseMatcher just passed over), `id`, and `matches` (the resulting matches from the PhraseMatcher).

Let's create a basic function that will iterate over each match and print off the match, its label, and the sentence in which it was found.

```
def on_match(matcher, doc, id, matches):
    for match in matches:
        lexeme, start, end = match
        print(nlp.vocab[lexeme].text, doc[start:end])
        print(f"Sentence: {doc[start].sent}")
```

Just as before, we will create our PhraseMatcher.

```
matcher = PhraseMatcher(nlp.vocab, attr="LOWER")
```

This time, however, when we add our patterns to the PhraseMatcher, we will also add the keyword argument `on_match` that will point to the above function.

```
matcher.add("HARRY_POTTER", [nlp("Harry Potter")], on_match=on_match)
```

All that is left to do is then create the Doc container from the text and then run the PhraseMatcher over the Doc container.

```
doc = nlp(text)
matches = matcher(doc)
```

```
HARRY_POTTER Harry Potter
Sentence: Harry Potter was the main character in the book.
```

Just like the PhraseMatcher, the Matcher also can take the `on_match` keyword argument.

12.4 Using RegEx with spaCy

12.4.1 What Is Regular Expressions (RegEx)?

Regular Expressions, or RegEx for short, is a way of achieving complex string matching based on simple or complex patterns. It can be used to perform finding and retrieving patterns or replacing matching patterns in a string with some other pattern. It was invented by an Stephen Cole Kleene in the 1950s and is still widely used today for numerous tasks, but particularly string matching in texts. RegEx are fully integrated with most search engines and can allow for more robust searching. Nearly all data scientists, especially those who work with texts, use RegEx at some stage in their workflow, from data searching, to cleaning data, to implementing machine learning models. It is an essential tool for any text-based researcher. For these reasons, it merits a few chapters in this textbook.

In spaCy it can be leveraged in a few different pipes (depending on the task at hand as we shall see), to identify things such as entities or pattern matching.

12.4.2 The Strengths of RegEx

There are several strengths to RegEx.

1. Due to its complex syntax, it can allow for programmers to write robust rules in short spaces.
2. It can allow the researcher to find all types of variance in strings.
3. It can perform remarkably quickly when compared to other methods.
4. It is universally supported.

12.4.3 The Weaknesses of RegEx

Despite these strengths, there are a few weaknesses to RegEx.

1. Its syntax is quite difficult for beginners. (I still find myself looking up how to do certain things.)
2. In order to work well, it requires a domain-expert to work alongside the programmer to think of all ways a pattern may vary in texts.

12.4.4 How to Use RegEx in Python

Python comes prepackaged with a RegEx library. We can import it like so:

```
import re
```

Now that we have it imported, we can begin to write out some RegEx rules. Let's say we want to find an occurrence of a date in a text. As noted in an earlier notebook, there are a finite number of ways this can be represented. Let's try to grab all instances of a day followed by a month first.

```
pattern = r"((\d){1,2}_
↪(January|February|March|April|May|June|July|August|September|_
↪October|November|December))"

text = "This is a date 2 February. Another date would be 14 August."
matches = re.findall(pattern, text)
print (matches)
```

```
[('2 February', '2', 'February'), ('14 August', '4', 'August')]
```

In this bit of code, we see a real-life RegEx formula at work. While this looks quite complex, its syntax is fairly straight forward. Let's break it down. The first (tells RegEx that I'm looking for something within the parentheses). In other words, I'm looking for a pattern that's going to match the whole pattern, not just components.

Next, we state `(\d){1,2}`. This means that we are looking for any digit (0–9) that occurs either once or twice, `{1,2}`.

Next, we have a space to indicate the space in the string that we would expect with a date.

Next, we have `(January|February|March|April|May|June|July|August|September|October|November|December)` – this indicates another component of the pattern (because it is parentheses). The “|” indicates the same concept as “or” in English, so either January, or February, or March, etc.

When we bring it together, this pattern will match anything that functions as a set of one or two numbers followed by a month. What happens when we try and do this with a date that is formed the opposite way?

```
text = "This is a date February 2. Another date would be 14 August."
matches = re.findall(pattern, text)
print (matches)
```

```
[('14 August', '4', 'August')]
```

It fails. But this is no fault of RegEx. Our pattern cannot accommodate that variation. Nevertheless, we can account for it by adding it as a possible variation. Possible variations are accounted for with a “*”.

```
pattern = r"((\d){1,2}_
↪(January|February|March|April|May|June|July|August|September|_
↪October|November|December))|((January|February|March|April|_
May|June|July|August|September|October|November|December )(\d){1,2}))"

text = "This is a date February 2. Another date would be 14 August."
matches = re.findall(pattern, text)
print (matches)
```

```
[('February 2', '', '', '', '', 'February 2', 'February ', 'February', '2'), ((
↪'14 August', '14 August', '4', ' August', 'August', '', '', '')]
```

There are more concise ways to write the same RegEx formula. I have opted here to be more verbose to make it a bit easier to read. You can see that we've allowed for two main options for our pattern matcher.

Notice, however, that we have a lot of superfluous information for each match. These are the components of each match. There are several ways we can remove them. One way is to use the command `finditer`, rather than `.findall` in RegEx.

```
text = "This is a date February 2. Another date would be 14 August."
iter_matches = re.finditer(pattern, text)
print (iter_matches)
```

```
<callable_iterator object at 0x00000217A415BC10>
```

This is an iterator object, we can loop over it, however, and get our results.

```
text = "This is a date February 2. Another date would be 14 August."
iter_matches = re.finditer(pattern, text)
print (iter_matches)
for hit in iter_matches:
    print (hit)
```

```
<callable_iterator object at 0x00000217A4256670>
<re.Match object; span=(15, 25), match='February 2'>
<re.Match object; span=(49, 58), match='14 August'>
```

Within each of these is some very salient information, such as the start and end location (inside the span) and the text itself (match). We can use the start and end location to grab the text within the string.

```
text = "This is a date February 2. Another date would be 14 August."
iter_matches = re.finditer(pattern, text)
for hit in iter_matches:
    start = hit.start()
    end = hit.end()
    print (text[start:end])
```

```
February 2
14 August
```

12.4.5 How to Use RegEx in spaCy

Things like dates, times, IP Addresses, etc., that have either consistent or fairly consistent structures are excellent candidates for RegEx. Fortunately, spaCy has easy ways to implement RegEx in three pipes: Matcher, PhraseMatcher, and EntityRuler. One of the major drawbacks to the Matcher and PhraseMatcher, is that they do not align the matches as `doc.ents`. Because this textbook is about NER and our goal is to store the entities in the `doc.ents`, we will focus on using RegEx with the EntityRuler. In the next notebook, we will examine other methods.

In the previous notebook, we saw how the code below allowed for us to capture the phone number in the string. I have modified it a bit here for reasons that will become a bit more clear below.

```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number 555-5555."
```

(continues on next page)

(continued from previous page)

```
#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-
#→matching)
patterns = [
    {"label": "PHONE_NUMBER", "pattern": [{"SHAPE": "ddd"}, {"ORTH": "-", "OP": "?"}, {"SHAPE": "dddd"}]}
]
#add patterns to ruler
ruler.add_patterns(patterns)

#create the doc
doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
INFO:tensorflow:Enabling eager execution
INFO:tensorflow:Enabling v2 tensorshape
INFO:tensorflow:Enabling resource variables
INFO:tensorflow:Enabling tensor equality
INFO:tensorflow:Enabling control flow v2
555-5555 PHONE_NUMBER
```

This method worked well for grabbing the phone number. But what if we wanted to use RegEx as opposed to linguistic features, such as shape? First, let's write some RegEx to capture 555-5555.

```
pattern = r"((\d)\{3\}-(\d)\{4\})"
text = "This is a sample number 555-5555."
matches = re.findall(pattern, text)
print (matches)
```

```
[('555-5555', '5', '5')]
```

Okay. So, now we know that we have a RegEx pattern that works. Let's try and implement it in the spaCy EntityRuler. We can do that with the code below. When we execute the code below, we have no output.

```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number (555) 555-5555."

#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")
```

(continues on next page)

(continued from previous page)

```
#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-
→matching)
patterns = [
    {
        "label": "PHONE_NUMBER", "pattern": [{"TEXT": {"REGEX": "((\d){3}-(\d){4})")}]
    }
]
#add patterns to ruler
ruler.add_patterns(patterns)

#create the doc
doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

This is for one very important reason. spaCy's EntityRuler cannot use RegEx to pattern match across tokens. The dash in the phone number throws off the EntityRuler. So, what are we to do in this scenario? Well, we have a few different options that we will explore in the next notebook. But before we get to that, let's try and use RegEx to capture the phone number with no hyphen.

```
#Import the requisite library
import spacy

#Sample text
text = "This is a sample number 5555555."
#Build upon the spaCy Small Model
nlp = spacy.blank("en")

#Create the Ruler and Add it
ruler = nlp.add_pipe("entity_ruler")

#List of Entities and Patterns (source: https://spacy.io/usage/rule-based-
→matching)
patterns = [
    {
        "label": "PHONE_NUMBER",
        "pattern": [{"TEXT": {"REGEX": "((\d){5})")}]
    }
]
#add patterns to ruler
ruler.add_patterns(patterns)

#create the doc
doc = nlp(text)

#extract entities
for ent in doc.ents:
    print (ent.text, ent.label_)
```

5555555 PHONE_NUMBER

Notice that without the dash and a few modifications to our RegEx, we were able to capture 5555555 because this is a single token in the spaCy Doc object. Let's explore how to solve the problem in the next notebook!

12.5 Working with Multi-Word Token Entities and RegEx in spaCy 3x

12.5.1 Key Concepts in This Notebook

1. Working with multi-word tokens and RegEx in spaCy 3x.
2. RegEx's `finditer`.
3. . Spans

12.5.2 Problems with Multi-Word Tokens in spaCy as Entities

We can use spaCy's Matcher to grab multi-word tokens, or tokens that span multiple tokens. The main problem with this, however, is that these multi-word tokens are not placed into the `doc.ents`. This means that we cannot access them the same way we would other entities. In this notebook, we will figure out how to solve that problem with a simple workflow:

1. Extract multi-word tokens with `re.finditer()`.
2. Reconstruct the spans in the spaCy doc.
3. Give priority to longer spans (optional).
4. Inject the spans into `doc.ents`.

We will cover each of these steps in turn.

12.5.3 Extract Multi-Word Tokens

First, we need to grab the multi-word tokens. In this notebook, we are going to try and grab a multi-word token. In this case, a person whose first name begins with Paul. In the RegEx below, we specify that we are looking for any string that starts with "Paul" and then is followed by a capitalized letter. We then tell it to grab the entire second word until the end of the word.

```
import re

text = "Paul Newman was an American actor, but Paul Hollywood is a British TV-Host. The name Paul is quite common."

pattern = r"Paul [A-Z]\w+"

matches = re.finditer(pattern, text)

for match in matches:
    print (match)

<re.Match object; span=(0, 11), match='Paul Newman'>
<re.Match object; span=(39, 53), match='Paul Hollywood'>
```

Note that we have not grabbed the final "Paul" which is not followed by a last name. In this case, we are not interested in that Paul. Now that we know how to grab the multi-word tokens, we need to have a way to parse them in spaCy.

12.5.4 Reconstruct Spans

This next stage is a bit more complicated, but works quite well once you understand the process. First, we need to import the libraries we will need. Note that we are also adding Span from spacy.tokens.

```
import re
import spacy
from spacy.tokens import Span
```

```
INFO:tensorflow:Enabling eager execution
INFO:tensorflow:Enabling v2 tensorshape
INFO:tensorflow:Enabling resource variables
INFO:tensorflow:Enabling tensor equality
INFO:tensorflow:Enabling control flow v2
```

We will do the same thing that we did above with our text and our pattern.

```
text = "Paul Newman was an American actor, but Paul Hollywood is a British TV_"
       ↳Host. The name Paul is quite common."
pattern = r"Paul [A-Z]\w+"
```

Here, we will create a blank spaCy English model and create the Doc object of the text. It will have no entities in it because we are working with a blank model that does not have an “ner” component.

```
nlp = spacy.blank("en")
doc = nlp(text)
```

Even though this part is unnecessary, it is good to do it here because in other situations you will have entities. If you do, you need to store them as a separate list to which we will append things.

```
original_ents = list(doc.ents)
```

Now, let’s iterate over the results from `re.finditer()`. In this cell, we are going to grab the start and end from each match. We will then create a temporary span that will be equal to where the characters start and end in the doc object. This is important because tokens and characters do not always align correctly. Finally, we append to `mwt_ents`, the start, end, and text. The text is not necessary but it will help with debugging.

```
mwt_ents = []
for match in re.finditer(pattern, doc.text):
    start, end = match.span()
    span = doc.char_span(start, end)
    if span is not None:
        mwt_ents.append((span.start, span.end, span.text))
```

12.5.5 Inject the Spans into the doc.ents

With that data, we can iterate over each entity and identify where it begins and ends in spaCy. Note, we are using the spaCy Span class. This allows us to create a span object and assign it a custom label. With this data, we can append each Span to `original_ents`.

```
for ent in mwt_ents:
    start, end, name = ent
    per_ent = Span(doc, start, end, label="PERSON")
    original_ents.append(per_ent)
```

And finally, we set `doc.ents` equal to `original_ents`. This effectively loads the spans back into the spaCy `doc.ents`.

```
doc.ents = original_ents
```

Let's iterate over the "ents" as we normally would.

```
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Paul Newman PERSON
Paul Hollywood PERSON
```

Note that these are now properly identified entities in our `doc.ents` class.

12.5.6 Give Priority to Longer Spans

Sometimes, the situation is not so neat. Sometimes our custom RegEx entities will overlap with spaCy's entities

```
import re
import spacy

text = "Paul Newman was an American actor, but Paul Hollywood is a British TV_
→Host."
pattern = r"Hollywood"

nlp = spacy.load("en_core_web_sm")

doc = nlp(text)
for ent in doc.ents:
    print (ent.text, ent.label_)
```

```
Paul Newman PERSON
American NORP
Paul Hollywood PERSON
British NORP
```

Let's say that we create a new entity. Maybe words associated with Cinema. So, we want to classify Hollywood as a tag "CINEMA". Now, in the above text, Hollywood is clearly associated with Paul Hollywood, but let's imagine for a moment that it is not. Let's try and run the same code as above. If we do, we notice that we get an error.

```
mwt_ents = []
original_ents = list(doc.ents)
for match in re.finditer(pattern, doc.text):
    print (match)
    start, end = match.span()
    span = doc.char_span(start, end)
    if span is not None:
        mwt_ents.append((span.start, span.end, span.text))
```

(continues on next page)

(continued from previous page)

```
for ent in mwt_ents:
    start, end, name = ent
    per_ent = Span(doc, start, end, label="CINEMA")
    original_ents.append(per_ent)

doc.ents = original_ents
```

```
<re.Match object; span=(44, 53), match='Hollywood'>
```

```
-----
ValueError                                                 Traceback (most recent call last)
<ipython-input-11-425d356ded45> in <module>
      12     original_ents.append(per_ent)
      13
--> 14 doc.ents = original_ents

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\spacy\
→tokens\doc.pyx in spacy.tokens.doc.Doc.ents.__set__()

c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\spacy\
→tokens\doc.pyx in spacy.tokens.doc.Doc.set_ents()

ValueError: [E1010] Unable to set entity information for token 9 which is_
→included in more than one span in entities, blocked, missing or outside.
```

This error tells us that one of our tokens from the `finditer()` overlapped with one that our “ner” component found. This is a problem that can be rectified with spaCy’s `filter_spans`. This gives primacy to longer spans. Notice how we have allowed the Paul Hollywood entity to be a PERSON, rather than CINEMA. This is because Hollywood is shorter than Paul Hollywood.

13

Solving a Domain-Specific Problem: A Case Study with Holocaust NER

Now that we have a basic understanding of the spaCy library, we can begin applying that knowledge towards a domain-specific problem. In our case, we will explore creating a rules-based NLP pipeline for performing named entity recognition (NER) on Holocaust-related documents.

13.1 Cultivating Good Datasets for Entities

13.1.1 Introduction to Datasets

One of the greatest challenges when developing a rules-based pipeline for named entity recognition is finding quality datasets. In certain domains, data is readily available, but in many of the domains associated with the humanities, datasets can be hard to find. For this reason, we often need to create our own datasets. In this section, we will set it upon ourselves to find a list of concentration camps from data that is publicly available on the web. The data we acquire will not be perfect, nor will it be complete, but it will provide a starting point for creating a dataset of known concentration camps.

13.1.2 Acquiring the Data

A good way to think about datasets for named entity recognition heuristics is as lists. We want to construct a list of known concentration camps that we can then pass to a spaCy EntityRuler pipe as a set of patterns to which we can assign the label CAMP. One of the first questions you will ask yourself in this pursuit, is “where can I acquire lists?” The answer is, unfortunately, “it depends.” Sometimes good datasets exist. There are a few good places to look, such as GitHub, Wikipedia, and academic digital projects. For each project, you must don your detective goggles and explore the web to find places to acquire this data. Most times, it will take a bit of work (and some Python code) to get the data into a structured form.

If we are looking to generate entities for concentration camps, we have a wealth of data, but this data is not necessarily cleaned or structured. Let’s examine three different locations where we can collate a list of concentration camps from the web and the strengths and weaknesses of those sources.

13.1.3 United States Holocaust Memorial Museum

The United States Holocaust Memorial Museum (USHMM), located in Washington, D.C. in the United States, is an excellent source for data on the Holocaust. When

searching the USHMM collections, one way to limit your search is by Key Camps (<https://www.ushmm.org/>). This list looks like this:

```
ushmm_camps = ['Alderney', 'Amersfoort', 'Auschwitz', 'Banjica', 'Bełżec',
    'Bergen-Belsen', 'Bernburg', 'Bogdanovka', 'Bolzano', 'Bor', 'Breendonk',
    'Breitenau', 'Buchenwald', 'CheÅ,mno', 'Dachau', 'Drancy', 'Falstad',
    'Flossenbürg', 'Fort VII', 'Fossoli', 'Grini', 'Gross-Rosen',
    'Herzogenbusch', 'Hinzert', 'Janowska', 'Jasenovac', 'Kaiserwald',
    'Kaunas', 'Kemna', 'Klooga', 'Le Vernet', 'Majdanek', 'Malchow',
    'Maly Trostenets', 'Mechelen', 'Mittelbau-Dora', 'Natzweiler-Struthof',
    'Neuengamme', 'Niederhagen', 'Oberer Kuhberg', 'Oranienburg',
    'Osthofen', 'PÅ,aszów', 'Ravensbrück', 'Risiera di San Sabba',
    'Sachsenhausen', 'Sajmište', 'Salaspils', 'Sobibór', 'Soldau', 'Stutthof',
    'Theresienstadt', 'Trawniki', 'Treblinka', 'Vaivara']
print (ushmm_camps)
```

```
['Alderney', 'Amersfoort', 'Auschwitz', 'Banjica', 'Bełżec', 'Bergen-Belsen',
    'Bernburg', 'Bogdanovka', 'Bolzano', 'Bor', 'Breendonk', 'Breitenau',
    'Buchenwald', 'CheÅ,mno', 'Dachau', 'Drancy', 'Falstad', 'Flossenbürg',
    'Fort VII', 'Fossoli', 'Grini', 'Gross-Rosen', 'Herzogenbusch', 'Hinzert',
    'Janowska', 'Jasenovac', 'Kaiserwald', 'Kaunas', 'Kemna', 'Klooga', 'Le
    Vernet', 'Majdanek', 'Malchow', 'Maly Trostenets', 'Mechelen', 'Mittelbau
    -Dora', 'Natzweiler-Struthof', 'Neuengamme', 'Niederhagen', 'Oberer Kuhberg',
    'Oranienburg', 'Osthofen', 'PÅ,aszów', 'Ravensbrück', 'Risiera di San Sabba
    ', 'Sachsenhausen', 'Sajmište', 'Salaspils', 'Sobibór', 'Soldau', 'Stutthof
    ', 'Theresienstadt', 'Trawniki', 'Treblinka', 'Vaivara']
```

13.1.4 Normalizing Data

While this dataset is cleaned and good, it has certain limitations. First, it is not complete. This is a list of *key* camps, not all camps. Note that subcamps are left off the list. The second problem we have is that these camps of certain characters in their names that reflect the accent marks or letters that are not in the English alphabet. Some Holocaust texts, however, use only English letters and characters. Therefore, searches for certain words, such as Bełżec will not return a hit in a search for Belzec. It is important, therefore, to make sure both forms of the word are represented in a rules-based search.

We can normalize accented text in Python with a single line of code via the `unidecode` library which can be installed with `pip install unidecode`. Once installed, we can import `unidecode`.

```
import unidecode
```

The `unidecode` library comes with the class `unidecode()` which will take a single argument, the string that we wish to standardize. Let's examine how this works in practice with the camp Bełżec.

```
for camp in ushmm_camps[4:5]:
    normalized = unidecode.unidecode(camp)
    print(camp, normalized)
```

```
Bełżec Belzec
```

Note the standardization of “Bełżec” as “Belzec” in the list. Both forms are now represented in our dataset, meaning we can develop a rules-based EntityRuler that can find both forms of these words in texts. While we were able to solve the first problem, that of

standardized data, we cannot solve the first, however. Should we wish, though, we could add this dataset to our Wikipedia datasets, but as we will see below, a larger dataset presents new challenges.

13.2 The Challenges of Holocaust NER

13.2.1 An Overview of the Problems

Developing an NER pipeline (especially if using a machine learning approach) for Holocaust documents has several issues that must be addressed. They fall into three categories: ethical, linguistic, and toponym resolution.

13.2.2 Ethical

When working with documents as sensitive and delicate as those belonging to the Holocaust, any data scientist or NLP/ML practitioner must consider the serious ethical implications of using machine learning. First, there is the issue of privacy. Named Entity Recognition, by definition, finds and extracted *named* entities. Do the individuals mentioned in the documents wish to be more discoverable? Do they wish to have their names removed from context and stored as metadata? These are some of the ethical questions that one should consider. That is not to say that the process cannot go forward, but if it does, it falls on the part of the creator of the NER to explain the ethical considerations that went into the process and the steps taken to remedy them.

When it comes to names, these can be victims of violence, perpetrators of violence, interviewers of victims, historians, etc. One must ask if it is acceptable to try and use a machine to introduce decisions about the function of the individual in context. Is it ethical or even responsible to try and get a machine to identify victims or perpetrators of violence? Possibly not. It really depends on the desired output and the degree to which the creator of the NER system wishes to justify their actions.

Further, machine learning is imprecise. This means that mistakes will happen. If your NER is trying to identify PEOPLE, GPE, and custom entities, such as CONC_CAMP (concentration camp) and GHETTO, is it ethically acceptable to have errors that result in a potential victim being labeled as a CONC_CAMP? From a machine learning point-of-view, this kind of error is possible and understandable. But imagine if a victim or family member of a victim is trying to use new technology to learn about their past and they see their own name or that of a friend or family member who was a victim of the Holocaust labeled as a CONC_CAMP. Why did this happen? For various reasons. If we think about this from the perspective of the victim, however, this could have a traumatic effect. For these reasons we have an ethical responsibility to introduce barriers to prevent things like this from occurring and/or introducing warnings and explanations for *why* these types of errors may occur.

13.2.3 Linguistic

In addition to the ethical concerns, there are also linguistic issues that make Holocaust data particularly challenging. First, the Holocaust covered a wide section of Europe and, as a result, those who were involved in or affected by the events surrounding the Holocaust were, necessarily of various linguistic backgrounds. In addition to this, many individuals had two mother tongues or spoke multiple languages. This has resulted in documents in the tens of languages. Furthermore, some documents are multilingual. For example, in the

oral testimonies at the USHMM, individuals may give testimony in English, but then use a Yiddish or Polish word. This sudden change in language may or may not be indicated in the notation.

Handling documents of multiple languages is a challenging task in NLP. Fortunately, the new advents of BERT and transformer-based machine learning models may provide the key. For now the steps I have taken in this notebook work strictly for single-language documents. If a word appears here or there in a text that is foreign, the steps I have provided will suffice. If, however, the documents are half German and half English, certain preprocessing steps need to be taken to handle each language separately.

In addition to multiple languages being present in a document, sometimes a document may contain peculiar dialects of a language. This will often return poorer results if an NER model has not been introduced to or experienced such dialectical variances. For example, in oral testimonies, a victim of the Holocaust may refer to their birthplace by a local name that is only used by a select few. It may be a village in Poland that most models will have never encountered. It is important to understand *why* these issues present problems for models and how to overcome them. As we saw in the previous notebook, the easiest way to overcome these issues, is to incorporate them into the training data.

13.2.4 Toponyms

There is one final problem associated with Holocaust documents and that is the problem of toponyms. **Toponyms** are proper nouns that are identical but have radically different meanings depending on the context in which it is used. For example, were I to say “Paris has an excellent climate this time of year.” Where would I be speaking about? If you read this, then you would probably assume that I was speaking of Paris, France. What if I told you that I made that statement while sitting at a coffee in Lexington, KY. The place in which I made that statement may mean that I was speaking, actually, about Paris, KY, a small town outside of Lexington. But now what if I told you that I made that statement in this context. “I just got back from a trip in Texas. Paris has an excellent climate this time of year.” Now, it is a bit more clear that I may be speaking about Paris, TX.

Without context, that single sentence could have many different meanings. Paris, in this example, is a toponym. In Holocaust documents, we often experience toponyms for places that have specific entity labels depending on context. Let’s consider this example. Imagine that I have a model that can identify two types of entities: LOCATION and GHETTO.

“Warsaw is a large city in Poland. During WWII, the Warsaw Ghetto was created.”

As a human, how would you annotate these two sentences? If you said, consider Warsaw in the first sentence a LOCATION and Warsaw in the second sentence a GHETTO, you’d be right. The trick in NER and NLP is to create systems that can perform this task. Holocaust documents are filled with examples just like this. The way in which we overcome these problems is by including correctly **toponym resolved** training sets, or training sets that have been manually annotated to ensure that toponyms are labeled correctly, to the model so it can learn to identify toponyms correctly.

13.3 Creating a Rules-Based Pipeline for Holocaust Documents

In this section, we will walk through how to develop a complete heuristic spaCy pipeline for performing named entity recognition. We will leverage a combination of EntityRuler pipes and custom spaCy pipes that use RegEx to find larger matches.

13.3.1 Creating a Blank spaCy Model

The first thing we need to do is import all of the different components from spaCy and other libraries that we will need. I will explain these as we go forward.

```
import spacy
from spacy.util import filter_spans
from spacy.tokens import Span
from spacy.language import Language
import re
import pandas as pd
import unidecode
from spacy import displacy
```

We will be using Pandas in this notebook to run data checks in a CSV file originally produced by the Holocaust Geographies Collaborative, headed by Anne Knowles, Tim Cole, Alberto Giordano, Paul Jaskot, and Anika Walke. We will be importing RegEx because a lot of our heuristics will rely on capturing multi-word tokens.

Now that we have imported everything, let's create a blank English pipeline in spaCy. As we work through this notebook, we will add pipes to it.

```
nlp = spacy.load("en_core_web_sm")
```

13.3.2 Creating EntityRulers

In the first section of this chapter, we looked at acquiring a camps dataset from the USHMM website. We will now build off that section by normalizing our dataset to include camp names with and without accent marks.

```
ushmm_camps = ['Alderney', 'Amersfoort', 'Auschwitz', 'Banjica', 'Belzec',
    'Bergen-Belsen', 'Bernburg', 'Bogdanovka', 'Bolzano', 'Bor', 'Breendonk',
    'Breitenau', 'Buchenwald', 'Chełmno', 'Dachau', 'Drancy', 'Falstad',
    'Flossenbürg', 'Fort VII', 'Fossoli', 'Grini', 'Gross-Rosen',
    'Herzogenbusch', 'Hinzert', 'Janowska', 'Jasenovac', 'Kaiserwald',
    'Kaunas', 'Kemna', 'Klooga', 'Le Vernet', 'Majdanek', 'Malchow',
    'Maly Trostenets', 'Mechelen', 'Mittelbau-Dora', 'Natzweiler-Struthof',
    'Neuengamme', 'Niederhagen', 'Oberer Kuhberg', 'Oranienburg',
    'Osthofen', 'PÅ,aszów', 'Ravensbrück', 'Risiera di San Sabba',
    'Sachsenhausen', 'Sajmište', 'Salaspils', 'Sobibór', 'Soldau', 'Stutthof',
    'Theresienstadt', 'Trawniki', 'Treblinka', 'Vaivara']

camp_patterns = []
for camp in ushmm_camps:
    normalized = unidecode.unidecode(camp)
    camp_patterns.append({"pattern": camp, "label": "CAMP"})
    camp_patterns.append({"pattern": normalized, "label": "CAMP"})
```

Note that we are creating a list called `camp_patterns`. These are a set of patterns expected by a spaCy EntityRuler. With our patterns created, we can create our EntityRuler. Note that we are placing the ruler before the NER pipe in the model. This is so that our heuristic EntityRuler will be able to annotate the doc container before the NER model. We are also giving it a custom name, `camp_ruler`. This is so that we know precisely what this ruler does in our pipeline when we examine the pipeline structure.

```
camp_ruler = nlp.add_pipe("entity_ruler", before="ner", name="camp_ruler")
```

With the ruler created, we can then populate it with the patterns.

```
camp_ruler.add_patterns(camp_patterns)
```

Now, it is time to test the pipeline to make sure it is working as intended.

```
doc = nlp("Auschwitz was a camp during WWII. Płaszów was also a camp. Another_
→spelling is Plaszow.")
displacy.render(doc, style="ent", jupyter=True)
```

```
<IPython.core.display.HTML object>
```

Note that our pipeline is now correctly identifying all forms of camps as camps correctly. This will ensure that our pipeline can function with different spellings of these words.

13.3.3 Creating Function for Matching RegEx

As noted in the previous section, one of the main challenges with Holocaust-related data is the high degree of toponyms, or words that have identical spelling but different classifications. We will see this issue with several of our classes. Many US ships, for example, bear the name of states. Additionally, named ghettos, such as the Warsaw Ghetto, also can function as a general place (GPE). If we want to use heuristics, therefore, we must create rather robust rules to ensure that our rules have a high degree of accuracy. We can use the following function to do just that by leveraging RegEx.

We will be using this function quite a bit for each of our custom spaCy pipes, so let's explore this function in depth.

```
def regex_match(doc, pattern, label, filter=True,
                context=False, context_list=[], window_start=100, window_end=100):
    text = doc.text
    new_ents = []
    original_ents = list(doc.ents)
    for match in re.finditer(pattern, doc.text):
        start, end = match.span()
        span = doc.char_span(start, end)
        if context==True:
            window = text[start-window_start:end+window_end]
            if any(term in window.lower() for term in context_list):
                if span is not None:
                    new_ents.append((span.start, span.end, span.text))
        else:
            if span is not None:
                new_ents.append((span.start, span.end, span.text))
    for ent in new_ents:
        start, end, name = ent
        new_ent = Span(doc, start, end, label=label)
        original_ents.append(new_ent)
    if filter==True:
        filtered = filter_spans(original_ents)
        final_ents = filtered
    else:
        final_ents = new_ents
    doc.ents = final_ents
    return doc
```

The purpose of this function is to leverage RegEx and spaCy to perform full-text matching. This allows us to use the strengths of both NLP techniques. With RegEx, we can perform complex fuzzy string matching beyond an individual token. With spaCy, we can leverage linguistic knowledge about the text. This function will let us pass a RegEx pattern, a Doc container, and a custom label, then find all items that match the pattern. It will then inject those RegEx matches into spaCy Span containers that will then be manually inserted into the found entities. Let's break this down in the code.

The first line of our code is as follows:

```
def regex_match(doc, pattern, label, filter=True,
                context=False, context_list=[], window_start=100, window_end=100):
```

Here, we are creating a function that will take three mandatory arguments: `doc`, `pattern`, and `label`. The `doc` parameter will be the Doc container that is created by the spaCy `nlp` pipeline. The `pattern` will be the RegEx pattern that we want to use to perform a match. Finally, the `label` is the label that we want to assign to the items matched.

We also have several keyword arguments:

- `filter`
- `context`
- `context_list`
- `window_start`
- `window_end`

These keyword arguments allow for greater versatility in the function. It allows a user to filter the entities and only keep the ones that are longest. Remember, spaCy entities are hard classifications, meaning a token can only belong to a single class, or label. This ensures that when we manually place our found RegEx matches into the Doc container's entities, there are no overlapping entities.

The `context` parameter is a Boolean that allows for a user to pass a `context_list` to the function. This will be a list of words that increases the probability that an identified token belongs to a specific class. We will see this at play when we write rules for the labels `GHETTO` and `SHIP`. As noted above, ghettos frequently function as `GHETTO` and `GPE`. The presence of certain words around a match, such as `ghetto`, radically increase the chance that our hit is a `GHETTO` and not a `GPE`. Likewise, with ships, many ships in the US fleet are named after states. This means we could potentially mark New York, for example, as a `SHIP`, rather than as a `GPE`.

The `window_start` and `window_end` control the window of the context. So it will only look forward and backward n-characters, or the number of characters of text, for the presence of a word in the `context_list`. This is a simple heuristic but one that proves quite effective.

The next bit of code is as follows:

```
text = doc.text
new_ents = []
original_ents = list(doc.ents)
```

Here, we are grabbing the raw text of the Doc container. This will be necessary if the user has enabled context searching. Next, we create an empty list of new entities. This will be the list to which we append our new hits. Finally, we grab the original entities already found by

earlier pipes. It is important to convert this to a list because `doc.ents` is a generator which prevents us from iterating over the data.

The next block of code looks like this:

```
for match in re.finditer(pattern, doc.text):
    start, end = match.span()
    span = doc.char_span(start, end)
    if context==True:
        window = text[start-window_start:end+window_end]
        if any(term in window.lower() for term in context_list):
            if span is not None:
                new_ents.append((span.start, span.end, span.text))
    else:
        if span is not None:
            new_ents.append((span.start, span.end, span.text))
```

Here, we are using `re.finditer()` to find all cases where our RegEx formula has matched in the `doc.text`. We iterate over each match and create a `Span` with the start and end character. If the user has enabled `context` searching, then we look for any term in their `context_list` to see if it appears in the given window they have established. If there is a match, then we append the `Span` to `new_ents`.

Our final section of code reads:

```
for ent in new_ents:
    start, end, name = ent
    new_ent = Span(doc, start, end, label=label)
    original_ents.append(new_ent)
if filter==True:
    filtered = filter_spans(original_ents)
    final_ents = filtered
else:
    final_ents = new_ents
doc.ents = final_ents
return doc
```

Here, we are iterating over each of our newly found matches. We then place the `new_ent` match into the `original_ents` list after assigning it the specified `label`. Finally, we filter the spans so that there are no overlapping entities.

Once complete, we reset the `doc.ents` to the `final_ents` and return the `doc` object back to the user.

13.3.4 Add Pipe for Finding Streets

Let's examine this function in practice by trying to find and extract all streets in a text. Because our data comes from Europe, we need to represent the way streets are represented in multiple languages. Let's look at the example below where we use RegEx to find ways in which a street may appear in German (and Dutch) as well as English. Unlike English, German street names often contain the word street (`strasse`) as a compound in the street name. English, on the other hand, has many different ways to render the concept of a street both in abbreviated and full forms. RegEx is a perfect tool for working with this degree of variance.

