IT Industry Readiness Bootcamp Program

A Project of Information Science & Technology Department, Government of Sindh

Data Science with Python

Course Manual

Prepared By:

Dr. Muhammad Hussain Mughal

Contents

Week 1: Introduction to Data Analysis and Reporting 14
Day 01 – INTRODUCTION 14
Basics of data analysis and reporting 14
What Is Data Science? 14
Why Python? 15
What is Artificial Intelligence? 15
What is Data Science & Machine Learning? 15
Different phases of a typical Analytics/Data Science projects and role of Python 16
Regression vs. Classification 16
Regression 16
Classification 17
Day 02 -PYTHON ESSENTIALS 17
Introduction to the installation of Anaconda 17
Introduction to Python Editors & IDE's (Anaconda, pycharm, Jupyter etc…) 17
Understand Jupyter notebook & Customize Settings 18
Overview of Python- Starting with Python 18
Installing Python 18
Accessing Source Code with ?? 20
Text Entry Shortcuts 23
Command History Shortcuts 24
IPython Magic Commands 25
Help on Magic Functions: ?, %magic, and %lsmagic 28
Profiling and Timing Code 28
Day 03 – Designing a Program 31
Input, Processing, and Output 31
Python Objects and data types 31
Strings and String Literals 32
Comments 32
Variables 32
Numeric Data Types and Literals 33
Escape Character 33
Day 04-Core built-in data structures – Lists, Tuples, Dictionaries, Sets 34
Introduction to Lists 34
Tuples 35

Dictionaries......35
Creating a Dictionary......35
Retrieving a Value from a Dictionary......35
Using the in and not in Operators to Test for a Value in a Dictionary......36
Sets......37
Lab activity -Sets......38
Day-05: Decision Structures and Boolean Logic......38
The if Statement......38
Boolean Expressions and Relational Operators......39
The if-else Statement......39
if-elif-else Statement......40
Repetition Structures......40
Condition-Controlled and Count-Controlled Loops......40
The while Loop: A Condition-Controlled Loop......40
The for Loop: A Count-Controlled Loop......41
Calculating a Running Total......41
Sentinels......42
Nested Loops......42
Lab Activity: Nested Loops......43
Week 2 -Data Manipulation and Cleaning......44
Day 01- Functions, Packages......44
Benefits of Modularizing a Program with Functions......44
Void Functions and Value-Returning Functions......44
Defining and Calling a Void Function......44
Function Names......44
Defining and Calling a Function......45
Calling a Function......45
Scope and Local Variables......46
Global Variables and Global Constants......46
Introduction to Value-Returning Functions: Generating Random Numbers......46
Day-02: String, built-in String methods, String Manipulation, and regular expressions......48
Basic String Operations......48
Lab Activtity- Python essentials......49
Day 03- EXPORTING DATA USING PYTHON MODULES (numpy)......53
NumPy Array Attributes......54
Array Indexing: Accessing Single Elements......55

Array Slicing: Accessing Subarrays 56
Multidimensional subarrays 57
Computation on NumPy Arrays: Universal Functions 62
Array arithmetic 62
Absolute value 63
Trigonometric functions 63
Specialized ufuncs 65
Advanced Ufunc Features 66
Aggregates 67
Aggregations: Min, Max, and Everything in Between 67
Summing the Values in an Array 67
Minimum and Maximum 68
Multidimensional aggregates 68
Computation on Arrays: Broadcasting 71
Modifying Values with Fancy Indexing 74
Sorting Arrays 77
Fast Sorting in NumPy: np.sort and np.argsort 77
Sorting along rows or columns 78
Partial Sorts: Partitioning 79
Day 04- Data Manipulation with Pandas 83
Pandas Introduction 83
Installing and Using Pandas 84
Introducing Pandas Objects 85
The Pandas Series Object 85
Series as specialized dictionary 87
The Pandas DataFrame Object 88
DataFrame as a generalized NumPy array 88
DataFrame as specialized dictionary 90
Constructing DataFrame objects 90
From a NumPy structured array. 91
The Pandas Index Object 92
Index as ordered set 93
Data Indexing and Selection 93
Data Selection in Series 93
Series as one-dimensional array 94
Data Selection in DataFrame 96

DataFrame as a dictionary 96
DataFrame as two-dimensional array 98
Additional indexing conventions 100
Operating on Data in Pandas 101
Index alignment in DataFrame 103
Handling Missing Data 106
Trade-Offs in Missing Data Conventions 106
Missing Data in Pandas 106
Operating on Null Values 109
Hierarchical Indexing 113
A Multiply Indexed Series 113
Methods of MultiIndex Creation 117
Indexing and Slicing a MultiIndex 120
Rearranging Multi-Indices 123
Data Aggregations on Multi-Indices 126
Combining Datasets: Merge and Join 127
Categories of Joins 127
Specification of the Merge Key 130
Specifying Set Arithmetic for Joins 134
Day 05- Descriptive Statistics 136
Cleansing Data with Pandas 136
Aggregation and Grouping 141
Planets Data 141
Simple Aggregation in Pandas 141
GroupBy: Split, Apply, Combine 143
Aggregate, filter, transform, apply. 147
Aggregation. 147
Filtering. 148
Transformation. 149
Week 3- Data Cleaning and Summerization 152
Day 01- Pandas String Operations 152
Tables of Pandas String Methods 153
Methods using regular expressions 155
Dates and Times in Python 158
Typed arrays of times: NumPy's datetime64 159
Dates and times in Pandas: Best of both worlds 162

Pandas Time Series: Indexing by Time 162
Pandas Time Series Data Structures 163
Frequencies and Offsets 166
Resampling, Shifting, and Windowing 168
Resampling and converting frequencies 169
Rolling windows 173
Where to Learn More 174
Example: Visualizing Seattle Bicycle Counts 174
Day-02: Digging into the data 179
High-Performance Pandas: eval() and query() 181
Motivating query() and eval(): Compound Expressions 181
pandas.eval() for Efficient Operations 183
DataFrame.eval() for Column-Wise Operations 185
DataFrame.query() Method 187
Performance: When to Use These Functions 188
Lab Activity -DataFrame Data Structure 189
Lab Activity -Merging DataFrames 193
Lab activity - DataFrame` Indexing and Loading 195
Day-03: Pivot Tables 197
Motivating Pivot Tables 197
Pivot Tables by Hand 198
Pivot Table Syntax 198
Multilevel pivot tables 199
Example: Birthrate Data 201
Lab Activity 206
Day-04: What Is Machine Learning? 209
Categories of Machine Learning 210
Qualitative Examples of Machine Learning Applications 210
Summary 219
Day-05: Introducing Scikit-Learn 220
Data Representation in Scikit-Learn 220
Data as table 220
Features matrix 221
Target array 221
Scikit-Learn's Estimator API 223
Basics of the API 224

Supervised learning example: Simple linear regression 224
Predict labels for unknown data. 227
Supervised learning example: Iris classification 228
Unsupervised learning example: Iris dimensionality 229
Unsupervised learning: Iris clustering 230
Application: Exploring Handwritten Digits 231
Week 4- Data visualization using Matplotlib 236
Day-01: Data visualization using Matplotlib 236
Introduction and brief histroy 236
Importing matplotlib 236
Setting Styles 236
show() or No show()? How to Display Your Plots 236
Plotting from a script 237
Plotting from an IPython shell 237
Plotting from an IPython notebook 238
Saving Figures to File 238
Two Interfaces for the Price of One 240
MATLAB-style interface 240
Object-oriented interface 241
Simple Line Plots 241
Adjusting the Plot: Line Colors and Styles 243
Adjusting the Plot: Axes Limits 245
Labeling Plots 248
Simple Scatter Plots 250
Scatter Plots with plt.plot 251
Scatter Plots with plt.scatter 253
plot Versus scatter: A Note on Efficiency 255
Visualizing Errors 256
Basic Errorbars 256
Continuous Errors 257
Density and Contour Plots 259
Day-02: Visualizing a Three-Dimensional Function 259
Histograms, Binnings, and Density 264
Day-03: Two-Dimensional Histograms and Binnings 266
plt.hist2d: Two-dimensional histogram 266
plt.hexbin: Hexagonal binnings 267

Kernel density estimation 267
Customizing Plot Legends 269
Choosing Elements for the Legend 271
Legend for Size of Points 272
Multiple Legends 274
Customizing Colorbars 275
Customizing Colorbars 276
Choosing the colormap 277
Color limits and extensions 280
Discrete colorbars 281
Handwritten Digits 282
Multiple Subplots 283
plt.axes: Subplots by Hand 284
plt.subplot: Simple Grids of Subplots 285
plt.subplots: The Whole Grid in One Go 287
plt.GridSpec: More Complicated Arrangements 288
Day-04: Text and Annotation 290
Example: Effect of Holidays on US Births 291
Transforms and Text Position 293
Arrows and Annotation 295
Customizing Ticks 298
Major and Minor Ticks 299
Hiding Ticks or Labels 300
Reducing or Increasing the Number of Ticks 301
Fancy Tick Formats 303
Summary of Formatters and Locators 305
Customizing Matplotlib: Configurations and Stylesheets 306
Plot Customization by Hand 306
Changing the Defaults: rcParams 308
Stylesheets 310
FiveThirtyEight style 311
ggplot 312
Dark background 313
Grayscale 313
Seaborn style 314
Day-05: Three-Dimensional Plotting in Matplotlib 315

Three-Dimensional Points and Lines 315
Three-Dimensional Contour Plots 316
Wireframes and Surface Plots 318
Surface Triangulations 320
Week 5: Data visualization with Seaborn 324
Day-01: Visualization with Seaborn 324
Seaborn Versus Matplotlib 324
Exploring Seaborn Plots 326
Histograms, KDE, and densities 326
Pair plots 330
Faceted histograms 331
Factor plots 332
Joint distributions 333
Bar plots 334
Example: Exploring Marathon Finishing Times 335
Day 02- Data Visualization on World Map- Geographic Data with Basemap 344
Map Projections 346
Cylindrical projections 347
Pseudo-cylindrical projections 348
Perspective projections 349
Conic projections 350
Drawing a Map Background 351
Plotting Data on Maps 353
Example: California Cities 354
Example: Surface Temperature Data 356
Day-03: Visualization using google maps and ArcGis(Iris Data) 358
Day-04: Discussion on projects and exploring other datasets 358
Day-05: Mid Assessments 358
Week 6 : Advance Data Analytics 359
Day-01: Hyperparameters and Model Validation 359
Thinking About Model Validation 359
Model validation the right way: Holdout sets 360
Model validation via cross-validation 360
Selecting the Best Model 363
The bias–variance trade-off 363
Validation curves in Scikit-Learn 366

Learning Curves ... 370
Learning curves in Scikit-Learn ... 372
Validation in Practice: Grid Search ... 373
Day-02: Feature Engineering ... 375
Categorical Features ... 375
Text Features ... 377
Image Features ... 378
Derived Features ... 378
Imputation of Missing Data ... 380
Feature Pipelines ... 381
Day-03: Linear Regression ... 382
Simple Linear Regression ... 382
Basis Function Regression ... 385
Regularization ... 389
Example: Predicting Bicycle Traffic ... 392
Day-04: Support Vector Machines ... 398
Motivating Support Vector Machines ... 398
Support Vector Machines: Maximizing the Margin ... 399
Fitting a support vector machine ... 400
Beyond linear boundaries: Kernel SVM ... 404
Day-05: Decision Trees and Random Forests ... 412
Motivating Random Forests: Decision Trees ... 413
Ensembles of Estimators: Random Forests ... 417
Random Forest Regression ... 419
Example: Random Forest for Classifying Digits ... 421
Summary of Random Forests ... 434
In Depth: Principal Component Analysis ... 435
Introducing Principal Component Analysis ... 435
PCA as Noise Filtering ... 442
Example: Eigenfaces ... 444
Week 7: Creating Reports and Dashboards ... 447
Day-01 : Introduction to Dashboards ... 447
Data Analytics Dashboard Benefits ... 447
What are some Data Analytics Dashboard Examples? ... 448
What are the Best Analytics Dashboard Tools? ... 449
Building interactive dashboards with libraries like Dash or Streamlit ... 450

1. How to import the required libraries and read input data 450
2. How to do a basic dashboard setup 451
3. How to design a user interface 451
Page title 451
Top-level filter 451
KPIs/summary cards 452
Interactive charts 452
Data table 453
4. How to refresh the dashboard for real-time or live data feed 453
5. How to auto-update components 453
Day-02: Develop Data Visualization Interfaces in Python With Dash 456
What Is Dash? 456
Get Started With Dash in Python 457
How to Set Up Your Local Environment 457
How to Build a Dash Application 458
Initializing Your Dash Application 458
Defining the Layout of Your Dash Application 459
Style Your Dash Application 462
How to Apply a Custom Style to Your Components 462
How to Improve the Looks of Your Dashboard 464
Add Interactivity to Your Dash Apps Using Callbacks 471
How to Create Interactive Components 472
How to Define Callbacks 476
Deploy Your Dash Application to PythonAnywhere 479
Day-03: Host, run, and code Python in the cloud! 479
How to Create a Free PythonAnywhere Account 479
How to Deploy Your Avocado Analytics App 480
Day-04: Interactive Data Visualization in Python With Bokeh 487
Building a visualization with Bokeh involves the following steps: 487
Prepare the Data 487
Determine Where the Visualization Will Be Rendered 487
Set up the Figure(s) 487
Connect to and Draw Your Data 488
Organize the Layout 488
Preview and Save Your Beautiful Data Creation 488
Generating Your First Figure 489

Getting Your Figure Ready for Data 492
Drawing Data With Glyphs 495
Day-05:Organizing Multiple Visualizations With Layouts 504
Adding Interaction 511
Configuring the Toolbar 511
Selecting Data Points 512
Adding Hover Actions 514
Linking Axes and Selections 516
Highlighting Data Using the Legend 522
Week 8: Text analysis and sentiment analysis 525
Day- 01 & 02: NLTK libaray for text analysis 525
Analyze your text 525
Tokenizing 525
Filtering Stop Words 526
Stemming 527
Lemmatizing 534
Chunking 536
Chinking 538
Getting Text to Analyze 542
Day-03 & 04: Sentiment Analysis: First Steps With Python's NLTK Library 551
Steps for Sentiment Analysis 552
Compiling Data 553
Creating Frequency Distributions 555
Installing and Importing scikit-learn 565
Day-05: Labs and Practice activities for sentiments analysis on various datasets 567
Week 9: Time series analysis and forecasting 568
Day-01: Time Series – Introduction 568
Time Series – Data Processing and Visualization 569
Converting to datetime object 569
Showing plots 570
Time Series – Modeling 570
Time Series Modeling Techniques 570
Naïve Methods 570
Auto Regression 570
ARIMA Model 570
Exponential Smoothing 571

LSTM 571
Time Series – Parameter Calibration 571
Methods for Calibration of Parameters 571
Hit-and-try 571
Grid Search 571
Genetic Algorithm 571
Time Series – Naïve Methods 572
Showing statistics 572
Showing 1st naïve method 572
Showing 2nd naïve method 573
Time Series – Auto Regression 573
Time Series – Moving Average 574
Time Series – ARIMA 575
Time Series – Variations of ARIMA 576
Day-02:Time Series – Exponential Smoothing 579
Time Series – Walk Forward Validation 580
Day-03:Time Series – LSTM Model 582
Neural Networks 582
Recurrent Neural Networks 583
LSTM 583
Day-04: Time Series – Error Metrics 586
Mean Square Error 586
Root Mean Square Error 587
Root Mean Square Error 587
Mean Absolute Error 587
Mean Absolute Percentage Error 588
Day-05: Time Series – Applications 588
Time Series – Further Scope 588
Time Series Data 588
Non-Time Series Data 589
Week 10 & 11: Data Analysis Projects 590
List of resources & Acknoledgements: 591

Week 1: Introduction to Data Analysis and Reporting

Day 01 – INTRODUCTION

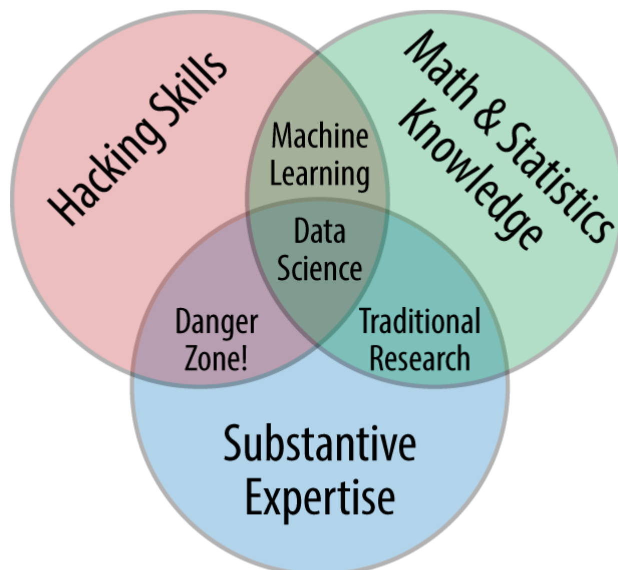
Data analysis is the process of inspecting, cleansing, transforming, and modeling data with the goal of discovering useful information, informing conclusions, and supporting decision-making. Data analysis has multiple facets and approaches, encompassing diverse techniques under a variety of names, and is used in different business, science, and social science domains. In today's business world, data analysis plays a role in making decisions more scientific and helping businesses operate more effectively.

Basics of data analysis and reporting

A data analysis report is a type of business report in which you present quantitative and qualitative data to evaluate your strategies and performance. Based on this data, you give recommendations for further steps and business decisions while using the data as evidence that backs up your evaluation.

Today, data analysis is one of the most important elements of business intelligence strategies as companies have realized the potential of having data-driven insights at hand to help them make data-driven decisions.

What Is Data Science?



The "data science" is fundamentally an *interdisciplinary* subject. Data science comprises three distinct and overlapping areas: the skills of a *statistician* who knows how to model and summarize datasets (which are growing ever larger); the skills of a *computer scientist* who can
design and use algorithms to efficiently store, process, and visualize this data; and the *domain expertise*—what we might think of as "classical" training in a subject—necessary both to formulate the right questions and to put their answers in context.

Defining data science

If science is a systematic method by which people study and explain domainspecific phenomena that occur in the natural world, you can think of data science as the scientific domain that's dedicated to knowledge discovery via data analysis. With respect to data science, the term domain-specific refers to the industry sector or subject matter domain that data science methods are being used to explore.

Data scientists use mathematical techniques and algorithmic approaches to derive solutions to complex business and scientific problems. Data science practitioners use its predictive methods to derive insights that are otherwise unattainable. In business and in science, data science methods can provide more robust decisionmaking capabilities:
Using data science skills, you can do cool things like the following:
»»Use machine learning to optimize energy usage and lower corporate carbon footprints.
»»Optimize tactical strategies to achieve goals in business and science.
»»Predict for unknown contaminant levels from sparse environmental datasets.
»»Design automated theft- and fraud-prevention systems to detect anomalies and trigger alarms based on algorithmic results.
»»Craft site-recommendation engines for use in land acquisitions and real estate development.
»»Implement and interpret predictive analytics and forecasting techniques for net increases in business value.

Why Python?

Python has emerged over the last couple of decades as a first-class tool for scientific computing tasks, including analyzing and visualizing large datasets. This may have surprised early proponents of the Python language: the language itself was not explicitly designed with data analysis or scientific computing in mind.
The usefulness of Python for data science stems primarily from the large and active ecosystem of third-party packages: NumPy for manipulation of homogeneous array-based data, Pandas for manipulation of heterogeneous and labeled data, SciPy for common scientific computing tasks, Matplotlib for publication-quality visualizations, IPython for interactive execution and sharing of code, Scikit-Learn for machine learning, and many more.

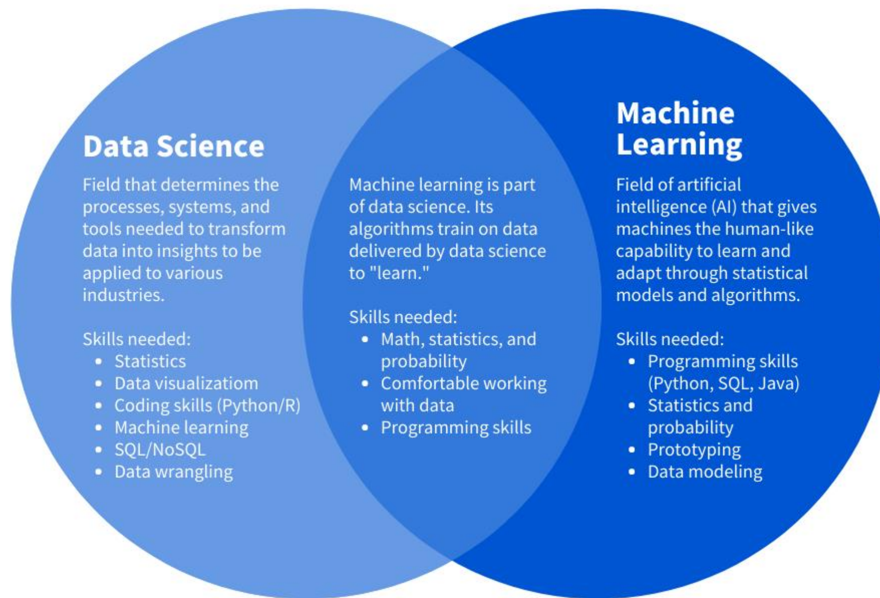
What is Artificial Intelligence?

Artificial intelligence is the simulation of human intelligence processes by machines, especially computer systems. Specific applications of AI include expert systems, natural language processing, speech recognition and machine vision.

What is Data Science & Machine Learning?

Data science is a field that studies data and how to extract meaning from it, whereas machine learning is a field devoted to understanding and building methods that utilize data to improve

performance or inform predictions. Machine learning is a branch of artificial intelligence



Different phases of a typical Analytics/Data Science projects and role of Python

Data Analytics Life Cycle Phases

- Phase 1: Data Discovery and Formation.
- Phase 2: Data Preparation and Processing.
- Phase 3: Design a Model.
- Phase 4: Model Building.
- Phase 5: Result Communication and Publication.
- Phase 6: Measuring Effectiveness.

Regression vs. Classification

Regression

In statistical modeling, regression analysis is a set of statistical processes for estimating the relationships between a dependent variable (often called the 'outcome' or 'response' variable, or a 'label' in machine learning parlance) and one or more independent variables (often called 'predictors', 'covariates', 'explanatory variables' or 'features'). The most common form of regression analysis is linear regression, in which one finds the line (or a more complex linear combination) that most closely fits the data according to a specific mathematical criterion. For example, the method of ordinary least squares computes the unique line (or hyperplane) that minimizes the sum of squared differences between the true data and that line (or hyperplane). For specific mathematical reasons (see linear regression), this allows the researcher to estimate the conditional expectation (or population average value) of the dependent variable when the independent variables take on a given set of values. Less common forms of regression use slightly different procedures to estimate alternative location parameters (e.g., quantile regression or Necessary Condition Analysis) or estimate the conditional expectation across a broader collection of non-linear models (e.g., nonparametric regression).

Regression analysis is primarily used for two conceptually distinct purposes. First, regression analysis is widely used for prediction and forecasting, where its use has substantial overlap with the field of

machine learning. Second, in some situations regression analysis can be used to infer causal relationships between the independent and dependent variables. Importantly, regressions by themselves only reveal relationships between a dependent variable and a collection of independent variables in a fixed dataset. To use regressions for prediction or to infer causal relationships, respectively, a researcher must carefully justify why existing relationships have predictive power for a new context or why a relationship between two variables has a causal interpretation. The latter is especially important when researchers hope to estimate causal relationships using observational data.

Classification

Classification is a process related to categorization, the process in which ideas and objects are recognized, differentiated and understood. Classification is the grouping of related facts into classes. It may also refer to a process which brings together like things and separates unlike things.

Day 02 -PYTHON ESSENTIALS

Installing Python and the suite of libraries that enable scientific computing is straightforward. This section will outline some of the considerations to keep in mind when setting up your computer.

Though there are various ways to install Python, the one I would suggest for use in data science is the Anaconda distribution, which works similarly whether you use Windows, Linux, or Mac OS X. The Anaconda distribution comes in two flavors:

Miniconda gives you the Python interpreter itself, along with a command-line tool called conda that operates as a cross-platform package manager geared toward Python packages, similar in spirit to the apt or yum tools that Linux usersmight be familiar with.

Anaconda includes both Python and conda, and additionally bundles a suite of other preinstalled packages geared toward scientific computing. Because of the size of this bundle, expect the installation to consume several gigabytes of disk space.

Introduction to the installation of Anaconda

Anaconda is a reasonably sophisticated installer. It supports installation from local and remote sources such as CDs and DVDs, images stored on a hard drive, NFS, HTTP, and FTP. Installation can be scripted with kickstart to provide a fully unattended installation that can be duplicated on scores of machines. It can also be run over VNC on headless machines. A variety of advanced storage devices including LVM, RAID, iSCSI, and multipath are supported from the partitioning program. Anaconda provides advanced debugging features such as remote logging, access to the python interactive debugger, and remote saving of exception dumps.

Introduction to Python Editors & IDE's (Anaconda, pycharm, Jupyter etc…)

Most data scientists and software developers prefer Python because of the various functionalities provided by Python and the best among those is its open-source feature. Anyone all over the globe can create their own package and make it public for others to use, hence improving the python backend daily.

There are various IDEs in the market to select from such as Spyder, Atom, Pycharm, Pydev etc. Data scientists prefer Spyder among all the different IDEs available and the driving fact behind this is that Spyder was built specifically for data science. Its interface allows the user to scroll through various data variables and also ready to use online help option. The output of the code can be viewed in the

python console on the same screen. You can work on different scripts at a moment and then try them out one by one in the same console or different as per your choice all the variables used will be stored in the variable explorer tab. It also provides an option to view graphs and visualizations in the plot window. You can also cover the basics concepts by taking up free Syder python and also check out Python Libraries for Machine Learning from Great Learning Academy.

Understand Jupyter notebook & Customize Settings

The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:

Overview of Python- Starting with Python

The Python interpreter can run Python programs that are saved in files or interactively execute Python statements that are typed at the keyboard. Python comes with a program named IDLE that simplifies the process of writing, executing, and testing programs.

Installing Python

Before you can try any of the programs shown in this book, or write any programs of your own, you need to make sure that Python is installed on your computer and properly configured. If you are working in a computer lab, this has probably been done already. If you are using your own computer, you can follow the instructions in Appendix A to download and install Python.

The Python Interpreter

You learned earlier that Python is an interpreted language. When you install the Python language on your computer, one of the items that is installed is the Python interpreter. The *Python interpreter* is a program that can read Python programming statements and execute them. (Sometimes, we will refer to the Python interpreter simply as the interpreter.) You can use the interpreter in two modes: interactive mode and script mode. In *interactive mode,* the interpreter waits for you to type Python statements on the keyboard. Once you type a statemen t, the interpreter executes it and then waits for you to type another statement. In *script mode,* the interpreter reads the contents of a file that contains Python statements. Such a file is known as a *Python program* or a *Python script*. The interpreter executes each statement in the Python program as it reads it.

Interactive Mode

Once Python has been installed and set up on your system, you start the interpreter in interactive mode by going to the operating system's command line and typing the following command:

```
python
```

If you are using Windows, you can alternatively type *Python* in the Windows search box. In the search results, you will see a program named something like *Python 3.11*. (The " 3.11" is the version of Python that is installed. At the time this is being written, Python 3.11 is the latest version.) Clicking this item will start the Python interpreter in interactive mode.

When the Python interpreter is running in interactive mode, it is commonly called the Python shell.

The >>> that you see is a prompt that indicates the interpreter is waiting for you to type a Python statement. Let's try it out. One of the simplest things that you can do in Python is print a message on the screen. For example, the following statement prints the message Python programming is fun! on the screen:

```
print('Python programming is fun!')
```

You can think of this as a command that you are sending to the Python interpreter. If you type the statement exactly as it is shown, the message Python programming is fun! Is printed on the screen. Here is an example of how you type this statement at the interpreter's prompt:

>>> print('Python programming is fun!') Press Enter

After typing the statement, you press the Enter key, and the Python interpreter executes the statement, as shown here:

>>> print('Python programming is fun!') Enter

Python programming is fun!

Launching the Jupyter Notebook

The Jupyter notebook is a browser-based graphical interface to the IPython shell, andbuilds on it a rich set of dynamic display capabilities. As well as executing Python/ IPython statements, the notebook allows the user to include formatted text, static and dynamic visualizations, mathematical equations, JavaScript widgets, and much more.Furthermore, these documents can be saved in a way that lets other people open themand execute the code on their own systems.

Though the IPython notebook is viewed and edited through your web browser win- dow, it must connect to a running Python process in order to execute code. To start this process (known as a "kernel"), run the following command in your system shell:

```
$ jupyter notebook
```

This command will launch a local web server that will be visible to your browser. It immediately spits out a log showing what it is doing; that log will look something likethis:

Upon issuing the command, your default browser should automatically open and navigate to the listed local URL; the exact address will depend on your system. If the browser does not open automatically, you can open a window and manually open this address (*http://localhost:8888/* in this example).

Help and Documentation in IPython

If you read no other section in this chapter, read this one: I find the tools discussed here to be the most transformative contributions of IPython to my daily workflow.

When a technologically minded person is asked to help a friend, family member, or colleague with a computer problem, most of the time it's less a matter of knowing theanswer as much as knowing how to quickly find an unknown answer. In data science it's the same: searchable web resources such as online documentation, mailing-list threads, and Stack Overflow answers contain a wealth of information, even (espe- cially?) if it is a topic you've found yourself searching before. Being an effective prac-titioner of data science is less about memorizing the tool or command you should usefor every possible situation, and more about learning to effectively find the informa- tion you don't know, whether through a web search engine or another means.

One of the most useful functions of IPython/Jupyter is to shorten the gap between theuser and the type of documentation and search that will help them do their work effectively. While web searches still play a role in answering complicated questions, an amazing amount of information can be found through IPython alone. Some examples of the questions IPython

can help answer in a few keystrokes:

- How do I call this function? What arguments and options does it have?
- What does the source code of this Python object look like?
- What is in this package I imported? What attributes or methods does this objecthave?

Here we'll discuss IPython's tools to quickly access this information, namely the ? character to explore documentation, the ?? characters to explore source code, and theTab key for autocompletion.

Accessing Documentation with ?

The Python language and its data science ecosystem are built with the user in mind, and one big part of that is access to documentation. Every Python object contains the

reference to a string, known as a *docstring*, which in most cases will contain a concise summary of the object and how to use it. Python has a built-in help() function that can access this information and print the results. For example, to see the documenta- tion of the built-in lenfunction, you can do the following:

```
In [1]: help(len)
Help on built-in function len in module builtins:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Depending on your interpreter, this information may be displayed as inline text, or in some separate pop-up window.

Because finding help on an object is so common and useful, IPython introduces the ? character as a shorthand for accessing this documentation and other relevant information:

```
In [2]: len?
Type:
            builtin_function_or_meth
odString form: <built-in function len>
Namespace:Python builtin
Docstring:
len(object) -> integer

Return the number of items of a sequence or mapping.
```

Accessing Source Code with ??

Because the Python language is so easily readable, you can usually gain another levelof insight by reading the source code of the object you're curious about. IPython pro-vides a shortcut to the source code with the double question mark (??):

```
In [8]: square?? Type:
            function
```

```
String form: <function square at
0x103713cb0>Definition: square(a)

Source:

def square(a):

    "Return the square of a"

    return a ** 2
```

For simple functions like this, the double question mark can give quick insight intothe under-the-hood details.

If you play with this much, you'll notice that sometimes the ?? suffix doesn't displayany source code: this is generally because the object in question is not implemented inPython, but in C or some other compiled extension language. If this is the case, the ??suffix gives the same output as the ? suffix. You'll find this particularly with many of Python's built-in objects and types, for example lenfrom above:

```
In [9]: len??

Type:
            builtin_function_or_meth
odString form: <built-in function len>
Namespace:Python builtin

Docstring:

len(object) -> integer


Return the number of items of a sequence or mapping.
```

Using ?and/or ??gives a powerful and quick interface for finding information about what any Python function or module does.

Exploring Modules with Tab Completion

IPython's other useful interface is the use of the Tab key for autocompletion and exploration of the contents of objects, modules, and namespaces. In the examples thatfollow, we'll use <TAB>to indicate when the Tab key should be pressed.

Tab completion of object contents

Every Python object has various attributes and methods associated with it. Like with the help function discussed before, Python has a built-in dir function that returns a list of these, but the tab-completion interface is much easier to use in practice. To seea list of all available attributes of an object, you can type the name of the object fol- lowed by a period (.) character and the Tab key:

```
In [10]: L.<TAB>
L.appen  L.copy   L.extend  L.insert  L.remove L.sort
d
L.clear  L.coun   L.index   L.pop     L.reverse
         t
```

To narrow down the list, you can type the first character or several characters of the name, and the Tab key will find the matching attributes and methods:

```
In [10]: L.c<TAB>

L.clear  L.copy   L.count
```

```
In [10]: L.co<TAB>
L.copy    L.count
```

If there is only a single option, pressing the Tab key will complete the line for you. Forexample, the following will instantly be replaced with L.count:

```
In [10]: L.cou<TAB>
```

Though Python has no strictly enforced distinction between public/externalattributes and private/internal attributes, by convention a preceding underscore isused to denote such methods. For clarity, these private methods and special methods are omitted from the list by default, but it's possible to list them by explicitly typing the underscore:

```
In [10]: L._<TAB>L.__add L._class
                L.__gt L._hash_L._reduce
                L._reduce_ex
```

For brevity, we've only shown the first couple lines of the output. Most of these are Python's special double-underscore methods (often nicknamed "dunder" methods).

Tab completion when importing

Tab completion is also useful when importing objects from packages. Here we'll use itto find all possible imports in the itertoolspackage that start with co:

```
In [10]: from itertools import co<TAB>
combinations
                                   compres
s       combinations_with_replacement
count
```

Similarly, you can use tab completion to see which imports are available on your sys-tem (this will change depending on which third-party scripts and modules are visible to your Python session):

```
In [10]: import <TAB>

Display all 399 possibilities? (y or n)
Crypto              dis                 py_compi
Cython              distutils           le
                                        pyclbr

...                 ...                 ...
difflib             pwd                 zmq

In [10]: import h<TAB>

hashlib             hmac                http

heapq               html                husl
```

(Note that for brevity, I did not print here all 399 importable packages and modules on my system.)

Beyond tab completion: Wildcard matching

Tab completion is useful if you know the first few characters of the object or attributeyou're looking for, but is little help if you'd like to match characters at the middle or end of the word. For this use case, IPython provides a means of wildcard matchingfor names using the *character.

For example, we can use this to list every object in the namespace that ends with

Warning:

```
In [10]: *Warning?
BytesWarning                RuntimeWarning
DeprecationWarning          SyntaxWarning
FutureWarning               UnicodeWarning
ImportWarning               UserWarning
PendingDeprecationWarning   Warning
ResourceWarning
```

Notice that the *character matches any string, including the empty string.

Similarly, suppose we are looking for a string method that contains the word find somewhere in its name. We can search for it this way:

Keyboard Shortcuts in the IPython Shell

If you spend any amount of time on the computer, you've probably found a use for keyboard shortcuts in your workflow. Most familiar perhaps are Cmd-C and Cmd-V (or Ctrl-C and Ctrl-V) for copying and pasting in a wide variety of programs and sys-tems. Power users tend to go even further: popular text editors like Emacs, Vim, and others provide users an incredible range of operations through intricate combina- tions of keystrokes.

The IPython shell doesn't go this far, but does provide a number of keyboard short- cuts for fast navigation while you're typing commands. These shortcuts are not in factprovided by IPython itself, but through its dependency on the GNU Readline library: thus, some of the following shortcuts may differ depending on your system configu- ration. Also, while some of these shortcuts do work in the browser-based notebook, this section is primarily about shortcuts in the IPython shell.

Once you get accustomed to these, they can be very useful for quickly performing certain commands without moving your hands from the "home" keyboard position.If you're an Emacs user or if you have experience with Linux-style shells, the follow- ing will be very familiar. We'll group these shortcuts into a few categories: *navigationshortcuts*, *text entry shortcuts*, *command history shortcuts*, and *miscellaneous shortcuts*.

Navigation Shortcuts

While the use of the left and right arrow keys to move backward and forward in the line is quite obvious, there are other options that don't require moving your hands from the "home" keyboard position:

Keystroke	Action
Ctrl-a	Move cursor to the beginning of the line
Ctrl-e	Move cursor to the end of the line
Ctrl-b (or the left arrow key)	Move cursor back one character
Ctrl-f (or the right arrow key)	Move cursor forward one character

Text Entry Shortcuts

While everyone is familiar with using the Backspace key to delete the previous char- acter,

reaching for the key often requires some minor finger gymnastics, and it only deletes a single character at a time. In IPython there are several shortcuts for remov- ing some portion of the text you're typing. The most immediately useful of these are the commands to delete entire lines of text. You'll know these have become second nature if you find yourself using a combination of Ctrl-b and Ctrl-d instead of reach- ing for the Backspace key to delete the previous character!

Keystroke	Action
Backspace key	Delete previous character in line
Ctrl-d	Delete next character in line
Ctrl-k	Cut text from cursor to end of line
Ctrl-u	Cut text from beginning fo line to cursor
Ctrl-y	Yank (i.e., paste) text that was previously cut
Ctrl-t	Transpose (i.e., switch) previous two characters

Command History Shortcuts

Perhaps the most impactful shortcuts discussed here are the ones IPython providesfor navigating the command history. This command history goes beyond your cur- rent IPython session: your entire command history is stored in a SQLite database in your IPython profile directory. The most straightforward way to access these is with the up and down arrow keys to step through the history, but other options exist as well:

Keystroke	Action
Ctrl-p (or the up arrow key)	Access previous command in history
Ctrl-n (or the down arrow key)	Access next command in history
Ctrl-r	Reverse-search through command history

The reverse-search can be particularly useful. Recall that in the previous section we defined a function called square. Let's reverse-search our Python history from a newIPython shell and find this definition again. When you press Ctrl-r in the IPython terminal, you'll see the following prompt:

```
In [1]:
(reverse-i-search)`':
```

If you start typing characters at this prompt, IPython will auto-fill the most recent command, if any, that matches those characters:

```
In [1]:
(reverse-i-search)`sqa': square??
```

At any point, you can add more characters to refine the search, or press Ctrl-r againto search further for another command that matches the query. If you followed along in the previous section, pressing Ctrl-r twice more gives:

```
In [1]:
(reverse-i-search)`sqa': def square(a):
    """Return the square of a"""
    return a ** 2
```

Once you have found the command you're looking for, press Return and the search will end. We can then use the retrieved command, and carry on with our session:

```
In [1]: def square(a):
    """Return the square of a"""
    return a ** 2

In [2]: square(2)
Out[2]: 4
```

Note that you can also use Ctrl-p/Ctrl-n or the up/down arrow keys to search through history, but only by matching characters at the beginning of the line. That is, if you type **def** and then press Ctrl-p, it would find the most recent command (if any)in your history that begins with the characters def.

Miscellaneous Shortcuts

Finally, there are a few miscellaneous shortcuts that don't fit into any of the preceding categories, but are nevertheless useful to know:

Keystroke	Action
Ctrl-l	Clear terminal screen
Ctrl-c	Interrupt current Python command
Ctrl-d	Exit IPython session

The Ctrl-c shortcut in particular can be useful when you inadvertently start a very long-running job.

While some of the shortcuts discussed here may seem a bit tedious at first, they quickly become automatic with practice. Once you develop that muscle memory, I suspect you will even find yourself wishing they were available in other contexts.

IPython Magic Commands

The previous two sections showed how IPython lets you use and explore Python effi-ciently and interactively. Here we'll begin discussing some of the enhancements that

IPython adds on top of the normal Python syntax. These are known in IPython as *magic commands*, and are prefixed by the % character. These magic commands are designed to succinctly solve various common problems in standard data analysis. Magic commands come in two flavors: *line magics*, which are denoted by a single % prefix and operate on a single line of input, and *cell magics*, which are denoted by a double %% prefix and operate on multiple lines of input. We'll demonstrate and dis- cuss a few brief examples here, and come back to more focused discussion of severaluseful magic commands later in the chapter.

Pasting Code Blocks: %paste and %cpaste

When you're working in the IPython interpreter, one common gotcha is that pasting multiline code blocks can lead to unexpected errors, especially when indentation and interpreter markers are involved. A common case is that you find some example codeon a website and want to paste it into your interpreter. Consider the following simple function:

```
>>> def donothing(x):
...  return x
```

The code is formatted as it would appear in the Python interpreter, and if you copy and paste this directly into IPython you get an error:

```
In [2]: >>> def donothing(x):
...:        ...        return x
                   ...:
                  File "<ipython-input-20-5a66c8964687>", line 2
...        return x
                               ^
            SyntaxError: invalid syntax
```

The interpreter is confused by the additional prompt characters in the direct paste. But never fear—IPython's %pastemagic function is designed to handle this exact typeof multiline, marked-up input:

```
In [3]: %paste
>>> def donothing(x):
...        return x
```

-- End pasted text --

The %paste command both enters and executes the code, so now the function isready to be used:

```
In [4]: donothing(10)
Out[4]: 10
```

A command with a similar intent is %cpaste, which opens up an interactive multiline prompt in which you can paste one or more chunks of code to be executed in a batch:

```
In [5]: %cpaste
Pasting code; enter '--' alone on the line to stop or  use Ctrl-D.
:>>> def  donothing(x):
:...        return x
:--
```

These magic commands, like others we will see, make available functionality that wouldbe difficult or impossible in a standard Python interpreter.

Running External Code: %run

As you begin developing more extensive code, you will likely find yourself working inboth IPython for interactive exploration and a text editor to store code youwant to reuse. Rather than running this code in a new window, running it within your IPython session can be convenient. This can be done with the %runmagic.

For example, imagine you've created a *myscript.py* file with the following contents:

```
#-------------------------------------

# file: myscript.py

def  square(x):
    """square  a  number"""
    return  x ** 2

for N in  range(1, 4):
    print(N, "squared is", square(N))
```

You can execute this from your IPython session as follows:

```
In [6]: %run  myscript.py
1 squared is  1
2  squared is  4
3  squared is  9
```

Note also that after you've run this script, any functions defined within it are availablefor use in your IPython session:

```
In [7]: square(5)
Out[7]: 25
```

There are several options to fine-tune how your code is run; you can see the docu- mentation in the normal way, by typing **%run?** in the IPython interpreter.

Timing Code Execution: %timeit

Another example of a useful magic function is %timeit, which will automatically determine the execution time of the single-line Python statement that follows it. For example, we may want to check the performance of a list comprehension:

```
In [8]: %timeit  L = [n ** 2 for n in range(1000)]1000
loops, best of 3: 325 µs per loop
```

The benefit of %timeit is that for short commands it will automatically perform mul-tiple

runs in order to attain more robust results. For multiline statements, adding a second % sign will turn this into a cell magic that can handle multiple lines of input. For example, here's the equivalent construction with a forloop:

```
In [9]: %%timeit
...: L = []
...: for n in range(1000):
...:         L.append(n ** 2)
...:
1000 loops, best of 3: 373 µs per loop
```

We can immediately see that list comprehensions are about 10% faster than the equivalent for loop construction in this case. We'll explore %timeit and otherapproaches to timing and profiling code in "Profiling and Timing Code" on page 25.

Help on Magic Functions: ?, %magic, and %lsmagic

Like normal Python functions, IPython magic functions have docstrings, and thisuseful documentation can be accessed in the standard manner. So, for example, toread the documentation of the %timeitmagic, simply type this:

```
In [10]: %timeit?
```

Documentation for other functions can be accessed similarly. To access a general description of available magic functions, including some examples, you can type this:

```
In [11]: %magic
```

For a quick and simple list of all available magic functions, type this:

```
In [12]: %lsmagic
```

Profiling and Timing Code

In the process of developing code and creating data processing pipelines, there are often trade-offs you can make between various implementations. Early in developing your algorithm, it can be counterproductive to worry about such things. As Donald Knuth famously quipped, "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

But once you have your code working, it can be useful to dig into its efficiency a bit. Sometimes it's useful to check the execution time of a given command or set of com-mands; other times it's useful to dig into a multiline process and determine where thebottleneck lies in some complicated series of operations. IPython provides access to a wide array of functionality for this kind of timing and profiling of code. Here we'll discuss the following IPython magic commands:

%time	Time the execution of a single statement
%timeit	Time repeated execution of a single statement for more accuracy
%prun	Run code with the profiler
%lprun	Run code with the line-by-line profiler

%memit	Measure the memory use of a single statement
%mprun	Run code with the line-by-line memory profiler

The last four commands are not bundled with IPython—you'll need to install the line_profiler and memory_profiler extensions, which we will discuss in the fol- lowing sections.

Timing Code Snippets: %timeit and %time

We saw the %timeitline magic and %%timeitcell magic in the introduction to magicfunctions in "IPython Magic Commands" ; %%timeit can be used to time the repeated execution of snippets of code:

```
In[1]: %timeit sum(range(100))

100000 loops, best of 3: 1.54 µs per loop
```

Note that because this operation is so fast, %timeitautomatically does a large numberof repetitions. For slower commands, %timeit will automatically adjust and performfewer repetitions:

```
In[2]: %%timeit
total = 0
    for i in range(1000):
        for j in range(1000): total
            += i * (-1) ** j

1 loops, best of 3: 407 ms per loop
```

Sometimes repeating an operation is not the best option. For example, if we have a list that we'd like to sort, we might be misled by a repeated operation. Sorting a pre- sorted list is much faster than sorting an unsorted list, so the repetition will skew the result:

```
In[3]: import random
    L = [random.random() for i in range(100000)]
    %timeit L.sort()

100 loops, best of 3: 1.9 ms per loop
```

For this, the %time magic function may be a better choice. It also is a good choice forlonger-running commands, when short, system-related delays are unlikely to affect the result. Let's time the sorting of an unsorted and a presorted list:

```
In[4]: import random
    L = [random.random() for i in range(100000)]
    print("sorting an unsorted list:")
    %time L.sort()

sorting an unsorted list:

CPU times: user 40.6 ms, sys: 896 µs, total: 41.5 ms
Wall time: 41.5 ms

In[5]: print("sorting an already sorted list:")
    %time L.sort()

sorting an already sorted list:
```

```
CPU times: user 8.18 ms, sys: 10 µs, total: 8.19 ms
Wall time: 8.24 ms
```

Notice how much faster the presorted list is to sort, but notice also how much longer the timing takes with %time versus %timeit, even for the presorted list! This is a result of the fact that %timeit does some clever things under the hood to prevent sys-tem calls from interfering with the timing. For example, it prevents cleanup of unused Python objects (known as *garbage collection*) that might otherwise affect the timing. For this reason, %timeitresults are usually noticeably faster than %timeresults.

For %time as with %timeit, using the double-percent-sign cell-magic syntax allows timing of multiline scripts:

```
In[6]: %%time
    total = 0
    for i in range(1000):
        for j in range(1000): total
            += i * (-1) ** j
```

```
CPU times: user 504 ms, sys: 979 µs, total: 505 ms
Wall time: 505 ms
```

For more information on %time and %timeit, as well as their available options, use the IPython help functionality (i.e., type **%time?** at the IPython prompt).

Profiling Full Scripts: %prun

A program is made of many single statements, and sometimes timing these state- ments in context is more important than timing them on their own. Python containsa built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function %prun.

By way of example, we'll define a simple function that does some calculations:

```
In[7]: def sum_of_lists(N):
        total = 0
        for i in range(5):
            L = [j ^ (j >> i) for j in range(N)]total
            += sum(L)
        return total
```

Now we can call %prunwith a function call to see the profiled results:

```
In[8]: %prun sum_of_lists(1000000)
```

In the notebook, the output is printed to the pager, and looks something like this:

14 function calls in 0.714 seconds

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall filename:lineno(function)
5	0.599	0.120	0.599	0.120 <ipython-input-19>:4(<listcomp>)
5	0.064	0.013	0.064	0.013 {built-in method sum}
1	0.036	0.036	0.699	0.699 <ipython-input-

```
                                        19>:1(sum_of_lists)
1      0.014    0.014    0.714          0.714 <string>:1(<module>)
1      0.000    0.000    0.714          0.714 {built-in method exec}
```

The result is a table that indicates, in order of total time on each function call, where the execution is spending the most time. In this case, the bulk of execution time is in the list comprehension inside sum_of_lists. From here, we could start thinkingabout what changes we might make to improve the performance in the algorithm.

Day 03 – Designing a Program

Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts
to create models of programs

Input, Processing, and Output

Input is data that the program receives. When a program receives data,
it usually processes it by performing some operation with it. The result
of the operation is sent out of the program as output.

Python Objects and data types

The following items are all considered objects in the Python programming
language:

»»Numbers
»»Strings
»»Lists
»»Tuples
»»Sets
»»Dictionaries
»»Functions
»»Classes

Additionally, all these items (except for the last two in the list) function as basic data types in plain ol' Python, which is Python with no external extensions added to it. (I introduce you to the external Python libraries NumPy, SciPy, Pandas, MatPlotLib, and Scikit-learn in the later section "Checking out some useful Python libraries." When you add these libraries, additional data types become available to you.)

In Python, functions do basically the same thing as they do in plain math — they accept data inputs, process them, and output the result. Output results depend wholly on the task the function was programmed to do. Classes, on the other hand, are prototypes of objects that are designed to output additional objects.

If your goal is to write fast, reusable, easy-to-modify code in Python, you must
use functions and classes. Doing so helps to keep your code efficient and
organized.

Sorting out the various Python data types

If you do much work with Python, you need to know how to work with different
data types. The main data types in Python and the general forms they take are
described in this list:

»»**Numbers:** Plain old numbers, obviously
»»**Strings:** '. . .' or ". . ."
»»**Lists:** [. . .] or [. . ., . . ., . . .]
»»**Tuples:** (. . .) or (. . ., . . ., . . .)

Numbers in Python
The Numbers data type represents numeric values that you can use to handle all
types of mathematical operations. Numbers come in the following types:
»»**Integer:** A whole-number format
»»**Long:** A whole-number format with an unlimited digit size
»»**Float:** A real-number format, written with a decimal point
»»**Complex:** An imaginary-number format, represented by the square root of –1

Strings and String Literals

Programs almost always work with data of some type. For example, Program 2-1 uses the
following three pieces of data:
'Kate Austen'
'123 Full Circle Drive
'Asheville, NC 28899'
These pieces of data are sequences of characters. In programming terms, a sequence of characters
that is used as data is called a *string*. When a string appears in the actual code of a program,
it is called a *string literal*. In Python code, string literals must be enclosed in quote marks.
As mentioned earlier, the quote marks simply mark where the string data begins and ends.
In Python, you can enclose string literals in a set of single-quote marks (') or a set of
double-
quote marks (").

Comments

CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the Python interpreter ignores them. They are intended for people who may be reading the source code.

Variables

CONCEPT: A variable is a name that represents a value stored in the computer's memory.

Variable Naming Rules

Although you are allowed to make up your own names for variables, you must follow these
rules:
• You cannot use one of Python's key words as a variable name. (See Table 1-2 for a
list of the key words.)
• A variable name cannot contain spaces.
• The first character must be one of the letters a through z, A through Z, or an underscore
character (_).
• After the first character you may use the letters a through z or A through Z, the digits
0 through 9, or underscores.
• Uppercase and lowercase characters are distinct. This means the variable name
ItemsOrdered is not the same as itemsordered.

NOTE: This style of naming is called camelCase because the uppercase characters
that appear in a name may suggest a camel's humps.

Variable Name Legal or Illegal?
Table 1:Sample variable names

Variable Name	Legal or Illegal?	Reason

units_per_day	Legal	
dayOfWeek	Legal	
3dGraph	Illegal.	Variable names cannot begin with a digit.
June1997	Legal	
Mixture#3	Illegal.	Variable names may only use letters, digits, or underscores.

Numeric Data Types and Literals

Python
uses *data types* to categorize values in memory. When an integer is stored in memory, it is classified as an int, and when a real number is stored in memory, it is classified as a float.
room = 503
dollars = 2.75

Storing Strings with the str Data Type

In addition to the int and float data types, Python also has a data type named str, which is used for storing strings in memory. The code in Program 2-11 shows how strings can be assigned to variables.

Reading Input from the Keyboard

CONCEPT: Programs commonly need to read input typed by the user on the keyboard. We will use the Python functions to do this.

Performing Calculations

CONCEPT: Python has numerous operators that can be used to perform mathematical calculations.

Python math operators

Symbol	Operation	Description
+	Addition	Adds
−	Subtraction	Subtracts
*	Multiplication	Multiplies
/	Division	Divides
a	floating-point	number
//	Integer	division
a	whole	number
%	Remainder	Divides
**	Exponent	Raises

Escape Character

Escape Character	Effect
\n	Causes output to be advanced to the next line.
\t	Causes output to skip over to the next horizontal tab position.
\'	Causes a single quote mark to be printed.
\"	Causes a double quote mark to be printed.

\\	Causes a backslash character to be printed.

Day 04-Core built-in data structures – Lists, Tuples, Dictionaries, Sets

A sequence is an object that holds multiple items of data, stored one after the other. You can perform operations on a sequence to examine and manipulate the items stored in it.

A *sequence* is an object that contains multiple items of data. The items that are in a sequence are stored one after the other. Python provides various ways to perform operations on the items that are stored in a sequence.

Introduction to Lists

A list is an object that contains multiple data items. Lists are mutable, which means that their contents can be changed during a program's execution. Lists are dynamic data structures, meaning that items may be added to them or removed from them. You can use indexing, slicing, and various methods to work with lists in a program.

even_numbers = [2, 4, 6, 8, 10]

Country = ["Pakistan", " Iran", "China", "Iraq"]

Lists Are Mutable

Lists in Python are *mutable*, which means their elements can be changed. Consequently, an expression in the form *list*[*index*] can appear on the left side of an assignment operator.

```
numbers = [1, 2, 3, 4, 5]
numbers[0] = 99
```

It will replace the first element with 99.

List Slicing

A slicing expression selects a range of elements from a sequence.

Copying Lists

To make a copy of a list, you must copy the list's elements.

```
list1 = [1, 2, 3, 4]
# Assign the list to the list2 variable.
list2 = list1
```

After this code executes, both variables list1 and list2 will reference the same list in Memory

One way to do this is with a loop that copies each element of the list.
Here is an example:

```
# Create a list with values.
list1 = [1, 2, 3, 4]
# Create an empty list.
list2 = []
# Copy the elements of list1 to list2.
for item in list1:
list2.append(item)
```

Tuples

A tuple is an immutable sequence, which means that its contents cannot be changed.

A *tuple* is a sequence, very much like a list. The primary difference between tuples and lists is that tuples are immutable. That means once a tuple is created, it cannot be changed. When you create a tuple, you enclose its elements in a set of parentheses, as shown in the following interactive session:

```
>>> my_tuple = (1, 2, 3, 4, 5)
```

In fact, tuples support all the same operations as lists, except those that change the contents of the list. Tuples support the following:

- Subscript indexing (for retrieving element values only)
- Methods such as index
- Built-in functions such as len, min, and max
- Slicing expressions
- The in operator
- The + and * operators

Tuples do not support methods such as append, remove, insert, reverse, and sort.

Dictionaries

A dictionary is an object that stores a collection of data. Each element in a dictionary has two parts: a key and a value. You use a key to locate a specific value.

Creating a Dictionary

You can create a dictionary by enclosing the elements inside a set of curly braces ({}). An element consists of a key, followed by a colon, followed by a value. The elements are separated by commas. The following statement shows an example:

phonebook = {'Chris':'555−1111', 'Katie':'555−2222', 'Joanne':'555−3333'}

This statement creates a dictionary and assigns it to the phonebook variable. The dictionary contains the following three elements:

- *The first element is 'Chris':'555−1111'. In this element, the key is 'Chris' and the value is '555−1111'.*
- *The second element is 'Katie':'555−2222'. In this element, the key is 'Katie' and the value is '555−2222'.*
- *The third element is 'Joanne':'555−3333'. In this element, the key is 'Joanne' and the value is '555−3333'.*

Retrieving a Value from a Dictionary

The elements in a dictionary are not stored in any particular order. For example, look at the following interactive session in which a dictionary is created and its elements are displayed:

```
>>> phonebook = {'Chris':'555−1111', 'Katie':'555−2222', 'Joanne':'555−3333'} Enter

>>> phonebook Enter

{'Chris': '555−1111', 'Joanne': '555−3333', 'Katie': '555−2222'}

>>>
```

Notice the order in which the elements are displayed is different than the order in which they were created. This illustrates how dictionaries are not sequences, like lists, tuples, and strings. As a result,

you cannot use a numeric index to retrieve a value by its position from a dictionary. Instead, you use a key to retrieve a value.

To retrieve a value from a dictionary, you simply write an expression in the following general format:

dictionary_name[key]

In the general format, dictionary_name is the variable that references the dictionary, and key is a key. If the key exists in the dictionary, the expression returns the value that is associated with the key. If the key does not exist, a KeyError exception is raised. The following interactive session demonstrates:

Using the in and not in Operators to Test for a Value in a Dictionary

As previously demonstrated, a KeyError exception is raised if you try to retrieve a value from a dictionary using a nonexistent key. To prevent such an exception, you can use the in operator to determine whether a key exists before you try to use it to retrieve a value. The following interactive session demonstrates:

```
1 >>> phonebook = {'Chris':'555−1111', 'Katie':'555−2222', 'Joanne':'555−3333'} Enter
2         >>> if 'Chris' in phonebook: Enter
3         print(phonebook['Chris']) Enter Enter
4
5 555−1111
6 >>>
```

Adding Elements to an Existing Dictionary

Dictionaries are mutable objects. You can add new key-value pairs to a dictionary with an assignment statement in the following general format:

dictionary_name[*key*] = *value*

Deleting Elements

You can delete an existing key-value pair from a dictionary with the del statement. Here is the general format:

del *dictionary_name*[*key*]

Some Dictionary Methods

Dictionary objects have several methods. In this section, we look at some of the more usefulones, which are summarized in Table 9−1.

Some of the dictionary methods

Method	Description
Clear	Clears the contents of a dictionary.
get	Gets the value associated with a specified key. If the key is not found, the methoddoes not raise an exception. Instead, it returns a default value.
items	Returns all the keys in a dictionary and their associated values as

a sequence oftuples.

keys Returns all the keys in a dictionary as a sequence of tuples.

pop Returns the value associated with a specified key and removes that key-value pair from the dictionary. If the key is not found, the method returns a default value.

popitem Returns a randomly selected key-value pair as a tuple from the dictionary andremoves that key-value pair from the dictionary.

values Returns all the values in the dictionary as a sequence of tuples.

The getMethod

You can use the get method as an alternative to the [] operator for getting a value froma dictionary. The get method does not raise an exception if the specified key is not found.Here is the method's general format:

dictionary.get(*key*, *default*)

Sets

A set contains a collection of unique values and works like a mathematical set.

A set is an object that stores a collection of data in the same way as mathematical sets. Here are some important things to know about sets:

- All the elements in a set must be unique. No two elements can have the same value.
- Sets are unordered, which means that the elements in a set are not stored in any particular order.
- The elements that are stored in a set can be of different data types.

Creating a Set

To create a set, you have to call the built-in set function. Here is an example of how you create an empty set:

myset = set()

myset = set('abc')

Finding the Union of Sets

The union of two sets is a set that contains all the elements of both sets.

set1.union(set2)

Finding the Intersection of Sets

The intersection of two sets is a set that contains only the elements that are found in both sets.

set1.intersection(set2)

other functions

set1.difference(set2) and equivalent *set1* − *set2*

set1.symmetric_difference(*set2*) and equivalent *set1* ˆ *set2*

set2.issubset(*set1*) and equivalen *set2* <= *set1*

set1.issuperset(*set2*) and equivalen *set1* >= *set2*

Lab activity -Sets

This lab activity needs to be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with # sign are comments in Python and are used to elaborate the code.

```
1	# This program demonstrates various set operations.
2	baseball = set(['Jodi', 'Carmen', 'Aida', 'Alicia'])
3	basketball = set(['Eva', 'Carmen', 'Alicia', 'Sarah']) 4
5	# Display members of the baseball set.
6	print('The following students are on the baseball team:')
7	for name in baseball:
8	print(name) 9
10	# Display members of the basketball set.
11	print()
12	print('The following students are on the basketball team:')
13	for name in basketball:
14	print(name) 15
16	# Demonstrate intersection
17	print()
18	print('The following students play both baseball and basketball:')
19	for name in baseball.intersection(basketball):
20	print(name) 21
22	# Demonstrate union
23	print()
24	print('The following students play either baseball or basketball:')
25	for name in baseball.union(basketball):
26	print(name) 27
28	# Demonstrate difference of baseball and basketball
29	print()
30	print('The following students play baseball, but not basketball:')
31	for name in baseball.difference(basketball):
32	print(name) 33
34	# Demonstrate difference of basketball and baseball
35	print()
36	print('The following students play basketball, but not baseball:')
37	for name in basketball.difference(baseball):
38	print(name) 39
40	# Demonstrate symmetric difference
41	print()
42	print('The following students play one sport, but not both:')
43	for name in baseball.symmetric_difference(basketball):
44	print(name)
```

Day-05: Decision Structures and Boolean Logic

The if Statement

CONCEPT: The if statement is used to create a decision structure, which allows a program to have more than one path of execution. The if statement causes one or more statements to execute only when a Boolean expression is true.

A control structure is a logical design that controls the order in which a set of statements execute. So far in this book, we have used only the simplest type of control structure: the sequence structure. A sequence structure is a set of statements that execute in the order in which they appear.

```
if condition:
    statement
    statement
    etc.
```

Code snippet

```
# This program gets three test scores and displays
# their average. It congratulates the user if the
# average is a high score.The HIGH_SCORE named constant holds the value that is

# considered a high score.

HIGH_SCORE = 95

test1 = int(input('Enter the score for test 1: ' ))
test2 = int(input('Enter the score for test 2: ' ))
test3 = int(input('Enter the score for test 3: ' ))
# Calculate the average test score.
average = (test1 + test2 + test3) / 3
# Print the average.
print('The average score is', average)
# If the average is a high score,
# congratulate the user.
if average >= HIGH_SCORE:
print('Congratulations!')
print('That is a great average!')
```

Boolean Expressions and Relational Operators

As previously mentioned, the if statement tests an expression to determine whether it is true or false. The expressions that are tested by the if statement are called *Boolean expres- sions*, named in honor of the English mathematician George Boole. In the 1800s, Boole invented a system of mathematics in which the abstract concepts of true and false can be used in computations.

Typically, the Boolean expression that is tested by an if statement is formed with a relational operator. A *relational operator* determines whether a specific relationship existsbetween two values. For example, the greater than operator (>) determines whether one value is greater than another. The equal to the operator (==) determines whether two values are equal.

The if-else Statement

CONCEPT: An if-else statement will execute one block of statements if its condition is true, or another block if its condition is false.

The previous section introduced the single alternative decision structure (the if statement), which has one alternative path of execution. Now, we will look at the dual alternative deci- sion structure, which

has two possible paths of execution—one path is taken if a condition is true, and the other path is taken if the condition is false.

Indentation in the if-else Statement

When you write an if-else statement, follow these guidelines for indentation:

- Make sure the if clause and the else clause are aligned.
- The if clause and the else clause are each followed by a block of statements. Make sure the statements in the blocks are consistently indented.

Nested Decision Structures and the

if-elif-else Statement

CONCEPT: To test more than one condition, a decision structure can be nested inside another decision structure.

Logical Operators

CONCEPT: The logical "and" operator and the logical "or" operator allow you to connect multiple Boolean expressions to create a compound expression. The logical "not" operator reverses the truth of a Boolean expression.

Boolean Variables

CONCEPT: A Boolean variable can reference one of two values: True or False.

Boolean variables are commonly used as flags, which indicate whether specific conditions exist.

Repetition Structures

CONCEPT: A repetition structure causes a statement or set of statements to executerepeatedly.

Condition-Controlled and Count-Controlled Loops

We will look at two broad categories of loops: condition-controlled and count-controlled. A condition-controlled loop uses a true/false condition to control the number of times that it repeats. A count-controlled loop repeats a specific number of times. In Python, you use the while statement to write a condition-controlled loop, and you use the for statement to write a count-controlled loop. In this chapter, we will demonstrate how to write both types of loops.

The while Loop: A Condition-Controlled Loop

A condition-controlled loop causes a statement or set of statements to repeat as long as a condition is true. In Python, you use the while state- ment to write a condition-controlled loop.

The while loop gets its name from the way it works: while a condition is true, do some task. The loop has two parts: (1) a condition that is tested for a true or false value, and (2) a statement or set of statements that is repeated as long as the condition is true.

Syntax:

```
while condition:
    statement
    statement
    etc.
```

The while Loop Is a Pretest Loop

The while loop is known as a pretest loop, which means it tests its condition before per- forming an iteration. Because the test is done at the beginning of the loop, you usually have to perform some steps prior to the loop to make sure that the loop executes at least once. For example,

```
while keep_going == 'y':
```

Infinite Loops

An infinite loop continues to repeat until the program is interrupted. Infinite loops usually occur when the programmer forgets to write code inside the loop, making the test condition false. In most circumstances, you should avoid writing infinite loops.

The for Loop: A Count-Controlled Loop

A count-controlled loop iterates a specific number of times. In Python, you use the for the statement to write a count-controlled loop.

As mentioned at the beginning of this chapter, a count-controlled loop iterates a specific number of times. Count-controlled loops are commonly used in programs. For example, suppose a business is open six days per week, and you are going to write a program that calculates the total sales for a week. You will need a loop that iterates exactly six times. Each time the loop iterates, it will prompt the user to enter the sales for one day.

You use the for statement to write a count-controlled loop. In Python, the for statement is designed to work with a sequence of data items. When the statement executes, it iterates once for each item in the sequence. Here is the general format:

for variable in [value1, value2, etc.]: statement

statement etc.

We will refer to the first line as the for clause. In the for clause, variable is the name of a variable. Inside the brackets a sequence of values appears, with a comma separating each value. (In Python, a comma-separated sequence of data items that are enclosed in a set of brackets is called a list.

Beginning at the next line is a block of statements that is executed each time the loop iterates.

The for statement executes in the following manner: The variable is assigned the first value in the list, then the statements that appear in the block are executed. Then, variable is assigned the next value in the list, and the statements in the block are executed again. This continues until variable has been assigned the last value in the list.

```
for x in range(5):
 print('Hello world')

 for num in range(1, 10, 2):

          print(num)
```

Calculating a Running Total

A running total is a sum of numbers accumulating with each loop iteration. The variable used to keep the running total is called an accumulator.

Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business’s total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable that is used to accumulate the total of the numbers is called an accumulator. It is often said that the loop keeps a running total because it accumulates the total as it reads each number in the series.

```
1  # This program calculates the sum of a
   series
2
3  MAX = 5 # The maximum number

4

5  # Initialize an accumulator
   variable.total = 0.0
6

7

8  # Explain what we are doing.

9  print('This program calculates the
   sum of')print(MAX, 'numbers you will
10 enter.')

11
```

Statement	What It Does	Value of xafter the Statement
x = x + 4	Add 4 to x	10
x = x − 3	Subtracts 3 from x	3
x = x * 10	Multiplies xby 10	60
x = x / 2	Divides xby 2	3
x = x % 4	Assigns the remainder of x / 4to x	2

Sentinels

A sentinel is a special value that marks the end of a sequence of values.

- Simply ask the user, at the end of each loop iteration, if there is another value to process. If the sequence of values is long, however, asking this question at the end of each loop iteration might make the program cumbersome for the user.

- Ask the user at the beginning of the program how many items are in the sequence. This might also inconvenience the user, however. If the sequence is very long, and the user does not know the number of items it contains, it will require the user to count them.

When processing a long sequence of values with a loop, perhaps a better technique is touse a sentinel. A *sentinel* is a special value that marks the end of a sequence of items. Whena program reads the sentinel value, it knows it has reached the end of the sequence, so the loop terminates.

Nested Loops

CONCEPT: A loop that is inside another loop is called a nested loop.

Example:

for hours in range(24):

for minutes in range(60):

for seconds in range(60):

print(hours, ':', minutes, ':', seconds)

Lab Activity: Nested Loops

This lab activity needs to be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with # sign are comments in Python and are used to elaborate the code.

```
1	# This program averages test scores. It asks the user for the
2	# number of students and the number of test scores per student. 3
4	# Get the number of students.
5	num_students = int(input('How many students do you have? ')) 6
7	# Get the number of test scores per student.
8	num_test_scores = int(input('How many test scores per student? ')) 9
10 # Determine each student's average test score.
11	for	student in range(num_students):
12		# Initialize an accumulator for test scores.
13		total = 0.0
14		# Get a student's test scores.
15		print('Student number', student + 1)
16		print('——————————')
17		for test_num in range(num_test_scores):
18		print('Test number', test_num + 1, end='')
19		score = float(input(': '))
20		# Add the score to the accumulator.
21		total += score
22	23	# Calculate the average test score for this student.
24	average = total / num_test_scores 25
26	# Display the average.
27	print('The average for student number', student + 1,
28	'is:', average)
print()
```

Week 2 -Data Manipulation and Cleaning

Day 01- Functions, Packages

A function is a group of statements that exist within a program for the purpose of performing a specific task.

Benefits of Modularizing a Program with Functions

A program benefits in the following ways when it is broken down into functions:

Simpler Code

A program's code tends to be simpler and easier to understand when it is broken down into functions. Several small functions are much easier to read than one long sequence of statements.

Code Reuse

Functions also reduce the duplication of code within a program. If a specific operation is performed in several places in a program, a function can be written once to perform thatoperation, then be executed any time it is needed. This benefit of using functions is known as *code reuse* because you are writing the code to perform a task once, then reusing it each time you need to perform the task.

Better Testing

When each task within a program is contained in its own function, testing and debugging becomes simpler. Programmers can test each function in a program individually, to determine whether it correctly performs its operation. This makes it easier to isolate and fix errors.

Faster Development

Suppose a programmer or a team of programmers is developing multiple programs. They discover that each of the programs perform several common tasks, such as asking for a username and a password, displaying the current time, and so on. It doesn't make sense to write the code for these tasks multiple times. Instead, functions can be written for the commonly needed tasks, and those functions can be incorporated into each program that needs them.

Easier Facilitation of Teamwork

Functions also make it easier for programmers to work in teams. When a program is developed as a set of functions that each performs an individual task, then different programmers can be assigned the job of writing different functions.

Void Functions and Value-Returning Functions

You will learn to write two types of functions: void functions and value- returning functions. When you call a *void function*, it simply executes the statements it contains and then terminates. When you call a *value-returning function*, it executes the statements that it contains, then returns a value back to the statement that called it. The input function is an example of a value-returning function. When you call the input func-tion, it gets the data that the user types on the keyboard and returns that data as a string. The int and float functions are also examples of value-returning functions. You pass an argument to the int function, and it returns that argument's value converted to an integer. Likewise, you pass an argument to the float function, and it returns that argument's value converted to a floating-point number.

The first type of function that you will learn to write is the void function.

Defining and Calling a Void Function

The code for a function is known as a function definition. To execute the function, you write a statement that calls it.

Function Names

Before we discuss the process of creating and using functions, we should mention a few things about function names. Just as you name the variables that you use in a program, you

also name the functions. A function's name should be descriptive enough so anyone reading your code can reasonably guess what the function does.
Python requires that you follow the same rules that you follow when naming variables, which we recap here:

- You cannot use one of Python's key words as a function name. (See Table 1-2 for a list of the key words.)
- A function name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (_).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct.

Because functions perform actions, most programmers prefer to use verbs in function names. For example, a function that calculates gross pay might be named calculate_gross_pay. This name would make it evident to anyone reading the code that the function calculates something. What does it calculate? The gross pay, of course. Other examples of good function names would be get_hours, get_pay_rate, calculate_overtime, print_check, and so on. Each function name describes what the function does.

Defining and Calling a Function

To create a function, you write its *definition*. Here is the general format of a function definition in Python:

def *function_name*():
statement
statement
etc.

The first line is known as the *function header*. It marks the beginning of the function definition. The function header begins with the keyword def, followed by the name of the function, followed by a set of parentheses, followed by a colon.
Beginning at the following line is a set of statements known as a block. A *block* is simply a set of statements that belong together as a group. These statements are performed any time the function is executed. Notice in the general format that all of the statements in the block are indented. This indentation is required, because the Python interpreter uses it to tell where the block begins and ends.
Let's look at an example of a function. Keep in mind that this is not a complete program. We will show the entire program in a moment.

```
def message():
print('I am Arthur,')
print('King of the Britons.')
```

This code defines a function named message. The message function contains a block with two statements. Executing the function will cause these statements to execute.

Calling a Function

A function definition specifies what a function does, but it does not cause the function to execute. To execute a function, you must *call* it. This is how we would call the message function:

```
message()
```

Local Variables

A local variable is created inside a function and cannot be accessed by statements that are outside the function. Different functions can have local variables with the same names because the functions cannot see each other's local variables.
Anytime you assign a value to a variable inside a function, you create a *local variable*. A

local variable belongs to the function in which it is created, and only statements inside that function can access the variable. (The term *local* is meant to indicate that the variable can be used only locally, within the function in which it is created.)

Scope and Local Variables

A variable's *scope* is the part of a program in which the variable may be accessed. A variable is visible only to statements in the variable's scope. A local variable's scope is the function in which the variable is created.

Passing Arguments to Functions

An argument is any piece of data that is passed into a function when the function is called. A parameter is a variable that receives an argument that is passed into a function.

Sometimes it is useful not only to call a function, but also to send one or more pieces of data into the function. Pieces of data that are sent into a function are known as *arguments*. The function can use its arguments in calculations or other operations.

Keyword Arguments

Python language allows you to write an argument in the following format, to specify which parameter variable the argument should be passed to:
parameter_name=value

Global Variables and Global Constants

A global variable is accessible to all the functions in a program file.

When a variable is created by an assignment statement that is written outside all the functions in a program file, the variable is *global*. A global variable can be accessed by any statement in the program file, including the statements in any function.

Global Constants

A *global constant* is a global name that references a value that cannot be changed. Because a global constant's value cannot be changed during the program's execution, you do not have to worry about many of the potential hazards that are associated with the use of global variables.

Although the Python language does not allow you to create true global constants, you can simulate them with global variables. If you do not declare a global variable with the global key word inside a function, then you cannot change the variable's assignment inside that function. For example global constant declaration as below
CONTRIBUTION_RATE = 0.05

Introduction to Value-Returning Functions: Generating Random Numbers

A value-returning function is a function that returns a value back to the part of the program that called it. Python, as well as most other programming languages, provides a library of prewritten functions that perform commonly needed tasks. These libraries typically contain a function that generates random numbers.

A *value-returning function* is a special type of function. It is like a void function in the following ways.
- It is a group of statements that perform a specific task.
- When you want to execute the function, you call it.

Standard Library Functions and the import Statement

Python, as well as most programming languages, comes with a *standard library* of functions that have already been written for you. These functions, known as *library functions,* make a programmer's job easier because they perform many of the tasks that programmers commonly need to perform. For example,

import math
This statement causes the interpreter to load the contents of the math module into memory and makes all the functions in the math module available to the program.

The following statement shows an example of how you might call the randint function from Math libarary:
number = random.randint (1, 100)

Writing Your Own Value-Returning Functions

A value-returning function has a return statement that returns a value back to the part of the program that called it.
You write a value-returning function in the same way that you write a void function, with one exception: a value-returning function must have a return statement. Here is the general format of a value-returning function definition in Python:

```
def function_name():
    statement
    statement
    etc.
    return expression
```

Returning Multiple Values

```
def get_name():
# Get the user's first and last names.
first = input('Enter your first name: ')
last = input('Enter your last name: ')
# Return both names.
return first, last
```

When you call this function in an assignment statement, you need to use two variables on the left side of the = operator. Here is an example:
first_name, last_name = get_name()

Symbol	Operation	Description
+	Addition	Adds two numbers
−	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the result asa floating-point number
//	Integer division	Divides one number by another and gives the result asa whole number
%	Remainder	Divides one number by another and gives the remainder
**	Exponent	Raises a number to a power

Day-02: String, built-in String methods, String Manipulation, and regular expressions

Basic String Operations

CONCEPT: Python provides several ways to access the individual characters in a string. Strings also have methods that allow you to perform operations on them.

Indexing

Another way that you can access the individual characters in a string is with an index. Each character in a string has an index that specifies its position in the string. Indexing starts at 0, so the index of the first character is 0, the index of the second character is 1, and so forth.

Strings Are Immutable

In Python, strings are immutable, which means once they are created, they cannot be changed. Some operations, such as concatenation, give the impression that they modify strings, but in reality, they do not.

String Slicing

CONCEPT: You can use slicing expressions to select a range of characters from a string
When you take a slice from a string, you get a span of characters from within the string. String slices are also called *substrings*.

To get a slice of a string, you write an expression in the following general format:
string[start : end]

Testing, Searching, and Manipulating Strings

CONCEPT: Python provides operators and methods for testing strings, searching the contents of strings, and getting modified copies of strings.

Testing Strings with 'in' and 'not in'

In Python, you can use the in operator to determine whether one string is contained in another string. Here is the general format of an expression using the in operator with two strings:
string1 in *string2*
string1 and *string2* can be either string literals or variables referencing strings. The expression returns true if *string1* is found in *string2*. For example, look at the following code:

```
text = 'Four score and seven years ago'
if 'seven' in text:
print('The string "seven" was found.')
else:
print('The string "seven" was not found.')
```

This code determines whether the string 'Four score and seven years ago' contains the string 'seven'. If we run this code, it will display:
The string "seven" was found.
You can use the not in operator to determine whether one string is *not* contained in another string. Here is an example:

```
names = 'Bill Joanne Susan Chris Juan Katie'
if 'Pierre' not in names:
print('Pierre was not found.')
else:
print('Pierre was found.')
```

If we run this code, it will display:
Pierre was not found.

String Methods

Recall from Chapter 6 that a method is a function that belongs to an object and performs some operation on that object. Strings in Python have numerous methods.1 In this section, we will discuss several string methods for performing the following types of operations:

- Testing the values of strings
- Performing various modifications
- Searching for substrings and replacing sequences of characters

String Modification Methods

Method	Description
lower()	Returns a copy of the string with all alphabetic letters converted to lower- case. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
lstrip()	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (\n), and tabs (\t) that appear at the beginning of the string.
lstrip(char)	The char argument is a string containing a character. Returns a copy of the string with all instances of char that appear at the beginning of the string removed.
rstrip()	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (\n), and tabs (\t) that appear at the end of the string.
rstrip(char)	The char argument is a string containing a character. The method returns a copy of the string with all instances of char that appear at the end of the string removed.
strip()	Returns a copy of the string with all leading and trailing whitespace characters removed.
strip(char)	Returns a copy of the string with all instances of char that appear at the beginning and the end of the string removed.
upper()	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

Lab Activtity- Python essentials

This lab activity needs to be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with # sign are comments in Python and are used to elaborate the code.

```
x = 1
y = 2
x + y
y
def add_numbers(x, y):
    return x + y
add_numbers(1, 2)
# `add_numbers` updated to take an optional 3rd parameter. Using `print` allows printing of multiple expressions within a single cell.
def add_numbers(x, y, z=None):
    if (z == None):
        return x + y
    else:
        return x + y + z
print(add_numbers(1, 2))
print(add_numbers(1, 2, 3))

def add_numbers(x, y, z=None, flag=False):
```

```
    if (flag):
        print('Flag is true!')
    if (z == None):
        return x + y
    else:
        return x + y + z


print(add_numbers(1, 2, flag=True))
def add_numbers(x, y):
    return x + y


a = add_numbers
a(1, 2)

type('This is a string')

type(None)

type(1)

type(1.0)

type(add_numbers)
x = (1, 'a', 2, 'b')
type(x)
# Lists are a mutable data structure.
x = [1, 'a', 2, 'b']
type(x)

x.append(3.3)
print(x)


# This is an example of how to loop through each item in the list.

for item in x:
    print(item)
# Or using the indexing operator:
i = 0
while (i != len(x)):
    print(x[i])
    i = i + 1
# Use `+` to concatenate lists.
[1, 2] + [3, 4]
# Use `*` to repeat lists.

[1] * 3
# Use the `in` operator to check if something is inside a list.
1 in [1, 2, 3]
```

```
# Now let's look at strings. Use bracket notation to slice a string.

x = 'This is a string'
print(x[0])  #first character
print(x[0:1])  #first character, but we have explicitly set the end character
print(x[0:2])  #first two characters
x[-1]


# This will return the slice starting from the 4th element from the end and stopping before the 2nd element from the end.
x[-4:-2]


# This is a slice from the beginning of the string and stopping before the 3rd element.
x[:3]
# And this is a slice starting from the 4th element of the string and going all the way to the end.
x[3:]
firstname = 'Christopher'
lastname = 'Brooks'
print(firstname + ' ' + lastname)
print(firstname * 3)
print('Chris' in firstname)
firstname = 'Christopher Arthur Hansen Brooks'.split(' ')[0]  # [0] selects the first element of the list
lastname = 'Christopher Arthur Hansen Brooks'.split(' ')[-1]  # [-1] selects the last element of the list
print(firstname)
print(lastname)
# Make sure you convert objects to strings before concatenating.
'Chris' + 2
'Chris' + str(2)
# Dictionaries associate keys with values.
x = {'Christopher Brooks': 'brooksch@umich.edu', 'Bill Gates': 'billg@microsoft.com'}
x['Christopher Brooks']  # Retrieve a value by using the indexing operator
x['Kevyn Collins-Thompson'] = None
x['Kevyn Collins-Thompson']
# Iterate over all of the keys:
for name in x:
    print(x[name])
# Iterate over all of the values:
for email in x.values():
    print(email)
# Iterate over all of the items in the list:
for name, email in x.items():
    print(name)
    print(email)
# You can unpack a sequence into different variables:
x = ('Christopher', 'Brooks', 'brooksch@umich.edu')
fname, lname, email = x
fname
lname
```

```
# Make sure the number of values you are unpacking matches the number of variables being assigned.
x = ('Christopher', 'Brooks', 'brooksch@umich.edu', 'Ann Arbor')
fname, lname, email = x


# # The Python Programming Language: More on Strings
print('Chris' + 2)
print('Chris' + str(2))
# Python has a built in method for convenient string formatting.
# In[33]:
sales_record = {
   'price': 3.24,
   'num_items': 4,
   'person': 'Chris'}

sales_statement = '{} bought {} item(s) at a price of {} each for a total of {}'

print(sales_statement.format(sales_record['person'],
                             sales_record['num_items'],
                             sales_record['price'],
                             sales_record['num_items'] * sales_record['price']))


import datetime as dt
import time as tm

# `time` returns the current time in seconds since the Epoch. (January 1st, 1970)
tm.time()
# Convert the timestamp to datetime.
# In[47]:
dtnow = dt.datetime.fromtimestamp(tm.time())
dtnow
# Handy datetime attributes:
# In[48]:
dtnow.year, dtnow.month, dtnow.day, dtnow.hour, dtnow.minute, dtnow.second  # get year, month, day, etc.from a datetime
# `timedelta` is a duration expressing the difference between two dates.
# In[49]:
delta = dt.timedelta(days=100)  # create a timedelta of 100 days
delta
# `date.today` returns the current local date.
# In[50]:
today = dt.date.today()
# In[51]:
today - delta  # the date 100 days ago
# In[52]:
today > today - delta  # compare dates
# # The Python Programming Language: Objects and map()
# An example of a class in python:
```

```
# In[54]:
class Person:
    department = 'School of Information'  #a class variable

    def set_name(self, new_name):  #a method
        self.name = new_name

    def set_location(self, new_location):
        self.location = new_location
person = Person()
person.set_name('Christopher Brooks')
person.set_location('Ann Arbor, MI, USA')
print('{} live in {} and works in the department {}'.format(person.name, person.location,
person.department))
# Here's an example of mapping the `min` function between two lists.
store1 = [10.00, 11.00, 12.34, 2.34]
store2 = [9.00, 11.10, 12.34, 2.01]
cheapest = map(min, store1, store2)
cheapest
# Now let's iterate through the map object to see the values.
for item in cheapest:
    print(item)

# # The Python Programming Language: Lambda and List Comprehensions
# Here's an example of lambda that takes in three parameters and adds the first two.
my_function = lambda a, b, c: a + b
# In[60]:
my_function(1, 2, 3)
# Let's iterate from 0 to 999 and return the even numbers.
my_list = []
for number in range(0, 1000):
    if number % 2 == 0:
        my_list.append(number)
my_list

my_list = [number for number in range(0, 1000) if number % 2 == 0]
my_list
```

Day 03- EXPORTING DATA USING PYTHON MODULES (numpy)

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas are built around the NumPy array. This sec-tion will present several examples using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays. While the types of operationsshown here may seem a bit dry and pedantic, they comprise the building blocks of many other examples used throughout the book. Get to know them well!

We'll cover a few categories of basic array manipulations here:

Attributes of arrays

Determining the size, shape, memory consumption, and data types of arrays

Indexing of arrays

Getting and setting the value of individual array elements

Slicing of arrays

Getting and setting smaller subarrays within a larger array

Reshaping of arrays

Changing the shape of a given array

Joining and splitting of arrays

Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays: a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

```
In[1]: import numpy as np
       np.random.seed(0) # seed for reproducibility

       x1 = np.random.randint(10, size=6)  # One-dimensional array
       x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
       x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes ndim (the number of dimensions), shape (the size of each dimension), and size(the total size of the array):

```
In[2]: print("x3 ndim: ", x3.ndim)
       print("x3 shape:", x3.shape)
       print("x3 size: ", x3.size)

x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute is the dtype, the data type of the array:

```
In[3]: print("dtype:", x3.dtype)

dtype: int64
```

Other attributes include itemsize, which lists the size (in bytes) of each array ele- ment, and nbytes, which lists the total size (in bytes) of the array:

```
In[4]: print("itemsize:", x3.itemsize, "bytes")
```

```
print("nbytes:", x3.nbytes, "bytes")
itemsize: 8 bytes
nbytes: 480 bytes
```

In general, we expect that nbytes is equal to itemsize times size.

Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, you can access the ith value (counting fromzero) by specifying the desired index in square brackets, just as with Python lists:

```
In[5]: x1

Out[5]:  array([5,  0,  3,  3,  7,  9])

In[6]: x1[0]

Out[6]:      5

In[7]:    x1[4]

Out[7]: 7
```

To index from the end of the array, you can use negative indices:

```
In[8]:    x1[-1]

Out[8]:        9

In[9]:    x1[-2]

Out[9]: 7
```

In a multidimensional array, you access items using a comma-separated tuple of indices:

```
In[10]: x2

Out[10]: array([[3, 5, 2, 4],
[7, 6, 8, 8],
[1, 6, 7, 7]])

In[11]: x2[0, 0]

Out[11]: 3

In[12]: x2[2, 0]

Out[12]: 1

In[13]: x2[2, -1]

Out[13]: 7
```

You can also modify values using any of the above index notation:

```
In[14]: x2[0, 0] = 12
        x2
```

```
Out[14]: array([[12,  5,  2,  4],
[ 7,                6,  8,  8],
[ 1,                6,  7,  7]])
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, thevalue will be silently truncated. Don't be caught unaware by this behavior!

```
In[15]: x1[0] = 3.14159  # this will be truncated!
        x1
Out[15]: array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also usethem to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice ofan array x, use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values start=0, stop=*size of dimension*, step=1. We'll take a look at accessing subarrays in one dimension and inmultiple dimensions.

One-dimensional subarrays

```
In[16]: x = np.arange(10)
        x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[17]: x[:5]  # first five elements
Out[17]: array([0, 1, 2, 3, 4])
In[18]: x[5:]  # elements after index 5
Out[18]: array([5, 6, 7, 8, 9])
In[19]: x[4:7]  # middle subarray
Out[19]: array([4, 5, 6])
In[20]: x[::2]  # every other element
Out[20]: array([0, 2, 4, 6, 8])
In[21]: x[1::2]  # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])
```

A potentially confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped. This becomes a convenient way to reverse an array:

```
In[22]: x[::-1]  # all elements, reversed
Out[22]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
In[23]: x[5::-2]  # reversed every other from index 5
```

```
Out[23]: array([5, 3, 1])
```

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by com-mas. For example:

```
In[24]: x2
```

```
Out[24]: array([[12,  5,  2,  4],
[ 7,                  6,  8,  8],
[ 1,                  6,  7,  7]])
```

```
In[25]: x2[:2, :3]  # two rows, three columns
```

```
Out[25]: array([[12,  5,  2],
[ 7,  6,  8]])
```

```
In[26]: x2[:3, ::2]  # all rows, every other column
```

```
Out[26]: array([[12,  2],
[ 7,            8],
[ 1,            7]])
```

Finally, subarray dimensions can even be reversed together:

```
In[27]: x2[::-1, ::-1]
```

```
Out[27]: array([[ 7,  7,  6,  1],
[ 8,                 8,  6,  7],
[ 4,                 2,  5, 12]])
```

Accessing array rows and columns. One commonly needed routine is accessing single rows or columns of an array. You can do this by combining indexing and slicing, using an empty slice marked by a single colon (:):

```
In[28]: print(x2[:, 0])  # first column of x2
```

```
[12  7  1]
```

```
In[29]: print(x2[0, :])  # first row of x2
```

```
[12  5  2  4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

```
In[30]: print(x2[0])  # equivalent to x2[0, :]
```

```
[12  5  2  4]
```

Subarrays as no-copy views

One important—and extremely useful—thing to know about array slices is that they return *views* rather than *copies* of the array data. This is one area in which NumPy array slicing

differs from Python list slicing: in lists, slices will be copies. Consider ourtwo-dimensional array from before:

```
In[31]: print(x2)

[[12  5  2   4]
 [ 7   6  8   8]
 [ 1   6  7   7]]
```

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]
        print(x2_sub)
[[12  5]
 [ 7  6]]
```

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
In[33]: x2_sub[0, 0] = 99
        print(x2_sub)
[[99  5]
 [ 7  6]]
In[34]: print(x2)

[[99  5  2   4]
 [ 7   6  8   8]
 [ 1   6  7   7]]
```

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the copy()method:

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
        print(x2_sub_copy)
[[99  5]
 [ 7  6]]
```

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
        print(x2_sub_copy)
[[42  5]
 [ 7  6]]
In[37]: print(x2)

[[99  5  2   4]
 [ 7   6  8   8]
```

[1 6 7 7]]

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the reshape() method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

```
In[38]: grid = np.arange(1, 10).reshape((3, 3))
        print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for this to work, the size of the initial array must match the size of the reshaped array. Where possible, the reshape method will use a no-copy view of the initial array, but with noncontiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a one-dimensional arrayinto a two-dimensional row or column matrix. You can do this with the reshape method, or more easily by making use of the newaxis keyword within a slice opera- tion:

```
In[39]: x = np.array([1, 2, 3])

# row vector via reshape
        x.reshape((1, 3))
Out[39]: array([[1, 2, 3]])
In[40]: # row vector via newaxis
        x[np.newaxis,     :]
Out[40]: array([[1, 2, 3]])
In[41]: # column vector via reshape
x.reshape((3, 1))
Out[41]: array([[1],
[2],
[3]])
In[42]: # column vector via newaxis
x[:, np.newaxis]
Out[42]: array([[1],
[2],
[3]])
```

We will see this type of transformation often throughout the remainder of the book.

Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look

at those operations here.

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines np.concatenate, np.vstack, and np.hstack. np.concatenate takes a tuple or list of arrays as its first argument, as we can see here:

```
In[43]: x = np.array([1, 2, 3])
        y = np.array([3, 2, 1])
        np.concatenate([x, y])
Out[43]: array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
In[44]: z = [99, 99, 99]
        print(np.concatenate([x, y, z]))[
1 2 3 3 2 1 99 99 99]
```

np.concatenatecan also be used for two-dimensional arrays:

```
In[45]: grid = np.array([[1, 2, 3],
[4, 5, 6]])
In[46]:# concatenate along the first axis
np.concatenate([grid, grid])
Out[46]: array([[1, 2, 3],
[4, 5, 6],
[1, 2, 3],
[4, 5, 6]])
In[47]: # concatenate along the second axis (zero-indexed)
        np.concatenate([grid, grid], axis=1)
Out[47]: array([[1, 2, 3, 1, 2, 3],
[4, 5, 6, 4, 5, 6]])
```

For working with arrays of mixed dimensions, it can be clearer to use the np.vstack (vertical stack) and np.hstack(horizontal stack) functions:

```
In[48]: x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
[6, 5, 4]])

        # vertically stack the arrays
        np.vstack([x, grid])
Out[48]: array([[1, 2, 3],
[9, 8, 7],
[6, 5, 4]])
```

```
In[49]: # horizontally stack the arrays
y = np.array([[99],
[99]])
np.hstack([grid, y])
Out[49]: array([[ 9,  8,  7, 99],
[ 6,  5,  4, 99]])
```

Similarly, np.dstackwill stack arrays along the third axis.

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

```
In[50]: x = [1, 2, 3, 99, 99, 3, 2, 1]
        x1, x2, x3 = np.split(x, [3, 5])
        print(x1,  x2,  x3)
[1 2 3] [99 99] [3 2 1]
```

Notice that *N* split points lead to *N + 1* subarrays. The related functions np.hsplit and np.vsplitare similar:

```
In[51]: grid = np.arange(16).reshape((4, 4))
        grid

Out[51]: array([[ 0,  1,  2,  3],
[ 4,                  5,  6,  7],
[ 8,                  9, 10, 11],
[12,                  13, 14, 15]])
In[52]: upper, lower = np.vsplit(grid, [2])
        print(upper)
        print(lower)
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
In[53]: left, right = np.hsplit(grid, [2])
        print(left)
        print(right)

[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
```

```
[10 11]
[14 15]]
```

Similarly, np.dsplitwill split arrays along the third axis.

Computation on NumPy Arrays: Universal Functions

Up until now, we have been discussing some of the basic nuts and bolts of NumPy; inthe next few sections, we will dive into the reasons that NumPy is so important in thePython data science world. Namely, it provides an easy and flexible interface to opti-mized computation with arrays of data.

Computation on NumPy arrays can be very fast, or it can be very slow. The key tomaking it fast is to use *vectorized* operations, generally implemented through Num- Py's *universal functions* (ufuncs). This section motivates the need for NumPy's ufuncs,which can be used to make repeated calculations on array elements much more effi- cient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package.

Array arithmetic

NumPy's ufuncs feel very natural to use because they make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used:

```
In[7]: x = np.arange(4)
       print("x      =",        x)
       print("x + 5 =", x + 5)
       print("x - 5 =", x - 5)
       print("x * 2 =", x * 2)
       print("x / 2 =", x / 2)
       print("x // 2 =", x // 2)  # floor division
x      = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.     0.5  1.    1.5]
x // 2 = [0 0 1 1]
```

There is also a unary ufunc for negation, a ** operator for exponentiation, and a % operator for modulus:

```
In[8]: print("-x      =     ",      -x)
       print("x ** 2 = ", x ** 2)
       print("x % 2 = ", x % 2)
-x      =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

In addition, these can be strung together however you wish, and the standard orderof operations is respected:

```
In[9]: -(0.5*x + 1) ** 2
```

```
Out[9]: array([-1.  , -2.25, -4.  , -6.25])
```

All of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the + operator is a wrapper for the add function:

```
In[10]: np.add(x, 2)

Out[10]: array([2, 3, 4, 5])
```

Table . Arithmetic operators implemented in NumPy

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

```
In[11]: x = np.array([-2, -1, 0, 1, 2])
        abs(x)

Out[11]: array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs:

```
In[12]: np.absolute(x)

Out[12]: array([2, 1, 0, 1, 2])

In[13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])
```

This ufunc can also handle complex data, in which the absolute value returns the magnitude:

```
In[14]: x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
        np.abs(x)

Out[14]: array([ 5.,  5.,  2.,  1.])
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for

thedata scientist are the trigonometric functions. We'll start by defining an array of angles:

```
In[15]: theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

```
In[16]: print("theta      =     ",     theta)
        print("sin(theta)  =  ",  np.sin(theta))
        print("cos(theta)  =  ", np.cos(theta))
        print("tan(theta) = ", np.tan(theta))

theta      = [ 0.           1.57079633  3.14159265]
sin(theta) = [  0.00000000e+  1.00000000e   1.22464680e-
                00            +00           16]
cos(theta) = [  1.00000000e+  6.12323400e  -
                00            -17          1.00000000e+0
                                           0]
tan(theta) = [  0.00000000e+  1.63312394e  -1.22464680e-
                00            +16          16]
```

The values are computed to within machine precision, which is why values that should be zero do not always hit exactly zero. Inverse trigonometric functions are also available:

```
In[17]: x = [-1, 0, 1]
        print("x          = ", x)
        print("arcsin(x) = ", np.arcsin(x))
        print("arccos(x) = ", np.arccos(x))
        print("arctan(x) = ", np.arctan(x))

x          = [-1, 0, 1]
arcsin(x) = [-           0.          1.5707963
             1.57079633              3]
arccos(x)= [             1.570796  0.          ]
             3.14159265  33
arctan(x)= [-            0.          0.7853981
             0.78539816              6]
```

Exponents and logarithms

Another common type of operation available in a NumPy ufunc are the exponentials:

```
In[18]: x = [1, 2, 3]
        print("x     =", x)
        print("e^x   =", np.exp(x))
        print("2^x   =", np.exp2(x))
        print("3^x   =", np.power(3, x))

x    = [1, 2, 3]
e^x  = [  2.71828183  7.389056   20.0855369
                      1          2]
2^x  = [ 2.   4.   8.]
3^x  = [ 3   9 27]
```

The inverse of the exponentials, the logarithms, are also available. The basic np.log gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```
In[19]: x = [1, 2, 4, 10]
        print("x        =", x)
        print("ln(x)    =", np.log(x))
        print("log2(x)  =", np.log2(x))
        print("log10(x) =", np.log10(x))
```

```
x        = [1, 2, 4, 10]
ln(x)    = [ 0.          0.69314718  1.38629436  2.30258509]
log2(x)  = [ 0.          1.          2.          3.32192809]
log10(x) = [ 0.          0.30103     0.60205999  1.        ]
```

There are also some specialized versions that are useful for maintaining precision with very small input:

```
In[20]: x = [0, 0.001, 0.01, 0.1]
        print("exp(x) - 1 =", np.expm1(x))
        print("log(1 + x) =", np.log1p(x))
```

```
exp(x) - 1 = [ 0.          0.0010005   0.01005017  0.10517092]
log(1 + x) = [ 0.          0.0009995   0.00995033  0.09531018]
```

When x is very small, these functions give more precise values than if the raw np.log or np.exp were used.

Specialized ufuncs

NumPy has many more ufuncs available, including hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. A look through the NumPy documentation reveals a lot of interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the submodule scipy.special. If you want to compute some obscure mathematical function onyour data, chances are it is implemented in scipy.special. There are far too many functions to list them all, but the following snippet shows a couple that might come up in a statistics context:

```
In[21]: from scipy import special
```

```
In[22]: # Gamma functions (generalized factorials) and related functions
        x = [1, 5, 10]
        print("gamma(x)     =", special.gamma(x))
        print("ln|gamma(x)| =", special.gammaln(x))
        print("beta(x, 2)   =", special.beta(x, 2))
```

```
gamma(x)    = [  1.00000000e+002.40000000e+01
                 3.62880000e+05]ln|gamma(x)| = [  0.  3.17805383
12.80182748]

beta(x, 2)    = [ 0.5            0.03333333   0.00909091]
```

```
In[23]: # Error function (integral of
        Gaussian)# its complement, and its
        inverse

        x = np.array([0, 0.3, 0.7, 1.0])

        print("erf(x)  =", special.erf(x))

        print("erfc(x) =", special.erfc(x))

        print("erfinv(x) =", special.erfinv(x))
```

```
erf(x)  = [ 0.           0.32862676  0.67780119  0.84270079]

erfc(x) = [ 1.           0.67137324  0.32219881  0.15729921]

erfinv(x) = [ 0.          0.27246271  0.73286908       inf]
```

There are many, many more ufuncs available in both NumPy and scipy.special. Because the documentation of these packages is available online, a web search along the lines of "gamma function python" will generally find the relevant information.

Advanced Ufunc Features

Many NumPy users make use of ufuncs without ever learning their full set of features.We'll outline a few specialized features of ufuncs here.

Specifying output

For large calculations, it is sometimes useful to be able to specify the array where theresult of the calculation will be stored. Rather than creating a temporary array, you can use this to write computation results directly to the memory location where you'd like them to be. For all ufuncs, you can do this using the out argument of thefunction:

```
In[24]: x = np.arange(5)

        y = np.empty(5)
        np.multiply(x, 10, out=y)
        print(y)
```

```
[  0.  10.  20.  30.  40.]
```

This can even be used with array views. For example, we can write the results of a computation to every other element of a specified array:

```
In[25]: y = np.zeros(10)

        np.power(2, x, out=y[::2])

        print(y)
```

```
[  1.    0.    2.    0.    4.    0.    8.    0.  16.    0.]
```

If we had instead written y[::2] = 2 ** x, this would have resulted in the creation of a temporary array to hold the results of 2 ** x, followed by a second operation copying those values into the yarray. This doesn't make much of a difference for sucha small computation, but for very large arrays the memory savings from careful use ofthe outargument can be significant.

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to *reduce* an array with a particularoperation, we can use the reducemethod of any ufunc. A reduce repeatedly applies agiven operation to the elements of an array until only a single result remains.

For example, calling reduce on the add ufunc returns the sum of all elements in thearray:

```
In[26]: x = np.arange(1, 6)
        np.add.reduce(x)

Out[26]: 15
```

Similarly, calling reduce on the multiply ufunc results in the product of all array elements:

```
In[27]: np.multiply.reduce(x)

Out[27]: 120
```

If we'd like to store all the intermediate results of the computation, we can instead use accumulate:

```
In[28]:         np.add.accumulate(x)

Out[28]: array([ 1,  3,  6, 10, 15])
```

Aggregations: Min, Max, and Everything in Between

Often when you are faced with a large amount of data, a first step is to compute sum-mary statistics for the data in question. Perhaps the most common summary statistics are the mean and standard deviation, which allow you to summarize the "typical" val-ues in a dataset, but other aggregates are useful as well (the sum, product, median, minimum and maximum, quantiles, etc.).

NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

Summing the Values in an Array

As a quick example, consider computing the sum of all values in an array. Python itself can do this using the built-in sumfunction:

```
In[1]: import numpy as np

In[2]:              L              =
       np.random.random(100)
       sum(L)

Out[2]: 55.61209116604941
```

The syntax is quite similar to that of NumPy's sumfunction, and the result is the samein the simplest case:

```
In[3]: np.sum(L)

Out[3]: 55.612091166049424
```

However, because it executes the operation in compiled code, NumPy's version of the

operation is computed much more quickly:

```
In[4]: big_array = np.random.rand(1000000)
       %timeit sum(big_array)
       %timeit np.sum(big_array)
```

```
10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

Be careful, though: the sumfunction and the np.sumfunction are not identical, whichcan sometimes lead to confusion! In particular, their optional arguments have differ- ent meanings, and np.sumis aware of multiple array dimensions, as we will see in thefollowing section.

Minimum and Maximum

Similarly, Python has built-in minand maxfunctions, used to find the minimum valueand maximum value of any given array:

```
In[5]: min(big_array), max(big_array)
```

```
Out[5]: (1.1717128136634614e-06, 0.9999976784968716)
```

NumPy's corresponding functions have similar syntax, and again operate much morequickly:

```
In[6]: np.min(big_array), np.max(big_array)
```

```
Out[6]: (1.1717128136634614e-06, 0.9999976784968716)
```

```
In[7]: %timeit min(big_array)
       %timeit np.min(big_array)
```

```
10 loops, best of 3: 82.3 ms per loop
1000 loops, best of 3: 497 µs per loop
```

For min, max, sum, and several other NumPy aggregates, a shorter syntax is to use methods of the array object itself:

```
In[8]: print(big_array.min(), big_array.max(), big_array.sum())
```

```
1.17171281366e-06 0.999997678497 499911.628197
```

Whenever possible, make sure that you are using the NumPy version of these aggre- gates when operating on NumPy arrays!

Multidimensional aggregates

One common type of aggregation operation is an aggregate along a row or column. Say you have some data stored in a two-dimensional array:

```
In[9]: M = np.random.random((3, 4))
       print(M)
```

```
[[ 0.8967576   0.03783739  0.75952519  0.06682827]
 [ 0.8354065   0.99196818  0.19544769  0.43447084]
 [ 0.66859307  0.15038721  0.37911423  0.6687194 ]]
```

By default, each NumPy aggregation function will return the aggregate over the entirearray:

```
In[10]: M.sum()

Out[10]: 6.0850555667307118
```

Aggregation functions take an additional argument specifying the *axis* along which the aggregate is computed. For example, we can find the minimum value within eachcolumn by specifying axis=0:

```
In[11]: M.min(axis=0)

Out[11]: array([ 0.66859307,  0.03783739,  0.19544769,  0.06682827])
```

The function returns four values, corresponding to the four columns of numbers.Similarly,

we can find the maximum value within each row:

```
In[12]: M.max(axis=1)

Out[12]: array([ 0.8967576 ,  0.99196818,  0.6687194 ])
```

The way the axis is specified here can be confusing to users coming from other lan- guages. The axis keyword specifies the *dimension of the array that will be collapsed*, rather than the dimension that will be returned. So specifying axis=0 means that the first axis will be collapsed: for two-dimensional arrays, this means that values within each column will be aggregated.

Table. Aggregation functions available in NumPy

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute median of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance
np.min	np.nanmin	Find minimum value
np.max	np.nanmax	Find maximum value
np.argmin	np.nanargmin	Find index of minimum value
np.argmax	np.nanargmax	Find index of maximum value
np.median	np.nanmedian	Compute median of elements
np.percentile	np.nanpercentile	Compute rank-based statistics of elements
np.any	N/A	Evaluate whether any elements are true
np.all	N/A	Evaluate whether all elements are true

We will see these aggregates often throughout the rest of the book.

Example: What Is the Average Height of US Presidents?

Aggregates available in NumPy can be extremely useful for summarizing a set of val-ues. As a simple example, let's consider the heights of all US presidents. This data is available in the file *president_heights.csv*, which is a simple comma-separated list of labels and values:

```
In[13]: !head -4
data/president_heights.csv
order,name,height(cm)
1,George Washington,189
2,John Adams,170
3,Thomas
Jefferson,189
```

We'll use the Pandas package, which we'll explore more fully in Chapter 3, to read thefile and extract this information (note that the heights are measured in centimeters):

```
In[14]: import pandas as pd
        data =
        pd.read_csv('data/president_heights.csv')
        heights = np.array(data['height(cm)'])
        print(heights)

[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178 173
 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193 182 183
 177 185 188 188 182 185]
```

Now that we have this data array, we can compute a variety of summary statistics:

```
In[15]: print("Mean height:       ", heights.mean())
        print("Standard deviation:", heights.std())
        print("Minimum height:    ", heights.min())
        print("Maximum height:", heights.max())

Mean height:    179.738095238
Standard deviation: 6.93184344275
Minimum height: 163
Maximum height: 193
```

Note that in each case, the aggregation operation reduced the entire array to a single summarizing value, which gives us information about the distribution of values. We may also wish to compute quantiles:

```
In[16]: print("25th percentile:   ", np.percentile(heights, 25))
        print("Median:            ", np.median(heights))
        print("75th percentile:   ", np.percentile(heights, 75))

25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

We see that the median height of US presidents is 182 cm, or just shy of six feet.

Of course, sometimes it's more useful to see a visual representation of this data, whichwe

can accomplish using tools in Matplotlib (we'll discuss Matplotlib more fully in Chapter 4). For example, this code generates the chart shown in Figure 2-3:

```
In[17]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # set plot style
In[18]: plt.hist(heights)
        plt.title('Height  Distribution  of  US  Presidents')
        plt.xlabel('height (cm)')
        plt.ylabel('number');
```

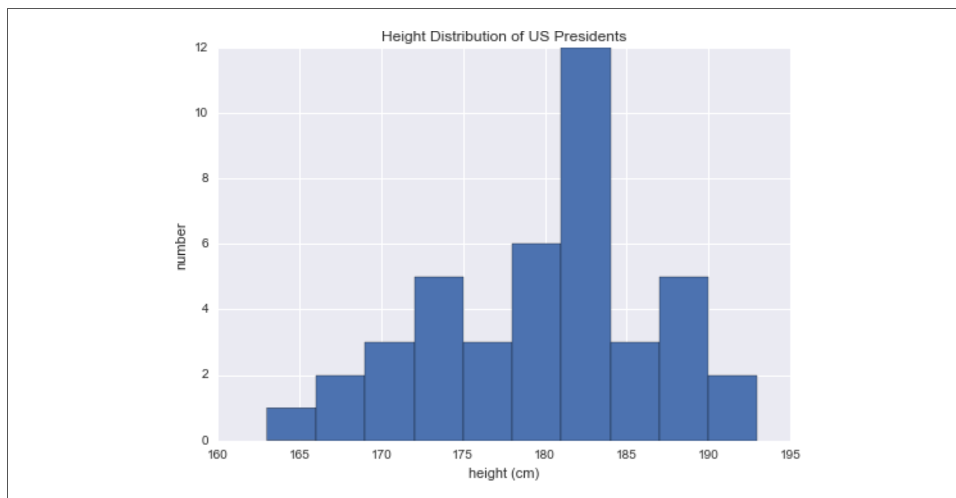


Figure 1:Histogram of presidential heights

Computation on Arrays: Broadcasting

We saw in the previous section how NumPy's universal functions can be used to *vec-torize* operations and thereby remove slow Python loops. Another means of vectoriz- ing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply aset of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

Introducing Broadcasting

Recall that for arrays of the same size, binary operations are performed on an element-by-element basis:

```
In[1]: import numpy as np
In[2]: a = np.array([0, 1, 2])
       b = np.array([5, 5, 5])a +
       b
Out[2]: array([5, 6, 7])
```

Broadcasting allows these types of binary operations to be performed on arrays of dif-ferent sizes—for example, we can just as easily add a scalar (think of it as a zero- dimensional array) to an array:

```
In[3]: a + 5
```

```
Out[3]: array([5, 6, 7])
```

We can think of this as an operation that stretches or duplicates the value 5 into the array [5, 5, 5], and adds the results. The advantage of NumPy's broadcasting is thatthis duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

We can similarly extend this to arrays of higher dimension. Observe the result when we add a one-dimensional array to a two-dimensional array:

```
In[4]: M = np.ones((3, 3))M

Out[4]: array([[ 1.,   1.,   1.],
[ 1.,               1.,   1.],
[ 1.,               1.,   1.]])
In[5]: M + a
Out[5]: array([[ 1.,   2.,   3.],
[ 1.,               2.,   3.],
[ 1.,               2.,   3.]])
```

Here the one-dimensional array a is stretched, or broadcast, across the second dimension in order to match the shape of M.

While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
In[6]: a = np.arange(3)
       b = np.arange(3)[:, np.newaxis]

       print(a)
       print(b)
[0 1 2]
[[0]
 [1]
 [2]]
In[7]: a + b
Out[7]: array([[0, 1, 2],
[1, 2, 3],
[2, 3, 4]])
```

Just as before we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* a and b to match a common shape, and the result is a two-dimensional array!

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an *N* by *D* matrix representing *N* points in *D* dimen- sions, such as the following points drawn from a two-dimensional normal distribu- tion:

```
In[13]: mean = [0, 0]
cov = [[1, 2],
```

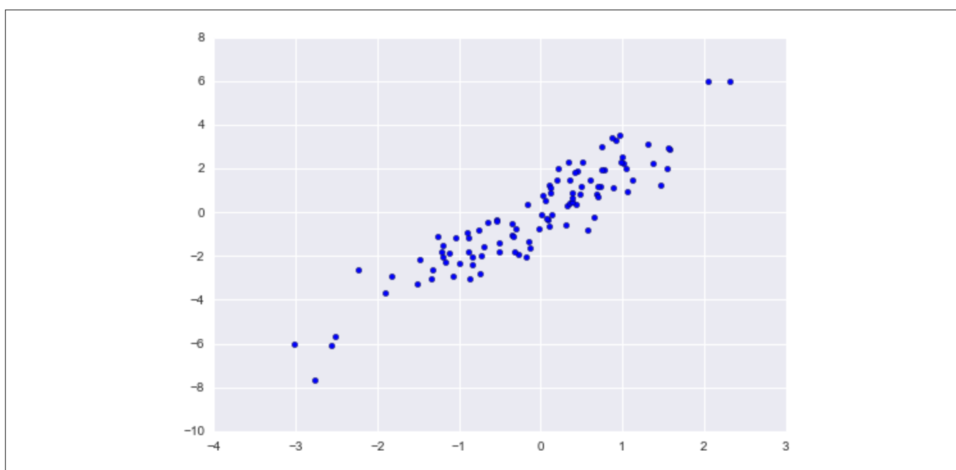
[2, 5]]

```
X = rand.multivariate_normal(mean, cov,
100)X.shape
```

Out[13]: (100, 2)

Using the plotting tools we will discuss in Chapter 4, we can visualize these points asa scatter plot (Figure 2-7):

```
In[14]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # for plot styling

        plt.scatter(X[:, 0], X[:, 1]);
```



Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20random indices with no repeats, and use these indices to select a portion of the origi- nal array:

```
In[15]: indices = np.random.choice(X.shape[0], 20,
        replace=False)indices
```

```
Out[15]: array([93, 45, 73, 81, 50, 10, 98, 94,  4, 64, 65, 89, 47, 84, 82,
         80, 25, 90, 63, 20])
```

```
In[16]: selection = X[indices]  # fancy indexing here
        selection.shape
```

Out[16]: (20, 2)

Now to see which points were selected, let's over-plot large circles at the locations ofthe selected points

```
In[17]: plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
```

```
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', s=200);
```

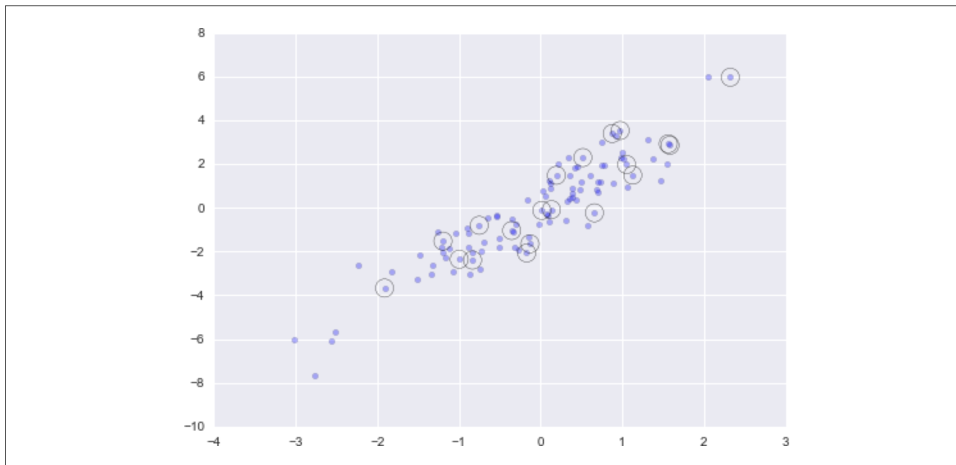


Figure . Random selection among points

This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models and in sampling approaches to answering statistical questions.

Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array. For example, imagine we have an array of indices and we'dlike to set the corresponding items in an array to some value:

```
In[18]: x = np.arange(10)
        i = np.array([2, 1, 8, 4])
        x[i] = 99
        print(x)

[ 0 99 99  3 99  5  6  7 99  9]
```

We can use any assignment-type operator for this. For example:

```
In[19]: x[i] -= 10
        print(x)

[ 0 89 89  3 89  5  6  7 89  9]
```

Notice, though, that repeated indices with these operations can cause some potentially unexpected results. Consider the following:

```
In[20]: x = np.zeros(10)
        x[[0, 0]] = [4, 6]
        print(x)

[ 6.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

Where did the 4 go? The result of this operation is to first assign x[0] = 4, followed by x[0] = 6. The result, of course, is that x[0]contains the value 6.

Fair enough, but consider this operation:

```
In[21]: i = [2, 3, 3, 4, 4, 4]
```

```
    x[i] += 1
    x
Out[21]: array([ 6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

You might expect that x[3] would contain the value 2, and x[4] would contain the value 3, as this is how many times each index is repeated. Why is this not the case? Conceptually, this is because x[i] += 1is meant as a shorthand of x[i] = x[i] + 1.x[i] + 1 is evaluated, and then the result is assigned to the indices in x. With this inmind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather nonintuitive results.

So what if you want the other behavior where the operation is repeated? For this, youcan use the at()method of ufuncs (available since NumPy 1.8), and do the following:

```
In[22]: x = np.zeros(10)
    np.add.at(x, i, 1)
    print(x)
[ 0.  0.  1.  2.  3.  0.  0.  0.  0.  0.]
```

The at() method does an in-place application of the given operator at the specified indices (here, i) with the specified value (here, 1). Another method that is similar in spirit is the reduceat() method of ufuncs, which you can read about in the NumPy documentation.

Example: Binning Data

You can use these ideas to efficiently bin data to create a histogram by hand. For example, imagine we have 1,000 values and would like to quickly find where they fallwithin an array of bins. We could compute it using ufunc.atlike this:

```
In[23]: np.random.seed(42)
    x = np.random.randn(100)

    # compute a histogram by
    hand bins = np.linspace(-5, 5,
    20)         counts          =
    np.zeros_like(bins)

    # find the appropriate bin for each x
    i = np.searchsorted(bins, x)

    # add 1 to each of these bins
    np.add.at(counts, i, 1)
```

The counts now reflect the number of points within each bin—in other words, a his-togram

```
In[24]: # plot the results
```

```
plt.plot(bins, counts, linestyle='steps');
```

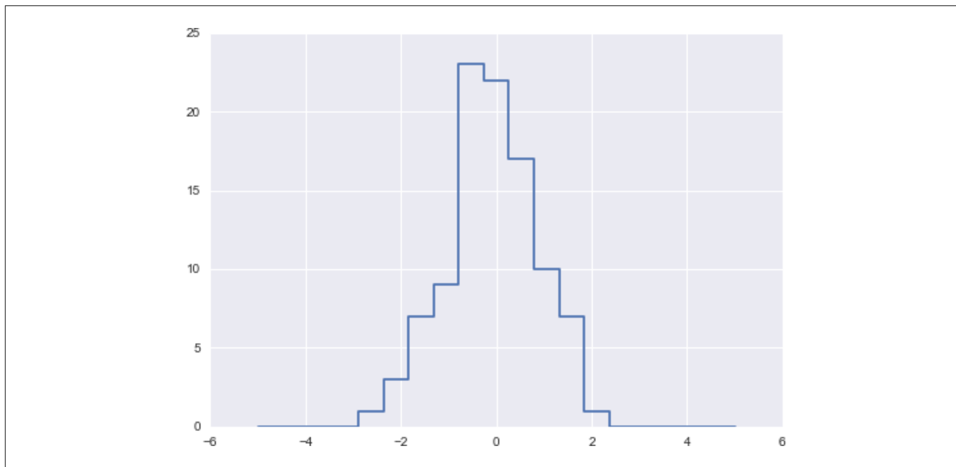


Figure. A histogram computed by hand

Of course, it would be silly to have to do this each time you want to plot a histogram.This is why Matplotlib provides the plt.hist() routine, which does the same in a single line:

```
plt.hist(x, bins, histtype='step');
```

This function will create a nearly identical plot to the one seen here. To compute the binning, Matplotlib uses the np.histogram function, which does a very similar com-putation to what we did before. Let's compare the two here:

```
In[25]: print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)
        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
10000 loops, best of 3: 97.6 µs per loop
Custom routine:
10000 loops, best of 3: 19.5 µs per loop
```

Our own one-line algorithm is several times faster than the optimized algorithm in NumPy! How can this be? If you dig into the np.histogram source code (you can do this in IPython by typing **np.histogram??**), you'll see that it's quite a bit more involved than the simple search-and-count that we've done; this is because NumPy's algorithm is more flexible, and particularly is designed for better performance when the number of data points becomes large:

```
In[26]: x = np.random.randn(1000000)
        print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)

        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
```

```
10 loops, best of 3: 68.7 ms per loop
Custom routine:

10 loops, best of 3: 135 ms per loop
```

Sorting Arrays

Up to this point we have been concerned mainly with tools to access and operate on array data with NumPy. This section covers algorithms related to sorting values in NumPy arrays. These algorithms are a favorite topic in introductory computer sci- ence courses: if you've ever taken one, you probably have had dreams (or, depending on your temperament, nightmares) about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more. All are means of accomplishing a similar task: sorting the values in a list or array.

For example, a simple *selection sort* repeatedly finds the minimum value from a list, and makes swaps until the list is sorted. We can code this in just a few lines of Python:

```
In[1]: import numpy as np


        def selection_sort(x):
            for i in range(len(x)):
                swap = i + np.argmin(x[i:]) (x[i],
                x[swap]) = (x[swap], x[i])
            return x
```

```
In[2]: x = np.array([2, 1, 4, 3, 5])
       selection_sort(x)

Out[2]: array([1, 2, 3, 4, 5])
```

Fortunately, Python contains built-in sorting algorithms that are *much* more efficient than either of the simplistic algorithms just shown. We'll start by looking at the Python built-ins, and then take a look at the routines included in NumPy and opti- mized for NumPy arrays.

Fast Sorting in NumPy: np.sort and np.argsort

Although Python has built-in sortand sortedfunctions to work with lists, we won't discuss them here because NumPy's np.sort function turns out to be much more efficient and useful for our purposes. By default np.sort uses an $N \log N$, *quick- sort* algorithm, though *mergesort* and *heapsort* are also available. For most applica- tions, the default quicksort is more than sufficient.

To return a sorted version of the array without modifying the input, you can use np.sort:

```
In[5]: x = np.array([2, 1, 4, 3, 5])
       np.sort(x)

Out[5]: array([1, 2, 3, 4, 5])
```

If you prefer to sort the array in-place, you can instead use the sortmethod of arrays:

```
In[6]: x.sort()
```

```
print(x)
```

[1 2 3 4 5]

A related function is argsort, which instead returns the *indices* of the sorted elements:

```
In[7]: x = np.array([2, 1, 4, 3, 5])i =
       np.argsort(x)
       print(i)
```

[1 0 3 2 4]

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used (via fancy indexing) to construct the sorted array if desired:

```
In[8]: x[i]
```

Out[8]: array([1, 2, 3, 4, 5])

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axisargument. For example:

```
In[9]:          rand          =
       np.random.RandomState(42)X =
       rand.randint(0,   10,   (4,   6))
       print(X)
```

[[6 3 7 4 6 9]

[2 6 7 4 3 7]

[7 2 5 4 1 7]

[5 1 4 0 9 5]]

```
In[10]: # sort each column of X
        np.sort(X, axis=0)
```

Out[10]: array([[2, 1, 4, 0, 1, 5],

[5, 2, 5, 4, 3, 7],

[6, 3, 7, 4, 6, 7],

[7, 6, 7, 4, 9, 9]])

```
In[11]: # sort each row of X
        np.sort(X, axis=1)
```

Out[11]: array([[3, 4, 6, 6, 7, 9],

[2, 3, 4, 6, 7, 7],

[1, 2, 4, 5, 7, 7],

[0, 1, 4, 5, 5, 9]])

Keep in mind that this treats each row or column as an independent array, and any

relationships between the row or column values will be lost!

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the*K* smallest values in the array. NumPy provides this in the np.partition function. np.partition takes an array and a number *K*; the result is a new array with the small-est *K* values to the left of the partition, and the remaining values to the right, in arbi- trary order:

```
In[12]: x = np.array([7, 2, 3, 1, 6, 5, 4])
        np.partition(x, 3)
Out[12]: array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in thearray, and the remaining array positions contain the remaining values. Within thetwo partitions, the elements have arbitrary order.

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

```
In[13]: np.partition(X, 2, axis=1)
Out[13]: array([[3, 4, 6, 7, 6, 9],
                [2, 3, 4, 7, 6, 7],
                [1, 2, 4, 5, 7, 7],
                [0, 1, 4, 5, 9, 5]])
```

The result is an array where the first two slots in each row contain the smallest valuesfrom that row, with the remaining values filling the remaining slots.

Finally, just as there is a np.argsort that computes indices of the sort, there is a np.argpartitionthat computes indices of the partition. We'll see this in action in the following section.

Example: k-Nearest Neighbors

Let's quickly see how we might use this argsort function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of 10 points on a two-dimensional plane. Using the standard convention, we'll arrangethese in a 10×2 array:

```
In[14]: X = rand.rand(10, 2)
```

To get an idea of how these points look, let's quickly scatter plot them:

```
In[15]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set() # Plot styling
```

```
plt.scatter(X[:, 0], X[:, 1], s=100);
```

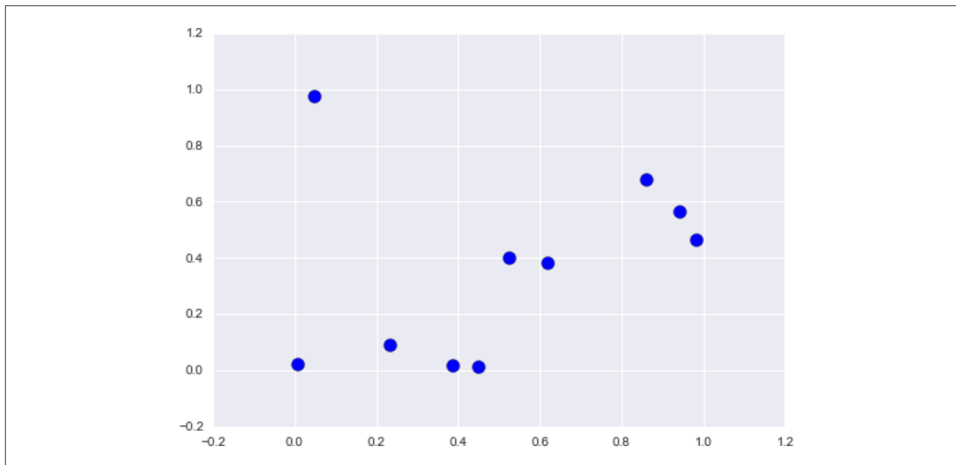


Figure . Visualization of points in the k-neighbors example

Now we'll compute the distance between each pair of points. Recall that the squared-distance between two points is the sum of the squared differences in each dimension; using the efficient broadcasting routines provided by NumPy, we can compute the matrix of square distances in a sin-gle line of code:

```
In[16]: dist_sq = np.sum((X[:,np.newaxis,:] - X[np.newaxis,:,:]) ** 2, axis=-1)
```

This operation has a lot packed into it, and it might be a bit confusing if you're unfa- miliar with NumPy's broadcasting rules. When you come across code like this, it canbe useful to break it down into its component steps:

```
In[17]: # for each pair of points, compute differences in their
        coordinates differences = X[:, np.newaxis, :] -
        X[np.newaxis, :, :] differences.shape

Out[17]: (10, 10, 2)

In[18]: # square the coordinate differences
        sq_differences = differences ** 2
        sq_differences.shape

Out[18]: (10, 10, 2)

In[19]: # sum the coordinate differences to get the squared distance

        dist_sq = sq_differences.sum(-1)
        dist_sq.shape

Out[19]: (10, 10)
```

Just to double-check what we are doing, we should see that the diagonal of this matrix(i.e., the set of distances between each point and itself) is all zero:

```
In[20]: dist_sq.diagonal()

Out[20]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

It checks out! With the pairwise square-distances converted, we can now use np.arg sort to sort along each row. The leftmost columns will then give the indices of the nearest neighbors:

```
In[21]: nearest = np.argsort(dist_sq, axis=1)
```

```
print(nearest)
```

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

Notice that the first column gives the numbers 0 through 9 in order: this is due to the fact that each point's closest neighbor is itself, as we would expect.

By using a full sort here, we've actually done more work than we need to in this case.If we're simply interested in the nearest *k* neighbors, all we need is to partition each row so that the smallest *k* + 1 squared distances come first, with larger distances fill- ing the remaining positions of the array. We can do this with the np.argpartition function:

```
In[22]: K = 2
        nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In order to visualize this network of neighbors, let's quickly plot the points along withlines representing the connections from each point to its two nearest neighbors:

```
In[23]: plt.scatter(X[:, 0], X[:, 1], s=100)

        # draw lines from each point to its two nearest neighbors
        K = 2

for i in range(X.shape[0]):
        for j in nearest_partition[i, :K+1]:
            # plot a line from X[i] to X[j]
            # use some zip magic to make it happen:
            plt.plot(*zip(X[j], X[i]), color='black')
```

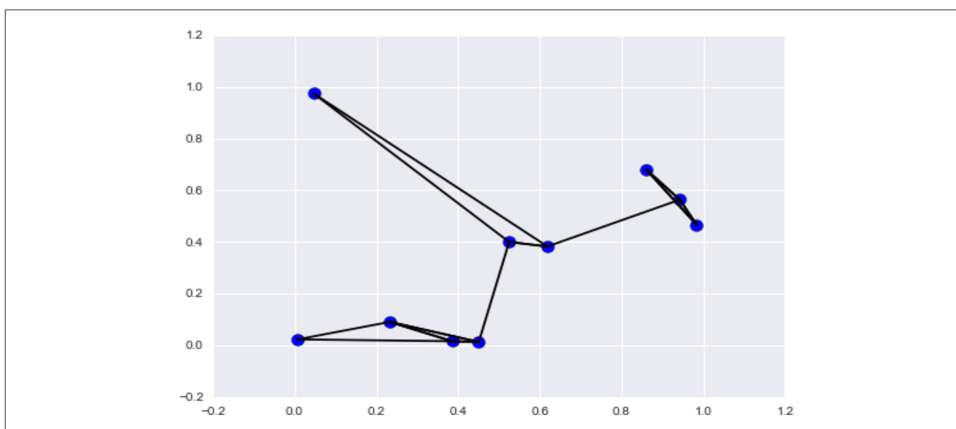


Figure : Visualization of the neighbors of each point

Lab activity : Reading and Writing CSV files

This lab activity needs to be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with # sign are comments in Python and are used to elaborate the code.

```
# # Reading and Writing CSV files
# Let's import our datafile mpg.csv, which contains fuel economy data for 234 cars.
#
# * mpg : miles per gallon
# * class : car classification
# * cty : city mpg
# * cyl : # of cylinders
# * displ : engine displacement in liters
# * drv : f = front-wheel drive, r = rear wheel drive, 4 = 4wd
# * fl : fuel (e = ethanol E85, d = diesel, r = regular, p = premium, c = CNG)
# * hwy : highway mpg
# * manufacturer : automobile manufacturer
# * model : model of car
# * trans : type of transmission
# * year : model year
import csv
get_ipython().run_line_magic('precision', '2')

with open('datasets/mpg.csv') as csvfile:
   mpg = list(csv.DictReader(csvfile))

mpg[:3]  # The first three dictionaries in our list.


# `csv.Dictreader` has read in each row of our csv file as a dictionary. `len` shows that our list is comprised of 234 dictionaries.
len(mpg)

# `keys` gives us the column names of our csv.

mpg[0].keys()

# This is how to find the average cty fuel economy across all cars. All values in the dictionaries are strings, so we need to convert to float.
# In[38]:
sum(float(d['cty']) for d in mpg) / len(mpg)
# Similarly this is how to find the average hwy fuel economy across all cars.
# In[39]:
sum(float(d['hwy']) for d in mpg) / len(mpg)
# Use `set` to return the unique values for the number of cylinders the cars in our dataset have.
cylinders = set(d['cyl'] for d in mpg)
cylinders
# Here's a more complex example where we are grouping the cars by number of cylinder, and finding the average cty mpg for each group.
CtyMpgByCyl = []

for c in cylinders:  # iterate over all the cylinder levels
```

```
    summpg = 0
    cyltypecount = 0
    for d in mpg:  # iterate over all dictionaries
        if d['cyl'] == c:  # if the cylinder level type matches,
            summpg += float(d['cty'])  # add the cty mpg
            cyltypecount += 1  # increment the count
    CtyMpgByCyl.append((c, summpg / cyltypecount))  # append the tuple ('cylinder', 'avg mpg')

CtyMpgByCyl.sort(key=lambda x: x[0])
CtyMpgByCyl
# Use `set` to return the unique values for the class types in our dataset.
vehicleclass = set(d['class'] for d in mpg)  # what are the class types
vehicleclass
# And here's an example of how to find the average hwy mpg for each class of vehicle in our dataset.
HwyMpgByClass = []

for t in vehicleclass:  # iterate over all the vehicle classes
    summpg = 0
    vclasscount = 0
    for d in mpg:  # iterate over all dictionaries
        if d['class'] == t:  # if the cylinder amount type matches,
            summpg += float(d['hwy'])  # add the hwy mpg
            vclasscount += 1  # increment the count
    HwyMpgByClass.append((t, summpg / vclasscount))  # append the tuple ('class', 'avg mpg')

HwyMpgByClass.sort(key=lambda x: x[1])
HwyMpgByClass
# # The Python Programming Language: Dates and Times
```

Day 04- Data Manipulation with Pandas

Pandas Introduction

This week we're going to deepen our investigation to how Python can be used to manipulate, clean, and query data by looking at the Pandas data tool kit. Pandas was created by Wes McKinney in 2008, and is an open source project under a very permissive license. As an open source project it's got a strong community, with over one hundred software developers all committing code to help make it better. Before pandas existed we had only a hodge podge of tools to use, such as numpy, the python core libraries, and some python statistical tools. But pandas has quickly become the defacto library for representing relational data for data scientists.

I want to take a moment here to introduce the question answering site Stack Overflow. Stack Overflow is used broadly within the software development community to post questions about programming, programming languages, and programming toolkits. What's special about Stack Overflow is that it's heavily curated by the community. And the Pandas community, in particular, uses it as their number one resource for helping new members. It's quite possible if you post a question to Stack Overflow, and tag it as being Pandas and Python related, that a core Pandas developer will actually respond to your question. In addition to posting questions, Stack Overflow is a great place to go to see what issues people are having and how they can be solved. You can learn a lot from browsing Stacks at Stack Overflow and with pandas, this is where the developer community is.

A second resource you might want to consider are books. In 2012 Wes McKinney wrote the definitive Pandas reference book called Python for Data Analysis and published by O'Reilly, and it's recently been update to a second edition. I consider this the go to book for understanding how Pandas works. I also appreciate the more brief book "Learning the Pandas Library" by Matt Harrison. It's not a comprehensive book on data analysis and statistics. But if you just want to learn the basics of Pandas and want to do so quickly, I think it's a well laid out volume and it can be had for a good price.

The field of data science is rapidly changing. There's new toolkits and method being created everyday. It can be tough to stay on top of it all. Marco Rodriguez and Tim Golden maintain a wonderful blog aggregator site called Planet Python. You can visit the webpage at planetpython.org, subscribe with an RSS reader, or get the latest articles from the @PlanetPython Twitter feed. There's lots of regular Python data science contributors, and I highly recommend it if you follow RSS feeds.

Here's my last plug on how to deepen your learning. Kyle Polich runs an excellent podcast called Data Skeptic. It isn't Python based per se, but it's well produced and it has a wonderful mixture of interviews with experts in the field as well as short educational lessons. Much of the word he describes is specific to machine learning methods. But if that's something you are planning to explore through this specialization this course is in, I would really encourage you to subscribe to his podcast.

That's it for a little bit of an introduction to this week of the course. Next we're going to dive right into Pandas library and talk about the series data structure.

Pandas is a newer package built on top of NumPy, and provides anefficient implementation of a DataFrame. DataFrames are essentially multidimen- sional arrays with attached row and column labels, and often with heterogeneous types and/or missing data. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

As we saw, NumPy's ndarray data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexi- bility (attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data avail- able in many forms in the world around us. Pandas, and in particular its Series and DataFrame objects, builds on the NumPy array structure and provides efficient accessto these sorts of "data munging" tasks that occupy much of a data scientist's time.

In this chapter, we will focus on the mechanics of using Series, DataFrame, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus.

Installing and Using Pandas

Installing Pandas on your system requires NumPy to be installed, and if you're build-ing the library from source, requires the appropriate tools to compile the C and

Cython sources on which Pandas is built. Details on this installation can be found in the Pandas documentation. If you followed the advice outlined in the preface andused the Anaconda stack, you already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

```
In[1]: import pandas
       pandas.______version
Out[1]: '0.18.1'
```

Just as we generally import NumPy under the alias np, we will import Pandas under the alias pd:

```
In[2]: import pandas as pd
```

Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. As we will see during the course of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further, let's introduce these three fundamental Pandas data structures: the Series, DataFrame, and Index.

We will start our code sessions with the standard NumPy and Pandas imports:

```
In[1]: import numpy as np
       import pandas as pd
```

The Pandas Series Object

A Pandas Seriesis a one-dimensional array of indexed data. It can be created from alist or array as follows:

```
In[2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
       data

Out[2]: 0   0.25
1           0.50
2           0.75
3           1.00
        dtype: float64
```

As we see in the preceding output, the Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes. The values are simply a familiar NumPy array:

```
In[3]: data.values

Out[3]: array([ 0.25,  0.5 ,  0.75,  1.  ])
```

The index is an array-like object of type pd.Index, which we'll discuss in more detail momentarily:

```
In[4]: data.index

Out[4]: RangeIndex(start=0, stop=4, step=1)
```

Like with a NumPy array, data can be accessed by the associated index via the familiarPython square-bracket notation:

```
In[5]: data[1]
```

```
Out[5]: 0.5

In[6]:    data[1:3]
Out[6]: 1    0.50
        2    0.75
        dtype: float64
```

As we will see, though, the Pandas Seriesis much more general and flexible than the one-dimensional NumPy array that it emulates.

Series as generalized NumPy array

From what we've seen so far, it may look like the Series object is basically inter- changeable with a one-dimensional NumPy array. The essential difference is the pres-ence of the index: while the NumPy array has an *implicitly defined* integer index usedto access the values, the Pandas Series has an *explicitly defined* index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

```
In[7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=['a', 'b', 'c', 'd'])
       data

Out[7]: a   0.25
b           0.50
c           0.75
d           1.00
        dtype: float64
```

And the item access works as expected:

```
In[8]:    data['b']

Out[8]: 0.5
```

We can even use noncontiguous or nonsequential indices:

```
In[9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=[2, 5, 3, 7])
       data

Out[9]: 2   0.25
5           0.50
3           0.75
7           1.00
        dtype:  float64

In[10]: data[5]

Out[10]: 0.5
```

Series as specialized dictionary

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure that maps typed keys to a set of typed values. Thistyping is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certainoperations.

We can make the Series-as-dictionary analogy even more clear by constructing a Seriesobject directly from a Python dictionary:

```
In[11]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois':    12882135}
        population = pd.Series(population_dict)
        population
```

```
Out[11]: California    383325
                       21
         Florida       195528
                       60
         Illinois      128821
                       35
         New York      196511
                       27
         Texa          264481
         s             93
         dtype: int64
```

By default, a Series will be created where the index is drawn from the sorted keys.From here, typical dictionary-style item access can be performed:

```
In[12]:    population['California']
```

```
Out[12]: 38332521
```

Unlike a dictionary, though, the Seriesalso supports array-style operations such asslicing:

```
In[13]: population['California':'Illinois']
```

```
Out[13]: California    383325
                       21
         Florida       195528
                       60
         Illinois      128821
                       35
         dtype: int64
```

Constructing Series objects

We’ve already seen a few ways of constructing a Pandas Series from scratch; all ofthem are

some version of the following:

```
>>> pd.Series(data, index=index)
```

where indexis an optional argument, and datacan be one of many entities.

For example, datacan be a list or NumPy array, in which case indexdefaults to aninteger sequence:

```
In[14]: pd.Series([2, 4, 6])
Out[14]: 0    2
         1    4
         2    6
         dtype: int64
```

datacan be a scalar, which is repeated to fill the specified index:

```
In[15]: pd.Series(5, index=[100, 200, 300])
Out[15]: 100   5
         200   5
         300   5
         dtype: int64
```

datacan be a dictionary, in which indexdefaults to the sorted dictionary keys:

```
In[16]:  pd.Series({2:'a',  1:'b',  3:'c'})
Out[16]: 1    b
         2    a
         3    c
         dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
In[17]:  pd.Series({2:'a',  1:'b',  3:'c'},  index=[3,  2])
Out[17]: 3    c
         2    a
         dtype: object
```

Notice that in this case, the Seriesis populated only with the explicitly identifiedkeys.

The Pandas DataFrame Object

The next fundamental structure in Pandas is the DataFrame. Like the Series object discussed in the previous section, the DataFrame can be thought of either as a gener- alization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

DataFrame as a generalized NumPy array

If a Series is an analog of a one-dimensional array with flexible indices, a DataFrameis an analog of a two-dimensional array with both flexible row indices and flexible column names. Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a DataFrame as a sequence of aligned Series objects. Here, by "aligned" we mean that they share thesame index.

To demonstrate this, let's first construct a new Serieslisting the area of each of

thefive states discussed in the previous section:

```
In[18]:
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
area
```

```
Out[18]: California    4239
                       67
         Florida       1703
                       12
         Illinois      1499
                       95
         New York      1412
                       97
         Texa          6956
         s             62
         dtype: int64
```

Now that we have this along with the populationSeriesfrom before, we can use adictionary to construct a single two-dimensional object containing this information:

```
In[19]: states = pd.DataFrame({'population': population,
'area': area})
        states
```

```
Out[19]              area      populatio
:                              n
         California  423967    38332521
         Florida     170312    19552860
         Illinois    149995    12882135
         New York    141297    19651127
         Texas       695662    26448193
```

Like the Seriesobject, the DataFramehas an indexattribute that gives access to the index labels:

```
In[20]:   states.index
Out[20]:
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

Additionally, the DataFramehas a columnsattribute, which is an Indexobject holding the column labels:

```
In[21]: states.columns

Out[21]: Index(['area', 'population'], dtype='object')
```

Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for access- ing the data.

DataFrame as specialized dictionary

Similarly, we can also think of a DataFrame as a specialization of a dictionary. Wherea dictionary maps a key to a value, a DataFrame maps a column name to a Series ofcolumn data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw earlier:

```
In[22]: states['area'] Out[22]:
California          423967
Florida       170312
Illinois      149995
New York      141297
Texas         695662
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimensional NumPy array, data[0]will return the first *row*. For a DataFrame, data['col0']will return the first*column*. Because of this, it is probably better to think about DataFrames as generalizeddictionaries rather than generalized arrays, though both ways of looking at the situa- tion can be useful.

Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples.

From a single Series object. A DataFrame is a collection of Series objects, and a single- column DataFramecan be constructed from a single Series:

```
In[23]: pd.DataFrame(population, columns=['population'])
```

```
Out[23]                 populatio
:                       n
          California    38332521
          Florida       19552860
          Illinois      12882135
          New York      19651127
          Texas         26448193
```

From a list of dicts. Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data:

```
In[24]: data = [{'a': i, 'b': 2 * i}
                for  i  in    range(3)]
        pd.DataFrame(data)
```

```
Out[24]   a   b
:
        0 0   0
        1 1   2
        2 2   4
```

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e.,"not a number") values:

```
In[25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

```
Out[25]:    a    b  c
         0  1.0  2  NaN
         1  NaN  3 4.0
```

From a dictionary of Series objects. As we saw before, a DataFrame can be constructed from a dictionary of Seriesobjects as well:

```
In[26]: pd.DataFrame({'population': population,
                      'area': area})
```

Out[26]:	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

From a two-dimensional NumPy array. Given a two-dimensional array of data, we can create a DataFramewith any specified column and index names. If omitted, an integerindex will be used for each:

```
In[27]: pd.DataFrame(np.random.rand(3, 2),
                     columns=['foo', 'bar'],
                     index=['a', 'b', 'c'])
```

Out[27]:	foo	bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

From a NumPy structured array.

A Pandas DataFrame operates much like a structured array, and can be created directly from one:

```
In[28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])A
Out[28]: array([(0, 0.0), (0, 0.0), (0, 0.0)],
               dtype=[('A', '<i8'), ('B', '<f8')])
In[29]: pd.DataFrame(A)
```

Out[29]:	A	B
0	0	0.0
1	0	0.0
2	0	0.0

The Pandas Index Object

We have seen here that both the Series and DataFrame objects contain an explicit *index* that lets you reference and modify data. This Index object is an interesting structure in itself, and it can be thought of either as an *immutable array* or as an *ordered set* (technically a multiset, as Index objects may contain repeated values). Those views have some interesting consequences in the operations available on Indexobjects. As a simple example, let's construct an Indexfrom a list of integers:

```
In[30]: ind = pd.Index([2, 3, 5, 7, 11])ind

Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

The Indexobject in many ways operates like an array. For example, we can use stan-dard Python indexing notation to retrieve values or slices:

```
In[31]:     ind[1]

Out[31]:        3

In[32]: ind[::2]

Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

Indexobjects also have many of the attributes familiar from NumPy arrays:

```
In[33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)5

(5,) 1 int64
```

One difference between Index objects and NumPy arrays is that indices are immuta-ble—that is, they cannot be modified via the normal means:

```
In[34]: ind[1] = 0

---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0

/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py ...
   1243
   1244     def _setitem_(self, key, value):
-> 1245         raise TypeError("Index does not support mutable
   operations")1246
   1247     def _getitem_(self, key):
```

```
TypeError: Index does not support mutable operations
```

This immutability makes it safer to share indices between multiple DataFrames and arrays, without the potential for side effects from inadvertent index modification.

Index as ordered set

Pandas objects are designed to facilitate operations such as joins across datasets,which depend on many aspects of set arithmetic. The Indexobject follows many of

the conventions used by Python's built-in set data structure, so that unions, intersec-tions, differences, and other combinations can be computed in a familiar way:

```
In[35]: indA = pd.Index([1, 3, 5, 7, 9])
        indB = pd.Index([2, 3, 5, 7, 11])In[36]:
indA & indB # intersection

Out[36]: Int64Index([3, 5, 7], dtype='int64')
In[37]: indA | indB  # union

Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
In[38]: indA ^ indB # symmetric difference

Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods—for example, indA.intersection(indB).

Data Indexing and Selection

We looked in detail at methods and tools to access, set, and modify val- ues in NumPy arrays. These included indexing (e.g., arr[2, 1]), slicing (e.g., arr[:,1:5]), masking (e.g., arr[arr > 0]), fancy indexing (e.g., arr[0, [1, 5]]), and combinations thereof (e.g., arr[:, [1, 5]]). Here we'll look at similar means of accessing and modifying values in Pandas Series and DataFrame objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very famil- iar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional Seriesobject, and then moveon to the more complicated two-dimensional DataFrameobject.

Data Selection in Series

As we saw in the previous section, a Series object acts in many ways like a one- dimensional NumPy array, and in many ways like a standard Python dictionary. If wekeep these two overlapping analogies in mind, it will help us to understand the pat- terns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the Seriesobject provides a mapping from a collection of keys to acollection of values:

```
In[1]: import pandas as pd
     data = pd.Series([0.25, 0.5, 0.75, 1.0],
 index=['a', 'b', 'c', 'd'])
```

```
        data
Out[1]: a   0.25
b           0.50
c           0.75
d           1.00
        dtype:  float64

In[2]: data['b'] Out[2]:

0.5
```

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
In[3]:  'a'  in  data

Out[3]: True In[4]:

data.keys()

Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')In[5]:

list(data.items())

Out[5]: [('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Series objects can even be modified with a dictionary-like syntax. Just as you canextend a dictionary by assigning to a new key, you can extend a Series by assigningto a new index value:

```
In[6]: data['e'] = 1.25
        data

Out[6]: a   0.25
b           0.50
c           0.75
d           1.00
e           1.25
        dtype: float64
```

This easy mutability of the objects is a convenient feature: under the hood, Pandas is making decisions about memory layout and data copying that might need to take place; the user generally does not need to worry about these issues.

Series as one-dimensional array

A Series builds on this dictionary-like interface and provides array-style item selec-tion via the same basic mechanisms as NumPy arrays—that is, *slices*, *masking*, and *fancy indexing*. Examples of these are as follows:

```
In[7]: # slicing by explicit index
        data['a':'c']

Out[7]: a   0.25
        b   0.50
        c   0.75
        dtype: float64
```

```
In[8]: # slicing by implicit integer index
       data[0:2]
Out[8]: a    0.25
        b    0.50
        dtype: float64
In[9]: # masking
       data[(data > 0.3) & (data < 0.8)]
Out[9]: b    0.50
        c    0.75
        dtype: float64
In[10]: # fancy indexing
        data[['a', 'e']]
Out[10]: a    0.25
         e    1.25
         dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when you are slicing with an explicit index (i.e., data['a':'c']), the final index is *included* in the slice, while when you're slicing with an implicit index (i.e., data[0:2]), the final index is *excluded* from the slice.

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, ifyour Serieshas an explicit integer index, an indexing operation such as data[1]willuse the explicit indices, while a slicing operation like data[1:3] will use the implicit Python-style index.

```
In[11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])data
Out[11]: 1    a
3    b
5    c
         dtype: object
In[12]: # explicit index when indexing
        data[1]
Out[12]: 'a'
In[13]: # implicit index when slicing
        data[1:3]
Out[13]: 3    b
5    c
         dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in

the Series.

First, the locattribute allows indexing and slicing that always references the explicit index:

```
In[14]: data.loc[1]

Out[14]: 'a'

In[15]:   data.loc[1:3]
Out[15]: 1    a
         3    b
         dtype: object
```

The iloc attribute allows indexing and slicing that always references the implicitPython-style index:

```
In[16]: data.iloc[1]

Out[16]: 'b'

In[17]:   data.iloc[1:3]
Out[17]: 3    b
         5    c
         dtype: object
```

A third indexing attribute, ix, is a hybrid of the two, and for Seriesobjects is equiva-lent to standard []-based indexing. The purpose of the ix indexer will become more apparent in the context of DataFrameobjects, which we will discuss in a moment.

One guiding principle of Python code is that "explicit is better than implicit." The explicit nature of locand ilocmake them very useful in maintaining clean and read-able code; especially in the case of integer indexes, I recommend using these both to make code easier to read and understand, and to prevent subtle bugs due to themixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a DataFrameacts in many ways like a two-dimensional or structured array,and in other ways like a dictionary of Series structures sharing the same index. These analogies can be helpful to keep in mind as we explore data selection within this structure.

DataFrame as a dictionary

The first analogy we will consider is the DataFrame as a dictionary of related Series objects. Let's return to our example of areas and populations of states:

```
In[18]: area = pd.Series({'California': 423967, 'Texas': 695662,
'New York': 141297, 'Florida': 170312,
                                    'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
'New York': 19651127, 'Florida': 19552860,
                                    'Illinois': 12882135})
```

```
data = pd.DataFrame({'area':area,
'pop':pop})data
```

```
Out[18]:             area    pop
         California  42396   383325
                     7       21
         Florida     17031   195528
                     2       60
         Illinois    14999   128821
                     5       35
         New York    14129   196511
                     7       27
         Texas       69566   264481
                     2       93
```

The individual Series that make up the columns of the DataFrame can be accessed via dictionary-style indexing of the column name:

```
In[19]: data['area']
```

```
Out[19]: California  423967
         Florida     170312
         Illinois    149995
         New York    141297
         Texas       695662
         Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

```
In[20]: data.area
```

```
Out[20]: California  423967
         Florida     170312
         Illinois    149995
         New York    141297
         Texas       695662
         Name: area, dtype: int64
```

This attribute-style column access actually accesses the exact same object as the dictionary-style access:

```
In[21]: data.area is data['area']Out[21]:

True
```

Though this is a useful shorthand, keep in mind that it does not work for all cases!For example, if the column names are not strings, or if the column names conflict with methods of the DataFrame, this attribute-style access is not possible. For exam- ple, the DataFrame has a pop()method, so data.popwill point to this rather than the"pop"column:

```
In[22]:  data.pop  is  data['pop']

Out[22]: False
```

In particular, you should avoid the temptation to try column assignment via attribute (i.e., use data['pop'] = zrather than data.pop = z).

Like with the Series objects discussed earlier, this dictionary-style syntax can also be used to modify the object, in this case to add a new column:

```
In[23]: data['density'] = data['pop'] / data['area']
        data
```

Out[23]:

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

This shows a preview of the straightforward syntax of element-by-element arithmetic between Series objects; we'll dig into this further in "Operating on Data in Pandas" on page 115.

DataFrame as two-dimensional array

As mentioned previously, we can also view the DataFrame as an enhanced two- dimensional array. We can examine the raw underlying data array using the values attribute:

```
In[24]: data.values
```

```
Out[24]: array([[ 4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
[ 1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
[ 1.49995000e+05, 1.28821350e+07, 8.58837628e+01],
[ 1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
[ 6.95662000e+05, 2.64481930e+07, 3.80187404e+01]])
```

With this picture in mind, we can do many familiar array-like observations on the DataFrame itself. For example, we can transpose the full DataFrame to swap rows and columns:

```
In[25]: data.T
```

Out[25]:

	California	Florida	Illinois	New York	Texas
area	4.239670e+05	1.703120e+05	1.499950e+05	1.412970e+05	6.956620e+05
pop	3.833252e+07	1.955286e+07	1.288214e+07	1.965113e+07	2.644819e+07
density	9.041393e+01	1.148061e+02	8.588376e+01	1.390767e+02	3.801874e+01

When it comes to indexing of DataFrame objects, however, it is clear that the dictionary-

style indexing of columns precludes our ability to simply treat it as a NumPy array. In particular, passing a single index to an array accesses a row:

```
In[26]: data.values[0]

Out[26]: array([  4.23967000e+05,            3.83325210e+07,
                  9.04139261e+01])
```

and passing a single "index" to a DataFrame accesses a column:

```
In[27]: data['area']

Out[27]: California  423967
         Florida     170312
         Illinois    149995
         New York    141297
         Texas       695662
         Name: area, dtype: int64
```

Thus for array-style indexing, we need another convention. Here Pandas again uses the loc, iloc, and ix indexers mentioned earlier. Using the iloc indexer, we can index the underlying array as if it is a simple NumPy array (using the implicit Python-style index), but the DataFrame index and column labels are maintained inthe result:

```
In[28]: data.iloc[:3, :2]
```

Out[28]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

```
In[29]: data.loc[:'Illinois', :'pop']
```

Out[29]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

The ixindexer allows a hybrid of these two approaches:

```
In[30]: data.ix[:3, :'pop']
```

Out[30]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Keep in mind that for integer indices, the ix indexer is subject to the same potential sources of confusion as discussed for integer-indexed Seriesobjects.

Any of the familiar NumPy-style data access patterns can be used within these index-ers. For example, in the loc indexer we can combine masking and fancy indexing as in the following:

```
In[31]: data.loc[data.density > 100, ['pop', 'density']]
Out[31]:            pop        density
          Florida   19552860   114.806121
          New York  19651127   139.076746
```

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working withNumPy:

```
In[32]: data.iloc[0, 2] = 90
        data

Out[32]              area    pop      density
:
          California 42396   383325   90.00000
                     7       21       0
          Florida    17031   195528   114.8061
                     2       60       21
          Illinois   14999   128821   85.88376
                     5       35       3
          New York   14129   196511   139.0767
                     7       27       46
          Texas      69566   264481   38.01874
                     2       93       0
```

To build up your fluency in Pandas data manipulation, I suggest spending some time with a simple DataFrame and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the pre- ceding discussion, but nevertheless can be very useful in practice. First, while *index- ing* refers to columns, *slicing* refers to rows:

```
In[33]: data['Florida':'Illinois']

Out[33]:            area     pop        density
          Florida   170312              19552860
          114.806121

          Illinois  149995   12882135 85.883763
```

Such slices can also refer to rows by number rather than by index:

```
In[34]: data[1:3]

Out[34]:            area     pop        density
          Florida   170312              19552860
          114.806121

          Illinois  149995   12882135 85.883763
```

Similarly, direct masking operations are also interpreted row-wise rather than

column-wise:

```
In[35]: data[data.density > 100]

Out[35]:           area     pop        density
         Florida   170312              19552860
         114.806121

         New York  141297  19651127  139.076746
```

These two conventions are syntactically similar to those on a NumPy array, and whilethese may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

Operating on Data in Pandas

One of the essential pieces of NumPy is the ability to perform quick element-wise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and loga- rithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the ufuncs that we introduced in "Computation on NumPy Arrays: Universal Func- tions" on page 50 are key to this.

Pandas includes a couple useful twists, however: for unary operations like negation and trigonometric functions, these ufuncs will *preserve index and column labels* in theoutput, and for binary operations such as addition and multiplication, Pandas will automatically *align indices* when passing the objects to the ufunc. This means that keeping the context of data and combining data from different sources—both poten- tially error-prone tasks with raw NumPy arrays—become essentially foolproof ones with Pandas. We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFramestructures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on Pandas Series and DataFrame objects. Let's start by defining a simple Series and DataFrameon which to demonstrate this:

```
In[1]: import pandas as pd
       import numpy as np

In[2]: rng = np.random.RandomState(42)

       ser = pd.Series(rng.randint(0, 10, 4))
       ser


Out[2]: 0   6
1           3
2           7
3           4
         dtype: int64

In[3]: df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                         columns=['A', 'B', 'C', 'D'])

       df


Out[3] A     B  C  D
:
```

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

If we apply a NumPy ufunc on either of these objects, the result will be another Pan-das object *with the indices preserved:*

```
In[4]: np.exp(ser)
```

```
Out[4]: 0    403.428793
1              20.085537
2              1096.633158
3              54.598150
dtype: float64
```

Or, for a slightly more complex calculation:

```
In[5]: np.sin(df * np.pi / 4)
```

Out[5]:	A	B	C	D
0	-1.000000	7.071068e-01	1.000000	-1.000000e+00
1	-0.707107	1.224647e-16	0.707107	-7.071068e-01
2	-0.707107	1.000000e+00	-0.707107	1.224647e-16

Any of the ufuncs discussed in "Computation on NumPy Arrays: Universal Func-tions" on page 50 can be used in a similar manner.

UFuncs: Index Alignment

For binary operations on two Seriesor DataFrameobjects, Pandas will align indicesin the process of performing the operation. This is very convenient when you are working with incomplete data, as we'll see in some of the examples that follow.

Index alignment in Series

As an example, suppose we are combining two different data sources, and find only the top three US states by *area* and the top three US states by *population*:

```
In[6]: area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
'New York': 19651127}, name='population')
```

Let's see what happens when we divide these to compute the population density:

```
In[7]: population / area Out[7]:
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

The resulting array contains the *union* of indices of the two input arrays, which we could determine using standard Python set arithmetic on these indices:

```
In[8]: area.index | population.index
```

```
Out[8]: Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked with NaN, or "Not a Number," which is how Pandas marks missing data. This index matching is imple mented this way for any of Python's built-in arithmetic expressions; any missing val-ues are filled in with NaN by default:

```
In[9]: A = pd.Series([2, 4, 6], index=[0, 1, 2])
       B = pd.Series([1, 3, 5], index=[1, 2, 3])A +
       B
```

```
Out[9]: 0    NaN
1    5.0
2    9.0
3    NaN
dtype: float64
```

If using NaN values is not the desired behavior, we can modify the fill value using appropriate object methods in place of the operators. For example, calling A.add(B)is equivalent to calling A + B, but allows optional explicit specification of the fill valuefor any elements in Aor Bthat might be missing:

```
In[10]: A.add(B, fill_value=0)
```

```
Out[10]: 0   2.0
1            5.0
2            9.0
3            5.0
dtype: float64
```

Index alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when you are performing operations on DataFrames:

```
In[11]: A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
columns=list('AB'))
A
```

```
Out[11]:    A   B
0  1  11
1  5   1
```

```
In[12]: B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
columns=list('BAC'))
B
```

```
Out[12]:    B  A  C0
         4  0  9

         1  5  8  0

         2  9  2  6
```

```
In[13]: A + B
```

```
Out[13]:      A      B   C0
            1.0  15.0 NaN

         1  13.0   6.0 NaN

         2   NaN  NaN NaN
```

Notice that indices are aligned correctly irrespective of their order in the two objects, and indices in the result are sorted. As was the case with Series, we can use the asso-ciated object's arithmetic method and pass any desired fill_valueto be used in placeof missing entries. Here we'll fill with the mean of all values in A(which we computeby first stacking the rows of A):

```
In[14]:   fill   =   A.stack().mean()
        A.add(B, fill_value=fill)
```

```
Out[14]:  A        B      C

      0    1.0  15.0  13.5

      1   13.0  6.0    4.5
      2    6.5  13.5  10.5
```

Table. Lists Python operators and their equivalent Pandas object methods.

Table . Mapping between Python operators and Pandas methods

Python operator	Pandas method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

Ufuncs: Operations Between DataFrame and Series

When you are performing operations between a DataFrame and a Series, the index and column alignment is similarly maintained. Operations between a DataFrame and a Series are similar to operations between a two-dimensional and one-dimensional NumPy array. Consider one common operation, where we find the difference of a two-dimensional array and one of its rows:

```
In[15]: A = rng.randint(10, size=(3, 4))A
```

Out[15]: array([[3, 8, 2, 4],

[2, 6, 4, 8],

[6, 1, 3, 8]])

```
In[16]: A - A[0]
```

Out[16]: array([[0, 0, 0, 0],
[-1, -2, 2, 4],
[3, -7, 1, 4]])

In Pandas, the convention similarly operates row-wise by default:

```
In[17]: df = pd.DataFrame(A,
        columns=list('QRST'))df - df.iloc[0]
```

Out[17]:

	Q	R	S	T
0	0	0	0	0
1	-1	-2	2	4
2	3	-7	1	4

If you would instead like to operate column-wise, you can use the object methods mentioned earlier, while specifying the axiskeyword:

```
In[18]: df.subtract(df['R'], axis=0)
```

Out[18]:

	Q	R	S	T
0	-5	0	-6	-4
1	-4	0	-2	2
2	5	0	2	7

Note that these DataFrame/Series operations, like the operations discussed before, will automatically align indices between the two elements:

```
In[19]: halfrow = df.iloc[0, ::2]
        halfrow
```

Out[19]: Q 3

S 2

Name: 0, dtype: int64

```
In[20]: df - halfrow
```

Out[20]:

	Q	R	S	T
0	0.0	NaN	0.0	NaN
1	-1.0	NaN	2.0	NaN
2	3.0	NaN	1.0	NaN

This preservation and alignment of indices and columns means that operations ondata in Pandas will always maintain the data context, which prevents the types of sillyerrors that might come up when you are working with heterogeneous and/or mis- aligned data in raw

NumPy arrays.

Handling Missing Data

The difference between data found in many tutorials and data in the real world is thatreal-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more compli-cated, different data sources may indicate missing data in different ways.

we will discuss some general considerations for missing data, discuss how Pandas chooses to represent it, and demonstrate some built-in Pandas tools for handling missing data in Python. Here and throughout the book, we'll refer to miss- ing data in general as *null*, *NaN*, or *NA* values.

Trade-Offs in Missing Data Conventions

A number of schemes have been developed to indicate the presence of missing data ina table or DataFrame. Generally, they revolve around one of two strategies: using a *mask* that globally indicates missing values, or choosing a *sentinel value* that indicatesa missing entry.

> In the masking approach, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the sentinel approach, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be repre- sented, and may require extra (often non-optimized) logic in CPU and GPU arith- metic. Common special values like NaN are not available for all data types.

As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the R language uses reserved bit pat-terns within each data type as sentinel values indicating missing data, while the SciDBsystem uses an extra byte attached to every cell to indicate a NA state.

Missing Data in Pandas

> The way in which Pandas handles missing values is constrained by its reliance on theNumPy package, which does not have a built-in notion of NA values for non-floating-point data types.

Pandas could have followed R's lead in specifying bit patterns for each individual data type to indicate nullness, but this approach turns out to be rather unwieldy. While R contains four basic data types, NumPy supports *far* more than this: for example,while R has a single integer type, NumPy supports *fourteen* basic integer types once you account for available precisions, signedness, and endianness of the encoding. Reserving a specific bit pattern in all available NumPy types would lead to anunwieldy amount of overhead in special-casing various operations for various types, likely even requiring a new fork of the NumPy package. Further, for the smaller data types (such as 8-bit integers), sacrificing a bit to use as a mask will significantly reduce the range of values it can represent.

NumPy does have support for masked arrays—that is, arrays that have a separate Boolean

mask array attached for marking data as “good” or “bad.” Pandas could havederived from this, but the overhead in both storage, computation, and code mainte- nance makes that an unattractive choice.

With these constraints in mind, Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating- point NaN value, and the Python None object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

None: Pythonic missing data

The first sentinel value used by Pandas is None, a Python singleton object that is oftenused for missing data in Python code. Because None is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object'(i.e., arrays of Python objects):

```
In[1]: import numpy as np
       import pandas as pd

In[2]: vals1 = np.array([1, None, 3, 4])
       vals1

Out[2]: array([1, None, 3, 4], dtype=object)
```

This dtype=object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen forarrays with native types:

```
In[3]: for dtype in ['object', 'int']:
           print("dtype =", dtype)
           %timeit np.arange(1E6, dtype=dtype).sum()
           print()
dtype = object
10 loops, best of 3: 78.2 ms per loop

dtype = int
100 loops, best of 3: 3.06 ms per loop
```

The use of Python objects in an array also means that if you perform aggregationslike sum()or min()across an array with a Nonevalue, you will generally get an error:

```
In[4]: vals1.sum()

TypeError                          Traceback (most recent call last)

<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()


/Users/jakevdp/anaconda/lib/python3.5/site-packages/numpy/core/_methods.py ...
    30
```

```
31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32      return   umr_sum(a,   axis,   dtype,   out,
keepdims)33
34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):


TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

This reflects the fact that addition between an integer and Noneis undefined.

NaN: Missing numerical data

The other missing data representation, NaN (acronym for *Not a Number*), is different;it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
       vals2.dtype

Out[5]: dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. You should be aware that NaN is a bit like a data virus—it infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN:

```
In[6]: 1 + np.nan

Out[6]: nan

In[7]:   0   *   np.nan

Out[7]: nan
```

Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
In[8]: vals2.sum(), vals2.min(), vals2.max()

Out[8]: (nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

```
In[9]:          np.nansum(vals2),          np.nanmin(vals2),

np.nanmax(vals2)Out[9]: (8.0, 1.0, 4.0)
```

Keep in mind that NaN is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

NaN and None in Pandas

NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

```
In[10]: pd.Series([1, np.nan, 2, None])
Out[10]: 0    1.0
         1    NaN
```

```
2      2.0
3      NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-castswhen NA values are present. For example, if we set a value in an integer array to np.nan, it will automatically be upcast to a floating-point type to accommodate the NA:

```
In[11]: x = pd.Series(range(2), dtype=int)x
Out[11]: 0     0
         1     1
         dtype: int64
In[12]: x[0] = Nonex
Out[12]: 0     NaN
1      1.0
         dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automati- cally converts the None to a NaN value. (Be aware that there is a proposal to add a native integer NA to Pandas in the future; as of this writing, it has not been included.)

While this type of magic may feel a bit hackish compared to the more unified approach to NA values in domain-specific languages like R, the Pandas sentinel/cast-ing approach works quite well in practice and in my experience only rarely causes issues.

Operating on Null Values

As we have seen, Pandas treats None and NaN as essentially interchangeable for indi- cating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

isnull()

Generate a Boolean mask indicating missing values

notnull()

Opposite of isnull()

dropna()

Return a filtered version of the data

fillna()

Return a copy of the data with missing values filled or imputed

We will conclude this section with a brief exploration and demonstration of these routines.

Detecting null values

Pandas data structures have two useful methods for detecting null data: isnull()and notnull(). Either one will return a Boolean mask over the data. For example:

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])

In[14]: data.isnull()

Out[14]: 0    False
         1     True
         2    False
         3     True
         dtype: bool

In[15]: data[data.notnull()]

Out[15]: 0        1
         2    hello
         dtype: object
```

The isnull() and notnull() methods produce similar Boolean results for Data Frames.

Dropping null values

In addition to the masking used before, there are the convenience methods, dropna()(which removes NA values) and fillna() (which fills in NA values). For a Series,the result is straightforward:

```
In[16]: data.dropna()
Out[16]: 0        1
         2    hello
         dtype: object
```

For a DataFrame, there are more options. Consider the following DataFrame:

```
In[17]: df = pd.DataFrame([[1,      np.nan, 2],
[2,       3,       5],
[np.nan, 4,       6]])
                     df
Out[17]:     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

We cannot drop single values from a DataFrame; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so dropna()gives a number of options for a DataFrame.

By default, dropna() will drop all rows in which *any* null value is present:

```
In[18]:      df.dropna()
Out[18]:     0    1  2
         1  2.0  3.0  5
```

Alternatively, you can drop NA values along a different axis; axis=1 drops all columns containing a null value:

```
In[19]: df.dropna(axis='columns')
```

```
Out[19]:    2
         0  2
         1  5
         2  6
```

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specifiedthrough the how or thresh parameters, which allow fine control of the number of nulls to allow through.

The default is how='any', such that any row or column (depending on the axis key-word) containing a null value will be dropped. You can also specify how='all', which will only drop rows/columns that are *all* null values:

```
In[20]: df[3] = np.nan
        df
```

```
Out[20]:    0    1    2  3
         0  1.0  NaN  2  NaN
         1  2.0  3.0  5  NaN
         2  NaN  4.0  6  NaN
```

```
In[21]: df.dropna(axis='columns', how='all')
```

```
Out[21]:    0    1    2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

For finer-grained control, the thresh parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
In[22]: df.dropna(axis='rows', thresh=3)
Out[22]:     0    1  2   3
          1  2.0  3.0  5 NaN
```

Here the first and last row have been dropped, because they contain only two non- null values.

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place usingthe isnull() method as a mask, but because it is such a common operation Pandas provides the fillna() method, which returns a copy of the array with the null valuesreplaced.

Consider the following Series:

```
In[23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
        data
```

```
Out[23]: a    1.0
         b    NaN
         c    2.0
         d    NaN
         e    3.0
         dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
In[24]: data.fillna(0)
```

```
Out[24]: a   1.0
b            0.0
c            2.0
d            0.0
e            3.0
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
In[25]: # forward-fill
        data.fillna(method='ffill')
```

```
Out[25]: a   1.0
b            1.0
c            2.0
d            2.0
e            3.0
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
In[26]: # back-fill
        data.fillna(method='bfill')
```

```
Out[26]: a   1.0
b            2.0
c            2.0
d            3.0
e            3.0
dtype: float64
```

For DataFrames, the options are similar, but we can also specify an axisalong which the fills take place:

```
In[27]: df
```

Out[27]:	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN

```
 2  NaN  4.0  6  NaN
```

```
In[28]: df.fillna(method='ffill', axis=1)
```

```
Out[28]:    0    1    2    3
         0  1.0  1.0  2.0  2.0
         1  2.0  3.0  5.0  5.0
         2  NaN  4.0  6.0  6.0
```

Notice that if a previous value is not available during a forward fill, the NA valueremains.

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two- dimensional data, stored in Pandas Seriesand DataFrameobjects, respectively. Oftenit is useful to go beyond this and store higher-dimensional data—that is, data indexedby more than one or two keys. While Pandas does provide Paneland Panel4Dobjectsthat natively handle three-dimensional and four-dimensional data a far more common pattern in practice is to make use of *hierarchical indexing* (also known as *multi-indexing*) to incorporate multiple index *levels* within a single index. In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

In this section, we'll explore the direct creation of MultiIndexobjects; considerationsaround indexing, slicing, and computing statistics across multiply indexed data; and useful routines for converting between simple and hierarchically indexed representa- tions of your data.

We begin with the standard imports:

```
In[1]: import pandas as pd
       import numpy as np
```

A Multiply Indexed Series

Let's start by considering how we might represent two-dimensional data within a one-dimensional Series. For concreteness, we will consider a series of data where each point has a character and numerical key.