```
nlp = spacy.load("en_core_web_sm")
german_streets = r"[A-Z][a-z]*([strasse|straße|straat])\b"
```

(continues on next page)

(continued from previous page)

```

english_streets = r"([A-Z][a-z]*_"
↪(Street|St|Boulevard|Blvd|Avenue|Ave|Road|Rd|Lane|Ln|Place|Pl) (\.)*)"
@Language.component("find_streets")
def find_streets(doc):
    doc = regex_match(doc, german_streets, "STREET")
    doc = regex_match(doc, english_streets, "STREET")
    return (doc)
nlp.add_pipe("find_streets", before="ner")

doc = nlp("Berlinstrasse was a famous street. In America, many towns have a Main_"
↪Street. It can also be spelled Main St.")
displacy.render(doc, style="ent", jupyter=True)

```

<IPython.core.display.HTML object>

13.3.5 Creating a Pipe for Finding Ships

Ships frequently appear in Holocaust documents. We could provide a list of ships to an EntityRuler, but finding one of these patterns isn't enough. We want to ensure that the thing referenced is in fact a ship. Many of these terms could easily be toponyms, or entities that share the same spelling but mean different things in different contexts, e.g. the Ile de France, could easily be a GPE that refers to the area around Paris. General Hosey could easily be a PERSON. These are also known ships. To ensure toponym disambiguation, I set up several contextual clues, e.g. the list of nautical terms. If any of these words appear in area around the hit, then the heuristics assign that token the label of SHIP. If not, it ignores it and allows later pipes or the machine learning model to annotate it.

We can gather a list of known WWII ships from the WWII Database (<https://ww2db.com/>). One of the problems with the lists available here is that they are not "utf-8" encoded. This means that the text is not normalized. Inside this repository is a CSV file with a normalized version of these ship names. This CSV file also stores metadata about ships, namely their class, country of origin, and the year they were built.

In spaCy, we can store this metadata inside the entity Span container with a custom extension. This is done via a function known as a `getter` function that maps the data to the custom extension. Let's see how this works in practice. We will use a synthetic sentence in which the `Activity`, a British vessel, is referenced as is `New York`, a place as well as a US ship active during WWII.

```

def ship_metadata_getter(ent):
    df = pd.read_csv("../data/wwii-ships.csv")
    if ent.label_ == "SHIP":
        ship_name = ent.text.replace("The", "").replace("the", "").strip()
        row = df.loc[df.name==ship_name]
        if len(row) > 0:
            row = row.iloc[0]
            return {"class": row["class"], "country": row.country, "year_built":_
↪row.year}
        else:
            return None
    else:
        return False
nlp = spacy.load("en_core_web_sm")
df = pd.read_csv("../data/wwii-ships.csv")

```

(continues on next page)

(continued from previous page)

```

named_ships_pattern = f"(The|the) ({'|'.join(df.name.tolist())})"

@Language.component("named_ships")
def named_ships(doc):
    nautical = ["ship", "boat", "sail", "captain", "sea", "harbor", "aboard",
    ↪"admiral", "liner"]
    doc = regex_match(doc, named_ships_pattern, "SHIP", context=True, context_
    ↪list=nautical)
    for ent in doc.ents:
        if ent.label_ == "SHIP":
            ent.set_extension('ship_metadata', getter=ship_metadata_getter,
    ↪force=True)
    return (doc)
nlp.add_pipe("named_ships", before="ner")
doc = nlp("The Activity set sail from New York")
for ent in doc.ents:
    print(ent.text, ent.label_, ent._.ship_metadata)

```

```

The Activity SHIP {'class': 'Activity-class Escort Carrier', 'country':
    ↪'United Kingdom', 'year_built': '1942'}
New York GPE False

```

There is a lot happening in this section of code, so let's break down the different components here. The first function is our `getter` function: `ship_metadata_getter()`.

```

def ship_metadata_getter(ent):
    df = pd.read_csv("../data/wwii-ships.csv")
    if ent.label_ == "SHIP":
        ship_name = ent.text.replace("The", "").replace("the", "").strip()
        row = df.loc[df.name==ship_name]
        if len(row) > 0:
            row = row.iloc[0]
            return {"class": row["class"], "country": row.country, "year_built":_
    ↪row.year}
        else:
            return None
    else:
        return False

```

This function takes a single argument, the entity identified from the RegEx match. Here, we strip out the `The` from the entity's name and check to see if the ship found appears in our dataset of known ships. We then isolate the ship's metadata in the dataset and return that data as a dictionary.

```

df = pd.read_csv("../data/wwii-ships.csv")

named_ships_pattern = f"(The|the) ({'|'.join(df.name.tolist())})"

@Language.component("named_ships")
def named_ships(doc):
    nautical = ["ship", "boat", "sail", "captain", "sea", "harbor",
    ↪"aboard", "admiral", "liner"]
    doc = regex_match(doc, named_ships_pattern, "SHIP", context=True, context_
    ↪list=nautical)
    for ent in doc.ents:

```

(continues on next page)

(continued from previous page)

```

if ent.label_ == "SHIP":
    ent.set_extension('ship_metadata', getter=ship_metadata_
→getter, force=True)
return (doc)

```

In this next section of code, we load up the dataset to construct our pattern of names to find via our RegEx function. As we iterate over each entity in our Doc container, we check to see if that entity's label is a `SHIP`. If it is, then we set the extension of `ship_metadata` via the `getter` function referenced above.

Finally, we add our pipe to the spaCy pipeline and test the model with our constructed sentence.

```

nlp.add_pipe("named_ships", before="ner")
doc = nlp("The Activity set sail from New York")
for ent in doc.ents:
    print(ent.text, ent.label_, ent._.ship_metadata)

```

13.3.6 Create Pipe for Identifying a Military Personnel

Military personnel are often referenced in English documents with their military rank. This means that we can use basic RegEx rules for finding and extracting those individuals. When we encounter a sequence of tokens that are capitalized after a military rank, we can make a fairly safe presumption that these tokens will be associated nominally with the rank. For this pipeline, we can again use the WWII Database to isolate and grab all known military ranks during WWII for all countries.

```

nlp = spacy.load("en_core_web_sm")
with open("../data/military_ranks.txt", "r") as f:
    ranks = f.read().splitlines()
military_pattern = f"({'|'.join(ranks)})((?:\s[A-Z])(?:\s[A-Z][a-z\.\.]+)+)@"
@Language.component("find_military")
def find_military(doc):
    doc = regex_match(doc, military_pattern, "MILITARY")
    return (doc)
nlp.add_pipe("find_military", before="ner")

doc = nlp("Captain James T. Kirk commanded the ship.")
displacy.render(doc, style="ent", jupyter=True)

```

```
<IPython.core.display.HTML object>
```

13.3.7 Create Pipe for Identifying Spouses

Often times in historical documents the identity of people are referenced collectively. In some instances, such as those of spouses, this results in the name of the woman being attached to the name of her husband. The purpose of this `SPOUSAL` entity is to identify such constructs so that users can manipulate the output and reconstruct each individual singularly.

```

nlp = spacy.load("en_core_web_sm")
spousal_pattern = r"((Mr|Mrs|Miss|Dr)(\.)*) and ((Mr|Mrs|Miss|Dr)(\.)*)((?:\s[A-Z])(?:\s[A-Z][a-z\.\.]+)+)@"
@Language.component("find_spousal")

```

(continues on next page)

(continued from previous page)

```
def find_spousal(doc):
    doc = regex_match(doc, spousal_pattern, "SPOUSAL")
    return (doc)
nlp.add_pipe("find_spousal", before="ner")

doc = nlp("Mr. and Mrs. Smith are going to the movie")
for ent in doc.ents:
    print(ent.text, ent.label_)
```

Mr. and Mrs. Smith SPOUSAL

13.3.8 Creating a Pipe for Finding Ghettos

In Holocaust documents, identifying ghettos can be difficult. Frequently, a ghetto has the same name as the city in which it is found. To distinguish between the city and the ghetto, speakers and writers often specify that they are referencing the ghetto, not the city generally, by using the word “ghetto” in near proximity to the city name. The below pipe leverages this tendency by looking for capitalized words that precede the word “ghetto”.

```
ghettos_pattern1 = r"[A-Z]\w+((-| )*[A-Z]\w+)* (g|G)hetto"
ghettos_pattern2 = r"(g|G)hetto (of|in|at) ((?:\s[A-Z]) (?:\s[A-Z] [a-z\.]+)+)"
nlp = spacy.load("en_core_web_sm")
@Language.component("find_ghettos")
def find_ghettos(doc):
    doc = regex_match(doc, ghettos_pattern1, "GHETTO")
    doc = regex_match(doc, ghettos_pattern2, "GHETTO")
    return (doc)
nlp.add_pipe("find_ghettos", before="ner")
doc = nlp("The Warsaw Ghetto was in Poland. The ghetto at Warsaw.")
displacy.render(doc, style="ent", jupyter=True)
```

<IPython.core.display.HTML object>

13.3.9 Creating a Geography Pipe

Often in historical documents, the named geographic features of Europe are referenced. We can use both a list and a set of patterns to look for named mountains, forests, rivers, etc.

```
general_pattern = r"([A-Z]\w+)_"
↪(River|Mountain|Mountains|Forest|Forests|Sea|Ocean)*"
river_pattern =
↪"(the|T he) (Rhone|Volga|Danube|Ural|Dnieper|Don|Pechora|Kama|Oka|_
↪Belaya|Dniester|Rhine|Desna|Elbe|Donets|Vistula|Tagus|Daugava|Loire|_
↪Tisza|Ebro|Prut|Neman|Sava|Meuse|KubanRiver|Douro|Mezen|Oder|Guadiana|_
↪Rhône|Kuma|Warta|Seine|Mureş|Northern Dvina|Vychedga|Drava|Po|
Guadalquivir|Bolshoy Uzen|Siret|Maly Uzen|Terek|Olt|Vashka|Glomma|
Garonne|Usa|Kemijoki|Great Morava|Moselle>Main 525|Torne|Dalälven|Inn|
Maritsa|Marne|Neris|Júcar|Dordogne|Saône|Ume|Mur|Ångerman|Klarälven|
Lule|Gauja|Weser|Kalix|Vindel River|Ljusnan|Indalsälven|Vltava|Ponoy|
Ialomița|Onega|Somes|Struma|Adige|Skellefte|Tiber|Vah|Pite|Faxälven|
Vardar|Shannon|Charente|İskar|Tundzha|Ems|Tana|Scheldt|Timiș|Genil|_
↪Severn|Morava|Luga|Argeș|Ljungan|Minho|Venta|Thames|Drina|Jiu|Drin|
Segura|Torne|Osam|Arda|Yantra|Kamchiya|Mesta)"
```

(continues on next page)

(continued from previous page)

```
nlp = spacy.load("en_core_web_sm")

@Language.component("find_geography"):
def find_geography(doc):
    doc = regex_match(doc, general_pattern, "GEOGRAPHY")
    doc = regex_match(doc, river_pattern, "GEOGRAPHY")
    return (doc)
nlp.add_pipe("find_geography", before="ner")
doc = nlp("We entered the Black Forest and eventually walked to the Rhine.")
displacy.render(doc, style="ent", jupyter=True)
```

<IPython.core.display.HTML object>

13.3.10 Seeing the Pipes at Work

Let's now bring all this work together and assemble all our pipes into a single pipeline. We will disable the standard "ner" in this pipeline for now.

```
nlp = spacy.load("en_core_web_sm", disable=["ner"])
camp_ruler = nlp.add_pipe("entity_ruler", before="ner", name="camp_ruler")
camp_ruler.add_patterns(camp_patterns)
nlp.add_pipe("find_streets", before="ner")
nlp.add_pipe("find_spousal", before="ner")
nlp.add_pipe("find_ghettos", before="ner")
nlp.add_pipe("find_military", before="ner")
nlp.add_pipe("find_geography", before="ner")
nlp.add_pipe("named_ships", before="ner")
```

```
/home/wjbmattingly/anaconda3/envs/python-textbook/lib/python3.9/site-packages/
→spacy/language.py:1895: UserWarning: [W123] Argument disable with value [
→'ner'] is used instead of ['senter'] as specified in the config. Be aware_
→that this might affect other components in your pipeline.
warnings.warn(
```

<function __main__.named_ships(doc)>

Now that we have compiled our pipeline, let's test it. We will grab the raw text of a USHMM oral history testimony and process it through our pipeline. As we can see below, we were able to identify three different types of entities: CAMP, GHETTO, and MILITARY. We can also see the specific entities we grabbed. Note that the purpose of a heuristic pipeline is not to grab all entities, rather extract as many true positives as possible at the expense of missing a few examples. This can improve an overall pipeline that also leverages machine learning models. All 56 extracted entities are true positives.

```
import requests
import json
s = requests.get("https://collections.ushmm.org/search/catalog/irn505576.json")

data = json.loads(s.content)
doc = nlp(data["response"]["document"]["fnd_content_web"][0])
ents = list(doc.ents)
ent_types = [ent.label_ for ent in ents]
ent_types = list(set(ent_types))
```

(continues on next page)

(continued from previous page)

```
ent_types.sort()
print(f"Found a Total of {len(list(doc.ents))} in {len(ent_types)} categories:
↪{ent_types}")
for ent in ents:
    print(ent.text, ent.label_)
```

```
Found a Total of 56 in 3 categories: ['CAMP', 'GHETTO', 'MILITARY']
Sobibór CAMP
Sobibór CAMP
Sobibór CAMP
Oberscharführer Gustav Wagner MILITARY
Sobibór CAMP
Sobibór CAMP
Oberscharführer Karl Frenzel MILITARY
Sobibór CAMP
Sobibor CAMP
Scharführer Hubert Gomerski MILITARY
Scharführer Michel MILITARY
Oberscharführer Hermann Michel MILITARY
Sobibór CAMP
Majdanek CAMP
Treblinka CAMP
Treblinka CAMP
Sobibór CAMP
Treblinka CAMP
Belzec CAMP
Belzec CAMP
Sobibór CAMP
Sobibór CAMP
Treblinka CAMP
Belzec CAMP
Untersturmführer Neumann MILITARY
Hauptscharführer Johann Niemann MILITARY
Sobibór CAMP
Sobibór CAMP
Sobibór CAMP
Sobibór CAMP
Auschwitz CAMP
Auschwitz CAMP
Sobibór CAMP
Oberscharführer Gustav Wagner MILITARY
Sobibór CAMP
Scharführer Wolf MILITARY
Unterscharführer Franz Wolf MILITARY
Sobibor CAMP
Sobibór CAMP
ghetto in Helm GHETTO
Sobibór CAMP
Sobibór CAMP
Sobibór CAMP
```

(continues on next page)

(continued from previous page)

Sobibór CAMP
Sobibór CAMP
Sobibor CAMP
Sobibór CAMP
Sobibór CAMP
Sobibór CAMP

This pipeline can now extract structured metadata from unstructured raw text. With heuristics like the ones above, we can construct fairly robust NLP pipelines that leverage lists of known entities as well as spaCy's built in linguistic features to identify and extract useful and relevant from our text data. If we wanted to improve our pipeline, we could also train custom spaCy NER machine learning models, but this is beyond the scope of this book. With a strong basis in spaCy, however, and the plentiful resources available online, this should provide you with a good starting point to begin working with spaCy in your own projects.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

14

Topic Modeling: Concepts and Theory

The purposes of this part of the textbook is fivefold:

1. To introduce the reader to the core concepts of topic modeling and text classification
2. To provide an introduction to three libraries used for traditional topic modeling (Scikit Learn, Gensim, and spaCy) for those with limited Python knowledge
3. To detail the problems and solutions to working with various topic modeling problems
4. To provide an overview of transformer-based topic modeling
5. To provide code that will be easily reproducible for readers who wish to apply these methods to their own domains.

Throughout this part of the textbook, we will work with one dataset, a collection of short descriptions of violence in Apartheid South Africa, which comes from Volume 7 of the Truth and Reconciliation Commission's final report (hereafter, TRC Volume 7). I have chosen this dataset because I have experience with it and I know that the data is perfectly suited to topic modeling as hidden topics are found within it.

In every real world scenario you will know the data that you are working with. In this section I think it's worthwhile to spend a little bit of time talking about the data that we are going to be working with throughout this notebook. The first bit of data that we're going to be working with is a collection of descriptions of violence from apartheid South Africa from the 20th century. They come out of the Truth and Reconciliation Commission in the early 2000s. The TRC was a body put together at the end of the 20th century to catalog the violence that befell victims during the apartheid period. The first dataset is what is known as TRC Volume 7. It is organized as a collection of individuals and a brief description of the violence that befell them and the victim's age. Descriptions of violence are not structured in any way. Instead they have rich metadata within them. This metadata consists of things like dates, places, and other organizations during the 20th century.

The data that I have prepared for you is a JSON file of TRC Volume 7. The JSON file is structured as a dictionary with two keys the first key is names and that corresponds to a list of the victim names. The second key is descriptions. This is the key piece of the data that we will be working with. These are the descriptions of violence and we are trying to identify topics within these descriptions.

14.1 What Is Topic Modeling?

Topic modeling is an approach in NLP where we try to find hidden themes within a collection of documents in a corpus. These themes are the **topics**. Topic modeling is

particularly useful when we do not know the subject of each document in our collection and the corpus is too vast to manually tag each document with a specific topic name. It allows us to automate the tagging of each document without doing so manually.

Topic modeling is distinctly different from approach in NLP known as **text classification**. In text classification, we train a machine learning model to recognize specific known labels. We do this with training data that consists of texts and their corresponding labels. This method of using training data is similar to the machine learning NER approach we saw in Part III of this textbook. When we use labeled date to train a machine learning model, this is known as supervised learning. Topic modeling is an entirely different approach designed to work with an entirely different problem. Since we do not know our labels and do not have training data, supervised learning would not work. Topic modeling is an unsupervised learning approach to finding and identifying the labels.

Today, there are many approaches to topic modeling. Later in this chapter, we will meet LDA (Latent Dirichlet Allocation) model. While useful, this approach to topic modeling has largely been replaced with transformer-based topic models. Nevertheless, it is import to understand LDA topic models in order to appreciate the novelty of transformer-based approaches. Further, LDA topic modeling has a strong history in the digital humanities and it is useful to understand how and why this approach works to understand the literature produced by digital humanities during the early 21st century. Before we can explore each of these, however, it is import to have a key understanding of the themes and concepts of topic modeling. This will be the subject of this section.

14.1.1 Rules-Based Methods

A simple approach to the identification of subjects in a collection of documents is what we would call a rules-based approach. Here, we could argue that documents in which certain words appear indicate a specific topic. This manual approach may yield good results. For example, if I had a 100,000 documents and I knew that some dealt with medical details, I could look for a handful of terms, such as `doctor`, `medicine`, `hospital`, etc. to extract those documents as dealing with a topic of `medical`.

The key problem with this approach is scale. We cannot practically construct lists like this for all possible topics, especially if we do not know the topics found within a corpus. Further, constructing lists like this requires detailed knowledge about a subject. For these reasons, a rules-based approach to classifying documents is not practical for large corpora.

14.1.2 Machine Learning-Based Methods

Another option to identify topics in a text is via a machine learning-based approach. In this method, we do not give a computer system a set of rules, rather we let the computer generate its own rules to identify topics in a corpus. This is done in two different ways: supervised and unsupervised learning.

In supervised learning, we know the key subjects in a corpus. We give a computer system a set of documents with their corresponding label to teach it to identify the characteristics that make that particular topic or class unique. This is mostly used for text classification.

Another approach is via unsupervised learning. In unsupervised learning, we do not know the topics of our documents and, instead, we want let the system identify those topics and cluster (or group together) the ones of a high degree of similarity together. We then examine the words that occur the most frequently in each cluster to get a sense of the topics at hand.

14.1.3 Why Use Topic Modeling?

All of this leads to a vital question: Why use topic modeling? Topic modeling affords researchers the ability to learn a lot about their corpus very quickly. It is often used when the corpus is so large that no single human could read it in a single lifetime.

In both a rules-based and machine learning-based approach, a researcher can see what major subjects are discussed in a corpus. This information can be used to perform targeted research by weeding out the documents that likely do not contain the information the researcher needs. Additionally, the information drawn from topic modeling can be used to make large deductions about the corpus at hand. We will see that topic modeling can be used to draw imprecise or incorrect conclusions.

It is vital, however, to understand the limitations of topic modeling. There is always a potential for the researcher to use topic modeling to validate a wrong presumption about the data. Throughout this part of the textbook, I will emphasize methodological steps that can (and should) be taken to limit these mistakes. Despite this potential for error, topic modeling can provide valuable insight, relatively quickly about a large corpus.

14.2 Topics and Clusters

14.2.1 What Are Topics?

Topics are labels assigned to textual data that detail the subjects contained within a given text. In topic modeling we try to create computer systems that can assign topics the way a human would. In order to understand this process, it's best if we take a step back and think about how we assign topics as humans.

To do this, let's examine these two texts.

Number	Text
Text 1	Thomas enjoys playing basketball. He is an exceptionally good point guard.
Text 2	Victoria enjoys playing baseball. She is an exceptionally good at playing first base.

If I asked you to provide two topics to these texts, what might they be? Basketball and baseball are likely two top candidates. Text 1 would have the topic of basketball, while Text 2 would have the topic of baseball. Now, let's consider these same texts, but add two more into the mix.

Number	Text
Text 1	Thomas enjoys playing basketball. He is an exceptionally good point guard.
Text 2	Victoria enjoys playing baseball. She is an exceptionally good at playing first base.
Text 3	John is a talented chef. He enjoys making pasta professionally.
Text 4	Jeff is a talented cook. He owns a pizzeria.

Now, if I asked you to assign two topics to all four texts, what might those topics be? It is likely that your answer changed. No longer are the two topics of baseball and basketball relevant because Text 3 and Text 4 do not align well with those topics. Instead, a better pair of topics might be sports and cooking, or something like that. What changed? The collection of texts in our corpus changed.

What does this demonstrate? It tells us that topics are corpus-dependent, meaning the topics we assign to texts depend on their context against surrounding texts. The same holds true for topic modeling via computer systems.

14.2.2 What Are Clusters?

In topic modeling, computer systems do not generate topics, rather they generate a list of high concentration words. Texts that share common terms are clustered together by similarity. A **cluster** is nothing more than a collection of similar pieces of data. When we are working with texts, a cluster is a collection of texts that have similar overlapping themes.

There are various ways to cluster textual data that we will explore throughout this chapter.

14.3 Bigrams and Trigrams

Let's take a moment and step away from topic modeling. Instead, let's think about language. The essential medium of topic modeling is texts which are the products of language. And language is nothing more than a collection of words whose usage is rooted around grammar and syntax. A given word's meaning, in other words, is often defined by how, when, and where it is used in a given text. Words can also mean different things when used collectively.

14.3.1 Textual Ambiguity

If I use the word *apple*, you likely are thinking of something like the fruit. *Apple* is a simple word, yet it can mean different things in different contexts. What if I said the following: "My *Apple* is better than a PC." Now, what image comes to mind? Perhaps the computer product.

This is an example of textual ambiguity. Syntactical context helps eliminate textual ambiguity. *Apple* is a single word here that contains a single concept. It's a relatively simple word. Perhaps one of the earliest a native speaker of English learns as a child. And yet it has textual ambiguity. Because "*apple*" is a single word, it is known as a **unigram**. A **unigram** is a single word that represents a single concept.

Textual ambiguity, however, occurs in more dynamic ways when we think about concepts beyond the single span of a single word. In this section, we will focus on two such cases that are essential for natural language processing: bigrams and trigrams. **Bigrams** are two words that contain a distinct meaning when used together, while **trigrams** are three words that contain a distinct meaning when used together.

Understanding bigrams and trigrams are essential because in order for a computer to truly understand language the way a human does, it must be able to understand the nuances of a single word and how a word's meaning not only shifts in context, but shifts in meaning when used in conjunction with other words.

14.3.2 Bigrams

As noted above, a bigram is a combination of two words that have a distinct meaning. To demonstrate this, let us consider quickly the word "French". A single word, that may have multiple meanings. Perhaps the word French refers to the language, perhaps it references a French person.

Let's hold off on the word French for just a moment. Let's now use the word "revolution". Again, there is textual ambiguity, but perhaps I am referencing the concept of revolution in the sense of the Earth.

Now you may already see where I am going with this, but let's now think about what happens when I put those two textually ambiguous unigrams together "The French Revolution". "The" here is a stop word that is frequently dropped in natural language processing, so "French Revolution" is all that we should consider. These two words when combined have a distinct concept.

But we can think about language in even more nuanced ways.

14.3.3 Trigrams

Trigrams, as noted above, are the same as bigrams, except with three words, instead of two. Let's continue with our example of "French". What might you think about if I used the word "army". Perhaps something distinct to your own experiences with the word. For me, as a modern American, I think initially about the American Army in the modern sense of the word. So I may picture something like modern soldiers in my mind's eye.

For others, other images may be more relevant. However, when I use the phrase "The French Revolutionary Army", I now have something defined, something distinct. This is a trigram, a distinct concept consisting of three words that may have individual meanings when used alone.

14.3.4 Why Are These Important?

So, why are bigrams and trigrams so important? The reason comes down to getting machines to understand that when certain words are used together, they bear a distinct meaning. In order to produce a good topic model, therefore, the model must be able to understand and process words in this manner, the way we humans use the language we are trying to get the machine to understand.

14.4 LDA Topic Modeling

A common way to perform topic modeling in the digital humanities is via Latent Dirichlet Allocation (LDA) topic modeling. This method originated in population genomics in 2000¹ as a way to understand larger patterns in genomics data. In 2003², it was applied to machine learning, specifically texts to solve the problem of topic discovery. It leverages statistics to identify topics across a distributed set of data.

14.4.1 Process of Topic Modeling

In order to engage in LDA Topic Modeling, one must clean a corpus significantly. Common steps that we will cover in this section are:

- Removal of Stopwords.
- Lemmatization (optional).
- Removal of Punctuation.

¹<https://academic.oup.com/genetics/article/155/2/945/6048111?login=false>

²<https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>

Stopwords consist of words common across all texts. The official list from the NLTK library is as follows:

```
[‘i’, ‘me’, ‘my’, ‘myself’, ‘we’, ‘our’, ‘ours’, ‘ourselves’, ‘you’, ‘you’re”, “you’ve”, “you’ll”, “you’d”, ‘your’, ‘yours’, ‘yourself’, ‘yourselves’, ‘he’, ‘him’, ‘his’, ‘himself’, ‘she’, “she’s”, ‘her’, ‘hers’, ‘herself’, ‘it’, “it’s”, ‘its’, ‘itself’, ‘they’, ‘them’, ‘their’, ‘theirs’, ‘themselves’, ‘what’, ‘which’, ‘who’, ‘whom’, ‘this’, ‘that’, “that’ll”, ‘these’, ‘those’, ‘am’, ‘is’, ‘are’, ‘was’, ‘were’, ‘be’, ‘been’, ‘being’, ‘have’, ‘has’, ‘had’, ‘having’, ‘do’, ‘does’, ‘did’, ‘doing’, ‘a’, ‘an’, ‘the’, ‘and’, ‘but’, ‘if’, ‘or’, ‘because’, ‘as’, ‘until’, ‘while’, ‘of’, ‘at’, ‘by’, ‘for’, ‘with’, ‘about’, ‘against’, ‘between’, ‘into’, ‘through’, ‘during’, ‘before’, ‘after’, ‘above’, ‘below’, ‘to’, ‘from’, ‘up’, ‘down’, ‘in’, ‘out’, ‘on’, ‘off’, ‘over’, ‘under’, ‘again’, ‘further’, ‘then’, ‘once’, ‘here’, ‘there’, ‘when’, ‘where’, ‘why’, ‘how’, ‘all’, ‘any’, ‘both’, ‘each’, ‘few’, ‘more’, ‘most’, ‘other’, ‘some’, ‘such’, ‘no’, ‘nor’, ‘not’, ‘only’, ‘own’, ‘same’, ‘so’, ‘than’, ‘too’, ‘very’, ‘s’, ‘t’, ‘can’, ‘will’, ‘just’, ‘don’, “don’t”, ‘should’, “should’ve”, ‘now’, ‘d’, ‘ll’, ‘m’, ‘o’, ‘re’, ‘ve’, ‘y’, ‘ain’, ‘aren’, “aren’t”, ‘couldn’t’, “couldn’t”, ‘didn’t’, “didn’t”, ‘doesn’t’, “doesn’t”, ‘hadn’t’, “hadn’t”, ‘hasn’t’, “hasn’t”, ‘haven’t’, “haven’t”, ‘isn’t’, “isn’t”, ‘ma’, ‘mightn’t’, “mightn’t”, ‘mustn’t’, “mustn’t”, ‘needn’t’, “needn’t”, ‘shan’t’, “shan’t”, ‘shouldn’t’, “shouldn’t”, ‘wasn’t’, “wasn’t”, ‘weren’t’, “weren’t”, ‘won’t’, “won’t”, ‘wouldn’t’, “wouldn’t”]
```

Here, let's consider a simple example. We will work with the following document from the TRC Volume 7 dataset.

An ANCYL member who was shot and severely injured by SAP members at Lephoi, Bethulie, Orange Free State (OFS) on 17 April 1991. Police opened fire on a gathering at an ANC supporter's house following a dispute between two neighbours, one of whom was linked to the ANC and the other to the SAP and a councillor.

After we perform the cleaning steps above (with the exception of lemmatization), we have the following result:

```
[‘ancyl’, ‘member’, ‘shot’, ‘severely’, ‘injured’, ‘sap’, ‘members’, ‘lephoi’, ‘bethulie’, ‘orange’, ‘free’, ‘state’, ‘ofs’, ‘17’, ‘april’, ‘1991’, ‘police’, ‘opened’, ‘fire’, ‘gathering’, ‘anc’, ‘supporters’, ‘house’, ‘following’, ‘dispute’, ‘two’, ‘neighbours’, ‘one’, ‘linked’, ‘anc’, ‘sap’, ‘councillor’]
```

Note that we now have a list of only lowercase words split into a list. There are no punctuation marks. In addition, all stop words have been removed.

In addition to cleaning the corpus, one must also reduce all words to unique numbers. These numbers have now relevance to the word, rather they are a single integer. In this approach, a researcher creates a bag-of-words (BOW) dictionary. This dictionary is a collection of integers and their corresponding word. Each text is then reduced to a sequence of numbers. Once we perform this step, our documents now look like this:

```
[(0, 1), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1), (18, 1), (19, 1), (20, 1), (21, 1), (22, 1), (23, 1), (24, 2), (25, 1), (26, 1), (27, 1), (28, 1), (29, 1)]
```

This sequence of numbers refers to the presence of a word in this document. If we were to print off each individual word from our entire dictionary, we would see that it looks like this:

```
0 17
1 1991
2 anc
3 ancy1
4 april
5 bethulie
6 councillor
7 dispute
8 fire
9 following
10 free
11 gathering
12 house
13 injured
14 lephoi
15 linked
16 member
17 members
18 neighbours
19 ofs
20 one
21 opened
22 orange
23 police
24 sap
25 severely
26 shot
27 state
28 supporters
29 two
```

As we will see when we explore Top2Vec later in this chapter, a major drawback to this approach is that we do not retain any information about the syntax or semantics of the language in this approach. Rather, we just know if a word appears or not. Language is, however, far more complex than simply the presence or lack of presence of words. For words to have meaning, we must know how that word is used. Word order, for example, dramatically affects meaning of individual words due to context. Traditional LDA topic modeling does not offer solutions to these problems.

14.4.2 Knowing the Total Number of Topics

Another issue with LDA Topic Modeling is that one must know a specified amount of topics to identify across the entire corpus. Assigning this number can be quite challenging and will require a series of trial-and-error passes. As we will see in the next section, a lot of these issues, from cleaning a corpus to guessing the quantity of topics in the corpus, are eliminated with the use of transformers and more recent machine learning methods and algorithms.

14.4.3 Applying a Topic Model

Once an LDA topic model is trained, it can then be used to interrogate the corpus collectively. One can explore the topics identified and by reading the documents that the model placed in the same category. Depending on the corpus, certain topics can sometimes be difficult to understand. At this stage, the topics do not have actual names, rather they are purely numerical.

If we want to assign labels to each topic, we have two different options available to us. First, a content expert can look at the results and intuitively assign a label that makes the most sense given the corpus. Another approach is through automation. A researcher can find the most common and unique words that appear in a topic and automatically assign them to the label name.

14.4.4 Summary of Key Issues with LDA Topic Modeling

In summary, the concept of topic modeling is a worthwhile endeavor for many text-based classification problems. It is particularly useful when trying to understand large corpora that cannot possibly be read in a reasonable amount of time. It is also suited to understanding how documents cluster together in those corpora to understand large, latent (hidden) themes. LDA Topic Modeling, however, presents many limitations in this process, notably it requires a good deal of preprocessing, knowledge about the quantity of topics one wants to find in the corpus, and the inability to retain semantic and syntactic meaning.

14.5 Creating LDA in Python

In the last section, we learned a lot about the key idea and concepts behind LDA topic modeling. Let's now put those ideas into practice. In this section, we will be using the Gensim library to create our topic model and the PyLDAVis library to visualize it. You can install both libraries with pip with the following commands:

```
pip install gensim
```

and

```
pip install pyldavis
```

We will also need to install NLTK, or the Natural Language Toolkit, in order to get a list of stop words. We can install the library with pip:

```
pip install nltk
```

Once you have installed NLTK, you will need to download the list of English stop words. You can do so with the following command:

```
nltk.download('stopwords')
```

14.5.1 Importing the Required Libraries and Data

Now that we have our libraries installed correctly, we can import everything.

```
import pandas as pd
from nltk.corpus import stopwords
import string
import gensim.corpora as corpora
from gensim.models import LdaModel
import pyLDAvis.gensim_models
pyLDAvis.enable_notebook()
```

This will import the requisite model from Gensim. For this notebook, we will be using the `LdaModel` class. This class allows us to create an LDA model. Before we can populate our model, however, we must first load and clean our data.

```
df = pd.read_csv("../data/trc.csv")
df = df[["Last", "First", "Description"]]
df
```

	Last	First	\
0	AARON	Thabo	Simon
1	ABBOTT	Montaigne	
2	ABRAHAM	Nzaliseko	Christopher
3	ABRAHAMS	Achmat	Fardiel
4	ABRAHAMS	Annalene	Mildred
...
20829	XUZA	Mandla	
20830	YAKA	Mbangomuni	
20831	YALI	Khayalethu	
20832	YALO	Bikiwe	
20833	YALOLO-BOOYSEN	Geoffrey	Yali
		Description	
0	An ANCYL member who was shot and severely inju...		
1	A member of the SADF who was severely injured ...		
2	A COSAS supporter who was kicked and beaten wi...		
3	Was shot and blinded in one eye by members of ...		
4	Was shot and injured by members of the SAP in ...		
...
20829	Was severely injured when he was stoned by a f...		
20830	An IFP supporter and acting induna who was sho...		
20831	Was shot by members of the SAP in Lingelihle, ...		
20832	An IFP supporter whose house and possessions w...		
20833	An ANC supporter and youth activist who was to...		
[20834 rows x 3 columns]			

Our goal in this section will be to model all the descriptions of violence in the TRC Volume 7. We will, therefore, grab all documents and place them into a list.

```
docs = df.Description.tolist()
docs[0]
```

```
"An ANCYL member who was shot and severely injured by SAP members at Lephoi, ↵Bethulie, Orange Free State (OFS) on 17 April 1991. Police opened fire on a ↵gathering at an ANC supporter's house following a dispute between two ↵neighbours, one of whom was linked to the ANC and the other to the SAP and a ↵councillor."
```

14.5.2 Cleaning Documents

Now that we have our documents, let's go ahead and load up our stop words. These will be the words that we remove from our documents.

```
stop_words = stopwords.words('english')
```

Next, we need a function to clean our documents. The purpose of the function below is to take a single document as an input and return a cleaned sequence of words with no punctuation or stop words.

```
def clean_doc(doc):
    no_punct = ''
    for c in doc:
        if c not in string.punctuation:
            no_punct = no_punct+c
    # with list comprehension
    # no_punct = ''.join([c for c in doc if c not in string.punctuation])

    words = no_punct.lower().split()

    final_words = []
    for word in words:
        if word not in stop_words:
            final_words.append(word)

    # with list comprehension
    # final_words = [word for word in words if word not in stop_words]

    return final_words
```

Let's take a look and see what this looks like now when we run it over our first document.