The bad way

Suppose you would like to track data about states from two different years. Using thePandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
In[2]: index = [('California', 2000), ('California', 2010),
                ('New York', 2000), ('New York', 2010),
                ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
18976457, 19378102,
20851820, 25145561]
       pop = pd.Series(populations,
       index=index)pop
```

```
Out[2]: (California, 2000)   3387164
```

```
                    8
(California, 2010)  3725395
                    6
    (New York, 2000) 1897645
                    7
    (New York, 2010) 1937810
                    2
    (Texas, 2000)   2085182
                    0
    (Texas, 2010)   25145561
    dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```
In[3]: pop[('California', 2010):('Texas', 2000)]

Out[3]: (California, 2010)  3725395
                            6
        (New York, 2000)    1897645
                            7
        (New York, 2010)    1937810
                            2
        (Texas, 2000)       2085182
                            0
        dtype: int64
```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

```
In[4]: pop[[i for i in pop.index if i[1] == 2010]]

Out[4]: (California, 2010)  3725395
                            6
        (New York, 2010)    1937810
                            2
        (Texas, 2010)       2514556
                            1
        dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

The better way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas MultiIndextype gives us the type of opera-tions we wish to have. We can create a multi-index from the tuples as follows:

```
In[5]:              index                   =
    pd.MultiIndex.from_tuples(index)index

Out[5]: MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
                   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the MultiIndexcontains multiple *levels* of indexing—in this case, the statenames and the years, as well as multiple *labels* for each data point which encode theselevels.

If we reindex our series with this MultiIndex, we see the hierarchical representation of the data:

```
In[6]: pop = pop.reindex(index)
       pop
```

```
Out[6]: California  2000   3387164
                           8
2010                       3725395
                           6
New York 2000              1897645
                           7
2010                       1937810
                           2
        Texas        2000 20851820
                     2010  25145561
        dtype: int64
```

First two columns of the Series representation show the multiple index val-ues, while the third column shows the data. Notice that some entries are missing in the first column:in this multi-index representation, any blank entry indicates thesame value as the line above it.

Now to access all data for which the second index is 2010, we can simply use the Pan-das slicing notation:

```
In[7]: pop[:, 2010]
```

```
Out[7]: California    372539
                      56
        New York      193781
                      02
        Texa          251455
        s             61
        dtype: int64
```

The result is a singly indexed array with just the keys we're interested in. This syntaxis much more convenient (and the operation is much more efficient!) than the home- spun tuple-based multi-indexing solution that we started with. We'll now further dis- cuss this sort of indexing operation on hierarchically indexed data.

MultiIndex as extra dimension

You might notice something else here: we could easily have stored the same data using a simple DataFrame with index and column labels. In fact, Pandas is built with this equivalence in mind. The unstack() method will quickly convert a multiply- indexed Series into a conventionally indexed DataFrame:

```
In[8]: pop_df = pop.unstack()
       pop_df
```

```
Out[8]              2000     2010
:
        California  338716   372539
                    48       56
        New York    189764   193781
```

	57	02
Texas	208518 20	251455 61

Naturally, the stack()method provides the opposite operation:

```
In[9]: pop_df.stack()
```

```
Out[9]:  California  2000 3387164
                           8
                     2010 3725395
                           6
         New York    2000 1897645
                           7
                     2010 1937810
                           2
         Texas       2000 2085182
                           0
                     2010 2514556
                           1
         dtype: int64
```

Seeing this, you might wonder why would we would bother with hierarchical index- ing at all. The reason is simple: just as we were able to use multi-indexing to represent two-dimensional data within a one-dimensional Series, we can also use it to repre- sent data of three or more dimensions in a Series or DataFrame. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent. Concretely, we might want to add another column of demographic data for each state at each year (say, population under 18); with a MultiIndex this is as easy as adding another col- umn to the DataFrame:

```
In[10]: pop_df = pd.DataFrame({'total': pop,
'under18': [9267089, 9284094,
4687374, 4318033,
                                        5906301, 6879014]})
        pop_df
```

```
Out[10]                        total     under
:                                        18
         California  200  338716    92670
                     0    48        89
                     201  372539    92840
                     0    56        94
         New York    200  189764    46873
                     0    57        74
                     201  193781    43180
                     0    02        33
         Texas       200  208518    59063
                     0    20        01
                     201  251455    68790
                     0    61        14
```

Here we compute the fraction of people under 18 by year, given the above data:

```
In[11]: f_u18 = pop_df['under18'] / pop_df['total']
        f_u18.unstack()
```

```
Out[11]:             2000       2010
          California 0.273594   0.249211
          New York   0.247010   0.222831
          Texas      0.283251   0.273568
```

This allows us to easily and quickly manipulate and explore even high-dimensional data.

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed Seriesor DataFrame is to simply pass a list of two or more index arrays to the constructor. For example:

```
In[12]: df = pd.DataFrame(np.random.rand(4, 2),
                          index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                          columns=['data1', 'data2'])
        df
```

```
Out[12]:     data1     data2
         a 1 0.554233  0.356072
           2 0.925244  0.219474
         b 1 0.441759  0.610054
           2 0.171495  0.886688
```

The work of creating the MultiIndexis done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will auto- matically recognize this and use a MultiIndexby default:

```
In[13]: data = {('California', 2000): 33871648,
('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
        pd.Series(data)
```

```
Out[13]: California  2000 33871648
                     2010 37253956
         New York    2000 18976457
                     2010 19378102
```

```
Texas       2000 20851820
            2010 25145561
dtype: int64
```

Nevertheless, it is sometimes useful to explicitly create a MultiIndex; we'll see a cou-ple of these methods here.

Explicit MultiIndex constructors

For more flexibility in how the index is constructed, you can instead use the class method constructors available in the pd.MultiIndex. For example, as we did before, you can construct the MultiIndexfrom a simple list of arrays, giving the index valueswithin each level:

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can construct it from a list of tuples, giving the multiple index values of each point:

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can even construct it from a Cartesian product of single indices:

```
In[16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])

Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Similarly, you can construct the MultiIndex directly using its internal encoding by passing levels (a list of lists containing available index values for each level) and labels(a list of lists that reference these labels):

```
In[17]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                      labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

Out[17]: MultiIndex(levels=[['a', 'b'], [1, 2]],
labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can pass any of these objects as the indexargument when creating a Seriesor DataFrame, or to the reindexmethod of an existing Seriesor DataFrame.

MultiIndex level names

Sometimes it is convenient to name the levels of the MultiIndex. You can accomplishthis by passing the names argument to any of the above MultiIndex constructors, or by setting the namesattribute of the index after the fact:

```
In[18]: pop.index.names = ['state', 'year']
        pop

Out[18]: state       year
         California  2000 3387164
```

8
2010 3725395
6
New York 2000 1897645
7
2010 1937810
2
Texas 2000 2085182
0
2010 2514556
1
dtype: int64

With more involved datasets, this can be a useful way to keep track of the meaning ofvarious index values.

MultiIndex for columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rowscan have multiple levels of indices, the columns can have multiple levels as well. Con-sider the following, which is a mock-up of some (somewhat realistic) medical data:

```
In[19]:
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index,
columns=columns)health_data
```

```
Out[19]: subject     Bob        Guido       Sue
         type         HR  Temp    HR  Temp    HR  Temp
         year visit
         2013 1     31.0  38.7  32.0  36.7  35.0  37.2
              2     44.0  37.7  50.0  35.0  29.0  36.7
         2014 1     30.0  37.4  39.0  37.8  61.0  36.9
              2     47.0  37.8  48.0  37.3  51.0  36.5
```

Here we see where the multi-indexing for both rows and columns can come in *very* handy.

This is fundamentally four-dimensional data, where the dimensions are the subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top-level column by the person's name and get a full DataFramecontaining just that person's information:

```
In[20]: health_data['Guido']

Out[20]: type          HR   Temp
         year visit
         2013 1        32.0  36.7
              2        50.0  35.0
         2014 1        39.0  37.8
              2        48.0  37.3
```

For complicated records containing multiple labeled measurements across multiple times for many subjects (people, countries, cities, etc.), use of hierarchical rows and columns can be extremely convenient!

Indexing and Slicing a MultiIndex

Indexing and slicing on a MultiIndex is designed to be intuitive, and it helps if you think about the indices as added dimensions. We'll first look at indexing multiply indexed Series, and then multiply indexed DataFrames.

Multiply indexed Series

Consider the multiply indexed Seriesof state populations we saw earlier:

```
In[21]: pop

Out[21]: state        year
         California   2000    33871648
                      2010    37253956
         New York     2000    18976457
                      2010    19378102
         Texas        2000    20851820
                      2010    25145561
         dtype: int64
```

We can access single elements by indexing with multiple terms:

```
In[22]: pop['California', 2000]

Out[22]: 33871648
```

The MultiIndexalso supports *partial indexing*, or indexing just one of the levels inthe index. The result is another Series, with the lower-level indices maintained:

```
In[23]:    pop['California']
Out[23]: year
         2000    33871648
         2010    37253956
         dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted (see discussion in "Sorted and unsorted indices" on page 137):

```
In[24]: pop.loc['California':'New York']

Out[24]: state      year
         California 2000 3387164
                         8
                    2010 3725395
                         6
         New York   2000 1897645
                         7
                    2010 1937810
                         2
         dtype: int64
```

With sorted indices, we can perform partial indexing on lower levels by passing anempty slice in the first index:

```
In[25]: pop[:, 2000]

Out[25]: state
         California  33871648
         New York    18976457
         Texas       20851820
         dtype: int64
```

For example, selection based on Boolean masks:

```
In[26]: pop[pop > 22000000]

Out[26]: state      year
         California 2000 3387164
                         8
                    2010 3725395
                         6
         Texas      2010 2514556
                         1
         dtype:
         int64
```

Selection based on fancy indexing also works:

```
In[27]: pop[['California', 'Texas']]

Out[27]: state      year
         California 2000 3387164
                         8
                    2010 3725395
                         6
         Texas      2000 2085182
                         0
                    2010 2514556
                         1
         dtype: int64
```

Multiply indexed DataFrames

A multiply indexed DataFrame behaves in a similar manner. Consider our toy medical DataFramefrom before:

In[28]: health_data

```
Out[28]:
subject      Bob        Guid          Su
                        o             e
type          HR  Tem HR      Tem HR   Temp
                   p           p
year
visit
2013 1      31.  38. 32.0    36. 35.0 37.2
             0    7           7
```

Remember that columns are primary in a DataFrame, and the syntax used for multi- ply indexed Seriesapplies to the columns. For example, we can recover Guido's heartrate data with a simple operation:

In[29]: health_data['Guido', 'HR']

```
Out[29]:      visit
year
2013          1      32.0
              2      50.0
2014          1      39.0
              2      48.0
```

Name: (Guido, HR), dtype: float64

Also, as with the single-index case, we can use the loc, iloc, and ix indexers intro- duced in "Data Indexing and Selection" on page 107. For example:

In[30]: health_data.iloc[:2, :2]

```
Out[30]: subject      Bo
         type         b
         year visit   H   Te
                      R   mp
2013 1             31.0   38.
                          7
2                  44.0   37.
                          7
```

These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in locor iloccan be passed a tuple of multiple indices. Forexample:

In[31]: health_data.loc[:, ('Bob', 'HR')]

```
Out[31]:      visit
year
2013          1      31.0
              2      44.0
2014          1      30.0
              2      47.0
```

Name: (Bob, HR), dtype: float64

Working with slices within these index tuples is not especially convenient; trying to

create a slice within a tuple will lead to a syntax error:

```
In[32]: health_data.loc[(:, 1), (:, 'HR')]

  File "<ipython-input-32-8e3cc151e316>", line
    1health_data.loc[(:, 1), (:, 'HR')]
                      ^
SyntaxError: invalid syntax
```

You could get around this by building the desired slice explicitly using Python's built-in slice() function, but a better way in this context is to use an IndexSlice object, which Pandas provides for precisely this situation. For example:

```
In[33]: idx = pd.IndexSlice
        health_data.loc[idx[:, 1], idx[:, 'HR']]

Out[33]: subject    Bob  Guido   Sue
         type        HR    HR     HR
         year visit
         2013 1     31.0  32.0  35.0
         2014 1     30.0  39.0  61.0
```

There are so many ways to interact with data in multiply indexed Series and Data Frames, and as with many tools in this book the best way to become familiar with them is to try them out!

Rearranging Multi-Indices

One of the keys to working with multiply indexed data is knowing how to effectively transform the data. There are a number of operations that will preserve all the information in the dataset, but rearrange it for the purposes of various computations. Wesaw a brief example of this in the stack() and unstack() methods, but there are many more ways to finely control the rearrangement of data between hierarchicalindices and columns, and we'll explore them here.

Sorted and unsorted indices

Earlier, we briefly mentioned a caveat, but we should emphasize it more here. *Many ofthe* MultiIndex *slicing operations will fail if the index is not sorted.* Let's take a look at this here.

We'll start by creating some simple multiply indexed data where the indices are *not lexographically sorted*:

```
In[34]: index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
        data = pd.Series(np.random.rand(6), index=index)
        data.index.names = ['char', 'int']

        data

Out[34]: char  int
         a     1    0.003001
               2    0.164974
         c     1    0.741650
               2    0.569264
```

```
b    1    0.001693
     2    0.526226
dtype: float64
```

If we try to take a partial slice of this index, it will result in an error:

```
In[35]: try:
            data['a':'b']
        except KeyError as e:
            print(type(e))
            print(e)
```

```
<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Although it is not entirely clear from the error message, this is the result of the MultiIndex not being sorted. For various reasons, partial slices and other similar opera- tions require the levels in the MultiIndex to be in sorted (i.e., lexographical) order. Pandas provides a number of convenience routines to perform this type of sorting; examples are the sort_index() and sortlevel() methods of the DataFrame. We'll use the simplest, sort_index(), here:

```
In[36]: data = data.sort_index()
        data
```

```
Out[36]: char  int
         a     1    0.003001
               2    0.164974
         b     1    0.001693
               2    0.526226
         c     1    0.741650
               2    0.569264
         dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
In[37]: data['a':'b']
```

```
Out[37]: char  int
         a     1    0.003001
               2    0.164974
         b     1    0.001693
               2    0.526226
         dtype: float64
```

Stacking and unstacking indices

As we saw briefly before, it is possible to convert a dataset from a stacked multi-indexto a simple two-dimensional representation, optionally specifying the level to use:

```
In[38]: pop.unstack(level=0)
```

```
Out[38]:
state      California  New York  Texas
year
2000       33871648    18976457  20851820
2010       37253956    19378102  25145561
```

```
In[39]: pop.unstack(level=1)
```

```
Out[39]:
year        2000       2010
state
California  33871648   37253956
New York    18976457   19378102
Texas       20851820   25145561
```

The opposite of unstack()is stack(), which here can be used to recover the original series:

```
In[40]: pop.unstack().stack()
```

```
Out[40]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Index setting and resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the reset_index method. Calling this on the popula- tion dictionary will result in a DataFrame with a *state* and *year* column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
In[41]: pop_flat =
        pop.reset_index(name='population')
        pop_flat
```

```
Out[41]:  state       year   population
0         California  2000   33871648
1         California  2010   37253956
```

2	New York	2000	18976457
3	New York	2010	19378102
4	Texas	2000	20851820
5	Texas	2010	25145561

Often when you are working with data in the real world, the raw input data looks likethis and it's useful to build a MultiIndex from the column values. This can be done with the set_index method of the DataFrame, which returns a multiply indexed DataFrame:

```
In[42]: pop_flat.set_index(['state', 'year'])
```

Out[42]:

state	year	population
California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820
	2010	25145561

In practice, I find this type of reindexing to be one of the more useful patterns when Iencounter real-world datasets.

Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as mean(), sum(), and max(). For hierarchically indexed data, these can be passed a levelparameter that controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
In[43]: health_data
```

Out[43]:

subject		Bob		Guido		Sue	
type		HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2

Perhaps we'd like to average out the measurements in the two visits each year. We cando this by naming the index level we'd like to explore, in this case the year:

```
In[44]: data_mean =
        health_data.mean(level='year')
        data_mean
```

```
Out[44]: subject   Bob        Guido          Sue
         type      HR  Temp  HR  Temp   HR
                   Tempyear
         2013     37.5  38.2  41.0  35.85  32.0  36.95
         2014     38.5  37.6  43.5  37.55  56.0  36.70
```

By further making use of the axis keyword, we can take the mean among levels on the columns as well:

```
In[45]: data_mean.mean(axis=1, level='type')
```

```
Out[45]:      HR          Temp
type
year
2013          36.83333    37.00000
              3           0
2014          46.00000    37.28333
              0           3
```

Combining Datasets: Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the pd.merge func- tion, and we'll see a few examples of how this can work in practice.

Categories of Joins

The pd.merge() function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the pd.merge() interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

One-to-one joins

Perhaps the simplest type of merge expression is the one-to-one join, which is in many ways very similar to the column-wise concatenation. As a concrete example, consider the followingtwo DataFrames, which contain information on several employees in a company:

```
In[2]:
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
print(df1); print(df2)
```

```
df1                                  df2
        employee   group                   employee   hire_date
0       Bob        Accountin    0          Lisa       2004
                   g
1       Jake       Engineerin   1          Bob        2008
```

2	Lisa	Engineering	2	Jake	2012
3	Sue	HR	3	Sue	2014

To combine this information into a single DataFrame, we can use the pd.merge() function:

```
In[3]: df3 = pd.merge(df1, df2)
       df3
```

Out[3]:	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

The pd.merge()function recognizes that each DataFramehas an "employee" column,and automatically joins using this column as a key. The result of the merge is a new DataFrame that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the "employee" column differs between df1 and df2, and the pd.merge() function cor- rectly accounts for this.

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting DataFrame will preserve those dupli- cate entries as appropriate. Consider the following example of a many-to-one join:

```
In[4]: df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                           'supervisor': ['Carly', 'Guido', 'Steve']})
       print(df3); print(df4); print(pd.merge(df3, df4))
```

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3		HR	2014	Steve

e Su

The resulting DataFramehas an additional column with the “supervisor” information,where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless welldefined. If the key column in both the left and right array contains duplicates, thenthe result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group.

By performing a many-to-many join, we can recover the skills associated with any individual person:

```
In[5]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                     'Engineering', 'Engineering', 'HR', 'HR'],
                           'skills': ['math', 'spreadsheets', 'coding', 'linux',
                                      'spreadsheets', 'organization']})
print(df1); print(df5); print(pd.merge(df1, df5))
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df5

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

pd.merge(df1, df5)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6		HR	spreadsheet

7 Sue	HR	organization

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section, we'll consider some of the options pro- vided by pd.merge() that enable you to tune how the join operations work.

Specification of the Merge Key

We've already seen the default behavior of pd.merge(): it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and pd.merge() provides a variety of options for handling this.

The on keyword

Most simply, you can explicitly specify the name of the key column using the on key- word, which takes a column name or a list of column names:

```
In[6]: print(df1); print(df2); print(pd.merge(df1, df2, on='employee'))
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

pd.merge(df1, df2, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

This option works only if both the left and right DataFrames have the specified col- umn name.

The left_on and right_on keywords

At times you may wish to merge two datasets with different column names; for exam-ple, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the left_on and right_on keywords to specify the two column names:

```
In[7]:
```

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
print(df1); print(df3);
print(pd.merge(df1, df3, left_on="employee", right_on="name"))
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df3

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df1, df3, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

The result has a redundant column that we can drop if desired—for example, byusing the drop()method of DataFrames:

```
In[8]:
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

Out[8]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The left_index and right_index keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

```
In[9]: df1a = df1.set_index('employee')
       df2a                          =
```

```
df2.set_index('employee')
print(df1a); print(df2a)
```

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

You can use the index as the key for merging by specifying the left_index and/or right_index flags in pd.merge():

```
In[10]:
print(df1a); print(df2a);
print(pd.merge(df1a, df2a, left_index=True, right_index=True))
```

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

pd.merge(df1a, df2a, left_index=True, right_index=True)

	group	hire_date
employee		
Lisa	Engineering	2004
Bob	Accounting	2008
Jake	Engineering	2012
Sue	HR	2014

For convenience, DataFrames implement the join() method, which performs amerge that defaults to joining on indices:

```
In[11]: print(df1a); print(df2a); print(df1a.join(df2a))
```

df1a

	group
employ	

df2a

	hire_date
employ	

ee ee
Bob Accounting Lisa 2004
Jake Engineerin Bob 2008
g
Lisa Engineerin Jake 2012
g
Sue HR Sue 2014
df1a.join(df2a)

group hire_dat
e

employee
Bob 2008
Accountin
g
Jake 2012
Engineerin
g
Lisa 2004
Engineerin
g
Sue 2014
H
R

If you'd like to mix indices and columns, you can combine left_indexwith right_on

or left_onwith right_indexto get the desired behavior:

```
In[12]:
print(df1a); print(df3);
print(pd.merge(df1a, df3, left_index=True, right_on='name'))
```

df1a df3

employ group nam salary
ee e
Bob Accounting 0 Bo 70000
b
Jake Engineerin 1 Jak 80000
g e
Lisa Engineerin 2 Lisa 12000
g 0
Sue HR 3 Su 90000
e

pd.merge(df1a, df3, left_index=True,
right_on='name')group name salary

0 Accounting Bob 70000
1 Engineering Jake 80000
2 Engineering Lisa
1200003 HR Sue 90000

Specifying Set Arithmetic for Joins

In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

```
In[13]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],

                            'food': ['fish', 'beans', 'bread']},
                           columns=['name', 'food'])

        df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],

                            'drink': ['wine', 'beer']},
                           columns=['name', 'drink'])

        print(df6); print(df7); print(pd.merge(df6, df7))
```

```
df6                   df7               pd.merge(df6, df7)
   name  food            name drink        name  food  drink
0  Peter fish         0  Mary   wine    0  Mary  bread wine
1  Paul  beans        1  Joseph beer
2  Mary  bread
```

Here we have merged two datasets that have only a single "name" entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the how keyword, which defaults to 'inner':

```
In[14]: pd.merge(df6, df7,
how='inner')Out[14]:        name
            food drink

         0  Mary  bread  wine
```

Other options for the how keyword are 'outer', 'left', and 'right'. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

```
In[15]: print(df6); print(df7); print(pd.merge(df6, df7,
how='outer'))df6   df7               pd.merge(df6, df7,
how='outer')

   name  food       name   drink      name   food  drink
0  Peter fish    0  Mary   wine   0   Peter  fish  NaN
1  Paul  beans   1  Joseph beer   1   Paul   beans NaN
2  Mary  bread                    2   Mary   bread wine
                                  3   Joseph NaN   beer
```

The *left join* and *right join* return join over the left entries and right entries, respec-tively. For example:

```
In[16]: print(df6); print(df7); print(pd.merge(df6, df7, how='left'))
df6                  df7                 pd.merge(df6, df7, how='left')
```

```
    name  food          name drink        name  food drink
0  Peter fish      0   Mary    wine  0   Peter   fish   NaN
1  Paul  beans     1   Joseph  beer  1   Paul    beans  NaN
2  Mary  bread                       2   Mary    bread  wine
```

The output rows now correspond to the entries in the left input. Using how='right' works in a similar manner.

All of these options can be applied straightforwardly to any of the preceding jointypes.

Overlapping Column Names: The suffixes Keyword

Finally, you may end up in a case where your two input DataFrames have conflicting column names. Consider this example:

```
In[17]: df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'rank': [1, 2, 3, 4]})
        df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                            'rank': [3, 1, 4, 2]})
        print(df8); print(df9); print(pd.merge(df8, df9, on="name"))
```

```
df8                df9               pd.merge(df8, df9, on="name")
   name  rank         name  rank        name rank_x rank_y
0  Bob     1       0  Bob     3      0  Bob      1      3
1  Jake    2       1  Jake    1      1  Jake     2      1
2  Lisa    3       2  Lisa    4      2  Lisa     3      4
3  Sue     4       3  Sue     2      3  Sue      4      2
```

Because the output would have two conflicting column names, the merge function automatically appends a suffix _x or _y to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the suffixes keyword:

```
In[18]:
print(df8); print(df9);
print(pd.merge(df8, df9, on="name", suffixes=["_L", "_R"]))
```

```
df8                df9
   name  rank         name  rank
0  Bob     1       0  Bob     3
1  Jake    2       1  Jake    1
2  Lisa    3       2  Lisa    4
3  Sue     4       3  Sue     2
```

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
   name rank_L rank_R
0  Bob     1      3
1  Jake    2      1
2  Lisa    3      4
```

3 Sue 4 2

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

Day 05- Descriptive Statistics

Cleansing Data with Pandas

Example: US States Data

Merge and join operations come up most often when one is combining data from different sources. Here we will consider an example of some data about US states and their populations. The data files can be found at *http://github.com/jakevdp/data-USstates/*:

```
In[19]:
# Following are shell commands to download the data

# !curl -O
https://raw.githubusercontent.com/jakevdp/#
      data-USstates/master/state-
population.csv

# !curl  -O
https://raw.githubusercontent.com/jakevdp/#
      data-USstates/master/state-areas.csv

# !curl  -O
https://raw.githubusercontent.com/jakevdp/#
      data-USstates/master/state-abbrevs.csv
```

Let's take a look at the three datasets, using the Pandas read_csv()function:

```
In[20]: pop = pd.read_csv('state-population.csv')
        areas = pd.read_csv('state-areas.csv')
        abbrevs = pd.read_csv('state-
        abbrevs.csv')

        print(pop.head()); print(areas.head()); print(abbrevs.head())
```

pop.head()

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

areas.head()

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
3	Arkansas	53182
4	California	163707

abbrevs.head()

state abbreviation

```
0 Alabama      AL
1 Alaska       AK
2 Arizona      AZ
3 Arkansas     AR
4 California   CA
```

Given this information, say we want to compute a relatively straightforward result: rank US states and territories by their 2010 population density. We clearly have the data here to find this result, but we'll have to combine the datasets to get it.

We'll start with a many-to-one merge that will give us the full state name within the population DataFrame. We want to merge based on the state/regioncolumn of pop,and the abbreviation column of abbrevs. We'll use how='outer' to make sure no data is thrown away due to mismatched labels.

```
In[21]: merged = pd.merge(pop, abbrevs, how='outer',
                          left_on='state/region', right_on='abbreviation')
        merged = merged.drop('abbreviation', 1) # drop duplicate
        info merged.head()
```

```
Out[21]: state/region   ages     year   population   state
     0           AL     under18  2012   1117489.0    Alabama
     1           AL     total    2012   4817528.0    Alabama
     2           AL     under18  2010   1130966.0    Alabama
     3           AL     total    2010   4785570.0    Alabama
     4           AL     under18  2011   1125763.0    Alabama
```

Let's double-check whether there were any mismatches here, which we can do by looking for rows with nulls:

```
In[22]: merged.isnull().any()
```

```
Out[22]: state/region   False
         ages           False
         year           False
         population      True
         state           True
         dtype: bool
```

Some of the populationinfo is null; let's figure out which these are!

```
In[23]: merged[merged['population'].isnull()].head()
```

```
Out[23]: state/region   ages     year   population   state
         2448   PR      under18  1990   NaN          NaN
         2449   PR      total    1990   NaN          NaN
         2450   PR      total    1991   NaN          NaN
```

2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

More importantly, we see also that some of the new stateentries are also null, whichmeans that there was no corresponding entry in the abbrevs key! Let’s figure out which regions lack this match:

```
In[24]: merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

```
Out[24]: array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in thestate abbreviation key. We can fix these quickly by filling in appropriate entries:

```
In[25]: merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
        merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
        merged.isnull().any()
```

```
Out[25]: state/region   False
         ages           False
         year           False
         population      True
         state          False
         dtype: bool
```

No more nulls in the statecolumn: we’re all set!

Now we can merge the result with the area data using a similar procedure. Examiningour results, we will want to join on the statecolumn in both:

```
In[26]: final = pd.merge(merged, areas, on='state', how='left')
        final.head()
```

Out[26]:

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0

3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Again, let's check for nulls to see if there were any mismatches:

```
In[27]: final.isnull().any()
```

```
Out[27]: state/region     False
         ages             False
         year             False
         population       True
         state            False
         area (sq. mi)    True
         dtype: bool
```

There are nulls in the area column; we can take a look to see which regions were ignored here:

```
In[28]: final['state'][final['area (sq. mi)'].isnull()].unique()
```

```
Out[28]: array(['United States'], dtype=object)
```

We see that our areas DataFrame does not contain the area of the United States as a whole. We could insert the appropriate value (using the sum of all state areas, for instance), but in this case we'll just drop the null values because the population den- sity of the entire United States is not relevant to our current discussion:

```
In[29]: final.dropna(inplace=True)
        final.head()
```

Out[29]:	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Now we have all the data we need. To answer the question of interest, let's first selectthe portion of the data corresponding with the year 2000, and the total population. We'll use the query() function to do this quickly (this requires the numexpr packageto be installed;

```
In[30]: data2010 = final.query("year == 2010 & ages ==
        'total'")data2010.head()
```

```
Out[30]:     state/region  ages  year  population     state  area (sq. mi)3
                       AL  total  2010  4785570.0   Alabama       52423.0

        91             AK  total  2010   713868.0    Alaska      656425.0
        101     AZ         total  201    6408790  Arizona       114006.0
                                  0
        189     AR         total  201    2922280  Arkansas      53182.0
                                  0
        197     CA         total  201    3733360  California    163707.0
                                  0         1
```

Now let's compute the population density and display it in order. We'll start by rein- dexing our data on the state, and then compute the result:

```
In[31]: data2010.set_index('state', inplace=True)

        density = data2010['population'] / data2010['area (sq. mi)']
```

```
In[32]: density.sort_values(ascending=False,
        inplace=True)density.head()
```

```
Out[32]: state
        District of Columbia  8898.89705
                              9
        Puerto Rico           1058.66514
                              9
        New Jersey            1009.25326
                              8
        Rhode Island          681.339159
        Connecticut           645.600649
        dtype: float64
```

The result is a ranking of US states plus Washington, DC, and Puerto Rico in order oftheir 2010 population density, in residents per square mile. We can see that by far thedensest region in this dataset is Washington, DC (i.e., the District of Columbia); among states, the densest is New Jersey.

We can also check the end of the list:

```
In[33]: density.tail()
```

```
Out[33]: state
South Dakota    10.583512
North Dakota     9.537565
Montana          6.736171
Wyoming          5.768079
Alaska           1.087509
dtype:
float64
```

We see that the least dense state, by far, is Alaska, averaging slightly over one residentper square mile.

This type of messy data merging is a common task when one is trying to answer questions using real-world data sources. I hope that this example has given you an idea of the ways you can combine tools we've covered in order to gain insight from your data!

Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like sum(), mean(), median(), min(), and max(), in which a single num-ber gives insight into the nature of a potentially large dataset. In this section, we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a groupby.

Planets Data

Here we will use the Planets dataset, available via the Seaborn package (see "Visuali- zation with Seaborn" on page 311). It gives information on planets that astronomers have discovered around other stars (known as extrasolar planets or exoplanets for short). It can be downloaded with a simple Seaborn command:

```
In[2]: import seaborn as sns

       planets = sns.load_dataset('planets')
       planets.shape

Out[2]: (1035, 6)
```

```
In[3]: planets.head()
```

Out[3]:	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

This has some details on the 1,000+ exoplanets discovered up to 2014.

Simple Aggregation in Pandas

Earlier we explored some of the data aggregations available for NumPy arrays ("Aggregations: Min, Max, and Everything in Between" on page 58). As with a one-dimensional NumPy array, for a Pandas Seriesthe aggregates return a single value:

```
In[4]: rng =
       np.random.RandomState(42)
       ser = pd.Series(rng.rand(5)) ser

Out[4]: 0   0.374540
1           0.950714
2           0.731994
3           0.598658
4           0.156019
        dtype: float64
```

```
In[5]: ser.sum()

Out[5]: 2.8119254917081569
```

```
In[6]: ser.mean()

Out[6]: 0.56238509834163142
```

For a DataFrame, by default the aggregates return results within each column:

```
In[7]: df = pd.DataFrame({'A': rng.rand(5),
                          'B': rng.rand(5)})
       df
```

Out[7]:	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

```
In[8]: df.mean()

Out[8]: A   0.477888
        B   0.443420
        dtype: float64
```

By specifying the axis argument, you can instead aggregate within each row:

```
In[9]: df.mean(axis='columns')

Out[9]: 0   0.088290
1           0.513997
2           0.849309
3           0.406727
4           0.444949
        dtype: float64
```

Pandas Series and DataFrames include all of the common aggregates mentioned in "Aggregations: Min, Max, and Everything in Between" on page 58; in addition, thereis a convenience method describe() that computes several common aggregates for each column and returns the result. Let's use this on the Planets data, for now drop- ping rows with missing values:

```
In[10]: planets.dropna().describe()
```

Out[10]:	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510

std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

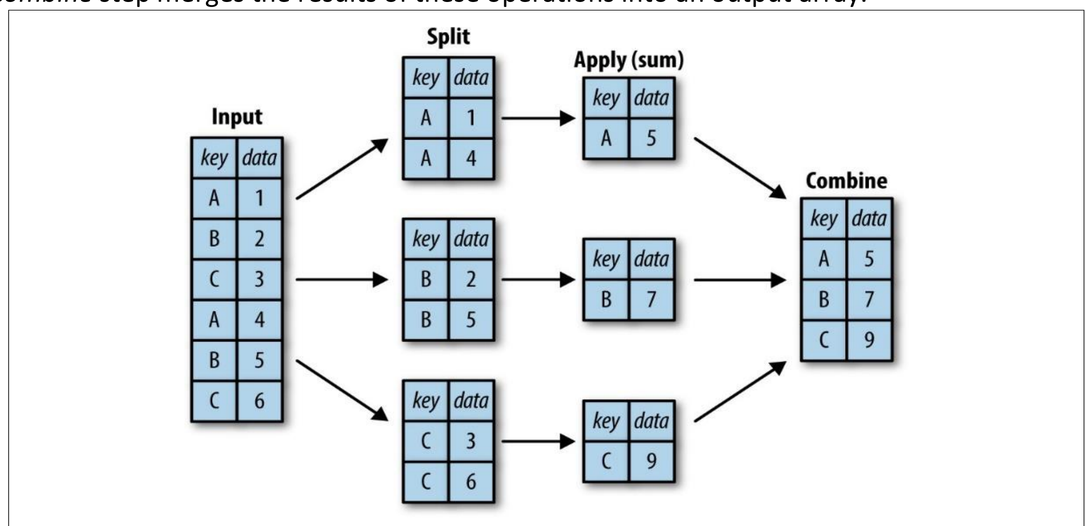
This can be a useful way to begin understanding the overall properties of a dataset. For example, we see in the year column that although exoplanets were discovered asfar back as 1989, half of all known exoplanets were not discovered until 2010 or after.This is largely thanks to the *Kepler* mission, which is a space-based telescope specifi-cally designed for finding eclipsing planets around other stars.

GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so- called groupby operation. The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.

Split, apply, combine

- The *split* step involves breaking up and grouping a DataFramedepending on thevalue of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transfor-mation, or filtering, within the individual groups.
- The *combine* step merges the results of these operations into an output array.



While we could certainly do this manually using some combination of the masking, aggregation, and merging commands covered earlier, it's important to realize that *the intermediate splits do not need to be explicitly instantiated*. Rather, the GroupBy can (often) do this in a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way. The power of the GroupBy is that it abstracts away these steps: the user need not think about *how* the computation isdone under the hood, but rather thinks about the *operation as a whole*.

We'll start by creating the input DataFrame:

```
In[11]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data': range(6)}, columns=['key', 'data'])
        df
```

Out[11]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

We can compute the most basic split-apply-combine operation with the groupby() method of DataFrames, passing the name of the desired key column:

```
In[12]: df.groupby('key')

Out[12]: <pandas.core.groupby.DataFrameGroupBy object at 0x117272160>
```

Notice that what is returned is not a set of DataFrames, but a DataFrameGroupBy object. This object is where the magic is: you can think of it as a special view of the DataFrame, which is poised to dig into the groups but does no actual computation until the aggregation is applied. This "lazy evaluation" approach means that commonaggregates can be implemented very efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this DataFrameGroupBy object, which will perform the appropriate apply/combine steps to produce the desired result:

```
In[13]: df.groupby('key').sum()
```

Data	key
A	3

B 5

C 7

The sum() method is just one possibility here; you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid DataFrame operation, as we will see in the following discussion.

The GroupBy object

The GroupBy object is a very flexible abstraction. In many ways, you can simply treatit as if it's a collection of DataFrames, and it does the difficult things under the hood. Let's see some examples using the Planets data.

Perhaps the most important operations made available by a GroupBy are *aggregate*, *filter*, *transform*, and *apply*. We'll discuss each of these more fully in "Aggregate, filter, transform, apply" on page 165, but before that let's introduce some of the other functionality that can be used with the basic GroupByoperation.

Column indexing

The GroupBy object supports column indexing in the same way asthe DataFrame, and returns a modified GroupByobject. For example:

```
In[14]: planets.groupby('method')

Out[14]: <pandas.core.groupby.DataFrameGroupBy object at

0x1172727b8>In[15]: planets.groupby('method')['orbital_period']

Out[15]: <pandas.core.groupby.SeriesGroupBy object at 0x117272da0>
```

Here we've selected a particular Seriesgroup from the original DataFramegroup by reference to its column name. As with the GroupBy object, no computation is done until we call some aggregate on the object:

```
In[16]: planets.groupby('method')['orbital_period'].median()

Out[16]: method
    Astrometry                        631.180000
    Eclipse Timing Variations        4343.500000
    Imaging                         27500.000000
    Microlensing                     3300.000000
    Orbital Brightness Modulation       0.342887
    Pulsar Timing                      66.541900
```

```
Pulsation Timing Variations      1170.000000
Radial Velocity                   360.200000
Transit                             5.714932
Transit Timing Variations          57.011000
Name: orbital_period, dtype: float64
```

Iteration over groups. The GroupBy object supports direct iteration over the groups, returning each group as a Seriesor DataFrame:

```
In[17]: for (method, group) in planets.groupby('method'):
            print("{0:30s} shape={1}".format(method, group.shape))
```

```
Astrometry                    shape=(2, 6)
Eclipse Timing Variations     shape=(9, 6)
Imaging                       shape=(38,
6)

Microlensing                  shape=(23,
6) Orbital Brightness Modulation
shape=(3, 6) Pulsar Timing    shape=(5, 6)
Pulsation Timing Variations   shape=(1, 6)
Radial Velocity               shape=(553,
6)

Transit                       shape=(397, 6)

Transit Timing Variations     shape=(4, 6)
```

This can be useful for doing certain things manually, though it is often much faster touse the built-in applyfunctionality, which we will discuss momentarily.

Dispatch methods. Through some Python class magic, any method not explicitly implemented by the GroupByobject will be passed through and called on the groups, whether they are DataFrame or Series objects. For example, you can use the describe() method of DataFrames to perform a set of aggregations that describe eachgroup in the data:

```
In[18]: planets.groupby('method')['year'].describe().unstack()
```

Out[18]: method	count	mean	std	min	25% \\
Astrometry	2.0	2011.500000	2.121320	2010.0	2010.75
Eclipse Timing Variations	9.0	2010.000000	1.414214	2008.0	2009.00
Imaging	38.0	2009.131579	2.781901	2004.0	2008.00

Microlensing	23.0	2009.782609	2.859697	2004.0	2008.00
Orbital Brightness Modulation	3.0	2011.666667	1.154701	2011.0	2011.00

Aggregate, filter, transform, apply.

The preceding discussion focused on aggregation for the combine operation, but there are more options available. In particular, GroupBy objects have aggregate(), filter(), transform(), and apply() methods that efficiently implement a variety ofuseful operations before combining the grouped data.

For the purpose of the following subsections, we'll use this DataFrame:

```
In[19]: rng = np.random.RandomState(0)
        df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                           'data1': range(6),
                           'data2': rng.randint(0, 10, 6)},
                           columns = ['key', 'data1', 'data2'])
        df
```

Out[19]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation. We're now familiar with GroupBy aggregations with sum(), median(), and the like, but the aggregate()method allows for even more flexibility. It can takea string, a function, or a list thereof, and compute all the aggregates at once. Here is aquick example combining all these:

```
In[20]: df.groupby('key').aggregate(['min', np.median, max])
```

Out[20]:

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Another useful pattern is to pass a dictionary mapping column names to operationsto be applied on that column:

```
In[21]: df.groupby('key').aggregate({'data1': 'min',
```

```
'data2': 'max'})
```

Out[21]: data1 data2key

A	0	5
B	1	7
C	2	9

Filtering. A filtering operation allows you to drop data based on the group proper- ties. For example, we might want to keep all groups in which the standard deviation islarger than some critical value:

In[22]:

```
def filter_func(x):
    return x['data2'].std() > 4


print(df); print(df.groupby('key').std());
print(df.groupby('key').filter(filter_func))
```

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').std()

key	data1	data2
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

df.groupby('key').filter(filter_func)

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

The filter() function should return a Boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the result.

Transformation. While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean:

```
In[23]: df.groupby('key').transform(lambda x: x - x.mean())Out[23]:  data1 data2
0  -1.5   1.0
1  -1.5  -3.5
2  -1.5  -3.0
3   1.5  -1.0
4   1.5   3.5
5   1.5   3.0
```

The apply() method. The apply() method lets you apply an arbitrary function to the group results. The function should take a DataFrame, and return either a Pandasobject (e.g., DataFrame, Series) or a scalar; the combine operation will be tailored tothe type of output returned.

For example, here is an apply() that normalizes the first column by the sum of the second:

```
In[24]: def norm_by_data2(x):
            # x is a DataFrame of group valuesx['data1'] /= x['data2'].sum() return x

        print(df);
print(df.groupby('key').apply(norm_by_data2))df
df.groupby('key').apply(norm_by_data2)
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3

apply()within a GroupByis quite flexible: the only criterion is that the function takesa DataFrame and returns a Pandas object or scalar; what you do in the middle is up toyou!

Specifying the split key

In the simple examples presented before, we split the DataFrame on a single column name. This is just one of many options by which the groups can be defined, and we'llgo through some other options for group specification here.

A list, array, series, or index providing the grouping keys. The key can be any series or list with a length matching that of the DataFrame. For example:

```
In[25]: L = [0, 1, 0, 1, 2, 0]
print(df);  print(df.groupby(L).sum())
df                     df.groupby(L).sum()

  key  data1  data2       data1
data20    A      0     5  0          7
    17

1   B      1      0     1      4      3
2   C      2      3     2      4      7
3   A      3      3
4   B      4      7
5   C      5      9
```

Of course, this means there's another, more verbose way of accomplishing the df.groupby('key') from before:

```
In[26]:  print(df);
print(df.groupby(df['key']).sum())df
key    data1 data2    data1         data
                          df.groupby(df['key']).sum
0   A 0      5        A        3     8
1() B 1      0        B        5     7
2   C 2      3        C        7     12
3   A 3      3
4   B 4      7
```

A dictionary or series mapping index to group. Another method is to provide a dictionary that maps index values to the group keys:

```
In[27]: df2 = df.set_index('key')
        mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
        print(df2); print(df2.groupby(mapping).sum())
```

df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(mapping).sum()

	data1	data2
consonant	12	19
vowel	3	8

Week 3- Data Cleaning and Summarization

Descriptive statistics and data summarization

Data visualization using Matplotlib and Seaborn

Exploring relationships and patterns in data

Day 01- Pandas String Operations

We saw in previous sections how tools like NumPy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. For example:

```
In[1]: import numpy as np
       x = np.array([2, 3, 5, 7, 11, 13])
       x * 2

Out[1]: array([ 4,  6, 10, 14, 22, 26])
```

This *vectorization* of operations simplifies the syntax of operating on arrays of data: we no longer have to worry about the size or shape of the array, but just about what operation we want done. For arrays of strings, NumPy does not provide such simple access, and thus you're stuck using a more verbose loop syntax:

```
In[2]: data = ['peter', 'Paul', 'MARY', 'gUIDO']
       [s.capitalize() for s in data]

Out[2]: ['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with some data, but it will break if there are any missing values. For example:

```
In[3]: data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
       [s.capitalize() for s in data]
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the str attribute of Pandas Series and Index objects containing strings. So, for example, suppose we create a Pandas Serieswith this data:

```
In[4]: import pandas as pd
       names =
       pd.Series(data)names
```

```
Out[4]: 0     peter
        1      Paul
        2     None
        3      MARY
        4     gUIDO
        dtype:
        object
```

We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
In[5]: names.str.capitalize()
Out[5]: 0     Peter
        1      Paul
        2      None
        3      Mary
        4     Guido
        dtype:
        object
```

Using tab completion on this str attribute will list all the vectorized string methods available to Pandas.

Tables of Pandas String Methods

If you have a good understanding of string manipulation in Python, most of Pandas' string syntax is intuitive enough that it's probably sufficient to just list a table of avail-able methods; we will start with that here, before diving deeper into a few of the sub-tleties. The examples in this section use the following series of names:

```
In[6]: monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                          'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

Methods similar to Python string methods

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized stringmethod. Here is a list of Pandas strmethods that mirror Python string methods:

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()

zfill() index() isalpha() split()

strip() rindex() isdigit() rsplit()

rstrip() capitalize() isspace() partition()

lstrip() swapcase() istitle() rpartition()

Notice that these have various return values. Some, like lower(), return a series of strings:

```
In[7]: monte.str.lower()
Out[7]: 0     graham
chapman
1         john cleese
2       terry gilliam
3           eric idle
4         terry jones
5       michael
palindtype:
object
```

But some others return numbers:

```
In[8]: monte.str.len()

Out[8]: 0   14
1           11
2           13
3           9
4           11
5           13
dtype: int64
```

Or Boolean values:

```
In[9]: monte.str.startswith('T')
Out[9]: 0     False
1     False
2      True
3     False
4      True
5     False
dtype:
bool
```

Still others return lists or other compound values for each element:

```
In[10]: monte.str.split()
Out[10]: 0    [Graham,
Chapman]

        1       [John, Cleese]
        2     [Terry, Gilliam]
        3         [Eric, Idle]
        4       [Terry, Jones]
        5     [Michael,
        Palin]dtype: object
```

We'll see further manipulations of this kind of series-of-lists object as we continue our discussion.

Methods using regular expressions

In addition, there are several methods that accept regular expressions to examine the content of each string element, and follow some of the API conventions of Python's built-in remodule.

Table . Mapping between Pandas methods and functions in Python's re module

Method	Description
match()	Call re.match()on each element, returning a Boolean.
extract()	Call re.match() on each element, returning matched groups as strings.
findall()	Call re.findall()on each element.
replace()	Replace occurrences of pattern with some other string.
contains()	Call re.search() on each element, returning a Boolean.
count()	Count occurrences of pattern.
split()	Equivalent to str.split(), but accepts regexps.
rsplit()	Equivalent to str.rsplit(), but accepts regexps.

With these, you can do a wide range of interesting operations. For example, we can extract the first name from each by asking for a contiguous group of characters at the beginning of each element:

```
In[11]: monte.str.extract('([A-Za-z]+)')

Out[11]: 0     Graham
```

```
1        John
2       Terry
3        Eric
4       Terry
5     Michae
l dtype:
object
```

Or we can do something more complicated, like finding all names that start and end with a consonant, making use of the start-of-string (^) and end-of-string ($) regular expression characters:

```
In[12]:
monte.str.findall(r'^[^AEIOU].*[^aeiou]$')
Out[12]: 0    [Graham Chapman]

         1                  []

         2     [Terry Gilliam]

         3                  []

         4       [Terry Jones]
         5     [Michael
         Palin]dtype: object
```

The ability to concisely apply regular expressions across Series or DataFrame entries opens up many possibilities for analysis and cleaning of data.

Vectorized item access and slicing. The get() and slice() operations, in particular, enable vectorized element access from each array. For example, we can get a slice of the first three characters of each array using str.slice(0, 3). Note that this behav- ior is also available through Python's normal indexing syntax—for example, df.str.slice(0, 3)is equivalent to df.str[0:3]:

```
In[13]:
monte.str[0:3]
Out[13]: 0    Gra

         1    Joh
         2    Ter
         3    Eri
         4    Ter
         5    Mic
         dtype: object
```

Indexing via df.str.get(i) and df.str[i] is similar.

These get()and slice()methods also let you access elements of arrays returned

bysplit(). For example, to extract the last name of each entry, we can combinesplit()and get():

```
In[14]: monte.str.split().str.get(-1)
Out[14]: 0    Chapman
1     Cleese
2    Gilliam
3       Idle
4      Jones
5      Palin
dtype: object
```

Indicator variables. Another method that requires a bit of extra explanation is the get_dummies() method. This is useful when your data has a column containing somesort of coded indicator. For example, we might have a dataset that contains informa- tion in the form of codes, such as A="born in America," B="born in the United King-dom," C="likes cheese," D="likes spam":

```
In[15]:
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C',
                           'B|C|D']})
full_monte
Out[15]:    info           name
0  B|C|D  Graham Chapman
1    B|D     John Cleese
2    A|C   Terry Gilliam
3    B|D       Eric Idle
4    B|C     Terry Jones
5  B|C|D   Michael Palin
```

The get_dummies()routine lets you quickly split out these indicator variables into a DataFrame:

```
In[16]: full_monte['info'].str.get_dummies('|')
```

```
Out[16]:    A  B  C  D
         0  0  1  1  1
         1  0  1  0  1
         2  1  0  1  0
         3  0  1  0  1
         4  0  1  1  0
         5  0  1  1  1
```

With these operations as building blocks, you can construct an endless range of stringprocessing procedures when cleaning your data.

Miscellaneous methods

Finally, there are some miscellaneous methods that enable other convenient operations.

Table . Other Pandas string methods

Method	Description
get()	Index each element
slice()	Slice each element
slice_replace()	Replace slice in each element with passed value
cat()	Concatenate strings
repeat()	Repeat values
normalize()	Return Unicode form of string
pad()	Add whitespace to left, right, or both sides of strings
wrap()	Split long strings into lines with length less than a given
width join()	Join strings in each element of the Series with passed
separatorget_dummies()	Extract dummy variables as a DataFrame

Dates and Times in Python

The Python world has several available representations of dates, times, deltas, and timespans. While the time series tools provided by Pandas tend to be the most useful for data science applications, it is helpful to see their relationship to other packages used in Python.

Native Python dates and times: datetime and dateutil

Python’s basic objects for working with dates and times reside in the built-in date time module. Along with the third-party dateutil module, you can use it to quicklyperform a

host of useful functionalities on dates and times. For example, you can manually build a date using the datetimetype:

```
In[1]: from datetime import datetime
       datetime(year=2015, month=7,
       day=4)

Out[1]: datetime.datetime(2015, 7, 4, 0, 0)
```

Or, using the dateutilmodule, you can parse dates from a variety of string formats:

```
In[2]: from dateutil import parser

       date = parser.parse("4th of July,
       2015")date

Out[2]: datetime.datetime(2015, 7, 4, 0, 0)
```

Once you have a datetimeobject, you can do things like printing the day of the week:

```
In[3]: date.strftime('%A')

Out[3]: 'Saturday'
```

In the final line, we've used one of the standard string format codes for printing dates ("%A"), which you can read about in the strftime section of Python's datetime documentation. Documentation of other useful date utilities can be found in dateutil's online documentation. A related package to be aware of is pytz, which contains toolsfor working with the most migraine-inducing piece of time series data: time zones.

The power of datetime and dateutil lies in their flexibility and easy syntax: you can use these objects and their built-in methods to easily perform nearly any operation you might be interested in. Where they break down is when you wish to work with large arrays of dates and times: just as lists of Python numerical variables are subopti-mal compared to NumPy-style typed numerical arrays, lists of Python datetimeobjects are suboptimal compared to typed arrays of encoded dates.

Typed arrays of times: NumPy's datetime64

The weaknesses of Python's datetime format inspired the NumPy team to add a set of native time series data type to NumPy. The datetime64dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly. The date time64requires a very specific input format:

```
In[4]: import numpy as np

       date = np.array('2015-07-04',
       dtype=np.datetime64)date
```

```
Out[4]: array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

Once we have this date formatted, however, we can quickly do vectorized operations on it:

```
In[5]: date +
np.arange(12)Out[5]:
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
       '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
       '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
        dtype='datetime64[D]')
```

Because of the uniform type in NumPy datetime64 arrays, this type of operation canbe accomplished much more quickly than if we were working directly with Python's datetime objects, especially as arrays get large.

One detail of the datetime64 and timedelta64 objects is that they are built on a *fundamental time unit*. Because the datetime64 object is limited to 64-bit precision, the range of encodable times is 2^{64} times this fundamental unit. In other words, date time64imposes a trade-off between *time resolution* and *maximum time span*.

For example, if you want a time resolution of one nanosecond, you only have enough information to encode a range of 2^{64} nanoseconds, or just under 600 years. NumPy will infer the desired unit from the input; for example, here is a day-based datetime:

```
In[6]: np.datetime64('2015-07-04')
Out[6]: numpy.datetime64('2015-
07-04')
```

Here is a minute-based datetime:

```
In[7]: np.datetime64('2015-07-04 12:00')
Out[7]: numpy.datetime64('2015-07-04T12:00')
```

Notice that the time zone is automatically set to the local time on the computer executing the code. You can force any desired fundamental unit using one of many for- mat codes; for example, here we'll force a nanosecond-based time:

```
In[8]: np.datetime64('2015-07-04 12:59:59.50', 'ns')
Out[8]: numpy.datetime64('2015-07-
04T12:59:59.500000000')
```

Table 3-6. Description of date and time codes

Code	Meaning	Time span (relative)	Time span (absolute)
Y	Year	± 9.2e18 years	[9.2e18 BC, 9.2e18 AD]
M	Month	± 7.6e17 years	[7.6e17 BC, 7.6e17 AD]
W	Week	± 1.7e17 years	[1.7e17 BC, 1.7e17 AD]
D	Day	± 2.5e16 years	[2.5e16 BC, 2.5e16 AD]
h	Hour	± 1.0e15 years	[1.0e15 BC, 1.0e15 AD]
m	Minute	± 1.7e13 years	[1.7e13 BC, 1.7e13 AD]
s	Second	± 2.9e12 years	[2.9e9 BC, 2.9e9 AD]
ms	Millisecond	± 2.9e9 years	[2.9e6 BC, 2.9e6 AD]
us	Microsecond	± 2.9e6 years	[290301 BC, 294241 AD]
ns	Nanosecond	± 292 years	[1678 AD, 2262 AD]
ps	Picosecond	± 106 days	[1969 AD, 1970 AD]
fs	Femtosecond	± 2.6 hours	[1969 AD, 1970 AD]
as	Attosecond	± 9.2 seconds	[1969 AD, 1970 AD]

For the types of data we see in the real world, a useful default is datetime64[ns], as itcan encode a useful range of modern dates with a suitably fine precision.

Finally, we will note that while the datetime64 data type addresses some of the defi- ciencies of the built-in Python datetime type, it lacks many of the convenient meth- ods and functions provided by datetime and especially dateutil. More information can be found in NumPy's datetime64 documentation.

Dates and times in Pandas: Best of both worlds

Pandas builds upon all the tools just discussed to provide a Timestamp object, which combines the ease of use of datetime and dateutil with the efficient storage and vectorized interface of numpy.datetime64. From a group of these Timestamp objects, Pandas can construct a DatetimeIndex that can be used to index data in a Series or DataFrame; we'll see many examples of this below.

For example, we can use Pandas tools to repeat the demonstration from above. We can parse a flexibly formatted string date, and use format codes to output the day of the week:

```
In[9]: import pandas as pd

       date = pd.to_datetime("4th of July,
       2015")date

Out[9]: Timestamp('2015-07-04 00:00:00')

In[10]:

date.strftime('%A')

Out[10]: 'Saturday'
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
In[11]: date + pd.to_timedelta(np.arange(12), 'D')

Out[11]: DatetimeIndex(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
                        '2015-07-08', '2015-07-09', '2015-07-10', '2015-07-11',
                        '2015-07-12', '2015-07-13', '2015-07-14', '2015-07-15'],
                       dtype='datetime64[ns]', freq=None)
```

In the next section, we will take a closer look at manipulating time series data with the tools provided by Pandas.

Pandas Time Series: Indexing by Time

Where the Pandas time series tools really become useful is when you begin to *index data by timestamps*. For example, we can construct a Series object that has time- indexed data:

```
In[12]: index = pd.DatetimeIndex(['2014-07-04', '2014-08-04',
                                  '2015-07-04', '2015-08-04'])
```

```
data = pd.Series([0, 1, 2, 3], index=index)
data
```

```
Out[12]: 2014-07-04  0
         2014-08-04  1
         2015-07-04  2
         2015-08-04  3
         dtype: int64
```

Now that we have this data in a Series, we can make use of any of the Seriesindex-ing patterns we discussed in previous sections, passing values that can be coerced intodates:

```
In[13]: data['2014-07-04':'2015-07-
04']Out[13]: 2014-07-04     0
         2014-08-04  1
         2015-07-04  2
         dtype: int64
```

There are additional special date-only indexing operations, such as passing a year to obtain a slice of all data from that year:

```
In[14]: data['2015']
Out[14]: 2015-07-04


2
         2015-08-04  3
         dtype: int64
```

Later, we will see additional examples of the convenience of dates-as-indices. But first,let's take a closer look at the available time series data structures.

Pandas Time Series Data Structures

This section will introduce the fundamental Pandas data structures for working with time series data:

For *time stamps*, Pandas provides the Timestamp type. As mentioned before, it isessentially a replacement for Python's native datetime, but is based on the more efficient numpy.datetime64 data type. The associated index structure isDatetimeIndex.

- For *time periods*, Pandas provides the Period type. This encodes a fixed-frequency interval based on numpy.datetime64. The associated index structure isPeriodIndex.
- For *time deltas* or *durations*, Pandas provides the Timedeltatype. Timedeltais amore efficient replacement for Python's native datetime.timedelta type, and is based on numpy.timedelta64. The associated index structure is TimedeltaIndex.

The most fundamental of these date/time objects are the Timestamp and DatetimeIndex objects. While these class objects can be invoked directly, it is more common to use the pd.to_datetime() function, which can parse a wide variety of formats. Pass-ing a single date to pd.to_datetime() yields a Timestamp; passing a series of dates by default yields a DatetimeIndex:

```
In[15]: dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                               '2015-Jul-6', '07-07-2015', '20150708'])
        dates
Out[15]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                        '2015-07-08'],
                       dtype='datetime64[ns]', freq=None)
```

Any DatetimeIndexcan be converted to a PeriodIndexwith the to_period()function with the addition of a frequency code; here we'll use 'D' to indicate daily frequency:

```
In[16]: dates.to_period('D')
Out[16]: PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                      '2015-07-08'],
                     dtype='int64', freq='D')
```

A TimedeltaIndexis created, for example, when one date is subtracted from another:

```
In[17]: dates - dates[0]
Out[17]:
TimedeltaIndex(['0 days', '1 days', '3 days', '4 days', '5 days'],
               dtype='timedelta64[ns]', freq=None)
```

Regular sequences: pd.date_range()

To make the creation of regular date sequences more convenient, Pandas offers a few functions for this purpose: pd.date_range() for timestamps, pd.period_range()forperiods, and pd.timedelta_range() for time deltas. We've seen that Python's

range()and NumPy's np.arange()turn a startpoint, endpoint, and optional stepsizeinto a sequence. Similarly, pd.date_range() accepts a start date, an end date, and anoptional frequency code to create a regular sequence of dates. By default, the fre- quency is one day:

```
In[18]: pd.date_range('2015-07-03', '2015-07-10')

Out[18]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                        '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                        dtype='datetime64[ns]', freq='D')
```

Alternatively, the date range can be specified not with a start- and endpoint, but with a startpoint and a number of periods:

```
In[19]: pd.date_range('2015-07-03', periods=8)

Out[19]: DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-05', '2015-07-06',
                        '2015-07-07', '2015-07-08', '2015-07-09', '2015-07-10'],
                        dtype='datetime64[ns]', freq='D')
```

You can modify the spacing by altering the freq argument, which defaults to D. For example, here we will construct a range of hourly timestamps:

```
In[20]: pd.date_range('2015-07-03', periods=8, freq='H')

Out[20]: DatetimeIndex(['2015-07-03 00:00:00', '2015-07-03 01:00:00',
                        '2015-07-03 02:00:00', '2015-07-03 03:00:00',
                        '2015-07-03 04:00:00', '2015-07-03 05:00:00',
                        '2015-07-03 06:00:00', '2015-07-03 07:00:00'],
                        dtype='datetime64[ns]', freq='H')
```

To create regular sequences of period or time delta values, the very similar pd.period_range()and pd.timedelta_range()functions are useful. Here are some monthly periods:

```
In[21]: pd.period_range('2015-07', periods=8,
freq='M')Out[21]:
PeriodIndex(['2015-07', '2015-08', '2015-09', '2015-10', '2015-11', '2015-12',
                  '2016-01', '2016-02'],
                 dtype='int64', freq='M')
```

And a sequence of durations increasing by an hour:

```
In[22]: pd.timedelta_range(0, periods=10,
freq='H')Out[22]:
TimedeltaIndex(['00:00:00', '01:00:00', '02:00:00', '03:00:00', '04:00:00',
                    '05:00:00', '06:00:00', '07:00:00', '08:00:00', '09:00:00'],
                   dtype='timedelta64[ns]', freq='H')
```

All of these require an understanding of Pandas frequency codes, which we'll summa-rize in the next section.

Frequencies and Offsets

Fundamental to these Pandas time series tools is the concept of a frequency or date offset. Just as we saw the D (day) and H (hour) codes previously, we can use such codes to specify any desired frequency spacing. Table 3-7 summarizes the main codes available.

Table 3-7. Listing of Pandas frequency codes

Code	Description	Code	Description
D	Calendar day	B	Business day
W	Weekly		
M	Month end	BM	Business month end
Q	Quarter end	BQ	Business quarter end
A	Year end	BA	Business year end
H	Hours	BH	Business hours
T	Minutes		
S	Seconds		
L	Milliseonds		
U	Microseconds		

N	Nanoseconds

The monthly, quarterly, and annual frequencies are all marked at the end of the specified period. Adding an S suffix to any of these marks it instead at the beginning (Table 3-8).

Table. Listing of start-indexed frequency codes

Code	Description
MS	Month start
BMS	Business month start
QS	Quarter start
BQS	Business quarter start
AS	Year start
BAS	Business year start

Additionally, you can change the month used to mark any quarterly or annual codeby adding a three-letter month code as a suffix:

- Q-JAN, BQ-FEB, QS-MAR, BQS-APR, etc.
- A-JAN, BA-FEB, AS-MAR, BAS-APR, etc.

In the same way, you can modify the split-point of the weekly frequency by adding a three-letter weekday code:

- W-SUN, W-MON, W-TUE, W-WED, etc.

On top of this, codes can be combined with numbers to specify other frequencies. Forexample, for a frequency of 2 hours 30 minutes, we can combine the hour (H) and minute (T) codes as follows:

```
In[23]: pd.timedelta_range(0, periods=9,
freq="2H30T")Out[23]:
TimedeltaIndex(['00:00:00', '02:30:00', '05:00:00', '07:30:00', '10:00:00',
```

```
               '12:30:00', '15:00:00', '17:30:00', '20:00:00'],
              dtype='timedelta64[ns]', freq='150T')
```

All of these short codes refer to specific instances of Pandas time series offsets, whichcan be found in the pd.tseries.offsetsmodule. For example, we can create a busi-ness day offset directly as follows:

```
In[24]: from pandas.tseries.offsets import BDay
        pd.date_range('2015-07-01', periods=5,
        freq=BDay())

Out[24]: DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
                        '2015-07-07'],
                       dtype='datetime64[ns]', freq='B')
```

For more discussion of the use of frequencies and offsets, see the "DateOffset objects"section of the Pandas online documentation.

Resampling, Shifting, and Windowing

The ability to use dates and times as indices to intuitively organize and access data is an important piece of the Pandas time series tools. The benefits of indexed data in general (automatic alignment during operations, intuitive data slicing and access,etc.) still apply, and Pandas provides several additional time series–specific operations.

We will take a look at a few of those here, using some stock price data as an example. Because Pandas was developed largely in a finance context, it includes some very spe-cific tools for financial data. For example, the accompanying pandas-datareader package (installable via conda install pandas-datareader) knows how to import financial data from a number of available sources, including Yahoo finance, GoogleFinance, and others. Here we will load Google's closing price history:

```
In[25]: from pandas_datareader import data

    goog = data.DataReader('GOOG', start='2004', end='2016',
                           data_source='google')
     goog.head()
```

Out[25]:

Date	Open	High	Low	Close	Volume
2004-08-19	49.96	51.98	47.93	50.12	NaN

2004-08-20	50.69	54.49	50.20	54.10	NaN
2004-08-23	55.32	56.68	54.47	54.65	NaN
2004-08-24	55.56	55.74	51.73	52.38	NaN
2004-08-25	52.43	53.95	51.89	52.95	NaN

For simplicity, we'll use just the closing price:

```
In[26]: goog = goog['Close']
```

We can visualize this using the plot() method, after the normal Matplotlib setupboilerplate :

```
In[27]: %matplotlib inline

        import matplotlib.pyplot as
        pltimport seaborn;
        seaborn.set()

In[28]: goog.plot();
```

Figure 3-5. Google's closing stock price over time

Resampling and converting frequencies

One common need for time series data is resampling at a higher or lower frequency.You can do this using the resample() method, or the much simpler asfreq()method. The primary difference between the two is that resample()is fundamentallya *data aggregation*, while asfreq()is fundamentally a *data selection*.

Taking a look at the Google closing price, let's compare what the two return when we down-sample the data. Here we will resample the data at the end of business year :

```
In[29]: goog.plot(alpha=0.5, style='-')
        goog.resample('BA').mean().plot(style='
        :')

        goog.asfreq('BA').plot(style='--');

        plt.legend(['input', 'resample', 'asfreq'],
                   loc='upper left');
```

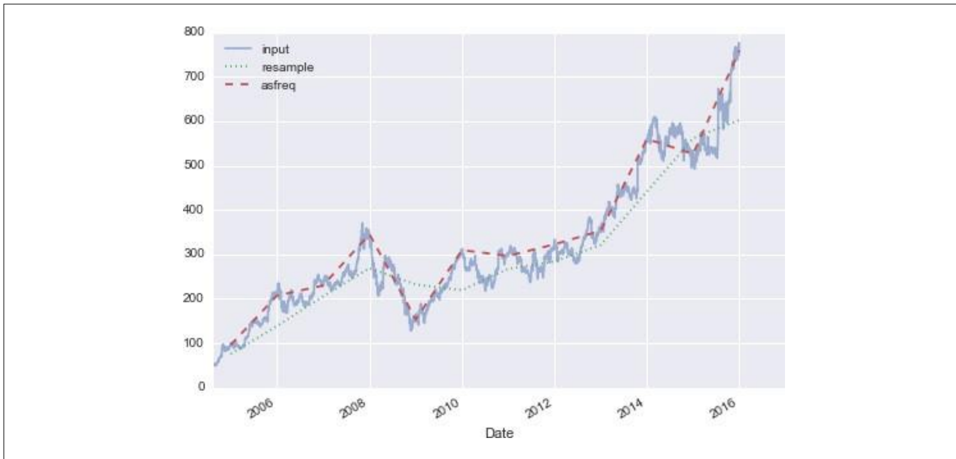


Figure . Resamplings of Google's stock price

Notice the difference: at each point, resample reports the *average of the previous year*, while asfreqreports the *value at the end of the year*.

For up-sampling, resample() and asfreq() are largely equivalent, though resamplehas many more options available. In this case, the default for both methods is to leavethe up-sampled points empty—that is, filled with NA values. Just as with the pd.fillna()function discussed previously, asfreq()accepts a methodargument tospecify how values are imputed. Here, we will resample the business day data at a daily frequency (i.e., including weekends); see Figure 3-7:

```
In[30]: fig, ax = plt.subplots(2, sharex=True)
        data = goog.iloc[:10]

        data.asfreq('D').plot(ax=ax[0], marker='o')

        data.asfreq('D', method='bfill').plot(ax=ax[1], style='-o')

        data.asfreq('D', method='ffill').plot(ax=ax[1], style='--o')
```

```
ax[1].legend(["back-fill", "forward-fill"]);
```

The top panel is the default: non-business days are left as NA values and do not appear on the plot. The bottom panel shows the differences between two strategiesfor filling the gaps: forward-filling and backward-filling.

Time-shifts

Another common time series–specific operation is shifting of data in time. Pandashas two closely related methods for computing this: shift() and tshift(). In short,the difference between them is that shift() *shifts the data*, while tshift() *shifts the index*. In both cases, the shift is specified in multiples of the frequency.

Here we will both shift()and tshift()by 900 days (Figure 3-8):

```
In[31]: fig, ax = plt.subplots(3, sharey=True)

        # apply a frequency to the data
        goog = goog.asfreq('D', method='pad')

        goog.plot(ax=ax[0])
        goog.shift(900).plot(ax=ax[1]
        )
        goog.tshift(900).plot(ax=ax[2
        ])

        # legends and annotations
        local_max = pd.to_datetime('2007-
        11-05')offset = pd.Timedelta(900, 'D')

        ax[0].legend(['input'], loc=2)
        ax[0].get_xticklabels()[4].set(weight='heavy',
        color='red')ax[0].axvline(local_max, alpha=0.3,
        color='red')

        ax[1].legend(['shift(900)'], loc=2)
        ax[1].get_xticklabels()[4].set(weight='heavy',
        color='red')ax[1].axvline(local_max + offset, alpha=0.3,
        color='red')
```

```
ax[2].legend(['tshift(900)'], loc=2)
ax[2].get_xticklabels()[1].set(weight='heavy',
```

```
color='red')ax[2].axvline(local_max + offset, alpha=0.3,
color='red');
```

Figure . Comparison between shift and tshift

We see here that shift(900) shifts the *data* by 900 days, pushing some of it off the end of the graph (and leaving NA values at the other end), while tshift(900) shifts the *index values* by 900 days.

A common context for this type of shift is computing differences over time. For example, we use shifted values to compute the one-year return on investment for Google stock over the course of the dataset (Figure 3-9):

```
In[32]: ROI = 100 * (goog.tshift(-365) / goog - 1)
        ROI.plot()

        plt.ylabel('% Return on Investment');
```

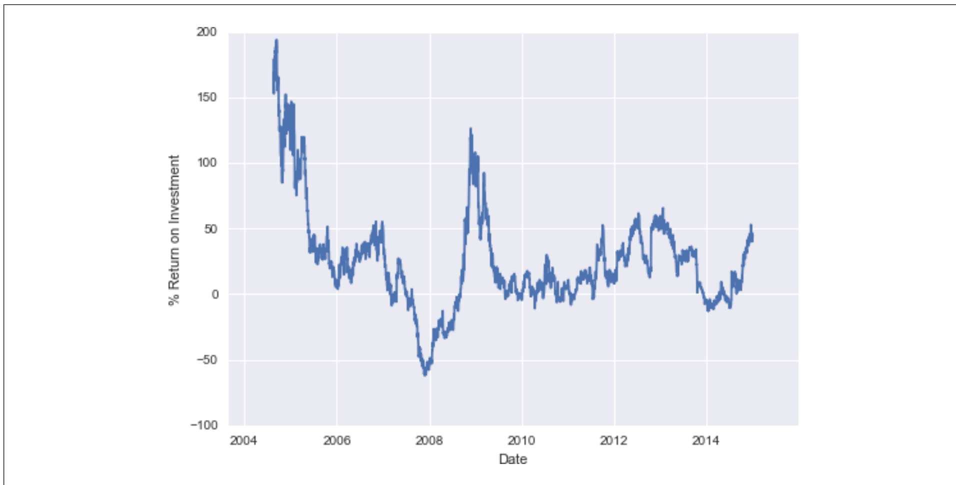


Figure . Return on investment to present day for Google stock

This helps us to see the overall trend in Google stock: thus far, the most profitable times to invest in Google have been (unsurprisingly, in retrospect) shortly after its IPO, and in the middle of the 2009 recession.

Rolling windows

Rolling statistics are a third type of time series–specific operation implemented by Pandas. These can be accomplished via the rolling() attribute of Series and DataFrame objects, which returns a view similar to what we saw with the groupby operation. This rolling view makes availablea number of aggregation operations by default.

For example, here is the one-year centered rolling mean and standard deviation of theGoogle stock prices (Figure 3-10):

```
In[33]: rolling = goog.rolling(365, center=True)


        data = pd.DataFrame({'input': goog,

                             'one-year rolling_mean':
                             rolling.mean(),'one-year rolling_std':
                             rolling.std()})

        ax = data.plot(style=['-', '--', ':'])
        ax.lines[0].set_alpha(0.3)
```

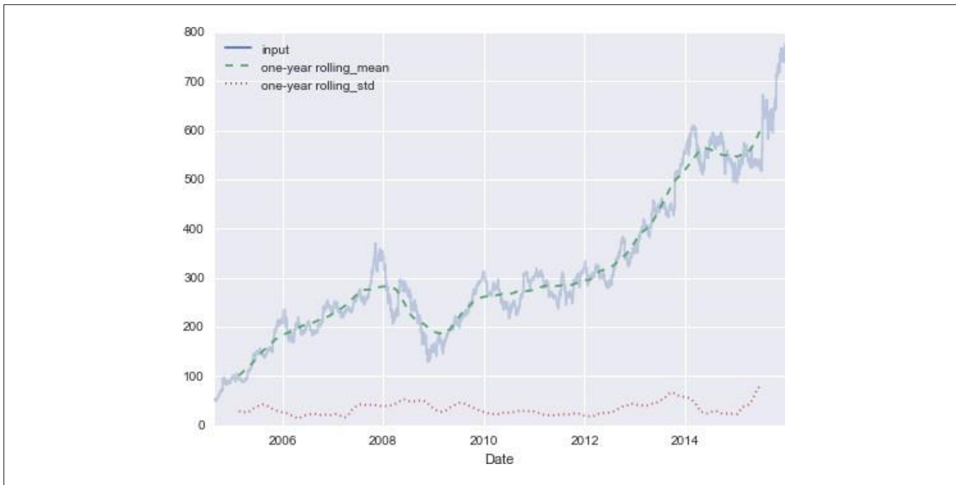


Figure 3-10. Rolling statistics on Google stock prices

As with groupby operations, the aggregate() and apply() methods can be used for custom rolling computations.

Where to Learn More

This section has provided only a brief summary of some of the most essential featuresof time series tools provided by Pandas; for a more complete discussion, you can referto the "Time Series/Date" section of the Pandas online documentation.

Another excellent resource is the textbook *Python for Data Analysis* by Wes McKin- ney (O'Reilly, 2012). Although it is now a few years old, it is an invaluable resource onthe use of Pandas. In particular, this book emphasizes time series tools in the context of business and finance, and focuses much more on particular details of business cal- endars, time zones, and related topics.

As always, you can also use the IPython help functionality to explore and try further options available to the functions and methods discussed here. I find this often is the best way to learn a new Python tool.

Example: Visualizing Seattle Bicycle Counts

As a more involved example of working with some time series data, let's take a look at bicycle counts on Seattle's Fremont Bridge. This data comes from an automated bicy-cle counter, installed in late 2012, which has inductive sensors on the east and west sidewalks of the bridge. The hourly bicycle counts can be downloaded from *http:// data.seattle.gov/;* here is the direct link to the dataset.

As of summer 2016, the CSV can be downloaded as follows:

In[34]:

```
# !curl -o FremontBridge.csv
# https://data.seattle.gov/api/views/65db-xm6k/rows.csv?accessType=DOWNLOAD
```

Once this dataset is downloaded, we can use Pandas to read the CSV output into a DataFrame. We will specify that we want the Date as an index, and we want these dates to be automatically parsed:

In[35]:

```
data = pd.read_csv('FremontBridge.csv', index_col='Date',
parse_dates=True)data.head()
```

Out[35]:

Date		Fremont Bridge West Sidewalk \\
2012-10-03	00:00:00	4.0
2012-10-03	01:00:00	4.0
2012-10-03	02:00:00	1.0
2012-10-03	03:00:00	2.0
2012-10-03	04:00:00	6.0

Date		Fremont Bridge East Sidewalk
2012-10-03	00:00:00	9.0
2012-10-03	01:00:00	6.0
2012-10-03	02:00:00	1.0
2012-10-03	03:00:00	3.0
2012-10-03	04:00:00	1.0

For convenience, we'll further process this dataset by shortening the column names and adding a "Total" column:

```
In[36]: data.columns = ['West', 'East']
        data['Total'] = data.eval('West + East')
```

Now let's take a look at the summary statistics for this data:

```
In[37]: data.dropna().describe()
```

Out[37]:

	West	East	Total
count	33544.000000	33544.000000	33544.000000
mean	61.726568	53.541706	115.268275
std	83.210813	76.380678	144.773983
min	0.000000	0.000000	0.000000
25%	8.000000	7.000000	16.000000
50%	33.000000	28.000000	64.000000
75%	80.000000	66.000000	151.000000
max	825.000000	717.000000	1186.000000

Visualizing the data

We can gain some insight into the dataset by visualizing it. Let's start by plotting theraw data:

```
In[38]: %matplotlib inline
        import seaborn; seaborn.set()
In[39]: data.plot()
        plt.ylabel('Hourly Bicycle Count');
```

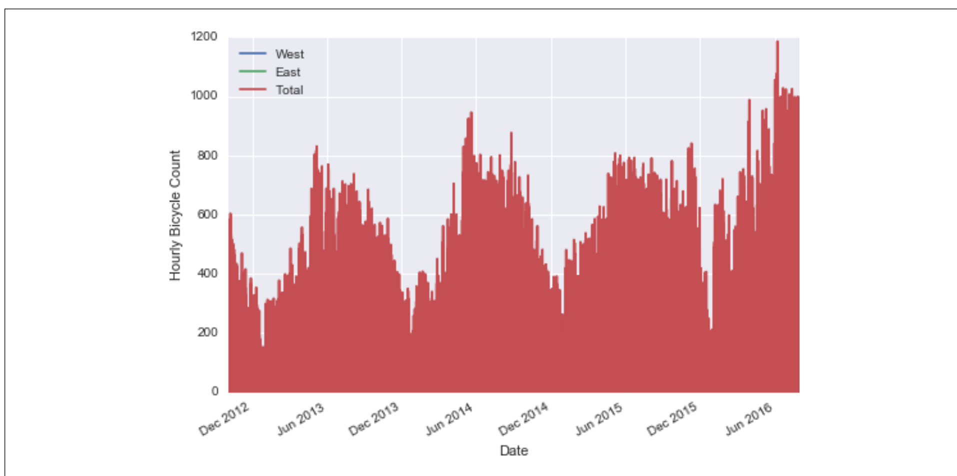


Figure . Hourly bicycle counts on Seattle's Fremont bridge

The ~25,000 hourly samples are far too dense for us to make much sense of. We can gain more insight by resampling the data to a coarser grid. Let's resample by week:

```
In[40]: weekly =
        data.resample('W').sum()
        weekly.plot(style=[':', '--', '-'])
        plt.ylabel('Weekly bicycle count');
```

This shows us some interesting seasonal trends: as you might expect, people bicycle more in the summer than in the winter, and even within a particular season the bicycle use varies from week to week .

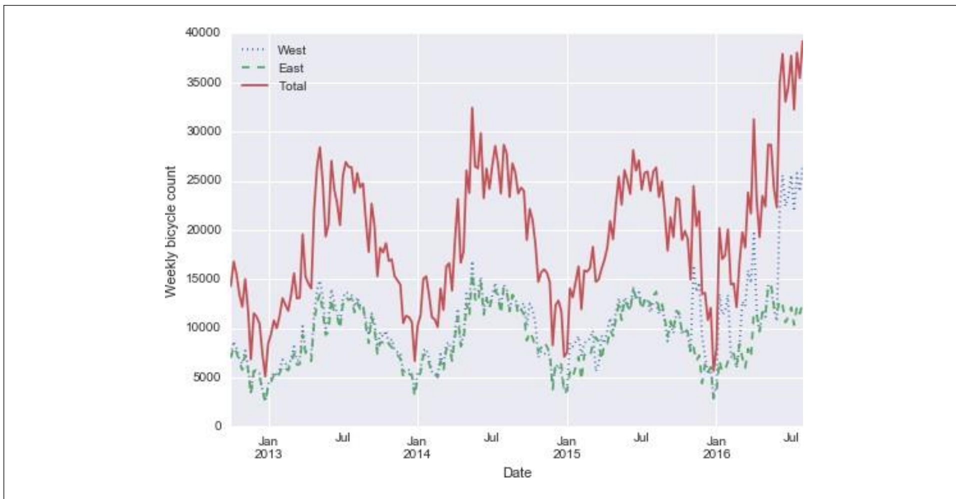


Figure . Weekly bicycle crossings of Seattle's Fremont bridge

Another way that comes in handy for aggregating the data is to use a rolling mean, utilizing the pd.rolling_mean()function. Here we'll do a 30-day rolling mean of ourdata, making sure to center the window :

```
In[41]: daily = data.resample('D').sum()
```

```
daily.rolling(30, center=True).sum().plot(style=[':', '--', '-'])
plt.ylabel('mean hourly count');
```

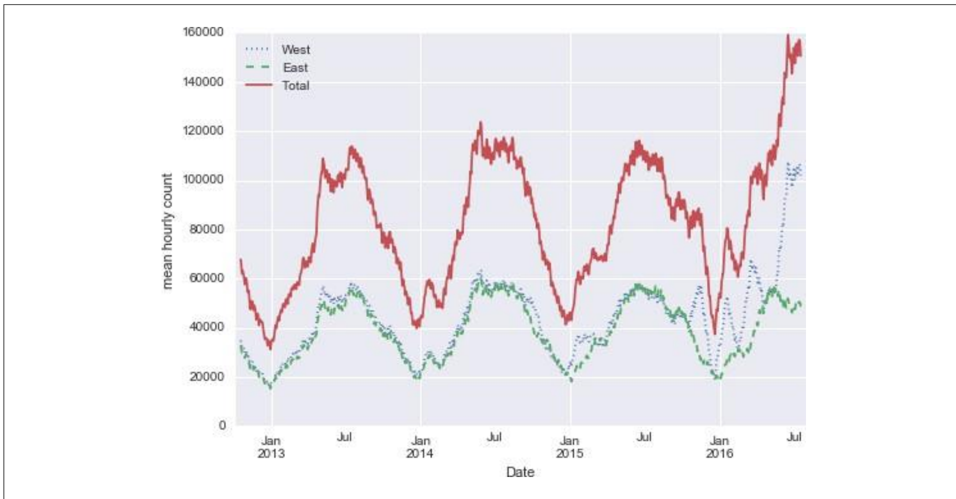


Figure . Rolling mean of weekly bicycle counts

The jaggedness of the result is due to the hard cutoff of the window. We can get a smoother version of a rolling mean using a window function—for example, a Gaus- sian window. The following code specifies both the widthof the window (we chose 50 days) and the width of the Gaussian within the window (we chose 10 days):

```
In[42]:
daily.rolling(50, center=True,
              win_type='gaussian').sum(std=10).plot(style=[':', '--', '-']);
```

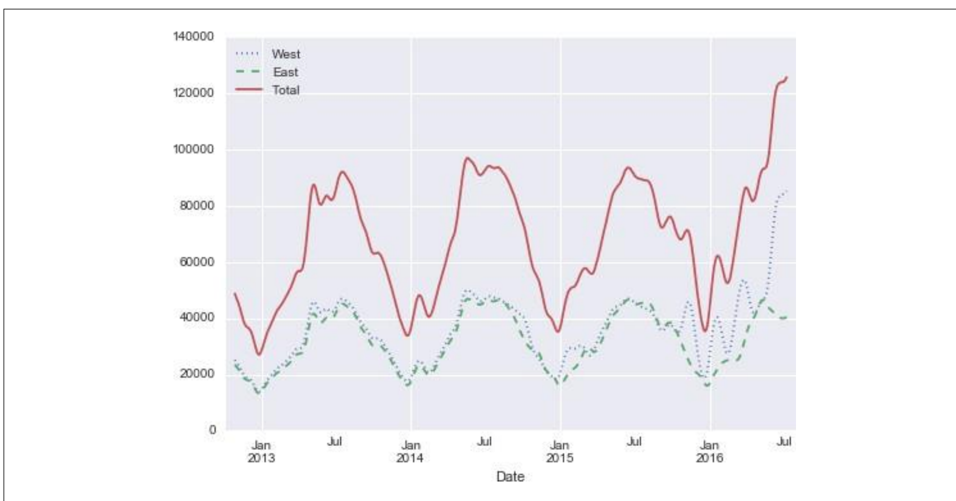


Figure. Gaussian smoothed weekly bicycle counts

Day-02: Digging into the data

While the smoothed data views in Figure are useful to get an idea of the general trend in the data, they hide much of the interesting structure. For example, we might want to look at the average traffic as a function of the time of day. We can do this using the GroupBy functionality discussed in:

```
In[43]: by_time = data.groupby(data.index.time).mean()
        hourly_ticks = 4 * 60 * 60 * np.arange(6)
        by_time.plot(xticks=hourly_ticks, style=[':', '--', '-']);
```

The hourly traffic is a strongly bimodal distribution, with peaks around 8:00 in the morning and 5:00 in the evening. This is likely evidence of a strong component of commuter traffic crossing the bridge. This is further evidenced by the differences between the western sidewalk (generally used going toward downtown Seattle), which peaks more strongly in the morning, and the eastern sidewalk (generally used going away from downtown Seattle), which peaks more strongly in the evening.

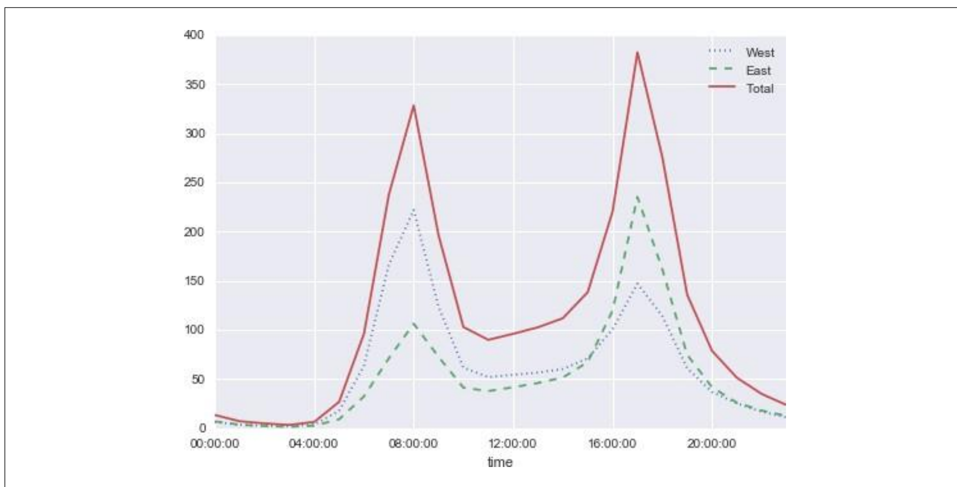


Figure . Average hourly bicycle counts

We also might be curious about how things change based on the day of the week. Again, we can do this with a simple groupby:

```
In[44]: by_weekday = data.groupby(data.index.dayofweek).mean()
        by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
```

```
by_weekday.plot(style=[':', '--', '-']);
```

Figure 3-16. Average daily bicycle counts

This shows a strong distinction between weekday and weekend totals, with around twice as many average riders crossing the bridge on Monday through Friday than on Saturday and Sunday.

With this in mind, let's do a compound groupby and look at the hourly trend on weekdays versus weekends. We'll start by grouping by both a flag marking the week-end, and the time of day:

```
In[45]: weekend = np.where(data.index.weekday < 5, 'Weekday',
        'Weekend')by_time = data.groupby([weekend,
        data.index.time]).mean()
```

Now we'll use some of the Matplotlib tools described in "Multiple Subplots" on page262 to plot two panels side by side (Figure 3-17):

```
In[46]: import matplotlib.pyplot as plt

        fig, ax = plt.subplots(1, 2, figsize=(14, 5))
        by_time.ix['Weekday'].plot(ax=ax[0], title='Weekdays',

                                   xticks=hourly_ticks, style=[':', '--', '-'])
        by_time.ix['Weekend'].plot(ax=ax[1], title='Weekends',
```

```
xticks=hourly_ticks, style=[':', '--', '-']);
```

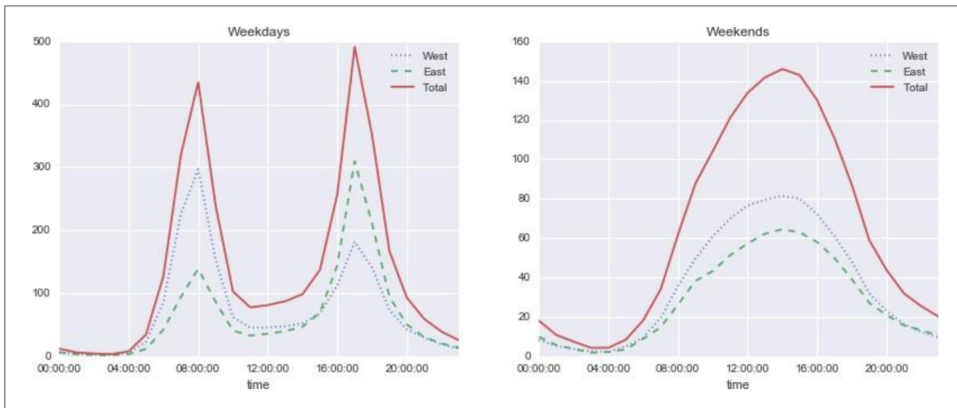


Figure 3-17. Average hourly bicycle counts by weekday and weekend

The result is very interesting: we see a bimodal commute pattern during the work week, and a unimodal recreational pattern during the weekends. It would be interesting to dig through this data in more detail, and examine the effect of weather, temperature, time of year, and other factors on people's commuting patterns.

High-Performance Pandas: eval() and query()

As we've already seen in previous chapters, the power of the PyData stack is built upon the ability of NumPy and Pandas to push basic operations into C via an intu- itive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effec tive for many common use cases, they often rely on the creation of temporary inter- mediate objects, which can cause undue overhead in computational time and memory use.

As of version 0.13 (released January 2014), Pandas includes some experimental toolsthat allow you to directly access C-speed operations without costly allocation of inter-mediate arrays. These are the eval() and query() functions, which rely on the Numexpr package. In this notebook we will walk through their use and give some rules of thumb about when you might think about using them.

Motivating query() and eval(): Compound Expressions

We've seen previously that NumPy and Pandas support fast vectorized operations; forexample, when you are adding the elements of two arrays:

```
In[1]: import numpy as np

    rng =
    np.random.RandomState(42)
    x = rng.rand(1E6)
```

```
y = rng.rand(1E6)
%timeit x + y
```

100 loops, best of 3: 3.39 ms per loop

As discussed in "Computation on NumPy Arrays: Universal Functions" on page 50, this is much faster than doing the addition via a Python loop or comprehension:

```
In[2]:
%timeit np.fromiter((xi + yi for  xi, yi in  zip(x, y)),
                    dtype=x.dtype, count=len(x))
```

1 loop, best of 3: 266 ms per loop

But this abstraction can become less efficient when you are computing compound expressions. For example, consider the following expression:

```
In[3]: mask = (x > 0.5) & (y < 0.5)
```

Because NumPy evaluates each subexpression, this is roughly equivalent to the following:

```
In[4]: tmp1 = (x > 0.5)
       tmp2 = (y < 0.5)
       mask = tmp1 &
       tmp2
```

In other words, *every intermediate step is explicitly allocated in memory*. If the x and y arrays are very large, this can lead to significant memory and computational over- head. The Numexpr library gives you the ability to compute this type of compound expression element by element, without the need to allocate full intermediate arrays. The Numexpr documentation (https://github.com/pydata/numexpr) has more details, but for the time being it is sufficientto say that the library accepts a *string* giving the NumPy-style expression you'd like tocompute:

```
In[5]:import  numexpr
       mask_numexpr = numexpr.evaluate('(x > 0.5) & (y <
       0.5)')np.allclose(mask, mask_numexpr)
```

Out[5]: True

The benefit here is that Numexpr evaluates the expression in a way that does not use

full-sized temporary arrays, and thus can be much more efficient than NumPy, espe- cially for large arrays. The Pandas eval() and query() tools that we will discuss here are conceptually similar, and depend on the Numexpr package.

pandas.eval() for Efficient Operations

The eval() function in Pandas uses string expressions to efficiently compute opera- tions using DataFrames. For example, consider the following DataFrames:

```
In[6]: import pandas as pd
       nrows, ncols = 100000, 100
       rng = np.random.RandomState(42)
       df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                             for i in range(4))
```

To compute the sum of all four DataFrames using the typical Pandas approach, we canjust write the sum:

```
In[7]: %timeit df1 + df2 + df3 + df4

10 loops, best of 3: 87.1 ms per loop
```

We can compute the same result via pd.eval by constructing the expression as astring:

```
In[8]: %timeit pd.eval('df1 + df2 + df3 + df4')

10 loops, best of 3: 42.2 ms per loop
```

The eval()version of this expression is about 50% faster (and uses much less mem-ory), while giving the same result:

```
In[9]: np.allclose(df1 + df2 + df3 + df4,
                   pd.eval('df1 + df2 + df3 + df4'))

Out[9]: True
```

Operations supported by pd.eval()

As of Pandas v0.16, pd.eval()supports a wide range of operations. To demonstratethese, we’ll use the following integer DataFrames:

```
In[10]: df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100, 3)))
                                   for i in range(5))
```

Arithmetic operators. pd.eval() supports all arithmetic operators. For example:

```
In[11]: result1 = -df1 * df2 / (df3 + df4) - df5

        result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
        np.allclose(result1, result2)

Out[11]: True
```

Comparison operators. pd.eval() supports all comparison operators, including chained expressions:

```
In[12]: result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
        result2 = pd.eval('df1 < df2 <= df3 != df4')
        np.allclose(result1, result2)

Out[12]: True
```

Bitwise operators. pd.eval() supports the & and | bitwise operators:

```
In[13]: result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)

        result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
        np.allclose(result1, result2)

Out[13]: True
```

In addition, it supports the use of the literal and and or in Boolean expressions:

```
In[14]: result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
        np.allclose(result1, result3)

Out[14]: True
```

Object attributes and indices. pd.eval() supports access to object attributes via the obj.attr syntax, and indexes via the obj[index] syntax:

```
In[15]: result1 = df2.T[0] + df3.iloc[1]

        result2 = pd.eval('df2.T[0] + df3.iloc[1]')
        np.allclose(result1, result2)
```

Out[15]: True

Other operations. Other operations, such as function calls, conditional statements, loops, and other more involved constructs, are currently *not* implemented in pd.eval(). If you'd like to execute these more complicated types of expressions, youcan use the Numexpr library itself.

DataFrame.eval() for Column-Wise Operations

Just as Pandas has a top-level pd.eval() function, DataFrames have an eval()method that works in similar ways. The benefit of the eval()method is that columnscan be referred to *by name*. We'll use this labeled array as an example:

```
In[16]: df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B',
        'C'])df.head()
```

Out[16]:	A	B	C
	0	0.40693	0.06993
	0.375506	9	8
	1	0.23561	0.15437
	0.069087	5	4
	2	0.43383	0.65232
	0.677945	9	4
	3	0.80805	0.34719
	0.264038	5	7
	4	0.25241	0.55778
	0.589161	8	9

Using pd.eval()as above, we can compute expressions with the three columns likethis:

```
In[17]: result1 = (df['A'] + df['B']) / (df['C'] - 1) result2
        = pd.eval("(df.A + df.B) / (df.C - 1)")
        np.allclose(result1, result2)
```

Out[17]: True

The DataFrame.eval()method allows much more succinct evaluation of expressionswith the columns:

```
In[18]: result3 = df.eval('(A + B) / (C - 1)')
        np.allclose(result1, result3)
```

Out[18]: True

Notice here that we treat *column names as variables* within the evaluated expression,and the result is what we would wish.

Assignment in DataFrame.eval()

In addition to the options just discussed, DataFrame.eval() also allows assignment to any column. Let's use the DataFramefrom before, which has columns 'A', 'B', and'C':

```
In[19]: df.head()
```

```
Out[19]:           A         B         C
         0  0.375506  0.406939  0.069938
         1  0.069087  0.235615  0.154374
         2  0.677945  0.433839  0.652324
         3  0.264038  0.808055  0.347197
         4  0.589161  0.252418  0.557789
```

We can use df.eval()to create a new column 'D'and assign to it a value computedfrom the other columns:

```
In[20]: df.eval('D = (A + B) / C', inplace=True)
        df.head()
```

```
Out[20]:           A         B         C          D
         0  0.375506  0.406939  0.069938  11.187620
         1  0.069087  0.235615  0.154374   1.973796
         2  0.677945  0.433839  0.652324   1.704344
         3  0.264038  0.808055  0.347197   3.087857
         4  0.589161  0.252418  0.557789   1.508776
```

In the same way, any existing column can be modified:

```
In[21]: df.eval('D = (A - B) / C', inplace=True)
        df.head()
```

Out[21]: A B
C D0 0.375506
0.406939 0.069938 -0.449425
1 0.2356 0.154374 -
0.069087 15 1.078728
2 0.4338 0.652324
0.677945 39 0.374209

3	0.8080	0.347197 -
0.264038	55	1.566886
4	0.2524	0.557789
0.589161	18	0.603708

Local variables in DataFrame.eval()

The DataFrame.eval() method supports an additional syntax that lets it work withlocal Python variables. Consider the following:

```
In[22]: column_mean = df.mean(1)
        result1 = df['A'] +
        column_mean

        result2 = df.eval('A +
        @column_mean')
        np.allclose(result1, result2)

Out[22]: True
```

The @ character here marks a *variable name* rather than a *column name*, and lets you efficiently evaluate expressions involving the two "namespaces": the namespace of columns, and the namespace of Python objects. Notice that this @ character is only supported by the DataFrame.eval() *method*, not by the pandas.eval() *function*, because the pandas.eval()function only has access to the one (Python) namespace.

DataFrame.query() Method

The DataFrame has another method based on evaluated strings, called the query()

method. Consider the following:

```
In[23]: result1 = df[(df.A < 0.5) & (df.B < 0.5)]

        result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
        np.allclose(result1, result2)

Out[23]: True
```

As with the example used in our discussion of DataFrame.eval(), this is an expres- sion involving columns of the DataFrame. It cannot be expressed using the Data Frame.eval() syntax, however! Instead, for this type of filtering operation, you can use the query() method:

```
In[24]: result2 = df.query('A < 0.5 and B < 0.5')
        np.allclose(result1, result2)
```

```
Out[24]: True
```

In addition to being a more efficient computation, compared to the masking expres- sion this is much easier to read and understand. Note that the query() method also accepts the @flag to mark local variables:

```
In[25]: Cmean = df['C'].mean()

        result1 = df[(df.A < Cmean) & (df.B <
        Cmean)] result2 = df.query('A < @Cmean
        and B < @Cmean')np.allclose(result1,
        result2)

Out[25]: True
```

Performance: When to Use These Functions

When considering whether to use these functions, there are two considerations: *com- putation time* and *memory use*. Memory use is the most predictable aspect. As already mentioned, every compound expression involving NumPy arrays or Pandas Data Frames will result in implicit creation of temporary arrays: For example, this:

```
In[26]: x = df[(df.A < 0.5) & (df.B < 0.5)]
```

is roughly equivalent to this:

```
In[27]: tmp1 = df.A < 0.5

        tmp2 = df.B < 0.5
        tmp3 = tmp1 &
        tmp2x =
        df[tmp3]
```

If the size of the temporary DataFrames is significant compared to your available sys-tem memory (typically several gigabytes), then it’s a good idea to use an eval() or query() expression. You can check the approximate size of your array in bytes usingthis:

```
In[28]:

df.values.nbytes

Out[28]: 32000
```

On the performance side, eval() can be faster even when you are not maxing out your system memory. The issue is how your temporary DataFrames compare to the size of the

L1 or L2 CPU cache on your system (typically a few megabytes in 2016); ifthey are much bigger, then eval()can avoid some potentially slow movement of val-ues between the different memory caches. In practice, I find that the difference in computation time between the traditional methods and the eval/query method is usually not significant—if anything, the traditional method is faster for smallerarrays! The benefit of eval/query is mainly in the saved memory, and the sometimescleaner syntax they offer.

We've covered most of the details of eval()and query()here; for more informationon these, you can refer to the Pandas documentation. In particular, different parsers and engines can be specified for running these queries; for details on this, see the dis-cussion within the "Enhancing Performance" section.

Lab Activity -DataFrame Data Structure

This lab activity must be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with the # sign are comments in Python and are used to elaborate the code.

```
==========================================================================
# The DataFrame data structure is the heart of the Panda's library. It is #a primary object you will work with in data analysis and #cleaning #tasks.
# The DataFrame is conceptually a two-dimensional series object, where # there is an index and multiple columns of content, with each column #having a label. The distinction between a column and a row is #only a conceptual distinction. Moreover, you can think of the #DataFrame as simply a two-axes labeled array.
# Lets start by importing our pandas library
import pandas as pd
## I'm going to jump in with an example. Lets create three school records for students and their
# class grades. I'll create each as a series which has a student name, #the class name, and the score.
record1 = pd.Series({'Name': 'Ali','Class': 'Physics', 'Score': 85})
record2 = pd.Series({'Name': 'Javed','Class': 'Chemistry','Score': 82})
record3 = pd.Series({'Name': 'Hafeez', 'Class': 'Biology','Score': 90})
# Like a Series, the DataFrame object is index. Here I'll use a group of series, where each series
# represents a row of data. Just like the Series function, we can pass in our individual items
# in an array, and we can pass in our index values as a second arguments
df = pd.DataFrame([record1, record2, record3],index=['school1', 'school2', 'school1'])
# And just like the Series we can use the head() function to see the first several rows of the
# dataframe, including indices from both axes, and we can use this to verify the columns and the rows
```

```
df.head()
#The results of the
# dataframe. So we have the index, which is the leftmost column and is #the school name, and
# then we have the rows of data, where each row has a column header which #was given in our initial
# record dictionaries
# An alternative method is that you could use a list of dictionaries, where each dictionary
# represents a row of data.
students = [{'Name': 'ali',
             'Class': 'Physics',
             'Score': 85},
            {'Name': 'Javed',
             'Class': 'Chemistry',
             'Score': 82},
            {'Name': 'Hafeez',
             'Class': 'Biology',
             'Score': 90}]
# Then we pass this list of dictionaries into the DataFrame function
df = pd.DataFrame(students, index=['school1', 'school2', 'school1'])
# And lets print the head again
df.head()
# Similar to the series, we can extract data using the .iloc and .loc #attributes. Because the
# DataFrame is two-dimensional, passing a single value to the loc #indexing operator will return
# the series if there's only one row to return.
# For instance, if we wanted to select data associated with school2, we #would just query the
# .loc attribute with one parameter.
df.loc['school2']
# You'll note that the name of the series is returned as the index value, #while the column name is included in the output.
#We can check the data type of the return using the python type function.
type(df.loc['school2'])
# It's important to remember that the indices and column names along #either axes horizontal or
# vertical, could be non-unique. In this example, we see two records for #school1 as different rows.
# If we use a single value with the DataFrame lock attribute, multiple #rows of the DataFrame will
# return, not as a new series, but as a new DataFrame.
# Lets query for school1 records
df.loc['school1']
# And we can see the the type of this is different too
type(df.loc['school1'])
# One of the powers of the Panda's DataFrame is that you can quickly #select data based on multiple axes.
```

```
# For instance, if you wanted to just list the student names for school1, #you would supply two parameters to .loc, one being the row index and the #other being the column name.
# For instance, if we are only interested in school1's student names
df.loc['school1', 'Name']
# Remember, just like the Series, the pandas developers have implemented #this using the indexing operator and not as parameters to a function.
# What would we do if we just wanted to select a single column though? #Well, there are a few
# mechanisms. Firstly, we could transpose the matrix. This pivots all of #the rows into columns
#and all of the columns into rows, and is done with the T attribute
df.T
# Then we can call .loc on the transpose to get the student names only
df.T.loc['Name']
# However, since iloc and loc are used for row selection, Panda reserves #the indexing operator
# directly on the DataFrame for column selection. In a Panda's DataFrame, #columns always have a name.
# So this selection is always label based, and is not as confusing as it #was when using the square
# bracket operator on the series objects. For those familiar with #relational databases, this operator
# is analogous to column projection.
df['Name']
# In practice, this works really well since you're often trying to add or #drop new columns. However,
# this also means that you get a key error if you try and use .loc with a #column name
df.loc['Name']
#Note too that the result of a single column projection is a Series object
type(df['Name'])

# Since the result of using the indexing operator is either a DataFrame #or Series, you can chain
# operations together. For instance, we can select all of the rows which #related to school1 using
# .loc, then project the name column from just those rows
df.loc['school1']['Name']
# If you get confused, use type to check the responses from resulting #operations
print(type(df.loc['school1'])) #should be a DataFrame
print(type(df.loc['school1']['Name'])) #should be a Series
# Chaining, by indexing on the return type of another index, can come #with some costs and is
# best avoided if you can use another approach. In particular, chaining #tends to cause Pandas
# to return a copy of the DataFrame instead of a view on the DataFrame.
# For selecting data, this is not a big deal, though it might be slower #than necessary.
# If you are changing data though this is an important distinction and #can be a source of error.
# Here's another approach. As we saw, .loc does row selection, and it can #take two parameters,
# the row index and the list of column names. The .loc attribute also #supports slicing.
# If we wanted to select all rows, we can use a colon to indicate a full #slice from beginning to end.
```

```
# This is just like slicing characters in a list in python. Then we can #add the column name as the
# second parameter as a string. If we wanted to include multiple columns, #we could do so in a list.
# and Pandas will bring back only the columns we have asked for.

# Here's an example, where we ask for all the names and scores for all #schools using the .loc operator.
df.loc[:,['Name', 'Score']]
# Take a look at that again. The colon means that we want to get all of #the rows, and the list in the second argument position is the list of #columns we want to get back
# That's selecting and projecting data from a DataFrame based on row and #column labels. The key
# concepts to remember are that the rows and columns are really just for #our benefit. Underneath
# this is just a two axes labeled array, and transposing the columns is #easy. Also, consider the
# issue of chaining carefully, and try to avoid it, as it can cause #unpredictable results, where
# your intent was to obtain a view of the data, but instead Pandas #returns to you a copy.
# Before we leave the discussion of accessing data in DataFrames, lets #talk about dropping data.
# It's easy to delete data in Series and DataFrames, and we can use the #drop function to do so.
# This function takes a single parameter, which is the index or row #label, to drop. This is another
# tricky place for new users -- the drop function doesn't change the #DataFrame by default! Instead,
#the drop function returns to you a copy of the DataFrame with the given #rows removed.
df.drop('school1')
# But if we look at our original DataFrame we see the data is still intact.
df
# Drop has two interesting optional parameters. The first is called #inplace, and if it's
# set to true, the DataFrame will be updated in place, instead of a copy #being returned.
# The second parameter is the axes, which should be dropped. By default, #this value is 0,
# indicating the row axis. But you could change it to 1 if you want to #drop a column.
# For example, lets make a copy of a DataFrame using .copy()
copy_df = df.copy()
# Now lets drop the name column in this copy
copy_df.drop("Name", inplace=True, axis=1)
copy_df
# There is a second way to drop a column, and that's directly through the #use of the indexing
# operator, using the del keyword. This way of dropping data, however, #takes immediate effect
# on the DataFrame and does not return a view.
del copy_df['Class']
copy_df
## Finally, adding a new column to the DataFrame is as easy as assigning #it to some value using
# the indexing operator. For instance, if we wanted to add a class #ranking column with default
# value of None, we could do so by using the assignment operator after #the square brackets.
# This broadcasts the default value to the new column immediately.
df['ClassRanking'] = None
```

df
 # In this LAB ACTIVTIY you've learned about the data structure you'll use #the most in pandas, the DataFrame. The dataframe is indexed both by row #and column, and you can easily select individual rows and project the #columns you're interested in using the familiar indexing methods from #the Series class.

Lab Activity -Merging DataFrames

This lab activity needs to be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with # sign are comments in Python and are used to elaborate the code.

==

```
# In this lab we're going to address how you can bring multiple dataframe #objects together, either by merging them horizontally, or by #concatenating them vertically. import pandas as pd

# First we create two DataFrames, staff and students.
staff_df = pd.DataFrame([{'Name': 'Kiran', 'Role': 'Director of HR'},
                 {'Name': 'Salma', 'Role': 'Course liasion'},
                 {'Name': 'Jameel', 'Role': 'Grader'}])
# And lets index these staff by name
staff_df = staff_df.set_index('Name')
# Now we'll create a student dataframe
student_df = pd.DataFrame([{'Name': 'Jameel', 'School': 'Business'},
                  {'Name': 'Mushahid', 'School': 'Law'},
                  {'Name': 'Salma', 'School': 'Engineering'}])
# And we'll index this by name too
student_df = student_df.set_index('Name')

# And lets just print out the dataframes
print(staff_df.head())
print(student_df.head())
# There's some overlap in these DataFrames in that Jameel and Salma are #both students and staff, but Mushahid and
# Kiran are not. Importantly, both DataFrames are indexed along the value
pd.merge(staff_df, student_df, how='outer', left_index=True, right_index=True)
pd.merge(staff_df, student_df, how='inner', left_index=True, right_index=True)
pd.merge(staff_df, student_df, how='left', left_index=True, right_index=True)
pd.merge(staff_df, student_df, how='right', left_index=True, right_index=True)
staff_df = staff_df.reset_index()
```

```
student_df = student_df.reset_index()
# Now lets merge using the on parameter
pd.merge(staff_df, student_df, how='right', on='Name')
# So what happens when we have conflicts between the DataFrames? Let's #take a look by creating new staff and
# student DataFrames that have a location information added to them.
staff_df = pd.DataFrame([{'Name': 'Kiran', 'Role': 'Director of HR',
                          'Location': 'Sukkur'},
                         {'Name': 'Salma', 'Role': 'Course liasion',
                          'Location': 'Karachi'},
                         {'Name': 'Jameel', 'Role': 'Grader',
                          'Location': 'Hyderabad'}])
student_df = pd.DataFrame([{'Name': 'Jameel', 'School': 'Business',
                            'Location': 'Lateefabad 7'},
                           {'Name': 'Mushahid', 'School': 'Law',
                            'Location': 'Nawab Shah'},
                           {'Name': 'Salma', 'School': 'Engineering',
                            'Location': 'Korangi 2'}])

pd.merge(staff_df, student_df, how='left', on='Name')

# Here's an example with some new student and staff data
staff_df = pd.DataFrame([{'First Name': 'Kiran', 'Last Name': 'Khan',
                          'Role': 'Director of HR'},
                         {'First Name': 'Salma', 'Last Name': 'Mughal',
                          'Role': 'Course liasion'},
                         {'First Name': 'Jameel', 'Last Name': 'Malik',
                          'Role': 'Grader'}])
student_df = pd.DataFrame([{'First Name': 'Jameel', 'Last Name': 'Mughal',
                            'School': 'Business'},
                           {'First Name': 'Mushahid', 'Last Name': 'Uqaili',
                            'School': 'Law'},
                           {'First Name': 'Salma', 'Last Name': 'Mughal',
                            'School': 'Engineering'}])
# As you see here, Jameel malik and Jameel Mughal don't match on both keys since they have different last names. So we would expect that an #inner join doesn't include these individuals in the output, and only #Salma  Mughal will be retained.
pd.merge(staff_df, student_df, how='inner', on=['First Name','Last Name'])

get_ipython().run_cell_magic('capture', '', 'df_2011 =
pd.read_csv("datasets/college_scorecard/MERGED2011_12_PP.csv",
```

```
error_bad_lines=False)\ndf_2012 =
pd.read_csv("datasets/college_scorecard/MERGED2012_13_PP.csv",
error_bad_lines=False)\ndf_2013 =
pd.read_csv("datasets/college_scorecard/MERGED2013_14_PP.csv", error_bad_lines=False)\n')
df_2011.head(3)
print(len(df_2011))
print(len(df_2012))
print(len(df_2013))
#Let's see what it looks like
frames = [df_2011, df_2012, df_2013]
pd.concat(frames)
# As you can see, we have more observations in one dataframe and columns remain the same. If we
scroll down to
# the bottom of the output, we see that there are a total of 30,832 rows after concatenating three
dataframes.
# Let's add the number of rows of the three dataframes and see if the two numbers match
len(df_2011)+len(df_2012)+len(df_2013)
# Now let's try it out
pd.concat(frames, keys=['2011','2012','2013'])

# Now you know how to merge and concatenate datasets together. You will #find such functions
very useful for combining data to get more complex #or complicated results and to do analysis
with. A solid understanding of #how to merge data is absolutely essentially when you are procuring,
#cleaning, and manipulating data. It's worth knowing how to join #different datasets quickly, and
the different options you can use when #joining datasets, and I would encourage you to check out
the pandas docs #for joining and concatenating data.
```

Lab activity - DataFrame` Indexing and Loading

This lab activity needs to be performed using Jupyter Notebook, PyCharm, or any other IDLE . The lines starting with # sign are comments in Python and are used to elaborate the code.

==

```
#lets look at the content of a CSV file
get_ipython().system('more Admission_Predict.csv')
import pandas as pd
df = pd.read_csv('Admission_Predict.csv')
df.head()
df = pd.read_csv('datasets/Admission_Predict.csv', index_col=0)
df.head()
new_df=df.rename(columns={'GRE Score':'GRE Score', 'TOEFL Score':'TOEFL Score',
```

```
            'University Rating':'University Rating',
            'SOP': 'Statement of Purpose','LOR': 'Letter of Recommendation',
            'CGPA':'CGPA', 'Research':'Research',
            'Chance of Admit':'Chance of Admit'})
new_df.head()
new_df.columns
# way would be to change a column by including the space in the name
new_df=new_df.rename(columns={'LOR ': 'Letter of Recommendation'})
new_df.head()
# What if that was a tab instead of a space? Or two spaces?
# Another way is to create some function that does the cleaning and then #tell renamed to apply
that function
# across all of the data. Python comes with a handy string function to strip white space called
"strip()".
# When we pass this in to rename we pass the function as the mapper #parameter, and then
indicate whether the
# axis should be columns or index (row labels)
new_df=new_df.rename(mapper=str.strip, axis='columns')
new_df.head()
df.columns
cols = list(df.columns)
# Then a little list comprehenshion
cols = [x.lower().strip() for x in cols]
# Then we just overwrite what is already in the .columns attribute
df.columns=cols
# And take a look at our results
df.head()
```

Day-03: Pivot Tables

We have seen how the GroupByabstraction lets us explore relationships within a data-set. A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data. The pivot table takes simple column- wise data as input, and groups the entries into a two-dimensional table that providesa multidimensional summarization of the data. The difference between pivot tables and GroupBy can sometimes cause confusion; it helps me to think of pivot tables as essentially a *multidimensional* version of GroupBy aggregation. That is, you split- apply-combine, but both the split and the combine happen across not a one- dimensional index, but across a two-dimensional grid.

Motivating Pivot Tables

For the examples in this section, we'll use the database of passengers on the *Titanic*, available through the Seaborn library (see "Visualization with Seaborn"):

```
In[1]: import numpy as np
       import pandas as pd
       import seaborn as
       sns

       titanic = sns.load_dataset('titanic')

In[2]: titanic.head()

Out[2]:

    survived  pclass     sex   age  sibsp  parch     fare embarked  class
\\0        0       3    male  22.0      1      0   7.2500        S Third

1          1       1  female  38.0      1      0  71.2833        C  First

2          1       3  female  26.0      0      0   7.9250        S  Third

3          1       1  female  35.0      1      0  53.1000        S  First

4          0       3    male  35.0      0      0   8.0500        S  Third


       who adult_male deck  embark_town alive  alone

0      man       True  NaN  Southampton    no  False
1    woman      False    C    Cherbourg yes  False
2    woman      False  NaN  Southampton     yes      True
3    woman      False    C  Southampton     yes  False
4      man       True  NaN  Southampton     no      True
```

This contains a wealth of information on each passenger of that ill-fated voyage, including gender, age, class, fare paid, and much more.

Pivot Tables by Hand

To start learning more about this data, we might begin by grouping it according to gender, survival status, or some combination thereof. If you have read the previous section, you might be tempted to apply a GroupBy operation—for example, let's look at survival rate by gender:

```
In[3]: titanic.groupby('sex')[['survived']].mean()
```

Out[3]:

dsex	survive
female	0.742038
male	0.188908

This immediately gives us some insight: overall, three of every four females on boardsurvived, while only one in five males survived!

This is useful, but we might like to go one step deeper and look at survival by both sexand, say, class. Using the vocabulary of GroupBy, we might proceed using something like this: we *group by* class and gender, *select* survival, *apply* a mean aggregate, *com-bine* the resulting groups, and then *unstack* the hierarchical index to reveal the hidden multidimensionality. In code:

```
In[4]: titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

Out[4]:

class sex	First	Second	Third
femal e	0.96808 5	0.92105 3	0.50000 0
male	0.36885 2	0.15740 7	0.13544 7

This gives us a better idea of how both gender and class affected survival, but the code is starting to look a bit garbled. While each step of this pipeline makes sense in light of the tools we've previously discussed, the long string of code is not particularlyeasy to read or use. This two-dimensional GroupBy is common enough that Pandas includes a convenience routine, pivot_table, which succinctly handles this type of multidimensional aggregation.

Pivot Table Syntax

Here is the equivalent to the preceding operation using the pivot_table method of DataFrames:

```
In[5]: titanic.pivot_table('survived', index='sex', columns='class')
```

```
Out[5]: class   First     Second    Third
        sex
        femal   0.96808   0.92105   0.50000
        e       5         3         0
        male    0.36885   0.15740   0.13544
                2         7         7
```

This is eminently more readable than the GroupBy approach, and produces the same result. As you might expect of an early 20th-century transatlantic cruise, the survival gradient favors both women and higher classes. First-class women survived with nearcertainty (hi, Rose!), while only one in ten third-class men survived (sorry, Jack!).

Multilevel pivot tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple lev-els, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the pd.cutfunction:

```
In[6]: age = pd.cut(titanic['age'], [0, 18, 80])
       titanic.pivot_table('survived', ['sex', age], 'class')
```

```
Out[6]:  class             First     Second    Third
         sex     age
         femal   (0, 18]   0.90909   1.00000   0.51162
         e                 1         0         8
                 (18, 80]  0.97297   0.90000   0.42372
                           3         0         9
         male    (0, 18]   0.80000   0.60000   0.21568
                           0         0         6
                 (18, 80]  0.37500   0.07142   0.13366
                           0         9         3
```

We can apply this same strategy when working with the columns as well; let's add infoon the fare paid using pd.qcutto automatically compute quantiles:

```
In[7]: fare = pd.qcut(titanic['fare'], 2)
       titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

```
Out[7]
:
fare             [0, 14.454]
class            First       Second    Third          \\
sex     age
femal   (0, 18]  NaN         1.00000   0.71428
e                            0         6
        (18, 80] NaN         0.88000   0.44444
                             0         4
male    (0, 18]  NaN         0.00000   0.26087
```

	(18, 80]	0.0	0.098039	0.125000

fare		(14.454, 512.329]		
class		First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.318182
	(18, 80]	0.972973	0.914286	0.391304
male	(0, 18]	0.800000	0.818182	0.178571
	(18, 80]	0.391304	0.030303	0.192308

The result is a four-dimensional aggregation with hierarchical indices, shown in a grid demonstrating the relationship between the values.

Additional pivot table options

The full call signature of the pivot_table method of DataFrames is as follows:

```
# call signature as of Pandas 0.18
DataFrame.pivot_table(data, values=None, index=None, columns=None,

                      aggfunc='mean', fill_value=None,
                      margins=False,dropna=True,
                      margins_name='All')
```

We've already seen examples of the first three arguments; here we'll take a quick lookat the remaining ones. Two of the options, fill_value and dropna, have to do with missing data and are fairly straightforward; we will not show examples of them here.
The aggfunc keyword controls what type of aggregation is applied, which is a mean by default. As in the GroupBy, the aggregation specification can be a string represent-ing one of several common choices ('sum', 'mean', 'count', 'min', 'max', etc.) or afunction that implements an aggregation (np.sum(), min(), sum(), etc.). Additionally,it can be specified as a dictionary mapping a column to any of the above desired options:

```
In[8]: titanic.pivot_table(index='sex', columns='class',

                           aggfunc={'survived':sum, 'fare':'mean'})
```

Out[8]: fare survived

class First Second Third First Second Thirdsex

female	106.125798	21.970121	16.118810	91.0	70.0	72.0
male	67.226127	19.741782	12.661633	45.0	17.0	47.0

Notice also here that we've omitted the values keyword; when you're specifying a mapping for aggfunc, this is determined automatically.

At times it's useful to compute totals along each grouping. This can be done via the margins keyword:

```
In[9]: titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

Out[9]: class sex	First	Second	Third	All
female	0.968085	0.921053	0.500000	0.742038
male	0.368852	0.157407	0.135447	0.188908
All	0.629630	0.472826	0.242363	0.383838

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 38%.The margin label can be specified with the margins_name keyword, which defaults to "All".

Example: Birthrate Data

As a more interesting example, let's take a look at the freely available data on births inthe United States, provided by the Centers for Disease Control (CDC). This data can be found at *https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv* (this dataset has been analyzed rather extensively by Andrew Gelman and his group; see, for example, this blog post):

```
In[10]:

# shell command to download the data:

# !curl -O
https://raw.githubusercontent.com/jakevdp/data-
CDCbirths/# master/births.csv

In[11]: births = pd.read_csv('births.csv')
```

Taking a look at the data, we see that it's relatively simple—it contains the number ofbirths grouped by date and gender:

```
In[12]: births.head()

Out[12]:   year   month day gender   births
```

```
0  1969  1    1    F  4046
1  1969  1    1    M  4440
2  1969  1    2    F  4454
3  1969  1    2    M  4548
4  1969  1    3    F  4548
```

We can start to understand this data a bit more by using a pivot table. Let's add a dec-ade column, and take a look at male and female births as a function of decade:

```
In[13]:
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')
```

Out[13]: gender decade	F	M
1960	1753634	1846572
1970	16263075	17121550
1980	18310351	19243452
1990	19479454	20420553
2000	18229309	19106428

We immediately see that male births outnumber female births in every decade. To see this trend a bit more clearly, we can use the built-in plotting tools in Pandas to visual-ize the total number of births by year (Figure 3-2; see Chapter 4 for a discussion of plotting with Matplotlib):

```
In[14]:
%matplotlib inline
import matplotlib.pyplot as plt
sns.set()  # use Seaborn styles
births.pivot_table('births', index='year', columns='gender',
aggfunc='sum').plot()plt.ylabel('total births per year');
```

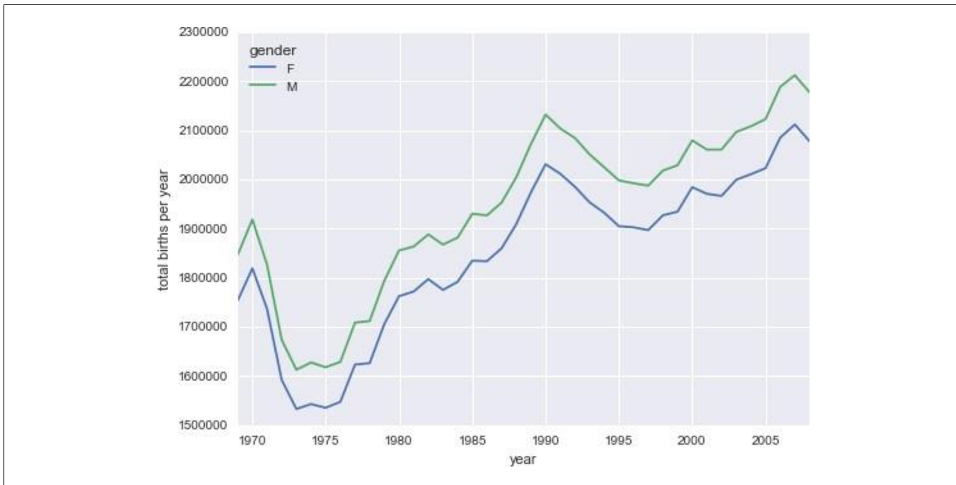


Figure . Total number of US births by year and gender

With a simple pivot table and plot() method, we can immediately see the annual trend in births by gender. By eye, it appears that over the past 50 years male births have outnumbered female births by around 5%.

Further data exploration

Though this doesn't necessarily relate to the pivot table, there are a few more interest-ing features we can pull out of this dataset using the Pandas tools covered up to this point. We must start by cleaning the data a bit, removing outliers caused by mistypeddates (e.g., June 31st) or missing values (e.g., June 99th). One easy way to remove these all at once is to cut outliers; we'll do this via a robust sigma-clipping operation:[1]

```
In[15]: quartiles = np.percentile(births['births'], [25, 50, 75])
        mu = quartiles[1]
        sig = 0.74 * (quartiles[2] - quartiles[0])
```

This final line is a robust estimate of the sample mean, where the 0.74 comes from the interquartile range of a Gaussian distribution. With this we can use the query() method to filter out rows with births outside these values:

```
In[16]:
births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')
```

Next we set the day column to integers; previously it had been a string because some columns in the dataset contained the value 'null':

```
In[17]: # set 'day' column to integer; it originally was a string due to nulls
```

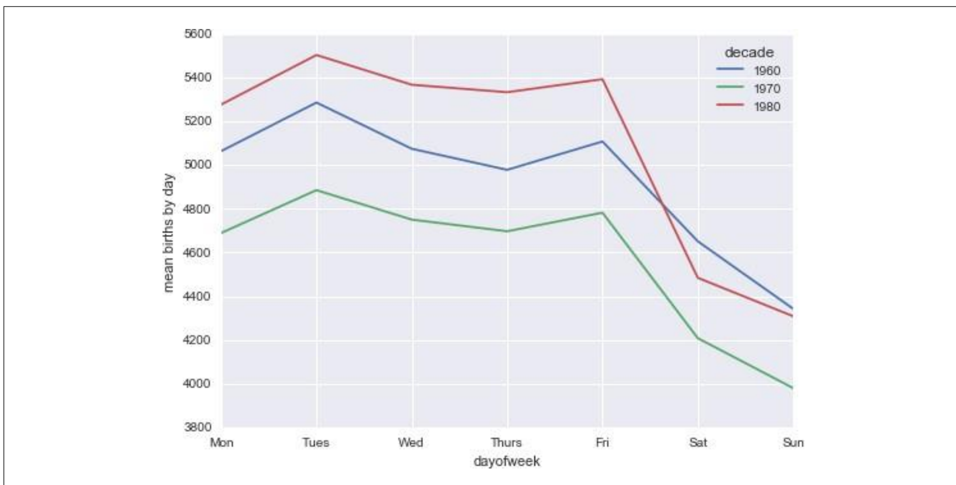
```
births['day'] = births['day'].astype(int)
```

Finally, we can combine the day, month, and year to create a Date index (see “Work-ing with Time Series” on page 188). This allows us to quickly compute the weekday corresponding to each row:

```
In[18]: # create a datetime index from the year, month, day
        births.index = pd.to_datetime(10000 * births.year +
                                      100 * births.month +
                                      births.day,
                                      format='%Y%m%d')

        births['dayofweek'] = births.index.dayofweek
```

Using this we can plot births by weekday for several decades (Figure 3-3):

```
In[19]:

import matplotlib.pyplot as
pltimport matplotlib as mpl

births.pivot_table('births', index='dayofweek',
                   columns='decade', aggfunc='mean').plot()
plt.gca().set_xticklabels(['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'])
```




```
plt.ylabel('mean births by day');
```

Figure . Average daily births by day of week and decade

Apparently births are slightly less common on weekends than on weekdays! Note thatthe 1990s and 2000s are missing because the CDC data contains only the month of birth starting in 1989.

Another interesting view is to plot the mean number of births by the day of the *year*. Let's first group the data by month and day separately:

```
In[20]:
births_by_date = births.pivot_table('births',
                                    [births.index.month, births.index.day])
births_by_date.head()
```

```
Out[20]: 1      4009.225
1
2              4247.400
3              4500.900
4              4571.350
5              4603.625
         Name: births, dtype: float64
```

The result is a multi-index over months and days. To make this easily plottable, let's turn these months and days into a date by associating them with a dummy year vari- able (making sure to choose a leap year so February 29th is correctly handled!)

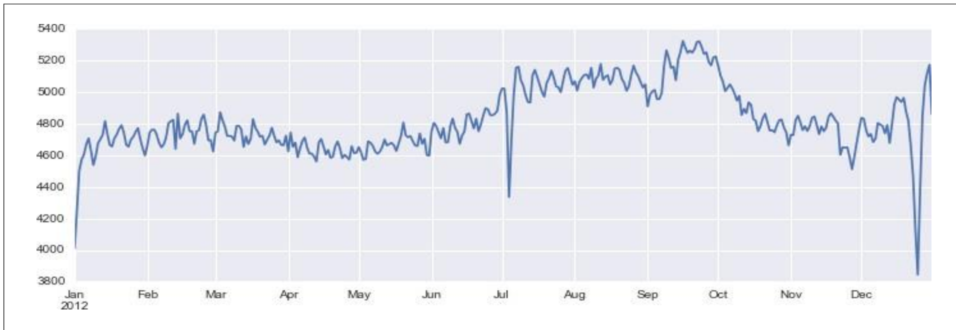
```
In[21]: births_by_date.index = [pd.datetime(2012, month, day)
                                for (month, day) in  births_by_date.index]
        births_by_date.head()
```

```
Out[21]: 2012-01-    4009.225
01
2012-01-02          4247.400
2012-01-03          4500.900
2012-01-04          4571.350
2012-01-05          4603.625
         Name: births, dtype: float64
```

Focusing on the month and day only, we now have a time series reflecting the average number of births by date of the year. From this, we can use the plot method to plot the data. It reveals some interesting trends:

```
In[22]: # Plot the results
```

```
fig, ax = plt.subplots(figsize=(12, 4))
```




```
births_by_date.plot(ax=ax);
```

Figure. Average daily births by date

Lab Activity

This lab activity need to performed using Jupyter Notebook, PyCharm, or any other IDLE

```
# Pivot Tables
# A pivot table is a way of summarizing data in a DataFrame for a #particular purpose. It makes heavy use of the aggregation function. A #pivot table is itself a DataFrame, where the rows represent one variable #that you're interested in, the columns another, and the cell's some #aggregate value. A pivot table also tends to includes marginal values as #well, which are the sums for each column and row. This allows you to be #able to see the relationship between two variables at just a glance.
import pandas as pd
import numpy as np
#Here we have the Times Higher Education World University Ranking dataset, #which is one of the most
#influential university measures. Let's import the dataset and see what it #looks like
df = pd.read_csv('cwurData.csv')
df.head()
# Here we can see each institution's rank, country, quality of education, #other metrics, and overall score.
# Let's say we want to create a new column called Rank_Level, where #institutions with world ranking 1-100 are
# categorized as first tier and those with world ranking 101 - 200 are #second tier, ranking 201 - 300 are # third tier, after 301 is other top #universities.
# Now, you actually already have enough knowledge to do this, so why don't #you pause the video and give it try?
```

```
# Create a function called create_category which will operate on the first
# column in the dataframe, world_rank
def create_category(ranking):
   if (ranking >= 1) & (ranking <= 100):
      return "First Tier Top Unversity"
   elif (ranking >= 101) & (ranking <= 200):
      return "Second Tier Top Unversity"
   elif (ranking >= 201) & (ranking <= 300):
      return "Third Tier Top Unversity"
   return "Other Top Unversity"
# Now we can apply this to a single column of data to create a new series
df['Rank_Level'] = df['world_rank'].apply(lambda x: create_category(x))
# And lets look at the result
df.head()
# A pivot table allows us to pivot out one of these columns a new column #headers and compare it against
# another column as row indices. Let's say we want to compare rank level #versus country of the universities
# and we want to compare in terms of overall score
# To do this, we tell Pandas we want the values to be Score, and index to #be the country and the columns to be
# the rank levels. Then we specify that the aggregation function, and here #we'll use the NumPy mean to get the
# average rating for universities in that country
df.pivot_table(values='score', index='country', columns='Rank_Level', aggfunc=[np.mean]).head()
# We can see a  hierarchical dataframe where the index, or rows, are by #country and the columns have two levels, the top level indicating that the #mean value is being used and the second level being our ranks. In this #example we only have one variable, the mean, that we are looking at, so we #don't really need a heirarchical index.
# We notice that there are some NaN values, for example, the first row, Argentia. The NaN values indicate that
# Argentia has only observations in the "Other Top Unversities" category
# Now, pivot tables aren't limited to one function that you might want to #apply. You can pass a named
# parameter, aggfunc, which is a list of the different functions to apply, #and pandas will provide you with
# the result using hierarchical column names.  Let's try that same query, #but pass in the max() function too
df.pivot_table(values='score', index='country', columns='Rank_Level', aggfunc=[np.mean, np.max]).head()
# So now we see we have both the mean and the max. As mentioned earlier, we #can also
```

```
summarize the values
# within a given top level column. For instance, if we want to see an #overall average for the country for the
# mean and we want to see the max of the max, we can indicate that we want #pandas to provide marginal values
df.pivot_table(values='score', index='country', columns='Rank_Level', aggfunc=[np.mean, np.max],
            margins=True).head()
# A pivot table is just a multi-level dataframe, and we can access series #or cells in the dataframe in a similar way
# as we do so for a regular dataframe.
# Let's create a new dataframe from our previous example
new_df=df.pivot_table(values='score', index='country',columns='Rank_Level', aggfunc=[np.mean, np.max], margins=True)
# Now let's look at the index
print(new_df.index)
# And let's look at the columns
print(new_df.columns)
# We can see the columns are hierarchical. The top level column indices #have two categories: mean and max, and
# the lower level column indices have four categories, which are the four #rank levels. How would we query this
# if we want to get the average scores of First Tier Top Unversity levels #in each country? We would just need
# to make two dataframe projections, the first for the mean, then the #second for the top tier
new_df['mean']['First Tier Top Unversity'].head()
# We can see that the output is a series object which we can confirm by #printing the type. Remember that when
# you project a single column of values out of a DataFrame you get series.
type(new_df['mean']['First Tier Top Unversity'])
# What if we want to find the country that has the maximum average score on #First Tier Top University level?
# We can use the idxmax() function.
new_df['mean']['First Tier Top Unversity'].idxmax()
# Now, the idxmax() function isn't special for pivot tables, it's a built in function to the Series object.
# We don't have time to go over all pandas functions and attributes, and I #want to encourage you to explore
# the API to learn more deeply what is available to you.
# If you want to achieve a different shape of your pivot table, you can do #so with the stack and unstack
# functions. Stacking is pivoting the lowermost column index to become the #innermost row index. Unstacking is
```

```
# the inverse of stacking, pivoting the innermost row index to become the #lowermost column index. An example
# will help make this clear
# Let's look at our pivot table first to refresh what it looks like
new_df.head()
# Now let's try stacking, this should move the lowermost column, so the #tiers of the university rankings, to
# the inner most row
new_df=new_df.stack()
new_df.head()
# In the original pivot table, rank levels are the lowermost column, after stacking, rank levels become the
# innermost index, appearing to the right after country
# Now let's try unstacking
new_df.unstack().head()
# That seems to restore our dataframe to its original shape. What do you #think would happen if we unstacked twice in a row?
new_df.unstack().unstack().head()
# We actually end up unstacking all the way to just a single column, so a #series object is returned. This column is just a "value", the meaning of #which is denoted by the #heirarachical index of operation, rank, and #country.
# So that's pivot tables. This has been a pretty short description, but #they're incredibly useful when dealing with numeric data, especially if #you're trying to summarize the data in some form. You'll regularly be #creating new pivot tables on slices of data, whether you're exploring the #data yourself or preparing data for others to report on. And of course, #you can pass any function you want to the aggregate function, including those that you define yourself.
```

Day-04: What Is Machine Learning?

Before we take a look at the details of various machine learning methods, let’s start bylooking at what machine learning is, and what it isn’t. Machine learning is often cate-gorized as a subfield of artificial intelligence, but I find that categorization can often be misleading at first brush. The study of machine learning certainly arose from research in this context, but in the data science application of machine learning meth-ods, it’s more helpful to think of machine learning as a means of *building models of data*.

Fundamentally, machine learning involves building mathematical models to help understand data. “Learning” enters the fray when we give these models *tunable parameters* that can be adapted to observed data; in this way the program can be con-sidered to be “learning” from the data. Once these models have been fit to previouslyseen data, they can be used to predict and understand aspects of newly observed data.I’ll leave to the reader the more philosophical digression regarding the extent to which this type of mathematical, model-based “learning” is

similar to the “learning” exhibited by the human brain.

Understanding the problem setting in machine learning is essential to using these tools effectively, and so we will start with some broad categorizations of the types of approaches we’ll discuss here.

Categories of Machine Learning

At the most fundamental level, machine learning can be categorized into two main types: supervised learning and unsupervised learning.

Supervised learning involves somehow modeling the relationship between measured features of data and some label associated with the data; once this model is deter- mined, it can be used to apply labels to new, unknown data. This is further subdivi- ded into *classification* tasks and *regression* tasks: in classification, the labels are discrete categories, while in regression, the labels are continuous quantities. We will see examples of both types of supervised learning in the following section.

Unsupervised learning involves modeling the features of a dataset without reference to any label, and is often described as “letting the dataset speak for itself.” These models include tasks such as *clustering* and *dimensionality reduction.* Clustering algorithms

identify distinct groups of data, while dimensionality reduction algorithms search for more succinct representations of the data. We will see examples of both types of unsupervised learning in the following section.

In addition, there are so-called *semi-supervised learning* methods, which fall some- where between supervised learning and unsupervised learning. Semi-supervised learning methods are often useful when only incomplete labels are available.

Qualitative Examples of Machine Learning Applications

To make these ideas more concrete, let’s take a look at a few very simple examples of a machine learning task. These examples are meant to give an intuitive, non- quantitative overview of the types of machine learning tasks we will be looking at in this chapter. In later sections, we will go into more depth regarding the particular models and how they are used. For a preview of these more technical aspects, you canfind the Python source that generates the figures in the online appendix.

Classification: Predicting discrete labels

We will first take a look at a simple *classification* task, in which you are given a set oflabeled points and want to use these to classify some unlabeled points.

Imagine that we have the data shown in Figure 5-1 (the code used to generate this figure, and all figures in this section, is available in the online appendix).

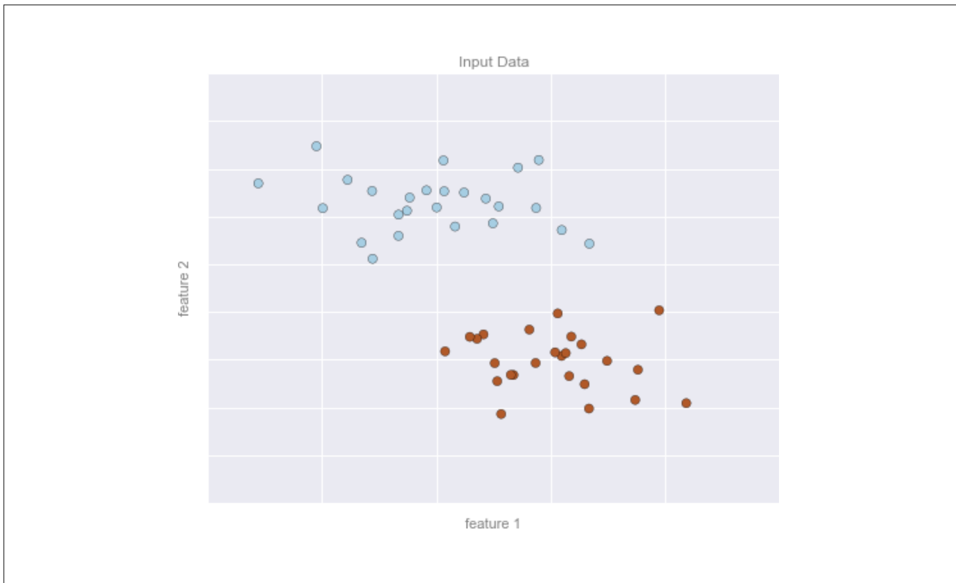


Figure 5-1. A simple data set for ***classification***

Here we have two-dimensional data; that is, we have two *features* for each point, rep-resented by the *(x,y)* positions of the points on the plane. In addition, we have one of two *class labels* for each point, here represented by the colors of the points. From these features and labels, we would like to create a model that will let us decide whether a new point should be labeled "blue" or "red."

There are a number of possible models for such a classification task, but here we willuse an extremely simple one. We will make the assumption that the two groups canbe separated by drawing a straight line through the plane between them, such that points on each side of the line fall in the same group. Here the *model* is a quantitativeversion of the statement "a straight line separates the classes," while the *model param-eters* are the particular numbers describing the location and orientation of that linefor our data. The optimal values for these model parameters are learned from thedata (this is the "learning" in machine learning), which is often called *training the model*.

Figure 5-2 is a visual representation of what the trained model looks like for this data.

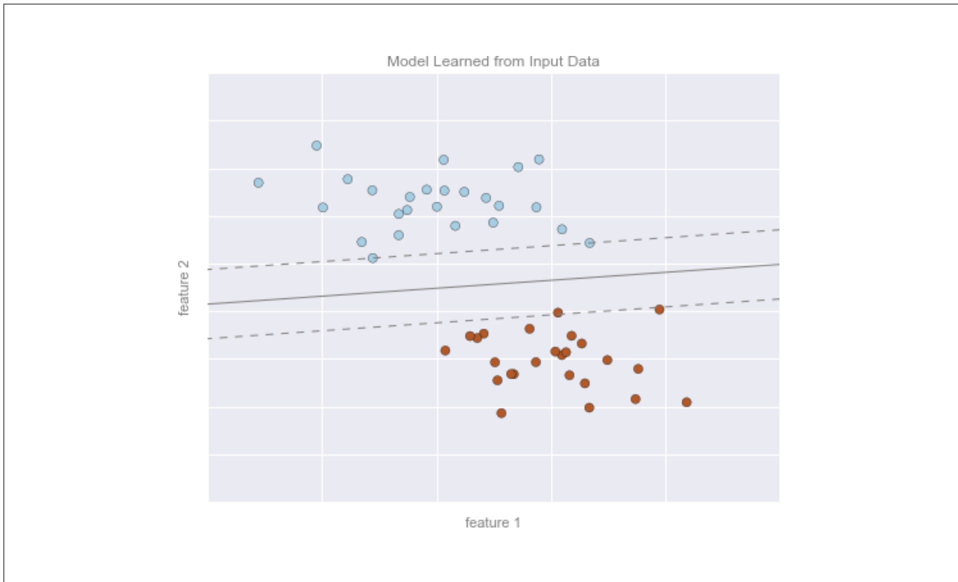


*Figure 5-2. A simple **classification** model*

Now that this model has been trained, it can be generalized to new, unlabeled data. Inother words, we can take a new set of data, draw this model line through it, andassign labels to the new points based on this model. This stage is usually called *predic-tion*. See Figure 5-3.

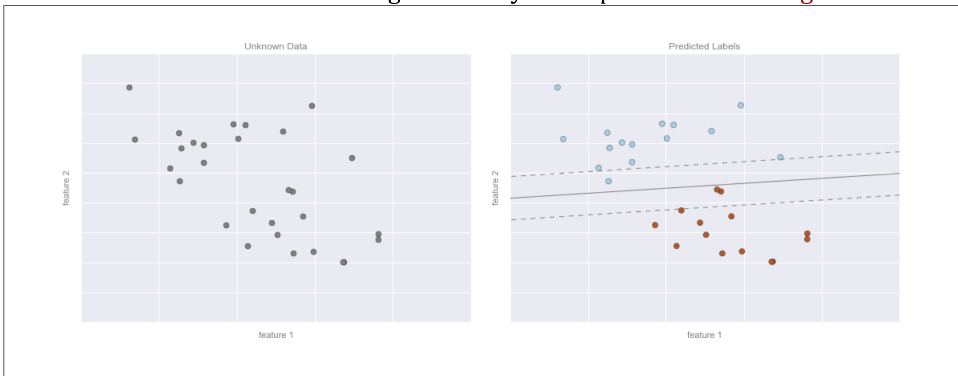


*Figure 5-3. Applying a **classification** model to new data*

This is the basic idea of a classification task in machine learning, where "classifica- tion" indicates that the data has discrete class labels. At first glance this may look fairly trivial: it would be relatively easy to simply look at this data and draw such a discriminatory line to accomplish this classification. A benefit of the machine learn- ing approach, however, is that it can generalize to much larger datasets in many moredimensions.

For example, this is similar to the task of automated spam detection for email; in this case, we might use the following features and labels:

- *feature 1*, *feature 2*, etc. → normalized counts of important words or phrases("Viagra," "Nigerian prince," etc.)
- *label* → "spam" or "not spam"

For the training set, these labels might be determined by individual inspection of a small representative sample of emails; for the remaining emails, the label would be determined using the model. For a suitably trained classification algorithm withenough well-constructed features (typically thousands or millions of words or phrases), this type of approach can be very effective. We will see an example of suchtext-based classification in "In Depth: Naive Bayes Classification" on page 382.

Some important classification algorithms that we will discuss in more detail are Gaus-sian naive Bayes (see "In Depth: Naive Bayes Classification" on page 382), support vector machines (see "In-Depth: Support Vector Machines" on page 405), and ran- dom forest classification (see "In-Depth: Decision Trees and Random Forests" on page 421).

Regression: Predicting continuous labels

In contrast with the discrete labels of a classification algorithm, we will next look at asimple *regression* task in which the labels are continuous quantities.

Consider the data shown in Figure 5-4, which consists of a set of points, each with a continuous label.

Figure 5-4. A simple dataset for regression

As with the classification example, we have two-dimensional data; that is, there are two features describing each data point. The color of each point represents the con-tinuous label for that point.

There are a number of possible regression models we might use for this type of data, but here we will use a simple linear regression to predict the points. This simple linearregression model assumes that if we treat the label as a third spatial dimension, we can fit a plane to the data. This is a higher-level generalization of the well-known problem of fitting a line to data with two coordinates.

We can visualize this setup as shown in Figure 5-5.

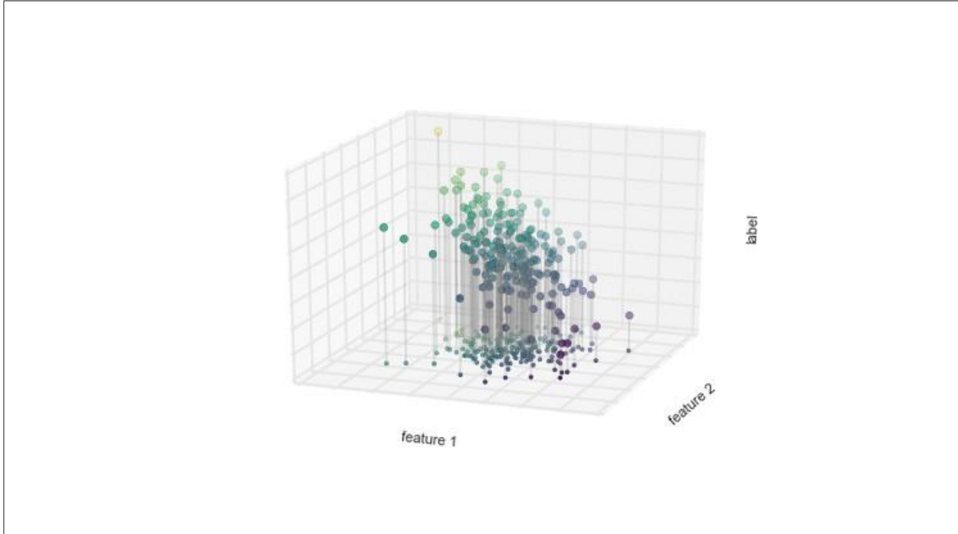


Figure 5-5. A three-dimensional view of the regression data

Notice that the *feature 1–feature 2* plane here is the same as in the two-dimensional plot from before; in this case, however, we have represented the labels by both color and three-dimensional axis position. From this view, it seems reasonable that fitting aplane through this three-dimensional data would allow us to predict the expectedlabel for any set of input parameters. Returning to the two-dimensional projection, when we fit such a plane we get the result shown in Figure 5-6.

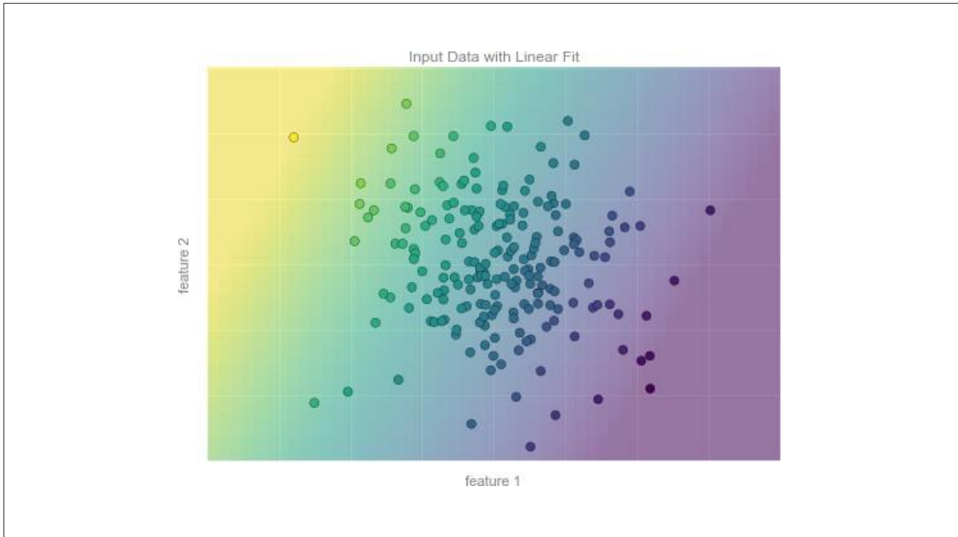


Figure 5-6. A representation of the regression model

This plane of fit gives us what we need to predict labels for new points. Visually, wefind the results shown in Figure 5-7.

Figure 5-7. Applying the regression model to new data

As with the classification example, this may seem rather trivial in a low number of dimensions. But the power of these methods is that they can be straightforwardly applied and evaluated in the case of data with many, many features.

For example, this is similar to the task of computing the distance to galaxies observedthrough a telescope—in this case, we might use the following features and labels:

- *feature 1*, *feature 2*, etc. → brightness of each galaxy at one of several wavelengthsor colors
- *label* → distance or redshift of the galaxy

The distances for a small number of these galaxies might be determined through an independent set of (typically more expensive) observations. We could then estimate distances to remaining galaxies using a suitable regression model, without the need toemploy the more expensive observation across the entire set. In astronomy circles, this is known as the “photometric redshift” problem.

Some important regression algorithms that we will discuss are linear regression (see “In Depth: Linear Regression” on page 390), support vector machines (see “In-Depth:Support Vector Machines” on page 405), and random forest regression (see “In- Depth: Decision Trees and Random Forests” on page 421).

Clustering: Inferring labels on unlabeled data

The classification and regression illustrations we just looked at are examples of super-vised learning algorithms, in which we are trying to build a model that will predict labels for new data. Unsupervised learning involves models that describe data withoutreference to any known labels.

One common case of unsupervised learning is “clustering,” in which data is automati-cally assigned to some number of discrete groups. For example, we might have some two-dimensional data like that shown in Figure 5-8.

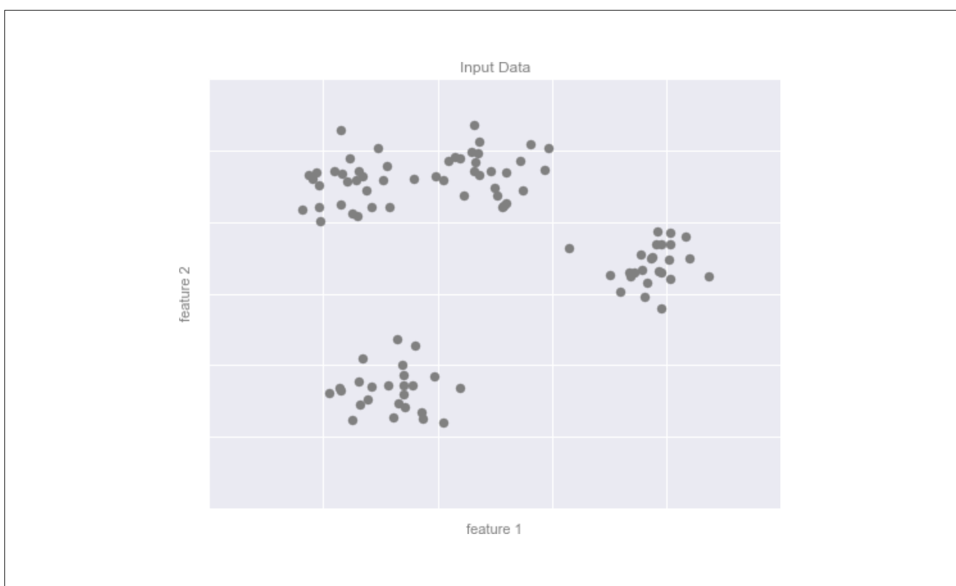


Figure 5-8. Example data for clustering

By eye, it is clear that each of these points is part of a distinct group. Given this input,a clustering model will use the intrinsic structure of the data to determine which points are related. Using the very fast and intuitive *k*-means algorithm (see “In Depth:k-Means Clustering” on page 462), we find the clusters shown in Figure 5-9.

k-means fits a model consisting of *k* cluster centers; the optimal centers are assumed to be those that minimize the distance of each point from its assigned center. Again, this might seem like a trivial exercise in two dimensions, but as our data becomes larger and more complex, such clustering algorithms can be employed to extract use- ful information from the dataset.

We will discuss the *k*-means algorithm in more depth in "In Depth: k-Means Cluster-ing" on page 462. Other important clustering algorithms include Gaussian mixture models (see "In Depth: Gaussian Mixture Models" on page 476) and spectral cluster-ing (see Scikit-Learn's clustering documentation).

Figure 5-9. Data labeled with a k-means clustering model

Dimensionality reduction: Inferring structure of unlabeled data

Dimensionality reduction is another example of an unsupervised algorithm, in which labels or other information are inferred from the structure of the dataset itself.Dimensionality reduction is a bit more abstract than the examples we looked at before, but generally it seeks to pull out some low-dimensional representation of datathat in some way preserves relevant qualities of the full dataset. Different dimension-ality reduction routines measure these relevant qualities in different ways, as we will see in "In-Depth: Manifold Learning" on page 445.

As an example of this, consider the data shown in Figure 5-10.

Visually, it is clear that there is some structure in this data: it is drawn from a one-dimensional line that is arranged in a spiral within this two-dimensional space. In a sense, you could say that this data is "intrinsically" only one dimensional, though thisone-dimensional data is embedded in higher-dimensional space. A suitable dimen- sionality reduction model in this case would be sensitive to this nonlinear embedded structure, and be able to pull out this

lower-dimensionality representation.

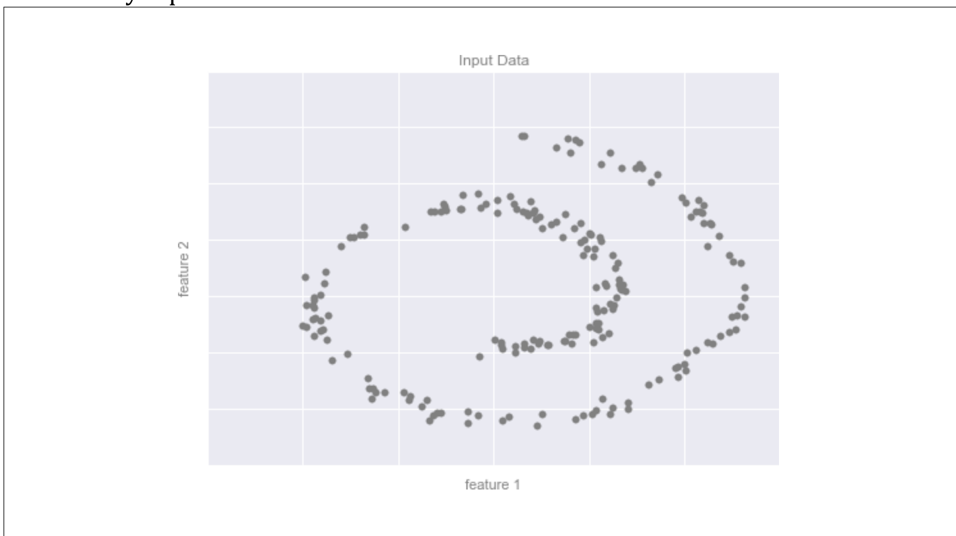


Figure 5-10. Example data for dimensionality reduction

Figure 5-11 presents a visualization of the results of the Isomap algorithm, a manifoldlearning algorithm that does exactly this.

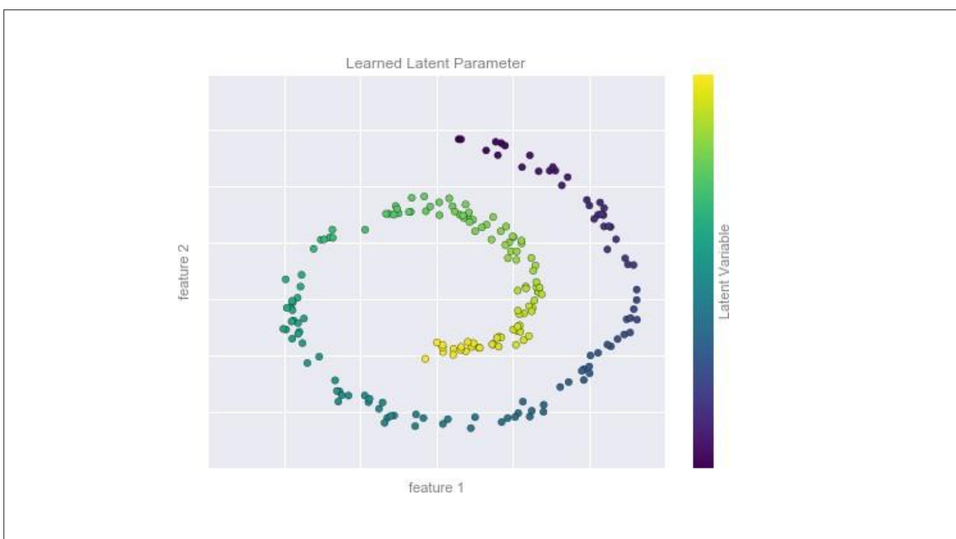


Figure 5-11. Data with a label learned via dimensionality reduction

Notice that the colors (which represent the extracted one-dimensional latent variable)change uniformly along the spiral, which indicates that the algorithm did in factdetect the structure we saw by eye. As with the previous examples, the power of

dimensionality reduction algorithms becomes clearer in higher-dimensional cases.For example, we might wish to visualize important relationships within a dataset that has 100 or 1,000 features. Visualizing 1,000-dimensional data is a challenge, and oneway we can make this more manageable is to use a dimensionality reduction techni- que to reduce the data to two or three dimensions.

Some important dimensionality reduction algorithms that we will discuss are princi- pal component analysis (see "In Depth: Principal Component Analysis" on page 433)and various manifold learning algorithms, including Isomap and locally linear embedding (see "In-Depth: Manifold Learning" on page 445).

Summary

Here we have seen a few simple examples of some of the basic types of machine learn-ing approaches. Needless to say, there are a number of important practical details thatwe have glossed over, but I hope this section was enough to give you a basic idea of what types of problems machine learning approaches can solve.

In short, we saw the following:

Supervised learning

Models that can predict labels based on labeled training data

Classification

Models that predict labels as two or more discrete categories

Regression

Models that predict continuous labels

Unsupervised learning

Models that identify structure in unlabeled data

Clustering

Models that detect and identify distinct groups in the data

Dimensionality reduction

Models that detect and identify lower-dimensional structure in higher-dimensional data

In the following sections we will go into much greater depth within these categories,and see some more interesting examples of where these concepts can be useful.

All of the figures in the preceding discussion are generated based on actual machinelearning computations; the code behind them can be found in the online appendix.

Day-05: Introducing Scikit-Learn

There are several Python libraries that provide solid implementations of a range of machine learning algorithms. One of the best known is Scikit-Learn, a package that provides efficient versions of a large number of common algorithms. Scikit-Learn is characterized by a clean, uniform, and streamlined API, as well as by very useful andcomplete online documentation. A benefit of this uniformity is that once you under- stand the basic use and syntax of Scikit-Learn for one type of model, switching to a new model or algorithm is very straightforward.

This section provides an overview of the Scikit-Learn API; a solid understanding of these API elements will form the foundation for understanding the deeper practical discussion of machine learning algorithms and approaches in the following chapters.

We will start by covering *data representation* in Scikit-Learn, followed by covering the *Estimator* API, and finally go through a more interesting example of using these toolsfor exploring a set of images of handwritten digits.

Data Representation in Scikit-Learn

Machine learning is about creating models from data: for that reason, we'll start by discussing how data can be represented in order to be understood by the computer. The best way to think about data within Scikit-Learn is in terms of tables of data.

Data as table

A basic table is a two-dimensional grid of data, in which the rows represent individ- ual elements of the dataset, and the columns represent quantities related to each of these elements. For example, consider the Iris dataset, famously analyzed by Ronald Fisher in 1936. We can download this dataset in the form of a Pandas `DataFrame` using the Seaborn library:

```
In[1]: import seaborn as sns

       iris = sns.load_dataset('iris')
       iris.head()
```

Out[1]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Here each row of the data refers to a single observed flower, and the number of rows is the

total number of flowers in the dataset. In general, we will refer to the rows of the matrix as *samples*, and the number of rows as n_samples.

Likewise, each column of the data refers to a particular quantitative piece of informa-tion that describes each sample. In general, we will refer to the columns of the matrixas *features*, and the number of columns as n_features.

Features matrix

This table layout makes clear that the information can be thought of as a two- dimensional numerical array or matrix, which we will call the *features matrix*. By con-vention, this features matrix is often stored in a variable named X. The features matrixis assumed to be two-dimensional, with shape [n_samples, n_features], and is most often contained in a NumPy array or a Pandas DataFrame, though some Scikit- Learn models also accept SciPy sparse matrices.

The samples (i.e., rows) always refer to the individual objects described by the dataset.For example, the sample might be a flower, a person, a document, an image, a sound file, a video, an astronomical object, or anything else you can describe with a set of quantitative measurements.

The features (i.e., columns) always refer to the distinct observations that describe each sample in a quantitative manner. Features are generally real-valued, but may be Boolean or discrete-valued in some cases.

Target array

In addition to the feature matrix X, we also generally work with a *label* or *target* array,which by convention we will usually call y. The target array is usually one dimen- sional, with length n_samples, and is generally contained in a NumPy array or Pan- das Series. The target array may have continuous numerical values, or discrete classes/labels. While some Scikit-Learn estimators do handle multiple target values inthe form of a two-dimensional [n_samples, n_targets] target array, we will pri- marily be working with the common case of a one-dimensional target array.

Often one point of confusion is how the target array differs from the other features columns. The distinguishing feature of the target array is that it is usually the quantitywe want to *predict from the data*: in statistical terms, it is the dependent variable. Forexample, in the preceding data we may wish to construct a model that can predict the species of flower based on the other measurements; in this case, the species columnwould be considered the feature.

With this target array in mind, we can use Seaborn (discussed earlier in “Visualiza-tion with Seaborn” on page 311) to conveniently visualize the data (see Figure 5-12):

```
In[2]: %matplotlib inline
```

```
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=1.5);
```

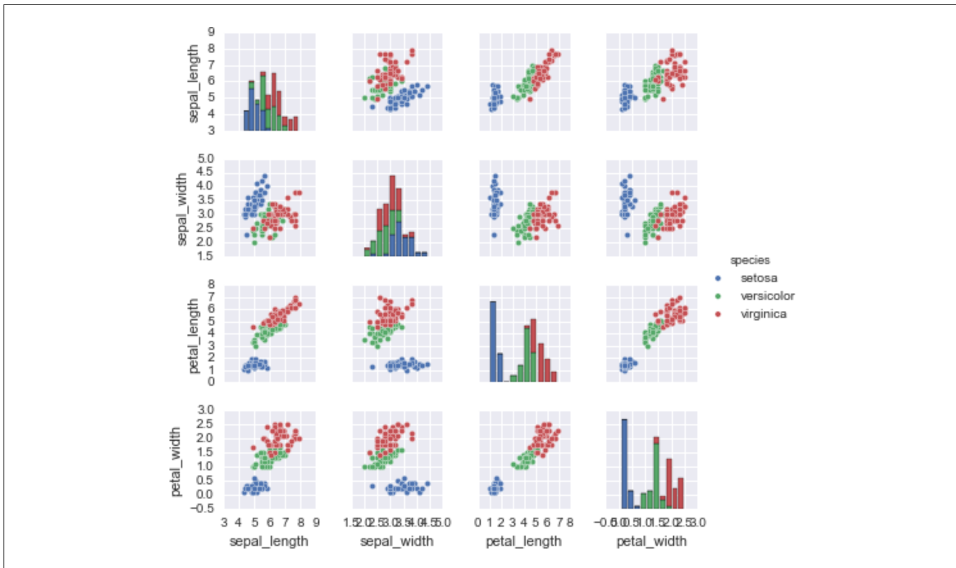


Figure 5-12. A visualization of the Iris dataset

For use in Scikit-Learn, we will extract the features matrix and target array from the `DataFrame`, which we can do using some of the Pandas `DataFrame` operations dis- cussed in Chapter 3:

```
In[3]: X_iris = iris.drop('species', axis=1)
       X_iris.shape

Out[3]: (150, 4)

In[4]: y_iris = iris['species']
       y_iris.shape

Out[4]: (150,)
```

To summarize, the expected layout of features and target values is visualized in Figure 5-13.

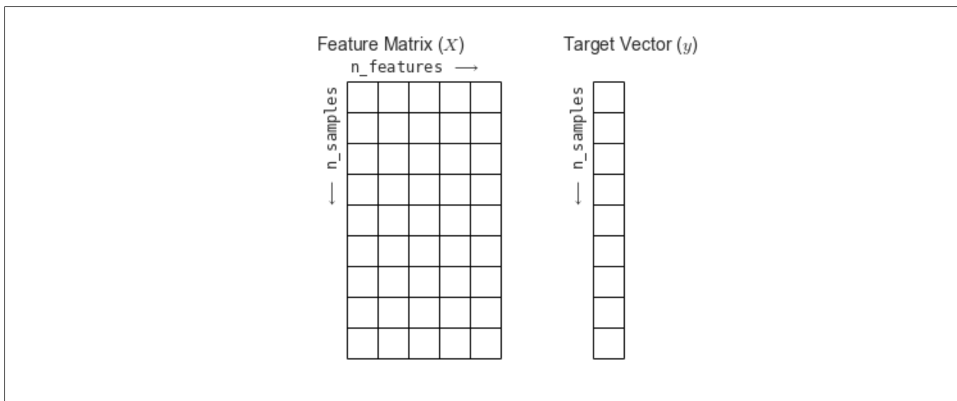


Figure 5-13. Scikit-Learn's data layout

With this data properly formatted, we can move on to consider the *estimator* API ofScikit-Learn.

Scikit-Learn's Estimator API

The Scikit-Learn API is designed with the following guiding principles in mind, as outlined in the Scikit-Learn API paper:

Consistency

All objects share a common interface drawn from a limited set of methods, withconsistent documentation.

Inspection

All specified parameter values are exposed as public attributes.

Limited object hierarchy

Only algorithms are represented by Python classes; datasets are represented in standard formats (NumPy arrays, Pandas `DataFrame`s, SciPy sparse matrices) andparameter names use standard Python strings.

Composition

Many machine learning tasks can be expressed as sequences of more fundamen-tal algorithms, and Scikit-Learn makes use of this wherever possible.

Sensible defaults

When models require user-specified parameters, the library defines an appropri- ate default value.

In practice, these principles make Scikit-Learn very easy to use, once the basic princi-ples are understood. Every machine learning algorithm in Scikit-Learn is imple- mented via the

Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API

Most commonly, the steps in using the Scikit-Learn estimator API are as follows (we will step through a handful of detailed examples in the sections that follow):

1. Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features matrix and target vector following the discussionfrom before.
4. Fit the model to your data by calling the `fit()` method of the model instance.
5. Apply the model to new data:

- For supervised learning, often we predict labels for unknown data using the `predict()` method.
- For unsupervised learning, we often transform or infer properties of the datausing the `transform()` or `predict()` method.

We will now step through several simple examples of applying supervised and unsu- pervised learning methods.

Supervised learning example: Simple linear regression

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to *x*, *y* data. We will use the following simple data for our regression example (Figure 5-14):

```
In[5]: import matplotlib.pyplot as plt
       import numpy as np

       rng = np.random.RandomState(42)
       x = 10 * rng.rand(50)

       y = 2 * x - 1 + rng.randn(50)
       plt.scatter(x, y);
```

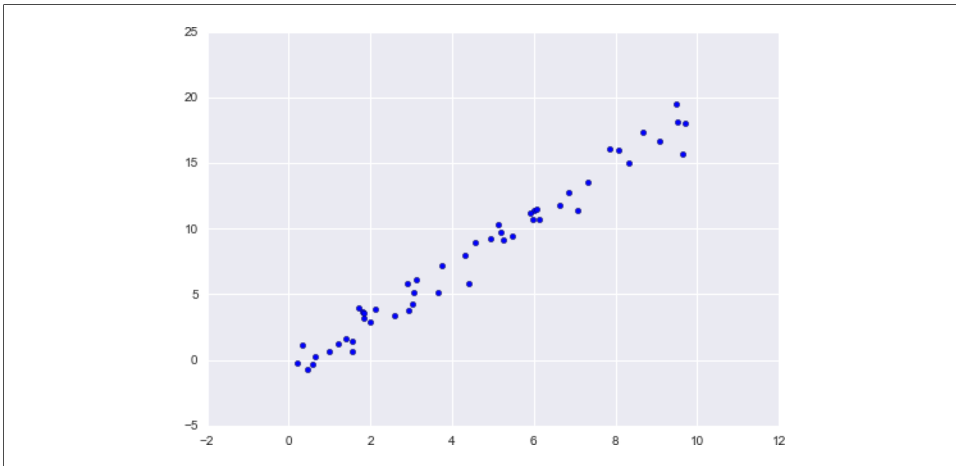


Figure 5-14. Data for linear regression

With this data in place, we can use the recipe outlined earlier. Let's walk through theprocess:

1. Choose a class of model.

   In Scikit-Learn, every class of model is represented by a Python class. So, for example, if we would like to compute a simple linear regression model, we can import the linear regression class:

   ```
   In[6]: from sklearn.linear_model import LinearRegression
   ```

   Note that other, more general linear regression models exist as well; you can readmore about them in the `sklearn.linear_model` module documentation.

2. Choose model hyperparameters.

   An important point is that *a class of model is not the same as an instance of a model*.

   Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answerone or more questions like the following:

   - Would we like to fit for the offset (i.e., intercept)?
   - Would we like the model to be normalized?
   - Would we like to preprocess our features to add model flexibility?
   - What degree of regularization would we like to use in our model?
   - How many model components would we like to use?

   These are examples of the important choices that must be made *once the modelclass is selected*. These choices are often represented as *hyperparameters*, or parameters that must be set before the model is fit to data. In Scikit-Learn, we choose hyperparameters by passing values at model instantiation. We will explore how you can quantitatively

motivate the choice of hyperparameters in “Hyperparameters and Model Validation” on page 359.

For our linear regression example, we can instantiate the LinearRegression class and specify that we would like to fit the intercept using the fit_inter cept hyperparameter:

```
In[7]: model = LinearRegression(fit_intercept=True)model

Out[7]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,
                         normalize=False)
```

Keep in mind that when the model is instantiated, the only action is the storingof these hyperparameter values. In particular, we have not yet applied the model to any data: the Scikit-Learn API makes very clear the distinctionbetween *choice of model* and *application of model to data*.

3. Arrange data into a features matrix and target vector.

   Previously we detailed the Scikit-Learn data representation, which requires a two-dimensional features matrix and a one-dimensional target array. Here our target variable y is already in the correct form (a length-n_samples array), but weneed to massage the data x to make it a matrix of size [n_samples, n_features].In this case, this amounts to a simple reshaping of the one-dimensional array:

```
In[8]: X = x[:, np.newaxis]X.shape

Out[8]: (50, 1)
```

4. Fit the model to your data.

   Now it is time to apply our model to data. This can be done with the fit() method of the model:

```
In[9]: model.fit(X, y)Out[9]:

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1,

                 normalize=False)
```

   This fit() command causes a number of model-dependent internal computa- tions to take place, and the results of these computations are stored in model- specific attributes that the user can explore. In Scikit-Learn, by convention all model parameters that were learned during the fit() process have trailing underscores; for example, in this linear model, we have the following:

```
In[10]: model.coef_

Out[10]: array([ 1.9776566])

In[11]: model.intercept_

Out[11]: -0.90331072553111635
```

These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing to the data definition, we see that they are very close to the input slope of 2 and intercept of –1.

One question that frequently comes up regards the uncertainty in such internal model parameters. In general, Scikit-Learn does not provide tools to draw conclusions from internal model parameters themselves: interpreting model parame- ters is much more a *statistical modeling* question than a *machine learning* question. Machine learning rather focuses on what the model *predicts*. If you would like to dive into the meaning of fit parameters within the model, other tools are available, including the StatsModels Python package.

Predict labels for unknown data.

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, we can do this using the `predict()` method. For the sake of this example, our "new data" will be a grid of *x* values, and we will ask what *y* values the model predicts:

```
In[12]: xfit = np.linspace(-1, 11)
```

As before, we need to coerce these *x* values into a `[n_samples, n_features]`

features matrix, after which we can feed it to the model:

```
In[13]: Xfit = xfit[:, np.newaxis]
        yfit = model.predict(Xfit)
```

Finally, let's visualize the results by plotting first the raw data, and then this model fit (Figure 5-15):

```
In[14]: plt.scatter(x, y)

        plt.plot(xfit, yfit);
```

Typically one evaluates the efficacy of the model by comparing its results to some known baseline, as we will see in the next example.

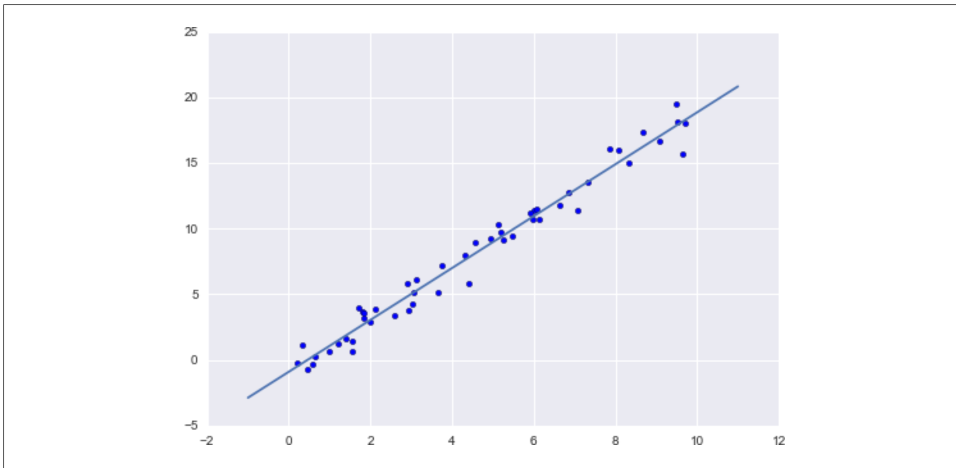


Figure 5-15. A simple linear regression fit to the data

Supervised learning example: Iris classification

Let's take a look at another example of this process, using the Iris dataset we discussedearlier. Our question will be this: given a model trained on a portion of the Iris data, how well can we predict the remaining labels?

For this task, we will use an extremely simple generative model known as Gaussian naive Bayes, which proceeds by assuming each class is drawn from an axis-aligned Gaussian distribution (see "In Depth: Naive Bayes Classification" on page 382 for more details). Because it is so fast and has no hyperparameters to choose, Gaussian naive Bayes is often a good model to use as a baseline classification, before you explore whether improvements can be found through more sophisticated models.

We would like to evaluate the model on data it has not seen before, and so we will split the data into a *training set* and a *testing set*. This could be done by hand, but it ismore convenient to use the `train_test_split` utility function:

```
In[15]: from sklearn.cross_validation import train_test_split

        Xtrain, Xtest, ytrain, ytest = train_test_split(X_iris, y_iris,

                                                        random_state=1)
```

With the data arranged, we can follow our recipe to predict the labels:

```
In[16]: from sklearn.naive_bayes import GaussianNB # 1. choose model class

        model = GaussianNB()                       # 2. instantiate model

        model.fit(Xtrain, ytrain)                  # 3. fit model to data

        y_model = model.predict(Xtest)             # 4. predict on new data
```

Finally, we can use the `accuracy_score` utility to see the fraction of predicted labelsthat

match their true value:

```
In[17]: from sklearn.metrics import accuracy_score
        accuracy_score(ytest, y_model)
```

```
Out[17]: 0.97368421052631582
```

With an accuracy topping 97%, we see that even this very naive classification algo- rithm is effective for this particular dataset!

Unsupervised learning example: Iris dimensionality

As an example of an unsupervised learning problem, let's take a look at reducing the dimensionality of the Iris data so as to more easily visualize it. Recall that the Iris datais four dimensional: there are four features recorded for each sample.

The task of dimensionality reduction is to ask whether there is a suitable lower- dimensional representation that retains the essential features of the data. Often dimensionality reduction is used as an aid to visualizing data; after all, it is much eas-ier to plot data in two dimensions than in four dimensions or higher!

Here we will use principal component analysis (PCA; see "In Depth: Principal Com- ponent Analysis" on page 433), which is a fast linear dimensionality reduction techni-que. We will ask the model to return two components—that is, a two-dimensional representation of the data.

Following the sequence of steps outlined earlier, we have:

```
In[18]:

from sklearn.decomposition import PCA  # 1. Choose the model class

model = PCA(n_components=2)            # 2. Instantiate the model with hyperparameters

model.fit(X_iris)                      # 3. Fit to data. Notice y is not specified!