```
cleaned = clean_doc(docs[0])
print(docs[0])
print(cleaned)
```

```
An ANCYL member who was shot and severely injured by SAP members at Lephoi, ↵Bethulie, Orange Free State (OFS) on 17 April 1991. Police opened fire on a ↵gathering at an ANC supporter's house following a dispute between two ↵neighbours, one of whom was linked to the ANC and the other to the SAP and a ↵councillor.
['ancyl', 'member', 'shot', 'severely', 'injured', 'sap', 'members', 'lephoi', ↵'bethulie', 'orange', 'free', 'state', 'ofs', '17', 'april', '1991', 'police ↵', 'opened', 'fire', 'gathering', 'anc', 'supporters', 'house', 'following', ↵'dispute', 'two', 'neighbours', 'one', 'linked', 'anc', 'sap', 'councillor']
```

With our function created, we can now process all our documents. In the line below, we will convert all documents into a list called `cleaned_docs`. This will be our documents that are now represented as a sequence of words.

```
cleaned_docs = [clean_doc(doc) for doc in docs]
```

14.5.3 Create ID-Word Index

Remember, an LDA topic model cannot look at words, rather it must look at numbers in order for the algorithm to work. This means that we need to convert all words into numbers. We can do this with a bag-of-words approach. In this approach, we create an ID-Word Index.

This is essentially a dictionary where each unique word has a unique number. The dictionary is sorted alphabetically.

```
id2word = corpora.Dictionary(cleaned_docs)
```

With the ID-Word Index created, let's query it and see what word maps to index number 250.

```
id2word[250]
```

```
'bmw'
```

As we can see, it is BMW, a political organization in South Africa. Now that we have our dictionary, we can convert all our documents into a sequence of numbers, rather than words. We can do this via the `doc2bow()` method.

```
corpus = [id2word.doc2bow(cleaned_doc) for cleaned_doc in cleaned_docs]
```

The `corpus` object now contains all the documents but represented as a bag-of-words index, rather than as a sequence of words. This is the precise data that our LDA model will expect.

```
print(corpus[0])
```

```
[ (0, 1), (1, 1), (2, 2), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (15, 1), (16, 1), (17, 1), (18, 1), (19, 1), (20, 1), (21, 1), (22, 1), (23, 1), (24, 2), (25, 1), (26, 1), (27, 1), (28, 1), (29, 1)]
```

In order to see what these numbers correspond to, let's take a look at our first document and map each number to the ID-Word Index.

```
for num in corpus[0]:
    num = num[0]
    print(f'{num}\t{id2word[num]}')
```

```

0      17
1      1991
2      anc
3      aencyl
4      april
5      bethulie
6      councillor
7      dispute
8      fire
9      following
10     free
11     gathering
12     house
13     injured
14     lephoi
15     linked
16     member
17     members
18     neighbours
19     ofs
```

(continues on next page)

(continued from previous page)

```

20      one
21      opened
22      orange
23      police
24      sap
25      severely
26      shot
27      state
28      supporters
29      two

```

14.5.4 Creating LDA Topic Model

With our `corpus` now prepared, we can pass it to our LDA Model. We need have a lot of parameters that we can use here, but three things are mandatory:

- `corpus` which will be our `corpus` object.
- `id2word` which will be our ID-Word Index.
- `num_topics` which will be the number of topics we want the model to find.

Now, we can train our topic model.

```
lda_model = LdaModel(corpus=corpus, id2word=id2word, num_topics=100)
```

Training a model may take several seconds to minutes. The more documents you have, the longer it will take. Once the model is trained, we can begin to analyze the model.

14.5.5 Analyze a Document

If we want to get the topics for each document in our `corpus`, we can do so via `get_document_topics()`. This will take a single argument, the `corpus` object. It will return a list of topics and a score for each topic.

```
topics = lda_model.get_document_topics(corpus)
len(topics)
```

20834

We can analyze the topics for our first document.

```
topics[0]
```

```
[ (46, 0.2453331),
(49, 0.1601346),
(57, 0.24741054),
(60, 0.064691655),
(65, 0.09262718),
(67, 0.06555787),
(87, 0.058863647),
(91, 0.037500985) ]
```

These topics were assigned to the following document:

```
print(docs[0])
```

An ANCYL member who was shot and severely injured by SAP members at Lephoi, →Bethulie, Orange Free State (OFS) on 17 April 1991. Police opened fire on a →gathering at an ANC supporter's house following a dispute between two →neighbours, one of whom was linked to the ANC and the other to the SAP and a →councillor.

Let's now print off some of the key words for each of the top 3 topics. For each topic, we will print off the top 10 words with the `get_topic_terms()` method which will take two arguments: the topic number and the amount of words to return.

```
for topic in topics[0][:3]:  
    terms = lda_model.get_topic_terms(topic[0], 10)  
    print(topic)  
    for num in terms:  
        num = num[0]  
        print(num, id2word[num])  
    print()
```

(46, 0.24563931)

19 ofs

27 state

10 free

22 orange

1250 wife

2 anc

4 april

279 13

754 received

17 members

(49, 0.16010018)

26 shot

8 fire

21 opened

24 sap

17 members

888 funeral

13 injured

23 police

85 dead

255 march

(57, 0.24714684)

1200 students

3044 parents

1268 william's

17 members

2040 amabutho

2560 beat

2284 associated

224 school

24 sap

237 assaulted

These results seem acceptable.

14.5.6 Analyze the Topic Model

It is often difficult to gauge the quality of a topic model by simply looking at raw output. For these reasons, it is useful to be able to study a model in its entirety. We can do this with PyLDAVis, a library that was designed to display in two-dimensional space the topics of a topic model and the key words associated with each topic.

```
vis = pyLDAVis.gensim_models.prepare(lda_model, id_docs, id2word, mds="mmds", R=30)
```

```
vis
```

```
PreparedData(topic_coordinates=
    ↪Freq
topic
44   -0.137228  0.331389      1      1  3.560200
32   -0.024440  0.469877      2      1  3.391703
38   -0.058394  0.320393      3      1  3.257853
75   -0.231691  -0.283164     4      1  2.487656
76   -0.021365  0.242234     5      1  2.386751
...
...
...
73   0.149740  0.107291     96      1  0.292228
6    0.413677  0.241535     97      1  0.261419
56   -0.155346  0.103791     98      1  0.244720
40   -0.343639  -0.348971    99      1  0.216838
98   0.407532  0.141427    100      1  0.171992

[100 rows x 5 columns], topic_info=
    ↪ Category logprob loglift
347      ifp  8027.000000  8027.000000 Default  30.0000  30.0000
28   supporters  10739.000000 10739.000000 Default  29.0000  29.0000
994      tvl  1377.000000  1377.000000 Default  28.0000  28.0000
349      natal  4464.000000  4464.000000 Default  27.0000  27.0000
2       anc  11185.000000 11185.000000 Default  26.0000  26.0000
...
...
...
29      two  6.005527  2344.859315 Topic100 -4.8719  0.3982
34      may  4.781922  1535.294958 Topic100 -5.0997  0.5938
17   members  4.859103  8027.289538 Topic100 -5.0837 -1.0443
305      died  3.874836  1066.417025 Topic100 -5.3101  0.7479
100      killed  4.079318  4407.214431 Topic100 -5.2586 -0.6196

[4696 rows x 6 columns], token_table=
    ↪ term
term
697      1  0.059547      1
697      3  0.011579      1
697      4  0.004962      1
697      8  0.047968      1
697      9  0.011579      1
...
...
...
2855     68  0.805436  'sharpeville'
3006     10  0.503830  'terrorist'
518      93  0.614500  'whites'
520      5  0.024642  's
520      7  0.936390  's

[10958 rows x 3 columns], R=30, lambda_step=0.01, plot_opts={'xlab': 'PC1',
    ↪'ylab': 'PC2'}, topic_order=[45, 33, 39, 76, 77, 20, 19, 28, 86, 31, 6, 18,
    ↪9, 50, 59, 30, 67, 94, 90, 3, 22, 87, 54, 10, 15, 40, 60, 36, 29, 4, 100,
    ↪61, 63, 27, 70, 80, 68, 16, 49, 88, 91, 96, 66, 51, 84, 55, 97, 42, 65, 48,
    ↪53, 43, 5, 85, 47, 72, 8, 98, 52, 73, 83, 35, 89, 81, 44, 17, 79, 32, 82,
    ↪34, 71, 2, 93, 11, 62, 46, 75, 26, 95, 23, 56, 12, 38, 37, 69, 24, 13, 64,
    ↪92, 14, 25, 58, 1, 21, 78, 74, 7, 57, 41, 99])
```

Now, we can see all 100 topics plotted in two-dimensional space and we can study the degree to which the words in each topic are unique to that topic when compared with the corpus as a whole.

While useful as a first step to exploring data, we have more powerful ways of clustering documents today. As we will see over the remainder of the chapter, other libraries and approaches that leverage state of the art language models are perhaps better suited to finding and extracting topics from your corpus.

14.6 Transformer Models

Transformer models language models are robust machine learning models that are capable of solving many complex problems. It is beyond the scope of this textbook to explain the architecture of transformers or how they work precisely. Nevertheless, it is important to understand a few things. First, transformer models are particularly suited for working with multi-lingual documents. Second, they are powerful, but slow. Third, the way they represent texts is different from other language models in that they are able to change the vector, or numerical representation, of an individual word based on context.

This makes them particularly suited to the problem of topic modeling because they can convert each document into a numerical representation that is not a sequence of numbers that correspond to the presence of an individual word. Instead, transformer models can convert a document into a deeply semantic representation that retains the context and syntax.

In order to leverage transformer models, we need to install Sentence Transformers (from HuggingFace). We can do that with pip via the command below:

```
pip install sentence_transformers
```

In this section, we will also be working with two other libraries: umap (for representing our complex numerical representation of documents in two-dimensional space) and hdbscan (for finding clusters). UMAP has gained popularity in recent years as a quick, effective, and fairly accurate way to represent higher dimensional data in lower dimensions. In Python, we can access the UMAP algorithm through the UMAP library which can be installed with pip by typing the following command:

```
pip install umap-learn
```

Note the `-learn` after `umap`. This is very important as `umap` is an entirely different library.

```
pip install hdbscan
```

Now that our libraries are installed, we can begin importing them.

14.6.1 Importing Libraries and Gathering Data

```
from sentence_transformers import SentenceTransformer
import umap
import hdbscan
import pandas as pd
```

```
/home/wjbmattingly/anaconda3/envs/python-textbook/lib/python3.9/site-packages/
→tqdm/auto.py:22: TqdmWarning: IPProgress not found. Please update jupyter and_
→ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.
→html
from .autonotebook import tqdm as notebook_tqdm
```

Before we begin leveraging advanced transformer-based topic modeling libraries, like Top2Vec, we should have a good understanding about how they work. Top2Vec leverages three libraries: Sentence Transformers, UMAP, and HDBScan. Here, we will explore each of these so that the reader will have a basic understanding of the theory and methods behind Top2Vec.

```
df = pd.read_csv("../data/trc.csv")
df = df[["Last", "First", "Description"]]
df.head(1)
```

	Last	First	Description
0	AARON	Thabo Simon	An ANCYL member who was shot and severely inju...

```
documents = df.Description.tolist()
```

14.6.2 Embedding the Documents

The first step in using transformers in topic modeling is to convert the text into a vector. We met vectors when we explored LDA topic modeling in the previous section. Arrays for LDA topic modeling were rooted in a bag-of-word (BOW) index. This index, while computationally light, did not retain semantic meaning or word order.

When we are working with transformers, we can create a vector for each document in our dataset. This vector is not an index of the words used, rather it is an embedding for the entire document that contains its semantic usages of words. It also preserves in this same vector space the word order to a degree. This document vector is similar to the word vector that we met in Part III of this textbook. Instead of embedding a single word, however, the entire document receives an embedding. This allows us to mathematically compare documents across an entire corpus.

To convert our documents into vectors, we first need a transformer model. Fortunately, the Sentence Transformer library from HuggingFace allows us to easily load robust pre-trained language models. In our case, we will be using the `all-MiniLM-L6-v2` model. We can load this model by calling the Sentence Transformer class from the `sentence_transformers` library.

```
model = SentenceTransformer('all-MiniLM-L6-v2')
```

Once our model class is created, we can use the `.encode()` method. This method will encode all the documents that we pass to it. In our case, this is the approximately 22,000 descriptions from the TRC dataset. The `.encode()` method takes a single mandatory argument, a list of data to embed.

```
doc_embeddings = model.encode(documents)
```

Now that we have the vectors for each document, let's examine one.

```
doc_embeddings[0][:10]
```

```
array([-0.07123438,  0.00332599, -0.05571468,  0.08363082,  0.09066874,
       0.05503598,  0.08029197,  0.01370712,  0.05912026,  0.06226278],
      dtype=float32)
```

As we can see, this looks remarkably similar to our word embeddings. While this is useful for examining mathematically comparing the similarity between documents, it can be difficult to parse this numerical data visually. For this reason, it is useful to flatten the data into two or three dimensions. This allows the data to be graphed. In the previous section, we learned how to flatten data with PCA. In this section, we will meet a new dimensionality reduction algorithm, UMAP.

14.6.3 Flattening the Data

Once you have installed UMAP correctly, you can access the `UMAP` class. This will take several parameters that can be adjusted to yield different results.

```
umap_proj = umap.UMAP(n_neighbors=10,
                      min_dist=0.01,
                      metric='correlation').fit_transform(doc_embeddings)
```

14.6.4 Isolating Clusters with HDBScan

Once our data has been flattened, we can automatically identify the number of clusters within it and assign documents to each cluster with the HDBScan algorithm.

```
hdbSCAN_labels = hdbscan.HDBSCAN(min_samples=2, min_cluster_size=3).fit_
predict(umap_proj)
print(len(set(hdbscan_labels)))
```

2317

```
df["x"] = umap_proj[:, 0]
df["y"] = umap_proj[:, 1]
df["topic"] = hdbSCAN_labels
df.head(1)
```

	Last	First	Description \
0	AARON	Thabo Simon	An ANCYL member who was shot and severely inju...
	x	y	topic
0	7.330942	-0.935302	-1

14.6.5 Analyzing the Labels

Now that we have the labels loaded into our DataFrame, we can use Pandas to examine that data. Let's grab a topic and examine it. Here, we will examine topic 100.

```
for d in df.loc[df.topic == 100].Description.tolist():
    print(d)
    print()
```

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

An ANC supporter who was shot dead on 6 April 1990 when a group of Inkatha supporters attacked UDF supporters and residents at Mpumalanga, KwaZulu, near Durban, in spite of a heavy police and military presence. Fourteen people were killed and at least one hundred and twenty homes were burnt down. One former IFP member was granted amnesty (AC/1999/0332).

An ANC supporter who was shot and killed when a group of Inkatha supporters and Caprivi trainees attacked a UDF meeting in a house at Mpumalanga, KwaZulu, near Durban, on 18 January 1988. Nine people were killed and an estimated two hundred people were injured in the attack. The group went on to destroy around eight houses. One former Inkatha member was granted amnesty (AC/1999/0332).

An ANC supporter who was shot and killed when a group of Inkatha supporters and Caprivi trainees attacked a UDF meeting in a house at Mpumalanga, KwaZulu, near Durban, on 18 January 1988. Nine people were killed and an estimated two hundred people were injured in the attack. The group went on to destroy around eight houses. One former Inkatha member was granted amnesty (AC/1999/0332).

An ANC supporter who was shot dead on 18 January 1988 when a group of Inkatha supporters, including some Caprivi trainees, opened fire on a UDF meeting in a house at Mpumalanga, KwaZulu, near Durban. Nine people were killed and an estimated two hundred people were injured. The group went on to destroy about eight houses. Amnesty applications were received for this incident.

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

Was abducted, interrogated and stabbed to death by UDF and ANC supporters at Mpumalanga, Natal, in July 1989. He was believed to be an Inkatha member. One UDF and ANC supporter was granted amnesty (AC/200/011).

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

An Inkatha supporter who was stabbed and severely injured by named ANC supporters in Mtwalume, near Umzinto, Natal, in political conflict in the area on 4 February 1990 following the unbanning of political organisations two days earlier. Two UDF supporters were granted amnesty for the attack (AC/2000/041).

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

(continues on next page)

(continued from previous page)

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

Was one of thirteen people killed in an attack by UDF and ANC supporters on Inkatha supporters in the Mahwaqa area, near Port Shepstone, Natal, on 24 March 1990. Two UDF supporters were granted amnesty (AC/2000/041).

Notice that these results for Topic 100 all clearly have overlapping similarity. In some cases, these are identical, in others they are quite similar. Nevertheless, we have one major limitation to this approach, outliers or noise. Let's examine Topic -1.

14.6.6 Outliers (Noise)

```
df.loc[df.topic == -1]
```

	Last	First	\
0	AARON	Thabo Simon	
7	ABRAHAMS	Moegsien	
16	ADAMS	Zwelinzima Sidwell	
31	AGGETT	Neil Hudson	
34	ALBERT	Nombuyiselo Francis	
...	
20804	XULU	Josephina	
20810	XULU	Mzomonje Phineas	
20820	XULU	Sipho Aubrey	
20821	XULU	Sipho Brigitte	
20830	YAKA	Mbangomuni	
		Description	x \
0	An ANCYL member who was shot and severely inju...	7.330942	
7	Was stabbed and stoned to death by a group of ...	5.861767	
16	Was severely beaten and shot in the leg in Gug...	5.548826	
31	Died in detention at John Vorster Square, Joha...	11.213459	
34	Was beaten and stabbed to death on 10 December...	6.867918	
...	
20804	An IFP supporter who had three rondavels burnt...	2.483765	
20810	Was stabbed to death in Inchanga, Natal, on 15...	5.103362	
20820	An ANC supporter who was shot and fatally woun...	7.560771	
20821	An MK operative who was executed in Pretoria C...	3.011799	
20830	An IFP supporter and acting induna who was sho...	8.195865	
	y topic		
0	-0.935302	-1	
7	10.907988	-1	
16	9.517207	-1	
31	8.340458	-1	
34	2.346848	-1	
...	
20804	3.955176	-1	
20810	11.622403	-1	
20820	0.817540	-1	
20821	-0.359574	-1	
20830	2.783923	-1	

[4579 rows x 6 columns]

We have 4,579 outliers, or documents that do not neatly cluster into any one category. This is a clear limitation of this approach. What do we do with these outliers? One option is to assign them to the nearest topic. The process of doing this, however, can be quite complex as the we calculate distance between a document vector and a topic can vary depending on which measurements we wish to use. In the final section of this chapter, we will examine a library that handles this problem for us.

14.7 Top2Vec in Python

Now that we understand the primary concepts behind leveraging transformers and sentence embeddings to perform topic modeling, let's examine a key library and making this entire workflow simplified with just a single line of Python. That library is Top2Vec.

Unlike a traditional topic modeling approach or even the manual approach with transformers that we saw in the previous section, Top2Vec will embed not only all the sentences, but all the words used in your corpus. In addition to this, it will also embed the topics themselves. This means that words, documents, and topics all occupy the same higher-dimensional space. On the surface, this may not seem that important, but it allows us to understand the relationship between three interconnected pieces of the process in topic modeling mathematically. This means, for example, Top2Vec let's us know the mathematical similarity between a given word and a document or a topic in general. Understanding these relationships lets us understand broader patterns in our topics that may otherwise be missed.

That said, Top2Vec does have several drawbacks. First, it will produce a high number of outliers. These are documents that do not fit neatly into any one topic. Top2Vec has a rather unique way of handling outliers, however. It generates an embedding for the topics 0 and above and then assigns outliers to their nearest topic embedding. This means that outliers may be quite pronounced for certain topics. For some workflows, this may not be an issue. Second, Top2Vec can produce results that are occasionally difficult to understand. Certain topics may not seem to have any coherency. This is common in all approaches to topic modeling.

In order to model our data, we once again need to load it. Again, we will use Pandas.

```
import pandas as pd

df = pd.read_json("https://raw.githubusercontent.com/wjbmattingly/bap_sent_
↪embedding/main/data/vol7.json")
df
```

	names \
0	AARON, Thabo Simon
1	ABBOTT, Montaigne
2	ABDUL WAHAB, Zakier
3	ABRAHAM, Nzaliseko Christopher
4	ABRAHAMS, Achmat Fardiel
...	...
21742	ZWENI, Ernest
21743	ZWENI, Lebuti
21744	ZWENI, Louis
21745	ZWENI, Mpantesa William

(continues on next page)

(continued from previous page)

```

21746           ZWENI, Xolile Milton

                           descriptions
0      An ANCYL member who was shot and severely inju...
1      A member of the SADF who was severely injured ...
2      A member of QIBLA who disappeared in September...
3      A COSAS supporter who was kicked and beaten wi...
4      Was shot and blinded in one eye by members of ...
...
21742 One of two South African Police (SAP) members ...
21743 An ANC supporter who was shot dead by a named ...
21744 Was shot dead in Tokoza, Transvaal, on 22 May ...
21745 His home was lost in an arson attack by Witdoe...
21746 A Transkei Defence Force (TDF) soldier who was...

[21747 rows x 2 columns]

```

Once again, we will convert all our descriptions into a single list of documents.

```

docs = df.descriptions.tolist()
docs[0]

```

```

"An ANCYL member who was shot and severely injured by SAP members at_
↳Lephoi, Bethulie, Orange Free State (OFS) on 17 April 1991. Police_
↳opened fire on a gathering at an ANC supporter's house following a_
↳dispute between two neighbours, one of whom was linked to the ANC_
↳and the other to the SAP and a councillor."

```

14.7.1 Creating a Top2Vec Model

Now that we have our data, we can get started with Top2Vec. First, you will need to install the library onto your system. In order to do this, you can use pip with the following command:

```
pip install top2vec
```

This will install Top2Vec as well as all its dependencies. Once it is installed, you will be able to import the Top2Vec class with the following line:

```
from top2vec import Top2Vec
```

```

/home/wjbmattingly/anaconda3/envs/python-textbook/lib/python3.9/site-packages/
↳tqdm/auto.py:22: TqdmWarning: IPProgress not found. Please update jupyter and_
↳ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.
↳html
    from .autonotebook import tqdm as notebook_tqdm

```

Once we have imported the Top2Vec class, we can create our Top2Vec topic model with just one line of Python. We can pass many key word arguments to our Top2Vec class, but the result from the default settings is usually quite good. It may take several minutes or even hours to run the code below, depending on your system requirements.

```
model = Top2Vec(docs)
```

```
2022-11-13 13:58:09,857 - top2vec - INFO - Pre-processing documents for_
↪training
2022-11-13 13:58:11,379 - top2vec - INFO - Creating joint document/word_
↪embedding
2022-11-13 13:58:53,163 - top2vec - INFO - Creating lower dimension embedding_
↪of documents
2022-11-13 13:59:16,683 - top2vec - INFO - Finding dense areas of documents
2022-11-13 13:59:18,018 - top2vec - INFO - Finding topics
```

14.7.2 Analyzing Our Topic Model

Once our Top2Vec topic model has finished training, we can then analyze the results. A good first step is to analyze the topic sizes. We can do so with the model `.get_topic_sizes()`. This method will return a list of two lists: the size of each topic and its number.

```
topic_sizes, topic_nums = model.get_topic_sizes()
```

We can better understand these results by zipping the two lists together and iterating over the first five topics to see how large they are.

```
for topic_size, topic_num in zip(topic_sizes[:5], topic_nums[:5]):
    print(f"Topic Num {topic_num} has {topic_size} documents.")
```

```
Topic Num 0 has 763 documents.
Topic Num 1 has 346 documents.
Topic Num 2 has 247 documents.
Topic Num 3 has 232 documents.
Topic Num 4 has 231 documents.
```

Notice that the higher number the topic number, the smaller the quantity of documents. This is because Top2Vec organizes the topics by size.

We can use the `.get_topics()` method to extract information about our topics from the model. This will take one argument, the quantity of topics you want to examine. It will return three items: a list of words for each topic, a list of word scores for each list, and a list of the topic numbers.

```
topic_words, word_scores, topics = model.get_topics(2)
```

Let's now examine the data for Topic 1.

```
for words, scores, num in zip(topic_words[1:], word_scores[1:], topics[1:]):
    print(f"Topic {num}")
    for word, score in zip(words, scores):
        print(word, score)
```

```
Topic 1
matlala 0.8607659
bk 0.8543051
gamatlala 0.8241883
proposed 0.794874
lebowa 0.74230933
resisted 0.67116654
independence 0.65576303
africa 0.6098439
chief 0.49963036
```

(continues on next page)

(continued from previous page)

```

she 0.39457357
because 0.3908882
goederede 0.36000514
february 0.3470583
mahlangu 0.34650403
home 0.34591654
down 0.34060025
south 0.33364677
dennilton 0.3321176
burnt 0.32735756
would 0.3133811
supported 0.31247702
incorporation 0.3080003
stolen 0.28846616
from 0.28013867
had 0.27430868
it 0.2732216
lost 0.2716071
kwandebelo 0.266503
congress 0.26175743
allegedly 0.25916496
supporters 0.25534543
village 0.25463325
her 0.25239387
mr 0.2508838
homeland 0.24748528
spite 0.24678819
about 0.24613273
maboloko 0.24536304
resulted 0.24119411
into 0.24059604
opposed 0.23594634
pietersburg 0.23558588
alleged 0.22179778
possessions 0.21927707
many 0.21916199
house 0.21861681
imbokodo 0.21361831
leeuwfontein 0.21190558
tribal 0.21142614
zeerust 0.21082786

```

This gives us a word list of 50 words most closely associated with Topic 1. Each word has a score. Remember, words, documents, and topics all occupy the same higher-dimensional space. This is an example of this advantage in practice. We are able to calculate the degree to which a word is relevant to a topic because of its proximity to the center of a topic. The higher the score, the closer a word is to a topic's center and, therefore, in theory more closely represents that cluster.

We can also use the `search_documents_by_topic()` method to learn about the documents that reside within a topic. This method takes two arguments: `topic_num` (the number of the topic you wish to examine) and `num_docs` (the number of documents you want to return). This method will return three things: the documents, the scores, or the degree to which they represent the topic, and their unique ids.

```

documents, document_scores, document_ids = model.search_documents_by_topic(topic_
→num=0, num_docs=10)

```

Now that we have our data, we can iterate them all simultaneously with the example provided in the Top2Vec README on GitHub.

```
for doc, score, doc_id in zip(documents, document_scores, document_ids):
    print(f"Document: {doc_id}, Score: {score}")
    print("-----")
    print(doc)
    print("-----")
    print()
```

```
Document: 15060, Score: 0.9702943563461304
-----
An ANC supporter who had her house burnt down by IFP supporters in Sonkombo, Ndewedwe, ↵KwaZulu, near Durban, on 16 March 1994. See Sonkombo arson attacks.
-----

Document: 5686, Score: 0.965286374092102
-----
She lost her house in Sonkombo, Ndewedwe, KwaZulu, near Durban, in an arson attack by IFP- ↵supporters on 16 March 1994. See Sonkombo arson attacks.
-----

Document: 21471, Score: 0.9650641083717346
-----
An ANC supporter whose house was burnt down by IFP supporters in Sonkombo, Ndewedwe, ↵KwaZulu, near Durban, on 16 March 1994. See Sonkombo arson attacks.
-----

Document: 2957, Score: 0.9647031426429749
-----
His home was burnt down by IFP supporters on 16 March 1994 at Sonkombo, Ndewedwe, KwaZulu, ↵near Durban, in intense political conflict in the area. See Sonkombo arson attacks.
-----

Document: 714, Score: 0.9634069204330444
-----
An ANC supporter who had her home burnt down by IFP supporters at Sonkombo, Ndewedwe, ↵KwaZulu, near Durban, on 16 March 1994. See Sonkombo arson attacks.
-----

Document: 9855, Score: 0.9628040194511414
-----
He had his house burnt down by IFP supporters in Sonkombo, Ndewedwe, KwaZulu, near Durban, ↵on 16 March 1994. See Sonkombo arson attacks.
-----

Document: 759, Score: 0.9617692232131958
-----
Had her home burnt down by IFP supporters at Sonkombo, Ndewedwe, KwaZulu, near Durban, on ↵16 March 1994. See Sonkombo arson attacks.
-----

Document: 519, Score: 0.9617335796356201
-----
An ANC supporter who had her house burnt down by IFP supporters on 16 March 1994 at ↵Sonkombo, Ndewedwe, KwaZulu, near Durban, in intense political conflict in the area. See ↵Sonkombo arson attacks.
-----

Document: 3424, Score: 0.9609926342964172
-----
An ANC supporter whose home was burnt down by IFP supporters on 20 March 1994 at Sonkombo, ↵Ndewedwe, KwaZulu, near Durban. See Sonkombo arson attacks.
-----
```

(continues on next page)

(continued from previous page)

```
Document: 12659, Score: 0.9603915810585022
-----
An ANC supporter who had her house burnt down by IFP supporters in Sonkombo, Ndwedwe,-
→KwaZulu, near Durban, on 20 March 1994. See Sonkombo arson attacks.
-----
```

These results are good. These are all individuals who have near identical descriptions in the TRC Volume 7. In other words, Top2Vec has clearly isolated documents of identical or overlapping similarity.

14.7.3 Working with Bigrams and Trigrams

Top2Vec also wraps around the Gensim library and allows users to automatically create bigrams and trigrams. We can initiate this process by setting the keyword argument `ngram_vocab` to True.

```
model_ngram = Top2Vec(docs, ngram_vocab=True)
```

```
2022-11-13 13:59:18,293 - top2vec - INFO - Pre-processing documents for_
→training
2022-11-13 13:59:19,889 - top2vec - INFO - Creating joint document/word_
→embedding
2022-11-13 14:00:07,661 - top2vec - INFO - Creating lower dimension embedding_
→of documents
2022-11-13 14:00:19,175 - top2vec - INFO - Finding dense areas of documents
2022-11-13 14:00:19,748 - top2vec - INFO - Finding topics
```

Now that we have our `model_ngram` trained, let's explore Topic 1 once again.

```
topic_sizes_ngram, topic_nums_ngram = model_ngram.get_topic_sizes()
```

```
topic_words_ngram, word_scores_ngram, topics_ngram = model_ngram.get_topics(2)
for words, scores, num in zip(topic_words_ngram[1:], word_scores_ngram[1:],_
→topics_ngram[1:]):
    print(f"Topic {num}")
    for word, score in zip(words, scores):
        print(word, score)
```

```
Topic 1
island 0.7406236
imprisonment 0.7383562
years 0.7350636
robben 0.73397404
half years 0.7247492
years imprisonment 0.7206887
sentenced 0.6986825
served 0.695978
arrested 0.66353947
charged 0.66336066
banned 0.66275764
imprisoned 0.6580816
five years 0.6498868
suspended sentence 0.6363463
prison sentence 0.6340719
```

(continues on next page)

(continued from previous page)

```

trial 0.6326254
sentence 0.62952983
activist 0.6273678
months 0.62542427
detained 0.6249351
prison 0.6236554
until 0.6223862
poqo 0.6147149
custody 0.6120131
year sentence 0.6114016
held 0.6040434
worcester 0.60011864
pac 0.5935757
released 0.5893775
poqo activities 0.58773917
charges 0.5808439
zwelethemba worcester 0.57607645
zweletemba worcester 0.575092
arrest 0.5745207
robben island 0.571052
under 0.56998795
worcester cape 0.56770444
detention 0.5670366
with public 0.56048626
ashton 0.55939454
later acquitted 0.5478358
tortured 0.5445701
charged with 0.54265773
act 0.5392955
interrogation 0.5369543
without 0.53288996
paarl 0.53190106
public 0.5305038
prison paarl 0.5296649
year prison 0.5292457

```

It is important to note that this Topic 1 is entirely different from the Topic 1 of our first model as both the initialization of the process is random and the data used for embedding documents is also altered (because we are accepting bigrams). Our data does reflect a deeper representation of our corpus. `Prison sentence`, for example, is a collective word that has a distinct meaning.

14.7.4 Saving and Loading a Top2Vec Model

Once we have a model that we are happy with, we can save it using the `save()` method.

```
model_ngram.save("../data/top2vec_ngram")
```

Likewise, we can load up our previous model with `load()`.

```
new_model = Top2Vec.load("../data/top2vec_ngram")
```

And for good measure we can run the same code as above to print off our bigrams once again.

```
topic_words_ngram, word_scores_ngram, topics_ngram = new_model.get_topics(2)
for words, scores, num in zip(topic_words_ngram[1:], word_scores_ngram[1:],  
    topics_ngram[1:]):
    print(f"Topic {num}")
    for word, score in zip(words, scores):
        print(word, score)
```

```
Topic 1
island 0.7406236
imprisonment 0.7383562
years 0.7350636
robben 0.73397404
half years 0.7247492
years imprisonment 0.7206887
sentenced 0.6986825
served 0.695978
arrested 0.66353947
charged 0.66336066
banned 0.66275764
imprisoned 0.6580816
five years 0.6498868
suspended sentence 0.6363463
prison sentence 0.6340719
trial 0.6326254
sentence 0.62952983
activist 0.6273678
months 0.62542427
detained 0.6249351
prison 0.6236554
until 0.6223862
poqo 0.6147149
custody 0.6120131
year sentence 0.6114016
held 0.6040434
worcester 0.60011864
pac 0.5935757
released 0.5893775
poqo activities 0.58773917
charges 0.5808439
zwelethemba worcester 0.57607645
zweletemba worcester 0.575092
arrest 0.5745207
robben island 0.571052
under 0.56998795
worcester cape 0.56770444
detention 0.5670366
with public 0.56048626
ashton 0.55939454
later acquitted 0.5478358
tortured 0.5445701
charged with 0.54265773
act 0.5392955
interrogation 0.5369543
without 0.53288996
paarl 0.53190106
public 0.5305038
prison paarl 0.5296649
year prison 0.5292457
```

If you are working with topic modeling or just want to get a general sense of how your documents align in a large corpus, then Top2Vec is a great solution to your situation. It offers a lot of features for a single line of code. There are other options available that do similar things, notable BerTopic. One should never rely on a single topic modeling approach, nor should one rest arguments heavily upon the results. Topic modeling, regardless of method, is imperfect. It is, however, a great way to explore your data and begin finding patterns that you may otherwise not notice.

15

Text Analysis with BookNLP

In this chapter, we will learn about BookNLP, a powerful Python library for text analysis on large text documents, namely works of fiction. We will learn about the core aspects of BookNLP, how to process texts with it, and how to interpret the output files that it produces. Finally, we will create a few functions for better interpreting the results of the output files.

15.1 Introduction to BookNLP

15.1.1 What Is BookNLP?

BookNLP is a new Python library created by David Bamman. It was originally created as a Java library in 2014 under the same name, BookNLP by David Bamman, Ted Underwood, and Noah Smith (see, David Bamman, Ted Underwood and Noah Smith, “A Bayesian Mixed Effects Model of Literary Character,” ACL 2014). While Java is a powerful coding language, both in speed and ease-of-use, not many digital humanists code in Java primarily. This section will deal strictly with the Python library.

In the documentation, Bamman states:

“BookNLP is a natural language processing pipeline that scales to books and other long documents (in English), including:

- Part-of-speech tagging
- Dependency parsing
- Entity recognition
- Character name clustering (e.g., “Tom”, “Tom Sawyer”, “Mr. Sawyer”, “Thomas Sawyer” -> TOM_SAWYER) and coreference resolution
- Quotation speaker identification
- Supersense tagging (e.g., “animal”, “artifact”, “body”, “cognition”, etc.)
- Event tagging
- Referential gender inference (TOM_SAWYER -> he/him/his)”

Unlike its predecessor, the Java library, the Python library leverages the Python NLP library, spaCy, and the Python Sentence Transformers library from HuggingFace, rather than Stanford, to perform many of these tasks. In the last few years, spaCy has proven itself as a dominate force within the NLP community, outperforming many of its predecessors in accuracy and in its ability to perform at scale. HuggingFace is a library that allows one to create and leverage large and powerful transformer language models. It also allows users

to store these models in the cloud which are too large to store within GitHub or other comparable repositories.

BookNLP delivers in all the things it sets out to do, though it currently only supports English. Because it leverages transformer models, BookNLP's results can generalize well on non-standard English. I have seen it perform quite well with the South African dialect of English, by correctly identifying out-of-vocabulary (OOV) words, specifically the correct labeling of Afrikaans words for minivans as vehicles.

Although only available in English as of March 2022, there are clear plans to expand the library to include Spanish, Japanese, Russian, and German, as per their recent NEH grant, awarded in September 2020.

15.1.2 Why Books and Larger Documents?

Both the documentation and this textbook emphasize the word large here. The reason? Because most language models do not perform well with larger documents. Old RNN-based language models had a hard time remembering earlier words and while newer transformer-based models, such as BERT, have a larger memory and can look forwards and backwards, the size of the input they can take in is only 512 words. For larger documents, therefore, different solutions (and libraries) should be considered. This is where BookNLP comes in. It also addresses several problems associated with books and larger documents, such as:

- Characters (and people) are referenced by different names. BookNLP solves this problem with name clustering and coreference resolution. This is a task in NLP where we try and find all uses a name and correctly assign them to the same identifier, such as Harry, Harry Potter, and Mr. Harry Potter all being the same person, Harry Potter.
- An adjacent problem is referential gender inferencing. Like coreference resolution, often times in a book or larger document, a person will be referred to as a pronoun. This is where referential gender inferencing comes in. This allows a user to correctly assign the antecedent or postcedent to the correct pronoun. When done successfully, this also allows you to make decisions about the gender of the character or person based on how they are referenced in the text. Because this task is so delicate, given the delicate nature of assigning gender, BookNLP fortunately gives users the data with each pronoun used to reference a character and also includes non-binary pronouns.
- Another issue is quotation speaker identification. This is when we need to understand who is speaking, so that we can correctly link characters to their dialogues. It is possible to do this with spaCy, but it is extremely difficult to do well. BookNLP does a remarkable job of handling this problem and it does it with a fair degree of accuracy, from what I have seen.
- Event tagging is another key issue with longer documents and books. There are machine learning models that find events and you can easily cultivate a list of domain-specific events to improve a pipeline, but for BookNLP event is defined more broadly. From my experience, it is more based around key actions, rather than named events (as it is in named entity recognition). This has a tangential benefit known as triple extraction. In my opinion, it might be a bit better to view BookNLP events through this lens. Triple extraction is when we try and extract three pieces of information, such as (Actor, Action, Recipient) or (Actor, IS, Something). With these types of tuples, we can construct a knowledge tree about a corpus fairly easily. This a very challenging problem in NLP because triple extraction can be very domain-specific. BookNLP provides a great starting place for triple extraction with its events.

15.1.3 How to Install BookNLP

If you are using Linux, installation will be easy. Use the following code:

```
pip install booknlp
```

You can opt to create a custom environment (recommended but not necessary). If you are using Windows, however, as of March 3 2022, you will need to do a few additional steps which I have documented in this video below:

```
from IPython.display import HTML
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/
→315ERF3QX0M" title="YouTube video player" frameborder="0" allow="accelerometer;_
→autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture"_
→allowfullscreen></iframe>')
```

```
<IPython.core.display.HTML object>
```

15.2 Getting Started With BookNLP

Now that you have successfully installed BookNLP and the requisite spaCy small model, we can now dive in! I have tested BookNLP on Linux (Ubuntu 20.04) and Windows 10. The code provided in this section has worked on both systems once installed correctly. If you are receiving errors, feel free to submit an issue or pull request with the GitHub icon in the top right corner of the page.

In this section, I will introduce you to a lot of the code that can be found in the README.md file on the official repository for BookNLP as well as the official Google Colab Notebook. I always believe in following the docs early in this textbook, so that you can follow along a bit more easily.

The goal of this section is to teach you how to run BookNLP over a book (stored as a text file) and generate the requisite output: a series of files (explained in detail in the next section).

Unlike most NLP libraries, the main goal of BookNLP is to generate a series of files stored in a subdirectory. These files will contain all the requisite data to then begin analyzing the processed text.

15.2.1 Importing BookNLP and Creating a Pipeline

As with all libraries, we need to import BookNLP into our Python file (or notebook) in order to work with it. We will specifically need the BookNLP class, so let's go ahead and import everything with the command below.