X_2D = model.transform(X_iris)         # 4. Transform the data to two dimensions
```

Now let's plot the results. A quick way to do this is to insert the results into the origi- nal Iris `DataFrame`, and use Seaborn's `lmplot` to show the results (Figure 5-16):

```
In[19]: iris['PCA1'] = X_2D[:, 0]

        iris['PCA2'] = X_2D[:, 1]

        sns.lmplot("PCA1", "PCA2", hue='species', data=iris, fit_reg=False);
```

We see that in the two-dimensional representation, the species are fairly well separa- ted, even though the PCA algorithm had no knowledge of the species labels! This indicates to us that a relatively straightforward classification will probably be effectiveon the dataset, as we saw before.

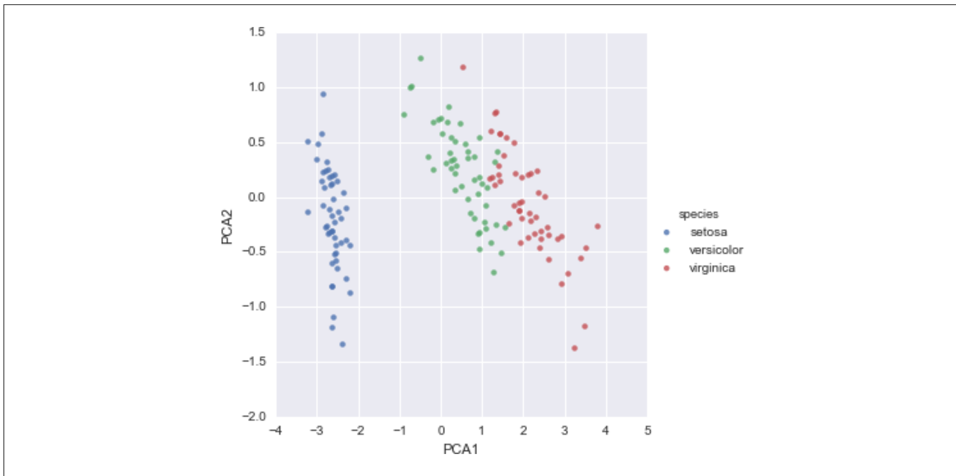


Figure 5-16. The Iris data projected to two dimensions

Unsupervised learning: Iris clustering

Let's next look at applying clustering to the Iris data. A clustering algorithm attemptsto find distinct groups of data without reference to any labels. Here we will use a powerful clustering method called a Gaussian mixture model (GMM), discussed in more detail in "In Depth: Gaussian Mixture Models" on page 476. A GMM attempts to model the data as a collection of Gaussian blobs.

We can fit the Gaussian mixture model as follows:

```
In[20]:

from sklearn.mixture import GMM        # 1. Choose the model class

model = GMM(n_components=3,

            covariance_type='full')  # 2. Instantiate the model w/ hyperparameters
model.fit(X_iris)                      # 3. Fit to data. Notice y is not specified!
y_gmm = model.predict(X_iris)          # 4. Determine cluster labels
```

As before, we will add the cluster label to the Iris `DataFrame` and use Seaborn to plotthe results (Figure 5-17):

```
In[21]:

iris['cluster'] = y_gmm

sns.lmplot("PCA1", "PCA2", data=iris, hue='species',
           col='cluster', fit_reg=False);
```

By splitting the data by cluster number, we see exactly how well the GMM algorithmhas recovered the underlying label: the *setosa* species is separated perfectly within cluster 0, while there remains a small amount of mixing between *versicolor* and *vir- ginica*. This means that even without an expert to tell us the species labels of the indi-vidual flowers, the

measurements of these flowers are distinct enough that we could *automatically* identify the presence of these different groups of species with a simple clustering algorithm! This sort of algorithm might further give experts in the fieldclues as to the relationship between the samples they are observing.

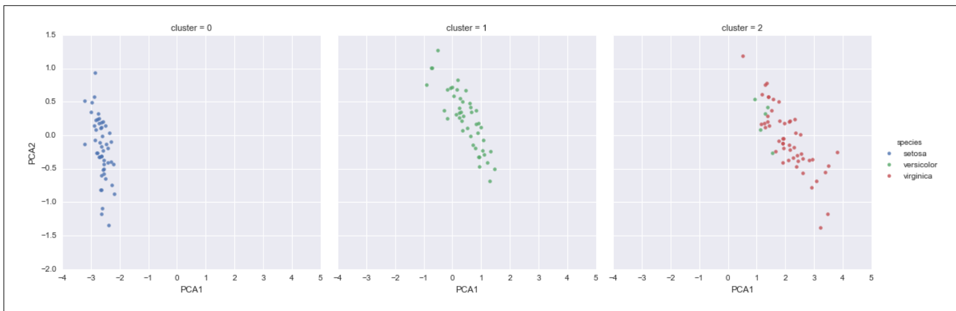


Figure 5-17. k-means clusters within the Iris data

Application: Exploring Handwritten Digits

To demonstrate these principles on a more interesting problem, let’s consider one piece of the optical character recognition problem: the identification of handwritten digits. In the wild, this problem involves both locating and identifying characters in an image. Here we’ll take a shortcut and use Scikit-Learn’s set of preformatted digits,which is built into the library.

Loading and visualizing the digits data

We’ll use Scikit-Learn’s data access interface and take a look at this data:

```
In[22]: from sklearn.datasets import load_digits
        digits = load_digits()
        digits.images.shape

Out[22]: (1797, 8, 8)
```

The images data is a three-dimensional array: 1,797 samples, each consisting of an 8×8 grid of pixels. Let’s visualize the first hundred of these (Figure 5-18):

```
In[23]: import matplotlib.pyplot as plt


        fig, axes = plt.subplots(10, 10, figsize=(8, 8),

                                 subplot_kw={'xticks':[], 'yticks':[]},
                                 gridspec_kw=dict(hspace=0.1, wspace=0.1))


        for i, ax in enumerate(axes.flat):

            ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
            ax.text(0.05, 0.05, str(digits.target[i]),

                    transform=ax.transAxes, color='green')
```

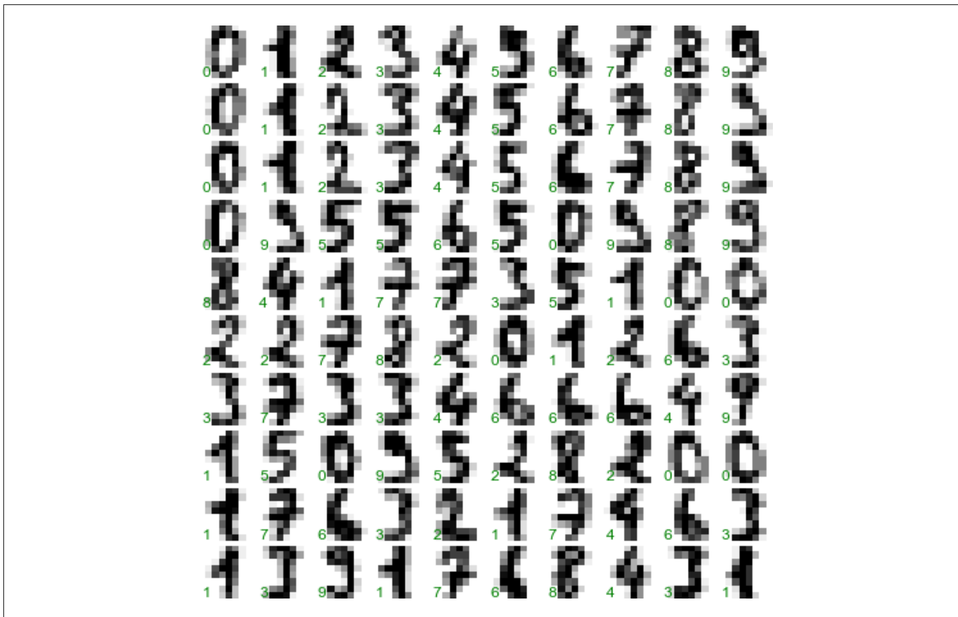


Figure 5-18. The handwritten digits data; each sample is represented by one 8×8 grid of pixels

In order to work with this data within Scikit-Learn, we need a two-dimensional, [n_samples, n_features] representation. We can accomplish this by treating each pixel in the image as a feature—that is, by flattening out the pixel arrays so that we have a length-64 array of pixel values representing each digit. Additionally, we need the target array, which gives the previously determined label for each digit. These twoquantities are built into the digits dataset under the data and target attributes,respectively:

```
In[24]: X = digits.data
        X.shape

Out[24]: (1797, 64)

In[25]: y = digits.target
        y.shape

Out[25]: (1797,)
```

We see here that there are 1,797 samples and 64 features.

Unsupervised learning: Dimensionality reduction

We'd like to visualize our points within the 64-dimensional parameter space, but it's difficult to effectively visualize points in such a high-dimensional space. Instead we'llreduce the dimensions to 2, using an unsupervised method. Here, we'll make use of a manifold learning algorithm called *Isomap* (see "In-Depth: Manifold Learning" onpage 445), and transform the

data to two dimensions:

```
In[26]: from sklearn.manifold import Isomap
        iso = Isomap(n_components=2)
        iso.fit(digits.data)

        data_projected = iso.transform(digits.data)
        data_projected.shape

Out[26]: (1797, 2)
```

We see that the projected data is now two-dimensional. Let's plot this data to see if wecan learn anything from its structure (Figure 5-19):

```
In[27]: plt.scatter(data_projected[:, 0], data_projected[:, 1], c=digits.target,
                    edgecolor='none', alpha=0.5,
                    cmap=plt.cm.get_cmap('spectral', 10))

        plt.colorbar(label='digit label', ticks=range(10))
        plt.clim(-0.5, 9.5);
```

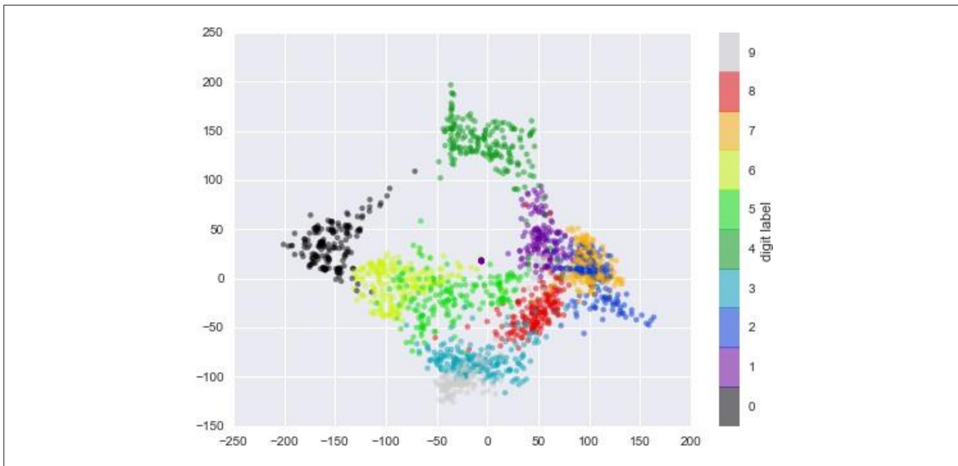


Figure 5-19. An Isomap embedding of the digits data

This plot gives us some good intuition into how well various numbers are separatedin the larger 64-dimensional space. For example, zeros (in black) and ones (in purple)have very little overlap in parameter space. Intuitively, this makes sense: a zero is empty in the middle of the image, while a one will generally have ink in the middle. On the other hand, there seems to be a more or less continuous spectrum between ones and fours: we can understand this by realizing that some people draw ones with "hats" on them, which cause them to look similar to fours.

Overall, however, the different groups appear to be fairly well separated in the param-eter space: this tells us that even a very straightforward supervised classification algo-rithm should perform suitably on this data. Let's give it a try.

Let's apply a classification algorithm to the digits. As with the Iris data previously, we will split the data into a training and test set, and fit a Gaussian naive Bayes model:

```
In[28]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
```

```
In[29]: from sklearn.naive_bayes import GaussianNB
        model = GaussianNB()

        model.fit(Xtrain, ytrain)
        y_model = model.predict(Xtest)
```

Now that we have predicted our model, we can gauge its accuracy by comparing the true values of the test set to the predictions:

```
In[30]: from sklearn.metrics import accuracy_score
        accuracy_score(ytest, y_model)

Out[30]: 0.83333333333333337
```

With even this extremely simple model, we find about 80% accuracy for classificationof the digits! However, this single number doesn't tell us *where* we've gone wrong—one nice way to do this is to use the *confusion matrix*, which we can compute with Scikit-Learn and plot with Seaborn (Figure 5-20):

```
In[31]: from sklearn.metrics import confusion_matrix

        mat = confusion_matrix(ytest, y_model)


        sns.heatmap(mat, square=True, annot=True, cbar=False)
        plt.xlabel('predicted value')

        plt.ylabel('true value');
```

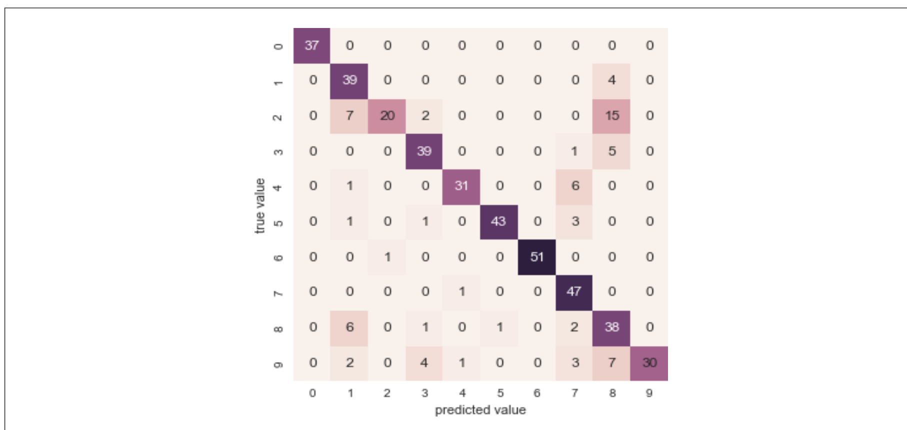


*Figure 5-20. A confusion matrix showing the frequency of **misclassifications** by our **classifier***

This shows us where the mislabeled points tend to be: for example, a large number oftwos here are misclassified as either ones or eights. Another way to gain intuition intothe characteristics of the model is to plot the inputs again, with their predicted labels. We'll use green for correct labels, and red for incorrect labels (Figure 5-21):

```
In[32]: fig, axes = plt.subplots(10, 10, figsize=(8, 8),

                                  subplot_kw={'xticks':[], 'yticks':[]},
                                  gridspec_kw=dict(hspace=0.1, wspace=0.1))

        for i, ax in enumerate(axes.flat):

            ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
            ax.text(0.05, 0.05, str(y_model[i]),

                    transform=ax.transAxes,

                    color='green' if (ytest[i] == y_model[i]) else 'red')
```

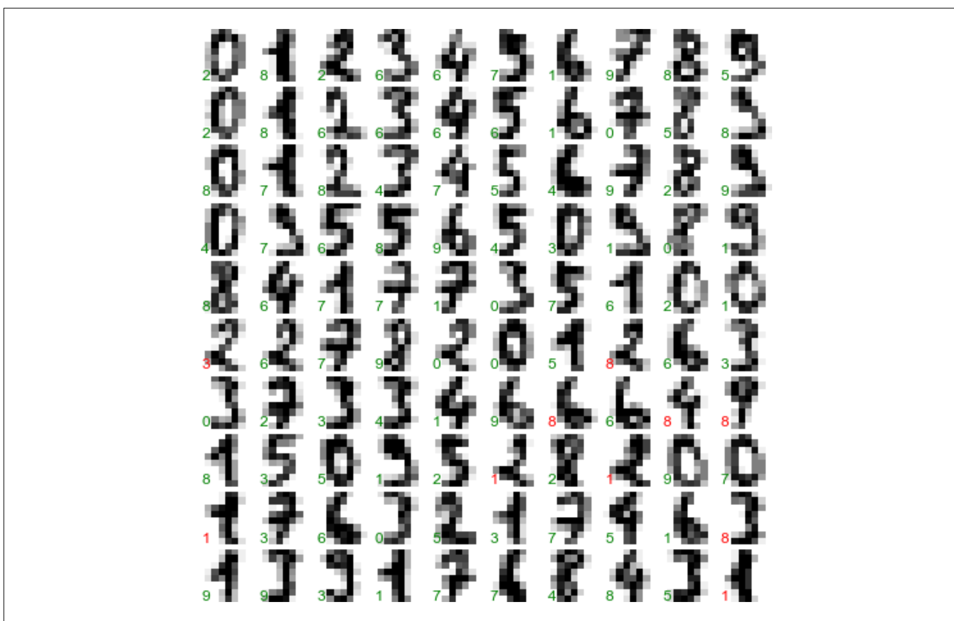


Figure 5-21. Data showing correct (green) and incorrect (red) labels; for a color versionof this plot, see the online appendix

Examining this subset of the data, we can gain insight regarding where the algorithm might not be performing optimally. To go beyond our 80% classification rate, we might move to a more sophisticated algorithm, such as support vector machines (see "In-Depth: Support Vector Machines" on page 405) or random forests (see "In- Depth: Decision Trees and Random Forests" on page 421), or another classification approach.

Week 4- Data visualization using Matplotlib

Day-01: Data visualization using Matplotlib

Introduction and brief histroy

Matplotlib is a multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plottingvia gnuplot from the IPython command line. IPython's creator, Fernando Perez, was at the time scrambling to finish his PhD, and let John know he wouldn't have time to review the patch for several months. John took this as a cue to set out on his own, andthe Matplotlib package was born, with version 0.1 released in 2003. It received an early boost when it was adopted as the plotting package of choice of the Space Tele- scope Science Institute (the folks behind the Hubble Telescope), which financially supported Matplotlib's development and greatly expanded its capabilities.

One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large userbase, which in turn has led to an active developer base and Mat-plotlib's powerful tools and ubiquity within the scientific Python world.

Importing matplotlib

Just as we use the np shorthand for NumPy and the pd shorthand for Pandas, we willuse some standard shorthands for Matplotlib imports:

```
In[1]: import matplotlib as mpl
       import matplotlib.pyplot as plt
```

The pltinterface is what we will use most often, as we'll see throughout this chapter.

Setting Styles

We will use the plt.styledirective to choose appropriate aesthetic styles for our fig-ures. Here we will set the classic style, which ensures that the plots we create use the classic Matplotlib style:

```
In[2]: plt.style.use('classic')
```

show() or No show()? How to Display Your Plots

A visualization you can't see won't be of much use, but just how you view your Mat-

plotlib plots depends on the context. The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in a script, in an IPython terminal, or in an IPython notebook.

Plotting from a script

If you are using Matplotlib from within a script, the function plt.show() is your friend. plt.show() starts an event loop, looks for all currently active figure objects,and opens one or more interactive windows that display your figure or figures.

So, for example, you may have a file called *myplot.py* containing the following:

```
# ------- file: myplot.py ------

import matplotlib.pyplot as pltimport
nump as np

x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))

plt.plot(x, np.cos(x))

plt.show()
```

You can then run this script from the command-line prompt, which will result in a window opening with your figure displayed:

```
$ python myplot.py
```

The plt.show() command does a lot under the hood, as it must interact with your system's interactive graphical backend. The details of this operation can vary greatly from system to system and even installation to installation, but Matplotlib does its best to hide all these details from you.

One thing to be aware of: the plt.show() command should be used *only once* per Python session, and is most often seen at the very end of the script. Multiple show() commands can lead to unpredictable backend-dependent behavior, and should mostly be avoided.

Plotting from an IPython shell

It can be very convenient to use Matplotlib interactively within an IPython shell (see Chapter 1). IPython is built to work well with Matplotlib if you specify Matplotlib mode. To enable this mode, you can use the %matplotlibmagic command after start-ing ipython:

```
In [1]: %matplotlib

Using matplotlib backend: TkAgg
```

```
In [2]: import matplotlib.pyplot as plt
```

At this point, any plt plot command will cause a figure window to open, and further commands can be run to update the plot. Some changes (such as modifying proper- ties of lines that are already drawn) will not draw automatically; to force an update, use plt.draw(). Using plt.show()in Matplotlib mode is not required.

Plotting from an IPython notebook

Plotting interactively within an IPython notebook can be done with the %matplotlib command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

- %matplotlib notebook will lead to *interactive* plots embedded within thenotebook
- %matplotlib inline will lead to *static* images of your plot embedded in the notebook

For this book, we will generally opt for %matplotlib inline:

```
In[3]: %matplotlib inline
```

After you run this command (it needs to be done only once per kernel/session), any cell within the notebook that creates a plot will embed a PNG image of the resulting graphic (Figure 4-1):

```
In[4]: import numpy as np
       x = np.linspace(0, 10, 100)
```

Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of for- mats. You can save a figure using the savefig() command. For example, to save the previous figure as a PNG file, you can run this:

```
In[5]: fig.savefig('my_figure.png')
```

We now have a file called *my_figure.png* in the current working directory:

```
In[6]: !ls -lh my_figure.png

-rw-r--r--  1 jakevdp  staff         16K Aug 11 10:59 my_figure.png
```

To confirm that it contains what we think it contains, let's use the IPython Image object to display the contents of this file .

```
In[7]: from IPython.display import Image
       Image('my_figure.png')
```

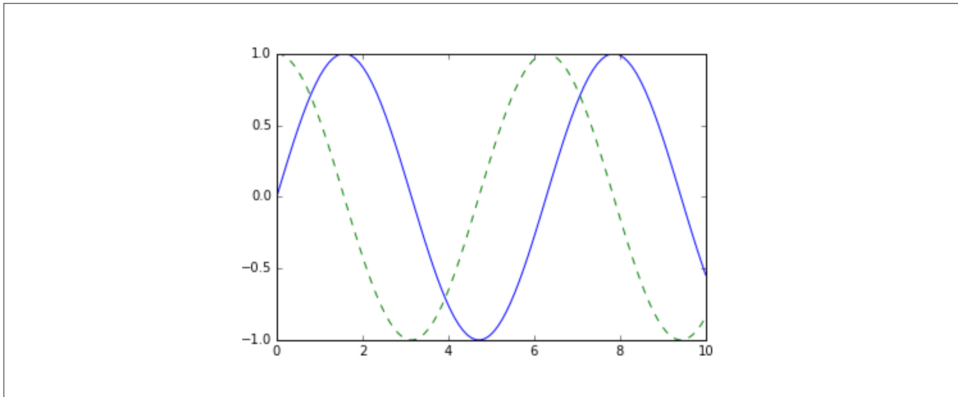


Figure. PNG rendering of the basic plot

In savefig(), the file format is inferred from the extension of the given filename. Depending on what backends you have installed, many different file formats are available. You can find the list of supported file types for your system by using the following method of the figure canvasobject:

```
In[8]: fig.canvas.get_supported_filetypes()Out[8]:
{'eps': 'Encapsulated Postscript',

        'jpeg': 'Joint Photographic Experts Group','jpg':
        'Joint Photographic Experts Group', 'pdf':
        'Portable Document Format',

        'pgf': 'PGF code for LaTeX',

        'png': 'Portable Network Graphics','ps':
        'Postscript',

        'raw': 'Raw RGBA bitmap', 'rgba':
        'Raw RGBA bitmap',

        'svg': 'Scalable Vector Graphics', 'svgz':
        'Scalable Vector Graphics','tif': 'Tagged
        Image File Format', 'tiff': 'Tagged Image
        File Format'}
```

Note that when saving your figure, it’s not necessary to use plt.show() or related commands discussed earlier.

Two Interfaces for the Price of One

A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface. We'll quickly highlight the differences between the two here.

MATLAB-style interface

Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface. For example, the following code will probably look quite familiar to MATLAB users (Figure):

```
In[9]: plt.figure()  # create a plot figure

       # create the first of two panels and set current axis
       plt.subplot(2, 1, 1) # (rows, columns, panel number)
       plt.plot(x, np.sin(x))

       # create the second panel and set current axis

       plt.subplot(2, 1, 2) plt.plot(x,np.cos(x));
```

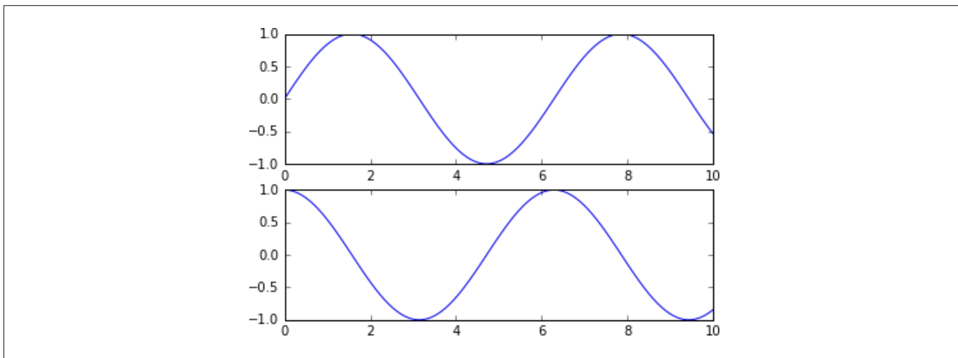


Figure. Subplots using the MATLAB-style interface

It's important to note that this interface is *stateful*: it keeps track of the "current" figure and axes, which are where all pltcommands are applied. You can get a reference to these using the plt.gcf() (get current figure) and plt.gca() (get current axes) routines.

While this stateful interface is fast and convenient for simple plots, it is easy to run into problems. For example, once the second panel is created, how can we go back and add something to the first? This is possible within the MATLAB-style interface, but a bit clunky. Fortunately, there is a better way.

Object-oriented interface

The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an “active” figure or axes, in the object-oriented interface the plotting func-tions are *methods* of explicit Figure and Axes objects. To re-create the previous plot using this style of plotting, you might do the following

```
In[10]: # First create a grid of plots
        # ax will be an array of two Axes objects
        fig, ax = plt.subplots(2)

        # Call plot() method on the appropriate object
        ax[0].plot(x, np.sin(x))
        ax[1].plot(x, np.cos(x));
```

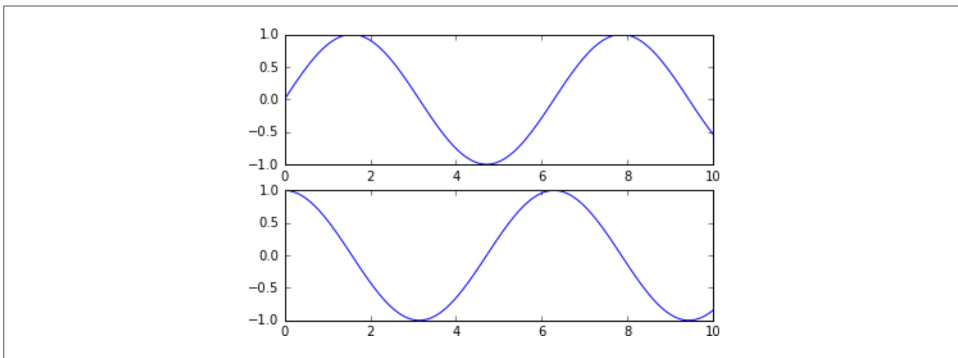


Figure. Subplots using the object-oriented interface

For more simple plots, the choice of which style to use is largely a matter of prefer- ence, but the object-oriented approach can become a necessity as plots become more complicated. Throughout this chapter, we will switch between the MATLAB-style and object-oriented interfaces, depending on what is most convenient. In most cases, the difference is as small as switching plt.plot() to ax.plot(), but there are a few gotchas that we will highlight as they come up in the following sections.

Simple Line Plots

Perhaps the simplest of all plots is the visualization of a single function $y = f(x)$. Here we will take a first look at creating a simple plot of this type. As with all the followingsections, we’ll start by setting up the notebook for plotting and importing the func- tions we will use:

```
In[1]: %matplotlib inline

       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')import
       numpy as np
```

For all Matplotlib plots, we start by creating a figure and an axes. In their simplest form, a figure and axes can be created as follows (Figure 4-5):

```
In[2]: fig = plt.figure()ax =
       plt.axes()
```

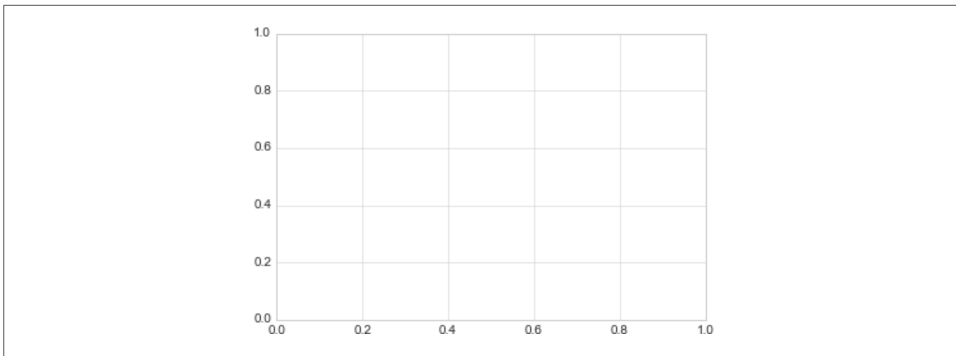
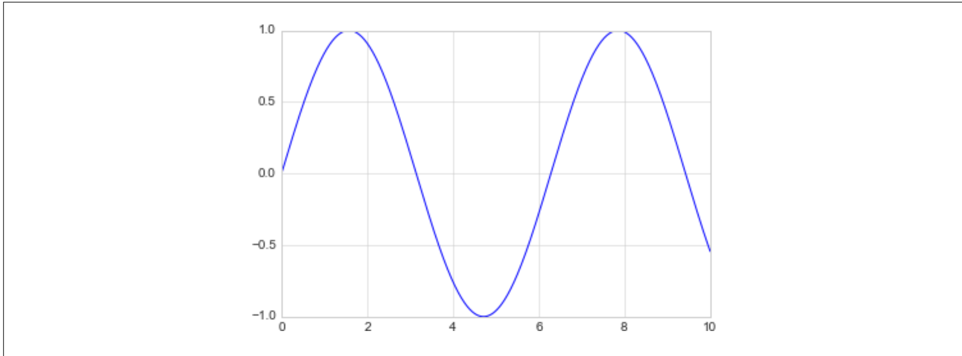


Figure. An empty gridded axes

In Matplotlib, the *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels. The *axes* (an instance of the class plt.Axes) is what we see above: a boundingbox with ticks and labels, which will eventually contain the plot elements that make up our visualization. Throughout this book, we'll commonly use the variable name fig to refer to a figure instance, and ax to refer to an axes instance or group of axes instances.

Once we have created an axes, we can use the ax.plot function to plot some data. Let's start with a simple sinusoid .

```
In[3]: fig = plt.figure()ax
       = plt.axes()

       x = np.linspace(0, 10,
       1000)ax.plot(x, np.sin(x));

In[4]: plt.plot(x, np.sin(x));
```



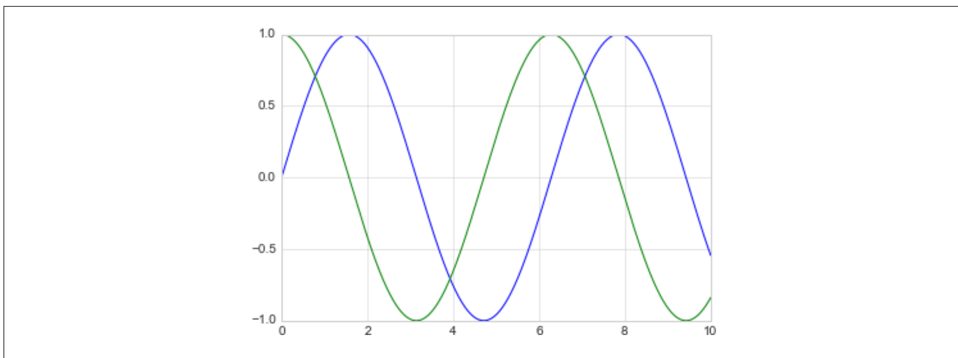


Figure . Over-plotting multiple lines

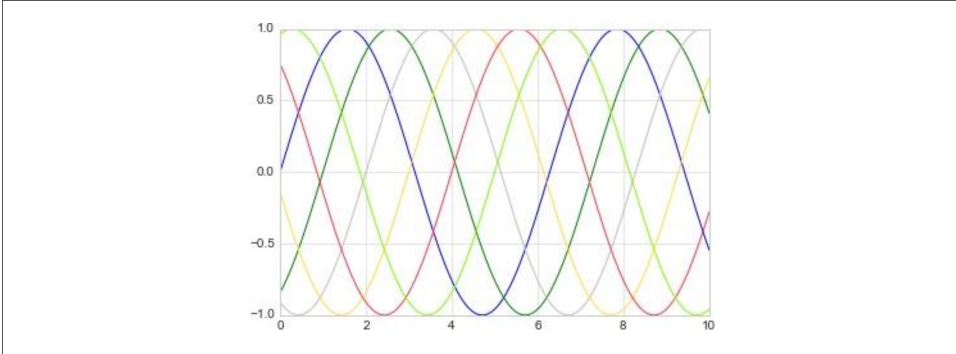
That's all there is to plotting simple functions in Matplotlib! We'll now dive into somemore details about how to control the appearance of the axes and lines.

Adjusting the Plot: Line Colors and Styles

The first adjustment you might wish to make to a plot is to control the line colors and styles. The plt.plot() function takes additional arguments that can be used to spec-ify these. To adjust the color, you can use the color keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways (Figure :

In[6]:

```
plt.plot(x, np.sin(x - 0), color='blue')          # specify color  by  name
plt.plot(x, np.sin(x - 1), color='g')             # short color code (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75')          # Grayscale between 0 and 1
plt.plot(x, np.sin(x - 3), color='#FFDD44')       # Hex code (RRGGBB from 00 to
FF)plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values 0 and 1
```



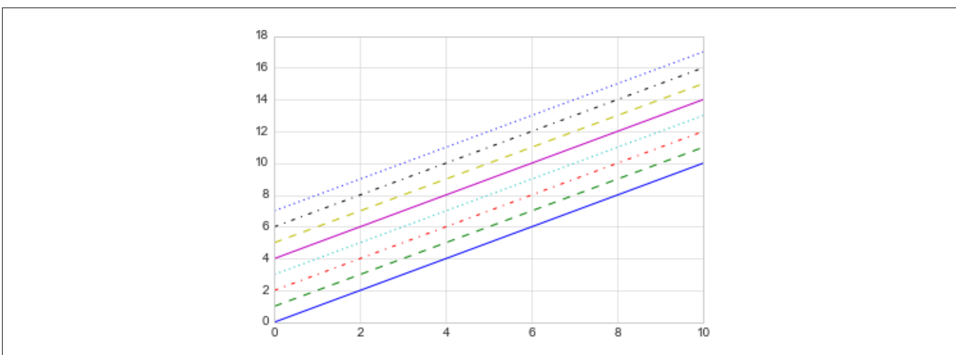
```
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all HTML color names
supported
```

Figure . Controlling the color of plot elements

If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines.

Similarly, you can adjust the line style using the linestyle keyword (Figure 4-10):

```
In[7]: plt.plot(x, x + 0, linestyle='solid')
       plt.plot(x, x + 1, linestyle='dashed')
       plt.plot(x, x + 2, linestyle='dashdot')
       plt.plot(x, x + 3, linestyle='dotted');


       # For short, you can use the following
       codes:plt.plot(x, x + 4, linestyle='-') # solid
```



```
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
```

Figure . Example of various line styles

If you would like to be extremely terse, these linestyle and color codes can be com-bined into a single nonkeyword argument to the plt.plot()function (Figure) :

```
In[8]: plt.plot(x, x + 0, '-g') # solid green
       plt.plot(x, x + 1, '--c') # dashed cyan
       plt.plot(x, x + 2, '-.k') # dashdot black
       plt.plot(x, x + 3, ':r'); # dotted red
```

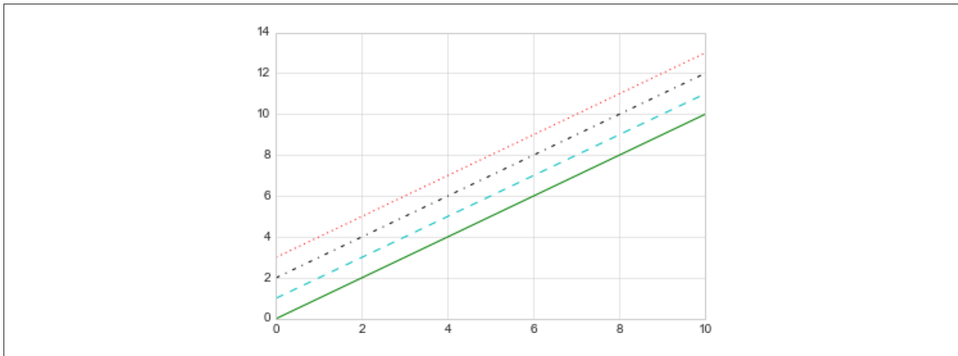


Figure . Controlling colors and styles with the shorthand syntax

These single-character color codes reflect the standard abbreviations in the RGB (Red/Green/Blue) and CMYK (Cyan/Magenta/Yellow/blacK) color systems, com- monly used for digital color graphics.

There are many other keyword arguments that can be used to fine-tune the appear- ance of the plot; for more details, I'd suggest viewing the docstring of the plt.plot()function using IPython's help tools (see "Help and Documentation in IPython").

Adjusting the Plot: Axes Limits

Matplotlib does a decent job of choosing default axes limits for your plot, but some- times it's nice to have finer control. The most basic way to adjust axis limits is to use the plt.xlim()and plt.ylim()methods (Figure 4-12):

```
In[9]: plt.plot(x, np.sin(x))
```

```
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```

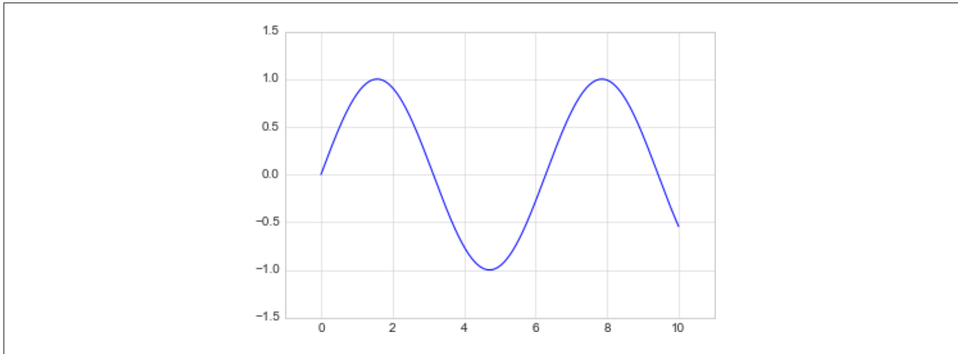


Figure 4-12. Example of setting axis limits

If for some reason you'd like either axis to be displayed in reverse, you can simply reverse the order of the arguments (Figure):

```
In[10]: plt.plot(x, np.sin(x))
        plt.xlim(10, 0)
        plt.ylim(1.2, -1.2);
```

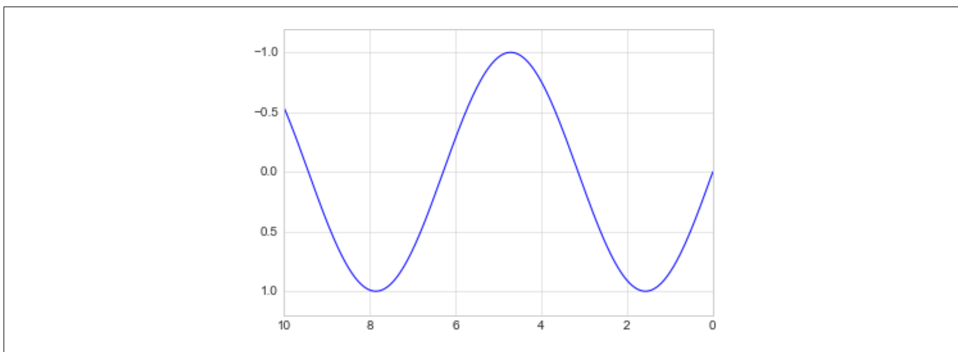


Figure . Example of reversing the y-axis

A useful related method is plt.axis() (note here the potential confusion between *axes* with an *e*, and *axis* with an *i*). The plt.axis() method allows you to set the x and y limits with a single call, by passing a list that specifies [xmin, xmax, ymin, ymax]

```
In[11]: plt.plot(x, np.sin(x))
```

```
plt.axis([-1, 11, -1.5, 1.5]);
```

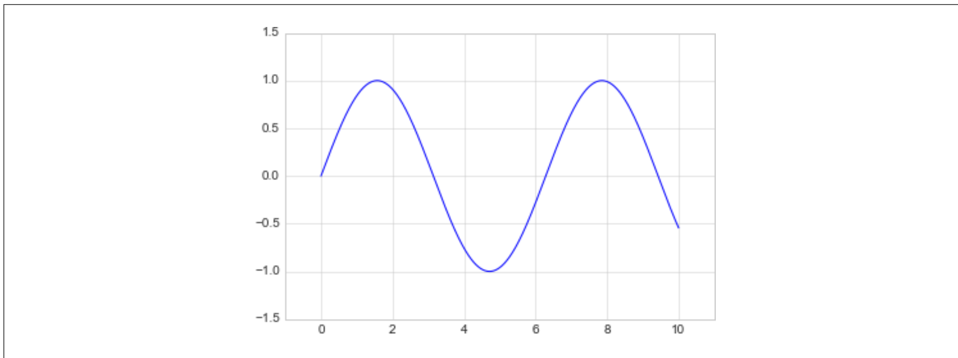


Figure .Setting the axis limits with plt.axis

The plt.axis()method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot (Figure 4-15):

```
In[12]: plt.plot(x, np.sin(x))
        plt.axis('tight');
```

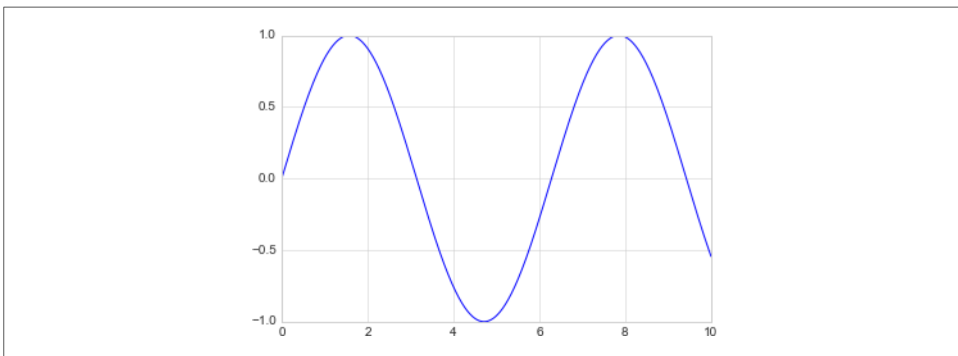


Figure . Example of a “tight” layout

It allows even higher-level specifications, such as ensuring an equal aspect ratio sothat on your screen, one unit in xis equal to one unit in y(Figure) :

```
In[13]: plt.plot(x, np.sin(x))
        plt.axis('equal');
```

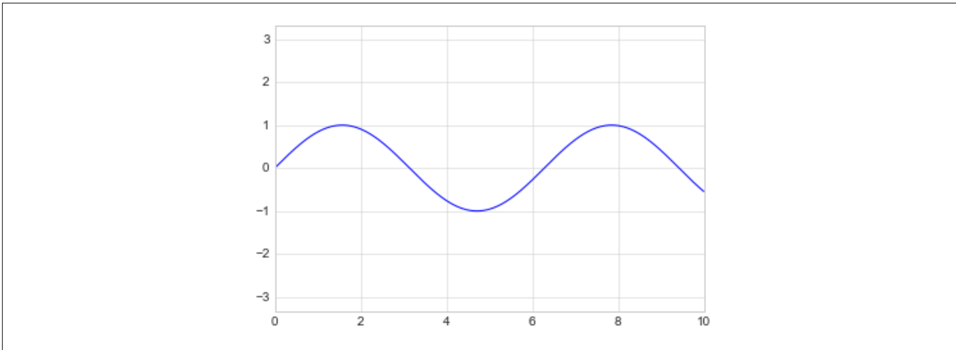


Figure . Example of an “equal” layout, with units matched to the output resolution

For more information on axis limits and the other capabilities of the plt.axis() method, refer to the plt.axis()docstring.

Labeling Plots

As the last piece of this section, we’ll briefly look at the labeling of plots: titles, axislabels, and simple legends.

Titles and axis labels are the simplest such labels—there are methods that can be usedto quickly set them .

```
In[14]: plt.plot(x, np.sin(x))
        plt.title("A Sine Curve")

        plt.xlabel("x")
        plt.ylabel("sin(x)");
```

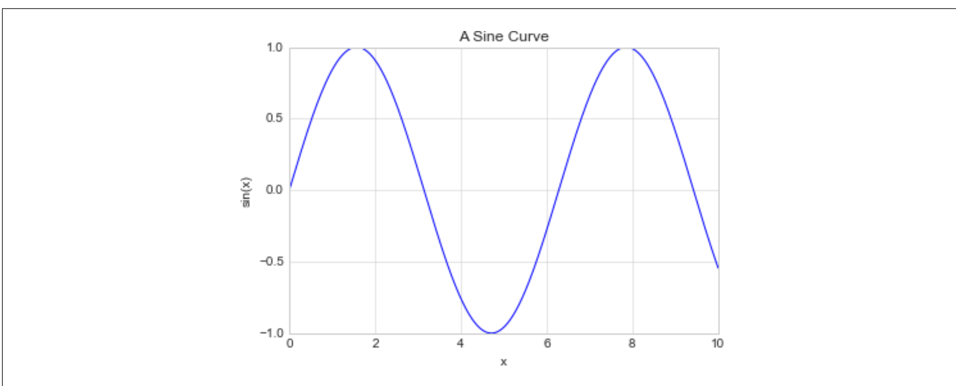


Figure . Examples of axis labels and title

You can adjust the position, size, and style of these labels using optional arguments tothe function. For more information, see the Matplotlib documentation and the doc- strings of each of these functions.

When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type. Again, Matplotlib has a built-in way of quickly creating such a legend. It is done via the (you guessed it) plt.legend() method. Though there are several valid ways of using this, I find it easiest to specify the label of each line using the labelkeyword of the plot function (Figure):

```
In[15]: plt.plot(x, np.sin(x), '-g', label='sin(x)')
        plt.plot(x, np.cos(x), ':b', label='cos(x)')
        plt.axis('equal')

        plt.legend();
```

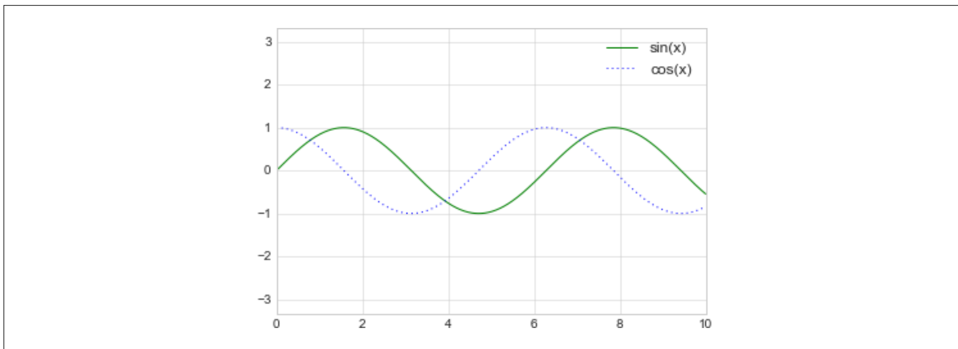


Figure . Plot legend example

As you can see, the plt.legend()function keeps track of the line style and color, and matches these with the correct label. More information on specifying and formatting plot legends can be found in the plt.legend() docstring;

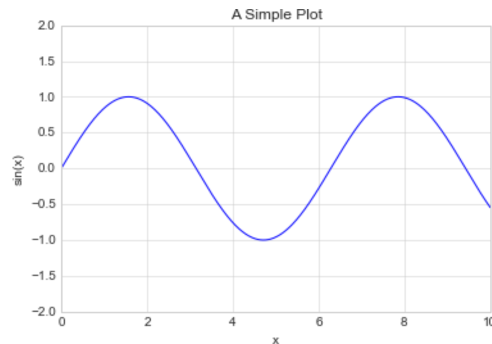
Matplotlib Gotchas

While most `plt` functions translate directly to `ax` methods (such as `plt.plot()` → `ax.plot()`, `plt.legend()` → `ax.legend()`, etc.), this is not the case for all commands. In particular, functions to set limits, labels, and titles are slightly modified. For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:

- `plt.xlabel()` → `ax.set_xlabel()`
- `plt.ylabel()` → `ax.set_ylabel()`
- `plt.xlim()` → `ax.set_xlim()`
- `plt.ylim()` → `ax.set_ylim()`
- `plt.title()` → `ax.set_title()`

In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the `ax.set()` method to set all these properties at once (Figure 4-19):

```
In[16]: ax = plt.axes()

        ax.plot(x, np.sin(x))
        ax.set(xlim=(0, 10), ylim=(-2, 2),

               xlabel='x', ylabel='sin(x)',
               title='A Simple Plot');
```



Simple Scatter Plots

Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented

individually with a dot, circle, or other shape. We'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline

       import matplotlib.pyplot as plt
       plt.style.use('seaborn-
       whitegrid')import numpy as np
```

Scatter Plots with plt.plot

In the previous section, we looked at plt.plot/ax.plotto produce line plots. It turnsout that this same function can produce scatter plots as well (Figure):

```
In[2]: x = np.linspace(0, 10, 30)
       y = np.sin(x)


       plt.plot(x, y, 'o', color='black');
```

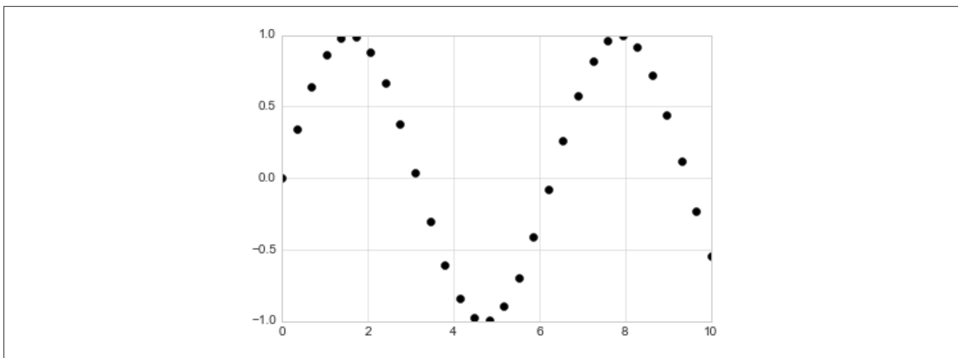


Figure . Scatter plot example

The third argument in the function call is a character that represents the type of sym- bol used for the plotting. Just as you can specify options such as '-'and '--'to con-trol the line style, the marker style has its own set of short string codes. The full list ofavailable symbols can be seen in the documentation of plt.plot, or in Matplotlib's online documentation. Most of the possibilities are fairly intuitive, and we'll show a number of the more common ones here (Figure):

```
In[3]: rng = np.random.RandomState(0)
       for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
```

```
plt.plot(rng.rand(5), rng.rand(5), marker,
         label="marker='{0}'".format(mark
         er))
```

```
plt.legend(numpoints
=1) plt.xlim(0, 1.8);
```

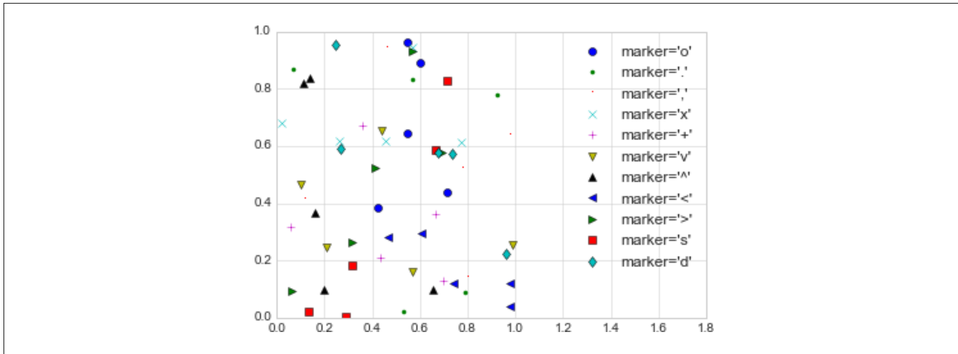


Figure. Demonstration of point numbers

For even more possibilities, these character codes can be used together with line andcolor codes to plot points along with a line connecting them (Figure):

```
In[4]: plt.plot(x, y, '-ok');       # line (-), circle marker (o), black (k)
```

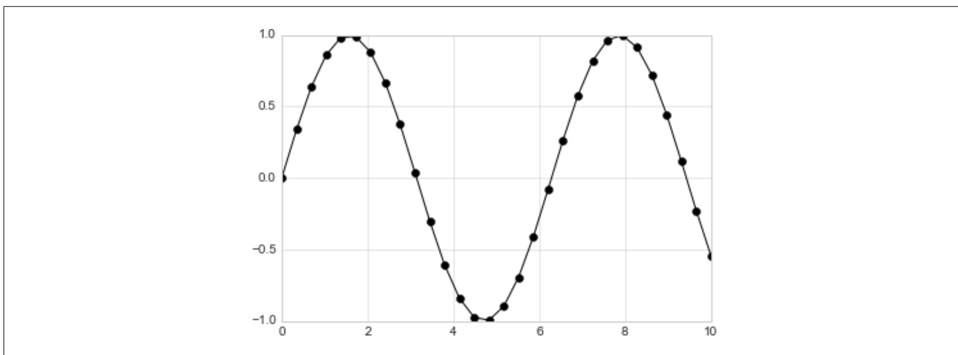


Figure . Combining line and point markers

Additional keyword arguments to plt.plotspecify a wide range of properties of thelines and markers (Figure):

```
In[5]: plt.plot(x, y, '-p', color='gray',
```

```
markersize=15,
linewidth=4,
markerfacecolor='white',
markeredgecolor='gray',
markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

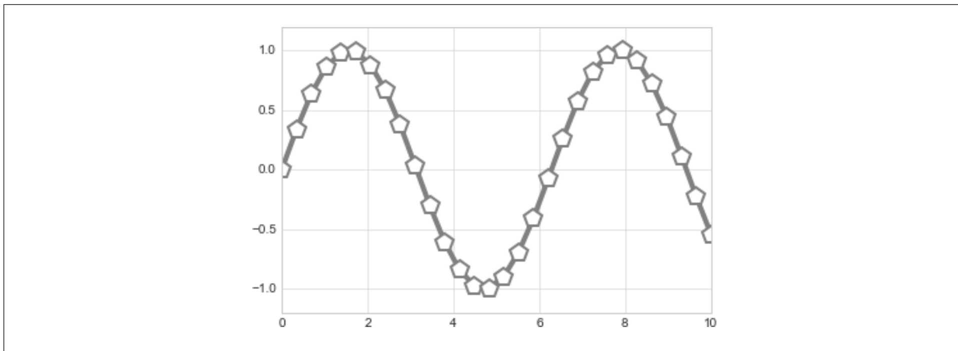


Figure . Customizing line and point numbers

This type of flexibility in the plt.plot function allows for a wide variety of possible visualization options. For a full description of the options available, refer to the plt.plot documentation.

Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the plt.scatter func- tion, which can be used very similarly to the plt.plotfunction (Figure):

```
In[6]: plt.scatter(x, y, marker='o');
```

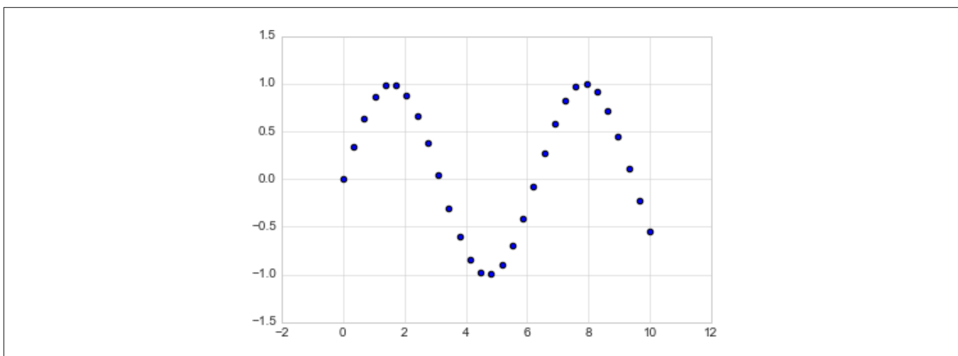


Figure . A simple scatter plot

The primary difference of plt.scatterfrom plt.plotis that it can be used to createscatter plots where the properties of each individual point (size, face color, edge color,etc.) can be individually controlled or mapped to data.

Let’s show this by creating a random scatter plot with points of many colors and sizes.In order to better see the overlapping results, we’ll also use the alpha keyword to adjust the transparency level

```
In[7]: rng =
        np.random.RandomState(0)
        x = rng.randn(100)

        y = rng.randn(100)
        colors =
        rng.rand(100)

        sizes = 1000 * rng.rand(100)


        plt.scatter(x, y, c=colors, s=sizes, alpha=0.3,
                    cmap='viridis')

        plt.colorbar();  # show color scale
```

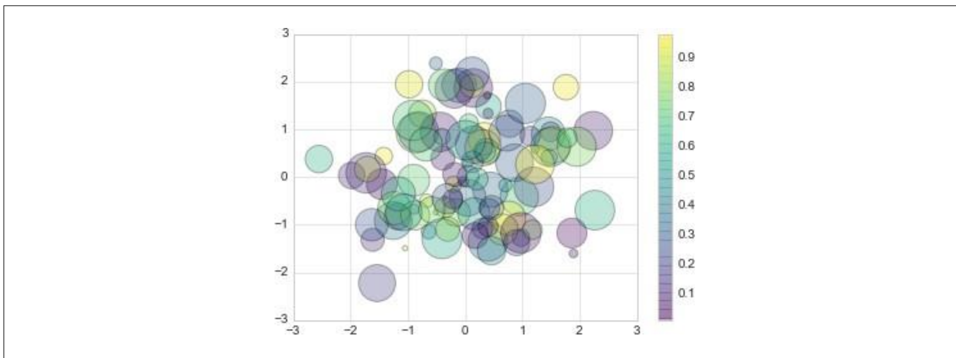


Figure . Changing size, color, and transparency in scatter points

Notice that the color argument is automatically mapped to a color scale (shown here by the colorbar() command), and the size argument is given in pixels. In this way, the color and size of points can be used to convey information in the visualization, in order to illustrate multidimensional data.

For example, we might use the Iris data from Scikit-Learn, where each sample is one of three types of flowers that has had the size of its petals and sepals carefully meas- ured.

```
In[8]: from sklearn.datasets import
        load_irisiris = load_iris()
```

```
features = iris.data.T


plt.scatter(features[0], features[1], alpha=0.2,
            s=100*features[3], c=iris.target,
            cmap='viridis')

plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
;
```

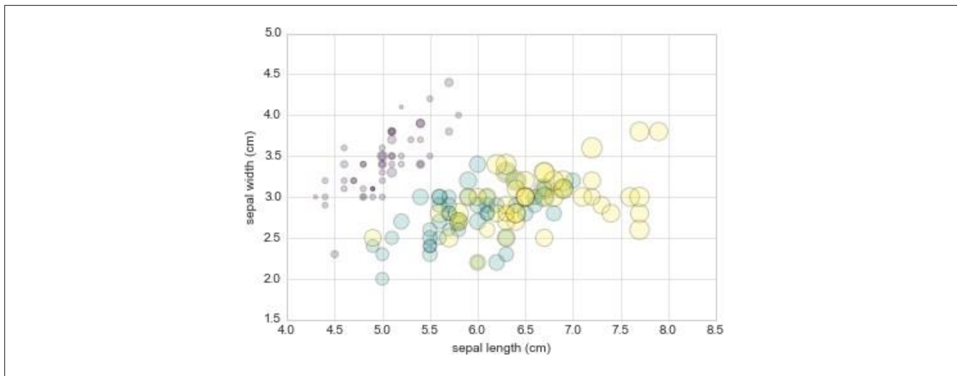


Figure . Using point properties to encode features of the Iris data

We can see that this scatter plot has given us the ability to simultaneously explore four different dimensions of the data: the (x, y) location of each point corresponds to the sepal length and width, the size of the point is related to the petal width, and the color is related to the particular species of flower. Multicolor and multifeature scatterplots like this can be useful for both exploration and presentation of data.

plot Versus scatter: A Note on Efficiency

Aside from the different features available in plt.plot and plt.scatter, why mightyou choose to use one over the other? While it doesn't matter as much for small amounts of data, as datasets get larger than a few thousand points, plt.plot can be noticeably more efficient than plt.scatter. The reason is that plt.scatter has the capability to render a different size and/or color for each point, so the renderer must do the extra work of constructing each point individually. In plt.plot, on the other hand, the points are always essentially clones of each other, so the work of determin- ing the appearance of the points is done only once for the entire set of data. For large datasets, the difference between these two can lead to vastly different performance, and for this reason, plt.plot should be preferred over plt.scatter for large datasets.

Visualizing Errors

For any scientific measurement, accurate accounting for errors is nearly as important,if not more important, than accurate reporting of the number itself. For example, imagine that I am using some astrophysical observations to estimate the Hubble Con-stant, the local measurement of the expansion rate of the universe. I know that the current literature suggests a value of around 71 (km/s)/Mpc, and I measure a value of74 (km/s)/Mpc with my method. Are the values consistent? The only correct answer, given this information, is this: there is no way to know.

Suppose I augment this information with reported uncertainties: the current litera- ture suggests a value of around 71 ± 2.5 (km/s)/Mpc, and my method has measured a value of 74 ± 5 (km/s)/Mpc. Now are the values consistent? That is a question that can be quantitatively answered.

In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

Basic Errorbars

A basic errorbar can be created with a single Matplotlib function call (Figure 4-27):

```
In[1]: %matplotlib inline

       import matplotlib.pyplot as plt
       plt.style.use('seaborn-
       whitegrid')import numpy as np

In[2]: x = np.linspace(0, 10, 50)
       dy = 0.8

       y = np.sin(x) + dy *

       np.random.randn(50)plt.errorbar(x, y,

       yerr=dy, fmt='.k');
```

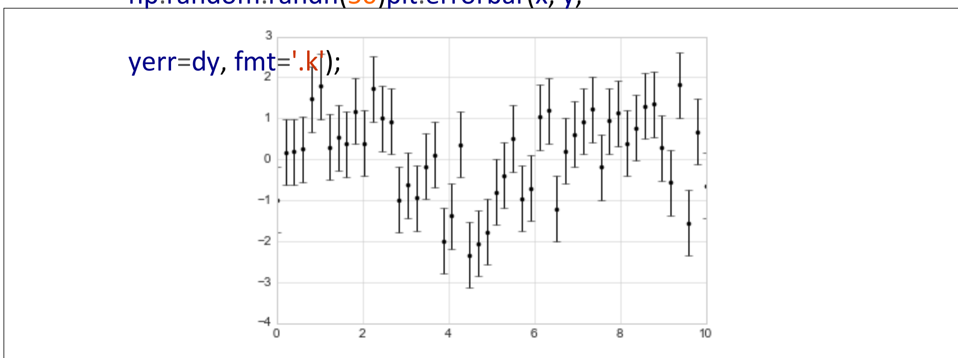


Figure . An errorbar example

In addition to these basic options, the errorbar function has many options to fine- tune the outputs. Using these additional options you can easily customize the aesthet-ics of your errorbar plot. I often find it helpful, especially in crowded plots, to make the errorbars lighter than the points themselves

```
In[3]: plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
                    ecolor='lightgray', elinewidth=3, capsize=0);
```

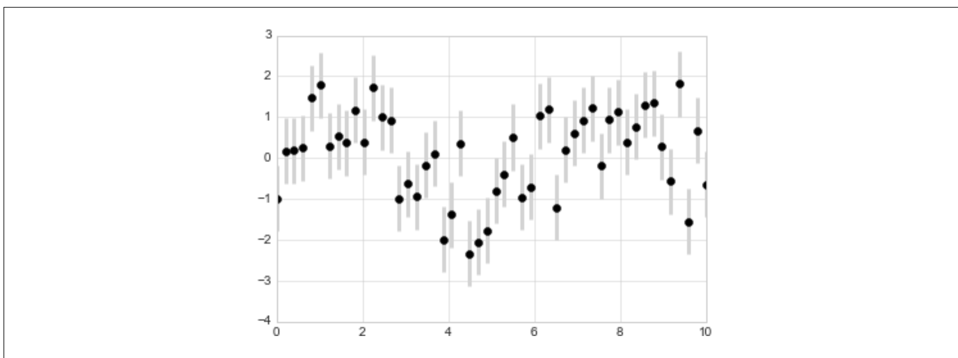


Figure. Customizing errorbars

In addition to these options, you can also specify horizontal errorbars (xerr), one- sided errorbars, and many other variants. For more information on the options avail- able, refer to the docstring of plt.errorbar.

Continuous Errors

In some situations it is desirable to show errorbars on continuous quantities. Though Matplotlib does not have a built-in convenience routine for this type of application, it's relatively easy to combine primitives like plt.plot and plt.fill_between for a useful result.

Here we'll perform a simple *Gaussian process regression* (GPR), using the Scikit-Learn API (see "Introducing Scikit-Learn" on page 343 for details). This is a method of fit- ting a very flexible nonparametric function to data with a continuous measure of the uncertainty. We won't delve into the details of Gaussian process regression at this point, but will focus instead on how you might visualize such a continuous error measurement:

```
In[4]: from sklearn.gaussian_process import GaussianProcess
```

```
# define the model and draw some data

model = lambda x: x * np.sin(x)
xdata = np.array([1, 3, 5, 6, 8])
ydata = model(xdata)


# Compute the Gaussian process fit

gp = GaussianProcess(corr='cubic', theta0=1e-2, thetaL=1e-4,
                     thetaU=1E-1,random_start=100)

gp.fit(xdata[:, np.newaxis], ydata)


xfit = np.linspace(0, 10, 1000)

yfit, MSE = gp.predict(xfit[:, np.newaxis],
eval_MSE=True) dyfit = 2 * np.sqrt(MSE)  # 2*sigma ~
95% confidence region
```

We now have xfit, yfit, and dyfit, which sample the continuous fit to our data. Wecould pass these to the plt.errorbar function as above, but we don't really want to plot 1,000 points with 1,000 errorbars. Instead, we can use the plt.fill_between function with a light color to visualize this continuous error:

```
In[5]: # Visualize the result

plt.plot(xdata, ydata, 'or')
plt.plot(xfit, yfit, '-', color='gray')


plt.fill_between(xfit, yfit - dyfit, yfit + dyfit,

                 color='gray', alpha=0.2)

plt.xlim(0, 10);
```

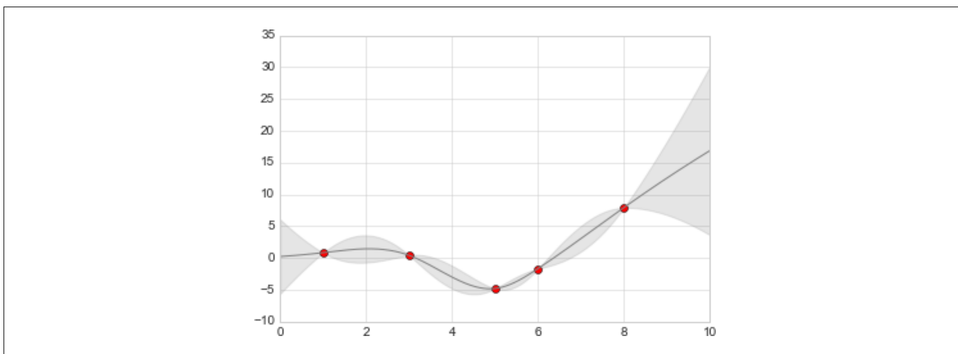


Figure. Representing continuous uncertainty with filled regions

Note what we've done here with the fill_between function: we pass an x value, thenthe lower y-bound, then the upper y-bound, and the result is that the area between these regions is filled.

The resulting figure gives a very intuitive view into what the Gaussian process regres-sion algorithm is doing: in regions near a measured data point, the model is strongly constrained and this is reflected in the small model errors. In regions far from a measured data point, the model is not strongly constrained, and the model errors increase.

For more information on the options available in plt.fill_between() (and the closely related plt.fill() function), see the function docstring or the Matplotlib documentation.

Finally, if this seems a bit too low level for your taste, refer to "Visualization with Sea-born" , where we discuss the Seaborn package, which has a more stream-lined API for visualizing this type of continuous errorbar.

Density and Contour Plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions. There are three Matplotlib functions that can be helpful for this task: plt.contour for contour plots, plt.contourf for filled contourplots, and plt.imshow for showing images. This section looks at several examples of using these. We'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline

import matplotlib.pyplot as
pltplt.style.use('seaborn-
white') import numpy as np
```

Day-02: Visualizing a Three-Dimensional Function

We'll start by demonstrating a contour plot using a function $z = f(x, y)$, using the fol- lowing particular choice for f (we've seen this before in "Computation on Arrays: Broadcasting" on page 63, when we used it as a motivating example for arraybroadcasting):

```
In[2]: def f(x, y):

        return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

A contour plot can be created with the plt.contour function. It takes three argu- ments: a grid of *x* values, a grid of *y* values, and a grid of *z* values. The *x* and *y* valuesrepresent

positions on the plot, and the *z* values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays:

```
In[3]: x = np.linspace(0, 5, 50)
       y = np.linspace(0, 5, 40)

       X, Y = np.meshgrid(x,
       y)Z = f(X, Y)
```

Now let's look at this with a standard line-only contour plot (Figure 4-30):

```
In[4]: plt.contour(X, Y, Z, colors='black');
```

A contour plot can be created with the plt.contour function. It takes three argu- ments: a grid of *x* values, a grid of *y* values, and a grid of *z* values. The *x* and *y* valuesrepresent positions on the plot, and the *z* values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the np.meshgrid function, which builds two-dimensional grids from one-dimensional arrays:

```
In[3]: x = np.linspace(0, 5, 50)
       y = np.linspace(0, 5, 40)

       X, Y = np.meshgrid(x,
       y)Z = f(X, Y)
```

Now let's look at this with a standard line-only contour plot (Figure 4-30):

```
In[4]: plt.contour(X, Y, Z, colors='black');
```

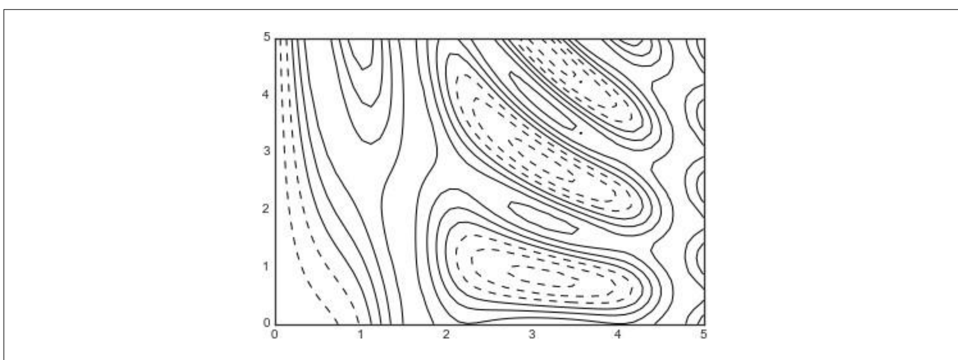



Figure . Visualizing three-dimensional data with contours

Notice that by default when a single color is used, negative values are represented by

dashed lines, and positive values by solid lines. Alternatively, you can color-code thelines by specifying a colormap with the cmap argument. Here, we'll also specify thatwe want more lines to be drawn—20 equally spaced intervals within the data range (Figure):

```
In[5]: plt.contour(X, Y, Z, 20, cmap='RdGy');
```

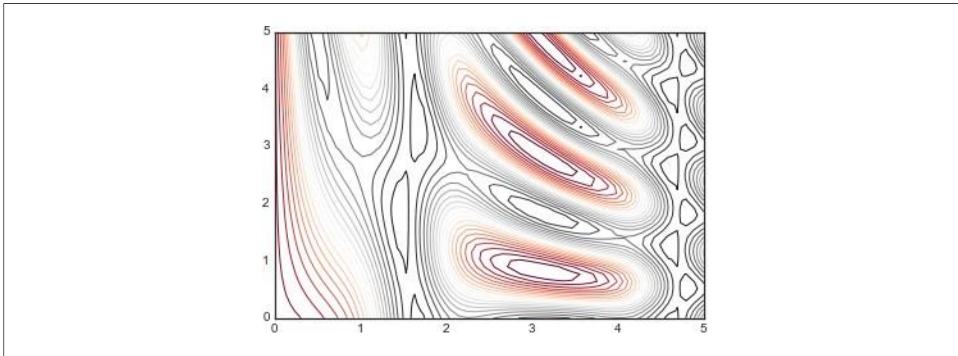


Figure . Visualizing three-dimensional data with colored contours

Here we chose the RdGy (short for *Red-Gray*) colormap, which is a good choice for centered data. Matplotlib has a wide range of colormaps available, which you can easily browse in IPython by doing a tab completion on the plt.cmmodule:

```
plt.cm.<TAB>
```

Our plot is looking nicer, but the spaces between the lines may be a bit distracting. We can change this by switching to a filled contour plot using the plt.contourf() function (notice the f at the end), which uses largely the same syntax as plt.con tour().

Additionally, we'll add a plt.colorbar() command, which automatically creates an additional axis with labeled color information for the plot (Figure):

```
In[6]: plt.contourf(X, Y, Z, 20,
         cmap='RdGy')plt.colorbar();
```

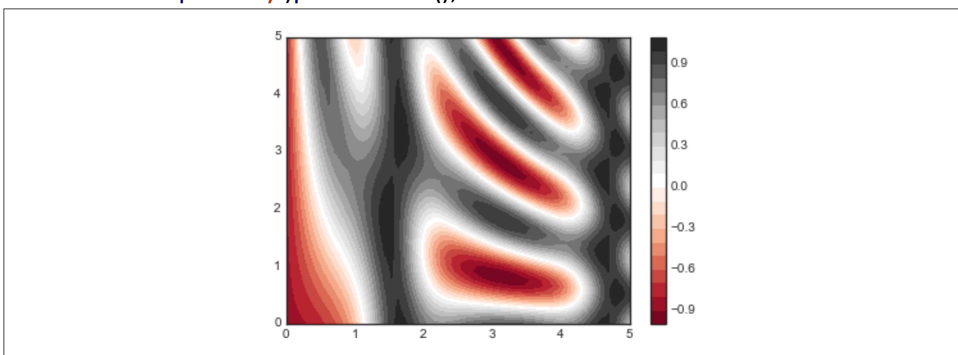


Figure. Visualizing three-dimensional data with filled contours

The colorbar makes it clear that the black regions are “peaks,” while the red regions are “valleys.”

One potential issue with this plot is that it is a bit “splotchy.” That is, the color steps are discrete rather than continuous, which is not always what is desired. You could remedy this by setting the number of contours to a very high number, but this results in a rather inefficient plot: Matplotlib must render a new polygon for each step in the level. A better way to handle this is to use the plt.imshow() function, which inter- prets a two-dimensional grid of data as an image.

Figure . shows the result of the following code:

```
In[7]: plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',

                    cmap='RdGy'
            ) plt.colorbar()
            plt.axis(aspect='image')
            ;
```

There are a few potential gotchas with imshow(), however:

- plt.imshow()doesn’t accept an *x* and *y* grid, so you must manually specify the

  extent [*xmin*, *xmax*, *ymin*, *ymax*]of the image on the plot.

- plt.imshow() by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. Thismust be changed when showing gridded data.
- plt.imshow() will automatically adjust the axis aspect ratio to match the input data; you can change this by setting, for example, plt.axis(aspect='image')to make *x* and *y* units match.

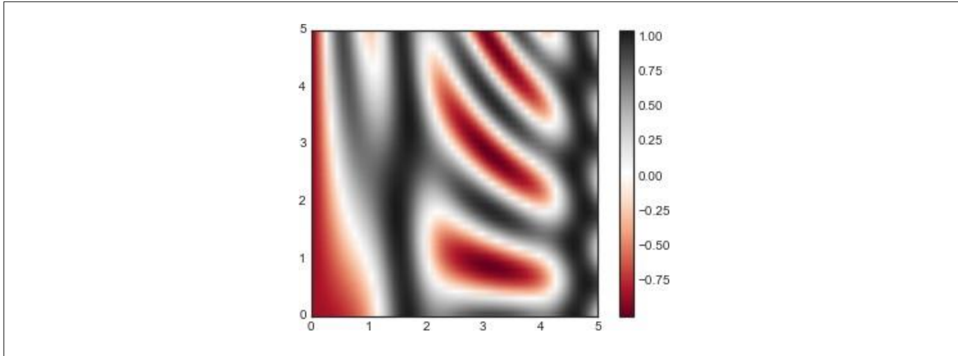


Figure 4-33. Representing three-dimensional data as an image

Finally, it can sometimes be useful to combine contour plots and image plots. For example, to create the effect shown in Figure , we'll use a partially transparent background image (with transparency set via the alpha parameter) and over-plot contours with labels on the contours themselves (using the plt.clabel()function):

```
In[8]: contours = plt.contour(X, Y, Z, 3, colors='black')
       plt.clabel(contours, inline=True, fontsize=8)


       plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
                  cmap='RdGy', alpha=0.5)

       plt.colorbar();
```

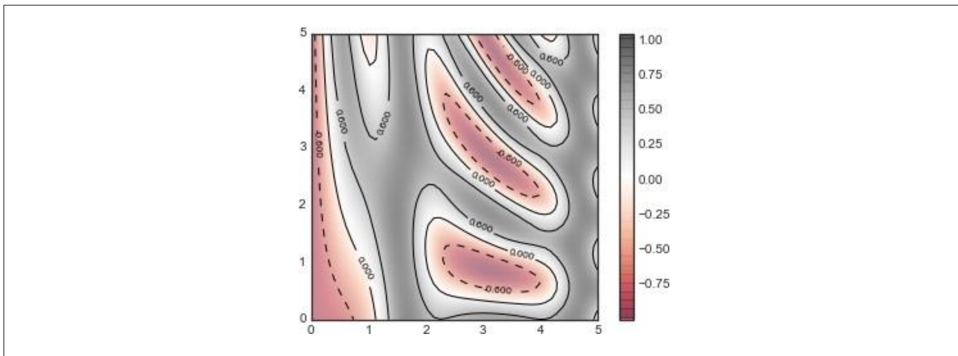


Figure 4-34. Labeled contours on top of an image

The combination of these three functions—plt.contour, plt.contourf, andplt.imshow—gives nearly limitless possibilities for displaying this sort of three- dimensional data

within a two-dimensional plot. For more information on theoptions available in these functions, refer to their docstrings. If you are interested in three-dimensional visualizations of this type of data, see “Three-Dimensional Plotting in Matplotlib” .

Histograms, Binnings, and Density

A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib’s histogram function (see “Comparisons, Masks, and Boolean Logic”), which creates a basic histogram in one line, once the normal boilerplate imports are done (Figure):

```
In[1]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-white')

       data =
np.random.randn(1000)In[2]:
plt.hist(data);
```

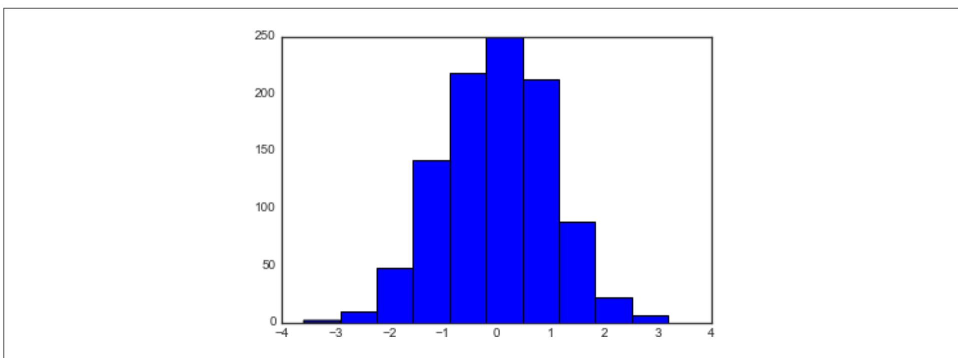


Figure . A simple histogram

The hist()function has many options to tune both the calculation and the display;here’s an example of a more customized histogram .

```
In[3]: plt.hist(data, bins=30, normed=True, alpha=0.5,
                histtype='stepfilled', color='steelblue',
                edgecolor='none');
```

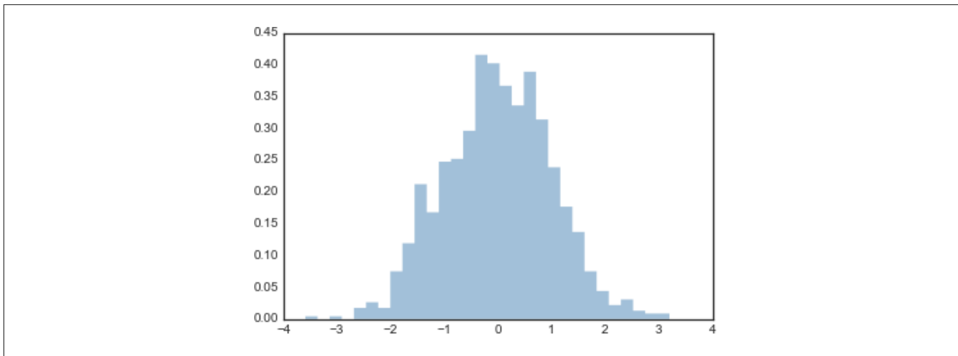


Figure . A customized histogram

The plt.hist docstring has more information on other customization options avail- able. I find this combination of histtype='stepfilled' along with some transpar- ency alpha to be very useful when comparing histograms of several distributions:

```
In[4]: x1 = np.random.normal(0, 0.8, 1000)
       x2 = np.random.normal(-2, 1, 1000)
       x3 = np.random.normal(3, 2, 1000)

       kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True,
       bins=40)plt.hist(x1, **kwargs)
       plt.hist(x2, **kwargs)
       plt.hist(x3, **kwargs);
```

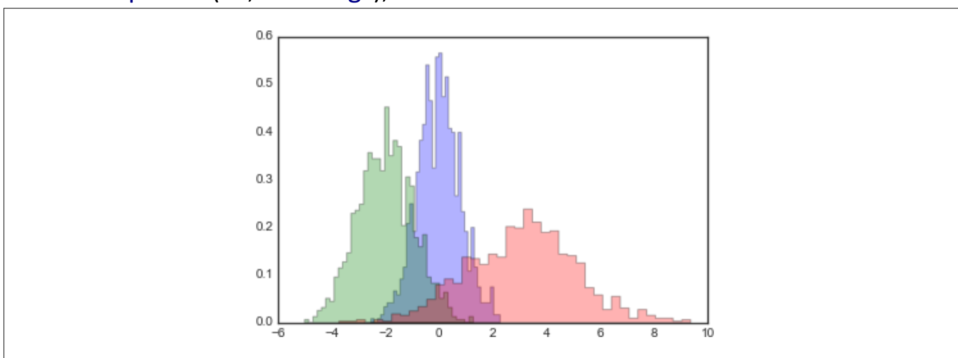


Figure Over-plotting multiple histograms

If you would like to simply compute the histogram (that is, count the number ofpoints in a given bin) and not display it, the np.histogram()function is available:

```
In[5]: counts, bin_edges = np.histogram(data, bins=5)
        print(counts)
[ 12 190 468 301
29]
```

Day-03: Two-Dimensional Histograms and Binnings

Just as we create histograms in one dimension by dividing the number line into bins, we can also create histograms in two dimensions by dividing points among two- dimensional bins. We'll take a brief look at several ways to do this here. We'll start bydefining some data—an x and y array drawn from a multivariate Gaussian distribution:

```
In[6]: mean = [0, 0]
        cov = [[1, 1], [1, 2]]
        x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

plt.hist2d: Two-dimensional histogram

One straightforward way to plot a two-dimensional histogram is to use Matplotlib's plt.hist2dfunction (Figure):

```
In[12]: plt.hist2d(x, y, bins=30,
         cmap='Blues')cb = plt.colorbar()
         cb.set_label('counts in bin')
```

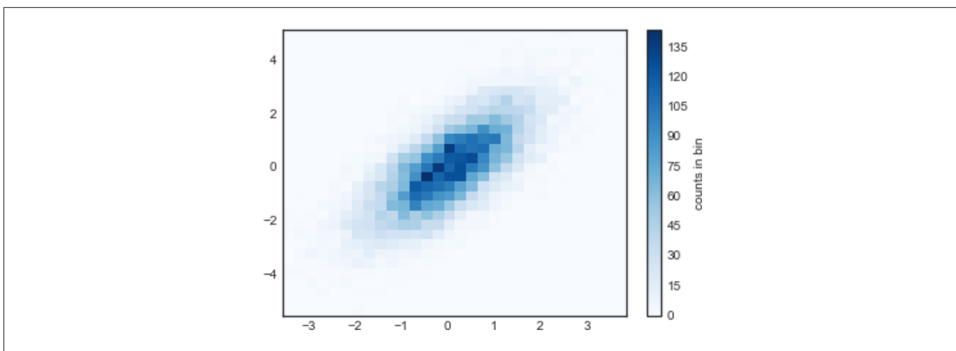


Figure . A two-dimensional histogram with plt.hist2d

Just as with plt.hist, plt.hist2dhas a number of extra options to fine-tune the plot and the binning, which are nicely outlined in the function docstring. Further, just

as plt.hist has a counterpart in np.histogram, plt.hist2d has a counterpart in np.histogram2d, which can be used as follows:

```
In[8]: counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

For the generalization of this histogram binning in dimensions higher than two, see the np.histogramddfunction.

plt.hexbin: Hexagonal binnings

The two-dimensional histogram creates a tessellation of squares across the axes. Another natural shape for such a tessellation is the regular hexagon. For this purpose,Matplotlib provides the plt.hexbin routine, which represents a two-dimensional dataset binned within a grid of hexagons.

```
In[9]: plt.hexbin(x, y, gridsize=30, cmap='Blues')
       cb = plt.colorbar(label='count in bin')
```

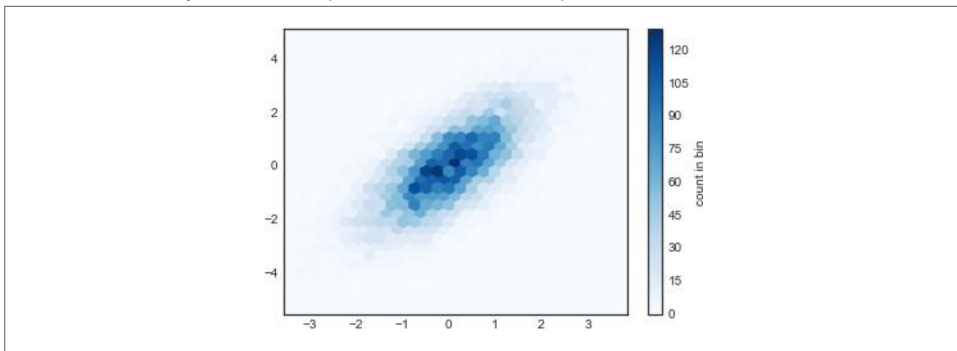


Figure 4-39. A two-dimensional histogram with plt.hexbin

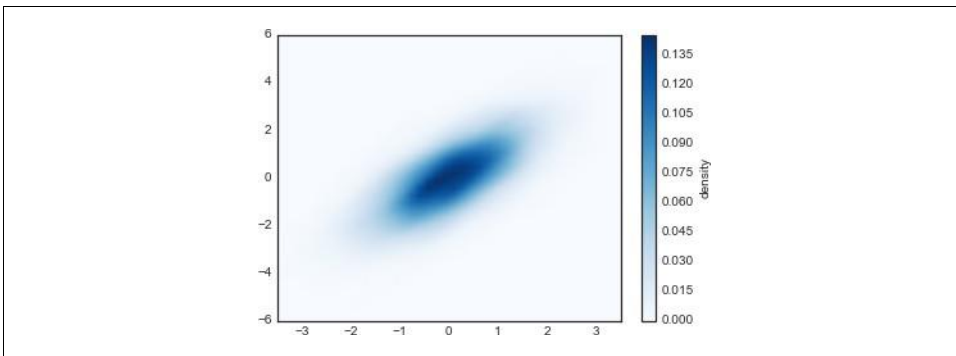
plt.hexbin has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

Kernel density estimation

Another common method of evaluating densities in multiple dimensions is *kernel density estimation* (KDE). We'll simply mention that KDE can bethought of as a way to "smear out" the points in space and add up the result to obtain a smooth function. One extremely quick and simple KDE implementation exists in the scipy.stats package. Here is a quick example of using the KDE on this data:

```
In[10]: from scipy.stats import gaussian_kde

        # fit an array of size [Ndim, Nsamples]
```

```
data = np.vstack([x, y])
kde =
gaussian_kde(data)


# evaluate on a regular grid
xgrid = np.linspace(-3.5, 3.5, 40)
ygrid = np.linspace(-6, 6, 40)
Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))
# Plot the result as an image
plt.imshow(Z.reshape(Xgrid.shape),

            origin='lower',
            aspect='auto',extent=[-3.5,
            3.5, -6, 6],

            cmap='Blue
s')       cb =
```




```
plt.colorbar()
cb.set_label("density")
```

Figure . A kernel density representation of a distribution

KDE has a smoothing length that effectively slides the knob between detail and smoothness (one example of the ubiquitous bias–variance trade-off). The literatureon choosing an appropriate smoothing length is vast: gaussian_kde uses a rule of thumb to attempt to find a nearly optimal smoothing length for the input data.

Other KDE implementations are available within the SciPy ecosystem, each with its own various strengths and weaknesses; see, for example, sklearn.neighbors.KernelDensity and

statsmodels.nonparametric.kernel_density.KDEMultivariate. Forvisualizations based on KDE, using Matplotlib tends to be overly verbose. The Sea- born library, discussed in "Visualization with Seaborn" on page 311, provides a muchmore terse API for creating KDE-based visualizations.

Customizing Plot Legends

Plot legends give meaning to a visualization, assigning labels to the various plot elements. We previously saw how to create a simple legend; here we'll take a look at customizing the placement and aesthetics of the legend in Matplotlib.

The simplest legend can be created with the plt.legend() command, which automatically creates a legend for any labeled plot elements (Figure 4-41):

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
In[2]: %matplotlib inline
       import numpy as np
In[3]: x = np.linspace(0, 10, 1000)
       fig, ax = plt.subplots()
       ax.plot(x, np.sin(x), '-b', label='Sine')
       ax.plot(x, np.cos(x), '--r', label='Cosine')
       ax.axis('equal')
       leg = ax.legend();
```

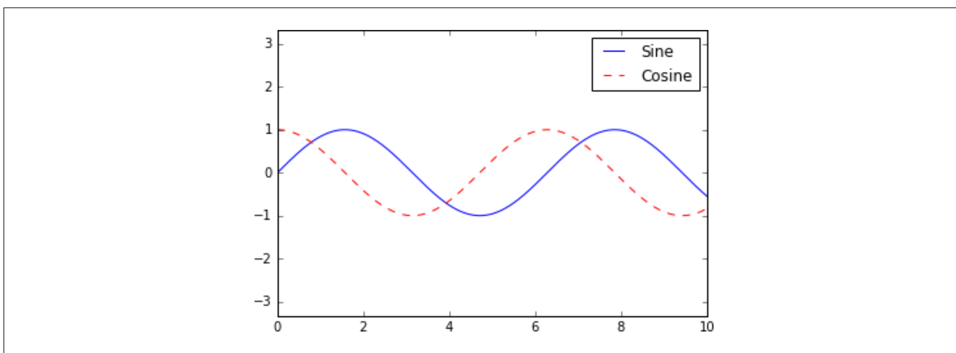


Figure. A default plot legend

But there are many ways we might want to customize such a legend. For example, wecan specify the location and turn off the frame .

```
In[4]: ax.legend(loc='upper left',
         frameon=False)fig
```

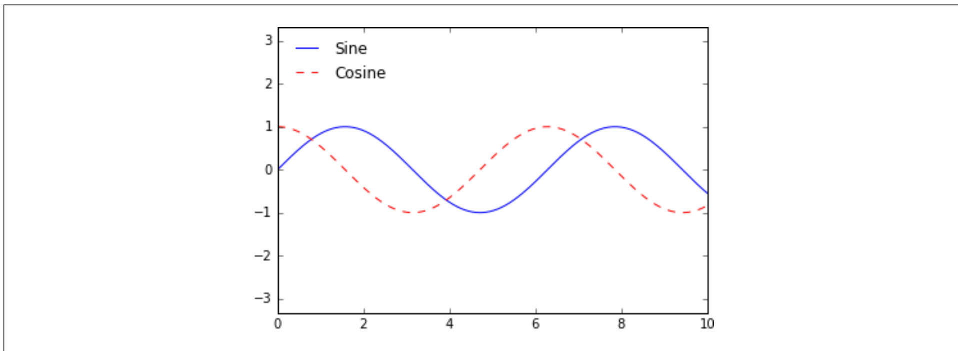


Figure . A customized plot legend

We can use the ncol command to specify the number of columns in the legend

```
In[5]: ax.legend(frameon=False, loc='lower center',
         ncol=2)fig
```

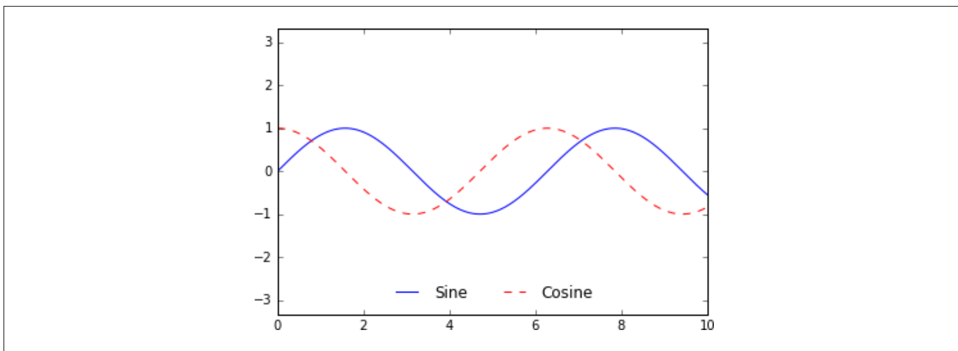


Figure . A two-column plot legend

We can use a rounded box (fancybox) or add a shadow, change the transparency (alpha value) of the frame, or change the padding around the text.

```
In[6]: ax.legend(fancybox=True, framealpha=1, shadow=True,
        borderpad=1)fig
```

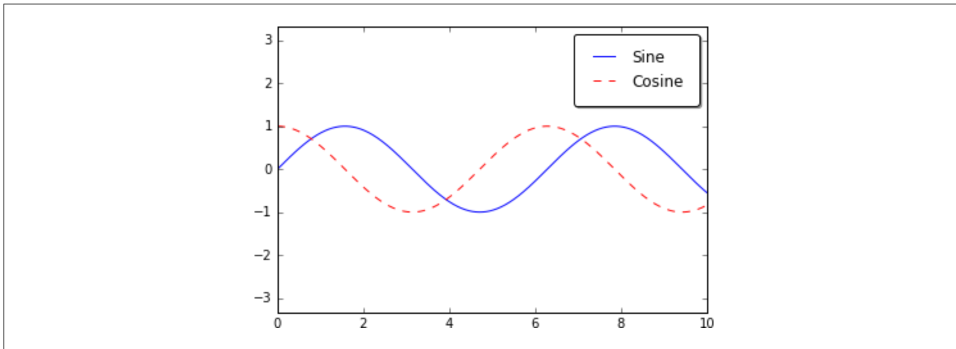


Figure . A fancybox plot legend

For more information on available legend options, see the plt.legenddocstring.

Choosing Elements for the Legend

As we've already seen, the legend includes all labeled elements by default. If this is not what is desired, we can fine-tune which elements and labels appear in the legend by using the objects returned by plot commands. The plt.plot() command is able to create multiple lines at once, and returns a list of created line instances. Passing any ofthese to plt.legend() will tell it which to identify, along with the labels we'd like to specify.

```
In[7]: y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
        lines = plt.plot(x, y)

        # lines is a list of plt.Line2D instances

        plt.legend(lines[:2], ['first', 'second']);
```

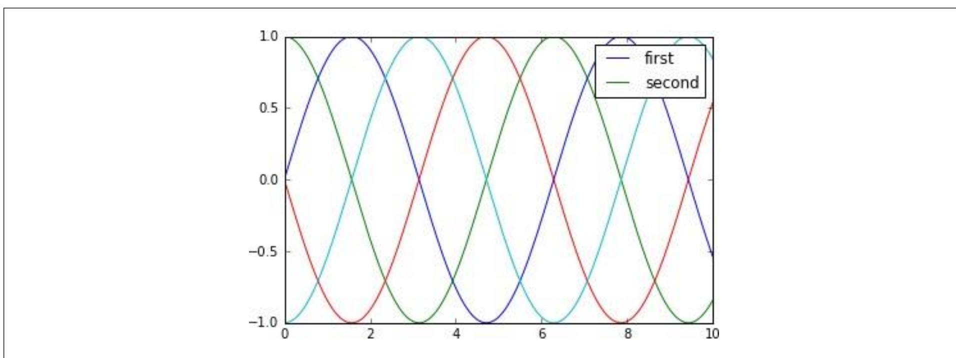
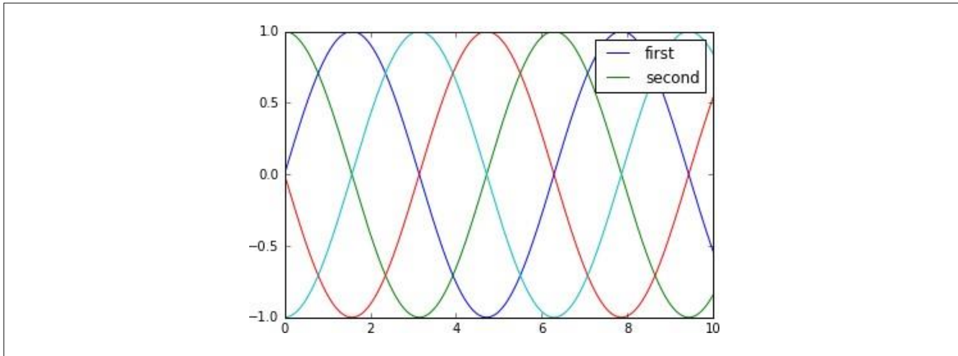


Figure . Customization of legend elements

I generally find in practice that it is clearer to use the first method, applying labels tothe

plot elements you'd like to show on the legend.

```
In[8]: plt.plot(x, y[:, 0], label='first')
       plt.plot(x, y[:, 1], label='second')
       plt.plot(x, y[:, 2:])
       plt.legend(framealpha=1,
```




```
              frameon=True);
```

Figure 4-46. Alternative method of customizing legend elements

Notice that by default, the legend ignores all elements without a labelattribute set.

Legend for Size of Points

Sometimes the legend defaults are not sufficient for the given visualization. For exam-ple, perhaps you're using the size of points to mark certain features of the data, and want to create a legend reflecting this. Here is an example where we'll use the size ofpoints to indicate populations of California cities. We'd like a legend that specifies the

scale of the sizes of the points, and we'll accomplish this by plotting some labeled datawith no entries.

```
In[9]: import pandas as pd
       cities = pd.read_csv('data/california_cities.csv')

       # Extract the data we're interested in
       lat, lon = cities['latd'], cities['longd']
       population, area = cities['population_total'], cities['area_total_km2']

       # Scatter the points, using size and color but no label
       plt.scatter(lon, lat, label=None,
```

```
                    c=np.log10(population),
                    cmap='viridis',s=area, linewidth=0,
                    alpha=0.5)

plt.axis(aspect='equal')
plt.xlabel('longitude') plt.ylabel('latitude')
plt.colorbar(label='log$_{10}$(population
)')plt.clim(3, 7)


# Here we create a legend:

# we'll plot empty lists with the desired size and label

for area in [100, 300, 500]:

    plt.scatter([], [], c='k', alpha=0.3, s=area,
                label=str(area) + ' km$^2$')

plt.legend(scatterpoints=1,
           frameon=False, labelspacing=1,
           title='City Area')


plt.title('California Cities: Area and Population');
```

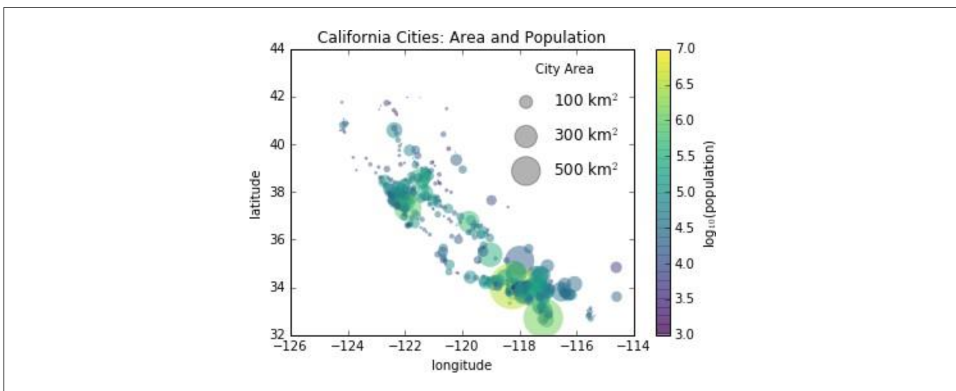


Figure . Location, geographic size, and population of California cities

The legend will always reference some object that is on the plot, so if we'd like to dis-play a particular shape we need to plot it. In this case, the objects we want (gray cir- cles) are not on the plot, so we fake them by plotting empty lists. Notice too that the legend only lists plot elements that have a label specified.

By plotting empty lists, we create labeled plot objects that are picked up by the legend,and now our legend tells us some useful information. This strategy can be useful for creating

more sophisticated visualizations.

Finally, note that for geographic data like this, it would be clearer if we could show state boundaries or other map-specific elements. For this, an excellent choice of tool is Matplotlib's Basemap add-on toolkit.

Multiple Legends

Sometimes when designing a plot you'd like to add multiple legends to the same axes. Unfortunately, Matplotlib does not make this easy: via the standard legend interface,it is only possible to create a single legend for the entire plot. If you try to create a second legend using plt.legend() or ax.legend(), it will simply override the firstone. We can work around this by creating a new legend artist from scratch, and then using the lower-level ax.add_artist() method to manually add the second artist to the plot (Figure):

```
In[10]: fig, ax = plt.subplots()

        lines = []
        styles = ['-', '--', '-.', ':']
        x = np.linspace(0, 10, 1000)

        for i in range(4):
            lines += ax.plot(x, np.sin(x - i * np.pi / 2),
                             styles[i], color='black')
        ax.axis('equal')

        # specify the lines and labels of the first legend
        ax.legend(lines[:2], ['line A', 'line B'],
                  loc='upper right',
                  frameon=False)

        # Create the second legend and add the artist manually.
        from matplotlib.legend import Legend
        leg = Legend(ax, lines[2:], ['line C', 'line D'],
                     loc='lower right', frameon=False)
        ax.add_artist(leg);
```

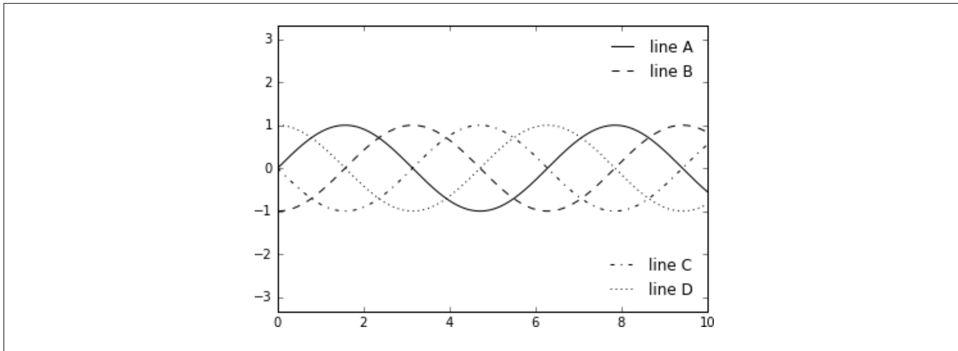


Figure . A split plot legend

This is a peek into the low-level artist objects that compose any Matplotlib plot. If you examine the source code of ax.legend() (recall that you can do this within the IPy- thon notebook using ax.legend??) you'll see that the function simply consists of some logic to create a suitable Legend artist, which is then saved in the legend_ attribute and added to the figure when the plot is drawn.

Customizing Colorbars

Plot legends identify discrete labels of discrete points. For continuous labels based onthe color of points, lines, or regions, a labeled colorbar can be a great tool. In Mat- plotlib, a colorbar is a separate axes that can provide a key for the meaning of colors in a plot. Because the book is printed in black and white, this section has an accompa-nying online appendix where you can view the figures in full color. We'll start by setting up the note- book for plotting and importing the functions we will use:

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
In[2]: %matplotlib inline
       import numpy as np
```

As we have seen several times throughout this section, the simplest colorbar can be created with the plt.colorbarfunction.

```
In[3]: x = np.linspace(0, 10, 1000)
       I = np.sin(x) * np.cos(x[:, np.newaxis])


       plt.imshow(I)
       plt.colorbar();
```

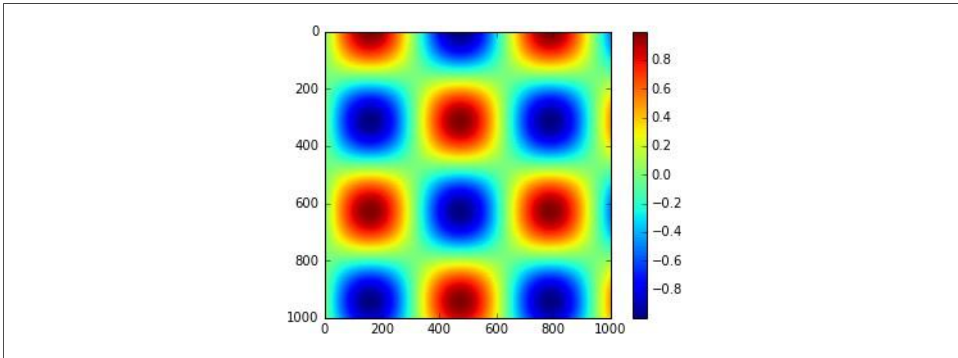


Figure . A simple colorbar legend

We'll now discuss a few ideas for customizing these colorbars and using them effectively in various situations.

Customizing Colorbars

We can specify the colormap using the cmapargument to the plotting function that iscreating the visualization.

```
In[4]: plt.imshow(I, cmap='gray');
```

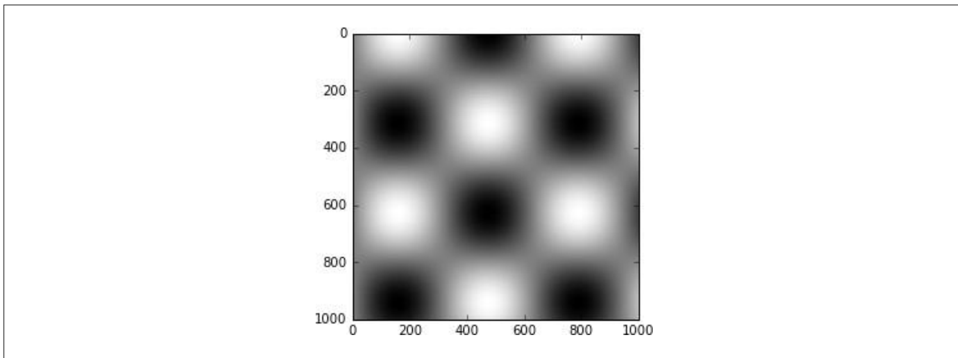


Figure . A grayscale colormap

All the available colormaps are in the plt.cm namespace; using IPython's tab-completion feature will give you a full list of built-in possibilities:

```
plt.cm.<TAB>
```

But being *able* to choose a colormap is just the first step: more important is how to *decide* among the possibilities! The choice turns out to be much more subtle than you might initially expect.

Choosing the colormap

A full treatment of color choice within visualization is beyond the scope of this book,but for entertaining reading on this subject and others, see the article "Ten Simple Rules for Better Figures". Matplotlib's online documentation also has an interesting discussion of colormap choice.

Broadly, you should be aware of three different categories of colormaps:

Sequential colormaps

These consist of one continuous sequence of colors (e.g., binaryor viridis).

Divergent colormaps

These usually contain two distinct colors, which show positive and negative devi-ations from a mean (e.g., RdBuor PuOr).

Qualitative colormaps

These mix colors with no particular sequence (e.g., rainbowor jet).

The jet colormap, which was the default in Matplotlib prior to version 2.0, is an example of a qualitative colormap. Its status as the default was quite unfortunate, because qualitative maps are often a poor choice for representing quantitative data. Among the problems is the fact that qualitative maps usually do not display any uni- form progression in brightness as the scale increases.

We can see this by converting the jetcolorbar into black and white (Figure):

```
In[5]:
from matplotlib.colors import LinearSegmentedColormap


def grayscale_cmap(cmap):
    """Return a grayscale version of the given colormap"""
    cmap =
    plt.cm.get_cmap(cmap) colors
    = cmap(np.arange(cmap.N))


    # convert RGBA to perceived grayscale
    luminance#  cf.
    http://alienryderflex.com/hsp.html
    RGB_weight = [0.299, 0.587, 0.114]
```

```
luminance = np.sqrt(np.dot(colors[:, :3] ** 2,
RGB_weight))colors[:, :3] = luminance[:, np.newaxis]


return LinearSegmentedColormap.from_list(cmap.name + "_gray", colors,
cmap.N)

def view_colormap(cmap):
    """Plot a colormap with its grayscale equivalent"""

    cmap =
    plt.cm.get_cmap(cmap) colors
    = cmap(np.arange(cmap.N))

    cmap = grayscale_cmap(cmap)
    grayscale =
    cmap(np.arange(cmap.N))

    fig, ax = plt.subplots(2, figsize=(6, 2),

                           subplot_kw=dict(xticks=[],
    yticks=[]))ax[0].imshow([colors], extent=[0, 10, 0, 1])

    ax[1].imshow([grayscale], extent=[0, 10, 0, 1])

In[6]: view_colormap('jet')
```

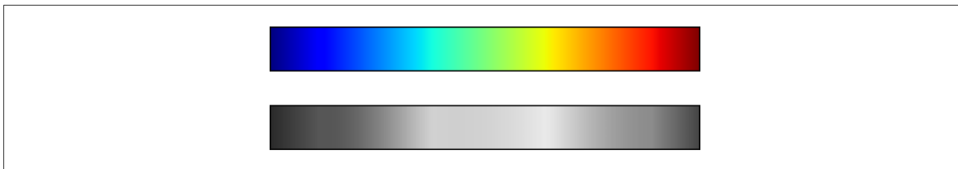


Figure . The jet colormap and its uneven luminance scale

Notice the bright stripes in the grayscale image. Even in full color, this uneven bright-ness means that the eye will be drawn to certain portions of the color range, which will potentially emphasize unimportant parts of the dataset. It's better to use a color- map such as viridis (the default as of Matplotlib 2.0), which is specifically construc-ted to have an even brightness variation across the range. Thus, it not only plays well with our color perception, but also will translate well to grayscale printing.

```
In[7]: view_colormap('viridis')
```

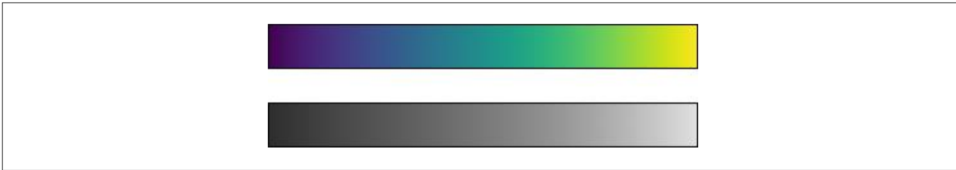


Figure . The viridis colormap and its even luminance scale

If you favor rainbow schemes, another good option for continuous data is the cubehelixcolormap.

```
In[8]: view_colormap('cubehelix')
```

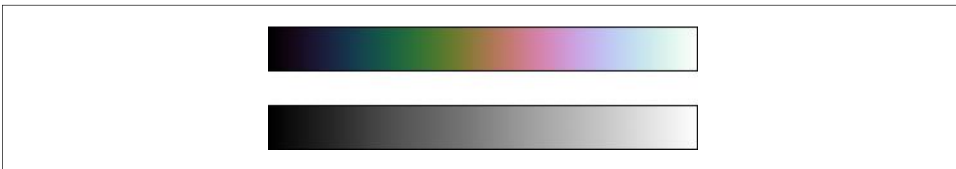


Figure. The cubehelix colormap and its luminance

For other situations, such as showing positive and negative deviations from somemean, dual-color colorbars such as RdBu(short for *Red-Blue*) can be useful. However,

as you can see in Figure , it's important to note that the positive-negative information will be lost upon translation to grayscale!

```
In[9]: view_colormap('RdBu')
```

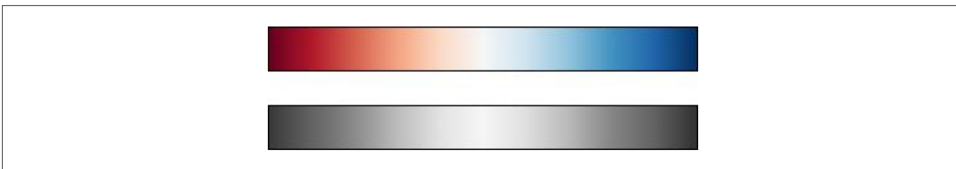


Figure . The RdBu (Red-Blue) colormap and its luminance

We'll see examples of using some of these color maps as we continue.

There are a large number of colormaps available in Matplotlib; to see a list of them, you can use IPython to explore the plt.cm submodule. For a more principled approach to colors in Python, you can refer to the tools and documentation withinthe Seaborn library (see "Visualization with Seaborn").

Color limits and extensions

Matplotlib allows for a large range of colorbar customization. The colorbar itself is simply an instance of plt.Axes, so all of the axes and tick formatting tricks we've learned are applicable. The colorbar has some interesting flexibility; for example, we can narrow the color limits and indicate the out-of-bounds values with a triangular arrow at the top and bottom by setting the extend property. This might come in handy, for example, if you're displaying an image that is subject to noise (Figure):

```
In[10]: # make noise in 1% of the image pixels
        speckles = (np.random.random(I.shape) < 0.01)
        I[speckles] = np.random.normal(0, 3,

        np.count_nonzero(speckles))plt.figure(figsize=(10, 3.5))


        plt.subplot(1, 2, 1)
        plt.imshow(I,
        cmap='RdBu')
        plt.colorbar()


        plt.subplot(1, 2, 2)
        plt.imshow(I,
        cmap='RdBu')
        plt.colorbar(extend='bot
        h')plt.clim(-1, 1);
```

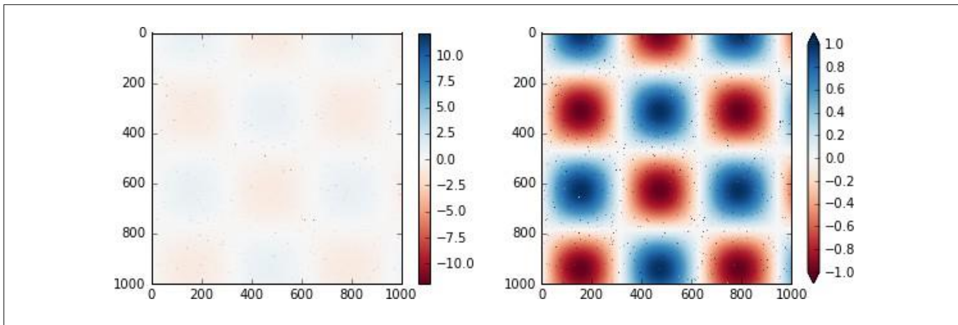


Figure . Specifying colormap extensions

Notice that in the left panel, the default color limits respond to the noisy pixels, and the range of the noise completely washes out the pattern we are interested in. In the right panel, we manually set the color limits, and add extensions to indicate values that are above or below those limits. The result is a much more useful visualization ofour data.

Discrete colorbars

Colormaps are by default continuous, but sometimes you'd like to represent discrete values. The easiest way to do this is to use the plt.cm.get_cmap()function, and passthe name of a suitable colormap along with the number of desired bins (Figure 4-56):

```
In[11]: plt.imshow(I, cmap=plt.cm.get_cmap('Blues',
        6))plt.colorbar()
        plt.clim(-1, 1);
```

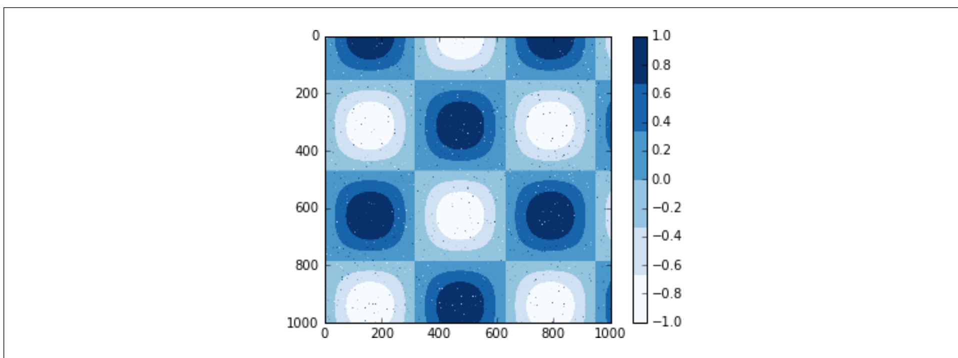


Figure . A discretized colormap

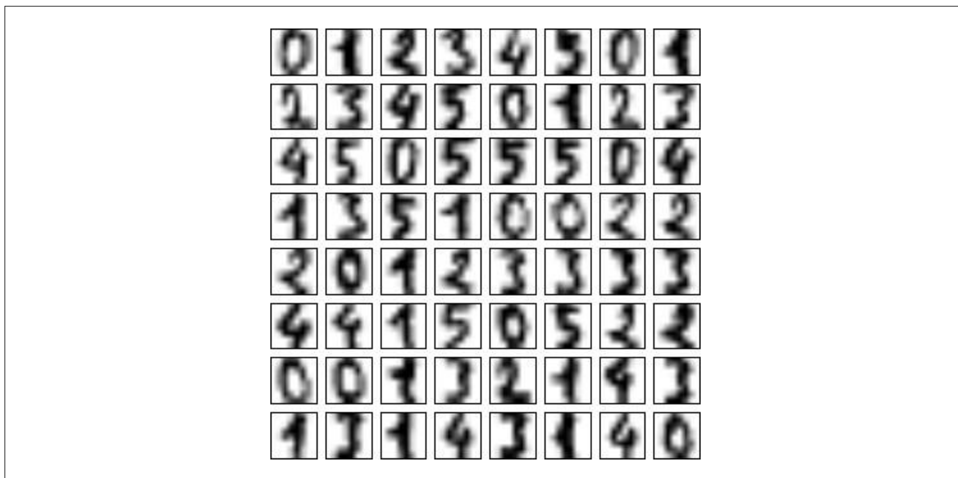
The discrete version of a colormap can be used just like any other colormap.

Handwritten Digits

For an example of where this might be useful, let's look at an interesting visualizationof some handwritten digits data. This data is included in Scikit-Learn, and consists ofnearly 2,000 8×8 thumbnails showing various handwritten digits.

For now, let's start by downloading the digits data and visualizing several of the exam-ple images with plt.imshow()(Figure 4-57):

```
In[12]: # load images of the digits 0 through 5 and visualize several of them

        from sklearn.datasets import
        load_digitsdigits =
        load_digits(n_class=6)


        fig, ax = plt.subplots(8, 8, figsize=(6, 6))

        for i, axi in enumerate(ax.flat):
            axi.imshow(digits.images[i],
```



```
            cmap='binary')axi.set(xticks=[], yticks=[])
```

Figure . Sample of handwritten digit data

Because each digit is defined by the hue of its 64 pixels, we can consider each digit tobe a point lying in 64-dimensional space: each dimension represents the brightness ofone pixel. But visualizing relationships in such high-dimensional spaces can beextremely difficult. One way to approach this is to use a *dimensionality reduction* technique such as manifold learning to reduce the dimensionality of the data while maintaining the relationships of interest. Dimensionality reduction is an example of unsupervised machine learning.

Deferring the discussion of these details, let's take a look at a two-dimensional mani-fold learning projection of this digits data.

```
In[13]: # project the digits into 2 dimensions using IsoMap

        from sklearn.manifold import
        Isomapiso =
        Isomap(n_components=2)

        projection = iso.fit_transform(digits.data)
```

We'll use our discrete colormap to view the results, setting the ticks and clim to improve the aesthetics of the resulting colorbar.

```
In[14]: # plot the results

        plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,

                    c=digits.target, cmap=plt.cm.get_cmap('cubehelix',
        6))plt.colorbar(ticks=range(6), label='digit value')

        plt.clim(-0.5, 5.5)
```

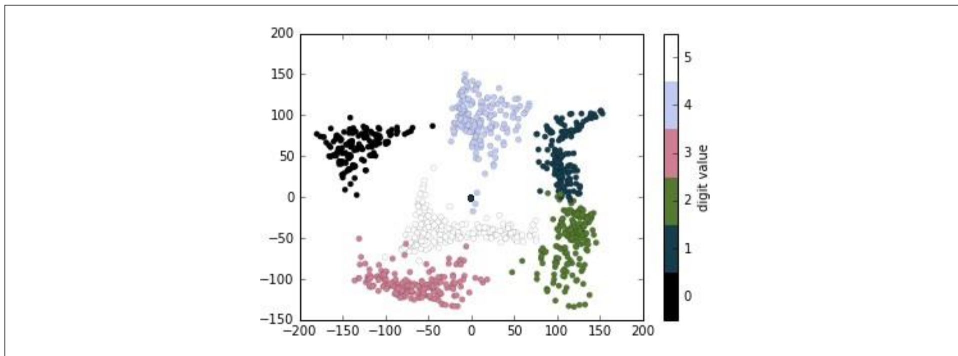


Figure. Manifold embedding of handwritten digit pixels

The projection also gives us some interesting insights on the relationships within the dataset: for example, the ranges of 5 and 3 nearly overlap in this projection, indicating that some handwritten fives and threes are difficult to distinguish, and therefore more likely to be confused by an automated classification algorithm. Other values, like 0 and 1, are more distantly separated, and therefore much less likely to be con- fused. This observation agrees with our intuition, because 5 and 3 look much more similar than do 0 and 1.

Multiple Subplots

Sometimes it is helpful to compare different views of data side by side. To this end,

Matplotlib has the concept of *subplots*: groups of smaller axes that can exist together within a single figure. These subplots might be insets, grids of plots, or other more complicated layouts. In this section, we'll explore four routines for creating subplots in Matplotlib. We'll start by setting up the notebook for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline

       import matplotlib.pyplot as
       pltplt.style.use('seaborn-
       white') import numpy as np
```

plt.axes: Subplots by Hand

The most basic method of creating an axes is to use the plt.axes function. As we've seen previously, by default this creates a standard axes object that fills the entire fig- ure. plt.axes also takes an optional argument that is a list of four numbers in the figure coordinate system. These numbers represent [*bottom*, *left*, *width*, *height*]in the figure coordinate system, which ranges from 0 at the bottom left of thefigure to 1 at the top right of the figure.

For example, we might create an inset axes at the top-right corner of another axes by setting the *x* and *y* position to 0.65 (that is, starting at 65% of the width and 65% of the height of the figure) and the *x* and *y* extents to 0.2 (that is, the size of the axes is 20% of the width and 20% of the height of the figure). Figure 4-59 shows the result ofthis code:

```
In[2]: ax1 = plt.axes()  # standard axes
       ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

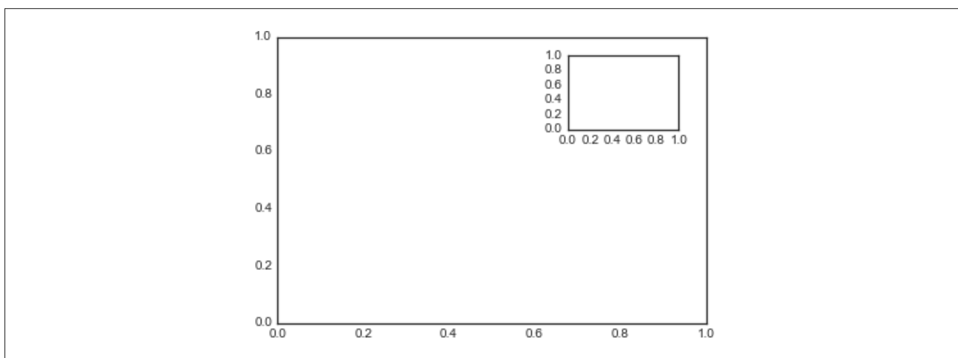


Figure . Example of an inset axes

The equivalent of this command within the object-oriented interface is

fig.add_axes(). Let's use this to create two vertically stacked axes (Figure 4-60):

```
In[3]: fig = plt.figure()
       ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
                          xticklabels=[], ylim=(-1.2, 1.2))
       ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
                          ylim=(-1.2, 1.2))

       x = np.linspace(0, 10)
       ax1.plot(np.sin(x))
       ax2.plot(np.cos(x));
```

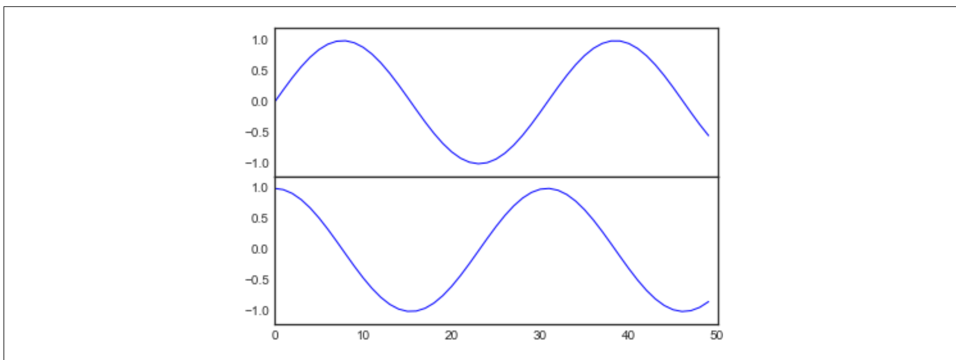


Figure . Vertically stacked axes example

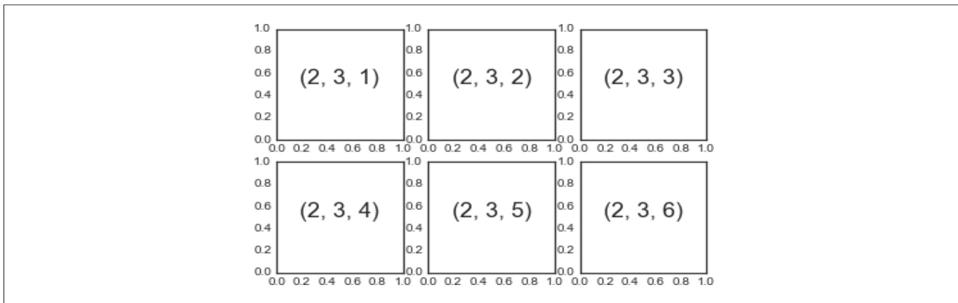
We now have two axes (the top with no tick labels) that are just touching: the bottomof the upper panel (at position 0.5) matches the top of the lower panel (at position 0.1 + 0.4).

plt.subplot: Simple Grids of Subplots

Aligned columns or rows of subplots are a common enough need that Matplotlib has several convenience routines that make them easy to create. The lowest level of theseis plt.subplot(), which creates a single subplot within a grid. As you can see, this command takes three integer arguments—the number of rows, the number of col- umns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right .

```
In[4]: for i in range(1, 7):
           plt.subplot(2, 3, i)
```

```
plt.text(0.5, 0.5, str((2, 3, i)),
             fontsize=18,
```




```
             ha='center')
```

Figure. A plt.subplot() example

The command plt.subplots_adjust can be used to adjust the spacing betweenthese plots. The following cod uses theequivalent object-oriented command, fig.add_subplot():

```
In[5]: fig = plt.figure()
       fig.subplots_adjust(hspace=0.4,
       wspace=0.4)for i in range(1, 7):

           ax = fig.add_subplot(2, 3, i)
           ax.text(0.5, 0.5, str((2, 3, i)),

                  fontsize=18, ha='center')
```

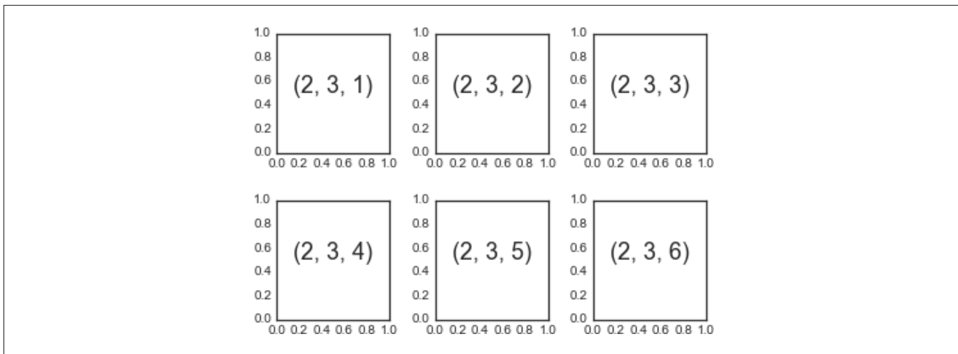


Figure plt.subplot() with adjusted margins

We've used the hspaceand wspacearguments of plt.subplots_adjust, which spec-ify the spacing along the height and width of the figure, in units of the subplot size (inthis case, the space is 40% of the subplot width and height).

plt.subplots: The Whole Grid in One Go

The approach just described can become quite tedious when you're creating a large grid of subplots, especially if you'd like to hide the x- and y-axis labels on the inner plots. For this purpose, plt.subplots()is the easier tool to use (note the sat the endof subplots). Rather than creating a single subplot, this function creates a full grid ofsubplots in a single line, returning them in a NumPy array. The arguments are the number of rows and number of columns, along with optional keywords sharex and sharey, which allow you to specify the relationships between different axes.

Here we'll create a 2×3 grid of subplots, where all axes in the same row share their y-axis scale, and all axes in the same column share their x-axis scale (Figure):

```
In[6]: fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```

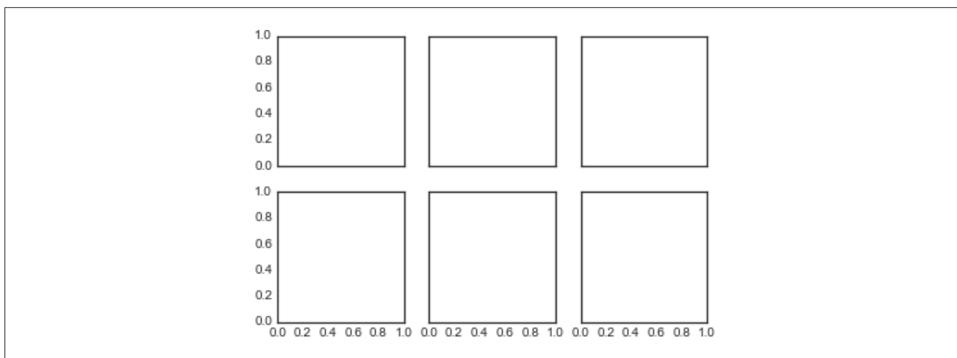


Figure .Shared x and y axis in plt.subplots()

Note that by specifying sharex and sharey, we've automatically removed inner labelson the grid to make the plot cleaner. The resulting grid of axes instances is returned within a NumPy array, allowing for convenient specification of the desired axes usingstandard array indexing notation.

```
In[7]: # axes are in a two-dimensional array, indexed by [row, col]
        for i in range(2):
            for j in range(3):
                ax[i, j].text(0.5, 0.5, str((i, j)),
                              fontsize=18, ha='center')
```

```
fig
```

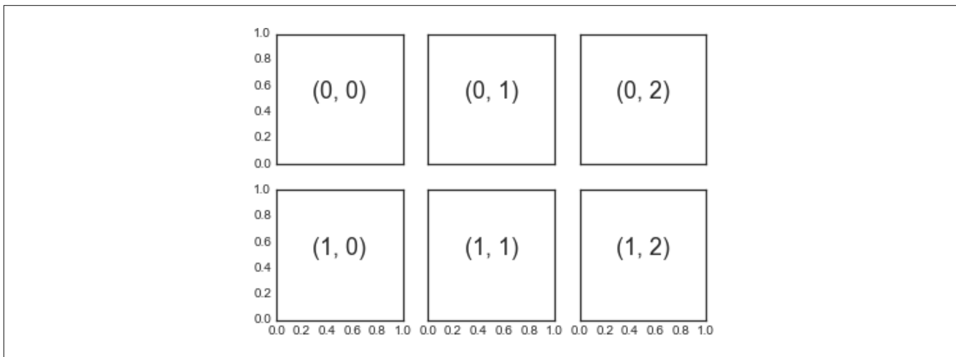


Figure . Identifying plots in a subplot grid

In comparison to plt.subplot(), plt.subplots()is more consistent with Python's conventional 0-based indexing.

plt.GridSpec: More Complicated Arrangements

To go beyond a regular grid to subplots that span multiple rows and columns, plt.GridSpec()is the best tool. The plt.GridSpec()object does not create a plot by itself; it is simply a convenient interface that is recognized by the plt.subplot() command. For example, a gridspec for a grid of two rows and three columns with some specified width and height space looks like this:

```
In[8]: grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

From this we can specify subplot locations and extents using the familiar Python slic-ing syntax (Figure 4-65):

```
In[9]: plt.subplot(grid[0, 0])
       plt.subplot(grid[0, 1:])
       plt.subplot(grid[1, :2])
```

```
plt.subplot(grid[1, 2]);
```

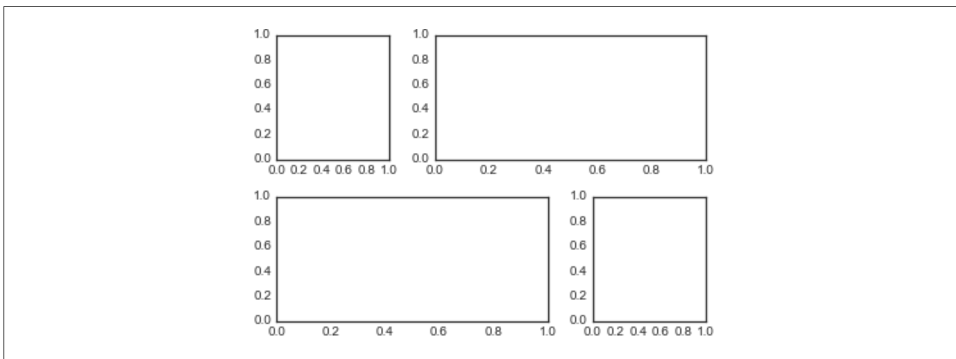


Figure . Irregular subplots with plt.GridSpec

This type of flexible grid alignment has a wide range of uses. I most often use it whencreating multi-axes histogram plots like the one shown here (Figure 4-66):

```
In[10]: # Create some normally distributed data
        mean = [0, 0]
        cov = [[1, 1], [1, 2]]
        x, y = np.random.multivariate_normal(mean, cov, 3000).T

      # Set up the axes with gridspec
        fig = plt.figure(figsize=(6, 6))
        grid = plt.GridSpec(4, 4, hspace=0.2,
        wspace=0.2)main_ax = fig.add_subplot(grid[:-
        1, 1:])
        y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[],
        sharey=main_ax)x_hist = fig.add_subplot(grid[-1, 1:],
        yticklabels=[], sharex=main_ax)

        # scatter points on the main axes
        main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

        # histogram on the attached axes
        x_hist.hist(x, 40, histtype='stepfilled',
                    orientation='vertical', color='gray')
        x_hist.invert_yaxis()
```

```
y_hist.hist(y, 40, histtype='stepfilled',
            orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```

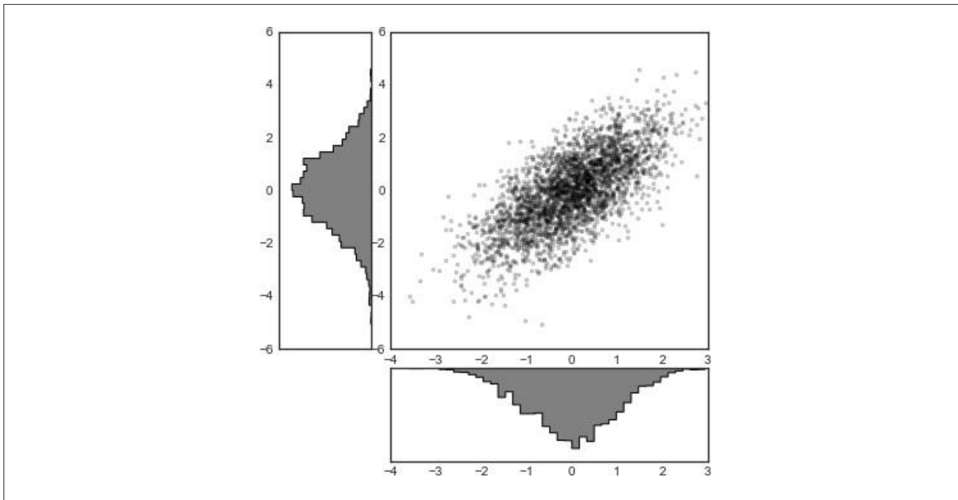


Figure 4-66. Visualizing multidimensional distributions with plt.GridSpec

This type of distribution plotted alongside its margins is common enough that it has its own plotting API in the Seaborn package; see "Visualization with Seaborn" for more details.

Day-04: Text and Annotation

Creating a good visualization involves guiding the reader so that the figure tells a story. In some cases, this story can be told in an entirely visual manner, without the need for added text, but in others, small textual cues and labels are necessary. Perhapsthe most basic types of annotations you will use are axes labels and titles, but the options go beyond this. Let's take a look at some data and how we might visualize andannotate it to help convey interesting information. We'll start by setting up the note- book for plotting and importing the functions we will use:

```
In[1]: %matplotlib inline

       import matplotlib.pyplot as plt
       import matplotlib as mpl
       plt.style.use('seaborn-
       whitegrid')import numpy as np

       import pandas as pd
```

Example: Effect of Holidays on US Births

Let's return to some data we worked with earlier in "Example: Birthrate Data" on page174, where we generated a plot of average births over the course of the calendar year; as already mentioned, this data can be downloaded at *https://raw.githubusercon tent.com/jakevdp/data-CDCbirths/master/births.csv*.

We'll start with the same cleaning procedure we used there, and plot the results (Figure 4-67):

```
In[2]:

births = pd.read_csv('births.csv')


quartiles = np.percentile(births['births'], [25, 50, 75]) mu,
sig = quartiles[1], 0.74 * (quartiles[2] - quartiles[0])

births = births.query('(births > @mu - 5 * @sig) & (births < @mu + 5 * @sig)')

births['day'] = births['day'].astype(int)

births.index = pd.to_datetime(10000 *

births.year +

                              100 * births.month +
                              births.day,
                              format='%Y%m%d')

births_by_date = births.pivot_table('births',

                                    [births.index.month,
births.index.day])births_by_date.index = [pd.datetime(2012, month,
day)

                          for (month, day) in  births_by_date.index]
```

```
In[3]: fig, ax = plt.subplots(figsize=(12, 4))
       births_by_date.plot(ax=ax);
```

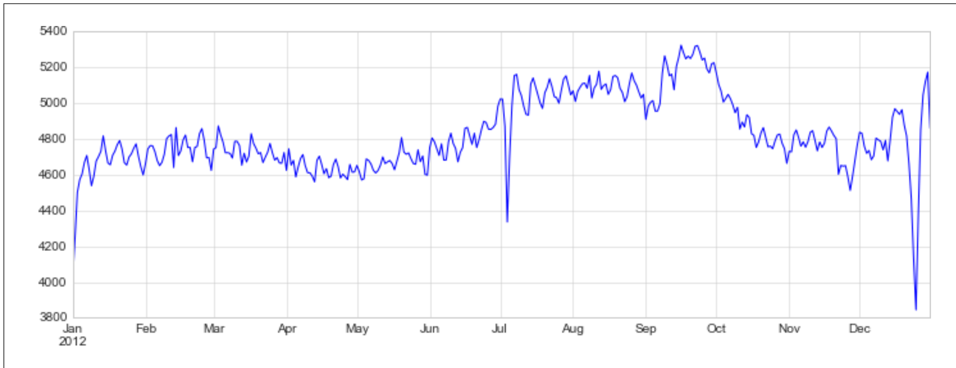
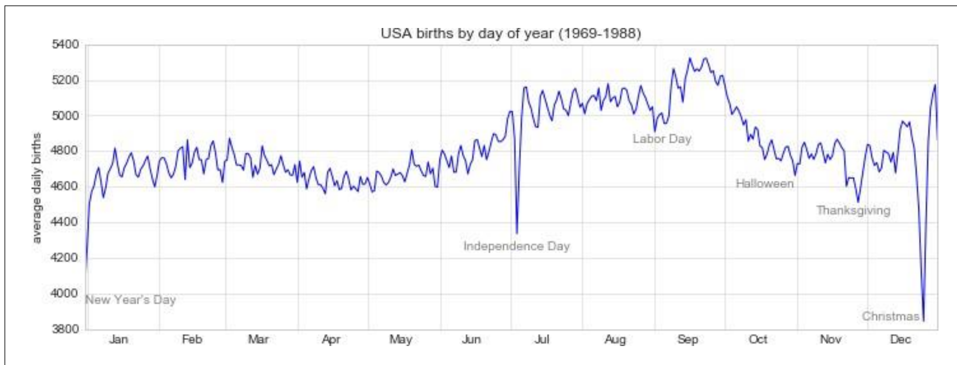


Figure . Average daily births by date

When we're communicating data like this, it is often useful to annotate certain fea- tures of the plot to draw the reader's attention. This can be done manually with the plt.text/ax.text command, which will place text at a particular *x*/*y* value:

```
In[4]: fig, ax = plt.subplots(figsize=(12, 4))
       births_by_date.plot(ax=ax)


       # Add labels to the plot

       style = dict(size=10, color='gray')


       ax.text('2012-1-1', 3950, "New Year's Day", **style)

       ax.text('2012-7-4', 4250, "Independence Day", ha='center',
       **style)ax.text('2012-9-4', 4850, "Labor Day", ha='center',
       **style) ax.text('2012-10-31', 4600, "Halloween", ha='right',
       **style) ax.text('2012-11-25', 4450, "Thanksgiving",
       ha='center', **style) ax.text('2012-12-25', 3850, "Christmas ",
       ha='right', **style)


       # Label the axes

       ax.set(title='USA births by day of year (1969-1988)',
              ylabel='average daily births')
```

```
# Format the x axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymont
hday=15)) ax.xaxis.set_major_formatter(plt.NullFormatter())
```




```
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h')
);
```

Figure . Annotated average daily births by date

The ax.textmethod takes an *x* position, a *y* position, a string, and then optional key-words specifying the color, size, style, alignment, and other properties of the text. Here we used ha='right' and ha='center', where ha is short for *horizonal align- ment*. See the docstring of plt.text() and of mpl.text.Text() for more informationon available options.

Transforms and Text Position

In the previous example, we anchored our text annotations to data locations. Some- times it's preferable to anchor the text to a position on the axes or figure, independentof the data. In Matplotlib, we do this by modifying the *transform*.

Any graphics display framework needs some scheme for translating between coordi- nate systems. For example, a data point at *x*, *y* = 1, 1 needs to somehow be repre- sented at a certain location on the figure, which in turn needs to be represented in pixels on the screen. Mathematically, such coordinate transformations are relatively straightforward, and Matplotlib has a well-developed set of tools that it uses inter- nally to perform them (the tools can be explored in the matplotlib.transforms sub-module).

The average user rarely needs to worry about the details of these transforms, but it is helpful knowledge to have when considering the placement of text on a figure. There are three predefined transforms that can be useful in this situation:

ax.transData

Transform associated with data coordinates

ax.transAxes

Transform associated with the axes (in units of axes dimensions)

fig.transFigure

Transform associated with the figure (in units of figure dimensions)

Here let's look at an example of drawing text at various locations using these trans- forms (Figure):

```
In[5]: fig, ax = plt.subplots(facecolor='lightgray')
       ax.axis([0, 10, 0, 10])

       # transform=ax.transData is the default, but we'll specify it anyway
       ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
       ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
       ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```

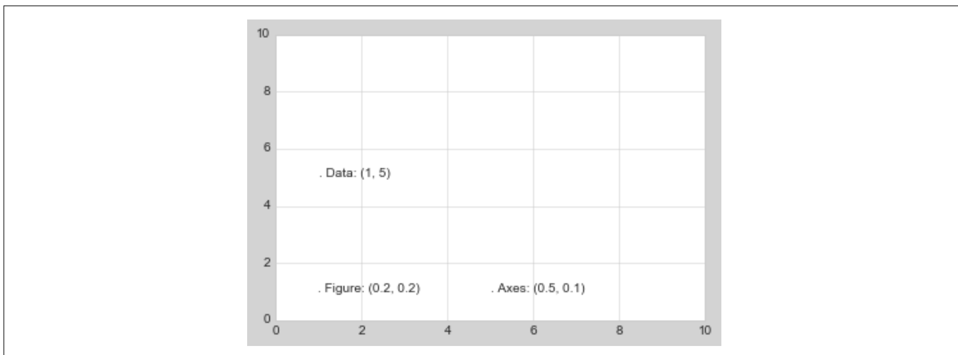


Figure . Comparing Matplotlib's coordinate systems

Note that by default, the text is aligned above and to the left of the specified coordi- nates; here the "." at the beginning of each string will approximately mark the given coordinate location.

The transDatacoordinates give the usual data coordinates associated with the x- andy-axis labels. The transAxes coordinates give the location from the bottom-left cor- ner of the axes (here the white box) as a fraction of the axes size. The transFigure coordinates are similar, but specify the position from the bottom left of the figure (here the gray box) as a fraction of the figure size.

Notice now that if we change the axes limits, it is only the transDatacoordinates thatwill be affected, while the others remain stationary .

```
In[6]: ax.set_xlim(0, 2)

       ax.set_ylim(-6, 6)
       fig
```

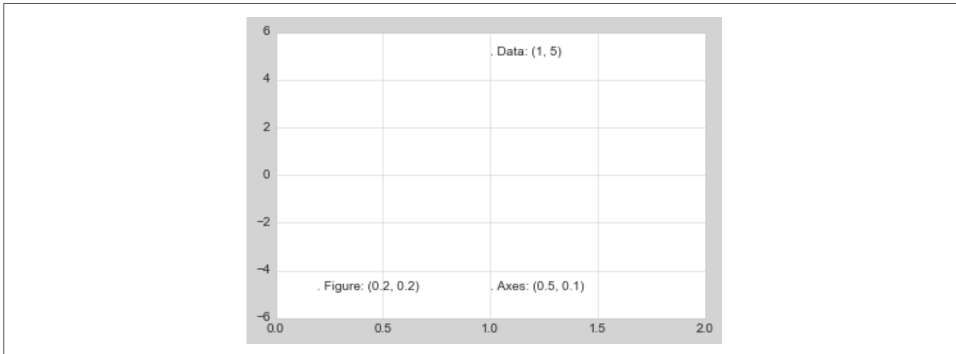


Figure . Comparing Matplotlib's coordinate systems

You can see this behavior more clearly by changing the axes limits interactively; if you are executing this code in a notebook, you can make that happen by changing %mat plotlib inline to %matplotlib notebook and using each plot's menu to interact with the plot.

Arrows and Annotation

Along with tick marks and text, another useful annotation mark is the simple arrow.

Drawing arrows in Matplotlib is often much harder than you might hope. While there is a plt.arrow() function available, I wouldn't suggest using it; the arrows it creates are SVG objects that will be subject to the varying aspect ratio of your plots, and the result is rarely what the user intended. Instead, I'd suggest using the plt.annotate()function. This function creates some text and an arrow, and the arrows can bevery flexibly specified.

Here we'll use annotatewith several of its options:

```
In[7]: %matplotlib inline


       fig, ax = plt.subplots()


       x = np.linspace(0, 20,
       1000)ax.plot(x, np.cos(x))

       ax.axis('equal')
```

```
ax.annotate('local maximum', xy=(6.28, 1), xytext=(10,
            4), arrowprops=dict(facecolor='black',
            shrink=0.05))

ax.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -
            6),arrowprops=dict(arrowstyle="->",
                              connectionstyle="angle3,angleA=0,angleB=-90"));
```

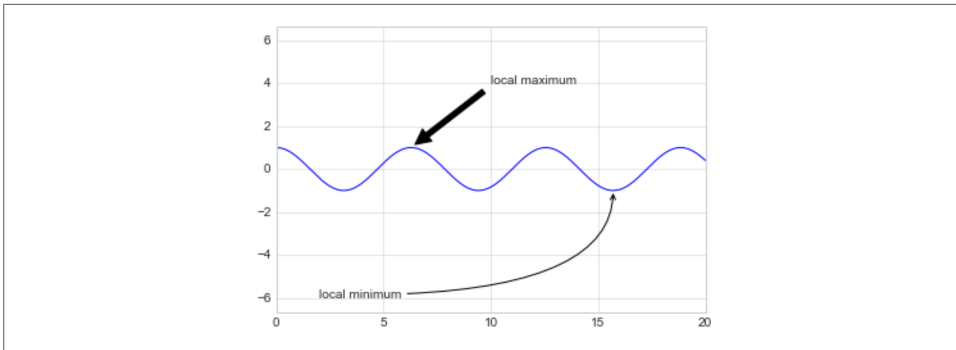


Figure .Annotation examples

The arrow style is controlled through the arrowprops dictionary, which has numer- ous options available. These options are fairly well documented in Matplotlib's online documentation, so rather than repeating them here I'll quickly show some of the pos- sibilities. Let's demonstrate several of the possible options using the birthrate plot from before.

```
In[8]:

fig, ax = plt.subplots(figsize=(12, 4))
births_by_date.plot(ax=ax)

# Add labels to the plot

ax.annotate("New Year's Day", xy=('2012-1-1', 4100),
            xycoords='data',xytext=(50, -30), textcoords='offset
            points', arrowprops=dict(arrowstyle="->",
                              connectionstyle="arc3,rad=-0.2"))
```

```
ax.annotate("Independence Day", xy=('2012-7-4', 4250),
            xycoords='data',bbox=dict(boxstyle="round", fc="none",
            ec="gray"),

            xytext=(10, -40), textcoords='offset points',
            ha='center',arrowprops=dict(arrowstyle="->"))


ax.annotate('Labor Day', xy=('2012-9-4', 4850), xycoords='data',
            ha='center',xytext=(0, -20), textcoords='offset points')

ax.annotate('', xy=('2012-9-1', 4850), xytext=('2012-9-7', 4850),

            xycoords='data', textcoords='data',
            arrowprops={'arrowstyle': '|-
            |,widthA=0.2,widthB=0.2', })


ax.annotate('Halloween', xy=('2012-10-31', 4600),
            xycoords='data',xytext=(-80, -40), textcoords='offset
            points', arrowprops=dict(arrowstyle="fancy",

                            fc="0.6", ec="none",
                            connectionstyle="angle3,angleA=0,angleB=
                            -90"))


ax.annotate('Thanksgiving', xy=('2012-11-25', 4500),
            xycoords='data',xytext=(-120, -60), textcoords='offset
            points', bbox=dict(boxstyle="round4,pad=.5",
            fc="0.9"), arrowprops=dict(arrowstyle="->",

                            connectionstyle="angle,angleA=0,angleB=80,rad=20"))

ax.annotate('Christmas', xy=('2012-12-25', 3850),
             xycoords='data',xytext=(-30, 0), textcoords='offset
             points',

             size=13, ha='right', va="center",
             bbox=dict(boxstyle="round",
             alpha=0.1),

             arrowprops=dict(arrowstyle="wedge,tail_width=0.5", alpha=0.1));

# Label the axes

ax.set(title='USA births by day of year (1969-1988)',
       ylabel='average daily births')
```

```
# Format the x axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymont
hday=15)) ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h')
);

ax.set_ylim(3600, 5400);
```

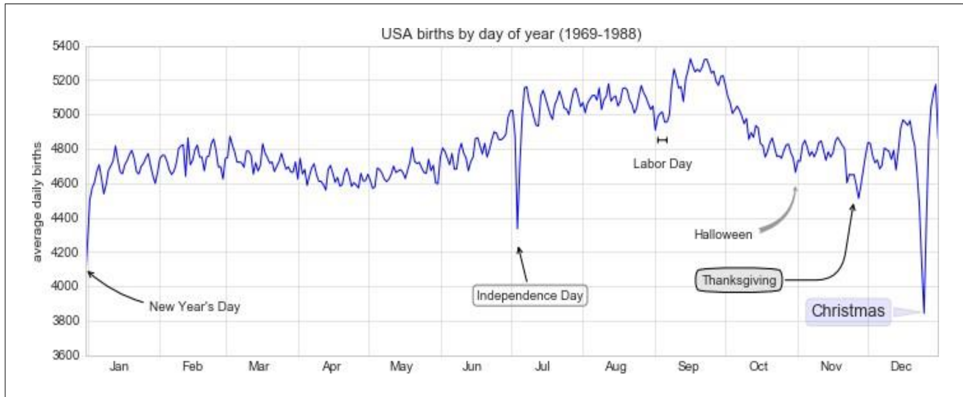


Figure. Annotated average birth rates by day

You'll notice that the specifications of the arrows and text boxes are very detailed: this gives you the power to create nearly any arrow style you wish. Unfortunately, it also means that these sorts of features often must be manually tweaked, a process that canbe very time-consuming when one is producing publication-quality graphics! Finally,I'll note that the preceding mix of styles is by no means best practice for presenting data, but rather included as a demonstration of some of the available options.

More discussion and examples of available arrow and annotation styles can be found in the Matplotlib gallery, in particular *http://matplotlib.org/examples/pylab_examples/annotation_demo2.html*.

Customizing Ticks

Matplotlib's default tick locators and formatters are designed to be generally sufficientin many common situations, but are in no way optimal for every plot. This section will give several examples of adjusting the tick locations and formatting for the par- ticular plot type you're interested in.

Before we go into examples, it will be best for us to understand further the object hierarchy of Matplotlib plots. Matplotlib aims to have a Python object representing everything that appears on the plot: for example, recall that the figure is the bound- ing

box within which plot elements appear. Each Matplotlib object can also act as a container of sub-objects; for example, each figure can contain one or more axes objects, each of which in turn contain other objects representing plot contents.

The tick marks are no exception. Each axeshas attributes xaxisand yaxis, which inturn have attributes that contain all the properties of the lines, ticks, and labels that make up the axes.

Major and Minor Ticks

Within each axis, there is the concept of a *major* tick mark and a *minor* tick mark. Asthe names would imply, major ticks are usually bigger or more pronounced, while minor ticks are usually smaller. By default, Matplotlib rarely makes use of minor ticks, but one place you can see them is within logarithmic plots (Figure):

```
In[1]: %matplotlib inline

       import matplotlib.pyplot as plt
       plt.style.use('seaborn-
       whitegrid')import numpy as np

In[2]: ax = plt.axes(xscale='log', yscale='log')
```

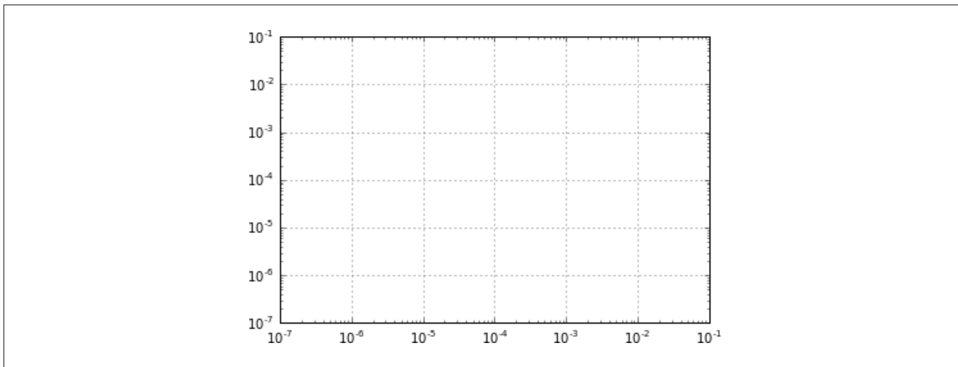


Figure . Example of logarithmic scales and labels

We see here that each major tick shows a large tick mark and a label, while each minor tick shows a smaller tick mark with no label.

We can customize these tick properties—that is, locations and labels—by setting the formatter and locator objects of each axis. Let's examine these for the x axis of the plot just shown:

```
In[3]:
       print(ax.xaxis.get_major_locator(
       ))
```

```
print(ax.xaxis.get_minor_locator
())
```

```
<matplotlib.ticker.LogLocator object at 0x107530cc0>
<matplotlib.ticker.LogLocator object at 0x107530198>
```

```
In[4]:
    print(ax.xaxis.get_major_formatter
    ())
    print(ax.xaxis.get_minor_formatte
    r())
```

```
<matplotlib.ticker.LogFormatterMathtext object at 0x107512780>
<matplotlib.ticker.NullFormatter object at 0x10752dc18>
```

We see that both major and minor tick labels have their locations specified by a LogLocator (which makes sense for a logarithmic plot). Minor ticks, though, have their labels formatted by a NullFormatter; this says that no labels will be shown.

We'll now show a few examples of setting these locators and formatters for various plots.

Hiding Ticks or Labels

Perhaps the most common tick/label formatting operation is the act of hiding ticks or labels. We can do this using plt.NullLocator() and plt.NullFormatter(), asshown here.

```
In[5]: ax = plt.axes()
ax.plot(np.random.rand(50))


ax.yaxis.set_major_locator(plt.NullLocator())
ax.xaxis.set_major_formatter(plt.NullFormattr())
```

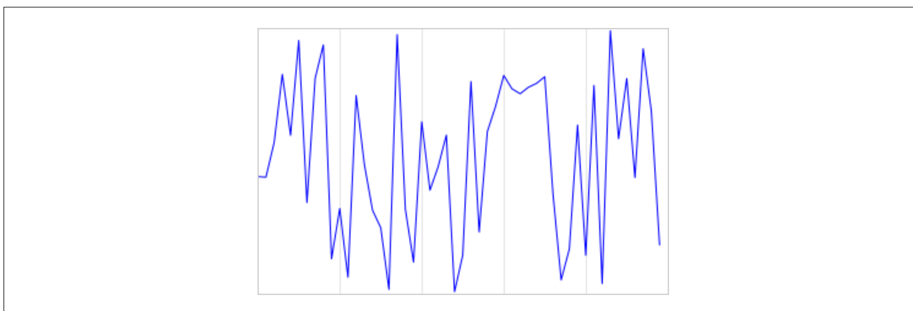


Figure . Plot with hidden tick labels (x-axis) and hidden ticks (y-axis)

Notice that we've removed the labels (but kept the ticks/gridlines) from the x axis, and removed the ticks (and thus the labels as well) from the y axis. Having no ticks atall can be useful in many situations—for example, when you want to show a grid of images. For instance, consider Figure below, which includes images of different faces, an example often used in supervised machine learning problems.

```
In[6]: fig, ax = plt.subplots(5, 5, figsize=(5, 5))
       fig.subplots_adjust(hspace=0, wspace=0)

       # Get some face data from scikit-learn

       from sklearn.datasets import fetch_olivetti_faces faces =
       fetch_olivetti_faces().images

       for i in range(5):
                   for j in range(5):
                       ax[i,     j].xaxis.set_major_locator(plt.NullLocator())
                       ax[i,     j].yaxis.set_major_locator(plt.NullLocator())
                       ax[i, j].imshow(faces[10 * i + j], cmap="bone")
```

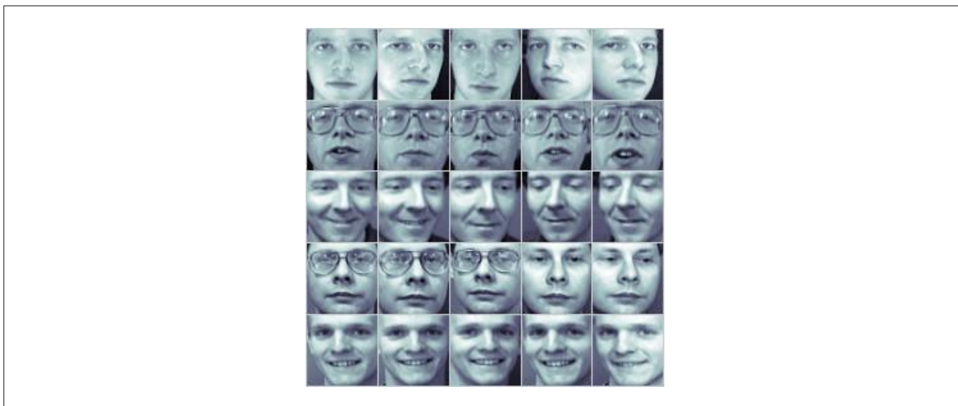


Figure. Hiding ticks within image plots

Notice that each image has its own axes, and we've set the locators to null because the tick values (pixel number in this case) do not convey relevant information for this particular visualization.

Reducing or Increasing the Number of Ticks

One common problem with the default settings is that smaller subplots can end up with crowded labels. We can see this in the plot grid shown in Figure .

```
In[7]: fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)
```

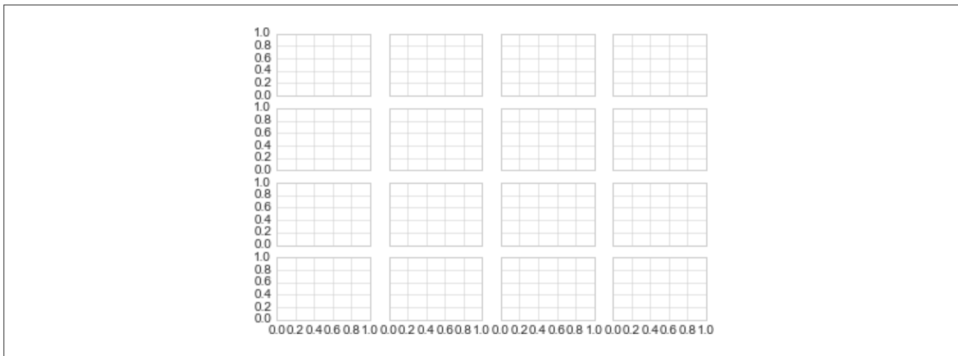


Figure 4-76. A default plot with crowded ticks

Particularly for the x ticks, the numbers nearly overlap, making them quite difficult to decipher. We can fix this with the plt.MaxNLocator(), which allows us to specify the maximum number of ticks that will be displayed. Given this maximum number, Matplotlib will use internal logic to choose the particular tick locations (Figure)

```
In[8]: # For every axis, set the x and y major locator
       for axi in ax.flat:
           axi.xaxis.set_major_locator(plt.MaxNLocator
           (3))
           axi.yaxis.set_major_locator(plt.MaxNLocator
           (3))

       fig
```

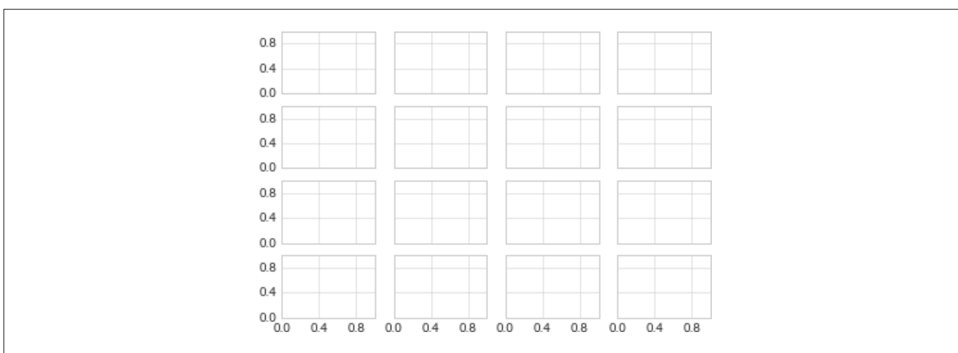


Figure . Customizing the number of ticks

This makes things much cleaner. If you want even more control over the locations of regularly spaced ticks, you might also use plt.MultipleLocator, which we'll discussin the following section.

Fancy Tick Formats

Matplotlib's default tick formatting can leave a lot to be desired; it works well as a broad default, but sometimes you'd like to do something more. Consider the plot shown in Figure below, a sine and a cosine:

```
In[9]: # Plot a sine and cosine curve
       fig, ax = plt.subplots()
       x = np.linspace(0, 3 * np.pi, 1000)
       ax.plot(x, np.sin(x), lw=3, label='Sine')
       ax.plot(x, np.cos(x), lw=3, label='Cosine')


       # Set up grid, legend, and
       limitsax.grid(True)
       ax.legend(frameon=False)
       ax.axis('equal')
       ax.set_xlim(0, 3 * np.pi);
```

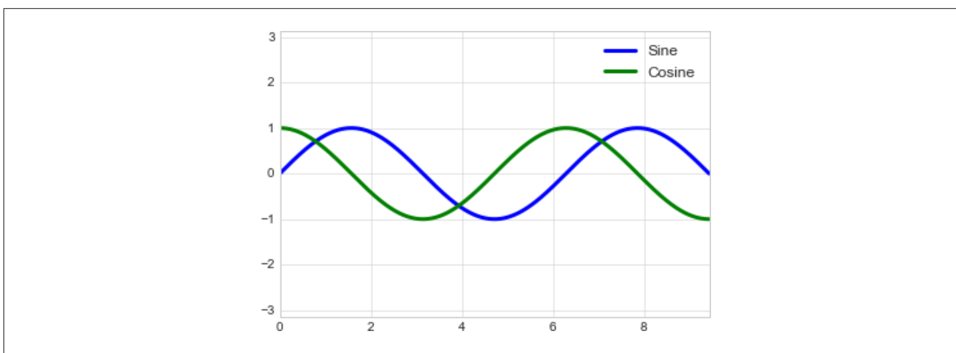
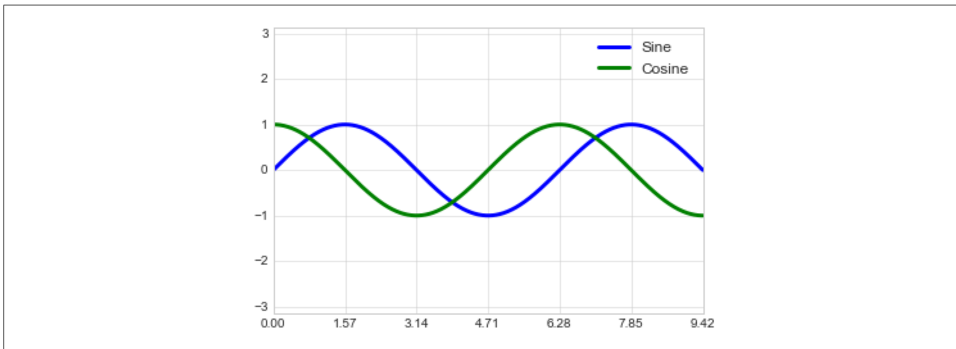


Figure. A default plot with integer ticks

There are a couple changes we might like to make. First, it's more natural for this datato space the ticks and grid lines in multiples of π. We can do this by setting a Multi pleLocator, which locates ticks at a multiple of the number you provide. For good measure, we'll add both major and minor ticks in multiples of $\pi/4$:

```
In[10]:  ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi  /
         2))
         ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi  /
```




```
         4))fig
```

Figure. Ticks at multiples of pi/2

But now these tick labels look a little bit silly: we can see that they are multiples of π,but the decimal representation does not immediately convey this. To fix this, we can change the tick formatter. There's no built-in formatter for what we want to do, so we'll instead use plt.FuncFormatter, which accepts a user-defined function giving fine-grained control over the tick outputs:

```
In[11]: def format_func(value, tick_number):
            # find number of multiples of pi/2
            N = int(np.round(2 * value / np.pi))
            if N == 0:
                return "0"
            elif N == 1:
                return r"$\pi/2$"
            elif N == 2:
                return r"$\pi$"
            elif N % 2 > 0:
                return r"${0}\pi/2$".format(N)
            else:
                return r"${0}\pi$".format(N // 2)
```

```
ax.xaxis.set_major_formatter(plt.FuncFormatter(format
_func)) fig
```

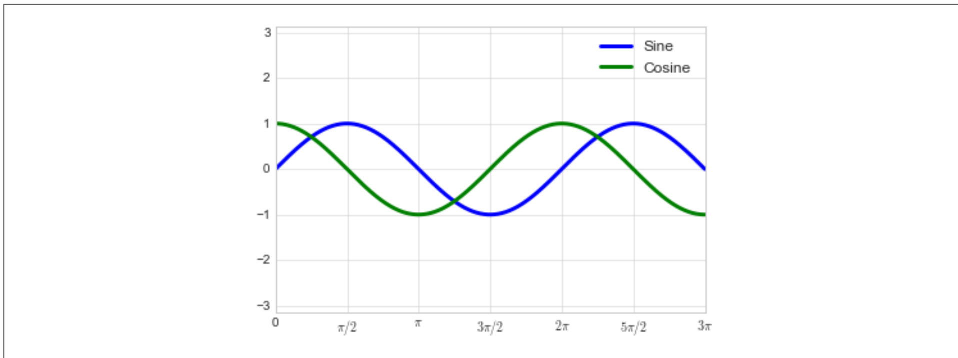


Figure. Ticks with custom labels

This is much better! Notice that we've made use of Matplotlib's LaTeX support, speci-fied by enclosing the string within dollar signs. This is very convenient for display of mathematical symbols and formulae; in this case, "π" is rendered as the Greek character π.

The plt.FuncFormatter()offers extremely fine-grained control over the appearanceof your plot ticks, and comes in very handy when you're preparing plots for presenta-tion or publication.