```
from booknlp.booknlp import BookNLP
```

```
INFO:tensorflow:Enabling eager execution
INFO:tensorflow:Enabling v2 tensorshape
INFO:tensorflow:Enabling resource variables
INFO:tensorflow:Enabling tensor equality
INFO:tensorflow:Enabling control flow v2
using device cuda
```

Now that we have imported the BookNLP class correctly, it's time to create the BookNLP pipeline and model in memory. In order to create a BookNLP pipeline, you will want to create a dictionary. Stick with the documentation here and call this object "model_params". This will be a dictionary that will have two keys: pipeline, and model.

- Pipeline will take a value that is a string within which are commas that separate the different components. You can play around with this later, but for now let's work with the entire pipeline which consists of:

- entity
- quote
- supersense
- event
- coref

- Model will have a key that states the size of the model. For now, use big as we are just trying to follow the docs and create an output that we can analyze in the next section.

```
model_params={
    "pipeline": "entity,quote,supersense,event,coref",
    "model": "big"
}

booknlp=BookNLP("en", model_params)
```

```
{'pipeline': 'entity,quote,supersense,event,coref', 'model': 'big'}
```

```
c:\users\wma22\appdata\local\programs\python\python39\lib\site-packages\spacy\util.py:833: UserWarning: [W095] Model 'en_core_web_sm' (3.1.0) was trained with spaCy v3.1 and may not be 100% compatible with the current version (3.2.2). If you see errors or degraded performance, download a newer compatible model or retrain your custom model with the current spaCy version. For more details and available updates, run: python -m spacy validate
warnings.warn(warn_msg)
```

```
--- startup: 7.314 seconds ---
```

15.2.2 Setting up the File and Directories

Next, we need to setup two objects: the `input_file` and the `output_directory`. The input file will be a string that corresponds to the location of your text file that contains the book or large document you want to analyze. For simplicity sake, I have placed our input file in `data`.

The output directory is the directory into which you want BookNLP to dump all the generated data files. Although there are ways to generate folders programmatically with `os`, I recommend to keep things simple and make the directories manually for now. In our case, I have already created a subfolder within `data` entitled "`harry_potter`". This is where the files I generate will be stored.

Finally, let's make a third object that will be a string. This will be our `book_id`. Think of this as a unique name that will be the basis for how the external files are named.

```
input_file="..../data/harry_potter_cleaned.txt"  
output_directory="..../data/harry_potter"  
book_id="harry_potter"
```

15.2.3 Running the Pipeline

Now that we have created the model and the necessary object names, let's process our text! To do this, we will use `booknlp.process()`. This will take three arguments, all of which we have already created:

- `input_file`
- `output_directory`
- `book_id`

The code below will take some time to run. Even on a powerful computer, it will take a few minutes for a 100k-word file. Do not be surprised if this takes 10+ minutes. For benchmarks, you can see the repository.

```
booknlp.process(input_file, output_directory, book_id)
```

```
--- spacy: 18.936 seconds ---  
--- entities: 88.072 seconds ---  
--- quotes: 0.105 seconds ---  
--- attribution: 28.766 seconds ---  
--- name coref: 0.545 seconds ---  
--- coref: 28.508 seconds ---  
--- TOTAL (excl. startup): 165.277 seconds ---, 99256 words
```

If all goes well, you should see an output like the one above that lists each process after it completes with the corresponding time it took to complete the task. You should also see the files generated in the output directory.

15.3 The Output Files

The output from the BookNLP pipeline is three types of files: TSV files (`.tokens`, `.entities`, `.quotes`, `.supersense`), a JSON file (`.book`) and an HTML file (`.book.html`). A good way to think about a TSV is as a CSV where tabs are used to separate tabular data, rather than commas. Essentially, this is a dataset that can be viewed and analyzed in Excel. A JSON file is a bit different. It stores data as you would expect to see it in Python, e.g. dictionaries, lists, etc.

The goal of this section is to explain what each of these files contains so that in the next few chapters, we can start extracting important data from them.

15.3.1 The .tokens File

The very first file that we should analyze is the .tokens file. Essentially, this is a tab separated value file (TSV) that contains all the tokens on each line of the file and some important data about those tokens. A token is a word or punctuation mark within a text. The very first line of the file will look something like this:

```
paragraph_ID sentence_ID token_ID_within_sentence token_ID_within_document word_
→lemma byte_onset byte_offset POS_tag fine_POS_tag dependency_relation_
→syntactic_head_ID event
```

As this can be a bit difficult to parse, I am going to load it up as a TSV file through Pandas so we can analyze it a bit better.

```
import pandas as pd

df = pd.read_csv("data/harry_potter/harry_potter.tokens", delimiter="\t")
df
```

	paragraph_ID	sentence_ID	token_ID_within_sentence	\		
0	0	0	0			
1	0	0	1			
2	0	0	2			
3	0	0	3			
4	0	0	4			
...		
99251	2995	6885	10			
99252	2995	6885	11			
99253	2995	6885	12			
99254	2995	6885	13			
99255	2995	6885	14			
	token_ID_within_document	word	lemma	byte_onset	byte_offset	\
0	0	Mr.	Mr.	0	3	
1	1	and	and	4	7	
2	2	Mrs.	Mrs.	8	12	
3	3	Dursley	Dursley	13	20	
4	4	,	,	20	21	
...
99251	99251	Dudley	Dudley	438929	438935	
99252	99252	this	this	438936	438940	
99253	99253	summer	summer	438941	438947	
99254	99254	438947	438951	
99255	99255	\t	438951	438952	PUNCT	
	POS_tag	fine_POS_tag	dependency_relation	syntactic_head_ID	event	
0	PROPN	NNP	nsubj	12	0	
1	CCONJ	CC	cc	0	0	
2	PROPN	NNP	compound	3	0	
3	PROPN	NNP	conj	0	0	
4	PUNCT	,	punct	0	0	
...
99251	PROPN	NNP	pobj	99250	0	
99252	DET	DT	det	99253	0	
99253	NOUN	NN	npadvmod	99245	0	
99254	PUNCT	.	punct	99243	0	
99255	"	punct	99243	0	NaN	
	[99256 rows x 13 columns]					

As you can see from the output above, we have something that looks like Excel, or tabular data. Let's break this down a bit and explain what each column represents:

- paragraph_ID – the index of the paragraph, starting at paragraph 1 being 0 and moving up to 3031 in our case.
- sentence_ID – same as the paragraph_ID, but with sentences.
- token_ID_within_sentence – same as the two above, but with a token count by sentence, resetting with each sentence.
- token_ID_within_document – same as above, but where tokens keep going up in value throughout the whole document, starting at 0 and ending, in our case, at 99400.
- word – this is the raw text of the word.
- lemma – this is the root of the word.
- byte_onset – think of this as the start character index.
- byte_offset – think of this as the concluding character index.
- POS_tag – the Part of Speech (based on spaCy).
- fine_POS_tag – a more granular understanding of the Part of Speech.
- dependency_relation – this is equivalent to spaCy's dep tag.
- syntactic_head_ID – This points to the head of the current token so that you can understand how a token relates to other words in the sentence.
- event – this tells you if the token is a trigger for an EVENT or not. You will see, 0, EVENT, or NaN here.

15.3.2 The .entities File

Let's do the same thing with the .entities file now!

```
df_entities = pd.read_csv("data/harry_potter/harry_potter.entities", delimiter="\t")
df_entities
```

	COREF	start_token	end_token	prop	cat	text
0	364	0	0	PROP	PER	Mr.
1	92	2	3	PROP	PER	Mrs. Dursley
2	1	9	10	PROP	FAC	Privet Drive
3	365	17	17	PRON	PER	they
4	366	23	23	PRON	PER	you
...
15858	2355	99227	99227	PRON	PER	They
15859	2351	99231	99231	PRON	PER	we
15860	441	99239	99239	NOM	FAC	home
15861	98	99241	99241	PRON	PER	I
15862	95	99251	99251	PROP	PER	Dudley

[15863 rows x 6 columns]

If you get an error that looks like this:

```
ParserError: Error tokenizing data. C error: EOF inside string starting at row 577
```

Fear not! This happens sometimes when the `.entities` file is corrupted with something like a " mark. You simply need to go into the file and remove the character that is causing the error. Use the row number as an indicator of where to go in the text file. Remember, add one row because row 1 is the header data.

Before:

577	478	3784	3784	PRON	PER	she
578	2385	3793	3794	NOM	PER	" everyone

After:

577	478	3784	3784	PRON	PER	Dumbledore
578	2385	3793	3794	NOM	PER	everyone
579	478	3799	3799	PRON	PER	she
580	105	3807	3807	PROP	PER	Dumbledore

Let's return to our data.

```
df_entities
```

	COREF	start_token	end_token	prop	cat	text
0	364	0	0	PROP	PER	Mr.
1	92	2	3	PROP	PER	Mrs. Dursley
2	1	9	10	PROP	FAC	Privet Drive
3	365	17	17	PRON	PER	they
4	366	23	23	PRON	PER	you
...
15858	2355	99227	99227	PRON	PER	They
15859	2351	99231	99231	PRON	PER	we
15860	441	99239	99239	NOM	FAC	home
15861	98	99241	99241	PRON	PER	I
15862	95	99251	99251	PROP	PER	Dudley

[15863 rows x 6 columns]

Here we see all the entities found within the text. In our case, we have 15,863 entities in the entire book. It is important to remember that some of these are, of course. Before we get to that, though, let's break down the columns.

- COREF – this is a COREF id that is a unique identifier for the person. This number will be used elsewhere to reference a person, such as in the `.quotes` file, to link the speaker with the block of text. It should be noted, that COREF is one of the more challenging problems in NLP. Expect this to not be even close to 90% accurate, rather around the 70% accuracy range, particularly when pronouns are used for the entity.
- start_token – this is the start token of the entity name.
- end_token – this is the end token of the entity name. Single token entities will have the same start and end, while multi-word tokens (MWTs) will increase by one for each additional token.
- prop – this will tell you if it is a PROP (proper noun) or PROPN (pronoun), or other categories.

- cat – this will be the entity type (in spaCy terms). BookNLP includes a few other useful categories, notable VEH for vehicle.
- text – this is the raw text that corresponds to the entity.

15.3.3 The .quotes File

The .quotes file will contain all the quotes in the book. Let's take a look at this data like we did above.

```
df_quotes = pd.read_csv("data/harry_potter/harry_potter.quotes", delimiter="\t")
df_quotes
```

	quote_start	quote_end	mention_start	mention_end	mention_phrase	\	
0	434	438	443	443	he		
1	1089	1108	1085	1085	they		
2	1343	1346	1347	1347	he		
3	1416	1460	1405	1405	he		
4	1603	1606	1608	1609	Mr. Dursley		
...	
2322	99133	99146	99147	99147	He		
2323	99163	99172	99161	99161	Hermione		
2324	99173	99184	99186	99186	Hermione		
2325	99202	99208	99210	99210	Harry		
2326	99226	99255	99210	99210	Harry		
char_id	quote						
0	93	Little tyke ,					
1	417	The Potters , that 's right , that 's what I ...					
2	93	Sorry ,					
3	435	Do n't be sorry , my dear sir , for nothing c...					
4	93	Shoo !					
...					
2322	119	Hurry up , boy , we have n't got all day .					
2323	220	See you over the summer , then .					
2324	220	Hope you have -- er -- a good holiday ,					
2325	98	Oh , I will ,					
2326	98	They do n't know we 're not allowed to use ma...					

[2327 rows x 7 columns]

In our case, we have 2,326 quotes in the entire book. Each quote contains some important metadata:

- quote_start – the start token of the quote.
- quote_end – the end token of the quote.
- mention_start – this is the start token of the speaker entity.
- mention_end – this is the end token of the speaker entity.
- char_id – this will be the unique identifier we saw above in the .entities file so that you can perform COREF and find all dialogues for a single character. Remember, there WILL LIKELY BE ERRORS here. Sometimes you may need to manually align two entity ids as a single character (as we will see).
- quote – this is the raw text of the quote.

15.3.4 The .supersense File

The final TSV file that we have is the `.supersense` file. This is something that I think is quite unique to BookNLP and an absolute delight to have. Here we have all supersense text found. A good way to think about supersense is as a more broadly defined entities file. Here, we not only have entities, like people, places, etc, but also things like “perception”.

```
df_supersense = pd.read_csv("data/harry_potter/harry_potter.supersense",  
                           delimiter="\t")  
df_supersense
```

	start_token	end_token	supersense_category	text
0	0	0	noun.person	Mr.
1	2	3	noun.person	Mrs. Dursley
2	6	6	noun.quantity	number
3	7	7	noun.quantity	four
4	9	10	noun.location	Privet Drive
...
29313	99239	99239	noun.location	home
29314	99245	99245	verb.perception	have
29315	99249	99249	noun.act	fun
29316	99251	99251	noun.person	Dudley
29317	99253	99253	noun.time	summer
[29318 rows x 4 columns]				

We can see that we have 29,318 different supersense items with four pieces of data:

- `start_token` – this is the start token for the supersense text.
- `end_token` – this is the end token for the supersense text.
- `supersense_category` – this is the part of speech and category to which the supersense belongs.
- `text` – this is the raw text of the supersense.

15.3.5 The .book File

Now that we have looked at all the TSV files, let’s take a look at the `.book` file. This is a large JSON file that contains information structured around the characters. In the next few chapters, we will learn a lot more about this file, but for now, let’s explore how it is structured.

```
import json  
  
with open ("data/harry_potter/harry_potter.book", "r") as f:  
    book_data = json.load(f)  
book_data.keys()  
  
dict_keys(['characters'])
```

It is a giant dictionary with one key: `characters`. The value of `characters` is a list. Let’s check out its length.

```
len(book_data["characters"])
```

723

So, we have 723 unique characters throughout the book. Again, expect errors here. For each character, we have a dictionary with 8 keys:

```
book_data["characters"][0].keys()
```

```
dict_keys(['agent', 'patient', 'mod', 'poss', 'id', 'g', 'count', 'mentions'])
```

These keys are as follows:

- agent – actions that character does.
- patient – actions done to that character.
- mod – adjectives that describe them in the text.
- poss – things the entity has (very broadly defined), e.g. relatives like aunt, uncle; or parts of the body, e.g. head, back, etc.
- id – their unique id (as seen above).
- g – analysis about gender pronouns used.
- count – number of times the entity appears.
- mentions – how the character is referenced.

```
book_data["characters"][0]["agent"][:1]
book_data["characters"][0]["patient"][:1]
book_data["characters"][0]["mod"][:1]
book_data["characters"][0]["poss"][:1]
book_data["characters"][0]["id"]
book_data["characters"][0]["g"]
book_data["characters"][0]["count"]
book_data["characters"][0]["mentions"].keys()
```

```
dict_keys(['proper', 'common', 'pronoun'])
```

```
book_data["characters"][0]["mod"][:10]
```

```
[{'w': 'name', 'i': 1206},
 {'w': 'older', 'i': 4370},
 {'w': 'famous', 'i': 4423},
 {'w': 'ready', 'i': 4533},
 {'w': 'special', 'i': 5645},
 {'w': 'famous', 'i': 5651},
 {'w': 'asleep', 'i': 5935},
 {'w': 'fast', 'i': 6318},
 {'w': 'small', 'i': 6338},
 {'w': 'skinny', 'i': 6340}]
```

```
book_data["characters"][0]["poss"][:10]
```

```
[{'w': 'aunt', 'i': 4356},
 {'w': 'uncle', 'i': 4358},
 {'w': 'name', 'i': 4461},
 {'w': 'blankets', 'i': 5622},
 {'w': 'cousin', 'i': 5698},
 {'w': 'Petunia', 'i': 5947},
 {'w': 'aunt', 'i': 5981},
 {'w': 'back', 'i': 6020},
 {'w': 'aunt', 'i': 6062},
 {'w': 'aunt', 'i': 6133}]
```

```
book_data["characters"][0]["id"]
```

```
98
```

For the g category, we see a few different keys:

- inference – the pronouns for the entity in order of highest frequency to lowest.
- argmax – the likely pronoun/gender for the entity.
- max – the degree to which that pronoun set is used compared to others, e.g. the percentage.

```
book_data["characters"][0]["g"]
```

```
{'inference': {'he/him/his': 0.811,
 'she/her': 0.112,
 'they/them/their': 0.077,
 'xe/xem/xyr/xir': 0.0,
 'ze/zem/zir/hir': 0.0},
 'argmax': 'he/him/his',
 'max': 0.811,
 'total': 200311.834}
```

```
book_data["characters"][0]["count"]
```

```
2005
```

For mentions, we have three special keys:

- proper – the way they are referenced as proper nouns.
- common – informal names.
- pronoun – the pronouns used to refer to them in prose and dialogue.

```
book_data["characters"][0]["mentions"].keys()
```

```
dict_keys(['proper', 'common', 'pronoun'])
```

```
book_data["characters"][0]["mentions"]["proper"]
```

```
[{"c": 664, "n": "Harry"},  
 {"c": 46, "n": "Potter"},  
 {"c": 23, "n": "Harry Potter"},  
 {"c": 11, "n": "Mr. Potter"},  
 {"c": 2, "n": "Mr. Harry Potter"},  
 {"c": 1, "n": "Harry Hunting"},  
 {"c": 1, "n": "Cokeworth Harry"},  
 {"c": 1, "n": "Both Harry"},  
 {"c": 1, "n": "The Harry Potter"},  
 {"c": 1, "n": "HARRY POTTER"},  
 {"c": 1, "n": "Even Harry"},  
 {"c": 1, "n": "POTTER"},  
 {"c": 1, "n": "the famous Harry Potter"}]
```

```
book_data["characters"][0]["mentions"]["common"]
```

```
[]
```

```
book_data["characters"][0]["mentions"]["pronoun"]
```

```
[{"c": 303, "n": "he"},  
 {"c": 217, "n": "his"},  
 {"c": 172, "n": "you"},  
 {"c": 144, "n": "He"},  
 {"c": 107, "n": "him"},  
 {"c": 99, "n": "I"},  
 {"c": 34, "n": "me"},  
 {"c": 30, "n": "your"},  
 {"c": 27, "n": "yeh"},  
 {"c": 27, "n": "You"},  
 {"c": 18, "n": "yer"},  
 {"c": 16, "n": "himself"},  
 {"c": 14, "n": "my"},  
 {"c": 12, "n": "His"},  
 {"c": 5, "n": "Your"},  
 {"c": 3, "n": "Yeh"},  
 {"c": 3, "n": "Yer"},  
 {"c": 3, "n": "My"},  
 {"c": 2, "n": "yeh've"},  
 {"c": 2, "n": "yeh'd"},  
 {"c": 2, "n": "ter"},  
 {"c": 2, "n": "myself"},  
 {"c": 2, "n": "yourself"},  
 {"c": 1, "n": "YOU"},  
 {"c": 1, "n": "mine"},  
 {"c": 1, "n": "yours"},  
 {"c": 1, "n": "Yeh'd"},  
 {"c": 1, "n": "yerself"},  
 {"c": 1, "n": "Yeh've"},  
 {"c": 1, "n": "yeh'll"}]
```

15.3.6 The .book.html File

The final file that is outputted from BookNLP is the .book.html file. This is a nicely organized, easy-to-read, HTML file that should open in your browser.

15.4 Character Analysis

This section is dedicated to analyzing the characters contained within the .book file. As you may recall from the last section, this is a JSON file. A lot of what I will cover here, can be found in the BookNLP repository, specifically in the Google Colab Jupyter Notebook. I am, however, making some modifications to the code there to make it a bit more useful for varying circumstances. I will specifically show you how to use this restructured data to pose narrow questions about characters in a text.

The following functions and imports will be necessary for this chapter. They allow us to load up the JSON data from the .book file and count the occurrences of certain things found within the .book file.

```
import json
from collections import Counter

def proc(filename):
    with open(filename) as file:
        data=json.load(file)
    return data

def get_counter_from_dependency_list(dep_list):
    counter=Counter()
    for token in dep_list:
        term=token["w"]
        tokenGlobalIndex=token["i"]
        counter[term]+=1
    return counter
```

Now that we have successfully created these functions, let's go ahead and load up our JSON data from the .book file. We can do this with the function above that we created called "proc". Essentially, this loads and parses the JSON file for us using the JSON library that comes standard with Python.

```
data=proc("data/harry_potter/harry_potter.book")
```

Now that we have loaded the data, we can start to analyze it!

15.4.1 Analyzing the Characters (From BookNLP Repo)

If you have had a chance to look at the Google Colab notebook provided by BookNLP, this function will look similar. I have made some modifications to the code presented there so that we can do a bit more with it. In the notebook, the original code printed off character data. My modifications and the fact that I have structured it as a function, allow us to do a bit more. We can actually begin analyzing the characters.

I have kept my function's code as close to the original as possible so that it can be better understood within the documentation.

```
def create_character_data(data, printTop):
    character_data = {}
    for character in data["characters"]:

        agentList=character["agent"]
```

(continues on next page)

(continued from previous page)

```

patientList=character["patient"]
possList=character["poss"]
modList=character["mod"]

character_id=character["id"]
count=character["count"]

referential_gender_distribution=referential_gender_prediction="unknown"

if character["g"] is not None and character["g"] != "unknown":
    referential_gender_distribution=character["g"]["inference"]
    referential_gender=character["g"]["argmax"]

mentions=character["mentions"]
proper_mentions=mentions["proper"]
max_proper_mention=""

#Let's create some empty lists that we can append to.
poss_items = []
agent_items = []
patient_items = []
mod_items = []

# just print out information about named characters
if len(mentions["proper"]) > 0:
    max_proper_mention=mentions["proper"][0]["n"]
    for k, v in get_counter_from_dependency_list(possList).most_
→common(printTop):
        poss_items.append((v,k))

    for k, v in get_counter_from_dependency_list(agentList).most_
→common(printTop):
        agent_items.append((v,k))

    for k, v in get_counter_from_dependency_list(patientList).most_
→common(printTop):
        patient_items.append((v,k))

    for k, v in get_counter_from_dependency_list(modList).most_
→common(printTop):
        mod_items.append((v,k))

# print(character_id, count, max_proper_mention, referential_gender)
character_data[character_id] = {"id": character_id,
                                "count": count,
                                "max_proper_mention": max_proper_mention,
                                "referential_gender": referential_gender,
                                "possList": poss_items,
                                "agentList": agent_items,
                                "patientList": patient_items,
                                "modList": mod_items
                                }

return character_data

```

This function expects two arguments:

- The data that we created above, i.e. the original .book JSON data.

- `printTop` which will be the number of items you seek to return about the character.

Let's go ahead and create some `character_data` now that will retain the top 10 items connected to each character. If you want to see all possible things connected to the character, simply set this item to a very high number, e.g. 20,000. This is not the cleanest, but it allowed me to keep this function as simple as possible.

This function will return a new data file that will be a dictionary where each unique id is a key and the corresponding character data will be populated as its value (also structured as a dictionary). I have kept the keys of this nested dictionary identical to the original Google Colab file.

```
character_data = create_character_data(data, 10)
```

Now that we have created this character data, let's take a look at the main Harry Potter id (which is 98).

```
print (character_data[98])
```

```
{'id': 98, 'count': 2029, 'max_proper_mention': 'Harry', 'referential_gender': 'he/him/his', 'possList': [(19, 'head'), (15, 'eyes'), (12, 'parents'), (12, 'cupboard'), (10, 'life'), (10, 'hand'), (9, 'aunt'), (8, 'mind'), (7, 'heart'), (7, 'uncle')], 'agentList': [(91, 'said'), (46, 'had'), (39, 'know'), (22, 'felt'), (22, 'saw'), (21, 'got'), (21, 'going'), (21, 'thought'), (18, 'heard'), (18, 'looked')], 'patientList': [(10, 'told'), (5, 'take'), (5, 'asked'), (4, 'kill'), (4, 'reminded'), (4, 'stop'), (4, 'got'), (4, 'tell'), (3, 'took'), (3, 'saw')], 'modList': [(8, 'sure'), (5, 'able'), (4, 'famous'), (3, 'glad'), (2, 'name'), (2, 'special'), (2, 'surprised'), (2, 'baby'), (2, 'stupid'), (2, 'afraid')]}
```

Notice that we can now see the main gender, verbs, possession items, etc. connected to Harry Potter. Having the data structured in this manner allows us to more easily start posing some questions to the original `.book` file.

15.4.2 Parsing Verb Usage

One of those questions can be about verb usage. I have created a brand new function that allows you to explore how certain verbs are used within the text based on the new character data file we just created. It expects one argument: the new character data file. We can pass an additional keyword argument that should be a list. This list will contain one or two of the following items:

- agent – the doer of the action.
- patient – the recipient of the action.

Again, this function is not something I would put in production. I have designed it to be easier to read so that you can do something similar and grab data you may find relevant for your own project or research.

```
def find_verb_usage(data, analysis=["agent", "patient"]):
    new_analysis = []
    for item in analysis:
        if item == "agent":
            new_analysis.append("agentList")
        elif item == "patient":
```

(continues on next page)

(continued from previous page)

```

        new_analysis.append("patientList")
main_agents = {}
main_patients = {}
for character in character_data:
    temp_data = character_data[character]
    for item in new_analysis:
        for verb in temp_data[item]:
            verb = verb[1].lower()
            if item == "agentList":
                if verb not in main_agents:
                    main_agents[verb] = [(character, temp_data["max_proper_
→mention"])]
            else:
                main_agents[verb].append((character, temp_data["max_"
→proper_mention"]))
            elif item == "patientList":
                if verb not in main_patients:
                    main_patients[verb] = [(character, temp_data["max_proper_
→mention"])]
                else:
                    main_patients[verb].append((character, temp_data["max_"
→proper_mention"]))
verb_usage = {"agent": main_agents,
              "patient": main_patients}
return verb_usage

```

Essentially, this function will read in the character data file that we created above and create a new dictionary that has two keys: agent and patient. Within each will be the verbs used in the text. These will be matched to a list of the characters connected to those verbs. Let's go ahead and create this verb data now.

```
verb_data = find_verb_usage(data)
```

By restructuring the data around the verbs, you can analyze the characters in a verb-centric manner. Let's say I was interested in what characters were the agents of the verb "reared". I could go into the dictionary at the agent key and look for the key of reared within the agent verbs. My output is the tuple of (character_id, most frequent name for that character). In this case: Firenze the centaur.

```
verb_data["agent"]["reared"]
```

```
[(352, 'Firenze')]
```

It is important to note two things here, however. First, our verbs are not lemmatized. I intentionally left this as the case because in some circumstances understanding how a verb is used is important. You may, for example, be interested in how "said" functioned in the next, not both "said" and "say". If you wanted to modify the code above, therefore, you could go into the tokens file to find that verb's lemma.

Another thing to note is that we are only seeing the results from the top 10 in this scenario. If you want to see how verbs area used by all characters, create a new character data file and make your top-n equal to a larger number.

15.5 Events Analysis

The only output file that details event data is the `.tokens` file. As a result, this file will be the focus of this section. Each subsection of this section will analyze the `.tokens` file in a deeper way to identify and extract event data. At the end of the chapter, we will bring everything together with a single function that can recreate these results on any BookNLP output `.tokens` file.

15.5.1 Exploring the Tokens File

Let's first go ahead and open up the `.tokens` file and take a look at it so we can remember precisely what the `.tsv` file looks like. If you remember from Part III, we can analyze the files a bit more easily if we use Pandas.

```
import pandas as pd
df = pd.read_csv("data/harry_potter/harry_potter.tokens", delimiter="\t")
df
```

	paragraph_ID	sentence_ID	token_ID_within_sentence	\		
0	0	0		0		
1	0	0		1		
2	0	0		2		
3	0	0		3		
4	0	0		4		
...		
99251	2995	6172		10		
99252	2995	6172		11		
99253	2995	6172		12		
99254	2995	6172		13		
99255	2995	6172		14		
	token_ID_within_document	word	lemma	byte_onset	byte_offset	\
0	0	Mr.	Mr.	0	3	
1	1	and	and	4	7	
2	2	Mrs.	Mrs.	8	12	
3	3	Dursley	Dursley	13	20	
4	4	,	,	20	21	
...
99251	99251	Dudley	Dudley	438929	438935	
99252	99252	this	this	438936	438940	
99253	99253	summer	summer	438941	438947	
99254	99254	438947	438951	
99255	99255	\t	438951	438952	PUNCT	
	POS_tag	fine_POS_tag	dependency_relation	syntactic_head_ID	event	
0	PROPN	NNP	nmod	3	0	
1	CCONJ	CC	cc	0	0	
2	PROPN	NNP	compound	3	0	
3	PROPN	NNP	nsubj	12	0	
4	PUNCT	,	punct	3	0	
...
99251	PROPN	NNP	pobj	99250	0	
99252	DET	DT	det	99253	0	
99253	NOUN	NN	npadvmod	99245	0	
99254	PUNCT	.	punct	99243	0	
99255	''	punct	99243	0	NaN	
[99256 rows x 13 columns]						

We have approximately 99,000 rows and 13 columns of data. Throughout this section, we will focus on only four columns in particular:

- sentence_ID
- word
- lemma
- event

As such, let's go ahead and remove all the extra data for now so that we can just view the columns we care about.

```
df = df[["sentence_ID", "word", "lemma", "event"]]
```

Excellent! Now we can analyze this event column a bit more easily.

15.5.2 Grabbing the Events

One of the things we can see above is that some event columns contain NaN. Ideally, we want to ignore these entirely. We can do this in pandas by using the `isnull()` method.

```
events = df[~df['event'].isnull()]
events
```

	sentence_ID	word	lemma	event
0	0	Mr.	Mr.	O
1	0	and	and	O
2	0	Mrs.	Mrs.	O
3	0	Dursley	Dursley	O
4	0	,	,	O
...
99250	6172	with	with	O
99251	6172	Dudley	Dudley	O
99252	6172	this	this	O
99253	6172	summer	summer	O
99254	6172	O

[94498 rows x 4 columns]

As we can see this eliminated roughly 5,000 rows. Let's take a closer look at the column event and see what kind of data we can expect to see here.

```
event_options = set(events.event.tolist())
print(event_options)
```

```
{'EVENT', 'O'}
```

By converting this column to a list and then to a set (which eliminates the duplicates), we can see that we have two types of data in the event column:

- EVENT
- O

If a row has “EVENT” in the column then it means the corresponding word was identified by the BookNLP pipeline as being an event-triggering word. Now that we know this, let's take a look at only the rows that have EVENT in the event column.

```
real_events = events.loc[df["event"] == "EVENT"]
real_events
```

	sentence_ID	word	lemma	event
242	9	shuddered	shudder	EVENT
308	12	woke	wake	EVENT
346	13	hummed	hum	EVENT
349	13	picked	pick	EVENT
361	13	gossiped	gossip	EVENT
...
99152	6167	hung	hang	EVENT
99185	6169	said	say	EVENT
99209	6170	said	say	EVENT
99215	6170	surprised	surprised	EVENT
99218	6170	grin	grin	EVENT

[6029 rows x 4 columns]

We now have only 6,029 rows to analyze!

15.5.3 Analyzing Events Words and Lemmas

Let's dig a little deeper. Let's try to analyze the words and lemmas of these rows to see how many unique words and lemmas we have.

```
event_words = set(real_events.word.tolist())
len(event_words)
```

1501

```
event_lemmas = list(set(real_events.lemma.tolist()))
event_lemmas.sort()
len(event_lemmas)
```

1021

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

```
['BOOM', 'Bludger', 'Pompously', 'Scowling', 'Smelting', 'Whispers', 'aback',
 'accept', 'ache', 'act']
```

While we have 1,501 unique words, we only have 1,021 unique lemmas. If we were interested in seeing the type of event words and lemmas appear in Harry Potter, we can now do that, but something I notice quickly is that some lemmas are capitalized. Let's eliminate all duplicates by lowering all lemmas.

```
final_lemmas = []
for lemma in event_lemmas:
    lemma = lemma.lower()
    if lemma not in final_lemmas:
        final_lemmas.append(lemma)

print(len(final_lemmas))
print(final_lemmas[:10])
```

```
1020
['boom', 'bludger', 'pompously', 'scowling', 'smelting', 'whispers', 'aback',
→'accept', 'ache', 'act']
```

We eliminated only one duplicate.

15.5.4 Grabbing Event Sentences

Now that we know how to grab individual event-triggering words, what about the sentences that contain events? To analyze this, we can use the sentence_ID column which contains a unique number for each sentence.

```
sentences = real_events.sentence_ID.tolist()
events = real_events.word.tolist()
print (sentences[:10])
print (events[:10])
```

```
[9, 12, 13, 13, 13, 13, 13, 14, 15, 15]
['shuddered', 'woke', 'hummed', 'picked', 'gossiped', 'wrestled', 'screaming',-
→'flutter', 'picked', 'pecked']
```

We can see that some sentences appear multiple times. This is because they contain multiple words that are event-triggering.

Let's take a look at our initial DataFrame once again.

```
df
```

	sentence_ID	word	lemma	event
0	0	Mr.	Mr.	O
1	0	and	and	O
2	0	Mrs.	Mrs.	O
3	0	Dursley	Dursley	O
4	0	,	,	O
...
99251	6172	Dudley	Dudley	O
99252	6172	this	this	O
99253	6172	summer	summer	O
99254	6172	O
99255	6172	\t	438951	NaN

[99256 rows x 4 columns]

Let's say we were interested in grabbing the first sentence from the first event, we can grab all rows that have a matching sentence_ID.

```
sentence1 = sentences[0]
result = df[df["sentence_ID"] == int(sentence1)]
result
```

	sentence_ID	word	lemma	event
240	9	The	the	O
241	9	Dursleys	Dursleys	O
242	9	shuddered	shudder	EVENT
243	9	to	to	O
244	9	think	think	O

(continues on next page)

(continued from previous page)

```

245      9      what      what      O
246      9      the       the       O
247      9      neighbors neighbor O
248      9      would    would    O
249      9      say       say       O
250      9      if        if        O
251      9      the       the       O
252      9      Potters   Potters   O
253      9      arrived   arrive   O
254      9      in        in        O
255      9      the       the       O
256      9      street    street    O
257      9      .         .         O

```

With this data, we can then grab all the words and reconstruct the sentence.

```

words = result.word.tolist()
resentence = " ".join(words)

```

```
print (resentence)
```

```

The Dursleys shuddered to think what the neighbors would say if the Potters_
→arrived in the street .

```

15.5.5 Bringing Everything Together

Let's now bring everything together we just learned in this chapter and make it into a function. This function will receive a file that corresponds to the `.tokens` file. It will find the relevant event rows and then reconstruct the sentences that correspond to each event word. The output will be a list of dictionaries that are event-centric. Each dictionary will have three keys:

- `event_word` – the event-triggering word.
- `event_lemma` – the `event_word`'s lemma.
- `sentence` – the sentence that the `event_word` is in.

```

def grab_event_sentences(file):
    df = pd.read_csv(file, delimiter="\t")
    real_events = df.loc[df["event"] == "EVENT"]
    sentences = real_events.sentence_ID.tolist()
    event_words = real_events.word.tolist()
    event_lemmas = real_events.lemma.tolist()
    final_sentences = []
    x=0
    for sentence in sentences:
        result = df[df["sentence_ID"] == int(sentence)]
        words = result.word.tolist()
        resentence = " ".join(words)
        final_sentences.append({"event_word": event_words[x],
                               "event_lemma": event_lemmas[x],
                               "sentence": resentence})
        x=x+1

```

(continues on next page)

(continued from previous page)

```

        })
x=x+1
return final_sentences

event_data = grab_event_sentences("data/harry_potter/harry_potter.tokens")

```

Let's take a look at the output now.

```

print (event_data[0])

{'event_word': 'shuddered', 'event_lemma': 'shudder', 'sentence': 'The_
→Dursleys shuddered to think what the neighbors would say if the Potters_
→arrived in the street .'}

```

15.5.6 Creating an `.events` File

This allows us to now analyze the events identified in the BookNLP pipeline a bit more easily. Since we don't have an `.events` output file, this is currently one way that we can simulate the same result by creating a special events-centric output. With this data, we can now create a new DataFrame.

```

new_df = pd.DataFrame(event_data)

new_df

   event_word  event_lemma           sentence
0    shuddered     shudder  The Dursleys shuddered to think what the neigh...
1       woke       wake  When Mr. and Mrs. Dursley woke up on the dull ...
2      hummed      hum  Mr. Dursley hummed as he picked out his most b...
3      picked      pick  Mr. Dursley hummed as he picked out his most b...
4   gossiped     gossip  Mr. Dursley hummed as he picked out his most b...
...       ...       ...
6024     hung      hang  Harry hung back for a last word with Ron and H...
6025     said      say  \t \t Hope you have -- er -- a good holiday , ...
6026     said      say  \t Oh , I will , \t said Harry , and they were...
6027  surprised  surprised  \t Oh , I will , \t said Harry , and they were...
6028     grin      grin  \t Oh , I will , \t said Harry , and they were...
[6029 rows x 3 columns]

```

We can also output it to the same subdirectory as the other files.

```
new_df.to_csv("data/harry_potter/harry_potter.events", index=False)
```

And now you have an `.events` file!

15.5.7 Conclusion

You should now have a basic understanding of BookNLP and what it can do. While the results will not be perfect, it will give you a great starting point for understanding the salient characters, extracting, quotes, and identifying the major events within a large work of fiction.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

16

Social Network Analysis

In this chapter, we will shift away from NLP and examine **social network analysis** (SNA). SNA is the process by which researchers study the networks of social relationships. These can be familial (parents to children), social (friends), etc. The goal of SNA is to reveal insights about a network or group of individuals by studying them collectively and often through visualization methods.

16.1 The Basic Concepts of Social Network Analysis

16.1.1 Basic Terminology

Visualized social networks are often studied through **graphs**, or visual representations of relationships in a network. These graphs today stem often from **graph theory**, a branch of mathematics. In graph theory, mathematics is used to study graph-based problems. Believe it or not, we are all beneficiaries of this discipline. Have you ever used Google Maps to go from point A to point B? This is graph theory at play. Behind the scenes is a complex set of relationships that allow Google to recommend certain paths over others to ensure that you have the fastest (or least expensive) route.

This chapter will not cover all the complexities of SNA, rather give enough of the basic terminology and concepts so that all readers can learn how to leverage Python to perform SNA. The chief goal of this chapter is to introduce the process by which we can perform SNA through structured data.

In a graph, there are a collection of **nodes**. These are dots in the graph that represent each piece of data. In a social network graph, each node would be a person or some kind of entity that a researcher wishes to map. Other entities may include things like businesses or agencies.

In order to understand how different nodes relate to one another, we represent the relationships between them with **edges**. In a graph, these edges look like lines.

Because graphs are often drawn with mathematics, it is important to know the **force** of a graph. Force in network theory is the direction of movement between two nodes. If we are mapping how characters move to different places in a graph, all people and places would receive a node in a graph. We would then inflict force with the person doing the movement towards the place. This would position the nodes in a particular way in a graph.

In a network graph, we can often map multi-modal networks, or networks where different types of relationships are overlapping. Often, we can do this in a graph by representing each type of relationship as a separate edge color.

Graphs are a useful way to explore complex relationships because in a single image, we can glean information that would otherwise be missed. We can examine our data quantitatively, meaning we can see the frequency that certain nodes appear in our data and,

more importantly, the frequency with which that node relates to other nodes in the graph. We could, of course, extract this information for a node relatively easily with Pandas, but a graph lets us see many different relationships between many different pieces of data all in a single image. For these reasons, it is often useful to be familiar with SNA generally and generally how to map nodal relationships in Python.

16.1.2 SNA Libraries in Python

Python has several libraries for performing SNA. In this section, we will look closely at two: NetworkX (and Matplotlib to visualize the graph) and PyVis. Each has its strengths and weaknesses.

NetworkX is a powerful library that allows users to hold complex graph-based data, such as nodes and edges, in a single class. It also allows us to perform basic mathematical calculations to discover things like centrality of a node in a graph, a concept we will learn about in the next section. NetworkX is designed to work alongside Matplotlib for plotting those graphs. As we shall see, there are some limitations to this approach. PyVis is another graph visualization library and while you can create graphs entirely independently of NetworkX, some workflows may benefit from creating the graph data in NetworkX and then passing that data to PyVis for visualization. You can download all of these libraries with pip

To install Matplotlib, use the following command in your terminal:

```
pip install matplotlib
```

For NetworkX, you can use this command:

```
pip install networkx
```

And for PyVis, you can use this command:

```
pip install pyvis
```

Note that each of these libraries are all in lowercase when we install them via pip.

16.2 Introduction to NetworkX

Now that we have installed our libraries correctly, we can begin working with NetworkX and Matplotlib. To import each library, we will use the code below. You will often see Matplotlib imported one of two ways:

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

or:

```
from matplotlib import pyplot as plt
```

Both import pyplot as plt and, therefore, are identical. The usage comes down to preference.

```
from matplotlib import pyplot as plt
import networkx as nx
```

Before we begin, we must first have a set of relationships that we wish to graph. Let's use a simple toy example, such as the five relationships below. Each person in these relationships will function as a single node in our graph. We will want to map these relationships by drawing edges between each individual's node.

```
rels = [
    ["Fred", "George"],
    ["Harry", "Rita"],
    ["Fred", "Ginny"],
    ["Tom", "Ginny"],
    ["Harry", "Ginny"]
]
```

In order to begin working with this data in NetworkX, we need to first populate a NetworkX Graph class. It is Pythonic to call this variable G.

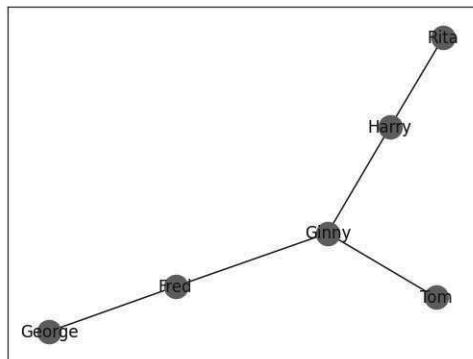
```
G = nx.Graph()
```

Once created, we can populate the graph with relationships in a loop. We will want to use the `.add_edge()` method from the Graph class. This will take two mandatory arguments: 1. the source node and 2. the destination node. The source and the destination are important in certain circumstances when force is applied in a graph. This controls how the map is displayed visually.

```
for rel in rels:
    source, dest = rel
    G.add_edge(source, dest)
```

With all our edges add into the graph, we can now plot it with Matplotlib. We will use the `draw_networkx()` function to create a plot in memory. This will take one argument, the graph of nodes we wish to visualize. Next, we will use `plt.show()` to display the map. Since we are working within a Jupyter Notebook, this will be displayed as an output.

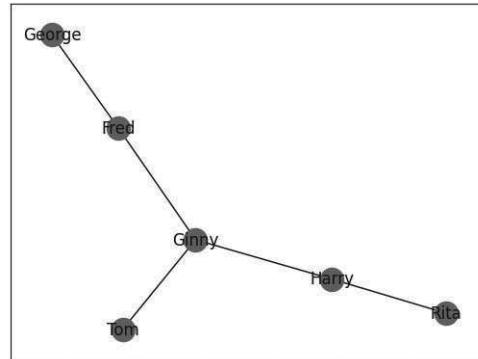
```
nx.draw_networkx(G)
plt.show()
```



16.2.1 Adding All Edges at Once

Sometimes, you will not want to add each edge individually. In these circumstances, we can leverage the `add_edges_from()` method to add a list of edges all at once. This method will expect a list of relationships.

```
G = nx.Graph()
G.add_edges_from(rels)
nx.draw_networkx(G)
plt.show()
```

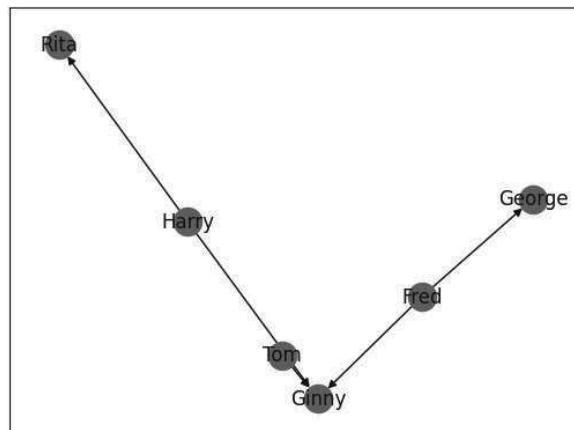


16.2.2 Asymmetrical Networks

The above examples are all cases of symmetrical networks. This is where each node has an equal attraction to the other. The best example of this is with colleagues. If Ginny is a colleague with Harry, then Harry is a colleague with Ginny. Not all relationships work in this manner. Some relationships are asymmetrical. This is where one person has a unique relationship with another that cannot be the same in reverse. A good example of this is with parents and children. If Fred is the father of George, then George cannot possibly be the father of Fred.

NetworkX affords us the ability to draw these types of relationships with a different graph class called a `DiGraph`. We can use the precise same code as above, but rather than using `.nx.Graph()`, we will use `nx.DiGraph()`.

```
G = nx.DiGraph()
G.add_edges_from(rels)
nx.draw_networkx(G)
plt.show()
```



Note the salient change in our graph, specifically the arrows that point towards specific nodes. These arrows in a graph indicate the direction, or **force**, between two nodes in a

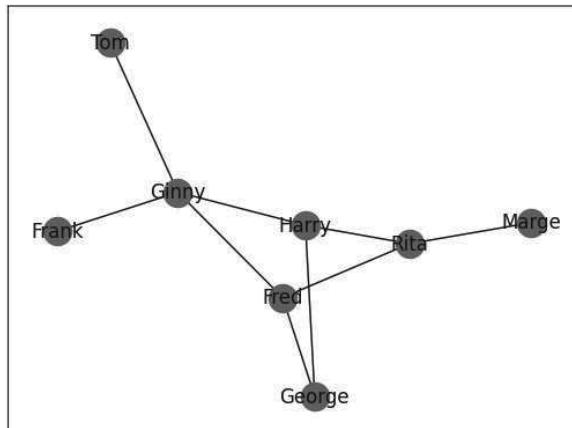
graph. In the graph above, for example, we can see that Harry has a different relationship to Rita than Rita does with Harry.

16.2.3 Calculating Shortest Distance

The above example is rather simple. Let's create a more complex graph with a few more relationships.

```
rels = [
    ["Fred", "George"],
    ["Harry", "Rita"],
    ["Fred", "Ginny"],
    ["Tom", "Ginny"],
    ["Harry", "Ginny"],
    ["Harry", "George"],
    ["Frank", "Ginny"],
    ["Marge", "Rita"],
    ["Fred", "Rita"]
]
```

```
G = nx.Graph()
G.add_edges_from(rels)
nx.draw_networkx(G)
plt.show()
```



A common problem in graph theory is calculating the shortest distance to individuals. A good way to think about this is with the classic game Seven Degrees of Kevin Bacon. If we wanted to know the shortest distance between Fred and Rita, we could realistically look at the graph above and figure that out. But what if we had thousands of nodes? What about millions? These are real-world problems that cannot be solved practically by simple human observation. NetworkX preserves the network as a graph that it can walk to easily identify the shortest path between two nodes at any given point. To calculate this, we can use `nx.shortest_path()`. This will take three arguments, the graph in which the nodes are found and the source node and target node. The result will be a list of the source node, followed by all **brokers** or intermediaries between the source and target node. These brokers will be followed by the target node.

```
nx.shortest_path(G, 'Fred', 'Marge')
```

```
['Fred', 'Rita', 'Marge']
```

16.2.4 Calculating Connections

Another common task in graph analysis is understanding the `degree`, or number of connections, a node has at any given moment. This allows us to understand the role an individual plays within a network. Nodes with higher degree values often indicate a higher degree of importance in the data, though this is not always the case. In humanities-based inquiry, we frequently work with imperfect or incomplete data due to losses in the historical record. It is important, therefore, to understand a degree value as a reflection of extant data, not a reflection of certain importance.

We can calculate a node's degree value with `nx.degree()` which will take two arguments, the graph in which the node is found and the node for whom we seek a degree value.

```
nx.degree(G, "Fred")
```

```
3
```

16.2.5 Identifying Major Actors in a Network

Another way to understand a specific node's role in a network, is to calculate a node's `centrality`, or degree of focus, in a network. We can calculate and grab the centrality of all nodes in our graph with `nx.degree_centrality()`. This will take a single argument, our graph. It will return a dictionary whose keys are each node and whose values are each node's centrality score. The higher the number, the more significant role that particular node plays in the graph.

```
centrality = nx.degree_centrality(G)
centrality
```

```
{'Fred': 0.42857142857142855,
'George': 0.2857142857142857,
'Harry': 0.42857142857142855,
'Rita': 0.42857142857142855,
'Ginny': 0.5714285714285714,
'Tom': 0.14285714285714285,
'Frank': 0.14285714285714285,
'Marge': 0.14285714285714285}
```

16.2.6 Limitations of Matplotlib and NetworkX

While NetworkX is quite useful for creating the data necessary to draw quick and (with practice) quality graphs via Matplotlib, this is largely suited to smaller networks. Often, when we are working with larger datasets of networks it will be important to visualize those networks more dynamically, where users can zoom in, zoom out, and select certain nodes. Most of these types of graphs are designed and implemented in JavaScript.

The graphs designed in JavaScript have the advantage of being able to be opened in any browser and distributed across the web or embedded in any website. Fortunately, the Python library PyVis allows us to produce JavaScript-based graphs with Python code.

16.3 Producing Dynamic Graphs with PyVis

PyVis¹ is a powerful Python library that is capable of outputting an HTML file which contains the data and JavaScript necessary for users to view and engage with network data. It has a fairly large community and is well-supported. One of its largest advantages over NetworkX and Matplotlib is in the dynamic nature of the graphs.

Dynamic graphs, or those that can be manipulated by a user in some capacity, are better suited than static graphs for networks that are large or more complex.

There are other libraries for doing similar steps as laid out here. Bokeh is a suitable alternative. The main issue with Bokeh is that it has a steep learning curve and, while it certainly offers greater flexibility and customization options than PyVis, it can be challenging for those newer to Python. As we will see, you can produce quality, dynamic network graphs with PyVis in just a few lines of Python code. For this reason, I am opting to focus on PyVis in this section.

You can install PyVis via pip by running in the command line:

```
pip install pyvis
```

This should install PyVis on your system.

16.3.1 The Basics of PyVis

Once you have installed PyVis, you will want to import the Network class. We can do so with the following line.

```
from pyvis.network import Network
```

The Network class holds all of the data that will be used to create the graph. Since we are working within a Jupyter Notebook, we will want to set the keyword argument notebook to True. This will ensure that when we create our network graph, it will not only output as an HTML file, but also load within the notebook.

We can create a Network object and load it into memory with the following command:

```
net = Network(notebook=True)
```

Local cdn resources have problems on chrome/safari when used in jupyter-notebook.

Again, we will work with the same toy data here.

```
rels = [
    ["Fred", "George"],
    ["Harry", "Rita"],
```

(continues on next page)

¹<https://pyvis.readthedocs.io/en/latest/index.html>

(continued from previous page)

```
[ "Fred", "Ginny"],
[ "Tom", "Ginny"],
[ "Harry", "Ginny"]

]
```

In PyVis, we want to create each node in the graph individually and then populate the edge between the two nodes in the relationship. Therefore, we will want to iterate over each relationship with a `for` loop. We will use the `.add_node` method for each node in the graph and the `.add_edge()` method to create an edge between the two nodes.

```
for rel in rels:
    source, dest = rel
    net.add_node(source)
    net.add_node(dest)
    net.add_edge(source, dest)
```

Once we have populated the `Network` class with all our data, we can visualize it with the method `.show()`. This will take a single argument, a string that will be the file created.

```
net.save_graph("simple_graph.html")
```

```
from IPython.display import HTML
HTML(filename="simple_graph.html")
```

```
<IPython.core.display.HTML object>
```

In the graph above, try to interact with the data. You will notice that you can drag-and-drop nodes. You can zoom in and out in the graph. This is a good demonstration of the power PyVis has over NetworkX and Matplotlib.

16.4 NetworkX and PyVis

While this works, we don't have all the advantages that NetworkX afforded us. PyVis, it is important to note, while useful, is primarily a wrapper for producing JavaScript graphs. It is not as robust as NetworkX when it comes to handling and querying complex network relationships. We cannot generate, for example, metrics for things like centrality. For these reasons, it is best to combine the best of NetworkX and PyVis by first passing your network data to NetworkX, creating the graph, and then passing the NetworkX graph to PyVis for conversion to an interactive graph.

```
import networkx as nx
```

Just as we did in the previous section, we will create a NetworkX `nx.Graph()` and populate it with our relationships.

```
G = nx.Graph()
rels = [
    [ "Fred", "George"],
```

(continues on next page)

(continued from previous page)

```
[ "Harry", "Rita"],  
[ "Fred", "Ginny"],  
[ "Tom", "Ginny"],  
[ "Harry", "Ginny"],  
[ "Harry", "George"],  
[ "Frank", "Ginny"],  
[ "Marge", "Rita"],  
[ "Fred", "Rita"]  
  
]  
G.add_edges_from(rels)
```

Now that we have our NetworkX Graph, we can create a PyVis Network class.

```
net = Network()
```

Using the `from_nx()` method, which will take a single argument, our NetworkX Graph, or G.

```
net.from_nx(G)
```

With the data now populated, we can save and view our PyVis Network.

```
net.save_graph("networkx-pyvis.html")
```

```
HTML(filename="networkx-pyvis.html")
```

```
<IPython.core.display.HTML object>
```

Notice that our graph is rendered precisely just as it had been with NetworkX and Matplotlib, but now that same data is dynamic. In other words, we were able to leverage the best of NetworkX and PyVis with this approach.

16.5 Adding Color to Nodes

Graphs where every node is the same color can be difficult to interpret. For this reason, we may want to represent each node as a distinct color given some trait of that node. In the example below, we will use a simple list of Harry Potter characters. We will use their houses to denote node color. Let's import the same libraries we have worked with so far in this chapter and throughout this textbook.

```
import networkx as nx  
from pyvis.network import Network  
import pandas as pd  
from IPython.display import HTML
```

Our data is stored as a csv file which we can load with Pandas.

```
node_df = pd.read_csv("../data/hp - nodes.csv")  
node_df
```

	name	house	color
0	Harry	Gryffindor	red
1	Hermione	Gryffindor	red
2	Ron	Gryffindor	red
3	Drako	Slytherine	green
4	Snape	Slytherine	green
5	Sedric	Hufflepuff	yellow
6	Luna	Ravenclaw	blue

Now that we have our data loaded, we can iterate over each Pandas row and populate a list of nodes. Notice, that when we append to `nodes`, we are not appending a string, rather a tuple. Index 0 of the tuple is the character name, while index 1 is a dictionary. This is where we can store special attributes associated with the node. Think of this as metadata. In our case, we want two pieces of metadata, `color` and `house`.

```
nodes = []
for idx, row in node_df.iterrows():
    nodes.append((row["name"], {"color": row.color, "house": row.house}))
nodes[:1]
```

```
[('Harry', {'color': 'red', 'house': 'Gryffindor'})]
```

Now that we have our node list created, let's populate it into a NetworkX Graph.

```
G = nx.Graph()
G.add_nodes_from(nodes)
```

From here, we can do the same thing with our edge list, which is also stored in a Pandas dataframe. NetworkX has built-in ways of creating a graph from a Pandas edge list, but we will do this manually here.

```
rel_df = pd.read_csv("../data/hp - rels.csv")
rel_df
```

	source	target
0	Harry	Hermione
1	Hermione	Ron
2	Ron	Harry
3	Drako	Harry
4	Snape	Sedric
5	Sedric	Harry
6	Luna	Harry
7	Snape	Harry
8	Luna	Hermione
9	Ron	Drako

```
edge_list = []
for idx, row in rel_df.iterrows():
    edge_list.append((row.source, row.target))
edge_list[:1]
```

```
[('Harry', 'Hermione')]
```

Now that we have our edge list, we can inject it into the same Graph object.

```
G.add_edges_from(edge_list)
```

At this stage, our NetworkX Graph is complete with all the data. We can now repeat the steps from the previous section and create our PyVis visualization from the data.

```
net = Network(notebook=True)
```

```
Local cdn resources have problems on chrome/safari when used in jupyter-  
→notebook.
```

```
net.from_nx(G)
```

```
net.save_graph("hp_network.html")
```

```
HTML(filename="hp_network.html")
```

```
<IPython.core.display.HTML object>
```

We now have a graph with nodes represented with different colors. Notice that the source node controls the edge color by default. NetworkX and PyVis both allow us to control the edge color in the same way as it did with nodes, but instead of adding `color` as an attribute of the `node`, we could also assign this to the `edge` as an attribute.

16.6 SNA on Humanities Data: Structuring the Data

In this section, we will apply the skills we have learned about SNA on real humanities data. Again, we will be working with the TRC Volume 7. Unlike the section on Topic Modeling, we will not be interested in how the descriptions of violence cluster together. Instead, we will be interested in exploring how organizations relate to specific individuals as they appear in their descriptions. It is important to note here, in this approach we do not know the relationship between the victim and the organization. There is equal possibility that they were a member or victim of the organization. To understand these relationships, we would want to use spaCy and a few NLP techniques to extract meaning about these relationships first.

16.6.1 Examining the Data

First, let's import our required libraries. We will only be concerned with structuring the data in this section, so we will only import Pandas. We will also import random because we want to assign a random color to each unique organization. This ensures that our approach scales quickly if we were to add thousands of new organizations into our dataset. For a polished, final version, one would want to manually assign a color to each organization so that the results would be more reproducible.

```
import pandas as pd  
import random
```

We will now load our data. We only need four of the columns: Last, First, ORG, and Place.

```
df = pd.read_csv("../data/trc.csv")
df = df[:1000]
df = df[["Last", "First", "ORG", "Place"]]
```

```
df
```

	Last	First	ORG	Place
0	AARON	Thabo Simon	ANC ANCYL Police SAP	Bethulie
1	ABBOTT	Montaigne	SADF	Messina
2	ABRAHAM	Nzaliseko Christopher	COSAS Police	Mdantsane
3	ABRAHAMS	Achmat Fardiel	SAP	Athlone
4	ABRAHAMS	Annalene Mildred	Police SAP	Robertson
..
995	CELE	Nompumelelo Iris 'Magwaza'	ANC	Ndwedwe
996	CELE	Nomvula Eunice	ANC	Umbumbulu
997	CELE	Nonhlanhla Evelina	ANC	Umzimkulu
998	CELE	Nozimpahla	NaN	Sonkombo
999	DLAMINI	Cedric Bongani	ANC	Durban

[1000 rows x 4 columns]

Now that we have our data, we can begin clean it and prepare it for inclusion in a NetworkX graph. Our goal is to create a list of nodes and edges separately which we will then store as two separate Pandas DataFrames. We will then be able to use this data for graph creation in the next section. To do this, we will use the following code:

```
nodes = []
edge_list = []
found_orgs = []
for idx, row in df.iterrows():
    node_id = f"{idx}_{(row.First)_{(row.Last)}}">
    place = row.Place
    nodes.append({"name": node_id, "color": "green", "place": place})
    if pd.isnull(row.ORG) == False:
        orgs = row.ORG.split("|")
        for org in orgs:
            if org not in found_orgs:
                color = "#" + ''.join([random.choice('0123456789ABCDEF') for j in range(6)])
                nodes.append({"name": org, "color": color})
                found_orgs.append(org)
    edge_list.append({"source": org, "target": node_id, "place": place})
print(nodes[:1])
print(edge_list[:1])
print(len(nodes))
```

```
[{"name": '0_Thabo Simon AARON', 'color': 'green', 'place': 'Bethulie'}]
[{"source": 'ANC', "target": '0_Thabo Simon AARON', 'place': 'Bethulie'}]
1021
```

Let's break this down. First, we create three separate lists that we will append to:

```
nodes = []
edge_list = []
found_orgs = []
```

The list `nodes` will contain a list of all nodes for the graph. The list `edge_list` will contain all the edges in our graph. The list `found_orgs` will be an easy way to know which organizations have already been found. This is to prevent us from adding an organization into our nodes list more than once.

Next, we begin iterating over our DataFrame and creating a unique `node_id` for each person. Remember, some individuals have the same first and last names. This means we need to assign a unique number to their name as well. We also want to give each node some extra metadata, namely the `place` that is referenced in their description and the `color` of green. This will keep all individuals in our graph the same node color.

```
for idx, row in df.iterrows():
    node_id = f"{idx}_{row.First} {row.Last}"
    place = row.Place
    nodes.append({"name": node_id, "color": "green", "place": place})
```

After this, we want to then add each organization to the node list if it does not appear already in there and then add an edge between it and the individual to which it is connected.

```
if pd.isnull(row.ORG) == False:
    orgs = row.ORG.split(" | ")
    for org in orgs:
        if org not in found_orgs:
            color = "#"+''.join([random.choice('0123456789ABCDEF') for j in range(6)])
            nodes.append({"name": org, "color": color})
            found_orgs.append(org)
            edge_list.append({"source": org, "target": node_id, "place": place})
```

Not all individual's, however, are connected to organizations. Sometimes, our `ORG` column is null. For this reason, we use the condition:

```
if pd.isnull(row.ORG) == False:
```

This checks to see if the `ORG` column is empty. If it is, we ignore it. If not, then we split up all the organizations into individual strings and then check to see if is in `found_orgs`. If not, then we assign it a random color and add it to our list of nodes. Next, we add it to `found_orgs` so that we do not add it again.

Once we have created our data, we can save each the node list and the edge list to CSV files.

```
node_df = pd.DataFrame(nodes)
node_df.to_csv("../data/nodes.csv", index=False)
node_df.head(1)
```

	name	color	place
0	0_Thabo	Simon AARON	green Bethulie

```
edge_df = pd.DataFrame(edge_list)
edge_df.to_csv("../data/edges.csv", index=False)
edge_df.head(1)
```

	source	target	place
0	ANC	0_Thabo	Simon AARON Bethulie

16.7 SNA on Humanities Data: Creating the Graph

Now that we have created our node data and edge data, let's go ahead and load them via Pandas.

```
import pandas as pd
```

```
nodes_df = pd.read_csv("../data/nodes.csv")
edges_df = pd.read_csv("../data/edges.csv")
```

```
nodes_df.head(1)
```

		name	color	place
0	0_Thabo	Simon AARON	green	Bethulie

```
edges_df.head(1)
```

	source	target	place	
0	ANC	0_Thabo	Simon AARON	Bethulie

With this data, we can now construct a node list and edge list with just a few lines of Python. In the code below, we will be iterating over each DataFrame and creating lists of nodes and lists of edges in the format that NetworkX expects. Remember, NetworkX wants to see each node in a node list rendered like this:

```
(NODE_ID, {METADATA})
```

Each edge in the edge list should be rendered like this:

```
(EDGE_SOURCE, EDGE_TARGET, {METADATA})
```

Remember, that these must be tuples and the metadata must be stored in the final index with each key being an attribute and each value being the value of that attribute.

```
nodes = []
edges = []
for idx, row in nodes_df.iterrows():
    nodes.append((row["name"], {"color": row.color, "place": row.place, "title": row["name"]}))
for idx, row in edges_df.iterrows():
    edges.append((row.source.strip(), row.target, {"place": row.place}))
```

```
nodes[0]
```

```
('0_Thabo Simon AARON',
 {'color': 'green', 'place': 'Bethulie', 'title': '0_Thabo Simon AARON'})
```

```
edges[0]
```

```
('ANC', '0_Thabo Simon AARON', {'place': 'Bethulie'})
```

16.7.1 Building the Network

```
from pyvis.network import Network
import networkx as nx
```

Now that we have our data structured correctly, we can begin constructing our network with NetworkX. We will first create our `Graph` class (`G`) and then add the nodes from our list of nodes and our edges from our list of edges.

```
G = nx.Graph()
G.add_nodes_from(nodes)
G.add_edges_from(edges)
```

When working with larger datasets, calculating the physics can often be time consuming when the PyVis HTML graph loads. For this reason, it can be useful to calculate the placement of the x and y coordinates beforehand. NetworkX gives us the ability to do this by leveraging the method of the algorithm we wish to use. In our case, we will use the `spring_layout()`. This method will take a single argument, the graph that we want to process. In our case, this is `G`.

This will return a list of lists. Each sublist will be x and y coordinates.

```
pos = nx.spring_layout(G)
pos['2_Nzaliseko Christopher ABRAHAM']
```

```
array([0.15706733, -0.01991105])
```

16.7.2 Visualizing the Network

First, we want to create our PyVis graph. We will be setting `notebook=True` because we are working within a Notebook.

```
net = Network(notebook=True)
net.from_nx(G)
```

```
Local cdn resources have problems on chrome/safari when used in jupyter-
→notebook.
```

Let's first take a look at the first node in our graph. We can access this individual's node by grabbing index 0 in the `nodes` attribute of our `net` object.

```
net.nodes[0]
```

```
{'color': 'green',
'place': 'Bethulie',
'title': '0_Thabo Simon AARON',
'size': 10,
'id': '0_Thabo Simon AARON',
'label': '0_Thabo Simon AARON',
'shape': 'dot'}
```

Now we want this node to have an x and a y coordinate as attributes and we want this data to come from the `pos` object above which contains the data about each node's position in the graph. To manually add these attributes to our nodes, we can use a `for` loop and create a new key of `x` and `y` and add those to each node.

```
for node in net.nodes:
    x, y = pos[node["id"]]
    node["x"] = x*10000
    node["y"] = y*10000
```

```
net.nodes[0]
```

```
{'color': 'green',
'place': 'Bethulie',
'title': '0_Thabo Simon AARON',
'size': 10,
'id': '0_Thabo Simon AARON',
'label': '0_Thabo Simon AARON',
'shape': 'dot',
'x': 220.5653302371502,
'y': 175.43811351060867}
```

Now that we have our PyVis graph created and assigned x and y coordinates, we can start to do some more advanced things. I can use `get_adj_list()` to create a dictionary of all nodes. Each key will be a unique node id. The values will be sets (lists with only unique items in each index) that correspond to the nodes to which they are connected. Because the purpose of this graph is to visualize relationships between people and organizations, set will include only organizations referenced in that individual's description.

```
neighbor_map = net.get_adj_list()
```

```
neighbor_map["0_Thabo Simon AARON"]
```

```
{'ANC', 'ANCYL', 'Police', 'SAP'}
```

In the PyVis official documentation, this data is used to generate a title that will pop up when an individual hovers over it. It is modified slightly to replace the `
` HTML tag with `\n`.

```
for node in net.nodes:
    node["title"] += " Neighbors:\n" + "\n".join(neighbor_map[node["id"]])
```

```
net.nodes[0]
```

```
{'color': 'green',
'place': 'Bethulie',
'title': '0_Thabo Simon AARON Neighbors:\nSAP\nANCYL\nPolice\nANC',
'size': 10,
'id': '0_Thabo Simon AARON',
'label': '0_Thabo Simon AARON',
'shape': 'dot',
'x': 220.5653302371502,
'y': 175.43811351060867}
```

The number of connections in our graph also tells us the relative importance of a node. Remember, we have all individuals connected to organizations; we do not have individuals connected to other individuals. This means the organizations in our graph will have a larger number of connections. What if we wanted to use that information to increase the size of each node in the graph based on the number of unique connections? For this, we can set a node's value based on the length of connections found in the `neighbor_map`.

```

for node in net.nodes:
    node["value"] = len(neighbor_map[node["id"]])

net.nodes[0]

{'color': 'green',
'place': 'Bethulie',
'title': '0_Thabo Simon AARON Neighbors:\nSAP\nANCYL\nPolice\nANC',
'size': 10,
'id': '0_Thabo Simon AARON',
'label': '0_Thabo Simon AARON',
'shape': 'dot',
'x': 220.5653302371502,
'y': 175.43811351060867,
'value': 4}

```

Now that we have all of our data, we can now visualize it all. Because we already calculated the physics of our graph, we do not want to enable physics, so we will set that to `False`.