Summary of Formatters and Locators

We've mentioned a couple of the available formatters and locators. We'll conclude this section by briefly listing all the built-in locator and formatter options. For more information on any of these, refer to the docstrings or to the Matplotlib online docu-mentation. Each of the following is available in the pltnamespace:

Locator class	Description
NullLocator	No ticks
FixedLocator	Tick locations are fixed
IndexLocator	Locator for index plots (e.g., where x = range(len(y)))
Locator class	Description
LinearLocator	Evenly spaced ticks from min to max
LogLocator	Logarithmically ticks from min to max

MultipleLocator Ticks and range are a multiple of base

MaxNLocator Finds up to a max number of ticks at nice

locationsAutoLocator (Default) MaxNLocator with

simple defaults AutoMinorLocator Locator for minor ticks

Formatter class	Description

NullFormatter No labels on the ticks

IndexFormatter Set the strings from a list of labels

FixedFormatter Set the strings manually for the

labelsFuncFormatter User-defined function sets the

labels FormatStrFormatter Use a format string for

each value ScalarFormatter (Default)

Formatter for scalar values LogFormatter

Default formatter for log axes

We'll see additional examples of these throughout the remainder of the book.

Customizing Matplotlib: Configurations and Stylesheets

Matplotlib's default plot settings are often the subject of complaint among its users. While much is slated to change in the 2.0 Matplotlib release, the ability to customize default settings helps bring the package in line with your own aesthetic preferences.

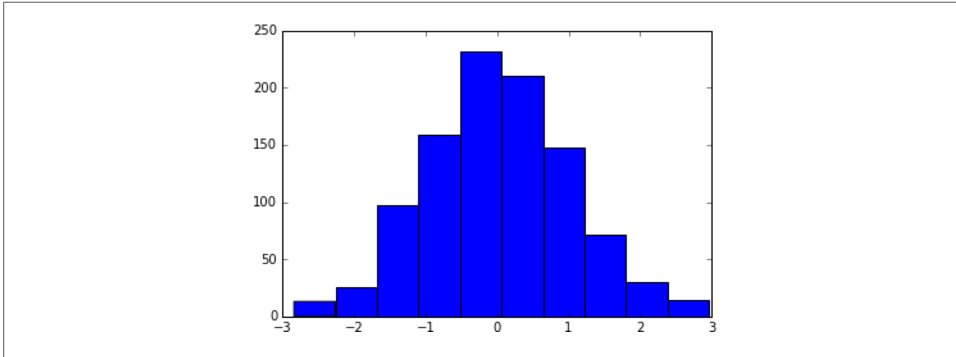
Here we'll walk through some of Matplotlib's runtime configuration (rc) options, andtake a look at the newer *stylesheets* feature, which contains some nice sets of default configurations.

Plot Customization by Hand

Throughout this chapter, we've seen how it is possible to tweak individual plot set- tings to end up with something that looks a little bit nicer than the default. It's possi- ble to do these customizations for each individual plot. For example, here is a fairly drab default histogram:

```
In[1]: import matplotlib.pyplot as plt
       plt.style.use('classic')
       import numpy as np
```

```
%matplotlib inline

In[2]: x =
       np.random.randn(1000
```




```
       )plt.hist(x);
```

Figure . A histogram in Matplotlib's default style

We can adjust this by hand to make it a much more visually pleasing plot, shown in Figure:

```
In[3]: # use a gray background

       ax =
       plt.axes(axisbg='#E6E6E6')
       ax.set_axisbelow(True)


       # draw solid white grid lines

       plt.grid(color='w', linestyle='solid')


       # hide axis spines

       for spine in
           ax.spines.values():
           spine.set_visible(False)


       # hide top and right ticks
       ax.xaxis.tick_bottom()
       ax.yaxis.tick_left()
```

```
# lighten ticks and labels
ax.tick_params(colors='gray',
direction='out')for tick in
ax.get_xticklabels():
    tick.set_color('gray')
for tick in ax.get_yticklabels():
    tick.set_color('gray')

# control face and edge color of histogram
ax.hist(x, edgecolor='#E6E6E6', color='#EE6666');
```

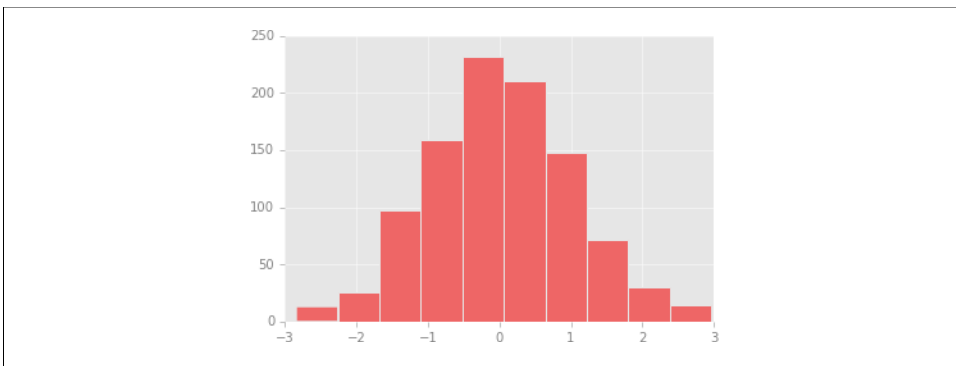


Figure . A histogram with manual customizations

This looks better, and you may recognize the look as inspired by the look of the R language's ggplot visualization package. But this took a whole lot of effort! We defi- nitely do not want to have to do all that tweaking each time we create a plot. Fortu- nately, there is a way to adjust these defaults once in a way that will work for all plots.

Changing the Defaults: rcParams

Each time Matplotlib loads, it defines a runtime configuration (rc) containing the default styles for every plot element you create. You can adjust this configuration at any time using the plt.rc convenience routine. Let's see what it looks like to modifythe rc parameters so that our default plot will look similar to what we did before.

We'll start by saving a copy of the current rcParams dictionary, so we can easily reset these changes in the current session:

```
In[4]: IPython_default = plt.rcParams.copy()
```

Now we can use the plt.rcfunction to change some of these settings:

```
In[5]: from matplotlib import
       cyclercolors = cycler('color',
                      ['#EE6666', '#3388BB', '#9988DD',
                       '#EECC55', '#88BB44', '#FFBBBB'])

       plt.rc('axes', facecolor='#E6E6E6',
              edgecolor='none',axisbelow=True,
              grid=True, prop_cycle=colors)

       plt.rc('grid', color='w', linestyle='solid')
       plt.rc('xtick', direction='out', color='gray')
       plt.rc('ytick', direction='out', color='gray')
       plt.rc('patch', edgecolor='#E6E6E6')
       plt.rc('lines', linewidth=2)
```

With these settings defined, we can now create a plot and see our settings in action.

```
In[6]: plt.hist(x);
```

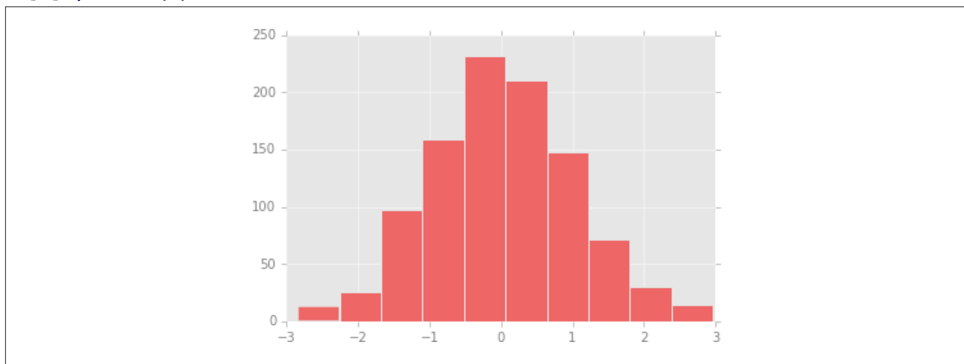


Figure 4-83. A customized histogram using rc settings

Let's see what simple line plots look like with these rcparameters:

```
In[7]: for i in range(4):
```

```
plt.plot(np.random.rand(10))
```

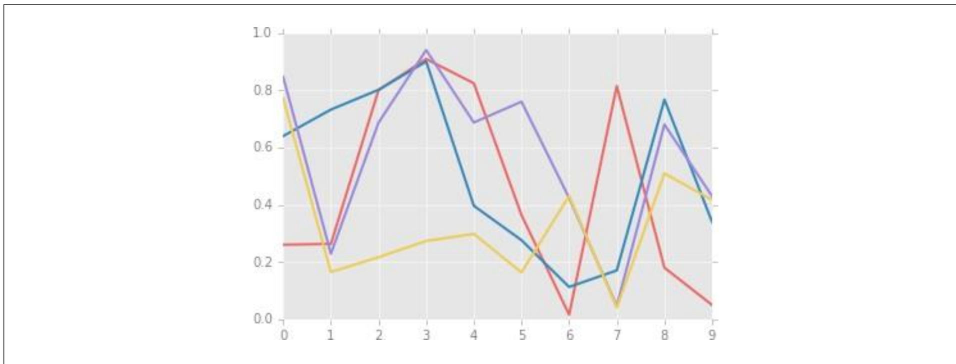


Figure. A line plot with customized styles

Stylesheets

The version 1.4 release of Matplotlib in August 2014 added a very convenient style module, which includes a number of new default stylesheets, as well as the ability to create and package your own styles. These stylesheets are formatted similarly tothe *.matplotlibrc* files mentioned earlier, but must be named with a *.mplstyle* extension.

Even if you don't create your own style, the stylesheets included by default are extremely useful. The available styles are listed in plt.style.available—here I'll list only the first five for brevity:

```
In[8]: plt.style.available[:5]
Out[8]: ['fivethirtyeight',

         'seaborn-pastel',
         'seaborn-
         whitegrid','ggplot',
         'grayscale']
```

The basic way to switch to a stylesheet is to call:

```
plt.style.use('stylename')
```

But keep in mind that this will change the style for the rest of the session! Alterna- tively, you can use the style context manager, which sets a style temporarily:

```
with
    plt.style.context('stylename'):
    make_a_plot()
```

Let's create a function that will make two basic types of plot:

```
In[9]: def hist_and_lines():
```

```
np.random.seed(0)

    fig, ax = plt.subplots(1, 2, figsize=(11, 4))
    ax[0].hist(np.random.randn(1000))

    for i in range(3):
        ax[1].plot(np.random.rand(1
        0))

    ax[1].legend(['a', 'b', 'c'], loc='lower left')
```

We'll use this to explore how these plots look using the various built-in styles.

Default style

The default style is what we've been seeing so far throughout the book; we'll start withthat. First, let's reset our runtime configuration to the notebook default:

```
In[10]: # reset rcParams

        plt.rcParams.update(IPython_default);
```

Now let's see how it looks:

```
In[11]: hist_and_lines()
```

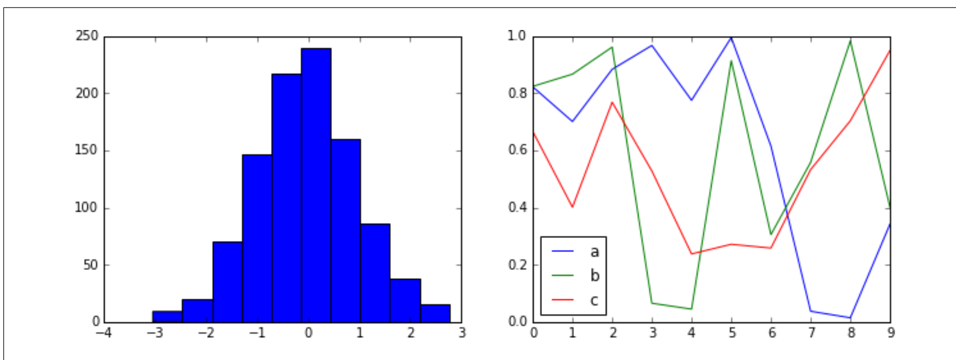


Figure. Matplotlib's default style

FiveThirtyEight style

The FiveThirtyEight style mimics the graphics found on the popular FiveThirtyEight website. As you can see in Figure , it is typified by bold colors, thick lines, and transparent axes.

```
In[12]: with plt.style.context('fivethirtyeight'):
            hist_and_lines()
```

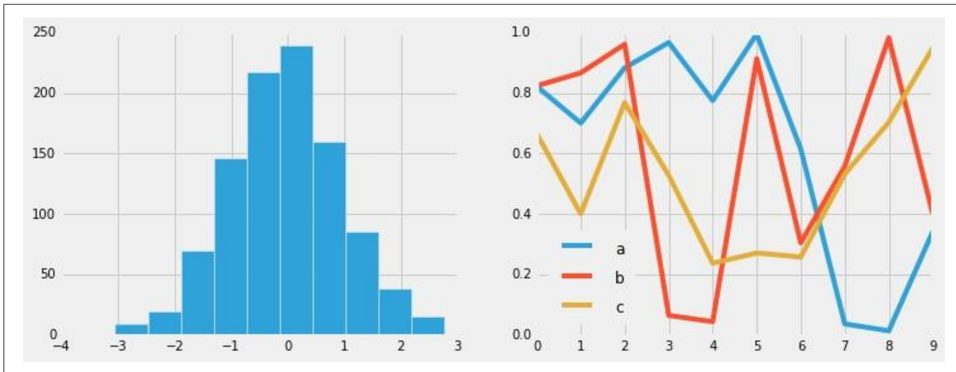


Figure. The FiveThirtyEight style

ggplot

The ggplotpackage in the R language is a very popular visualization tool. Matplot-lib's ggplotstyle mimics the default styles from that package:

```
In[13]: with plt.style.context('ggplot'):
            hist_and_lines()
```

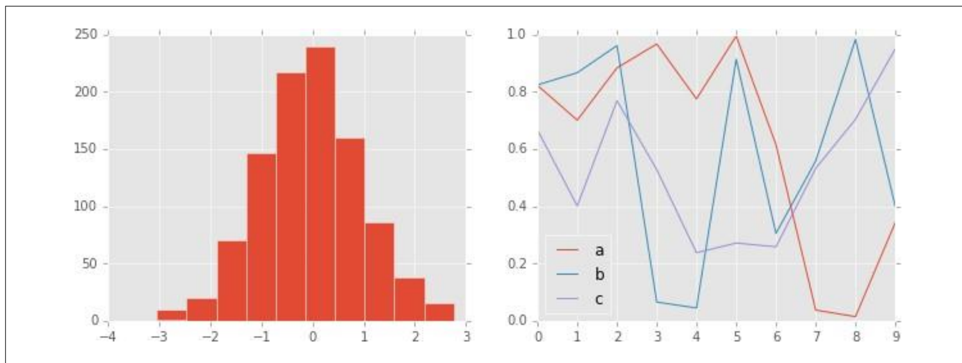
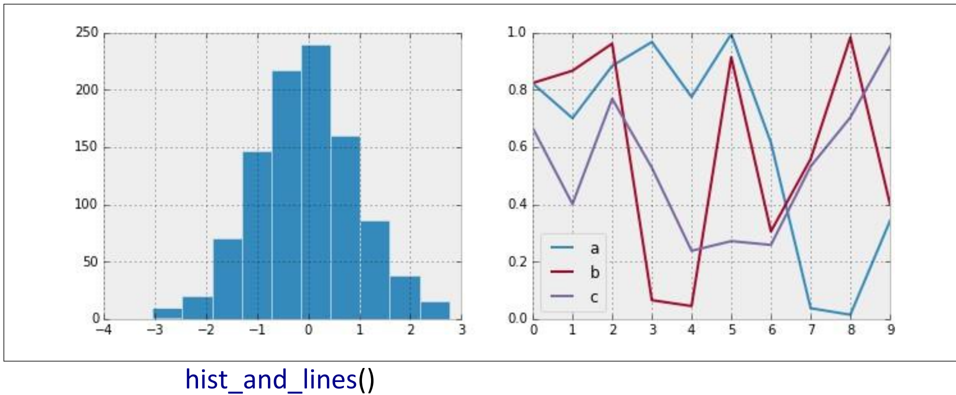


Figure. The ggplot style

Bayesian Methods for Hackers style

There is a very nice short online book called *Probabilistic Programming and BayesianMethods for Hackers*; it features figures created with Matplotlib, and uses a nice set ofrcparameters to create a consistent and visually appealing style throughout the book.This style is reproduced in the bmhstylesheet:

```
In[14]: with
        plt.style.context('bmh'):
```




```
        hist_and_lines()
```

Figure. The bmh style

Dark background

For figures used within presentations, it is often useful to have a dark rather than lightbackground. The dark_backgroundstyle provides this:

```
In[15]: with
        plt.style.context('dark_background'
        ):hist_and_lines()
```

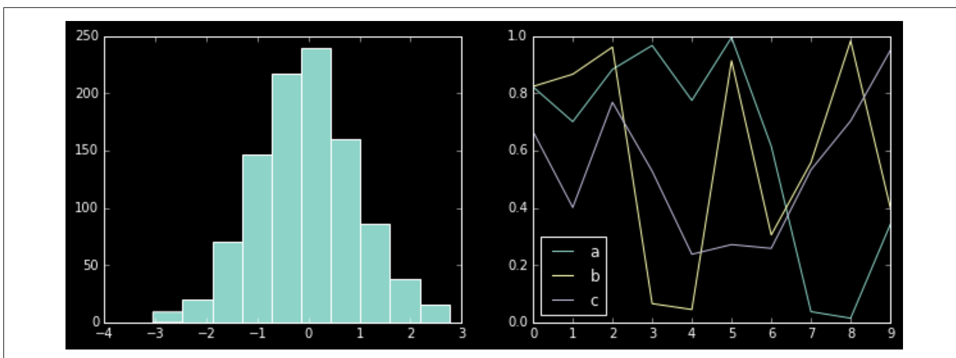


Figure . The dark_background style

Grayscale

Sometimes you might find yourself preparing figures for a print publication that doesnot accept color figures. For this, the grayscale style, shown in Figure, can bevery useful:

```
In[16]: with plt.style.context('grayscale'):
            hist_and_lines()
```

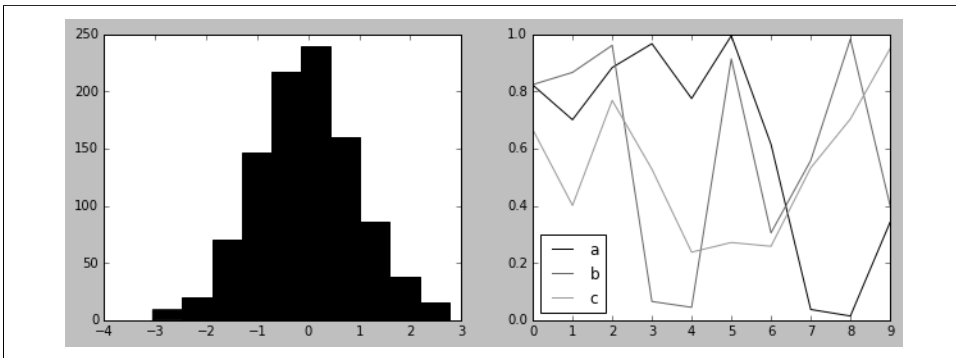


Figure 4-90. The grayscale style

Seaborn style

Matplotlib also has stylesheets inspired by the Seaborn library (discussed more fully in "Visualization with Seaborn"). As we will see, these styles are loadedautomatically when Seaborn is imported into a notebook. I've found these settings to be very nice, and tend to use them as defaults in my own data exploration:

```
In[17]: import seaborn
        hist_and_lines()
```

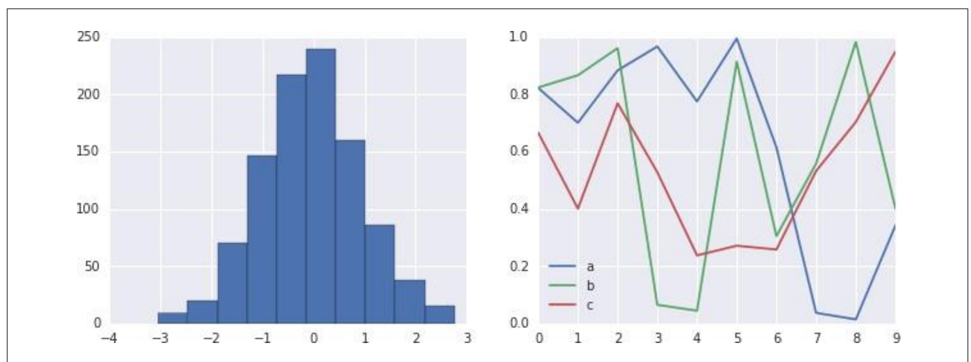


Figure. Seaborn's plotting style

With all of these built-in options for various plot styles, Matplotlib becomes much more useful for both interactive visualization and creation of figures for publication. Throughout this book, I will generally use one or more of these style conventions when

creating plots.

Day-05: Three-Dimensional Plotting in Matplotlib

Matplotlib was initially designed with only two-dimensional plotting in mind. Around the time of the 1.0 release, some three-dimensional plotting utilities were built on top of Matplotlib's two-dimensional display, and the result is a convenient (ifsomewhat limited) set of tools for three-dimensional data visualization. We enable three-dimensional plots by importing the mplot3d toolkit, included with the main Matplotlib installation:

```
In[1]: from mpl_toolkits import mplot3d
```

Once this submodule is imported, we can create a three-dimensional axes by passing the keyword projection='3d'to any of the normal axes creation routines:

```
In[2]: %matplotlib inline
       import numpy as np
       import matplotlib.pyplot as plt
In[3]: fig = plt.figure()
       ax = plt.axes(projection='3d')
```

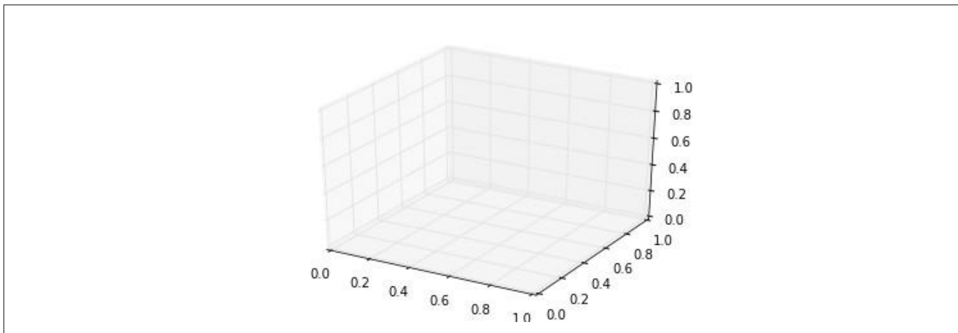


Figure. An empty three-dimensional axes

With this 3D axes enabled, we can now plot a variety of three-dimensional plot types. Three-dimensional plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook; recall that to use interactive figures, you can use %matplotlib notebook rather than %matplotlib inline when running this code.

Three-Dimensional Points and Lines

The most basic three-dimensional plot is a line or scatter plot created from sets of (x, y, z)

triples. In analogy with the more common two-dimensional plots discussed ear- lier, we can create these using the ax.plot3D and ax.scatter3D functions. The call signature for these is nearly identical to that of their two-dimensional counterparts,so you can refer to “Simple Line Plots” and for more information on controlling the output. Here we’ll plot a trigono- metric spiral, along with some points drawn randomly near the line:

```
In[4]: ax = plt.axes(projection='3d')

    # Data for a three-dimensional
    linezline = np.linspace(0, 15, 1000)
    xline = np.sin(zline)

    yline = np.cos(zline)

    ax.plot3D(xline, yline, zline, 'gray')

    # Data for three-dimensional scattered points

    zdata = 15 * np.random.random(100)

    xdata = np.sin(zdata) + 0.1 *
    np.random.randn(100)ydata = np.cos(zdata) +
    0.1 * np.random.randn(100)

    ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

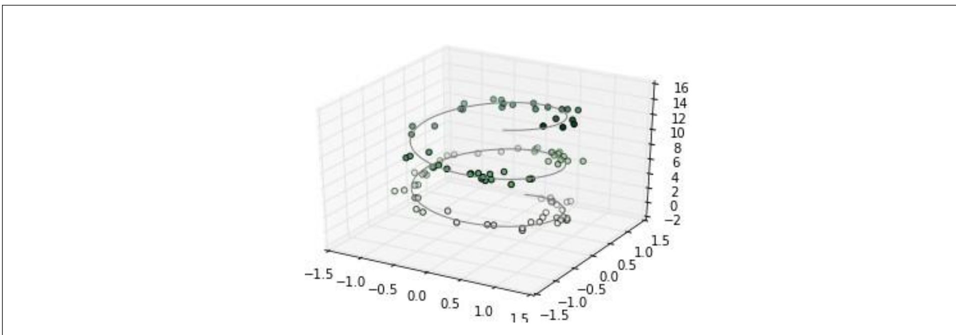


Figure. Points and lines in three dimensions

Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page. While the three-dimensional effect is sometimes difficult to see within a static image, an interactive view can lead to some nice intuition about the layout of the points.

Three-Dimensional Contour Plots

Analogous to the contour plots we explored in “Density and Contour Plots” , mplot3d

contains tools to create three-dimensional relief plots using the same inputs. Like two-dimensional ax.contour plots, ax.contour3D requires all the inputdata to be in the form of two-dimensional regular grids, with the Z data evaluated at each point. Here we'll show a three-dimensional contour diagram of a three- dimensional sinusoidal function:

```
In[5]: def f(x, y):
           return np.sin(np.sqrt(x ** 2 + y ** 2))


       x = np.linspace(-6, 6, 30)

       y = np.linspace(-6, 6, 30)


       X, Y = np.meshgrid(x,
       y)Z = f(X, Y)

In[6]: fig = plt.figure()

       ax = plt.axes(projection='3d')
       ax.contour3D(X, Y, Z, 50,
       cmap='binary')ax.set_xlabel('x')

       ax.set_ylabel('y')
       ax.set_zlabel('z');
```

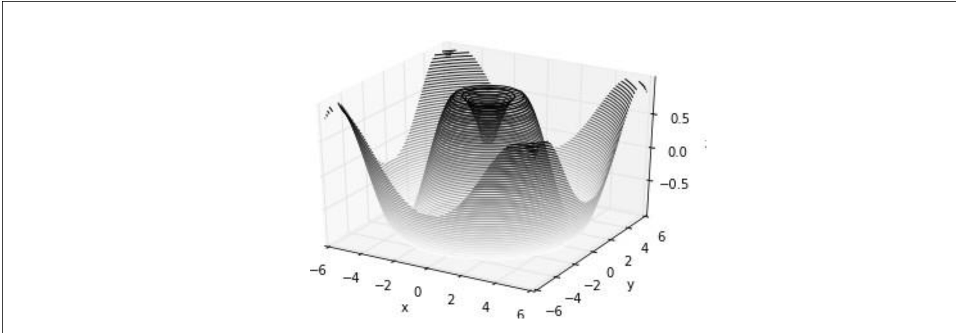


Figure. A three-dimensional contour plot

Sometimes the default viewing angle is not optimal, in which case we can use the view_init method to set the elevation and azimuthal angles. In this example (the result of which is shown in Figure), we'll use an elevation of 60 degrees (that is,60 degrees above the *x*-*y* plane) and an azimuth of 35 degrees (that is, rotated 35 degrees counter-clockwise about the *z*-axis):

```
In[7]: ax.view_init(60, 35)
       fig
```

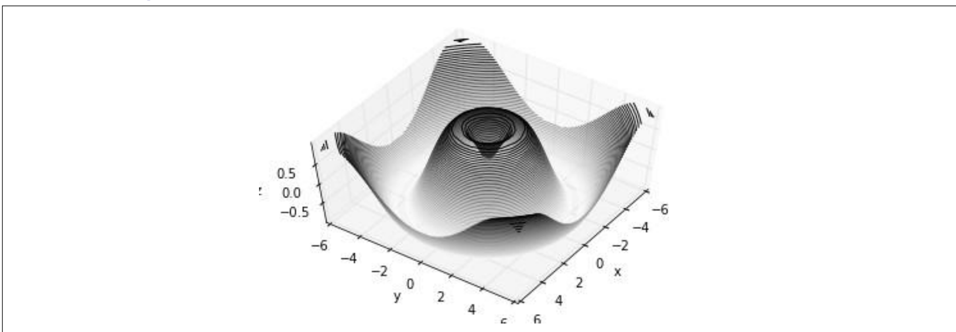


Figure. Adjusting the view angle for a three-dimensional plot

Again, note that we can accomplish this type of rotation interactively by clicking and dragging when using one of Matplotlib's interactive backends.

Wireframes and Surface Plots

Two other types of three-dimensional plots that work on gridded data are wireframes and surface plots. These take a grid of values and project it onto the specified three-dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize. Here's an example using a wireframe (Figure):

```
In[8]: fig = plt.figure()
```

```
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z,
color='black')ax.set_title('wireframe');
```

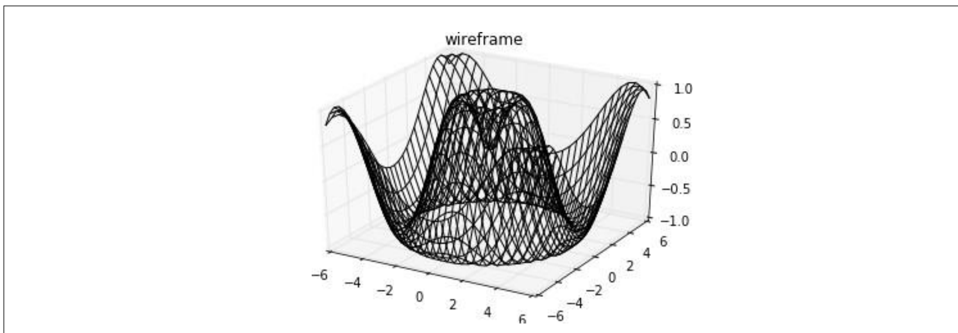


Figure. A wireframe plot

A surface plot is like a wireframe plot, but each face of the wireframe is a filled poly-gon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized .

```
In[9]: ax = plt.axes(projection='3d')
        ax.plot_surface(X, Y, Z, rstride=1, cstride=1,

                                cmap='viridis',
        edgecolor='none')ax.set_title('surface');
```

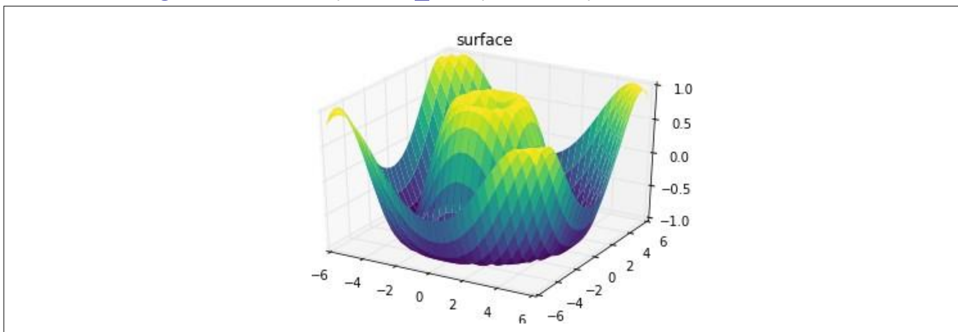


Figure. A three-dimensional surface plot

Note that though the grid of values for a surface plot needs to be two-dimensional, it need not be rectilinear. Here is an example of creating a partial polar grid, which when used with the surface3D plot can give us a slice into the function we're visualizing .

```
In[10]: r = np.linspace(0, 6, 20)

         theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
         r, theta = np.meshgrid(r, theta)
```

```
X = r * np.sin(theta)
Y = r * np.cos(theta)
Z = f(X, Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
```

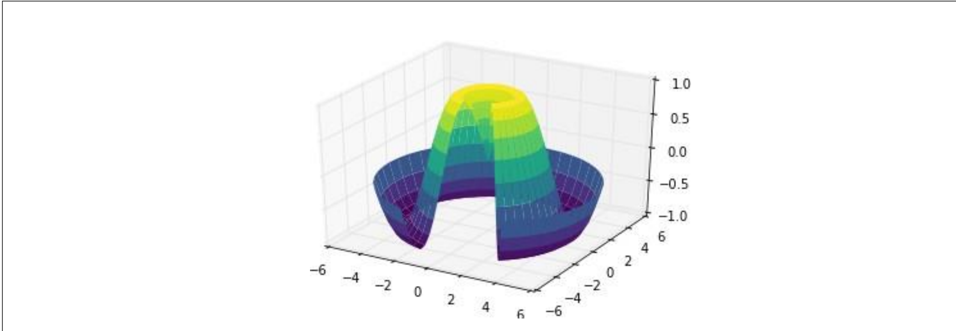


Figure. A polar surface plot

Surface Triangulations

For some applications, the evenly sampled grids required by the preceding routines are overly restrictive and inconvenient. In these situations, the triangulation-based plots can be very useful. What if rather than an even draw from a Cartesian or a polargrid, we instead have a set of random draws?

```
In[11]:    theta    =    2    *    np.pi    *
       np.random.random(1000)  r  =  6  *
       np.random.random(1000)

       x = np.ravel(r * np.sin(theta))
       y = np.ravel(r * np.cos(theta))
       z = f(x, y)
```

We could create a scatter plot of the points to get an idea of the surface we're samplingfrom:

```
In[12]: ax = plt.axes(projection='3d')

       ax.scatter(x, y, z, c=z, cmap='viridis', linewidth=0.5);
```

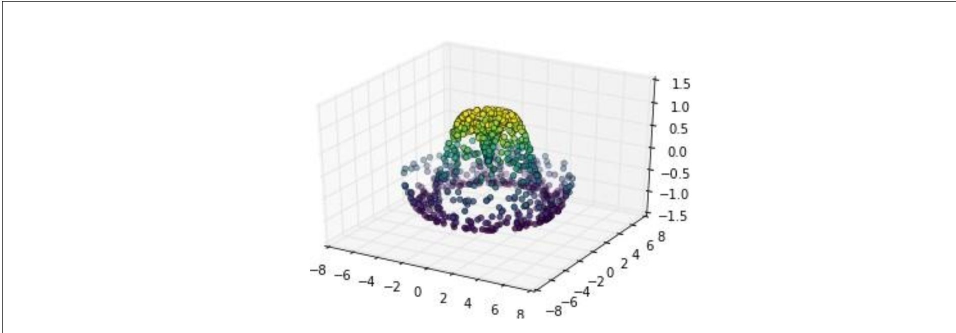


Figure. A three-dimensional sampled surface

This leaves a lot to be desired. The function that will help us in this case is ax.plot_trisurf, which creates a surface by first finding a set of triangles formed between adjacent points (the result is shown in Figure ; remember that x, y, andzhere are one-dimensional arrays):

```
In[13]: ax = plt.axes(projection='3d')
        ax.plot_trisurf(x, y, z,
                        cmap='viridis', edgecolor='none');
```

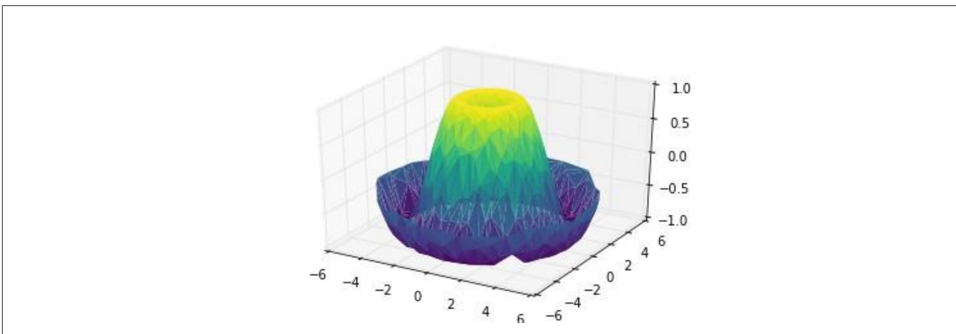


Figure. A triangulated surface plot

The result is certainly not as clean as when it is plotted with a grid, but the flexibility of such a triangulation allows for some really interesting three-dimensional plots. For example, it is actually possible to plot a three-dimensional Möbius strip using this, aswe'll see next.

Example: Visualizing a Möbius strip

A Möbius strip is similar to a strip of paper glued into a loop with a half-twist. Topologically, it's quite interesting because despite appearances it has only a single side! Here we will visualize such an object using Matplotlib's three-dimensional tools. Thekey to creating the Möbius strip is to think about its parameterization: it's a two-dimensional strip, so we need two intrinsic dimensions. Let's call them ϑ, which ranges

from 0 to 2π around the loop, and *w* which ranges from –1 to 1 across the width of the strip:

```
In[14]: theta = np.linspace(0, 2 * np.pi, 30)
        w = np.linspace(-0.25, 0.25, 8)

        w, theta = np.meshgrid(w, theta)
```

Now from this parameterization, we must determine the *(x, y, z)* positions of the embedded strip.

Thinking about it, we might realize that there are two rotations happening: one is the position of the loop about its center (what we've called ϑ), while the other is the twist-ing of the strip about its axis (we'll call this ϕ). For a Möbius strip, we must have the strip make half a twist during a full loop, or $\Delta\phi = \Delta\vartheta/2$.

```
In[15]: phi = 0.5 * theta
```

Now we use our recollection of trigonometry to derive the three-dimensional embed-ding. We'll define *r*, the distance of each point from the center, and use this to find the embedded *x*, *y*, *z* coordinates:

```
In[16]: # radius in x-y plane

        r = 1 + w * np.cos(phi)

        x = np.ravel(r * np.cos(theta))
        y = np.ravel(r * np.sin(theta))
        z = np.ravel(w * np.sin(phi))
```

Finally, to plot the object, we must make sure the triangulation is correct. The best way to do this is to define the triangulation *within the underlying parameterization*, and then let Matplotlib project this triangulation into the three-dimensional space of the Möbius strip. This can be accomplished as follows:

```
In[17]: # triangulate in the underlying
            parameterization

        from matplotlib.tri import Triangulation

        tri = Triangulation(np.ravel(w), np.ravel(theta))


        ax = plt.axes(projection='3d')

        ax.plot_trisurf(x, y, z, triangles=tri.triangles,

                        cmap='viridis', linewidths=0.2);

        ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1);
```

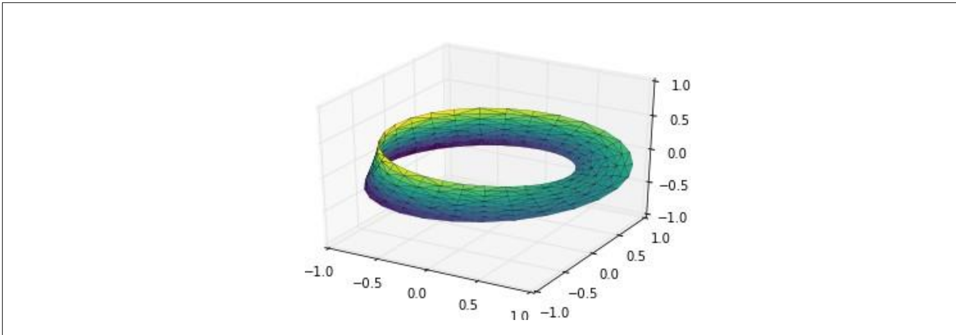


Figure. Visualizing a Möbius strip

Combining all of these techniques, it is possible to create and display a wide variety of three-dimensional objects and patterns in Matplotlib.

Week 5: Data visualization with Seaborn

Day-01: Visualization with Seaborn

Matplotlib has proven to be an incredibly useful and popular visualization tool, but even avid users will admit it often leaves much to be desired. There are several valid complaints about Matplotlib that often come up:

- Prior to version 2.0, Matplotlib's defaults are not exactly the best choices. It was based off of MATLAB circa 1999, and this often shows.
- Matplotlib's API is relatively low level. Doing sophisticated statistical visualiza- tion is possible, but often requires a *lot* of boilerplate code.
- Matplotlib predated Pandas by more than a decade, and thus is not designed for use with Pandas DataFrames. In order to visualize data from a Pandas DataFrame,you must extract each Series and often concatenate them together into the right format. It would be nicer to have a plotting library that can intelligently use the DataFramelabels in a plot.

An answer to these problems is Seaborn. Seaborn provides an API on top of Matplot-lib that offers sane choices for plot style and color defaults, defines simple high-level functions for common statistical plot types, and integrates with the functionality pro- vided by Pandas DataFrames.

Seaborn Versus Matplotlib

Here is an example of a simple random-walk plot in Matplotlib, using its classic plot formatting and colors. We start with the typical imports:

```
In[1]: import matplotlib.pyplot as plt

        plt.style.use('classic')

        %matplotlib inline
        import numpy as
        np import pandas
        as pd
```

Now we create some random walk data:

```
In[2]: # Create some data

        rng =
        np.random.RandomState(0)
        x = np.linspace(0, 10, 500)
```

```
y = np.cumsum(rng.randn(500, 6), 0)
```

And do a simple plot (Figure):

```
In[3]: # Plot the data with Matplotlib defaults
       plt.plot(x, y)
       plt.legend('ABCDEF', ncol=2, loc='upper left');
```

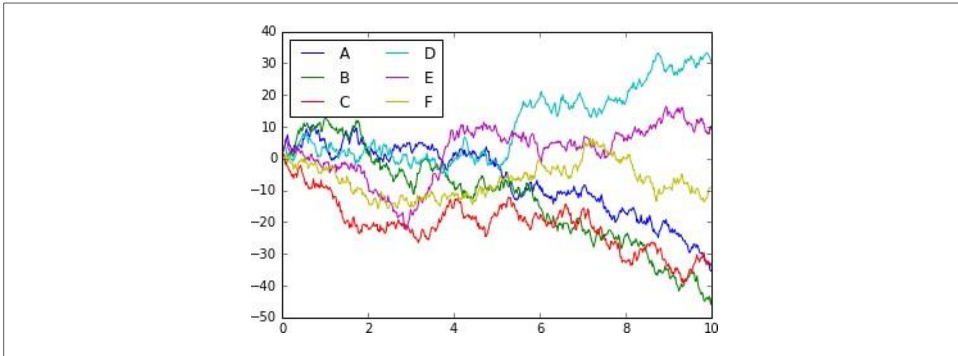


Figure. Data in Matplotlib's default style

Although the result contains all the information we'd like it to convey, it does so in a way that is not all that aesthetically pleasing, and even looks a bit old-fashioned in thecontext of 21st-century data visualization.

Now let's take a look at how it works with Seaborn. As we will see, Seaborn has manyof its own high-level plotting routines, but it can also overwrite Matplotlib's default parameters and in turn get even simple Matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's set() method. By convention, Sea- born is imported as sns:

```
In[4]: import seaborn as sns
       sns.set()
```

Now let's rerun the same two lines as before (Figure):

```
In[5]: # same plotting code as above!
       plt.plot(x, y)
```

```
plt.legend('ABCDEF', ncol=2, loc='upper left');
```

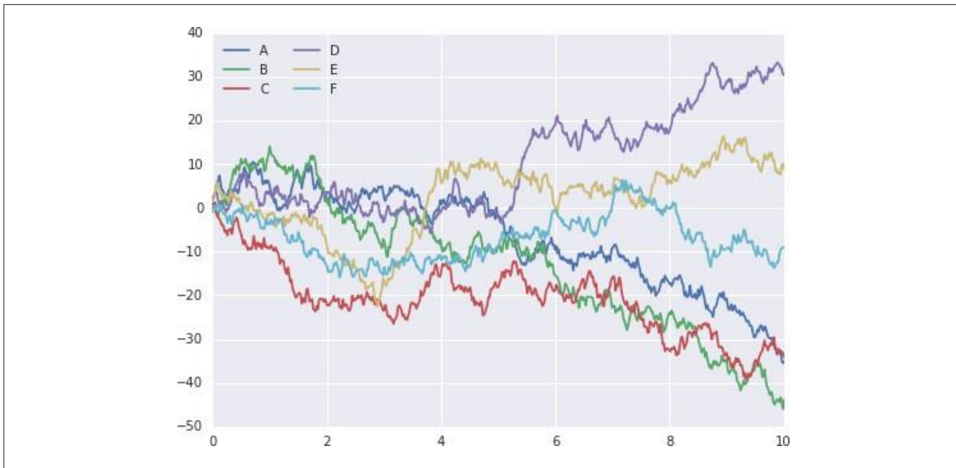


Figure. Data in Seaborn's default style

Exploring Seaborn Plots

The main idea of Seaborn is that it provides high-level commands to create a variety of plot types useful for statistical data exploration, and even some statistical model fitting.

Let's take a look at a few of the datasets and plot types available in Seaborn. Note thatall of the following *could* be done using raw Matplotlib commands (this is, in fact, what Seaborn does under the hood), but the Seaborn API is much more convenient.

Histograms, KDE, and densities

Often in statistical data visualization, all you want is to plot histograms and joint dis-tributions of variables. We have seen that this is relatively straightforward in Matplot-lib (Figure):

```
In[6]: data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]],
       size=2000)data = pd.DataFrame(data, columns=['x', 'y'])
       for col in 'xy':
           plt.hist(data[col], normed=True, alpha=0.5)
```

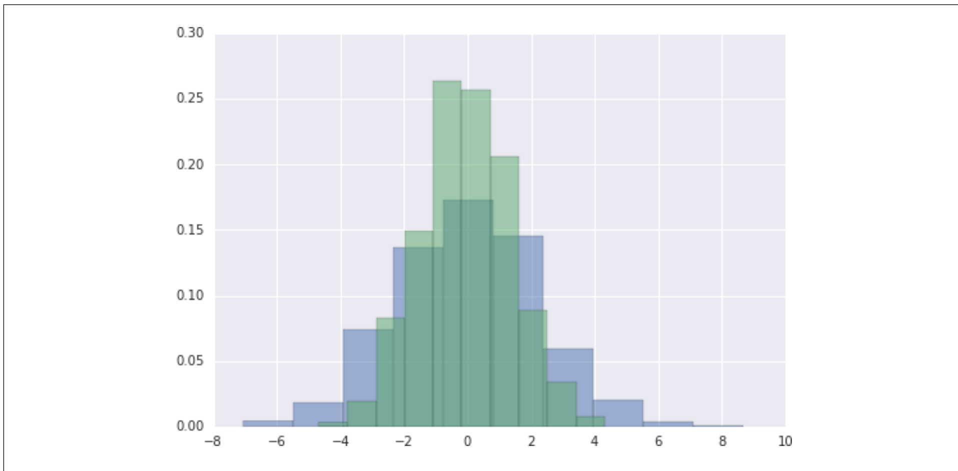


Figure. Histograms for visualizing distributions

Rather than a histogram, we can get a smooth estimate of the distribution using akernel density estimation, which Seaborn does with sns.kdeplot(Figure):

```
In[7]: for col in 'xy':
           sns.kdeplot(data[col], shade=True)
```

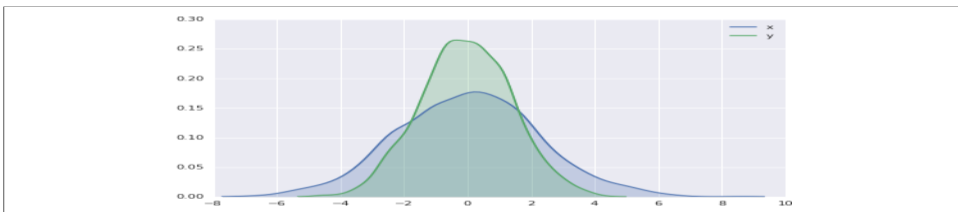


Figure. Kernel density estimates for visualizing distributions

Histograms and KDE can be combined using distplot(Figure):

```
In[8]: sns.distplot(data['x'])
       sns.distplot(data['y'])
```

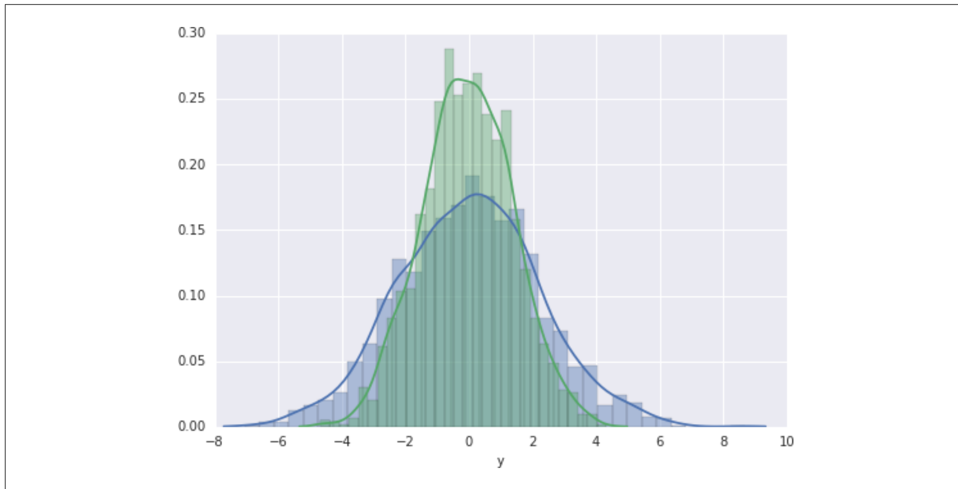


Figure. Kernel density and histograms plotted together

If we pass the full two-dimensional dataset to kdeplot, we will get a two-dimensionalvisualization of the data (Figure):

```
In[9]: sns.kdeplot(data);
```

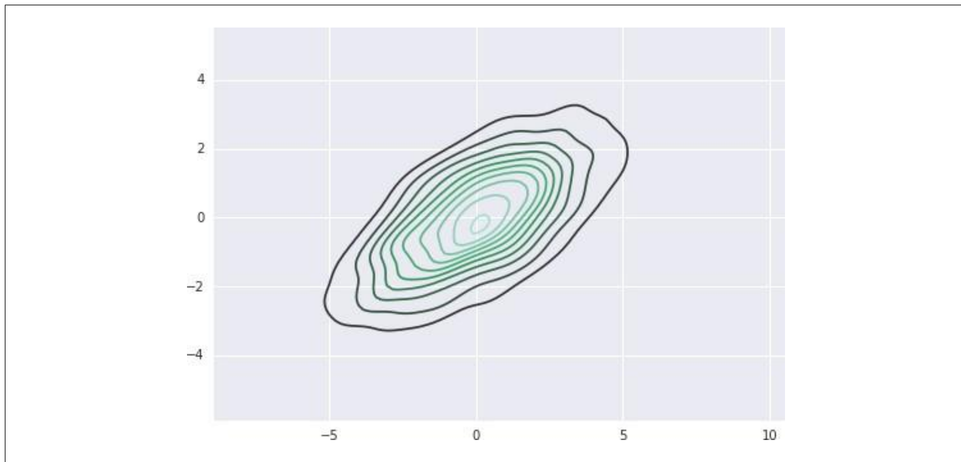


Figure. A two-dimensional kernel density plot

We can see the joint distribution and the marginal distributions together using sns.jointplot. For this plot, we'll set the style to a white background (Figure):

```
In[10]: with sns.axes_style('white'):
```

```
sns.jointplot("x", "y", data, kind='kde');
```

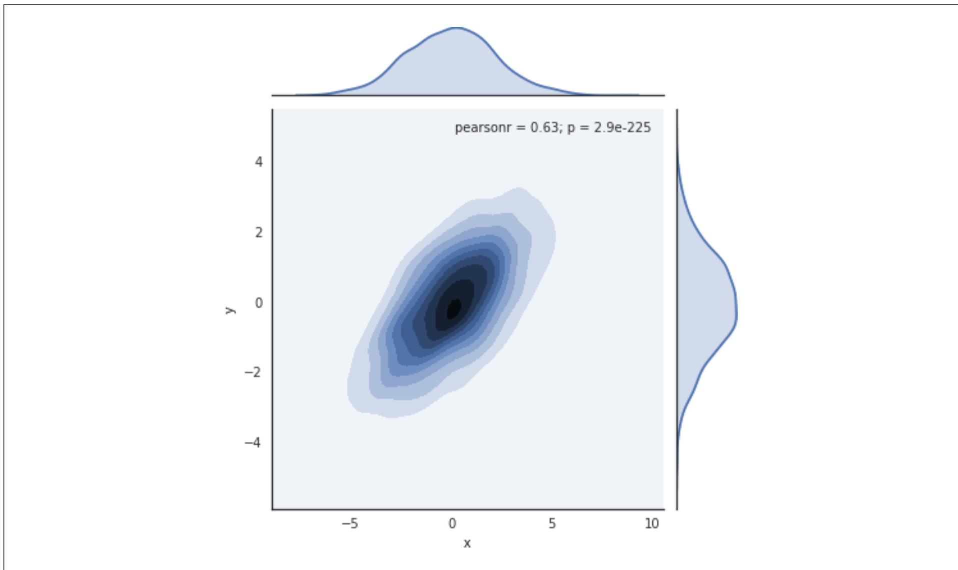


Figure. A joint distribution plot with a two-dimensional kernel density estimate

There are other parameters that can be passed to jointplot—for example, we canuse a hexagonally based histogram instead (Figure):

```
In[11]: with sns.axes_style('white'):
            sns.jointplot("x", "y", data, kind='hex')
```

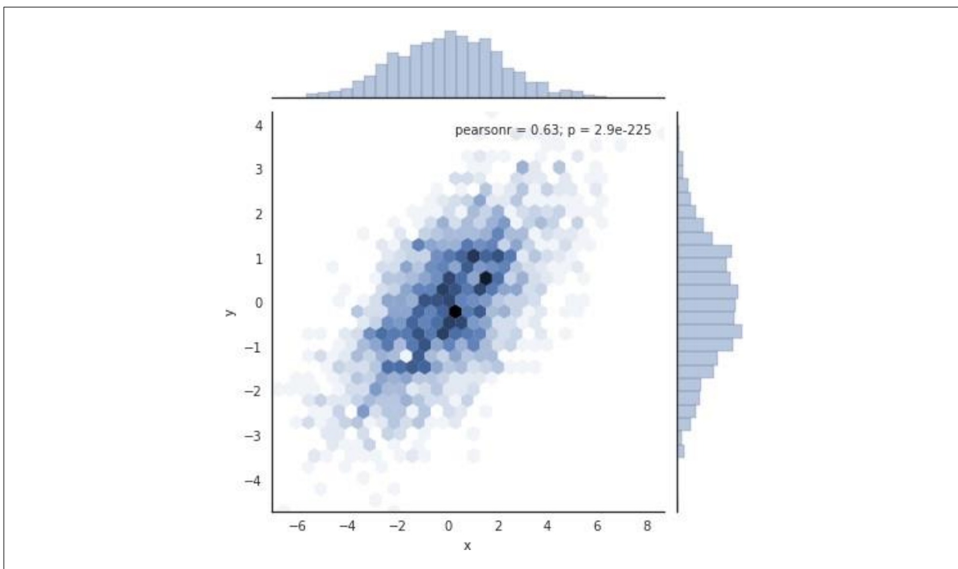


Figure. A joint distribution plot with a hexagonal bin representation

Pair plots

When you generalize joint plots to datasets of larger dimensions, you end up with *pair plots*. This is very useful for exploring correlations between multidimensional data, when you'd like to plot all pairs of values against each other.

We'll demo this with the well-known Iris dataset, which lists measurements of petals and sepals of three iris species:

```
In[12]: iris = sns.load_dataset("iris")
        iris.head()
```

Out[12]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Visualizing the multidimensional relationships among the samples is as easy as call-ing sns.pairplot:

```
In[13]: sns.pairplot(iris, hue='species', size=2.5);
```

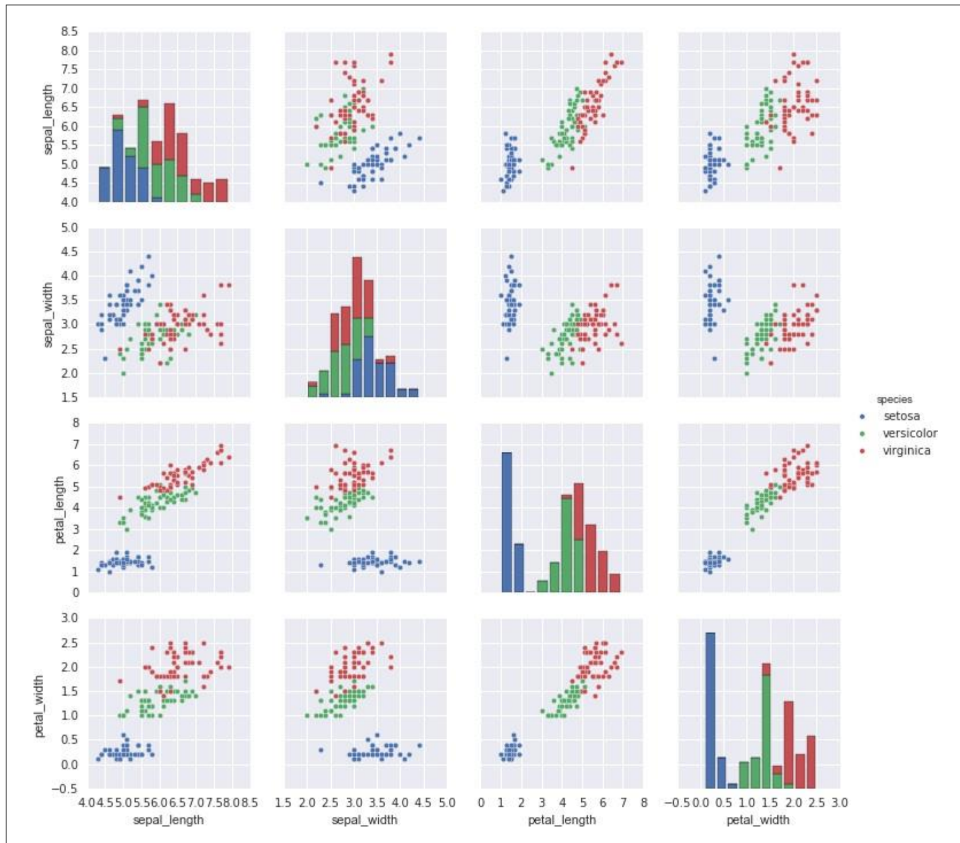


Figure. A pair plot showing the relationships between four variables

Faceted histograms

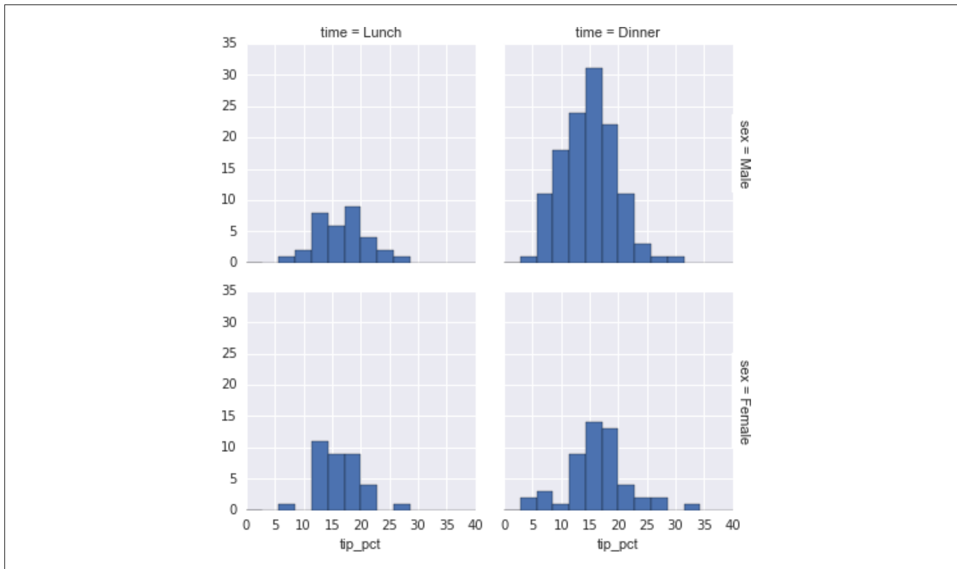
Sometimes the best way to view data is via histograms of subsets. Seaborn's FacetGrid makes this extremely simple. We'll take a look at some data that shows the amount that restaurant staff receive in tips based on various indicator data (Figure):

```
In[14]: tips = sns.load_dataset('tips')
        tips.head()
```

Out[14]:	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2

1	10.3	1.6 6	Male	No	Su n	Dinne r	3
4 2	21.0	3.5 0	Male	No	Su n	Dinne r	3
1 3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

```
In[15]: tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']


        grid = sns.FacetGrid(tips, row="sex", col="time",
        margin_titles=True)grid.map(plt.hist, "tip_pct",
```




```
        bins=np.linspace(0, 40, 15));
```

Figure. An example of a faceted histogram

Factor plots

Factor plots can be useful for this kind of visualization as well. This allows you to view the distribution of a parameter within bins defined by any other parameter (Figure):

```
In[16]: with sns.axes_style(style='ticks'):

            g = sns.factorplot("day", "total_bill", "sex", data=tips, kind="box")
            g.set_axis_labels("Day", "Total Bill");
```

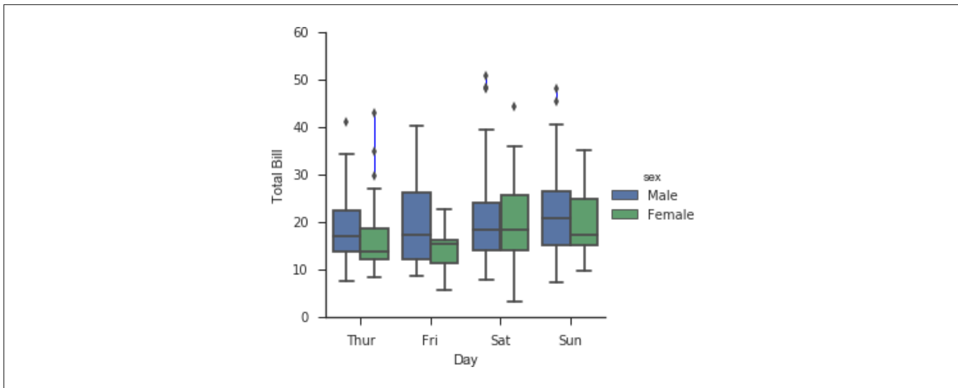


Figure. An example of a factor plot, comparing distributions given various discrete factors

Joint distributions

Similar to the pair plot we saw earlier, we can use sns.jointplot to show the joint distribution between different datasets, along with the associated marginal distributions (Figure):

```
In[17]: with sns.axes_style('white'):
            sns.jointplot("total_bill", "tip", data=tips, kind='hex')
```

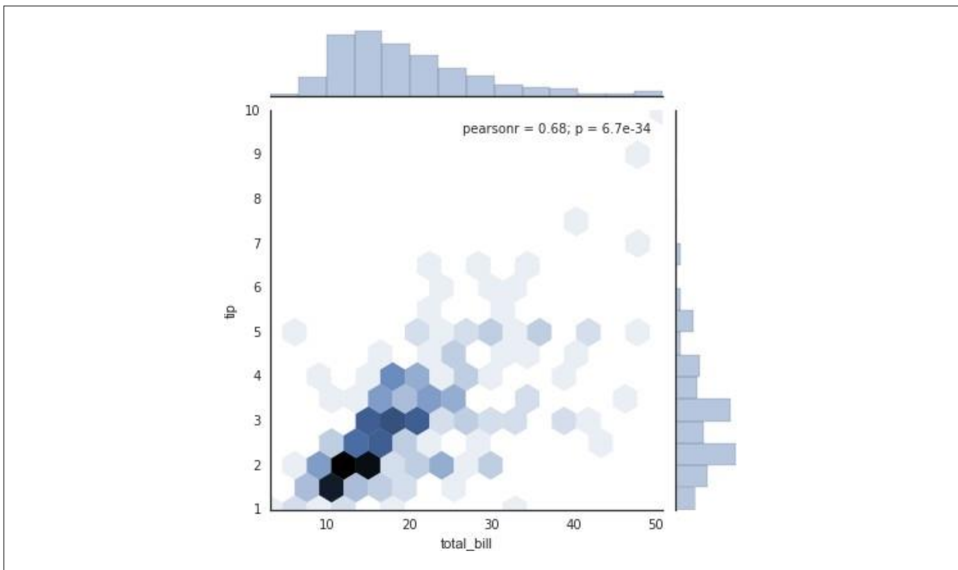


Figure. A joint distribution plot

The joint plot can even do some automatic kernel density estimation and regression (Figure):

```
In[18]: sns.jointplot("total_bill", "tip", data=tips, kind='reg');
```

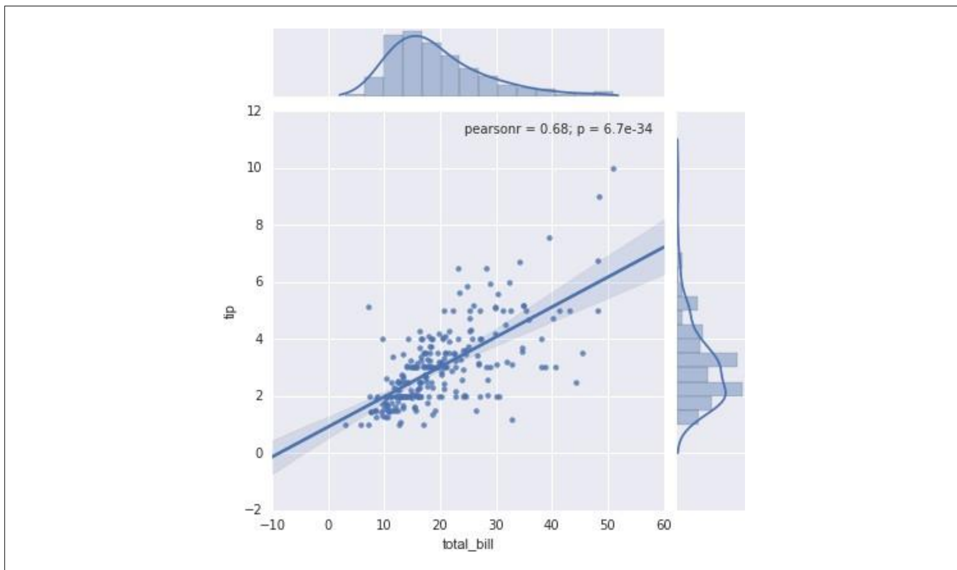


Figure. A joint distribution plot with a regression fit

Bar plots

Time series can be plotted with sns.factorplot. In the following example (visualizedin Figure)

```
In[19]: planets =
            sns.load_dataset('planets')
            planets.head()
```

Out[19]:	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

```
In[20]: with sns.axes_style('white'):
            g = sns.factorplot("year", data=planets, aspect=2,
```

```
kind="count",
color='steelblue')g.set_xticklabels(step=5)
```

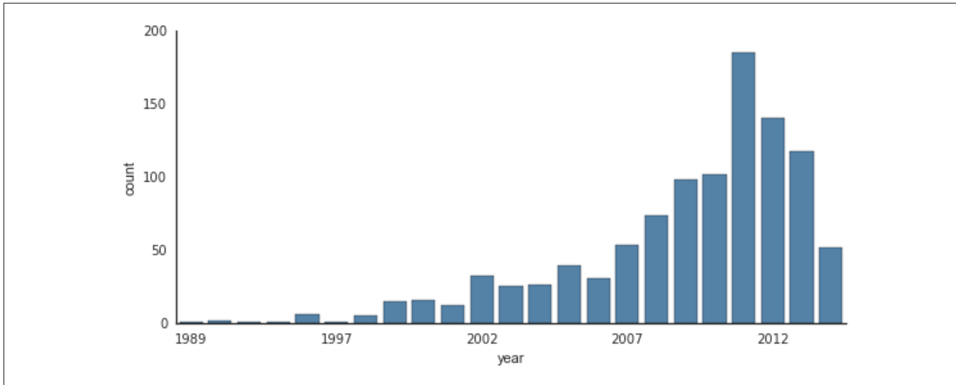


Figure 4-124. A histogram as a special case of a factor plot

We can learn more by looking at the *method* of discovery of each of these planets, as illustrated in following Figure:

```
In[21]: with sns.axes_style('white'):
            g = sns.factorplot("year", data=planets, aspect=4.0,
                               kind='count',hue='method',
                               order=range(2001, 2015))
            g.set_ylabels('Number of Planets Discovered')
```

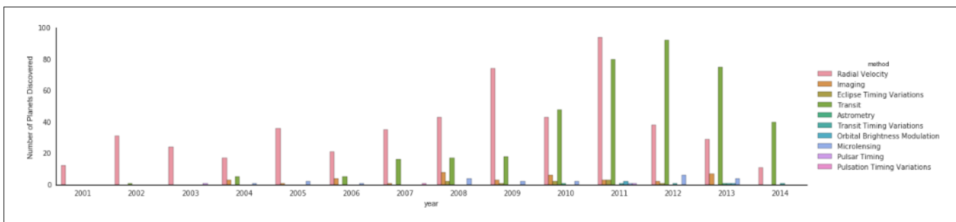


Figure . Number of planets discovered by year and type

Example: Exploring Marathon Finishing Times

Here we'll look at using Seaborn to help visualize and understand finishing results from a marathon. I've scraped the data from sources on the Web, aggregated it and removed any identifying information, and put it on GitHub where it can be downloa- ded (if you are interested in using Python for web scraping, I would recommend *WebScraping with Python* by Ryan Mitchell). We will start by downloading the data fromthe Web, and loading it into Pandas:

```
In[22]:

# !curl -O
https://raw.githubusercontent.com/jakevdp/marathon-
data/# master/marathon-data.csv
```

```
In[23]: data = pd.read_csv('marathon-
        data.csv')data.head()
```

```
Out[23]:   age gender   split      final
        0   33      M  01:05:38
        02:08:51

        1   32      M  01:06:26  02:09:28

        2   31      M  01:06:49  02:10:42

        3   38      M  01:06:16  02:13:45

        4   31      M  01:06:32  02:13:59
```

By default, Pandas loaded the time columns as Python strings (type object); we cansee this by looking at the dtypesattribute of the DataFrame:

```
In[24]: data.dtypes
Out[24]: age

                   int6

4

        gender   object

        split    object

        final
                 objec
        tdtype: object
```

Let’s fix this by providing a converter for the times:

```
In[25]: def convert_time(s):
            h, m, s = map(int, s.split(':'))
            return pd.datetools.timedelta(hours=h, minutes=m, seconds=s)

        data = pd.read_csv('marathon-data.csv',
                           converters={'split':convert_time, 'final':convert_time})
        data.head()
```

Out[25]:

	age	gender	split	final
0	33	M	01:05:38	02:08:51
1	32	M	01:06:26	02:09:28
2	31	M	01:06:49	02:10:42
3	38	M	01:06:16	02:13:45
4	31	M	01:06:32	02:13:59

```
In[26]: data.dtypes
```

```
Out[26]: age              int64
         gender          object
         split  timedelta64[ns]
         final  timedelta64[ns]
         dtype: object
```

That looks much better. For the purpose of our Seaborn plotting utilities, let's nextadd columns that give the times in seconds:

```
In[27]: data['split_sec'] = data['split'].astype(int) / 1E9
        data['final_sec'] = data['final'].astype(int) / 1E9
        data.head()
```

Out[27]:

	age	gender	split	final	split_sec	final_sec
0	33	M	01:05:38	02:08:51	3938.0	7731.0
1	32	M	01:06:26	02:09:28	3986.0	7768.0
2	31	M	01:06:49	02:10:42	4009.0	7842.0
3	38	M	01:06:16	02:13:45	3976.0	8025.0
4	31	M	01:06:32	02:13:59	3992.0	8039.0

To get an idea of what the data looks like, we can plot a jointplot over the data(Figure):

```
In[28]: with sns.axes_style('white'):

            g = sns.jointplot("split_sec", "final_sec", data, kind='hex')
            g.ax_joint.plot(np.linspace(4000, 16000),

                            np.linspace(8000, 32000), ':k')
```

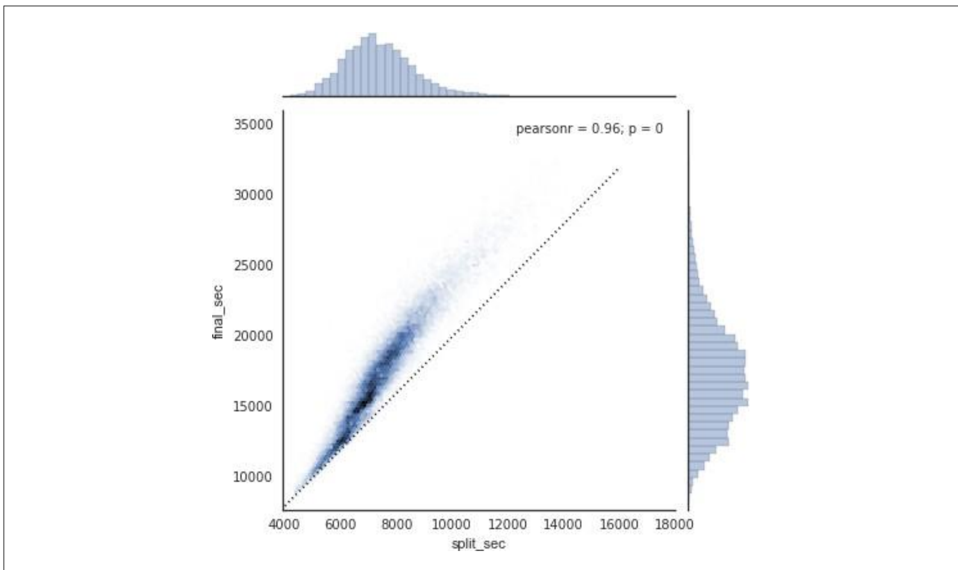


Figure 4-126. The relationship between the split for the first half-marathon and the fin-ishing time for the full marathon

The dotted line shows where someone's time would lie if they ran the marathon at a perfectly steady pace. The fact that the distribution lies above this indicates (as you might expect) that most people slow down over the course of the marathon. If you have run competitively, you'll know that those who do the opposite—run faster dur- ing the second half of the race—are said to have "negative-split" the race.

Let's create another column in the data, the split fraction, which measures the degree to which each runner negative-splits or positive-splits the race:

```
In[29]: data['split_frac'] = 1 - 2 * data['split_sec'] / data['final_sec']
        data.head()
```

Out[29]:	age	gender	split	final	split_sec	final_sec	split_frac
0	33	M	01:05:38	02:08:51	3938.0	7731.0	-0.01875

3				6
1	M 01:06:26 02:09:28	3986.0	7768.0	-
3				0.02626
2				2
2	M 01:06:49 02:10:42	4009.0	7842.0	-
3				0.02244
1				3
3	M 01:06:16 02:13:45	3976.0	8025.0	0.0090
3				97
8				
4	M 01:06:32 02:13:59	3992.0	8039.0	0.0068
3				42
1				

Where this split difference is less than zero, the person negative-split the race by thatfraction. Let's do a distribution plot of this split fraction (Figure):

```
In[30]: sns.distplot(data['split_frac'], kde=False);
        plt.axvline(0, color="k", linestyle="--");
```

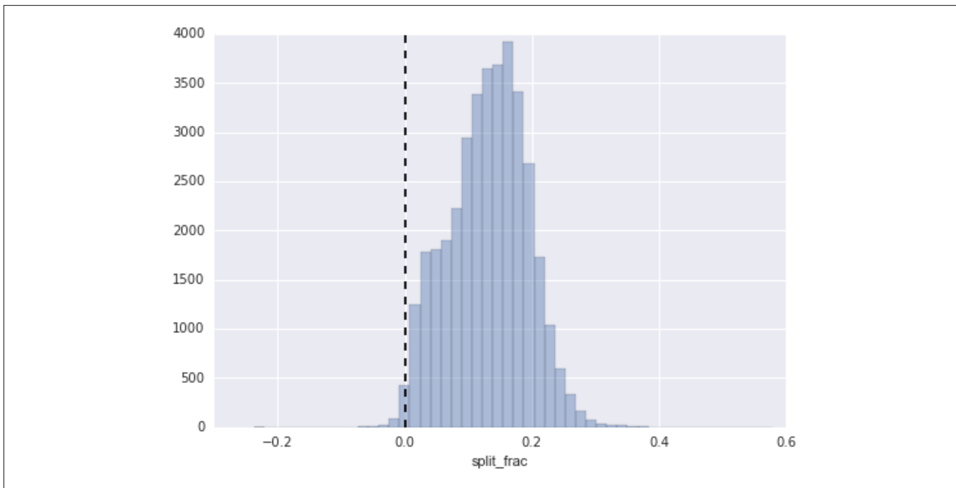


Figure . The distribution of split fractions; 0.0 indicates a runner who completedthe first and second halves in identical times

```
In[31]: sum(data.split_frac <
0)Out[31]: 251
```

Out of nearly 40,000 participants, there were only 250 people who negative-split their marathon.

Let’s see whether there is any correlation between this split fraction and other vari- ables. We’ll do this using a pairgrid, which draws plots of all these correlations (Figure):

```
In[32]:

g = sns.PairGrid(data, vars=['age', 'split_sec', 'final_sec', 'split_frac'],
                 hue='gender', palette='RdBu_r')

g.map(plt.scatter,
alpha=0.8)g.add_legend();
```

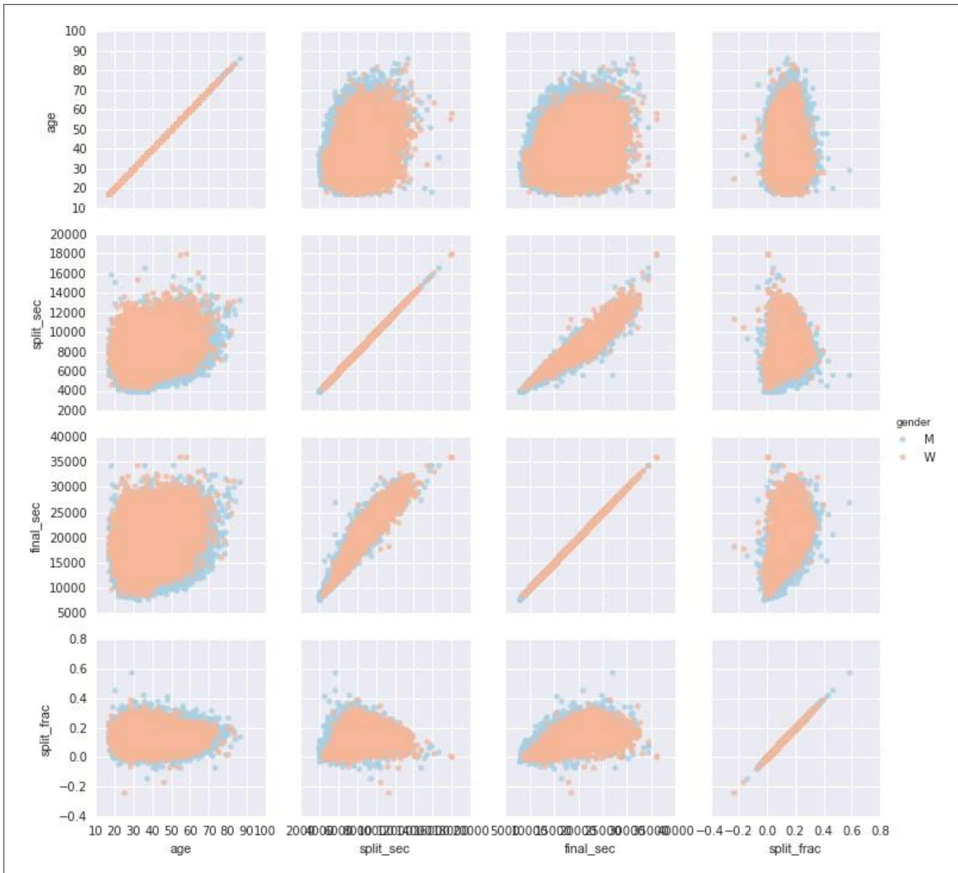


Figure . The relationship between quantities within the marathon dataset

It looks like the split fraction does not correlate particularly with age, but does corre- late with the final time: faster runners tend to have closer to even splits on their mara-thon time. (We see here that Seaborn is no panacea for Matplotlib’s ills when it comesto plot styles: in particular, the *x*-axis labels overlap. Because the output is a simple Matplotlib plot, however, the methods in “Customizing Ticks” can be used to adjust such things if

desired.)

The difference between men and women here is interesting. Let's look at the histo- gram of split fractions for these two groups (Figure):

```
In[33]: sns.kdeplot(data.split_frac[data.gender=='M'], label='men',
        shade=True) sns.kdeplot(data.split_frac[data.gender=='W'],
        label='women', shade=True)plt.xlabel('split_frac');
```

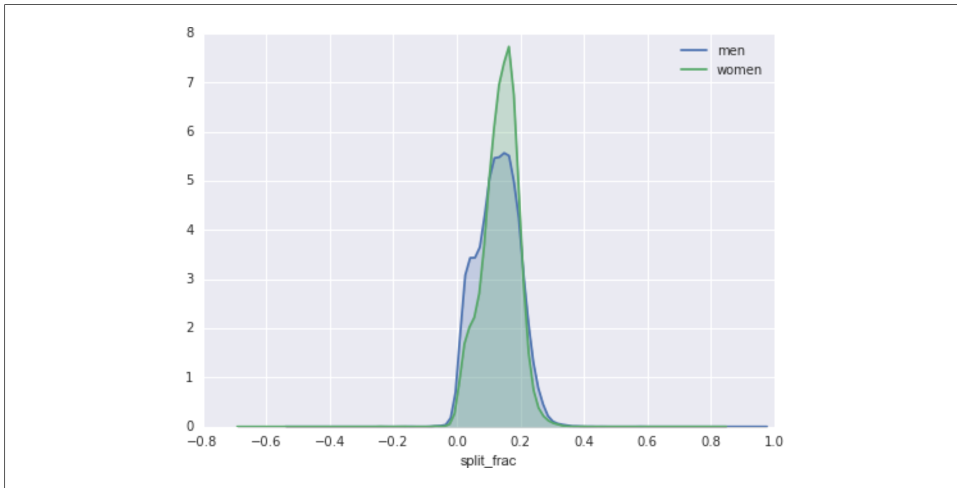


Figure . The distribution of split fractions by gender

The interesting thing here is that there are many more men than women who are running close to an even split! This almost looks like some kind of bimodal distribu- tion among the men and women. Let's see if we can suss out what's going on by look-ing at the distributions as a function of age.

A nice way to compare distributions is to use a *violin plot* (Figure :

```
In[34]:
sns.violinplot("gender", "split_frac", data=data,
palette=["lightblue", "lightpink"]);
```

Figure . A violin plot showing the split fraction by gender

This is yet another way to compare the distributions between men and women.

Let's look a little deeper, and compare these violin plots as a function of age. We'll start by creating a new column in the array that specifies the decade of age that each person is in (Figure):

```
In[35]: data['age_dec'] = data.age.map(lambda age: 10 * (age //
10))data.head()
```

Out[35]:

	age	gender	split	final	split_sec	final_sec	split_frac	age_dec
0	33	M	01:05:38	02:08:51	3938.0	7731.0	-0.018756	30
1	32	M	01:06:26	02:09:28	3986.0	7768.0	-0.026262	30
2	31	M	01:06:49	02:10:42	4009.0	7842.0	-0.022443	30
3	38	M	01:06:16	02:13:45	3976.0	8025.0	0.009097	30
4	31	M	01:06:32	02:13:59	3992.0	8039.0	0.006842	30

```
In[36]:

men = (data.gender == 'M')
women = (data.gender ==
'W')
```

```
with sns.axes_style(style=None):

    sns.violinplot("age_dec", "split_frac", hue="gender",
                   data=data,split=True, inner="quartile",
                   palette=["lightblue", "lightpink"]);
```

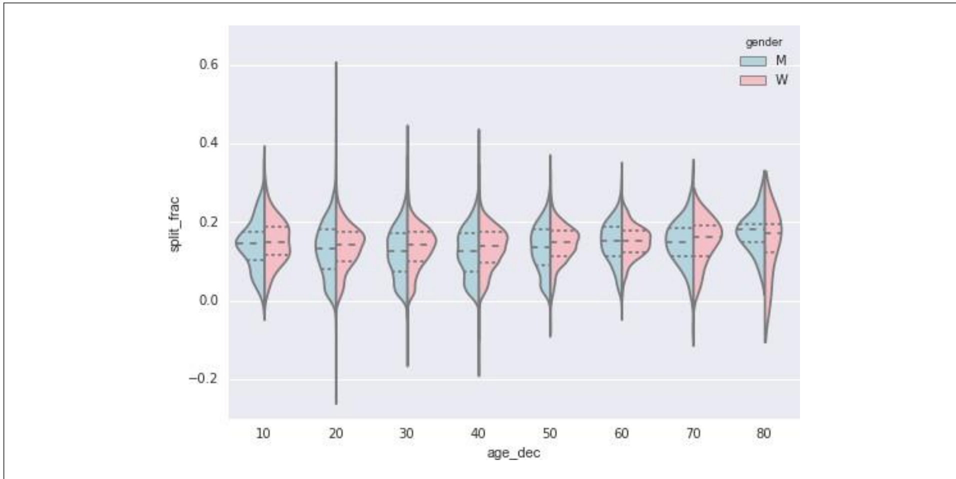


Figure . A violin plot showing the split fraction by gender and age

Looking at this, we can see where the distributions of men and women differ: the split distributions of men in their 20s to 50s show a pronounced over-density toward lower splits when compared to women of the same age (or of any age, for that matter).

Also surprisingly, the 80-year-old women seem to outperform *everyone* in terms of their split time. This is probably due to the fact that we're estimating the distribution from small numbers, as there are only a handful of runners in that range:

```
In[38]: (data.age >

80).sum()Out[38]: 7
```

Back to the men with negative splits: who are these runners? Does this split fraction correlate with finishing quickly? We can plot this very easily. We'll use regplot, which will automatically fit a linear regression to the data (Figure):

```
In[37]: g = sns.lmplot('final_sec', 'split_frac', col='gender', data=data,
```

```
                        markers=".",
scatter_kws=dict(color='c'))g.map(plt.axhline, y=0.1,
color="k", ls=":");
```

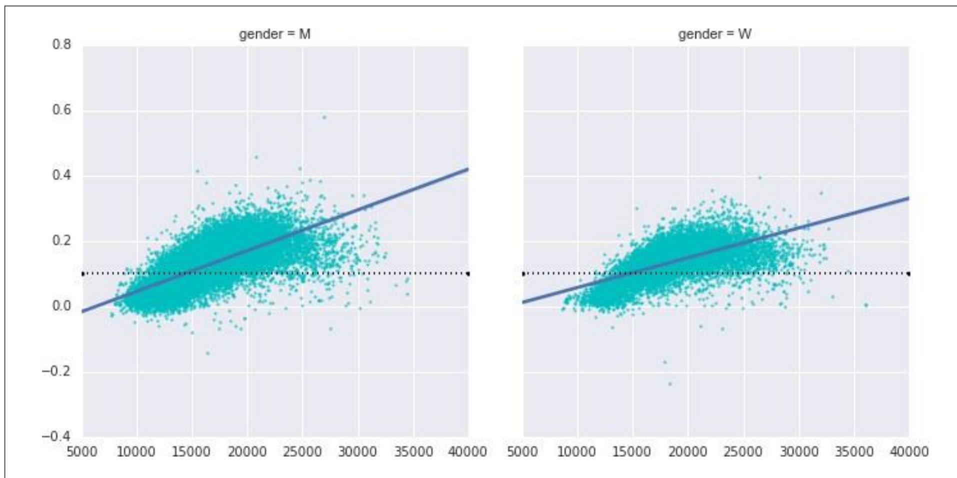


Figure . Split fraction versus finishing time by gender

Apparently the people with fast splits are the elite runners who are finishing within ~15,000 seconds, or about 4 hours. People slower than that are much less likely to have a fast second split.

Day 02- Data Visualization on World Map- Geographic Data with Basemap

One common type of visualization in data science is that of geographic data. Matplot-lib's main tool for this type of visualization is the Basemap toolkit, which is one of several Matplotlib toolkits that live under the mpl_toolkits namespace. Admittedly,Basemap feels a bit clunky to use, and often even simple visualizations take much longer to render than you might hope. More modern solutions, such as leaflet or the Google Maps API, may be a better choice for more intensive map visualizations. Still,Basemap is a useful tool for Python users to have in their virtual toolbelts. In this sec-tion, we'll show several examples of the type of map visualization that is possible withthis toolkit.

Installation of Basemap is straightforward; if you're using conda you can type this andthe package will be downloaded:

```
$ conda install basemap
```

We add just a single new import to our standard boilerplate:

```
In[1]: %matplotlib inline
       import numpy as np
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
```

Once you have the Basemap toolkit installed and imported, geographic plots are justa few lines away (the graphics in Figure also require the PIL package in Python2, or the pillowpackage in Python 3):

```
In[2]: plt.figure(figsize=(8, 8))
    m = Basemap(projection='ortho', resolution=None, lat_0=50,
    lon_0=-100)m.bluemarble(scale=0.5);
```

Figure. A "bluemarble" projection of the Earth

The meaning of the arguments to Basemap will be discussed momentarily.

The useful thing is that the globe shown here is not a mere image; it is a fully functioning Matplotlib axes that understands spherical coordinates and allows us to easily over-plot data on the map! For example, we can use a different map projection, zoomin to North America, and plot the location of Seattle. We'll use an etopo image (whichshows topographical features both on land and under the ocean) as the map back- ground (Figure):

```
In[3]: fig = plt.figure(figsize=(8, 8))
```

```
m = Basemap(projection='lcc',
            resolution=None,width=8E6,
            height=8E6,

            lat_0=45, lon_0=-
100,)m.etopo(scale=0.5,
alpha=0.5)

# Map (long, lat) to (x, y) for plotting
x, y = m(-122.3, 47.6)

plt.plot(x, y, 'ok', markersize=5)
plt.text(x, y, ' Seattle', fontsize=12);
```

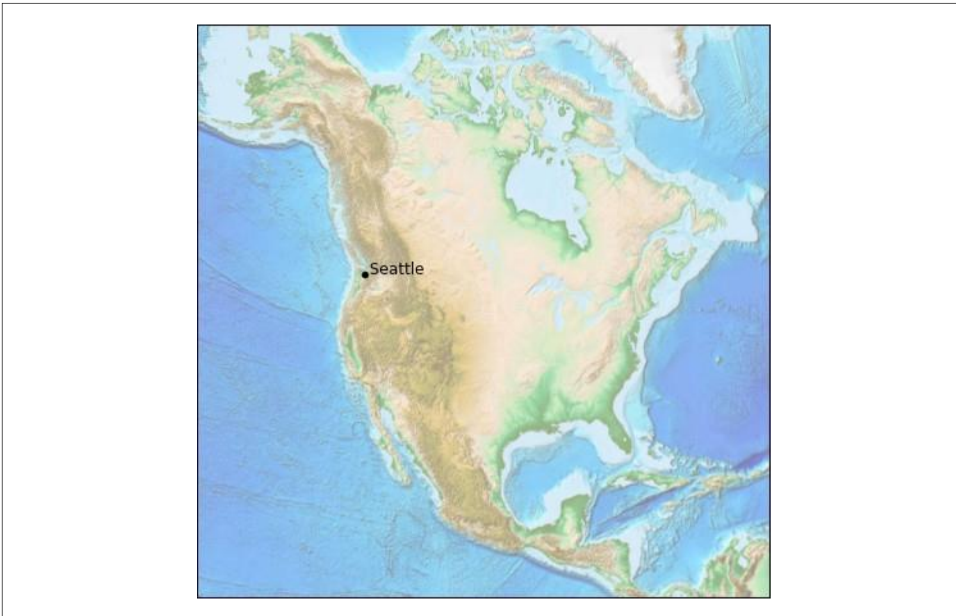


Figure. Plotting data and labels on the map

This gives you a brief glimpse into the sort of geographic visualizations that are possi-ble with just a few lines of Python. We'll now discuss the features of Basemap in moredepth, and provide several examples of visualizing map data. Using these brief exam-ples as building blocks, you should be able to create nearly any map visualization thatyou desire.

Map Projections

The first thing to decide when you are using maps is which projection to use. You're

probably familiar with the fact that it is impossible to project a spherical map, such asthat of the Earth, onto a flat surface without somehow distorting it or breaking its continuity. These projections have been developed over the course of human history, and there are a lot of choices! Depending on the intended use of the map projection, there are certain map features (e.g., direction, area, distance, shape, or other consider-ations) that are useful to maintain.

The Basemap package implements several dozen such projections, all referenced by ashort format code. Here we'll briefly demonstrate some of the more common ones.

We'll start by defining a convenience routine to draw our world map along with the longitude and latitude lines:

```
In[4]: from itertools import chain

       def draw_map(m, scale=0.2):
           # draw a shaded-relief image
           m.shadedrelief(scale=scale)
           # lats and longs are returned as a dictionary
           lats = m.drawparallels(np.linspace(-90, 90, 13))
           lons = m.drawmeridians(np.linspace(-180, 180, 13))
           # keys contain the plt.Line2D instances
           lat_lines = chain(*(tup[1][0] for tup in lats.items()))
           lon_lines = chain(*(tup[1][0] for tup in lons.items()))
           all_lines = chain(lat_lines, lon_lines)
           # cycle through these lines and set the desired style
           for line in all_lines:
               line.set(linestyle='-', alpha=0.3, color='w')
```

Cylindrical projections

The simplest of map projections are cylindrical projections, in which lines of constant latitude and longitude are mapped to horizontal and vertical lines, respectively. This type of mapping represents equatorial regions quite well, but results in extreme dis- tortions near the poles. The spacing of latitude lines varies between different cylindri-cal projections, leading to different conservation properties, and different distortion near the poles. In Figure 4-104, we show an example of the *equidistant cylindrical pro-jection*, which

chooses a latitude scaling that preserves distances along meridians. Other cylindrical projections are the Mercator (projection='merc') and the cylin- drical equal-area (projection='cea') projections.