```

net.toggle_physics(False)
net.save_graph("trc_graph.html")

```

The generated graph may be a bit difficult to interpret as the default window is quite zoomed out. This is because there are a lot of nodes in our graph that do not have any connections. A graph is a visual representation of mathematical relationships between nodes. The weaker nodes, or those with fewer connections, appear further away from the center of the graph. Likewise, stronger clusters, or those collection of nodes with the highest number of connections, will appear closer to the center. The gravity of these larger nodes push the smaller nodes with fewer connections further away in a spring loaded graph.

Because this is a dynamic graph, you can zoom in to examine the cluster if you are viewing this textbook online.

16.7.3 Adding Menus

Finding and isolating specific nodes in a large graph can be a bit difficult as well. What if you wanted to see where a specific node appeared? You would need to keep searching until you found that node. PyVis offers a solution to this problem with two types of menus. The first is a `Select` Menu. This allows you to select a specific node in the graph and highlight it. Depending on the graph size, it may take a few seconds for this visualization to take place. You can create a select menu with your graph by passing a keyword argument of `select_menu` and setting it to `True` when you first create your PyVis graph and call the `Network` class, like so:

```
net = Network(select_menu=True)
```

Note, if you are using a Jupyter Notebook to analyze your graph, this menu will appear but your graph data will not. This is due to the extra layer of JavaScript found within the HTML file. In order to visualize this graph, you will need to open it as a separate HTML file. The code below would create the same graph as above, but with a `Select` Menu.

```

net = Network(select_menu=True)
net.from_nx(G)

```

(continues on next page)

(continued from previous page)

```

neighbor_map = net.get_adj_list()
for node in net.nodes:
    x, y = pos[node["id"]]
    node["x"] = x*10000
    node["y"] = y*10000
    node["title"] += " Neighbors:\n" + "\n".join(neighbor_map[node["id"]])
    node["value"] = len(neighbor_map[node["id"]])
net.toggle_physics(False)
net.save_graph("trc_graph_select.html")

```

When you open the generated HTML file, you will see this:

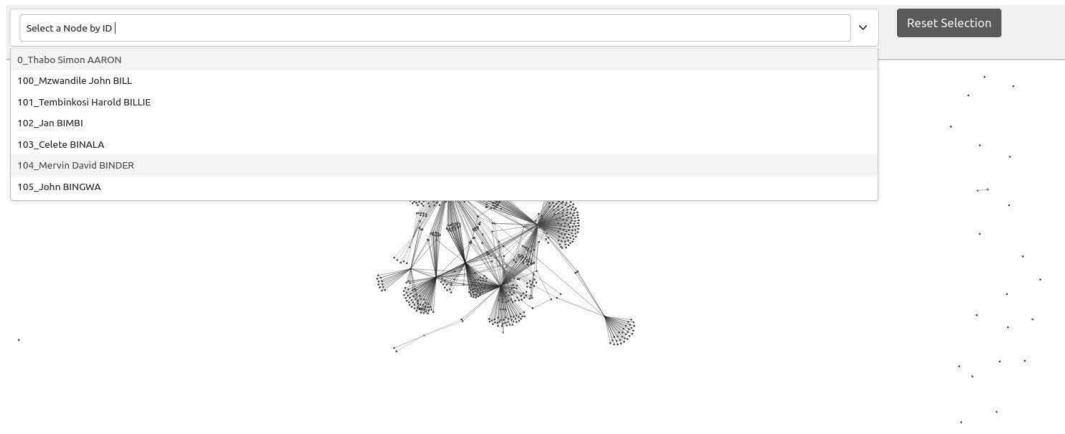


FIGURE 16.1

Demonstration of Network Graph with a Select Menu.

You can select a node in the graph and highlight that particular node. Let's say I wanted to see all nodes connected to the ANC in the graph. I would select ANC. The graph will then highlight that particular node and its connections.

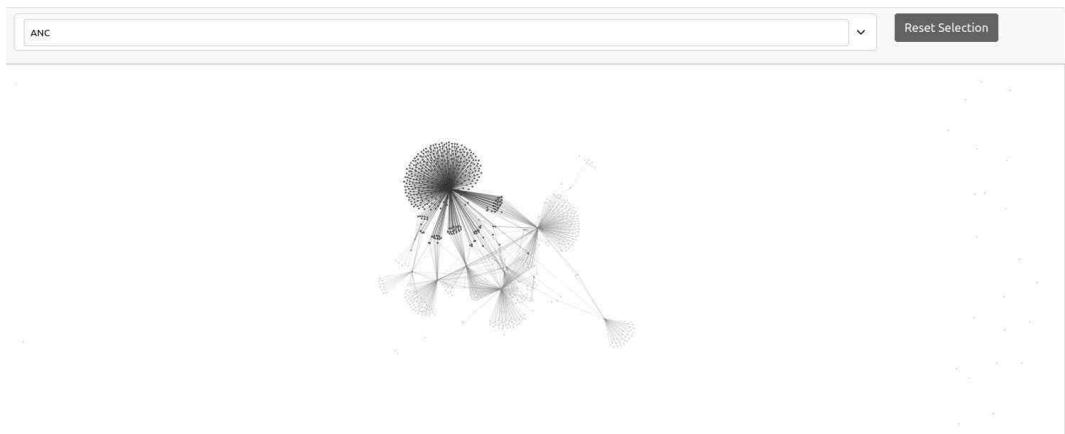
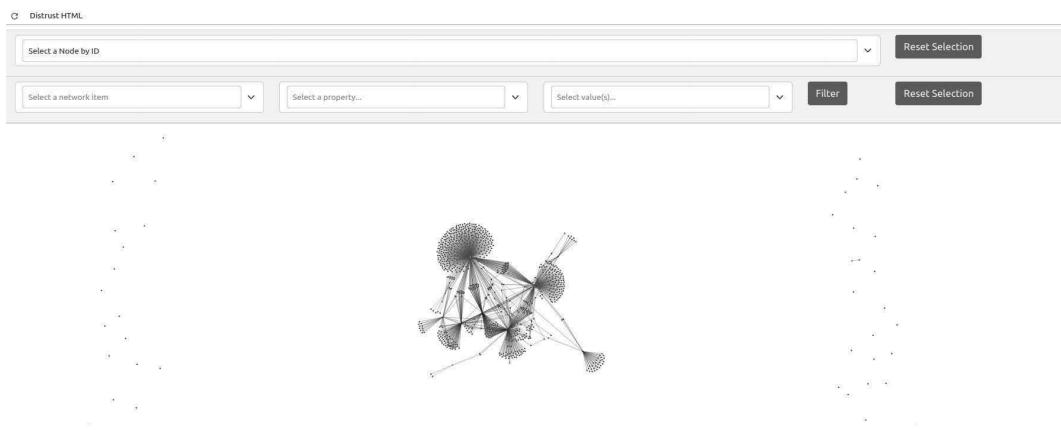
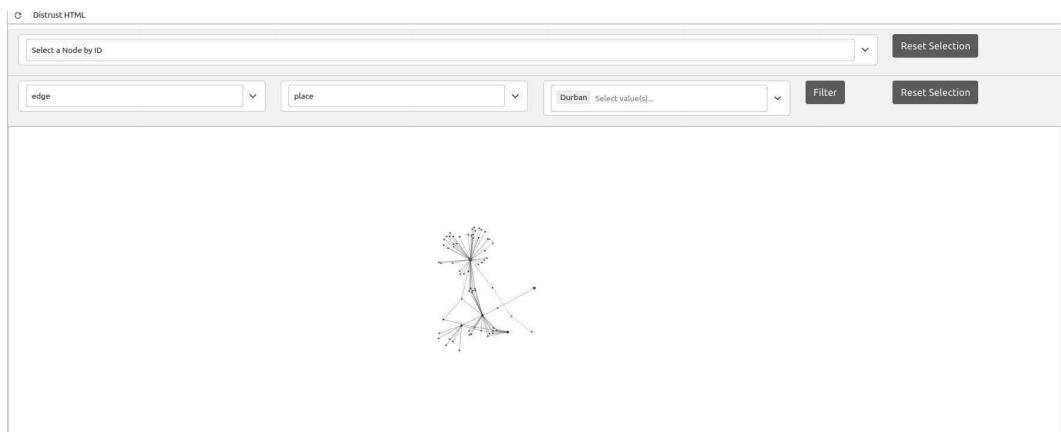


FIGURE 16.2

Demonstration of Selecting Item in Select Menu.

**FIGURE 16.3**

Demonstration of Filter Menu in the Application.

**FIGURE 16.4**

Demonstration of Selecting Filters in the Filter Menu.

PyVis also offers a way to filter the graph with a `Filter Menu`. The filter menu allows you to find nodes or edges that have specific metadata. In the previous notebook, we made sure that our nodes and edges contained metadata about the place that was connected to it. This means that we can isolate the relevant edges or nodes in the graph with this metadata. Again, this makes it a lot easier to find relevant material and identify patterns in your data that may not be so easy to do as raw data. We can create a `Filter Menu` by passing a keyword argument `filter_menu` when we create the `Network` class and setting it to `True`.

```
net = Network(select_menu=True, filter_menu=True)
net.from_nx(G)
neighbor_map = net.get_adj_list()
for node in net.nodes:
    x, y = pos[node["id"]]
    node["x"] = x*10000
    node["y"] = y*10000
```

(continues on next page)

(continued from previous page)

```
node["title"] += " Neighbors:\n" + "\n".join(neighbor_map[node["id"]])
node["value"] = len(neighbor_map[node["id"]])
net.toggle_physics(False)
net.save_graph("trc_graph_select_filter.html")
```

When you open the HTML file that is created, you will see the following graph:

Notice the addition of the Filter Menu below the Select Menu. You can select between nodes or edges as the item to filter and then select which piece of metadata. In our case, we want to filter by place. We then select the place that we want to isolate and view. In our case, let's view Durban. We can then press Filter and view the results.

16.7.4 Conclusion

Applying SNA to humanities data is not always the right solution to the problem, but if you are dealing with many pieces of data that are interconnected with different types of relationships, it can offer you a great way to quickly get a sense of patterns that you may otherwise miss. As a humanist, you can then use this information to generate questions or perhaps have a specific collection of sources or nodes that you can explore more closely. This chapter has not covered all aspects of SNA nor all the libraries for performing it via Python, but you should have a strong enough basis to begin applying it to your own data with minor modifications.

Part IV

Designing an Application with Streamlit



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

17

Introduction to Streamlit

In this part of the textbook, we will learn how to build a custom application in Python with Streamlit and host that application in the cloud. In Chapter 18, we cover the basic of Streamlit, its utility, the key terminology, and how to display data on a page for a user. Here, readers will gain an understanding of the main widgets Streamlit offers, how to use them and why their useful. In Chapter 19, we dive into more advanced features of Streamlit to produce better looking and more complex applications by creating data visualizations, controlling the layout of our application, caching data, leveraging custom HTML, and how to design multi-page apps. In Chapter 20, readers will build upon this knowledge and learn how to develop a custom application and deploy it on Streamlit Share. We will design a database querying application that will be based around Pandas.

17.1 Creating Our First App

17.1.1 Options for Application Development in Python

In Python, there are several options available to those interested in designing applications. Selecting the right one depends on a few different factors, from customization, deployment (where the application will sit), performance, and speed with which it can be designed. Five years ago, if one wanted to design an application in Python, one needed to use the library Tkinter. This would have a Python-based application that could run locally. The key issue with Tkinter is that it has a steep learning curve, it can only run locally (unless you are willing to package a massive and slow .exe for a simple application), and its aesthetics leaves something to be desired.

If one wanted to deploy an application on the web, there were two options available to researchers until a few years ago: flask and Django. Django has a steep learning curve but allows users to design and build entire websites that are entirely Python based. It also requires a detailed knowledge of servers in order to get an app running in the cloud effectively. Flask is a simplified version of Django that does not require knowledge of server side development for deployment.

Today, there are two options that allow researchers to make applications in minutes. The first is Bokeh. Bokeh allows one to create a Python server based application or write custom JavaScript functions that allow for an application to be compiled and distributed as a single HTML file. Bokeh excels at interactive applications where a user can manipulate a graph that simultaneously changes the output of a table, for example.

For this textbook, we will learn how to design applications with Streamlit. I have chosen Streamlit for this textbook for a few reasons. First, Streamlit is relatively easy to learn. You can have your first application up and running in minutes. Second, it is well-maintained. This means that new features are added regularly. The maintainers of the library listen

to their users and consistently provide features that they need. Third, Streamlit was just purchased by Snowflake which means it should continue to exist far into the future. This means that this section of the textbook will not likely go out of date quickly. Fourth, spaCy has pre-built Streamlit components, meaning you can create a spaCy-based application in seconds, not hours. Fifth, Streamlit has a large community and an active forum and Discord channel. This means that if you need to do something in Streamlit, there is likely a tutorial available; if you encounter a bug, there is likely a solution on the forum; if a solution is not on the forum, someone will help you if you ask.

All of these reasons make Streamlit the logical choice for a textbook designed for those with limited coding experience.

17.1.2 Installing Streamlit

In order to begin designing applications with Streamlit, you first need to install it. Like all other Python libraries, you can do so by using pip. Once you execute the code below, you will install Streamlit locally.

```
pip install streamlit
```

Remember, if you are installing Streamlit via a Jupyter Notebook, you will need to add a leading !:

```
!pip install streamlit
```

The entire Streamlit application used for teaching purposes in this book can be found in the main repository under `streamlit_application`. To view the application while reading the book, you can run the following command:

```
streamlit run Home.py
```

If you have installed Streamlit correctly, you should see a Streamlit application in your browser. In this section, we will learn about all the features covered in the demonstration application. In the final section of this chapter, we will learn how to build a basic application involving real-world humanities data.

17.1.3 Creating a Home Page

To get started with Streamlit, you only need one Python (.py) file. This will be the main page of your application. For our purposes, we will call this `Home.py`. Once you have created the Python file, you should import Streamlit. You can do so with the following command on the first line of the Python file.

```
import streamlit as st
```

Note that we are importing Streamlit specifically as `st`. This adheres to the official Streamlit documentation. One should stick to this convention as nearly all Streamlit users follow this and it is expected. It will, therefore, make your code easier to understand for others and it will also make it easier for you to debug issues that surface as your code will conform to the expected standards.

Once you have created your file, you can run the following command in your command line:

```
streamlit run Home.py
```

This will create your application on a local server (on your local computer) and populate that server in your browser. Streamlit functions by continuing to run the Python script in the background. This means that as you develop your application, you can see those developments in real time. In the top-right corner, you will see a hamburger icon (three horizontal lines, see Figure 17.1). If you click this button, you will see several options. One of these is Rerun. This will let you rerun your application in real-time.

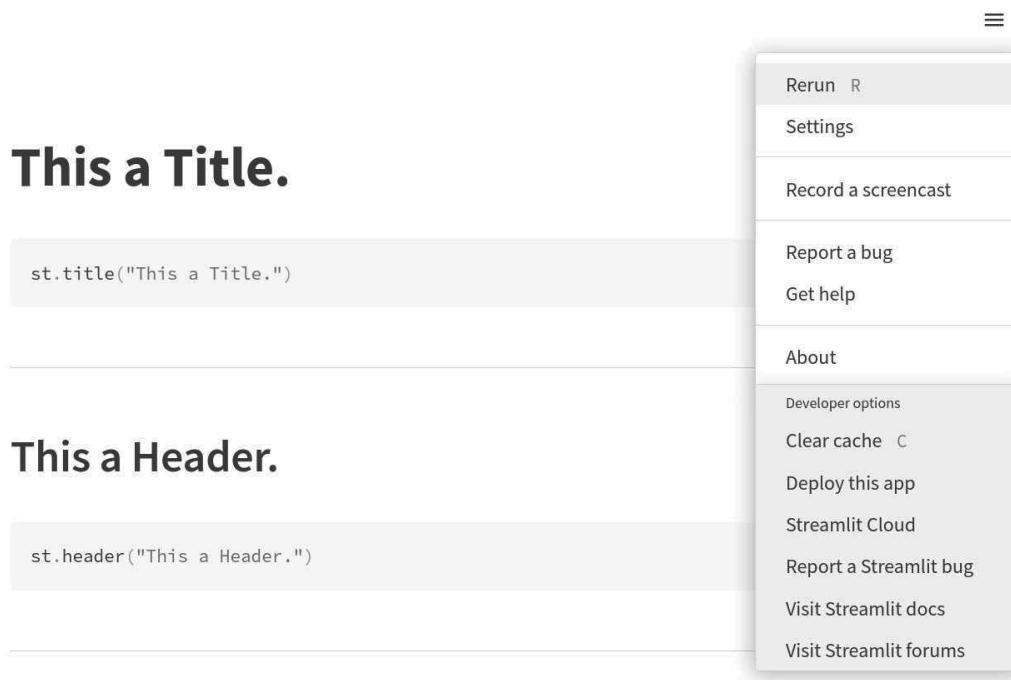


FIGURE 17.1
Location of the Hamburger Icon.

17.2 Displaying Data in Streamlit

Streamlit offers numerous ways to display different types of data. In this section, we will be looking at a few of those methods from raw text, to structured markdown, to Pandas DataFrames, and even images.

17.2.1 Displaying Text to Users

Once you have imported streamlit, you can use the Streamlit library to create an application with just a few lines of code. To follow along in the repository, you can view the 01-Displaying Data.py file or the Displaying Data page of the application.

Streamlit offers a number of ways to display information to the users of your application. One of the most common things we must convey to a user is text. This can be a title, it can

explain basic information about the application, or it can be to display results from some prompt. Streamlit offers several ways to display information to users.

```
st.title
```

Every application needs a name. If you want the title of your application to stand out on the page, you can use `st.title()` to output a title on your application page. This will take one argument, a string which will be your application title. It is entirely optional to have a title. To add the title, you would place the following line into your Python file.

```
st.title("This a Title.")
```

Once you have added this to your Python file, you can hit refresh in your Streamlit application, and you should see a title page now appear at the top of your page.

```
st.header
```

Now that we have a title, we can add some extra layers to our application, rather like a traditional HTML website. We can add headers, for example, with `st.header()`. Again, this will take a single argument, the text that you wish to display as a header. You can add a header to your application by adding the following line to the same Python file and hitting refresh:

```
st.header("This a Header.")
```

Notice that our header appears below our title. This is because Streamlit reads the Python file top-to-bottom as it reruns in the background. If you want an item to appear higher in the application, you must place it earlier in your Python file. Additionally, you can use containers which we will meet later in this chapter.

```
st.subheader
```

Just like `st.header`, `st.subheader` adds a subheading to your application to allow you to have even greater nested structure. If you add the following line to your Python file and hit refresh, you will see a subheading appear.

```
st.subheader("This a Subheader.")
```

```
st.write
```

The most common way to display text to a user is with `st.write()`. With this command, we can pass a single argument, some sort of data that we want to display. Let's use this command and display the string "This is text.". To do this, we would add the following line to our Python file:

```
st.write("This is text.")
```

Streamlit's `st.write` is quite powerful. As we will see below, it can display data structures, such as lists and dictionaries, as well as entire dataframes, automatically.

```
st.caption
```

In Streamlit, it may be necessary to caption something. As we will see, images can be captioned separately. To caption something in your application, you will use `st.caption()`. Again, this will take a single argument, the string that we wish to display as a caption. We can add a caption to our application by adding the following line to our Python file:

```
st.caption("This a Caption.")
```

```
st.markdown
```

Finally, we have `st.markdown`. Markdown is a type of language that allows you to structure text quickly. Markdown is easy-to-read for both humans and machines and is the standard language used for README pages. Markdown files end with a `.md` extension.

Streamlit allows users to leverage the power of markdown inside a Streamlit application. This means that we can read in a markdown file stored in the same directory as our application and automatically display that file's contents in our application. This is really useful when you have pages or sections on a page that require longer section of text or, perhaps, things like lists. These types of long strings can often crowd a Python file. In our demonstration Streamlit application, for example, we can see this in action with the following lines added to our Python file.

```
with open("./markdown/sample.md", "r") as f:  
    markdown_text = f.read()  
st.markdown(markdown_text)
```

As we will learn later in this book, `st.markdown` can also be used to display HTML which makes it even more powerful.

17.2.2 Displaying Python Data Structures

It is often necessary to display raw data inside an application either during the development phase of the application or in production (when users will engage with the app). Streamlit has two different ways to display data with the same results: `st.write()` or `st.json()`. On the surface, they both do the same thing, but `st.json` allows you to set an expanded keyword argument to `True` or `False`. This allows you to control if the data appears in its entirety in the application or as a expandable option within the application.

17.2.2.1 Data Structures with `st.write()`

Let's first test this out with `st.write()`. We can use the following two lines to create a dictionary called `names` and then `st.write()` to display that data.

```
names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}  
st.write(names)
```

The output will look like Figure 17.2 in your application.

17.2.2.2 Data Structures with `st.json()`

For our second option, we can use `st.json`, but note that we are able to specify expanded here as a keyword argument.

```
names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}  
st.json(names, expanded=False)
```

The output will look like Figure 17.3 in your application.

17.2.3 Displaying Tabular Data

There are four ways to display tabular data within Streamlit.

```

▼ { 📁
  ▼ "people" : [
    0 : "Tom" 📁
    1 : "Mary"
    2 : "Fred"
    3 : "Stephanie"
  ]
}

```

FIGURE 17.2

Example of Dictionary Output with `st.write()`.

```
▶ { ... }
```

FIGURE 17.3

Example of Dictionary Output with `st.json()`.

- `st.write()` (defaults to `st.dataframe()`)
- `st.dataframe()`
- `st.table()`
- `st.markdown()`

While on the surface, these may all seem to display the same data, understanding what each does is important so that you can have your application do precisely what you wish.

17.2.3.1 Tabular Data with `st.write()`

If you are trying to test an application quickly and just want to display tabular data without any extra customization, then `st.write()` is perfectly suitable. If Streamlit detects a Pandas DataFrame as the object that is being passed to `st.write()`, it will automatically output that data via `st.dataframe()`.

```
st.write(df)
```

17.2.3.2 Tabular Data with `st.dataframe()`

If you want to have more control over how your tabular data is displayed in your application, you may want to use `st.dataframe` instead. By using `st.dataframe`, you can control the width and height of the displayed dataframe.

```
st.dataframe(df, height=750)
```

The output will look like Figure 17.4 in your application.

With both `st.write()` and `st.dataframe`, users will be given a Streamlit dataframe display. This means that they can highlight certain parts of the dataframe, expand cells to

	PassengerId	Survived	Pclass	Name	Sex	Age	Si
0	1	0	3	Braund, Mr. Owen Harris	male	22.0000	
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38.0000	
2	3	1	3	Heikkinen, Miss. Laina	female	26.0000	
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0000	
4	5	0	3	Allen, Mr. William Henry	male	35.0000	
5	6	0	3	Moran, Mr. James	male	<NA>	
6	7	0	1	McCarthy, Mr. Timothy J	male	54.0000	
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0000	
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27.0000	
9	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14.0000	

FIGURE 17.4

Example of DataFrame Output with `st.dataframe()`.

read longer text, and sort the data. In other words, the dataframe is an entirely interactive display widget.

17.2.3.3 Tabular Data with `st.table()`

One of the downsides the `st.dataframe` display is that the interactivity comes at the cost of aesthetics. If you are working with humanities data, you may have a lot of text in your tables. That text can be difficult for viewers to read in the standard `st.dataframe` output. In these situations, `st.table` may be more appropriate.

```
st.table(df)
```

17.2.3.4 Tabular Data with `st.markdown()`

A key limitation of both the `st.table()` and `st.dataframe` is that they do not offer a way to display images. With markdown, we can easily display images within our tables. This, however, comes at the cost of not being able to sort the output. We will learn how to do this later when we work with custom HTML in our Streamlit application. For now, understand that you can convert a Pandas DataFrame to markdown by using the `to_markdown()` method.

```
st.markdown(df.to_markdown())
```

17.2.4 Displaying Multimedia in Streamlit

Streamlit also offers the ability to easily add multimedia into your application. For all three types of media (images, audio, and video), Streamlit allows you to place the media in the app in four ways:

- file path
- from a url
- from a NumPy Array
- from bytes

Each of these has its own uses. If the media you have is available locally within the app (such as logos and design elements), from file path usually makes the most sense. If your data sits on a server on the Web, then url is the right choice. These will be images that do not sit within the code of your application or in the local directory. When you have a user input media data into the app (via file upload), you will want to load the data via NumPy Array or Bytes. We will see these last two in action during the final chapter of this section when we apply Streamlit to develop real digital humanities applications.

17.2.4.1 *Images*

If we are working with images, we can load a local image with the following line:

```
st.image(path_to_image)
```

17.2.4.2 *Audio*

For audio, we would use the following line:

```
st.audio(path_to_image)
```

17.2.4.3 *Video*

For video we would use the following line:

```
st.video(path_to_image)
```

17.3 Streamlit Input Widgets

Often when you are designing your application, you will need a way to allow the user to interact with the app. When this occurs, you want a way to do something that that user input. There are many ways that we can allow a user to interact with our data in Streamlit. In this section, we will cover five of the main categories:

- text input
- numerical input
- date input
- boolean input
- selection input

Cheatsheet for Widgets in Streamlit

Widget	Return	Description
text_input	string	A small text region
text_area	string	A large text region
number_input	integer or float	A typed number input
slider	integer or float	A slider for number input
date_input	timeseries	A calendar for selecting a date
time_input	timeseries	A dropdown menu for selecting a time
checkbox	Boolean	A checkbox for marking something as true or false
button	Boolean	A button for triggering an event
radio	string	A selection for a single option
selectbox	string	A selection for a single option (dropdown menu)
multiselect	list	A selection for multiple options (dropdown menu)

17.3.1 Text Input Widgets

Streamlit offers two ways to allow users to input textual data into an application. You can either use `st.text_input()` or `st.text_area()`. Both essentially do precisely the same thing, that is, return a string from the user; but each should be used in specific situations. Streamlit's `st.text_input()` is designed for shorter text (such as names, queries, etc.), while `st.text_area()` should be used for longer string input data, such as text that can be processed via a spaCy pipeline.

17.3.1.1 `st.text_input()`

```
user_text = st.text_input("Input some text here")
st.write(user_text)
```

The output will look like Figure 17.5 in your application.



Input some text here

FIGURE 17.5
Example of the `st.text_input()`. Widget

17.3.1.2 `st.text_area()`

To create a text area style input, you can use the precise same code, but replace `text_input` with `text_area`. Both of these classes also let us pass an additional argument for some default text as the second parameter. We can add some default text that will prepopulate the text field with a predetermined string.

```
default_text = st.text_area("Input some text here", "default text")
st.write(default_text)
```

The output will look like Figure 17.6 in your application.

Input some text here

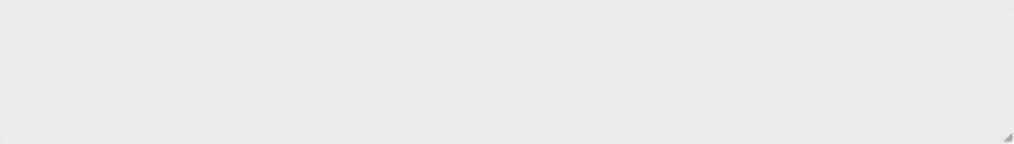


FIGURE 17.6
Example of `st.text_area()` Widget.

17.3.2 Numerical Input Widgets

While you could let a user input numerical data inside of `st.text_input()` or `st.text_area()`, this really would be inappropriate. Both of these input options return strings. This means that if you gave the user the ability to input numerical data, you would have to convert it to either an integer or float. In addition to this, the text input widgets do not offer any special keyword arguments that you can pass to the widget that are specific to numerical data, such as minimum value and maximum value.

It is far better in these scenarios to use one of two options for numerical input.

17.3.2.1 `st.number_input()`

The first option is `st.number_input()`. This feature lets a user input a numerical data that they can tick up or down via a minus or plus sign in the widget. The widget will return an integer or float, depending on how you structure the widget.

Users can also manually type in a specific number. Another feature of this widget is the ability to specify a minimum value (with the `min_value` argument) and maximum value (with the `max_value` argument). You can also give the user a default value by setting the `value` argument to a specific number. Finally, you can even provide a `step` argument which will step up every n-numbers, so a step of "2" would increase the value by two each time the user clicks the plus symbol in the widget.

```
user_number = st.number_input("Input Number",
                             min_value=1,
                             max_value=10,
                             value=5,
                             step=1)
st.write(user_number)
```

The output will look like Figure 17.7 in your application.

17.3.2.2 `st.slider()`

Another way to let a user input data is via the `st.slider()` widget. This widget will also return an integer or a float, depending on if your values are in decimal form.

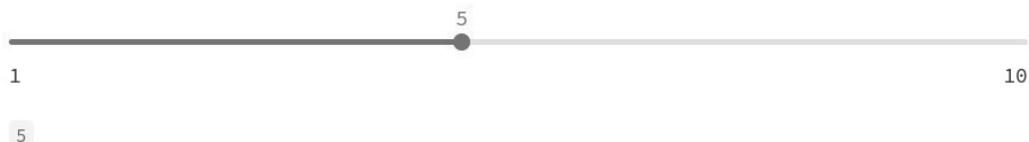
**FIGURE 17.7**

Example of `st.number_input()` Widget.

```
slider_number = st.slider("Select your Number",
                           min_value=1,
                           max_value=10,
                           value=5,
                           step=1)
st.write(slider_number)
```

The output will look like Figure 17.8 in your application.

Select your Number

**FIGURE 17.8**

Example of `st.slider()` Widget.

17.3.3 Date and Time Input Widgets

Working with dates and times is essential in a lot of applications and Streamlit has two widgets for receiving time-series data. Both will require the use of the built-in library `datetime`, so be sure to import this if you intend to work time time-series data in your application.

17.3.3.1 `st.date_input()`

The first widget is `st.date_input()`, this allows you to receive a date object which will allow you to structure robust logic, such as finding all data that fall between a start date and end date. When creating the widget, you can simply use the defaults, but if you expect all your data to fall between two dates, it may be wise to set minimum and maximum values. You can do this via the `datetime` library `date` method. You should ensure that all values in the `st.date_input()` widget conform to the following format:

```
datetime.date(2000, 6, 12)
```

In practice, this is what your widget would look like:

```
import datetime
user_date = st.date_input("Select your Date",
                         value = datetime.date(2000, 6, 12),
                         min_value = datetime.date(2000, 1, 12),
                         max_value = datetime.date(2001, 1, 12)
                         )

st.write(user_date)
```

The output will look like Figure 17.9 in your application.

Select your Date

2000/06/12

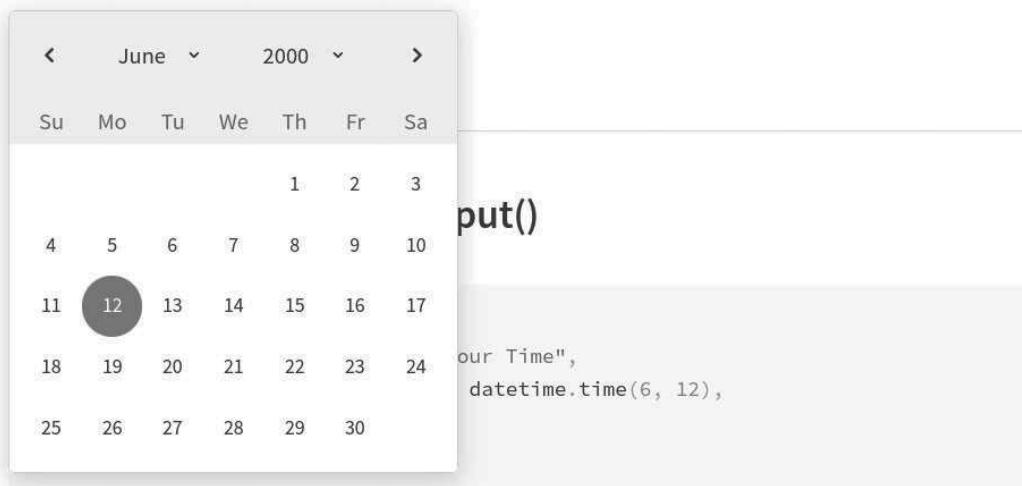


FIGURE 17.9
Example of `st.date_input()`. Widget.

17.3.3.2 `st.time_input()`

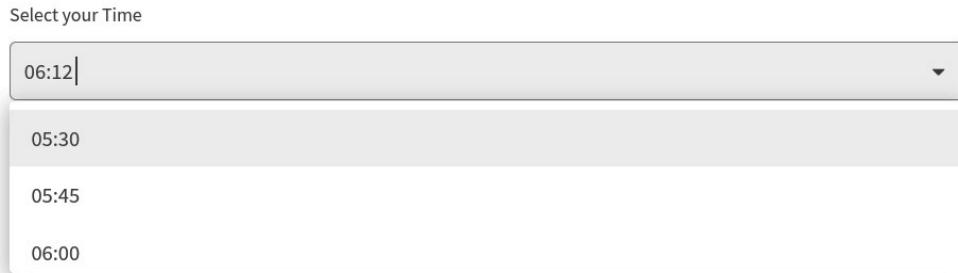
You can do precisely the same thing with time via the `st.time_input()` widget. With time, however, we cannot set min or max values. Also, unlike the `st.date_input()`, the `st.time_input()` will use the `time` method from the `datetime` library.

```
import datetime
user_time = st.time_input("Select your Time",
                         value = datetime.time(6, 12),
                         )
st.write(user_time)
```

The output will look like Figure 17.10 in your application.

17.3.4 Boolean Input Widgets

Another important feature for controlling the logic of your application is understanding Boolean (True or False) values from a user input. In Streamlit, we have two ways of using Boolean inputs to control the logic and they both function a bit differently.

**FIGURE 17.10**

Example of `st.time_input()`. Widget.

17.3.4.1 `st.checkbox()`

The first is `st.checkbox()`. This allows us to create a checkbox. Its state can be constantly changed. So a box can be checked or unchecked. As a user changes the state of the checkbox, the Boolean output from the widget will change. We can create a checkbox widget and write out the results with the following lines of code.

```
checked = st.checkbox("Select this checkbox")
st.write(f"Current state of checkbox: {checked}")
```

The output will look like Figure 17.11 in your application.

Select this checkbox

Current state of checkbox: False

FIGURE 17.11

Example of `st.checkbox()` Widget.

17.3.4.2 `st.button()`

Unlike the `st.checkbox()` widget, the `st.button()` widget will have a continuous state. This means that once the button is clicked, its Boolean output value will forever change unless you specifically change it in your script. In other words, the button's state at the start of the application is `False`, but once a user clicks the button, that state will be `True` continuously. This is important because it means that the click of a button can trigger a one-time event, such as draw a map or run a machine learning model.

You can create a basic button with the code sample below.

```
state = st.button("Click to Change current state")
st.write(f"Button has been pressed: {state}")
```

The output will look like Figure 17.12 in your application.

```
Press Button
Button has been pressed: False
```

FIGURE 17.12
Example of `st.button()` Widget.

17.3.5 Selection Widgets

The final collection of important widgets in Streamlit are the selection widgets. These allow you to give users a set of options to choose from. There are three types of selection widgets.

17.3.5.1 `st.radio()`

The first is `st.radio()`. This widget allows you to give the user the ability to select one item from a list of options. Only one option can be selected by the user.

You can use the code below to give the user the ability to choose between three different colors.

```
options = ["Red", "Blue", "Yellow"]
radio_selection = st.radio("Select Color", options)
st.write(f"Color selected is {radio_selection}")
```

The output will look like Figure 17.3 in your application.

```
Select Color
 Red
 Blue
 Yellow
Color selected is Red
```

FIGURE 17.13
Example of `st.radio()` Widget.

The `st.radio()` widget will return a string of the selected option.

17.3.5.2 `st.selectbox()`

The same logic holds true for the `st.selectbox()` widget. The only difference between this and the `st.radio()` widget is the aesthetic way the options are presented. A selectbox is often more appropriate if you are presenting a user with a larger selection of options, which would be clunky with radio buttons. This will return a string of the selected item.

```
options = ["Red", "Blue", "Yellow"]
selectbox_selection = st.selectbox("Select Color", options)
st.write(f"Color selected is {selectbox_selection}")
```

The output will look like Figure 17.14 in your application.

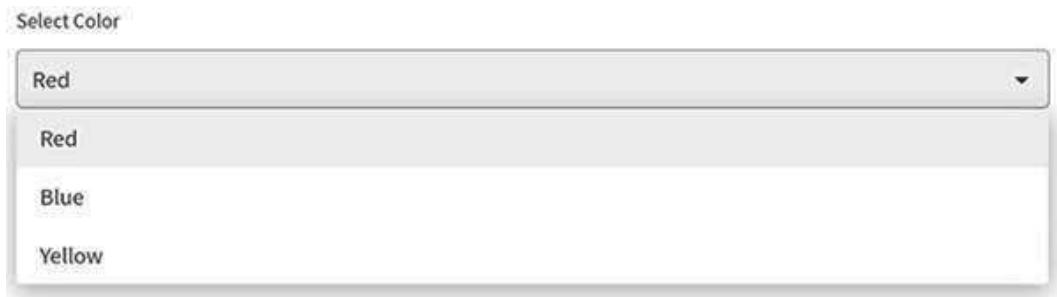


FIGURE 17.14
Example of `st.selectbox()` Widget.

17.3.5.3 `st.multiselect()`

The final selection widget is the `st.multiselect()` widget which allows a user to select multiple items from a selectbox. This will return a list of the selected items in the options.

```
options = ["Red", "Blue", "Yellow"]
multiselect_selection = st.multiselect("Select Color", options)
st.write(f"Color selected is {multiselect_selection}")
```

The output will look like Figure 17.15 in your application.



FIGURE 17.15
Example of `st.multiselect()` Widget.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

18

Advanced Streamlit Features

18.1 Data Visualization

When working with large quantities of data, it can often be difficult to present that data in a way that makes sense to non-experts. In these situations, we often rely on some form of chart to represent our data. Creating quality graphs in Python requires a lot of practice, but quality charts can be produced with Matplotlib, Altair, PyDeck, PyVis, Plotly, and Bokeh. Each has their own strengths. In this section, we will not go into these libraries, rather we will focus on how to present these different types of graphs in Streamlit.

In Streamlit, we can leverage these libraries to produce visually appealing charts in just a few lines of code. We will focus on three types of graphs: Basic Plot Graphs, Map Graphs, and Network Graphs.

18.1.1 Metrics

Before we address plots, we should spend a brief moment and think about how we display raw numerical quantitative data. We could display the length of a dataframe or the word count of some text `st.write()`. Again, while this would be quick to do, it would not allow you to display other important information, such as how that number has changed from a previous state. Nor would it allow you to easily represent the change in a positive or negative direction without complex JavaScript and HTML. In these scenarios, we would want to use `st.metric()`.

By default, the metric will display a numerical output of some sort. This number could also be a string representation of a number, e.g. temperature. Let's say, we wanted to create a simple application where a user could copy-and-paste some text into a `st.text_area()` field. The app would split up the words at every white space and then provide the user with the total word count.

```
import streamlit as st
text = st.text_area("Paste text here to get word count.", "This is some...  
→default text.")
word_count = len(text.split())
st.metric("Word Count", word_count)
```

The output will look like Figure 18.1.

Since we are using the `st.metric()` widget, however, we can also pass in a keyword argument that displays the degree to which the metric changed from the previous state. To do this, we will need to leverage the Streamlit Session State, which we will meet later in this chapter. This allows us to store a variable across different runs of the application. For now, we can ignore this bit of the code below and focus on the third argument that we passed to `metric, change`. This will display a change feature in the widget that will show the up or down trend of the change in green and red color, respectively.

Paste text here to get word count.

This is some default text.

Word Count

5

FIGURE 18.1

Example of Standard st.metric() Widget.

```
if "prev_word_count" not in st.session_state:
    st.session_state["prev_word_count"] = 5
text = st.text_area("Paste text here to get word count.", "This is some default text.")
word_count = len(text.split())
change = word_count - st.session_state.prev_word_count
st.metric("Word Count", word_count, change)
st.session_state.prev_word_count = word_count
```

The output will look like Figure 18.2.

Paste text here to get word count.

This is some default text. And this is some new text. |

Word Count

11

↑ 6

FIGURE 18.2

Example of st.metric() Widget with Change.

Metric is a useful feature that allows us to create apps that display numerical data in easy-to-understand ways. But in other situations, a single qualitative number may not be appropriate. Here is where charts come in handy.

18.1.2 Plotting Basic Graphs with Streamlit

We can plot basic graphs in Streamlit by passing a Pandas dataframe to different chart widgets in Streamlit. The first basic plot we can create is a line chart which we can create with the Streamlit widget `st.line_chart()`. We will be working with the Titanic dataset here that we first met in Part II of this textbook. To prepare the data for visualization, we need to modify it a bit and group everything by the specific value that we want to plot.

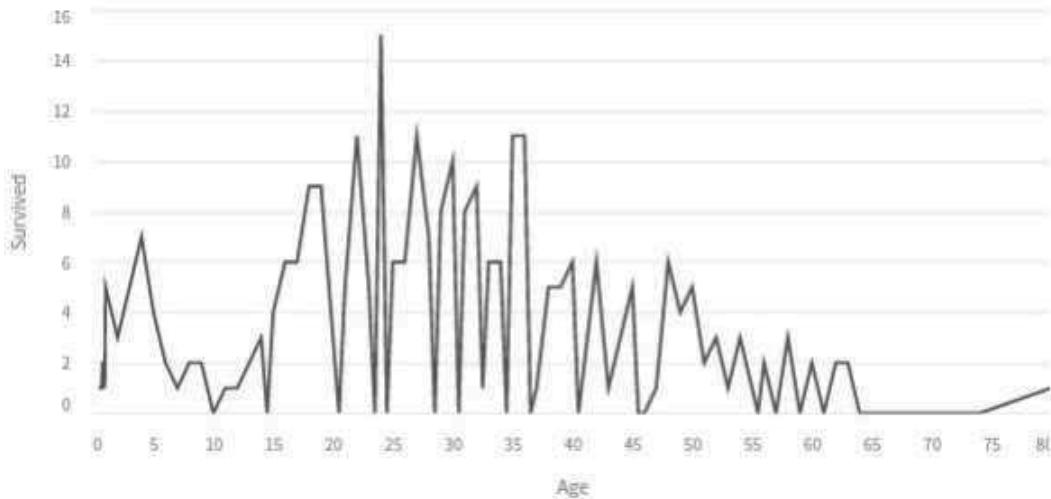


FIGURE 18.3
Example of Streamlit Line Chart.

In our case, we want to visualize the number of survivors for different age groups on the Titanic. We can prepare our dataframe with the code below.

```
df = pd.read_csv("data/titanic.csv")
df = df[["Age", "Survived"]]
chart_df = df.groupby(["Age"]).sum()
chart_df["Age"] = chart_df.index
```

18.1.2.1 Line Charts with `st.line_chart()`

Once we have created our new `chart_df`, we can pass it to `st.line_chart()`. Here, we will pass the entire dataframe as the first argument and specify our x axis and y axis on the graph. In our case, we want to view the `Age` column on the x axis and the `Survived` column on the y axis.

```
st.line_chart(chart_df, x="Age", y=["Survived"])
```

The output will look like Figure 18.3.

18.1.2.2 Bar Charts with `st.bar_chart()`

Likewise, we can present this same data as a bar chart with the widget `st.bar_chart()`. This will take the same arguments as above.

```
st.bar_chart(chart_df, x="Age", y=["Survived"])
```

The output will look like Figure 18.4.

18.1.2.3 Area Charts with `st.area_chart()`

And finally we can also use the same arguments to create an area chart with the `st.area_char()` widget.

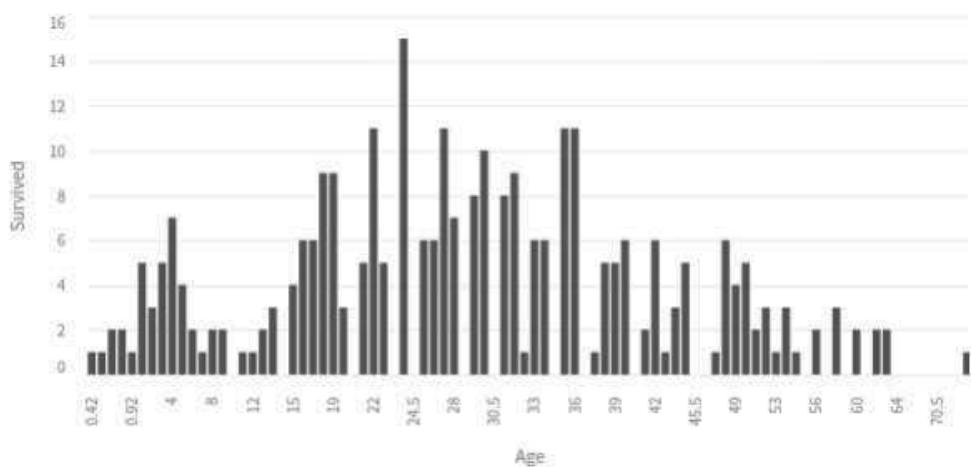


FIGURE 18.4
Example of Streamlit Bar Chart.

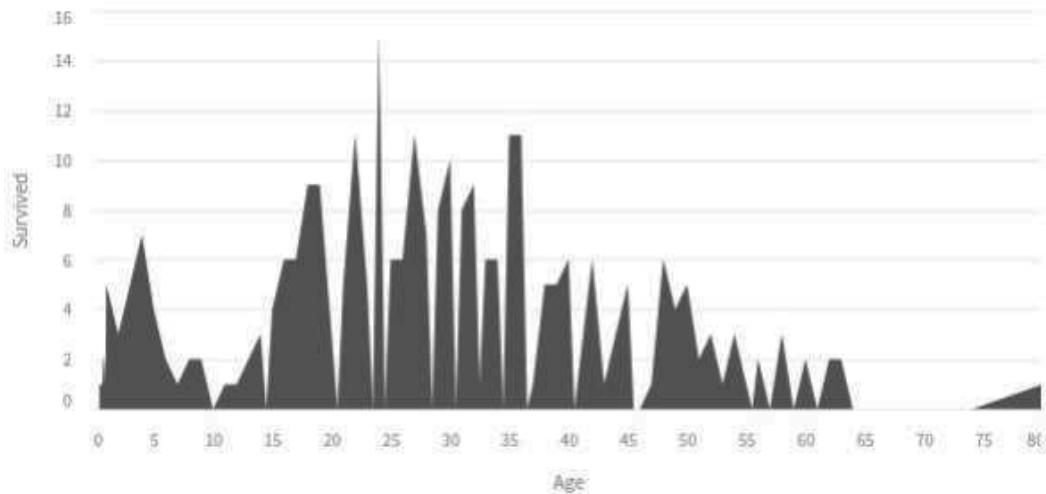


FIGURE 18.5
Example of Streamlit Area Chart.

```
st.area_chart(chart_df, x="Age", y=["Survived"])
```

The output will look like Figure 18.5.

18.1.3 Map Charts

A lot of digital humanities data is geospatial, or data that can be plotted on a map. Streamlit affords the ability to map geospatial data in several different ways; first, via the standard `st.map()` widget and second via the third-party chart libraries. Regardless of the library used, you will want to prepare your data well where your coordinates are labeled as either `lat` or `latitude` for the latitude and `lon` or `longitude` for the longitude. For this

demonstration, we will be working with data from South Africa's Truth and Reconciliation Commission that we met in Part IV of this textbook when we explore Social Network Analysis.

In order to prepare the dataframe, we can use the following code:

```
df = pd.read_feather("data/trc")
df = df.dropna()
df = df[["full_name", "long", "lat"]]
df["lat"] = pd.to_numeric(df["lat"], downcast="float")
df["long"] = pd.to_numeric(df["long"], downcast="float")
df.columns = ["full_name", "lon", "lat"]
```

18.1.3.1 Creating Maps with `st.map()`

Once the data is prepared properly, we can then graph it with the standard Streamlit widget `st.map()` with a single line of code:

```
st.map(df)
```

The output will look like Figure 18.6.

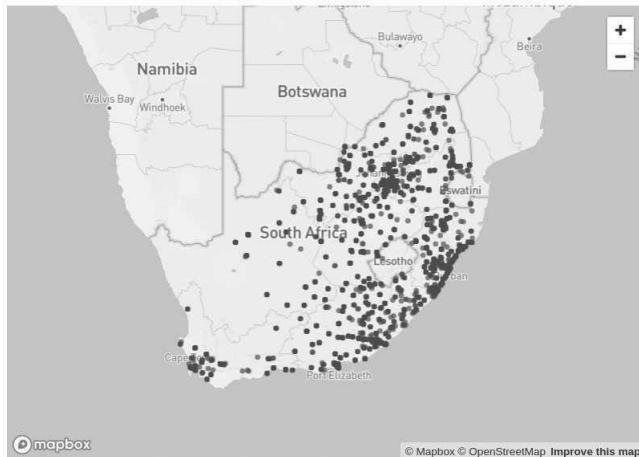


FIGURE 18.6

Example of `st.map()` Output.

Each node on this graph is a row in the dataframe. This is an interactive map that users can zoom in to each node on the graph. While this is useful for users to get a sense of the geospatial data quickly, the Streamlit `st.map()` widget is limited in what it can do.

18.1.3.2 Third-Party Maps – An Example with PyDeck

For more advanced mapping features, you will need to rely on third-party libraries. Fortunately, Streamlit has wrappers pre-designed so that you can leverage the power and versatility of these other libraries all within your application Python file.

For our purposes, we will use Streamlit's built-in PyDeck wrapper with the `st.pydeck_chart()` widget. The code below will create a similar graph, but note that because we are creating a PyDeck map, rather than a standard Streamlit map, we can leverage the full power of the PyDeck library, including giving tooltips that pop out for

each node, the radius of our nodes, the degree to which they come off the map in three dimensional space, the pitch of the map, and the default zoom.

```
st.pydeck_chart(pdk.Deck(
    map_style=None,
    initial_view_state=pdk.ViewState(
        latitude=-25.97,
        longitude=30.50,
        zoom=5,
        pitch=0,
    ),
    layers=[
        pdk.Layer(
            "ScatterplotLayer",
            df,
            pickable=True,
            opacity=0.8,
            stroked=True,
            filled=False,
            radius_scale=6,
            radius_min_pixels=1,
            radius_max_pixels=1000,
            line_width_min_pixels=5,
            get_position=[lon, lat],
            get_radius="radius",
            get_fill_color=[255, 140, 0],
            get_line_color=[255, 140, 0],
        ),
    ],
))
```

The output will look like Figure 18.7.

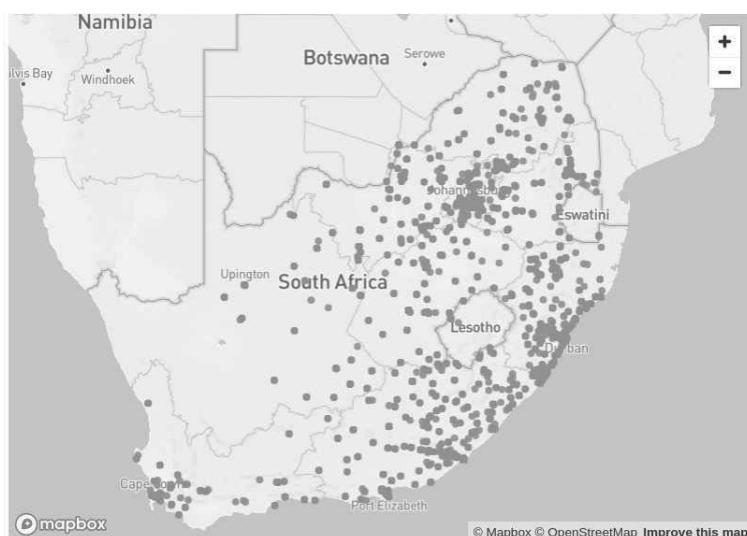


FIGURE 18.7
Example of Streamlit PyDeck Map.

18.2 Layout Design

Controlling the layout of an application is essential from both an aesthetic and programmatic perspective. In Streamlit, we have a number of ways to control our app's layout. Streamlit offers six different ways of controlling the layout of your application through widgets. Further customization is also possible by controlling the page configuration of your application as well as custom HTML. Here, we will focus on the first two; we will treat custom HTML later in this section.

18.2.1 Layout Widgets

18.2.1.1 Sidebar

The sidebar is one of the more useful layout features in Streamlit. If you are creating a multi-page app, you will have a sidebar by default. If, however, you are not using a multi-page app, you simply need to do something within the `st.sidebar()` widget and the sidebar will automatically appear for you. We can access the sidebar widget at any place in our Python file by writing `st.sidebar.[widget]()`.

If we were to use the following code in our Python file, for example, we would immediately have a sidebar with a header entitled `Sidebar Header`.

```
st.sidebar.header("Sidebar Header")
```

The output will look like Figure 18.8.

Notice on the left-hand side of the application, we have our header appear. We can populate any widget we desire into the sidebar by calling `st.sidebar` and then following this up with `.[widget]`.

18.2.1.2 Columns

We can also control the horizontal layout of our application with Streamlit's `st.columns()` widget. This will take a single mandatory argument, an integer that corresponds to the number of columns you wish to create. This will return a list of column widgets that you can write to. As with the sidebar, to write to the column widget, you will use its variable name proceeded by `.[widget] (write, header, etc.)`.

```
st.header("Columns")
cols = st.columns(2)
cols[0].write("Column 1")
cols[1].write("Column 2")
```

The output will look like Figure 18.9.

18.2.1.3 Expander

In some cases, we will want to allow a user to view a large amount of data, but this will detract from the application and put some of the main features very low on the main page of the app. In these, cases we want a user to be able to expand certain aspects of the application when they wish and compress them when they are finished using them. Streamlit comes built with this feature in the `st.expander()` widget.

Once we create an expander in our application, we can then insert any other Streamlit we wish inside of it, just like columns and the sidebar. In the code below, we are creating a simple expander and then writing into it.

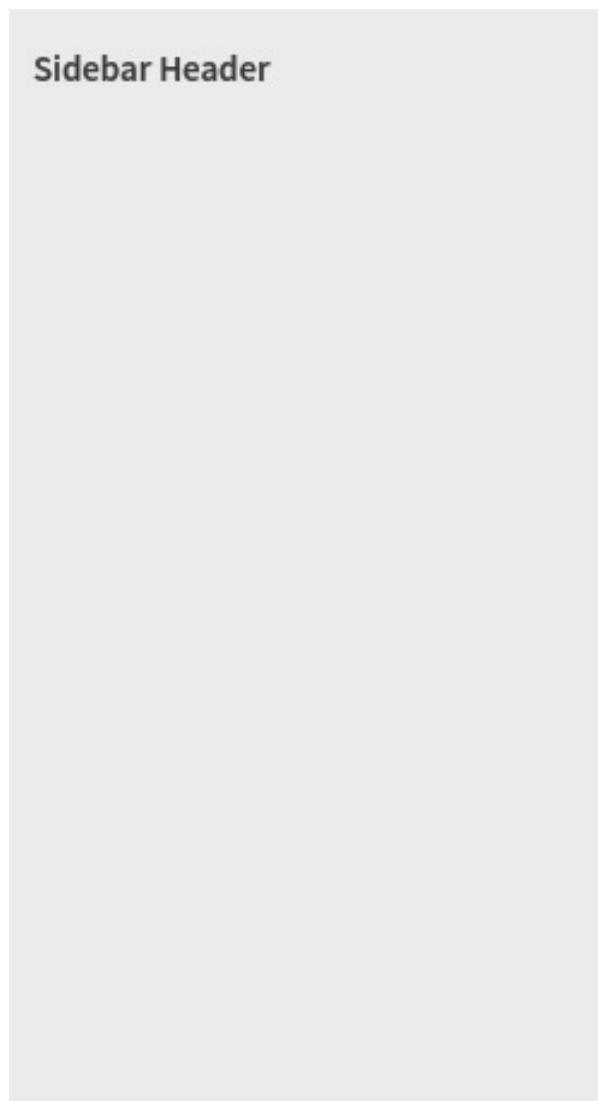


FIGURE 18.8
Example of a Streamlit Sidebar.

Columns

Column 1 Column 2

FIGURE 18.9
Example of Streamlit Columns.

```
expander = st.expander("This is an Expander")
expander.write("This is some text in an expander...")
```

The output will look like Figure 18.10.

Expander



FIGURE 18.10

Example off Streamlit Expander.

Notice how we see a new widget on our page. In the top-right corner, you can press the carrot and the expander will reveal the contents.

Expander



FIGURE 18.11

Example of Opened Expander.

18.2.1.4 Container

We also have the ability to create containers in our application with the `st.container()` widget. Remember, Streamlit widgets are populated in the order that Streamlit reads your Python file. This means that if your user performs an action and you want to display that data later in your file, then that data will populate at the bottom of your application. This is not ideal. In these scenarios, you need a way to populate a result higher in your application's page. You can do this with the container. The container will sit in that precise position and it can then be populated at a later time.

```
st.header("Container")
container = st.container()
container.write("This is some text inside a container...")
```

The output will look like Figure 18.12.

18.2.1.5 Tabs

In Streamlit, we can also make tabs so that a single page can host several different pages within it. This is useful if you are working with multiple datasets or collections of items that need to be displayed individually, but there is not enough space in your app to display all data sequentially. We create tabs in Streamlit via the `st.tabs()` widget. This will take a single argument which will be a list of tab names.

Container

This is some text inside a container...

FIGURE 18.12
Example of a Streamlit Container.

In the sample code below, we create the tabs on line 1 and then iterate over them to populate each tab with a unique message via `st.write()`.

```
tabs = st.tabs(["Tab 1", "Tab 2"])
for i, tab in enumerate(tabs):
    tabs[i].write(f"Tab {i+1}")
```

The output will look like Figure 18.13.

Tabs



FIGURE 18.13
Example of Streamlit Tabs.

18.2.1.6 Empty

The final layout widget in Streamlit is `st.empty()`. This takes no arguments. It sits on the page, rather like the `st.container()` widget, but unlike `st.container()`, `st.empty()` will only display the most recent item sent to it. This is useful in rather niche scenarios when you want to display data individually. You can use the following code to iterate over a list of names. Note in the image below that only the last name on the list appears. This is because it was the final item in the list.

```
st.header("Empty")
empty = st.empty()
items = ["Tom", "Fred", "Stephanie"]
for item in items:
    empty.write(item)
```

The output will look like Figure 18.14.

18.3 Streamlit Cache and Session States

Being able to make more modular, custom, and dynamic applications means getting to know how to leverage advanced features in Streamlit when it comes to working with data stored in memory. In Streamlit, we have two ways to store data in memory, either by caching the data with `@st.cache_data` or with the `st.session_state`.

Empty

Stephanie

FIGURE 18.14

Example of Streamlit Empty.

18.3.1 Caching Data with `@st.cache_data`

When working with large data-driven projects, run time will become an issue with Streamlit. This is because Streamlit reruns the Python file each time something changes in the application. With large datasets, this means that each time a user does anything within your application, Streamlit will need to reload all the data. For this reason, it is essential to know how to store large datasets (or models) in cache so that Streamlit does not need to reload large memory-intensive data or models each item it reruns.

We can cache our data with an `@st.cache_data` above a function that loads the data. If we wanted to load our Titanic dataset and store it in memory, therefore, we would use the following code snippet:

```
@st.cache_data
def load_df():
    df = pd.read_csv("./data/titanic.csv")
    return df
```

This is precisely the code that we will walk through when we create our first application in Streamlit later in this part of the textbook.

18.4 Storing Data with `st.session_state`

Aside from storing large data with cache, we can also store previous states of data with the `st.session_state`. The Streamlit Session State gives greater flexibility to an application. It functions as a dictionary that stores data that remains the same during any given state of the app. This means that if your app is rerun by the user because they interacted with the application, then the variable stored in the session state would remain the same.

This is essential for more complex data-driven applications. Let's consider the simple example that we saw earlier in this chapter when we examined the `st.metric()` widget.

```
if "prev_word_count" not in st.session_state:
    st.session_state["prev_word_count"] = 5
text = st.text_area("Paste text here to get word count.", "This is some... default text.")
word_count = len(text.split())
change = word_count - st.session_state.prev_word_count
st.metric("Word Count", word_count, change)
st.session_state.prev_word_count = word_count
```

In this sample, we start off with a conditional:

```
if "prev_word_count" not in st.session_state:
```

This line looks to see if a variable name that we want to use is stored in our session state. If it is not stored there, then we want to create that new key. We do that with the following line:

```
st.session_state["prev_word_count"] = 5
```

Here we are setting the `prev_word_count` key to 5.

Next, we give the user the ability to input some text for which they wish to receive a word count. In order for the metric to know if the new metric is higher or lower than the previous one, we must store the previous text's total word count. To do this, we access the previous session state in the final line of the snippet below.

```
text = st.text_area("Paste text here to get word count.", "This is some...  
→default text.")  
word_count = len(text.split())  
change = word_count-st.session_state.prev_word_count
```

Once we have populated those results, we then can update the `st.session_state.previous_word_count` value to the new value. This allows us to always know the state of the previous word count, so that when we display the change value, we know precisely how much our metric has changed.

```
st.metric("Word Count", word_count, change)  
st.session_state.prev_word_count = word_count
```

18.5 Custom HTML

Adding custom HTML in Streamlit allows you to develop more customized applications that fit your need. While it is not always necessary, it is useful to understand how to embed HTML within an application.

Custom HTML can be added via `st.markdown()`. In order for your HTML to appear on the page, however, you must pass a keyword argument `unsafe_allow_html=True`. This allows the HTML to be rendered. Let's look at a basic example where we want to display text with a background color of yellow. We can do this by wrapping our text in an `a` tag in HTML and setting the style's background color to yellow.

```
html = """  
<a style='background:yellow'>This text has a yellow background</a>  
"""  
st.header("Without unsafe_allow_html=True")  
st.markdown(html)
```

If we do not set `unsafe_allow_html` to True, then our result will look like Figure 18.15.

If we do set it to True, then our result will look like Figure 18.16.

While this lets you design more robust apps, it does introduce certain security issues.

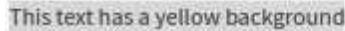
Without unsafe_allow_html=True

```
<a style='background:yellow'>This text has a yellow background</a>
```

FIGURE 18.15

Example of Custom HTML in Streamlit when Blocked.

With unsafe_allow_html=True



This text has a yellow background

FIGURE 18.16

Example of Custom HTML in Streamlit when Allowed.

18.6 Multi-Page Applications

Finally, Streamlit allows users to design applications that have multiple pages. In order to design a multi-page application, you must have a `pages` subfolder in your main directory. All Python files inside this directory will be considered to be pages for your application. If you wish to have your pages appear in a special order, you can do so by naming each page `01-page_name` and `02-page_name`. Streamlit will automatically remove the leading numbers and dash.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

19

Building a Database Query Application

Now that you know the basics of Streamlit and some of its more advanced features, then it comes time to put that knowledge to practice. In this chapter, we will design a Streamlit application from scratch. It will be built around the Titanic dataset that we used in the Pandas portion of this textbook. We will not only design this application, we will also put it in the cloud so that users can access it.

In Section 19.1 of this chapter we will walk through the basics of designing the application in Python. In Section 19.2, we will then focus on getting our application running in the cloud. The purpose of this chapter is to give you hands-on experience with designing an application as well as provide a template for you to use in a future project that requires querying a Pandas dataframe in Streamlit.

19.1 Building a Database Query Application

Throughout this section, we will be developing an application in Streamlit that looks like Figure 19.1.

We will be working with the following code:

```
import streamlit as st
import pandas as pd

# Cache our data
@st.cache_resource()
def load_df():
    df = pd.read_csv("./data/titanic.csv")
    survival_options = df.Survived.unique()
    p_class_options = df.Pclass.unique()
    sex_options = df.Sex.unique()
    embark_options = df.Embarked.unique()

    min_fare = df.Fare.min()
    max_fare = df.Fare.max()

    min_age = df.Age.min()
    max_age = df.Age.max()

    return df, survival_options, p_class_options, sex_options, embark_options, min_fare, max_fare, min_age, max_age

def check_rows(column, options):
    return res.loc[res[column].isin(options)]
```

(continues on next page)

Demo DataFrame Query App

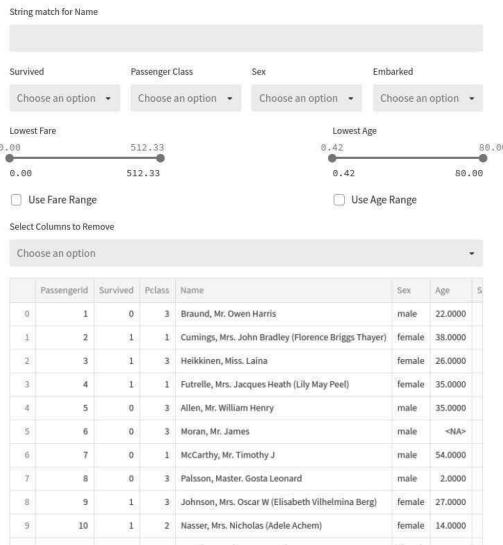


FIGURE 19.1
Example of the Demo Application.

(continued from previous page)

```
st.title("Demo DataFrame Query App")

df, survival_options, p_class_options, sex_options, embark_options, min_fare, max_fare, min_age, max_age = load_df()
res = df

name_query = st.text_input("String match for Name")

cols = st.columns(4)
survival = cols[0].multiselect("Survived", survival_options)
p_class = cols[1].multiselect("Passenger Class", p_class_options)
sex = cols[2].multiselect("Sex", sex_options)
embark = cols[3].multiselect("Embarked", embark_options)

range_cols = st.columns(3)
min_fare_range, max_fare_range = range_cols[0].slider("Lowest Fare", float(min_fare), float(max_fare),
                                                       [float(min_fare), float(max_fare)])
min_age_range, max_age_range = range_cols[2].slider("Lowest Age", float(min_age), float(max_age),
                                                       [float(min_age), float(max_age)])

if name_query != "":
    res = res.loc[res.Name.str.contains(name_query)]

if survival:
    res = check_rows("Survived", survival)

if p_class:
    res = check_rows("Pclass", p_class)

if sex:
    res = check_rows("Sex", sex)

if embark:
    res = check_rows("Embarked", embark)

if min_fare_range & max_fare_range:
    res = res[(res.Fare >= min_fare_range) & (res.Fare <= max_fare_range)]

if min_age_range & max_age_range:
    res = res[(res.Age >= min_age_range) & (res.Age <= max_age_range)]
```

(continues on next page)

(continued from previous page)

```

if p_class:
    res = check_rows("Pclass", p_class)
if sex:
    res = check_rows("Sex", sex)
if embark:
    res = check_rows("Embarked", embark)
if range_cols[0].checkbox("Use Fare Range"):
    res = res.loc[(res.Fare > min_fare_range) & (res.Age < max_fare_range)]
if range_cols[2].checkbox("Use Age Range"):
    res = res.loc[(res.Age > min_age_range) & (res.Age < max_age_range)]
removal_columns = st.multiselect("Select Columns to Remove", df.columns.tolist())
for column in removal_columns:
    res = res.drop(column, axis=1)
st.write(res)

```

By the end of this chapter, you will be able to understand and parse each line of this code.

19.1.1 Importing the Libraries

At the start of our Python file, we first import the required libraries. We are using Streamlit for the app development and Pandas for working with our data.

```

import streamlit as st
import pandas as pd

```

19.1.2 Caching Data

Let's examine the next section of code.

```

@st.cache_data
def load_df():
    df = pd.read_csv("./data/titanic.csv")
    survival_options = df.Survived.unique()
    p_class_options = df.Pclass.unique()
    sex_options = df.Sex.unique()
    embark_options = df.Embarked.unique()

    min_fare = df.Fare.min()
    max_fare = df.Fare.max()

    min_age = df.Age.min()
    max_age = df.Age.max()

    return df, survival_options, p_class_options, sex_options, embark_options, min_fare, max_fare, min_age, max_age

```

The first line is:

```
@st.cache_data
```

This decorator followed by `st.cache_data` establishes that resulting objects from the function that proceeds it should be cached into memory.

The next line begins the creation of our function.

```
def load_df():
```

Once we have defined our function we begin working with our data. First, we load the data:

```
df = pd.read_csv("./data/titanic.csv")
```

Next, our application will leverage four `st.multiselect()` widgets for four different columns in our dataframe. In order to populate a list of options for users to select, we need to know the unique values of each column. We can grab each unique value with `.unique()` on each column.

```
survival_options = df.Survived.unique()
p_class_options = df.Pclass.unique()
sex_options = df.Sex.unique()
embark_options = df.Embarked.unique()
```

Next, our application will also leverage two sliders: one for `Age` and one for `Fare`. These will allow a user to find results based on a person's age or the fare of their ticket. We need to know the max value and the minimum value for each of these so that we can automatically set the slider minimum and maximum values.

```
min_fare = df.Fare.min()
max_fare = df.Fare.max()

min_age = df.Age.min()
max_age = df.Age.max()
```

Finally, we return all of these values so that when the function is called each of these will be returned.

```
return df, survival_options, p_class_options, sex_options, embark_
options, min_fare, max_fare, min_age, max_age
```

Once we have created this function, we can call it and create all the objects that we need with the following code:

```
df, survival_options, p_class_options, sex_options, embark_options, min_
fare, max_fare, min_age, max_age = load_df()
```

We will also create another object whose variable name will be `res`. This will be the dataframe that gets manipulated by the user and populates the results in the app.

```
res = df
```

19.1.3 Creating Our App Layout

After preparing all the data, now comes the time to design the general layout of our application. We use the following code:

```
name_query = st.text_input("String match for Name")

cols = st.columns(4)
survival = cols[0].multiselect("Survived", survival_options)
p_class = cols[1].multiselect("Passenger Class", p_class_options)
sex = cols[2].multiselect("Sex", sex_options)
embark = cols[3].multiselect("Embarked", embark_options)
```

(continues on next page)

(continued from previous page)

```
range_cols = st.columns(3)
min_fare_range, max_fare_range = range_cols[0].slider("Lowest Fare", float(min_fare), float(max_fare),
                                                       [float(min_fare), float(max_fare)])
min_age_range, max_age_range = range_cols[2].slider("Lowest Age", float(min_age), float(max_age),
                                                       [float(min_age), float(max_age)])
```

Let's break down this section of the code. In the first line, we create an object `name_query`. This will be a string that is returned from a `st.text_input()` widget. We will use this input to query the `Name` field in the dataframe.

```
name_query = st.text_input("String match for Name")
```

Next, we will create four columns that we can populate with our `st.multiselect()` widgets.

```
cols = st.columns(4)
```

Now that we have our columns, we can create our four `st.multiselect()` widgets. The user will be able to select which items that want to see returned for each field in the dataframe. Each of these will return a list of options. We will gather data for four fields: `Survived`, `Pclass`, `Sex`, and `Embarked`.

```
survival = cols[0].multiselect("Survived", survival_options)
p_class = cols[1].multiselect("Passenger Class", p_class_options)
sex = cols[2].multiselect("Sex", sex_options)
embark = cols[3].multiselect("Embarked", embark_options)
```

Next, we need three new columns so that we can populate our two range sliders. We are using three columns here so that there is a large gap between the two sliders.

```
range_cols = st.columns(3)
```

We will populate the first and last slider, we will place two `st.slider()` widgets. We will use the minimum and maximum values for the `Age` and `Fare` fields in the dataframe.

```
min_fare_range, max_fare_range = range_cols[0].slider("Lowest Fare", float(min_fare), float(max_fare),
                                                       [float(min_fare), float(max_fare)])
min_age_range, max_age_range = range_cols[2].slider("Lowest Age", float(min_age), float(max_age),
                                                       [float(min_age), float(max_age)])
```

19.1.4 Using User Inputs to Produce a New DataFrame

With the general layout designed, we can then work with the input from the user to modify our `res` dataframe. The following code manipulates the dataframe through a set of conditions we generate from the user input.

```
if name_query != "":
    res = res.loc[res.Name.str.contains(name_query)]
if survival:
    res = check_rows("Survived", survival)
if p_class:
```

(continues on next page)

(continued from previous page)

```

res = check_rows("Pclass", p_class)
if sex:
    res = check_rows("Sex", sex)
if embark:
    res = check_rows("Embarked", embark)
if range_cols[0].checkbox("Use Fare Range"):
    res = res.loc[(res.Fare > min_fare_range) & (res.Age < max_fare_range)]
if range_cols[2].checkbox("Use Age Range"):
    res = res.loc[(res.Age > min_age_range) & (res.Age < max_age_range)]
removal_columns = st.multiselect("Select Columns to Remove", df.columns.tolist())
for column in removal_columns:
    res = res.drop(column, axis=1)
st.write(res)

```

First, we check to see if the user has written anything in the `st.text_input()` widget. If it is, then we will narrow the dataframe down to anything that matches the user's string.

```

if name_query != "":
    res = res.loc[res.Name.str.contains(name_query)]

```

For each of the `st.multiselect()` widget inputs, we check to see if the user has selected how to narrow the dataframe:

```

if survival:
    res = check_rows("Survived", survival)
if p_class:
    res = check_rows("Pclass", p_class)
if sex:
    res = check_rows("Sex", sex)
if embark:
    res = check_rows("Embarked", embark)

```

Next, we will use the input from the minimum and maximum values of the sliders for Age and Fare. So that we do not ignore results where Age or Fare are NaN in the dataset, we want to give the user the ability to check an `st.checkbox()` widget. This will allow the user to activate or deactivate the sliders.

```

if range_cols[0].checkbox("Use Fare Range"):
    res = res.loc[(res.Fare > min_fare_range) & (res.Age < max_fare_range)]
if range_cols[2].checkbox("Use Age Range"):
    res = res.loc[(res.Age > min_age_range) & (res.Age < max_age_range)]

```

We also want to give the user one final input, the ability to narrow down and delete columns from the dataset. The reason for this is because not all fields will be relevant to every user and since this is a large dataset, it makes sense to give them the ability to limit which fields they are seeing to the ones they want.

```

removal_columns = st.multiselect("Select Columns to Remove", df.columns.tolist())
for column in removal_columns:
    res = res.drop(column, axis=1)

```

As we have manipulated the `res` dataframe throughout each of these lines, we are finally ready to display the data:

```
st.write(res)
```

19.2 Deploying an App in the Cloud with Streamlit Share

Once you have designed an application and have tested it locally, it comes time to share it with others. We can do this with Streamlit by leveraging several different cloud-based services. Fortunately, Streamlit offers a free Streamlit Share service that lets users share their apps for free.

19.2.1 Create a GitHub Account

In order to deploy your application, you will need a place to store your application's source code. The easiest way to do this is via GitHub which Streamlit and many other application hosting services support.

19.2.2 Upload Application to GitHub

Once you have created your GitHub account, you can create a repository and upload the code into the repository. If you are just starting out with GitHub, getting used to Git can be a bit daunting. Git is powerful, but has a steep learning curve. It allows you to maintain your code, especially in teams, and version control everything. This means that your code is always backed up and you can access previous versions of your code at any single time.

To upload your application formally, you would want to be familiar with Git and how to perform basic tasks, such as cloning a repository, adding files to it, committing those changes, and then pushing them to the repository. While this is the better approach, it is not the only one. For now, you can get your application up and running entirely in your browser.

First, we will create a new repository. To do this, we will go to our GitHub main page and select `New`, found in the top-left side of your screen.

Once you click `New`, you will see the screen in Figure 19.3.

Here, we will fill out the details of our repository. We need to first give it a unique name. We can keep all other settings the default for now. Once done, we will click `Create repository`. After this, you will see a new screen that looks like Figure 19.4.

You will now select `creating a new file`. Once you do, you will see a screen that looks like Figure 19.5.

Here, you will create a file called `README.md` and assign some text to it. This will ideally describe your application. It will be what users see when they visit your GitHub repository. Once you have a description you are happy with, click `Commit new file`. This will lead to a final new screen.

Next, we need upload our application and data to our repository. To do this, we will click `Add file` and upload our local files that are necessary for running our Streamlit application. You will also want to make a `requirements.txt` file lists all required libraries, such as Pandas.

19.2.3 Connect Streamlit Share to your GitHub

Now that we have our repository created, we need to link Streamlit Share to our GitHub. To do so, you will need to visit <https://share.streamlit.io/>. Once on this page, you will see a screen that looks like Figure 19.6.

Select Continue with GitHub and use your GitHub credentials for login verification. Once you are logged in, you will see a screen that looks like this at the top:



19.2.4 Create a New App

Select New App. After a few seconds, you will be taken to a page that looks like this:

Click on each of these fields. First, select your repository. Streamlit will be connected to your GitHub, so you will be able to see all your repositories, both private and public. Next, select the branch of your repository. We have not covered branches in this textbook. Branches are a function of Git. By default, your application will be on the main branch of your repository. Finally, select the main Python file in which your application rests. Once you have filled out these three fields, click Deploy!.

19.2.5 Set Custom Subdomain

It will take several minutes (depending on the complexity of your application and the number of required libraries you wish to have installed). At this point, Streamlit Share is building an environment on a server to host your application in the cloud. Once complete, it will provide you with a unique domain for your application with the extension. This will

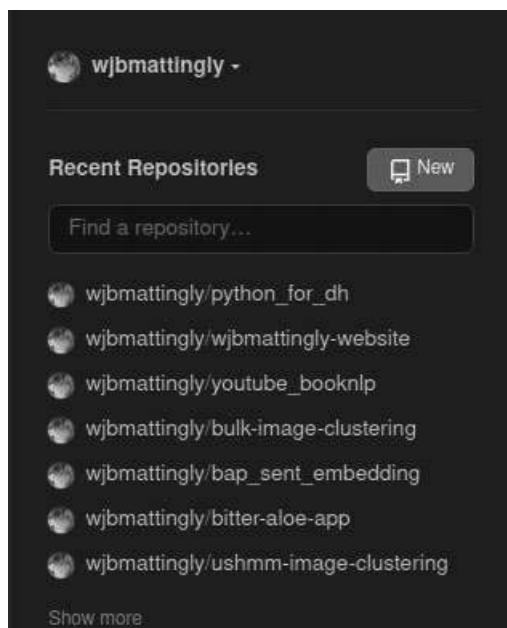


FIGURE 19.2

Creating a New GitHub Repository by Selecting New.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * Repository name *

wjbmattingly /

Great repository names are short and memorable. Need inspiration? How about fuzzy-octo-couscous?

Description (optional)

Public Anyone on the internet can see this repository. You choose who can commit.
 Private You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file This is where you can write a long description for your project. Learn more.

Add .gitignore

Choose which files not to track from a list of templates. Learn more.

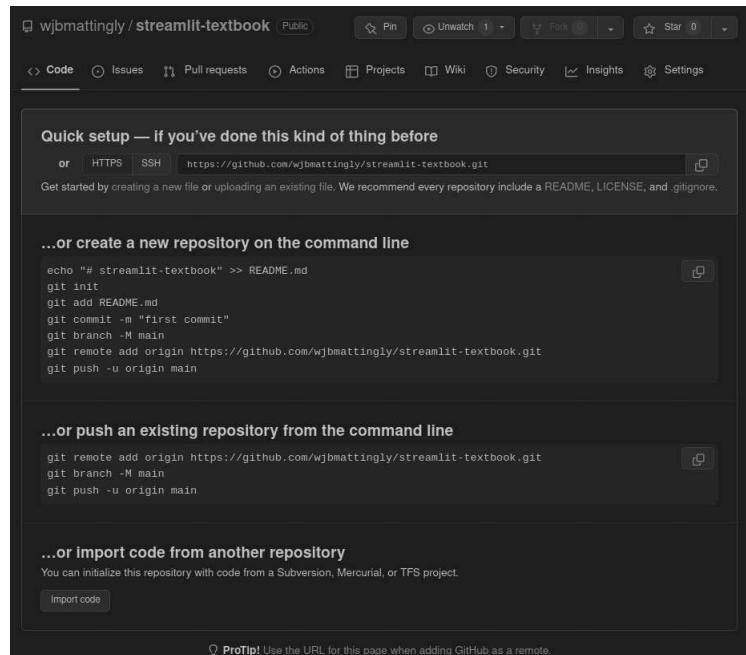
Choose a license

A license tells others what they can and can't do with your code. Learn more.

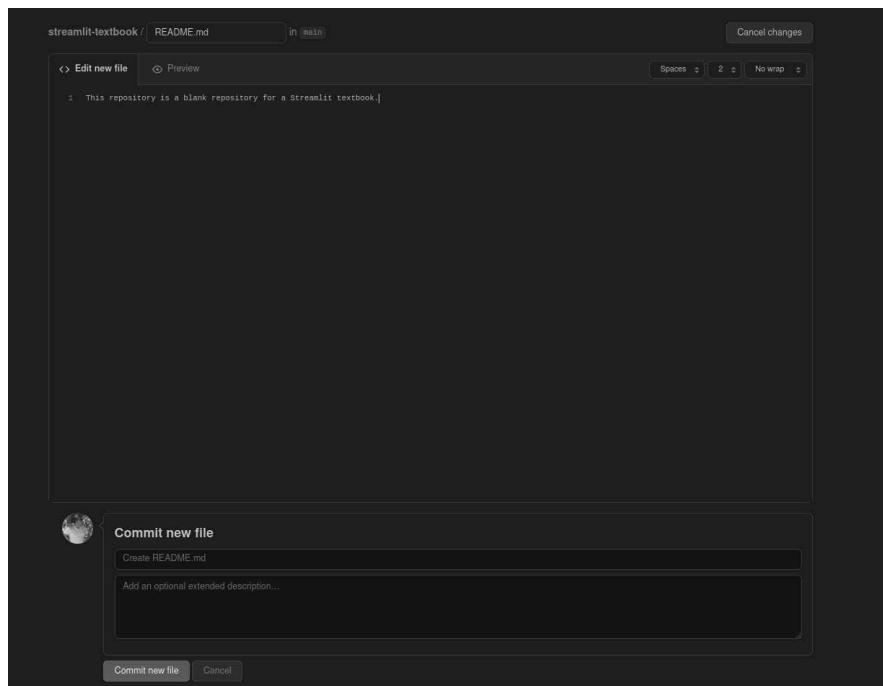
ⓘ You are creating a public repository in your personal account.

FIGURE 19.3

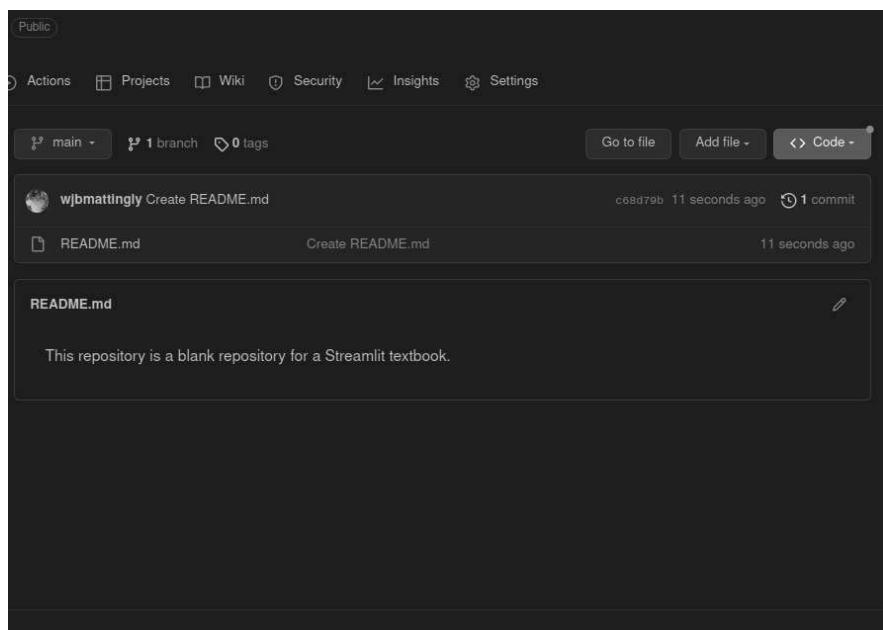
Example of GitHub Form for Creating New Repository.

**FIGURE 19.4**

Example of First Screen after Creating New Repository.

**FIGURE 19.5**

Example of Creating your First File and Commit in the New Repository.

**FIGURE 19.6**

Example of Adding Files to the New Repository.

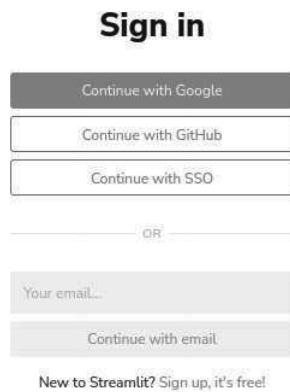


FIGURE 19.7
Connecting GitHub to Streamlit Share.

The image shows the 'Deploy an app' form. At the top left is a 'Back' button with a left arrow icon. The main title is 'Deploy an app'. There are three input fields: 'Repository' containing 'wjbmattingly/repo', 'Branch' containing 'master', and 'Main file path' containing 'streamlit_app.py'. To the right of the repository input is a 'Paste GitHub URL' link. At the bottom left is an 'Advanced settings...' link, and at the bottom right is a large 'Deploy!' button.

FIGURE 19.8
Form for Creating a New Application.

be based on your repository name and your GitHub username. We can create a custom subdomain for free by returning to our Streamlit Share homepage and selecting the three dots next to our application's name.

Once here, select *Settings* and you will see a page that looks like Figure 19.7.

Note that in the middle of the screen, you will see your long subdomain. You can now change it to something unique and easier to remember for distributing to users. After you change your subdomain, click *Save*.

Now, your app is in the cloud with a unique and easy-to-remember subdomain!



FIGURE 19.9
Select the Settings for your Application.

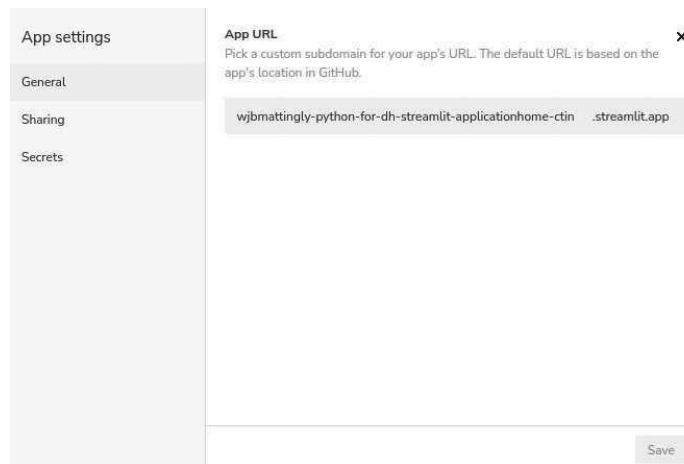


FIGURE 19.10
The Default URL for your Application.

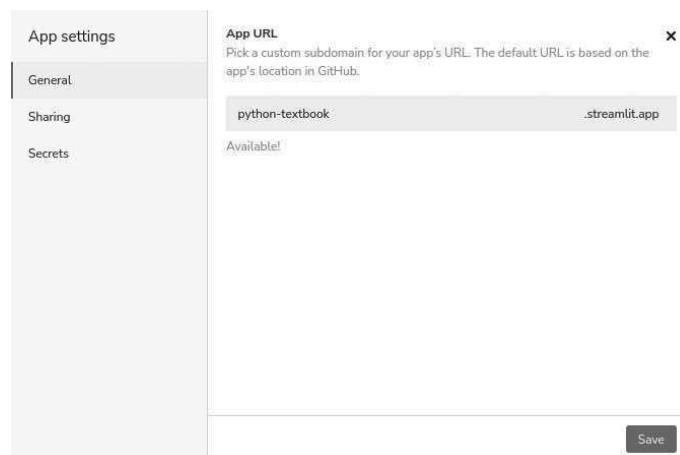


FIGURE 19.11
The New URL for your Application.

Part V

Conclusion



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

20

Conclusion

This textbook has introduced you to some of the key concepts and terminology associated with Python. It has also given you the tools necessary to begin applying Python to your own projects. While Part I sought to give you the basic coding skills necessary to begin programming in Python, the later parts of this textbook gave you the basis to begin applying Python to specific digital humanities problems, from data analysis with Pandas, to NLP with spaCy, topic modeling with Top2Vec, social network analysis with NetworkX and PyVis, and how to create applications in short time with Streamlit.

While this textbook was not able to cover all aspects of Python or even all the ways to apply it in the digital humanities, you should now have a resource to return to for code and exposition on key methods. This will not be the end of your programming journey, rather the beginning. From here, you can begin to learn on your own and expand your knowledge.



Taylor & Francis
Taylor & Francis Group
<http://taylorandfrancis.com>

Index

A

Advanced Pandas

- graphing network data with pandas
 - customize the graph, 139–140
 - data from pandas to NetworkX, 138–139
 - data graphing, 139
- plotting data with Pandas
 - bar and barh charts, 126–128
 - importing DataFrame, 125–126
 - pie charts, 128–130
 - scatter plots, 130–138
- time series data, 141
 - convert to time series datetime in Pandas, 149–153
 - data from float to int, 143–149
 - dataset, 141–143

Anaconda Navigator, 13–15

Append method, 33

AttributeError, 33

Attributes, 66

Automation, 5

B

Bar chart, 126–128

Barh charts, 126–128

BeautifulSoup, 55, 66–68, 71–73

BERT, 198, 240

Bigrams, 214–215

Binary, 30

Binder, 8, 9

Bitter Aloe Project, 141

Blank spaCy Model, 199

Bokeh, 269, 285

BookNLP

- book_id, 242

- booknlp.process(), 243

- with books and larger documents, 240

character analysis

- .book JSON data, 253
- character_data, 254
- function's code, 252–253
- printTop, 254
- verb usage, 254–255

creating pipeline, 241–242

description, 239

events analysis

- .events file, 261
- .tokens file, 256–257
- event columns, 257, 258
- event_lemma, 260
- event_word, 260
- isnull() method, 257
- sentence, 260
- sentence_ID column, 259–260
- words and lemmas, 258–259

event tagging, 240

importing, 241–242

input_file, 242

installation, 241

model key, 242

model_params, 242

on non-standard English, 240

out-of-vocabulary (OOV) words, 240

output_directory, 242

output files

- .book file, 248–251
- .book.html File, 252
- .entities file, 245–247
- .quotes file, 247
- .supersense file, 248
- .tokens file, 244–245

pipeline key, 242

quotation speaker identification, 240

referential gender inferencing, 240

Booleans, 30

Bugs, 21

Built-in types, 19–21

C

Capitalize method, 26
 Case sensitivity, 18
 Classes, 51
 adding functions to, 53–54
 creation, 52–53
 Close tag, 66
 Code learning, 3
 benefits, 5–6
 Coding basics
 bugs, 21
 built-in types, 19–21
 case sensitivity, 18
 objects, 17
 print function, 16–17
 reserved words, 18–19
 type function, 21
 variables, 17–18
 Comparison operators, 41
 Conditionals, 43
 elif functions, 44–45
 else statement, 44
 if statement, 43–44
 ‘in’ function, 44–45
 lists with ‘in’ and ‘not in,’ 45–46
 CSV data, 59
 CSV library, 77

D

Data, 23; *see also* Data structures
 booleans, 30
 float to int, 143–149
 graphing, 139
 numbers (integers and floats), 28
 as data, 29
 strings, 24–25
 strings as data, 25
 capitalize method, 26
 lower method, 26
 replace method, 26–27
 split method, 27–28
 upper method, 26
 Database query application
 app layout, 318–319
 caching data, 317–318
 demo application, 316
 deploying app with Streamlit share

connecting Streamlit Share to GitHub, 322, 324
 creating new app, 322
 custom subdomain creation, 322, 325, 326
 GitHub account creation, 321
 upload application to GitHub, 321, 323, 324
 developing application, 315–317
 importing libraries, 317
 new dataframe creation, 319–321
 DataFrame, 77
 add column to, 81–82
 cleaning
 convert DataFrame data types (Float to Int), 105–106
 drop Column in Pandas DataFrame, 101–103
 remove rows have NaN in any column, 104
 remove rows have NaN in specific column, 104–105
 create from dictionary, 78–79
 display, 79–80
 finding data
 dataset with describe(), 89
 find column data, 85–86
 grab specific range of rows with df.iloc[], 88
 query with “or” (|) on DataFrame, 93–95
 quick sense of dataset with df.head(), 86–87
 specific information in dataset with df.loc, 89–93
 Titanic dataset, 85
 grab specific row with iloc, 83
 iterating over with df.iterrows(), 84
 organization
 reverse sort data by single column, 97–98
 sort data by multiple columns, 98–100
 sort data by multiple columns with different values, 100–101
 sort data by single column, 95–97
 read from CSV, 80–81
 save to CSV, 80
 save to JSON, 81
 Data on web
 HTML, 65

attributes, 66
 diving into, 65–66
 parsing with BeautifulSoup, 66–68
 web pages scraping
 with BeautifulSoup, 71–73
 with requests, 70–71
 website’s HTML, 69–70

Data searching

- advanced searching on strings
 - features within string, 109–110
 - strings without feature, 110
 - using RegEx with Pandas, 110–112
- filter function, 113–116
- grouping, 121
 - with groupby() function, 121–122
 - with many subsets, 124
 - quantitative analysis with .count(), 122–123
 - quantitative analysis with .sum(), 122–123
 - working with multiple groups, 123–124
- query function, 116–120

Dataset, 141–143

Data structures, 23, 24, 30–31

- dictionaries, 34–35
- list, 31–32
- mutability *vs.* immutability, 33
- sets (bonus data structure), 33–34
- tuples, 32–33

Data types, 24

Debug, 25

Debugging, 21

def (defining function), 47

Dictionaries, 34–35

DiGraph, 266

Directory, 61–63

“Div” tags, 65, 67, 69, 70, 72

Django, 285

Docstring, 49

Domain knowledge, 3

E

Elif function, 44–45

Else statement, 44

EntityRuler spaCy

- adding to spaCy pipeline, 173
- analyze_pipes(), 175–176
- assigning after “ner” pipe, 176

complex rules, 177

demonstration of, 173–176

factory, 173

full documentation of, 173

model instructions, 174

purpose of, 173

variance, 177

Enumerate, 40–41

F

Filter function, 112–116

Final Report, 141

Flask, 285

Floats, 28, 29

For loop, 37–39

Functions, 47

- in action, 47–49
- docstring, 49
- keyword arbitrary arguments, 50–51
- keyword argument, 50
- with multiple arguments, 49–50

G

GitHub, 39

Glob, 60–61

Google Colab, 7–9

Graphing network data with pandas

- customize the graph, 139–140
- data from pandas to NetworkX, 138–139
- data graphing, 139

Graph theory, 263

Grouping, 121

- with groupby() function, 121–122
- with many subsets, 124
- quantitative analysis with .count(), 122–123
- quantitative analysis with .sum(), 122–123
- working with multiple groups, 123–124

H

Holocaust named entity recognition (NER) datasets

- concentration camps, 195
- normalizing data, 196–197
- rules-based EntityRuler, 196

- J**
- JavaScript Object Notation (JSON), 59, 80
 - JSON data, 59
 - reading data with `json.load()`, 60
 - writing data with `json.dump()`, 59–60
 - JupyterBook, 9–11
 - JupyterLab, 14, 16
 - JupyterNotebook, 54, 78
 - Jupyter Notebooks online, 11–13
- K**
- Keyword arbitrary arguments, 50–51
 - Keyword argument, 50
- L**
- Large digital projects, 3
 - Latent Dirichlet Allocation (LDA) topic modeling
 - bag-of-words (BOW) dictionary, 216
 - label assigning, 218
 - lemmatization, 216
 - limitations, 218
 - punctuation removal, 216
 - in Python
 - cleaning documents, 220
 - corpus, 222
 - Gensim library, 218
 - `get_document_topics()`, 222
 - `get_topic_terms()` method, 223
 - ID-Word Index creation, 220–222
 - id2word, 222
 - LdaModel class, 219
 - libraries and data importing, 218–219
 - Natural Language Toolkit (NLTK)
 - installation, 218
 - `num_topics`, 222
 - PyLDAVis library, 218, 224
 - stopwords removal, 216
 - trial-and-error passes, 217
 - word order, 217
 - Latin, 5
 - LDA topic modeling, *see* Latent Dirichlet Allocation (LDA) topic modeling
 - Libraries, 54
 - import a library, 55
 - installation, 54–55
 - List, 31–32
- I**
- If statement, 43–44
 - Immutable objects, 27
 - Indexing, 31
 - dictionaries, 34–35
 - a list, 31–32
 - a list with and without enumerate, 40–41
 - Index location, 83
 - Integers, 28, 29

- comprehension, 39–40
 - conditionals with ‘in’ and ‘not in’, 45–46
 - convert a column to, 83
 - indexing with and without enumerate, 40–41
 - Loops, 37
 - indexing list with and without enumerate, 40–41
 - list comprehension, 39–40
 - for loops, 37–39
 - operators, 41–42
 - while loops, 42–43
 - Lower method, 26
- ## M
- Machine learning-based approach, 212
 - Matcher spaCy
 - .add method, 178
 - adding in sequences, 182
 - basic pattern creation, 178
 - Doc container, 178
 - end token, 179
 - first_match object, 179
 - grabbing all proper nouns, 181
 - greedy keyword argument, 181–182
 - lexeme, 179
 - LIKE_EMAIL attribute, 178
 - Matcher class, 178
 - multi-word tokens, 181
 - nlp vocab, 178
 - rules-based pattern matchers, 178
 - start token, 179
 - token attributes, 179–180
 - Mathematical operations, 29
 - Methods, 51
 - Minimum viable product (MVP), 4
 - Multiple argument, 49–50
 - Multiple nested directories, 61
 - Mutability *vs.* immutability, 33
 - Mutable objects, 27
 - MVP, *see* Minimum viable product
- ## N
- NaN
 - cells, 150
 - remove rows in any column, 104
 - remove rows in specific column, 104–105
 - NetworkX, 138–139
- add_edges_from() method, 265–266
 - adding color to nodes, 271–273
 - asymmetrical networks, 266–267
 - connections calculation, 268
 - degree of focus, 268
 - destination node, 265
 - DiGraph, 266
 - draw_networkx() function, 265
 - limitations, 268–269
 - Matplotlib, 264
 - NetworkX Graph class, 265
 - NetworkX nx.Graph(), 270
 - node centrality calculation, 268
 - node degree value calculation, 268
 - nx.degree_centrality(), 268
 - plt.show(), 265
 - shortest distance calculation, 267–268
 - source node, 265
- New Notebook, 10, 11
 - Numbers (integers and floats), 28
 - as data, 29
- ## O
- Objects, 17
 - OCR, *see* Optical Character Recognition
 - Open-source, 54
 - Operators, 41–42
 - Optical Character Recognition (OCR), 5
- ## P
- Pandas, 7, 55, 77
 - add column to DataFrame, 81–82
 - cleaning DataFrame
 - convert DataFrame data types (Float to Int), 105–106
 - drop Column in Pandas DataFrame, 101–103
 - remove rows have NaN in any column, 104
 - remove rows have NaN in specific column, 104–105
 - convert column to list, 83
 - create DataFrame from dictionary, 78–79
 - display a DataFrame, 79–80
 - finding data in DataFrame
 - dataset with describe(), 89
 - find column data, 85–86

- grab specific range of rows with
 `df.iloc[]`, 88
- query with “or” (`|`) on DataFrame,
 93–95
- quick sense of dataset with `df.head()`,
 86–87
- specific information in dataset with
 `df.loc`, 89–93
- Titanic dataset, 85
- grab specific column, 82–83
- grab specific row of DataFrame with `iloc`,
 83
- import Pandas, 78
- installation, 78
- isolating unique values in column, 83
- iterating over DataFrame with
 `df.iterrows()`, 84
- organizing DataFrame
 - reverse sort data by single column,
 97–98
 - sort data by multiple columns, 98–100
 - sort data by multiple columns with
 different values, 100–101
 - sort data by single column, 95–97
- read DataFrame from CSV, 80–81
- save DataFrame to CSV, 80
- save DataFrame to JSON, 81
 - using RegEx with, 110–112
- Parse, 66
- Pclass, 90, 94, 96, 98, 116, 119
- PhraseMatcher
 - custom attribute, 184–185
 - on_match patterns, 185–186
 - PhraseMatcher class, 183
- PI, *see* Principal investigator
- Pie charts, 128–130
- Pip, 54, 55
- Pipe, 94
- Plotting data with Pandas
 - bar and barh charts, 126–128
 - importing DataFrame, 125–126
 - pie charts, 128–130
 - scatter plots, 130–138
- Positional arguments, 49, 50
- Principal investigator (PI), 3, 4
- Print function, 16–17
- Programming languages, 6
- Pseudo-code, 37, 46
- PyDeck, 305–306
- Python, 6; *see also* individual entries
- coding basics
 - bugs, 21
 - built-in types, 19–21
 - case sensitivity, 18
 - objects, 17
 - print function, 16–17
 - reserved words, 18–19
 - type function, 21
 - variables, 17–18
- installation, 7
 - Binder from JupyterBook, 9–11
 - Google Colab, 7–9
 - Jupyter Notebooks online, 11–13
 - local installation, 13–15
 - Trinket, 7
- Python 2, 6
- Python 3, 6
- Python 3 Kernel, 10
- Python script, 17, 25, 58
- PyVis
 - adding color to nodes, 271–273
 - basics of, 269–270
 - Bokeh, 269
 - installation, 269
 - PyVis Network class, 271
- Q**
- Query function, 112, 116–120
- R**
- Regular Expressions (RegEx), 25
 - finding and retrieving patterns, 186
 - matching RegEx, 200–202
 - multi-word token entities
 - multi-word tokens extraction, 191
 - priority to longer spans, 193–194
 - reconstruct spans, 192
 - Span to original_ents, 192–193
 - in Python, 186–188
 - replacing matching patterns, 186
 - search engines, 186
 - in spaCy, 188–191
 - strengths of, 186
 - weaknesses of, 186
- Replace method, 26–27
- Requests, 55
- Requests library, 70–71
- Reserved words, 18–19

Root directory, 62, 63
Rules-based methods, 212

S

Scatter plots, 130–138
Scraping websites, 66
Self-reliant digital humanist, 4–5
Self-reliant humanist, 3
Selenium, 66
Sets (bonus data structure), 33–34
Single argument, 49
Social network analysis (SNA)
 on humanities data
 data examination, 273–275
 edge_list, 275
 Filter Menu, 281, 282
 found_orgs, 275
 get_adj_list(), 278
 Graph class (G), 277
 nodes and edges, 274–276
 PyVis graph, 277–279
 Select Menu, 279, 280
 spring_layout(), 277
 NetworkX, 276, 277
 add_edges_from() method, 265–266
 adding color to nodes, 271–273
 asymmetrical networks, 266–267
 connections calculation, 268
 degree of focus, 268
 destination node, 265
 DiGraph, 266
 draw_networkx() function, 265
 limitations, 268–269
 Matplotlib, 264
 NetworkX Graph class, 265
 NetworkX nx.Graph(), 270
 node centrality calculation, 268
 node degree value calculation, 268
 nx.degree_centrality(), 268
 plt.show(), 265
 shortest distance calculation, 267–268
 source node, 265
 PyVis
 adding color to nodes, 271–273
 basics of, 269–270
 Bokeh, 269
 installation, 269
 PyVis Network class, 271
spaCy, 239

EntityRuler
 adding to spaCy pipeline, 173
 analyze_pipes(), 175–176
 assigning after “ner” pipe, 176
 complex rules, 177
 demonstration of, 173–176
 factory, 173
 full documentation of, 173
 model instructions, 174
 purpose of, 173
 variance, 177
GPE label, 174
Matcher
 adding in sequences, 182
 .add method, 178
 basic pattern creation, 178
 Doc container, 178
 end token, 179
 first_match object, 179
 grabbing all proper nouns, 181
 greedy keyword argument, 181–182
 lexeme, 179
 LIKE_EMAIL attribute, 178
 Matcher class, 178
 multi-word tokens, 181
 nlp.vocab, 178
 rules-based pattern matchers, 178
 start token, 179
 token attributes, 179–180
PhraseMatcher
 custom attribute, 184–185
 on_match patterns, 185–186
 PhraseMatcher class, 183
Regular Expressions (RegEx)
 finding and retrieving patterns, 186
 implementation, 188–191
 multi-word token entities, 191–194
 replacing matching patterns, 186
 search engines, 186
 strengths of, 186
 weaknesses of, 186
Split method, 27–28
StackOverflow, 6, 25, 40
Streamlit
 basic graphs plotting
 area charts, 303, 304
 bar charts, 303, 304
 line charts, 303
 st.line_chart(), 302
 cache, 310–311

- customHTML, 312–313
 - data storage, 311–312
 - data structures displaying
 - st.json(), 289, 290
 - st.write(), 289, 290
 - database query application (*see* Database query application)
 - home page creation, 286–287
 - input widgets
 - Boolean inputs, 296–298
 - cheatsheet, 293
 - date and time input, 295–296
 - numerical input, 294–295
 - selection widgets, 298–299
 - text input, 293–294
 - installation, 286
 - layout design
 - columns, 307, 308
 - container, 309, 310
 - empty, 310, 311
 - expander, 307, 309
 - sidebar, 307
 - tabs, 309–310
 - map charts
 - creating maps with st.map(), 305
 - PyDeck, 305–306
 - st.map() widget, 304
 - metrics, 301–302
 - multi-page applications, 313
 - multimedia displaying, 291–292
 - session states, 310–311
 - tabular data displaying
 - st.dataframe(), 290–291
 - st.markdown(), 291
 - st.table(), 291
 - st.write(), 290
 - text displaying to users
 - st.caption(), 288
 - st.header(), 288
 - st.markdown, 289
 - st.subheader, 288
 - st.title(), 288
 - st.write(), 288
 - Strings, 24–25
 - advanced searching
 - features within string, 109–110
 - strings without feature, 110
 - using RegEx with Pandas, 110–112
 - as data, 25
 - capitalize method, 26
 - lower method, 26
 - replace method, 26–27
 - split method, 27–28
 - upper method, 26
 - Supervised learning, 212
 - SyntaxError, 25
- T**
- Tabular data, 77
 - Tag, 65
 - Technical lead, 3, 4
 - Textual ambiguity, 214
 - Textual data, 57
 - “with” statement, 57
 - open text file, 57–58
 - write data to text file, 58
 - Third-party libraries, 25
 - Tighter code, 39
 - Time series data, 141
 - convert to time series datetime in Pandas, 149–153
 - data from float to int, 143–149
 - dataset, 141–143
 - Titanic dataset, 85
 - Tkinter, 285
 - Top2Vec, 226
 - .get_topic_sizes(), 232
 - .get_topics() method, 232
 - bigrams, 235–236
 - drawbacks, 230
 - keyword arguments, 231
 - loading, 236–237
 - pip, 231
 - README on GitHub, 234–235
 - saving, 236–237
 - search_documents_by_topic()
 - method, 233
 - trigrams, 235–236
 - Topic modeling
 - cluster, 214
 - definition, 211
 - LDA topic modeling (*see* Latent Dirichlet Allocation (LDA) topic modeling)
 - machine learning-based approach, 212
 - rules-based methods, 212
 - text classification, 212
 - topics, 213–214
 - transformer models
 - .encode() method, 226

analyzing labels, 227–229
bag-of-word (BOW) index, 226
converting text into vector, 226
HDBScan, 225–227
outliers (noise), 229–230
Sentence Transformers installation,
 225
Top2Vec, 226
 UMAP, 225, 227
Toponyms, 198
Transformer-based machine learning
 models, 198
Transformer models
 bag-of-word (BOW) index, 226
 HDBScan library, 225, 226
 sentence transformers installation, 225
 transformer-based topic modeling
 libraries, 226
 UMAP library, 225
Trigrams, 215
Trinket, 7
Truth and Reconciliation Commission
 (TRC), 211
Tuples, 32–33
Type function, 21
TypeError, 49

U

United States Holocaust Memorial Museum
 (USHMM), 195–196
Unsupervised learning, 212
Upper method, 26

V

Variables, 17–18

W

Web pages scraping
 with BeautifulSoup, 71–73
 with requests, 70–71
Web scraping, 65, 66
Website's HTML, 69–70
While loop, 42–43
Wikipedia, 69
“With” statement, 57
Working with external data
 working with JSON data, 59
 reading data with json.load(), 60
 writing data with json.dump(), 59–60
working with multiple files
 grabbing multiple nested directories,
 61
 walking a directory, 61–63
 working with glob, 60–61
working with textual data, 57
 “with” statement, 57
 open text file, 57–58
 write data to text file, 58

Z

Zero-index language, 31