```
In[5]: fig = plt.figure(figsize=(8, 6), edgecolor='w')m
        = Basemap(projection='cyl',
        resolution=None,

                llcrnrlat=-90, urcrnrlat=90,
                llcrnrlon=-180, urcrnrlon=180,
                )draw_map(m)
```

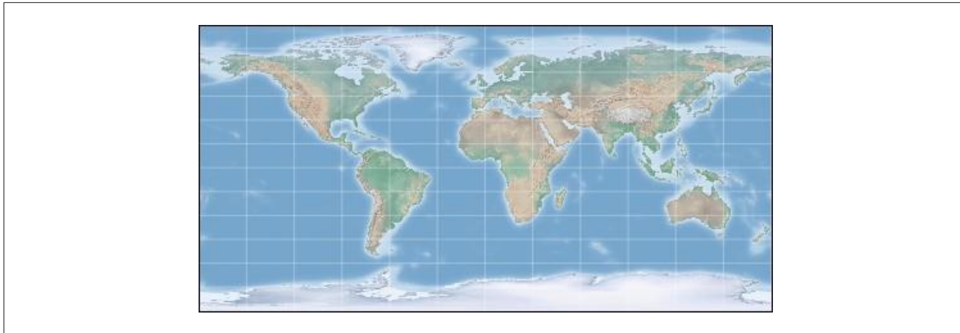


Figure. Cylindrical equal-area projection

The additional arguments to Basemap for this view specify the latitude (lat) and lon- gitude (lon) of the lower-left corner (llcrnr) and upper-right corner (urcrnr) for thedesired map, in units of degrees.

Pseudo-cylindrical projections

Pseudo-cylindrical projections relax the requirement that meridians (lines of constant longitude) remain vertical; this can give better properties near the poles of the projec- tion. The Mollweide projection (projection='moll') is one common example ofthis, in which all meridians are elliptical arcs (Figure). It is constructed so as topreserve area across the map: though there are distortions near the poles, the area of small patches reflects the true area. Other pseudo-cylindrical projections are the sinusoidal (projection='sinu') and Robinson (projection='robin') projections.

```
In[6]: fig = plt.figure(figsize=(8, 6), edgecolor='w')m
        = Basemap(projection='moll',
        resolution=None,

                lat_0=0, lon_0=0)
```

```
draw_map(m)
```

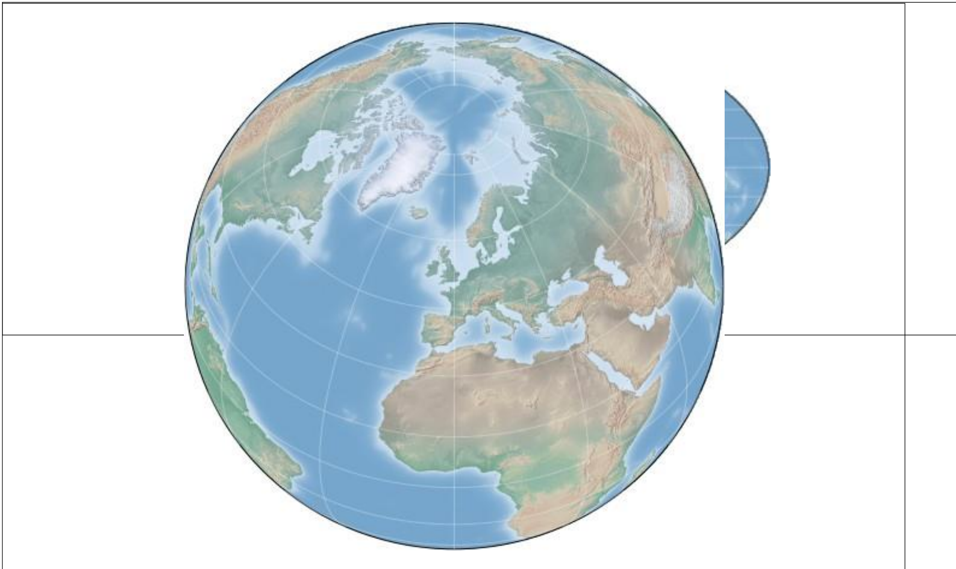


Figure. The Molleweide projection

The extra arguments to Basemap here refer to the central latitude (lat_0) and longi- tude (lon_0) for the desired map.

Perspective projections

Perspective projections are constructed using a particular choice of perspective point, similar to if you photographed the Earth from a particular point in space (a point which, for some projections, technically lies within the Earth!). One common exam- ple is the orthographic projection (projection='ortho'), which shows one side of the globe as seen from a viewer at a very long distance. Thus, it can show only half theglobe at a time. Other perspective-based projections include the gnomonic projection (projection='gnom') and stereographic projection (projection='stere'). These areoften the most useful for showing small portions of the map.

Here is an example of the orthographic projection:

```
In[7]: fig = plt.figure(figsize=(8, 8))

    m = Basemap(projection='ortho',
            resolution=None,lat_0=50,
            lon_0=0)draw_map(m);
```

Figure. The orthographic projection

Conic projections

A conic projection projects the map onto a single cone, which is then unrolled. This can lead to very good local properties, but regions far from the focus point of thecone may become very distorted. One example of this is the Lambert conformal conic projection (projection='lcc'), which we saw earlier in the map of North America.It projects the map onto a cone arranged in such a way that two standard parallels (specified in Basemap by lat_1 and lat_2) have well-represented distances, with scale decreasing between them and increasing outside of them. Other useful conic projec- tions are the equidistant conic (projection='eqdc') and the Albers equal-area (pro jection='aea') projection (Figure 4-107). Conic projections, like perspective projections, tend to be good choices for representing small to medium patches of the globe.

```
In[8]: fig = plt.figure(figsize=(8, 8))

       m = Basemap(projection='lcc',
                   resolution=None, lon_0=0,
                   lat_0=50, lat_1=45,
                   lat_2=55,width=1.6E7,
                   height=1.2E7)

       draw_map(m)
```

Figure. The Albers equal-area projection

Other projections

If you’re going to do much with map-based visualizations, I encourage you to read upon other available projections, along with their properties, advantages, and disadvan- tages. Most likely, they are available in the Basemap package. If you dig deep enoughinto this topic, you’ll find an incredible subculture of geo-viz geeks who will be readyto argue fervently in support of their favorite projection for any given application!

Drawing a Map Background

Earlier we saw the bluemarble()and shadedrelief()methods for projecting globalimages on the map, as well as the drawparallels() and drawmeridians() methods for drawing lines of constant latitude and longitude. The Basemap package contains arange of useful functions for drawing borders of physical features like continents, oceans, lakes, and rivers, as well as political boundaries such as countries and US states and counties. The following are some of the available drawing functions that you may wish to explore using IPython's help features:

- Physical boundaries and bodies of water

  drawcoastlines()

  Draw continental coast lines

  drawlsmask()

  Draw a mask between the land and sea, for use with projecting images onone or the other drawmapboundary()
  Draw the map boundary, including the fill color for oceans

  drawrivers()

  Draw rivers on the map

  fillcontinents()

  Fill the continents with a given color; optionally fill lakes with another color

- Political boundaries

  drawcountries()

  Draw country boundaries

  drawstates()

  Draw US state boundaries

  drawcounties()

  Draw US county boundaries

- Map features

  drawgreatcircle()

  Draw a great circle between two points

drawparallels()

Draw lines of constant latitude

drawmeridians()

Draw lines of constant longitude

drawmapscale()

Draw a linear scale on the map

- Whole-globe images

bluemarble()

Project NASA's blue marble image onto the map

shadedrelief()

Project a shaded relief image onto the map

etopo()

Draw an etopo relief image onto the map

warpimage()

Project a user-provided image onto the mapFor the boundary-based features, you must set the desired resolution when creating a Basemap image. The resolution argument of the Basemap class sets the level of detail in boundaries, either 'c'(crude), 'l'(low), 'i'(intermediate), 'h'(high), 'f'(full),or None if no boundaries will be used. This choice is important: setting high- resolution boundaries on a global map, for example, can be *very* slow.

Here's an example of drawing land/sea boundaries, and the effect of the resolution parameter. We'll create both a low- and high-resolution map of Scotland's beautiful Isle of Skye. It's located at 57.3°N, 6.2°W, and a map of 90,000×120,000 kilometers shows it well (Figure):

```
In[9]: fig, ax = plt.subplots(1, 2, figsize=(12, 8))
    for i, res in enumerate(['l', 'h']):
```

```
m = Basemap(projection='gnom', lat_0=57.3, lon_0=-6.2,
            width=90000, height=120000, resolution=res,
```

```
            ax=ax[i])m.fillcontinents(color="#FFDDCC",
            lake_color='#DDEEFF')
            m.drawmapboundary(fill_color="#DDEEFF")
            m.drawcoastlines()
            ax[i].set_title("resolution='{0}'".format(res));
```

Figure. Map boundaries at low and high resolution

Notice that the low-resolution coastlines are not suitable for this level of zoom, while high-resolution works just fine. The low level would work just fine for a global view, however, and would be *much* faster than loading the high-resolution border data for the entire globe! It might require some experimentation to find the correct resolution parameter for a given view; the best route is to start with a fast, low-resolution plot and increase the resolution as needed.

Plotting Data on Maps

Perhaps the most useful piece of the Basemap toolkit is the ability to over-plot a vari-ety of data onto a map background. For simple plotting and text, any plt function works on the map; you can use the Basemap instance to project latitude and longitudecoordinates to (x, y)coordinates for plotting with plt, as we saw earlier in the Seat-tle example.

In addition to this, there are many map-specific functions available as methods of the Basemap instance. These work very similarly to their standard Matplotlib counter- parts, but have an additional Boolean argument latlon, which if set to True allows you to pass raw latitudes and longitudes to the method, rather than projected (x, y) coordinates.

Some of these map-specific methods are:

contour()/contourf()

Draw contour lines or filled contours

imshow()

Draw an image

pcolor()/pcolormesh()

Draw a pseudocolor plot for irregular/regular meshes

plot()

Draw lines and/or markers

scatter()

Draw points with markers

quiver()

Draw vectors

barbs()

Draw wind barbs

drawgreatcircle()

Draw a great circle

We'll see examples of a few of these as we continue. For more information on these functions, including several example plots, see the online Basemap documentation.

Example: California Cities

Recall that in "Customizing Plot Legends" on page 249, we demonstrated the use of size and color in a scatter plot to convey information about the location, size, and population of California cities. Here, we'll create this plot again, but using Basemap toput the data in context.

We start with loading the data, as we did before:

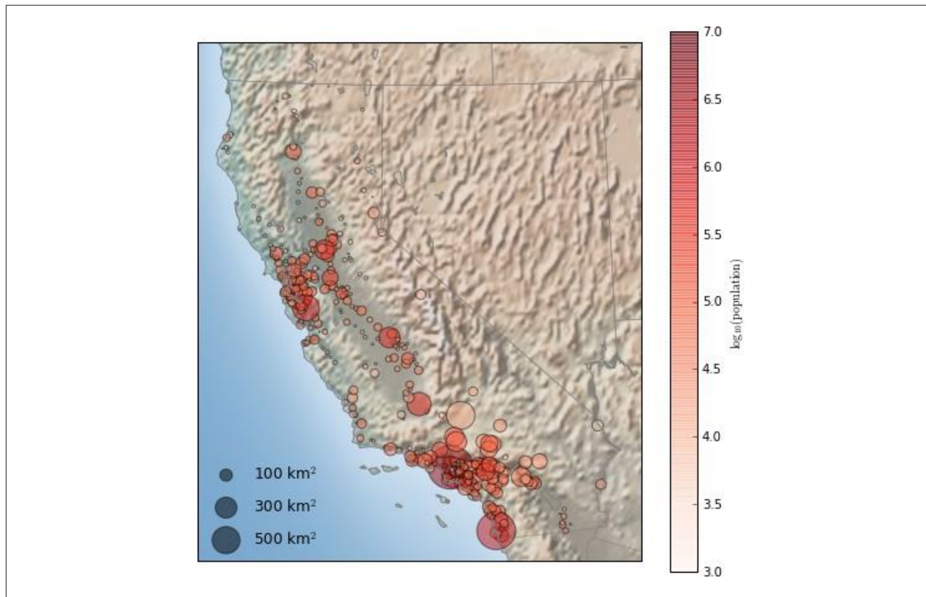
```
In[10]: import pandas as pd
        cities = pd.read_csv('data/california_cities.csv')
```

```
# Extract the data we're interested in

lat = cities['latd'].values
lon = cities['longd'].values

population =
cities['population_total'].valuesarea =
cities['area_total_km2'].values
```

Next, we set up the map projection, scatter the data, and then create a colorbar and legend:

```
In[11]:# 1. Draw the map background

        fig = plt.figure(figsize=(8, 8))

        m = Basemap(projection='lcc',
                    resolution='h',lat_0=37.5,
                    lon_0=-119, width=1E6,
                    height=1.2E6)

        m.shadedrelief()
        m.drawcoastlines(color='gra
        y')
        m.drawcountries(color='gra
        y')
        m.drawstates(color='gray')

        # 2. scatter city data, with color reflecting
        population# and size reflecting area

        m.scatter(lon, lat, latlon=True,

                  c=np.log10(population),
                  s=area,cmap='Reds',
                  alpha=0.5)

        # 3. create colorbar and legend
        plt.colorbar(label=r'$\log_{10}({\rm
        population})$')plt.clim(3, 7)

        # make legend with dummy points

        for a in [100, 300, 500]:

            plt.scatter([], [], c='k', alpha=0.5, s=a,
                        label=str(a) + ' km$^2$')
```

```
plt.legend(scatterpoints=1,
           frameon=False, labelspacing=1,
```




```
           loc='lower left');
```

Figure. Scatter plot over a map background

This shows us roughly where larger populations of people have settled in California: they are clustered near the coast in the Los Angeles and San Francisco areas, stretched along the highways in the flat central valley, and avoiding almost completelythe mountainous regions along the borders of the state.

Example: Surface Temperature Data

As an example of visualizing some more continuous geographic data, let's consider the "polar vortex" that hit the eastern half of the United States in January 2014. A great source for any sort of climatic data is NASA's Goddard Institute for Space Stud-ies. Here we'll use the GIS 250 temperature data, which we can download using shellcommands (these commands may have to be modified on Windows machines). The data used here was downloaded on 6/12/2016, and the file size is approximately 9 MB:

```
In[12]: # !curl  -O
        http://data.giss.nasa.gov/pub/gistemp/gistemp250.nc.gz#
        !gunzip gistemp250.nc.gz
```

The data comes in NetCDF format, which can be read in Python by the netCDF4 library. You can install this library as shown here:

```
$ conda install netcdf4
```

We read the data as follows:

```
In[13]: from netCDF4 import
        Dataset data =
        Dataset('gistemp250.nc')
```

The file contains many global temperature readings on a variety of dates; we need to select the index of the date we're interested in—in this case, January 15, 2014:

```
In[14]: from netCDF4 import date2index
        from datetime import datetime
        timeindex = date2index(datetime(2014, 1, 15),
                               data.variables['time'])
```

Now we can load the latitude and longitude data, as well as the temperature anomaly for this index:

```
In[15]: lat = data.variables['lat'][:] lon
        = data.variables['lon'][:] lon,
        lat = np.meshgrid(lon, lat)

        temp_anomaly = data.variables['tempanomaly'][timeindex]
```

Finally, we'll use the pcolormesh() method to draw a color mesh of the data. We'lllook at North America, and use a shaded relief map in the background. Note that for this data we specifically chose a divergent colormap, which has a neutral color at zeroand two contrasting colors at negative and positive values (Figure). We'll also lightly draw the coastlines over the colors for reference:

```
In[16]: fig = plt.figure(figsize=(10, 8))

        m = Basemap(projection='lcc',
                    resolution='c',width=8E6,
                    height=8E6,

                    lat_0=45, lon_0=-100,)
        m.shadedrelief(scale=0.5)
        m.pcolormesh(lon, lat,
        temp_anomaly,

                     latlon=True,
        cmap='RdBu_r')plt.clim(-8, 8)
        m.drawcoastlines(color='lightgray')
```

```
plt.title('January 2014 Temperature Anomaly')
plt.colorbar(label='temperature  anomaly  (°C)');
```

The data paints a picture of the localized, extreme temperature anomalies that happened during that month. The eastern half of the United States was much colder than normal, while the western half and Alaska were much warmer. Regions with no recorded temperature show the map background.

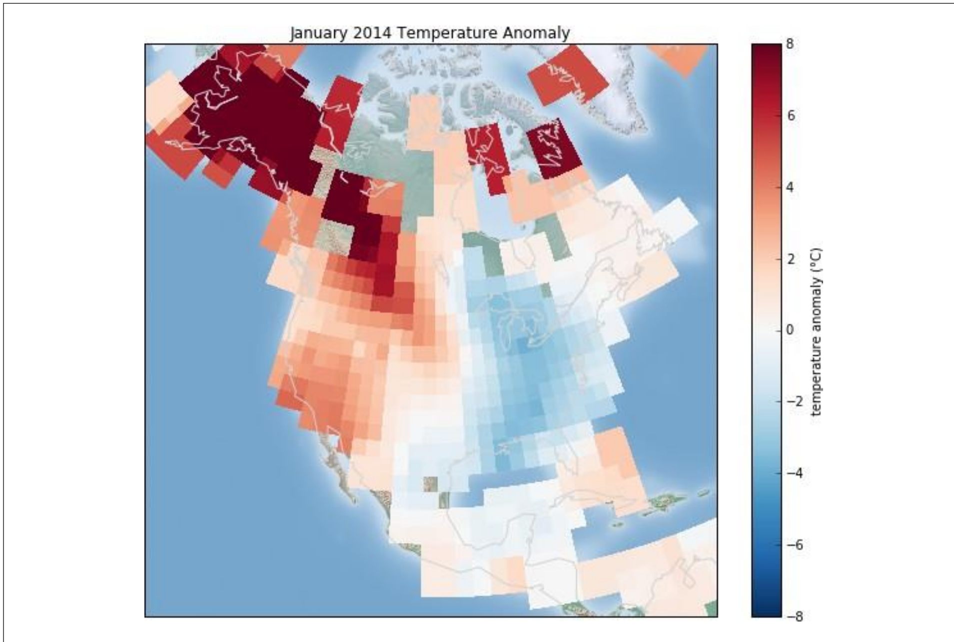


Figure. The temperature anomaly in January 2014

Day-03: Visualization using google maps and ArcGis(Iris Data)

Day-04: Discussion on projects and exploring other datasets

Day-05: Mid Assessments

Week 6: Advance Data Analytics

Day-01: Hyperparameters and Model Validation

In the previous section, we saw the basic recipe for applying a supervised machine learning model:

1. Choose a class of model.
2. Choose model hyperparameters.
3. Fit the model to the training data
4. Use the model to predict labels for new data

The first two pieces of this—the choice of model and choice of hyperparameters—areperhaps the most important part of using these tools and techniques effectively. In order to make an informed choice, we need a way to *validate* that our model and our hyperparameters are a good fit to the data. While this may sound simple, there are some pitfalls that you must avoid to do this effectively.

Thinking About Model Validation

In principle, model validation is very simple: after choosing a model and its hyper-parameters, we can estimate how effective it is by applying it to some of the training data and comparing the prediction to the known value.

The following sections first show a naive approach to model validation and why it fails, before exploring the use of holdout sets and cross-validation for more robust model evaluation.

Model validation the wrong way

Let's demonstrate the naive approach to validation using the Iris data, which we saw in the previous section. We will start by loading the data:

```
In[1]: from sklearn.datasets import load_iris
       iris = load_iris()

       X = iris.data

       y = iris.target
```

Next we choose a model and hyperparameters. Here we'll use a *k*-neighbors classifierwith `n_neighbors=1`. This is a very simple and intuitive model that says "the label of an unknown point is the same as the label of its closest training point":

```
In[2]: from sklearn.neighbors import KNeighborsClassifier
       model = KNeighborsClassifier(n_neighbors=1)
```

Then we train the model, and use it to predict labels for data we already know:

```
In[3]: model.fit(X, y)

       y_model = model.predict(X)
```

Finally, we compute the fraction of correctly labeled points:

```
In[4]: from sklearn.metrics import accuracy_score
       accuracy_score(y, y_model)

Out[4]: 1.0
```

We see an accuracy score of 1.0, which indicates that 100% of points were correctly labeled by our model! But is this truly measuring the expected accuracy? Have we really come upon a model that we expect to be correct 100% of the time?

As you may have gathered, the answer is no. In fact, this approach contains a funda- mental flaw: *it trains and evaluates the model on the same data*. Furthermore, the nearest neighbor model is an *instance-based* estimator that simply stores the training data, and predicts labels by comparing new data to these stored points; except in con-trived cases, it will get 100% accuracy *every time!*

Model validation the right way: Holdout sets

So what can be done? We can get a better sense of a model's performance using what's known as a *holdout set*; that is, we hold back some subset of the data from the trainingof the model, and then use this holdout set to check the model performance. We can do this splitting using the `train_test_split` utility in Scikit-Learn:

```
In[5]: from sklearn.cross_validation import train_test_split

       # split the data with 50% in each set

       X1, X2, y1, y2 = train_test_split(X, y, random_state=0,

                                         train_size=0.5)


       # fit the model on one set of data

       model.fit(X1, y1)


       # evaluate the model on the second set of data
       y2_model = model.predict(X2)
       accuracy_score(y2, y2_model)

Out[5]: 0.90666666666666662
```

We see here a more reasonable result: the nearest-neighbor classifier is about 90% accurate on this holdout set. The holdout set is similar to unknown data, because the model has not "seen" it before.

Model validation via cross-validation

One disadvantage of using a holdout set for model validation is that we have lost a portion of

our data to the model training. In the previous case, half the dataset does not contribute to the training of the model! This is not optimal, and can cause prob- lems—especially if the initial set of training data is small.

One way to address this is to use *cross-validation*—that is, to do a sequence of fits where each subset of the data is used both as a training set and as a validation set. Visually, it might look something like Figure 5-22.

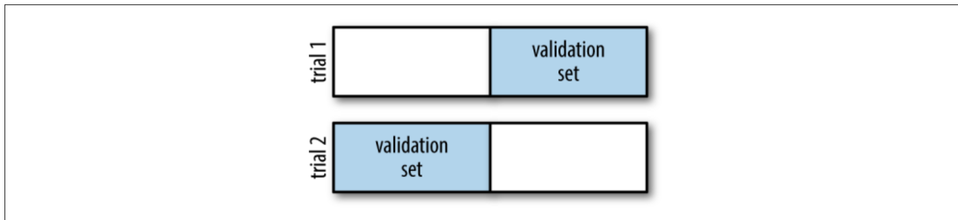


Figure 5-22. Visualization of two-fold cross-validation

Here we do two validation trials, alternately using each half of the data as a holdout set. Using the split data from before, we could implement it like this:

```
In[6]: y2_model = model.fit(X1, y1).predict(X2)
       y1_model = model.fit(X2, y2).predict(X1)

       accuracy_score(y1, y1_model), accuracy_score(y2, y2_model)

Out[6]: (0.95999999999999996, 0.90666666666666662)
```

What comes out are two accuracy scores, which we could combine (by, say, taking themean) to get a better measure of the global model performance. This particular form of cross-validation is a *two-fold cross-validation*—one in which we have split the datainto two sets and used each in turn as a validation set.

We could expand on this idea to use even more trials, and more folds in the data—forexample, Figure 5-23 is a visual depiction of five-fold cross-validation.

trial 1
validation set
trial 2
validation set
trial 3
validation set
trial 4
validation set
trial 5
validation set

*Figure 5-23. Visualization of **five-fold** cross-validation*

Here we split the data into five groups, and use each of them in turn to evaluate the model fit on the other 4/5 of the data. This would be rather tedious to do by hand,and so we can use Scikit-Learn's `cross_val_score` convenience routine to do it succinctly:

```
In[7]: from sklearn.cross_validation import cross_val_score
       cross_val_score(model, X, y, cv=5)
```

```
Out[7]: array([ 0.96666667,  0.96666667,  0.93333333,  0.93333333,  1.        ])
```

Repeating the validation across different subsets of the data gives us an even better idea of the performance of the algorithm.

Scikit-Learn implements a number of cross-validation schemes that are useful in par-ticular situations; these are implemented via iterators in the `cross_validation` mod-ule. For example, we might wish to go to the extreme case in which our number of folds is equal to the number of data points; that is, we train on all points but one in each trial. This type of cross-validation is known as *leave-one-out* cross-validation, and can be used as follows:

```
In[8]: from sklearn.cross_validation import LeaveOneOut

       scores = cross_val_score(model, X, y, cv=LeaveOneOut(len(X)))
       scores
```

```
Out[8]: array([  1.,  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
1.,               ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,
1.,          1.,  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
                  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,
1.,          1.,  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
                  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,
1.,          1.,  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
                  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,
1.,          1.,  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
                  ,   ,   ,   ,   ,   ,   ,   ,   ,   ,   ,
1.,          1.,  1.  1.  1.  0.  1.  0.  1.  1.  1.  1.  1.
```

```
1., 1., 1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1.])
```

Because we have 150 samples, the leave-one-out cross-validation yields scores for 150trials, and the score indicates either successful (1.0) or unsuccessful (0.0) prediction. Taking the mean of these gives an estimate of the error rate:

```
In[9]: scores.mean()

Out[9]: 0.95999999999999996
```

Other cross-validation schemes can be used similarly. For a description of what is available in Scikit-Learn, use IPython to explore the `sklearn.cross_validation` sub- module, or take a look at Scikit-Learn's online cross-validation documentation.

Selecting the Best Model

Now that we've seen the basics of validation and cross-validation, we will go into a little more depth regarding model selection and selection of hyperparameters. These issues are some of the most important aspects of the practice of machine learning,and I find that this information is often glossed over in introductory machine learn- ing tutorials.

Of core importance is the following question: *if our estimator is underperforming, how should we move forward?* There are several possible answers:

- Use a more complicated/more flexible model
- Use a less complicated/less flexible model
- Gather more training samples
- Gather more data to add features to each sample

The answer to this question is often counterintuitive. In particular, sometimes using amore complicated model will give worse results, and adding more training samples may not improve your results! The ability to determine what steps will improve your model is what separates the successful machine learning practitioners from the unsuccessful.

The bias–variance trade-off

Fundamentally, the question of "the best model" is about finding a sweet spot in the trade-off between *bias* and *variance*. Consider Figure 5-24, which presents tworegression fits to the same dataset.

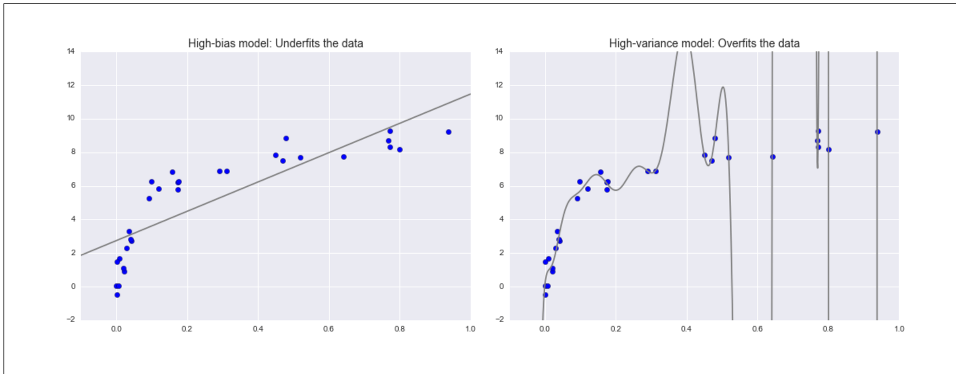


Figure 5-24. A high-bias and high-variance regression model

It is clear that neither of these models is a particularly good fit to the data, but they fail in different ways.

The model on the left attempts to find a straight-line fit through the data. Because thedata are intrinsically more complicated than a straight line, the straight-line model will never be able to describe this dataset well. Such a model is said to *underfit* the data; that is, it does not have enough model flexibility to suitably account for all the features in the data. Another way of saying this is that the model has high *bias*.

The model on the right attempts to fit a high-order polynomial through the data.Here the model fit has enough flexibility to nearly perfectly account for the fine fea- tures in the data, but even though it very accurately describes the training data, its precise form seems to be more reflective of the particular noise properties of the data rather than the intrinsic properties of whatever process generated that data. Such a model is said to *overfit* the data; that is, it has so much model flexibility that the model ends up accounting for random errors as well as the underlying data distribu- tion. Another way of saying this is that the model has high *variance*.

To look at this in another light, consider what happens if we use these two models to predict the *y*-value for some new data. In diagrams in Figure 5-25, the red/lighter points indicate data that is omitted from the training set.

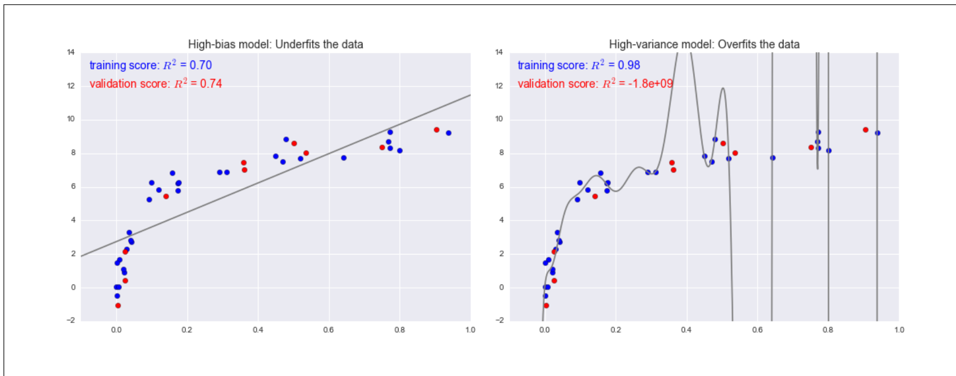


Figure 5-25. Training and validation scores in high-bias and high-variance models

The score here is the R^2 score, or coefficient of determination, which measures how well a model performs relative to a simple mean of the target values. $R^2 = 1$ indicates a perfect match, $R^2 = 0$ indicates the model does no better than simply taking the mean of the data, and negative values mean even worse models. From the scores asso-ciated with these two models, we can make an observation that holds more generally:

- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set.
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set.

If we imagine that we have some ability to tune the model complexity, we would expect the training score and validation score to behave as illustrated in Figure 5-26.

The diagram shown in Figure 5-26 is often called a *validation curve*, and we see the following essential features:

- The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
- For very low model complexity (a high-bias model), the training data is underfit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
- For very high model complexity (a high-variance model), the training data is overfit, which means that the model predicts the training data very well, but fails for any previously unseen data.
- For some intermediate value, the validation curve has a maximum. This level of complexity indicates a suitable trade-off between bias and variance.

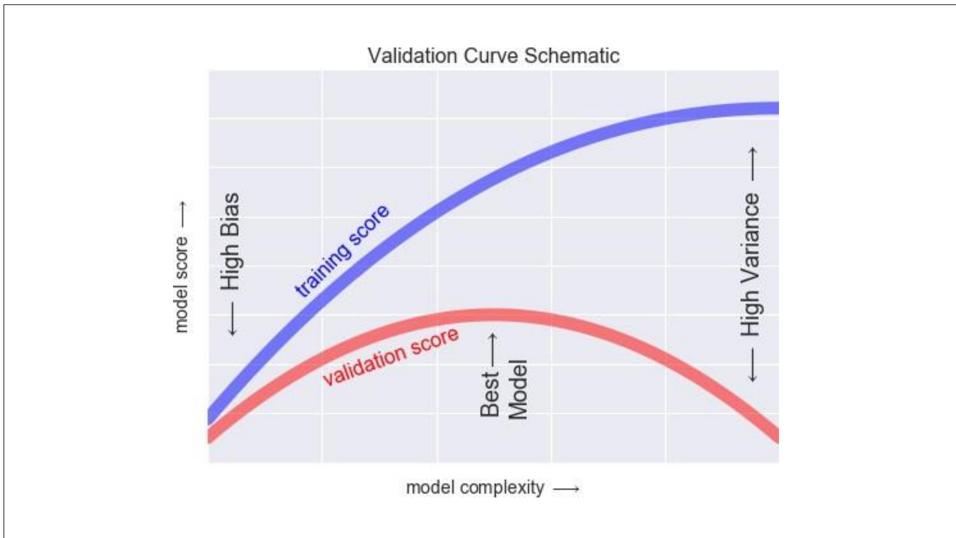


Figure 5-26. A schematic of the relationship between model complexity, training score,and validation score

The means of tuning the model complexity varies from model to model; when we discuss individual models in depth in later sections, we will see how each model allows for such tuning.

Validation curves in Scikit-Learn

Let's look at an example of using cross-validation to compute the validation curve fora class of models. Here we will use a *polynomial regression* model: this is a generalized linear model in which the degree of the polynomial is a tunable parameter. For example, a degree-1 polynomial fits a straight line to the data; for model parameters $a$and b:

$$y = ax + b$$

A degree-3 polynomial fits a cubic curve to the data; for model parameters a, b, c, d:

$$y = ax^3 + bx^2 + cx + d$$

We can generalize this to any number of polynomial features. In Scikit-Learn, we can implement this with a simple linear regression combined with the polynomial pre- processor. We will use a *pipeline* to string these operations together (we will discuss polynomial features and pipelines more fully in "Feature Engineering" on page 375):

```
In[10]: from sklearn.preprocessing import PolynomialFeatures
        from sklearn.linear_model import LinearRegression
        from sklearn.pipeline import make_pipeline

        def PolynomialRegression(degree=2, **kwargs):
            return make_pipeline(PolynomialFeatures(degree),
                                 LinearRegression(**kwargs))
```

Now let's create some data to which we will fit our model:

```
In[11]: import numpy as np

        def make_data(N, err=1.0, rseed=1):
            # randomly sample the data
            rng = np.random.RandomState(rseed)
            X = rng.rand(N, 1) ** 2
            y = 10 - 1. / (X.ravel() + 0.1)
            if err > 0:
                y += err * rng.randn(N)
            return X, y

        X, y = make_data(40)
```

We can now visualize our data, along with polynomial fits of several degrees (Figure 5-27):

```
In[12]: %matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # plot formatting

        X_test = np.linspace(-0.1, 1.1, 500)[:, None]

        plt.scatter(X.ravel(), y, color='black')
        axis = plt.axis()
        for degree in [1, 3, 5]:
            y_test = PolynomialRegression(degree).fit(X, y).predict(X_test)
            plt.plot(X_test.ravel(), y_test, label='degree={0}'.format(degree))
        plt.xlim(-0.1, 1.0)
```

```
plt.ylim(-2, 12)
plt.legend(loc='best');
```

The knob controlling model complexity in this case is the degree of the polynomial, which can be any non-negative integer. A useful question to answer is this: what degree of polynomial provides a suitable trade-off between bias (underfitting) and variance (overfitting)?

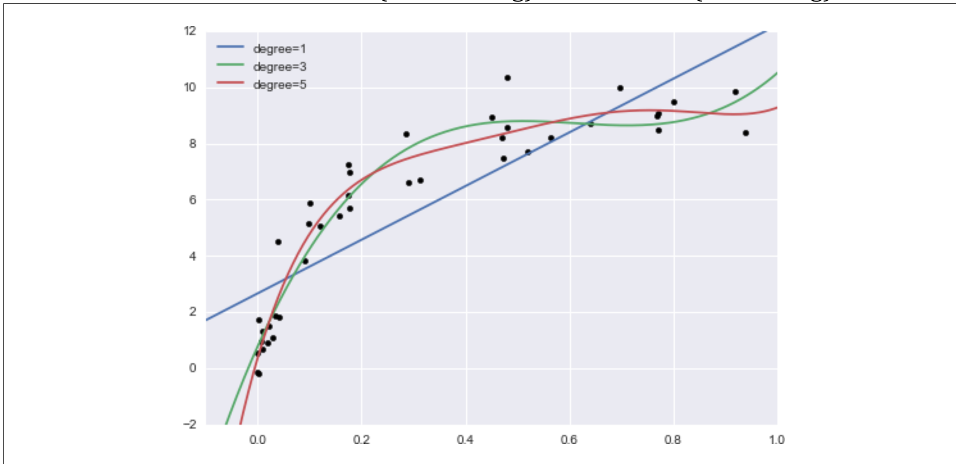


Figure 5-27. Three different polynomial models fit to a dataset

We can make progress in this by visualizing the validation curve for this particular data and model; we can do this straightforwardly using the `validation_curve` conve-nience routine provided by Scikit-Learn. Given a model, data, parameter name, and arange to explore, this function will automatically compute both the training score andvalidation score across the range (Figure 5-28):

```
In[13]:

from sklearn.learning_curve import validation_curve
degree = np.arange(0, 21)

train_score, val_score = validation_curve(PolynomialRegression(), X, y,

                                          'polynomialfeatures_degree',
                                          degree, cv=7)


plt.plot(degree, np.median(train_score, 1), color='blue', label='training score')
plt.plot(degree, np.median(val_score, 1), color='red', label='validation score')
plt.legend(loc='best')

plt.ylim(0, 1)
plt.xlabel('degree')
plt.ylabel('score');
```

This shows precisely the qualitative behavior we expect: the training score is every- where higher than the validation score; the training score is monotonically improvingwith increased model complexity; and the validation score reaches a maximumbefore dropping off as the

model becomes overfit.

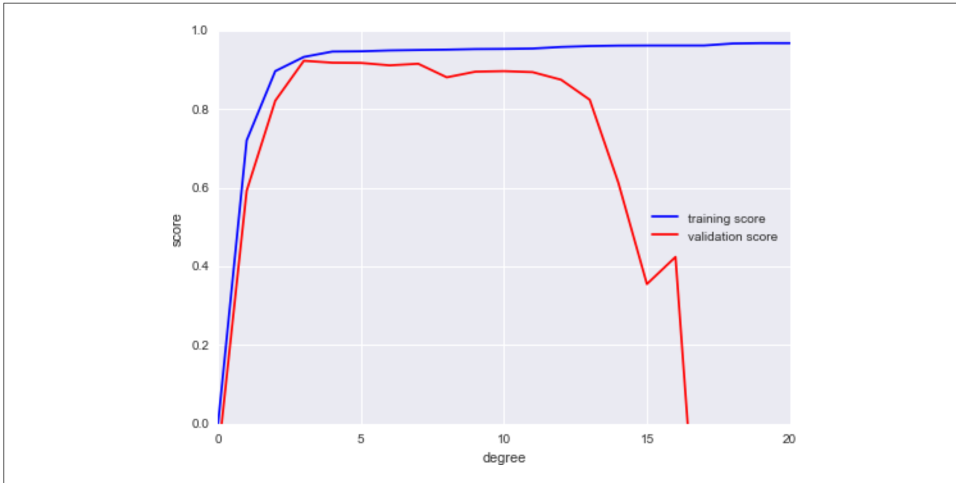


*Figure 5-28. **The** validation curves for the data in Figure 5-27 (cf. Figure 5-26)*

From the validation curve, we can read off that the optimal trade-off between biasand variance is found for a third-order polynomial; we can compute and display this fit over the original data as follows (Figure 5-29):

```
In[14]: plt.scatter(X.ravel(), y)
        lim = plt.axis()

        y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
        plt.plot(X_test.ravel(), y_test);

        plt.axis(lim);
```

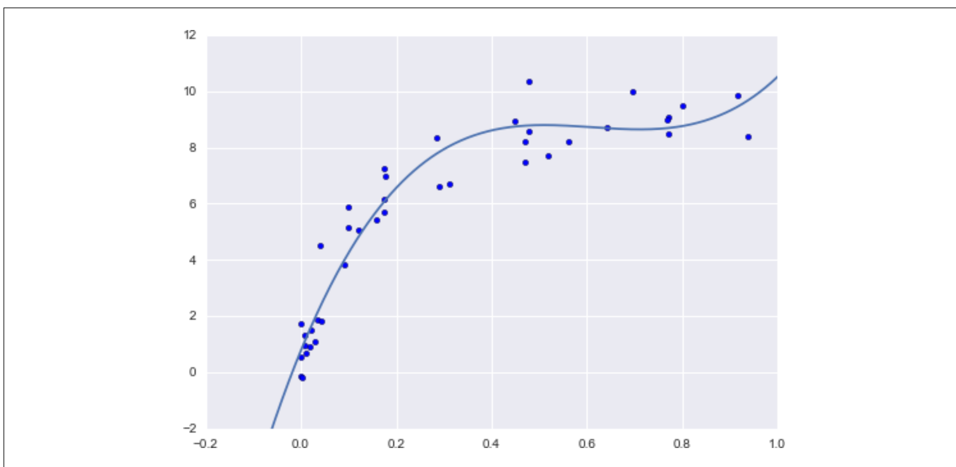


*Figure 5-29. **The** cross-validated optimal model for the data in Figure 5-27*

Notice that finding this optimal model did not actually require us to compute the training score, but examining the relationship between the training score and valida- tion score can give us useful insight into the performance of the model.

Learning Curves

One important aspect of model complexity is that the optimal model will generally depend on the size of your training data. For example, let's generate a new dataset with a factor of five more points (Figure 5-30):

```
In[15]: X2, y2 = make_data(200)
        plt.scatter(X2.ravel(), y2);
```

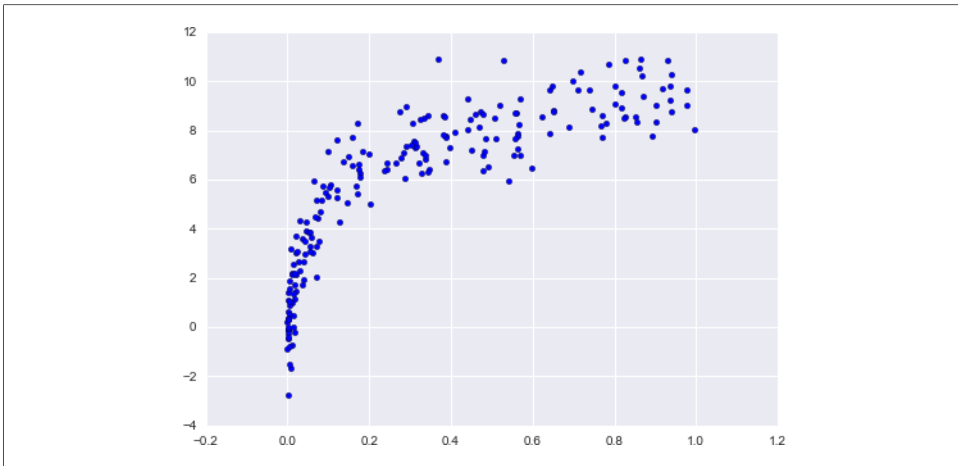


Figure 5-30. Data to demonstrate learning curves

We will duplicate the preceding code to plot the validation curve for this larger data-set; for reference let's over-plot the previous results as well (Figure 5-31):

```
In[16]:

degree = np.arange(21)

train_score2, val_score2 = validation_curve(PolynomialRegression(), X2, y2,

                                            'polynomialfeatures_degree',
                                            degree, cv=7)


plt.plot(degree, np.median(train_score2, 1), color='blue',
         label='training score')

plt.plot(degree, np.median(val_score2, 1), color='red', label='validation score')
plt.plot(degree, np.median(train_score, 1), color='blue', alpha=0.3,

         linestyle='dashed')

plt.plot(degree, np.median(val_score, 1), color='red', alpha=0.3,
         linestyle='dashed')
```

```
plt.legend(loc='lower center')
plt.ylim(0, 1)

plt.xlabel('degree')
plt.ylabel('score');
```

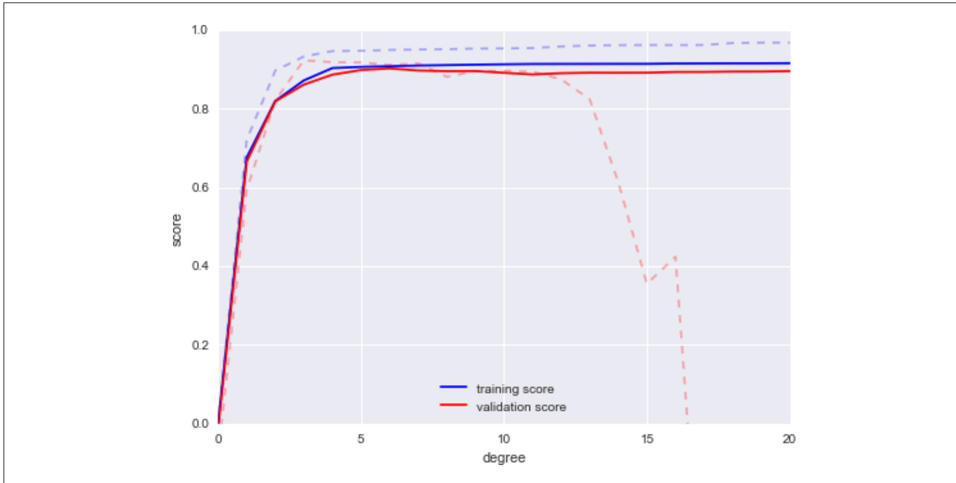


Figure 5-31. Learning curves for the polynomial model ***fit*** *to data in Figure 5-30*

The solid lines show the new results, while the fainter dashed lines show the results ofthe previous smaller dataset. It is clear from the validation curve that the larger data- set can support a much more complicated model: the peak here is probably around a degree of 6, but even a degree-20 model is not seriously overfitting the data—the vali-dation and training scores remain very close.

Thus we see that the behavior of the validation curve has not one, but two, important inputs: the model complexity and the number of training points. It is often useful to explore the behavior of the model as a function of the number of training points, which we can do by using increasingly larger subsets of the data to fit our model. A plot of the training/validation score with respect to the size of the training set is known as a *learning curve.*

The general behavior we would expect from a learning curve is this:

- A model of a given complexity will ***overfit*** a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
- A model of a given complexity will ***underfit*** a large dataset: this means that the training score will decrease, but the validation score will increase.
- A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.

With these features in mind, we would expect a learning curve to look qualitativelylike that shown in Figure 5-32.

Figure 5-32. Schematic showing the typical interpretation of learning curves

The notable feature of the learning curve is the convergence to a particular score as the number of training samples grows. In particular, once you have enough pointsthat a particular model has converged, *adding more training data will not help you!* The only way to increase model performance in this case is to use another (oftenmore complex) model.

Learning curves in Scikit-Learn

Scikit-Learn offers a convenient utility for computing such learning curves from yourmodels; here we will compute a learning curve for our original dataset with a second-order polynomial model and a ninth-order polynomial (Figure 5-33):

```
In[17]:

from sklearn.learning_curve import learning_curve


fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)


for i, degree in enumerate([2, 9]):

    N, train_lc, val_lc = learning_curve(PolynomialRegression(degree),

                                         X, y, cv=7,
                                         train_sizes=np.linspace(0.3, 1, 25))


    ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
    ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1], color='gray',

                 linestyle='dashed')
```

```
ax[i].set_ylim(0, 1)
ax[i].set_xlim(N[0], N[-1])
ax[i].set_xlabel('training size')
ax[i].set_ylabel('score')

ax[i].set_title('degree = {0}'.format(degree), size=14)
ax[i].legend(loc='best')
```

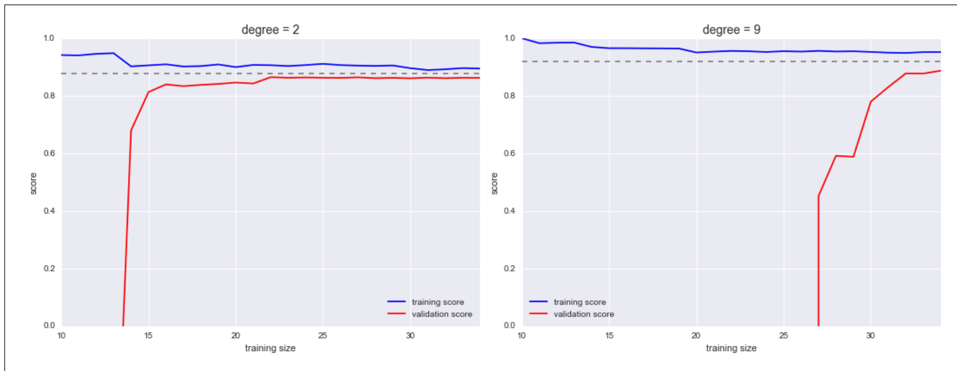


Figure 5-33. Learning curves for a low-complexity model ***(left)*** *and a high-complexity model (right)*

This is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing training data. In particular, when your learning curve has already converged (i.e., when the training and validation curves are already close to each other), *adding more training data will not significantly improve the fit!* This situa-tion is seen in the left panel, with the learning curve for the degree-2 model.

The only way to increase the converged score is to use a different (usually more com-plicated) model. We see this in the right panel: by moving to a much more compli- cated model, we increase the score of convergence (indicated by the dashed line), butat the expense of higher model variance (indicated by the difference between the training and validation scores). If we were to add even more data points, the learning curve for the more complicated model would eventually converge.

Plotting a learning curve for your particular choice of model and dataset can help youto make this type of decision about how to move forward in improving your analysis.

Validation in Practice: Grid Search

The preceding discussion is meant to give you some intuition into the trade-off between bias and variance, and its dependence on model complexity and training set size. In practice, models generally have more than one knob to turn, and thus plots of validation and learning curves change from lines to multidimensional surfaces. In these cases, such visualizations are difficult and we would rather simply find the par- ticular model that maximizes the validation score.

Scikit-Learn provides automated tools to do this in the grid_search module. Here isan example of using grid search to find the optimal polynomial model. We will explore a three-dimensional grid of model features—namely, the polynomial degree, the flag telling us whether to fit the intercept, and the flag telling us whether to nor- malize the problem. We can set this up using Scikit-Learn's GridSearchCV meta- estimator:

```
In[18]: from sklearn.grid_search import GridSearchCV

        param_grid = {'polynomialfeatures_degree': np.arange(21),
                      'linearregression_fit_intercept': [True, False],
                      'linearregression_normalize': [True, False]}

        grid = GridSearchCV(PolynomialRegression(), param_grid, cv=7)
```

Notice that like a normal estimator, this has not yet been applied to any data. Calling the fit() method will fit the model at each grid point, keeping track of the scores along the way:

```
In[19]: grid.fit(X, y);
```

Now that this is fit, we can ask for the best parameters as follows:

```
In[20]: grid.best_params_

Out[20]: {'linearregression_fit_intercept': False,
          'linearregression_normalize': True,
          'polynomialfeatures_degree': 4}
```

Finally, if we wish, we can use the best model and show the fit to our data using codefrom before (Figure 5-34):

```
In[21]: model = grid.best_estimator_


        plt.scatter(X.ravel(), y)
        lim = plt.axis()

        y_test = model.fit(X, y).predict(X_test)
        plt.plot(X_test.ravel(), y_test, hold=True);
        plt.axis(lim);
```

The grid search provides many more options, including the ability to specify a cus- tom scoring function, to parallelize the computations, to do randomized searches,and more. For information, see the examples in "In-Depth: Kernel Density Estima- tion" on page 491 and "Application: A Face Detection Pipeline" on page 506, or referto Scikit-Learn's grid search documentation.

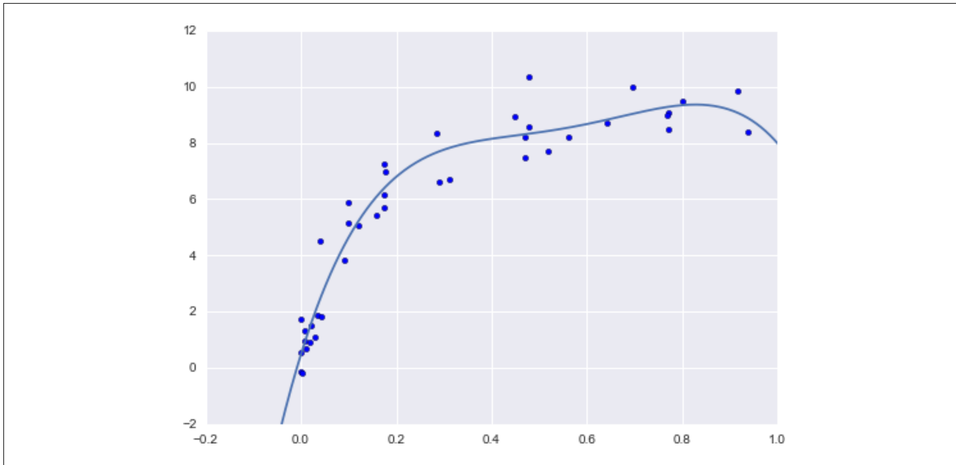


*Figure 5-34. **The best-fit** model determined via an automatic grid-search*

Day-02: Feature Engineering

The previous sections outline the fundamental ideas of machine learning, but all of the examples assume that you have numerical data in a tidy, `[n_samples, n_fea tures]` format. In the real world, data rarely comes in such a form. With this in mind,one of the more important steps in using machine learning in practice is *feature engi- neering*—that is, taking whatever information you have about your problem and turning it into numbers that you can use to build your feature matrix.

In this section, we will cover a few common examples of feature engineering tasks: features for representing *categorical data*, features for representing *text*, and features for representing *images*. Additionally, we will discuss *derived features* for increasingmodel complexity and *imputation* of missing data. Often this process is known as *vec-torization*, as it involves converting arbitrary data into well-behaved vectors.

Categorical Features

One common type of non-numerical data is *categorical* data. For example, imagine you are exploring some data on housing prices, and along with numerical featureslike “price” and “rooms,” you also have “neighborhood” information. For example, your data might look something like this:

```
In[1]: data = [
        {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
        {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
        {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
        {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
    ]
```

You might be tempted to encode this data with a straightforward numerical mapping:

```
In[2]: {'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3};
```

It turns out that this is not generally a useful approach in Scikit-Learn: the package's models make the fundamental assumption that numerical features reflect algebraic quantities. Thus such a mapping would imply, for example, that *Queen Anne < Fre- mont < Wallingford*, or even that *Wallingford - Queen Anne = Fremont*, which (nichedemographic jokes aside) does not make much sense.

In this case, one proven technique is to use *one-hot encoding*, which effectively createsextra columns indicating the presence or absence of a category with a value of 1 or 0,respectively. When your data comes as a list of dictionaries, Scikit-Learn's `DictVectorizer` will do this for you:

```
In[3]: from sklearn.feature_extraction import DictVectorizer
       vec = DictVectorizer(sparse=False, dtype=int)
       vec.fit_transform(data)
```

```
Out[3]:          0,     1,   0, 850000,      4],
array([[
[                1,     0,   0, 700000,      3],
[                0,     0,   1, 650000,      3],
[                1,     0,   0, 600000,      2]],
                                             dtype=int64)
```

Notice that the *neighborhood* column has been expanded into three separate columns, representing the three neighborhood labels, and that each row has a 1 in the column associated with its neighborhood. With these categorical features thus encoded, you can proceed as normal with fitting a Scikit-Learn model.

To see the meaning of each column, you can inspect the feature names:

```
In[4]: vec.get_feature_names()
```

```
Out[4]: ['neighborhood=Fremont',

         'neighborhood=Queen Anne',
         'neighborhood=Wallingford',
         'price',

         'rooms']
```

There is one clear disadvantage of this approach: if your category has many possible values, this can *greatly* increase the size of your dataset. However, because the enco- ded data contains mostly zeros, a sparse output can be a very efficient solution:

```
In[5]: vec = DictVectorizer(sparse=True, dtype=int)
       vec.fit_transform(data)
```

```
Out[5]: <4x5 sparse matrix of type '<class 'numpy.int64'>'

            with 12 stored elements in Compressed Sparse Row format>
```

Many (though not yet all) of the Scikit-Learn estimators accept such sparse inputs when fitting and evaluating models. `sklearn.preprocessing.OneHotEncoder` and

sklearn.feature_extraction.FeatureHasher are two additional tools that Scikit-Learn includes to support this type of encoding.

Text Features

Another common need in feature engineering is to convert text to a set of representa- tive numerical values. For example, most automatic mining of social media data relieson some form of encoding the text as numbers. One of the simplest methods of encoding data is by *word counts*: you take each snippet of text, count the occurrencesof each word within it, and put the results in a table.

For example, consider the following set of three phrases:

```
In[6]: sample = ['problem of evil',

                 'evil queen',
                 'horizon problem']
```

For a vectorization of this data based on word count, we could construct a column representing the word "problem," the word "evil," the word "horizon," and so on. While doing this by hand would be possible, we can avoid the tedium by using Scikit-Learn's CountVectorizer:

```
In[7]: from sklearn.feature_extraction.text import CountVectorizer


       vec = CountVectorizer()

       X = vec.fit_transform(sample)
       X

Out[7]: <3x5 sparse matrix of type '<class 'numpy.int64'>'

                with 7 stored elements in Compressed Sparse Row format>
```

The result is a sparse matrix recording the number of times each word appears; it is easier to inspect if we convert this to a DataFrame with labeled columns:

```
In[8]: import pandas as pd

       pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

Out[8]:

	evil	horizon	of	problem	queen
0	1	0	1	1	0
1	1	0	0	0	1
2	0	1	0	1	0

There are some issues with this approach, however: the raw word counts lead to fea-tures that put too much weight on words that appear very frequently, and this can besuboptimal in some classification algorithms. One approach to fix this is known as *term frequency–inverse document frequency* (*TF–IDF*), which weights the word countsby a measure of how often they appear in the documents. The syntax for computingthese features is similar to the previous example:

```
In[9]: from sklearn.feature_extraction.text import TfidfVectorizer
       vec = TfidfVectorizer()

       X = vec.fit_transform(sample)

       pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

```
Out[9]: evil        horizon   of        problem   queen
     0  0.517856    0.000000  0.680919  0.517856  0.000000
     1  0.605349    0.000000  0.000000  0.605349  0.795961
     2  0.000000    0.795961  0.000000  0.605349  0.000000
```

For an example of using TF–IDF in a classification problem, see "In Depth: NaiveBayes Classification" on page 382.

Image Features

Another common need is to suitably encode *images* for machine learning analysis. The simplest approach is what we used for the digits data in "Introducing Scikit- Learn" on page 343: simply using the pixel values themselves. But depending on the application, such approaches may not be optimal.

A comprehensive summary of feature extraction techniques for images is well beyondthe scope of this section, but you can find excellent implementations of many of the standard approaches in the Scikit-Image project. For one example of using Scikit- Learn and Scikit-Image together, see "Application: A Face Detection Pipeline" on page506.

Derived Features

Another useful type of feature is one that is mathematically derived from some input features. We saw an example of this in "Hyperparameters and Model Validation" on page 359 when we constructed *polynomial features* from our input data. We saw that we could convert a linear regression into a polynomial regression not by changing themodel, but by transforming the input! This is sometimes known as *basis function regression*, and is explored further in "In Depth: Linear Regression" on page 390.

For example, this data clearly cannot be well described by a straight line(Figure 5-35):

```
In[10]: %matplotlib inline

        import numpy as np

        import matplotlib.pyplot as plt


        x = np.array([1, 2, 3, 4, 5])
```

```
y = np.array([4, 2, 1, 3, 7])
plt.scatter(x, y);
```

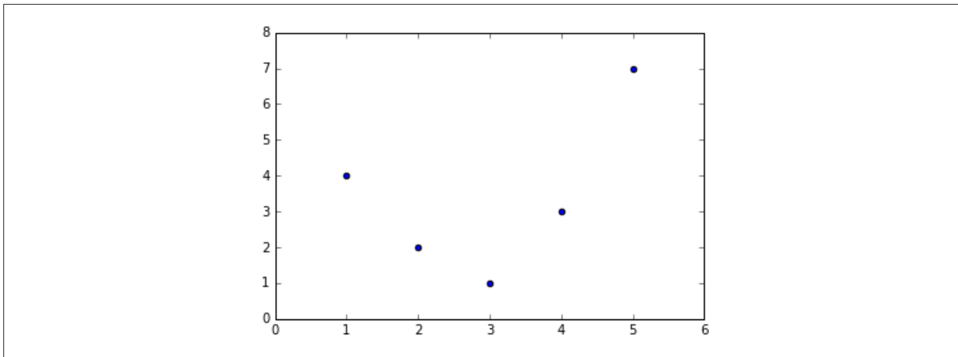


Figure 5-35. Data that is not well described by a straight line

Still, we can fit a line to the data using `LinearRegression` and get the optimal result(Figure 5-36):

```
In[11]: from sklearn.linear_model import LinearRegression
        X = x[:, np.newaxis]

        model = LinearRegression().fit(X, y)
        yfit = model.predict(X)
        plt.scatter(x, y)

        plt.plot(x, yfit);
```

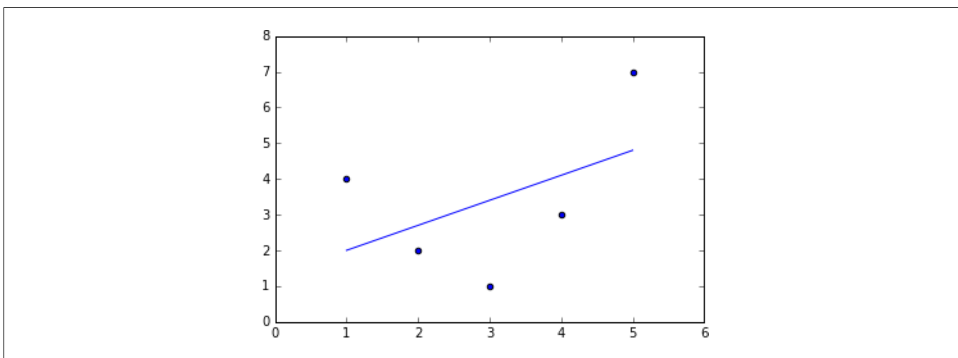


Figure 5-36. A poor straight-line fit

It's clear that we need a more sophisticated model to describe the relationshipbetween *x* and *y*. We can do this by transforming the data, adding extra columns of features to drive more flexibility in the model. For example, we can add polynomial features to the data this way:

```
In[12]: from sklearn.preprocessing import PolynomialFeatures
        poly = PolynomialFeatures(degree=3, include_bias=False)
        X2 = poly.fit_transform(X)

        print(X2)
```

```
[[  1.    1.     1.]
[   2.    4.     8.]
[   3.    9.    27.]
[   4.   16.    64.]
[   5.   25.   125.]
                  ]
```

The derived feature matrix has one column representing *x*, and a second column representing x^2, and a third column representing x^3. Computing a linear regression onthis expanded input gives a much closer fit to our data (Figure 5-37):

```
In[13]: model = LinearRegression().fit(X2, y)
        yfit = model.predict(X2)
        plt.scatter(x, y)

        plt.plot(x, yfit);
```

*Figure 5-37. A linear **fit** to polynomial features derived from the data*

This idea of improving a model not by changing the model, but by transforming the inputs, is fundamental to many of the more powerful machine learning methods. We explore this idea further in "In Depth: Linear Regression" on page 390 in the context of *basis function regression*. More generally, this is one motivational path to the pow-erful set of techniques known as *kernel methods*, which we will explore in "In-Depth:Support Vector Machines" on age 405.

Imputation of Missing Data

Another common need in feature engineering is handling missing data. We discussedthe handling of missing data in `DataFrame`s in "Handling Missing Data" on page 119,and saw that often the `NaN` value is used to mark missing values. For example, we might have a dataset that looks like this:

```
In[14]: from numpy import nan

        X = np.array([[ nan, 0,    3  ],
                      [    7,    9  ],
                     3,
                      [    5,    2  ],
                     3,
                      [    nan  6  ],
```

```
                              4,    ,
                               [    8,   1  ]])
                              8,
          y = np.array([14,  16,  -1,   8,  -5])
```

When applying a typical machine learning model to such data, we will need to first replace such missing data with some appropriate fill value. This is known as *imputa- tion* of missing values, and strategies range from simple (e.g., replacing missing valueswith the mean of the column) to sophisticated (e.g., using matrix completion or a robust model to handle such data).

The sophisticated approaches tend to be very application-specific, and we won't dive into them here. For a baseline imputation approach, using the mean, median, or most frequent value, Scikit-Learn provides the `Imputer` class:

```
In[15]: from sklearn.preprocessing import Imputer
        imp = Imputer(strategy='mean')

        X2 = imp.fit_transform(X)
        X2
```

```
Out[15]: array([[     0.    3.
4.5,              ,     ],
[ 3.  ,           7.    9.
                   ,    ],
[ 3.  ,           5.    2.
                   ,    ],
[ 4.  ,           5.    6.
                   ,    ],
[ 8.  ,           8.     1.
                   ,    ]])
```

We see that in the resulting data, the two missing values have been replaced with the mean of the remaining values in the column. This imputed data can then be fed directly into, for example, a `LinearRegression` estimator:

```
In[16]: model = LinearRegression().fit(X2, y)
        model.predict(X2)
```

```
Out[16]:

array([ 13.14869292,   14.3784627 ,   -1.15539732,   10.96606197,   -5.33782027])
```

Feature Pipelines

With any of the preceding examples, it can quickly become tedious to do the transfor-mations by hand, especially if you wish to string together multiple steps. For example,we might want a processing pipeline that looks something like this:

1. Impute missing values using the mean
2. Transform features to quadratic
3. Fit a linear regression

To streamline this type of processing pipeline, Scikit-Learn provides a pipeline object, which can be used as follows:

```
In[17]: from sklearn.pipeline import make_pipeline

        model = make_pipeline(Imputer(strategy='mean'),
                              PolynomialFeatures(degree=2),
                              LinearRegression())
```

This pipeline looks and acts like a standard Scikit-Learn object, and will apply all thespecified steps to any input data.

```
In[18]: model.fit(X, y)  # X with missing values, from above
        print(y)
        print(model.predict(X))

[14 16 -1  8 -5]
[ 14.  16.  -1.   8.  -5.]
```

All the steps of the model are applied automatically. Notice that for the simplicity of this demonstration, we've applied the model to the data it was trained on; this is why it was able to perfectly predict the result (refer back to "Hyperparameters and Model Validation" on page 359 for further discussion of this).

For some examples of Scikit-Learn pipelines in action, see the following section on naive Bayes classification as well as "In Depth: Linear Regression" on page 390 and "In-Depth: Support Vector Machines" on page 405.

Day-03: Linear Regression

Just as naive Bayes (discussed earlier in "In Depth: Naive Bayes Classification" on page 382) is a good starting point for classification tasks, linear regression models area good starting point for regression tasks. Such models are popular because they can be fit very quickly, and are very interpretable. You are probably familiar with the sim-plest form of a linear regression model (i.e., fitting a straight line to data), but such models can be extended to model more complicated data behavior.

In this section we will start with a quick intuitive walk-through of the mathematics behind this well-known problem, before moving on to see how linear models can be generalized to account for more complicated patterns in data. We begin with the stan-dard imports:

```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
       import numpy as np
```

Simple Linear Regression

We will start with the most familiar linear regression, a straight-line fit to data. A straight-line

fit is a model of the form $y = ax + b$ where a is commonly known as the *slope*, and b is commonly known as the *intercept*.

Consider the following data, which is scattered about a line with a slope of 2 and an intercept of –5 (Figure 5-42):

```
In[2]: rng = np.random.RandomState(1)
       x = 10 * rng.rand(50)

       y = 2 * x - 5 + rng.randn(50)
       plt.scatter(x, y);
```

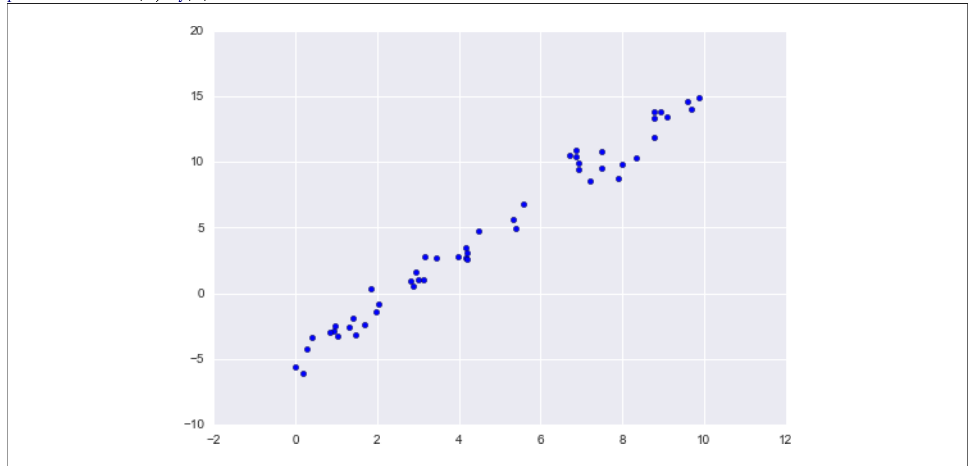


Figure 5-42. Data for linear regression

We can use Scikit-Learn's `LinearRegression` estimator to fit this data and constructthe best-fit line (Figure 5-43):

```
In[3]: from sklearn.linear_model import LinearRegression
       model = LinearRegression(fit_intercept=True)


       model.fit(x[:, np.newaxis], y)

       xfit = np.linspace(0, 10, 1000)
```

```
yfit = model.predict(xfit[:, np.newaxis])plt.scatter(x, y) plt.plot(xfit, yfit);
```

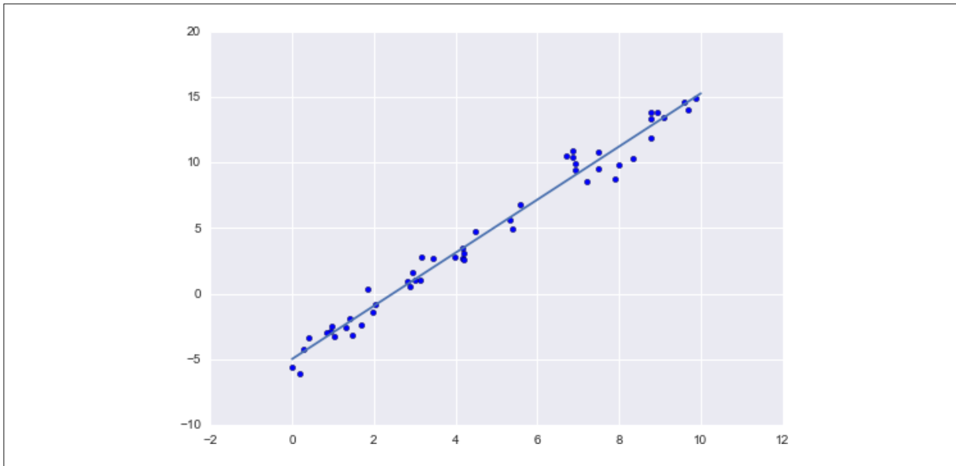


Figure 5-43. A linear regression model

The slope and intercept of the data are contained in the model's fit parameters, whichin Scikit-Learn are always marked by a trailing underscore. Here the relevant parame-ters are `coef_` and `intercept_`:

```
In[4]: print("Model slope:    ", model.coef_[0])

       print("Model intercept:", model.intercept_)

Model slope:     2.02720881036

Model intercept: -4.99857708555
```

We see that the results are very close to the inputs, as we might hope.

The `LinearRegression` estimator is much more capable than this, however—in addi-tion to simple straight-line fits, it can also handle multidimensional linear models of the form:

$$y = a_0 + a_1x_1 + a_2x_2 + \cdots$$

where there are multiple *x* values. Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a hyper-plane to points in higher dimensions.

The multidimensional nature of such regressions makes them more difficult to visu- alize, but we can see one of these fits in action by building some example data, using NumPy's matrix multiplication operator:

```
In[5]: rng = np.random.RandomState(1)
       X = 10 * rng.rand(100, 3)

       y = 0.5 + np.dot(X, [1.5, -2., 1.])
```

```
model.fit(X, y)
print(model.intercept_)
print(model.coef_)
```

```
0.5
[ 1.5 -2.   1. ]
```

Here the y data is constructed from three random x values, and the linear regression recovers the coefficients used to construct the data.

In this way, we can use the single `LinearRegression` estimator to fit lines, planes, or hyperplanes to our data. It still appears that this approach would be limited to strictly linear relationships between variables, but it turns out we can relax this as well.

Basis Function Regression

One trick you can use to adapt linear regression to nonlinear relationships between variables is to transform the data according to *basis functions*. We have seen one ver-sion of this before, in the `PolynomialRegression` pipeline used in "Hyperparameters and Model Validation" on page 359 and "Feature Engineering" on page 375. The ideais to take our multidimensional linear model:

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots$$

and build the x_1, x_2, x_3, and so on from our single-dimensional input x. That is, we let $x_n = f_n(x)$ where $f_n()$is some function that transforms our data.

For example, if $f_n(x) = x^n$, our model becomes a polynomial regression:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

Notice that this is *still a linear model*—the linearity refers to the fact that the coeffi- cients a_n never multiply or divide each other. What we have effectively done is taken our one-dimensional x values and projected them into a higher dimension, so that a linear fit can fit more complicated relationships between x and y.

Polynomial basis functions

This polynomial projection is useful enough that it is built into Scikit-Learn, using the `PolynomialFeatures` transformer:

```
In[6]: from sklearn.preprocessing import PolynomialFeatures
       x = np.array([2, 3, 4])

       poly = PolynomialFeatures(3, include_bias=False)
       poly.fit_transform(x[:, None])
```

```
Out[6]:
array([[ 2.,  4.,  8.],
       [ 3.,  9., 27.],
       [ 4., 16., 64.]])
```

We see here that the transformer has converted our one-dimensional array into a three-dimensional array by taking the exponent of each value. This new, higher- dimensional data representation can then be plugged into a linear regression.

As we saw in "Feature Engineering" on page 375, the cleanest way to accomplish thisis to use a pipeline. Let's make a 7th-degree polynomial model in this way:

```
In[7]: from sklearn.pipeline import make_pipeline
       poly_model = make_pipeline(PolynomialFeatures(7),
                                  LinearRegression())
```

With this transform in place, we can use the linear model to fit much more compli- cated relationships between *x* and *y*. For example, here is a sine wave with noise (Figure 5-44):

```
In[8]: rng = np.random.RandomState(1)
       x = 10 * rng.rand(50)
       y = np.sin(x) + 0.1 * rng.randn(50)

       poly_model.fit(x[:, np.newaxis], y)
       yfit = poly_model.predict(xfit[:, np.newaxis])

       plt.scatter(x, y)
       plt.plot(xfit, yfit);
```

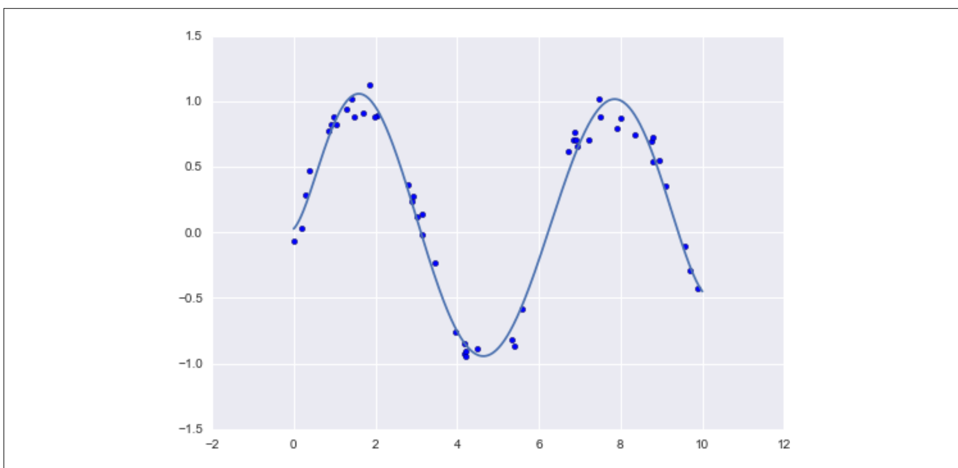


Figure 5-44. A linear polynomial fit to nonlinear training data

Our linear model, through the use of 7th-order polynomial basis functions, can provide an excellent fit to this nonlinear data!

Gaussian basis functions

Of course, other basis functions are possible. For example, one useful pattern is to fit a model that is not a sum of polynomial bases, but a sum of Gaussian bases. The result might look something like Figure 5-45.

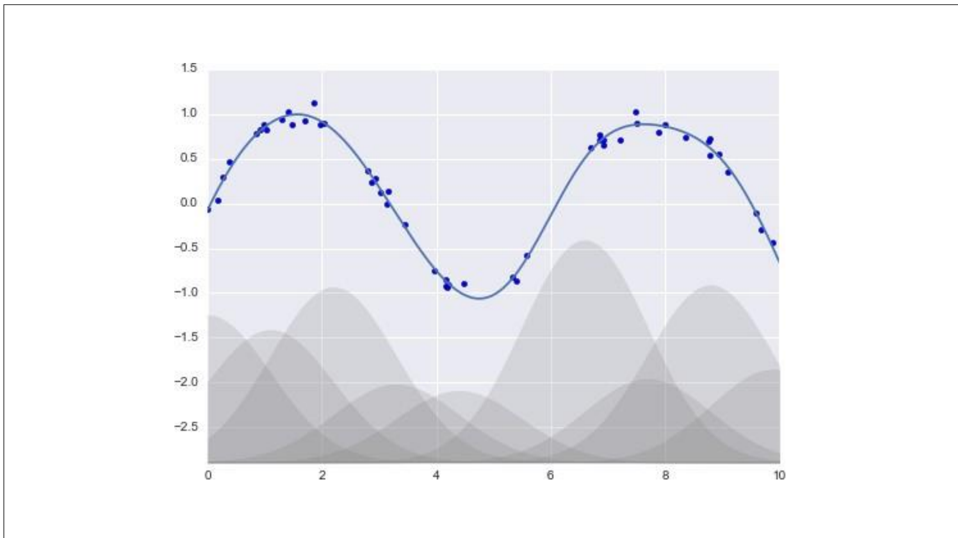


*Figure 5-45. A Gaussian basis function **fit** to nonlinear data*

The shaded regions in the plot shown in Figure 5-45 are the scaled basis functions, and when added together they reproduce the smooth curve through the data. These Gaussian basis functions are not built into Scikit-Learn, but we can write a custom transformer that will create them, as shown here and illustrated in Figure 5-46(Scikit-Learn transformers are implemented as Python classes; reading Scikit-Learn'ssource is a good way to see how they can be created):

```
In[9]:

from sklearn.base import BaseEstimator, TransformerMixin


class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""


    def __init__(self, N, width_factor=2.0):
        self.N = N

        self.width_factor = width_factor
```

```
    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```

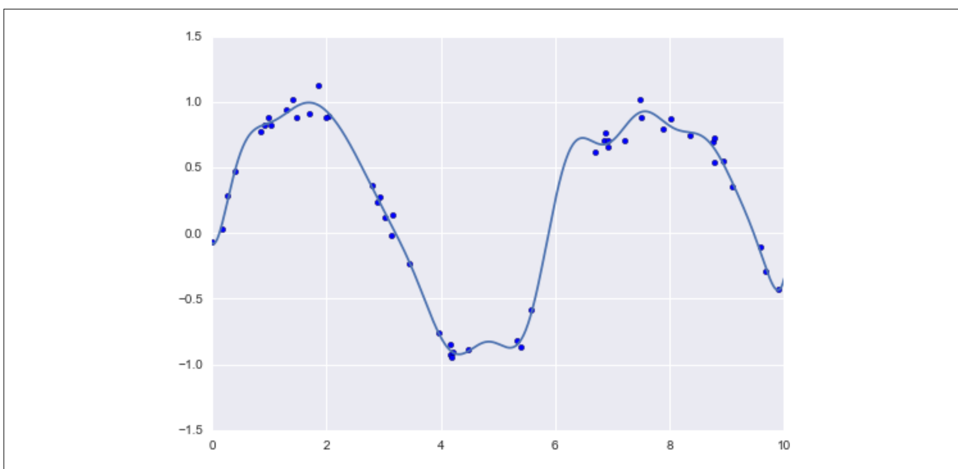


*Figure 5-46. A Gaussian basis function **fit** computed with a custom transformer*

We put this example here just to make clear that there is nothing magic about poly- nomial basis functions: if you have some sort of intuition into the generating process of your data that makes you think one basis or another might be appropriate, you can use them as well.

Regularization

The introduction of basis functions into our linear regression makes the model much more flexible, but it also can very quickly lead to overfitting (refer back to "Hyper- parameters and Model Validation" on page 359 for a discussion of this). For example,if we choose too many Gaussian basis functions, we end up with results that don't look so good (Figure 5-47):

```
In[10]: model = make_pipeline(GaussianFeatures(30),
                              LinearRegression())
        model.fit(x[:, np.newaxis], y)

        plt.scatter(x, y)
        plt.plot(xfit, model.predict(xfit[:, np.newaxis]))
        plt.xlim(0, 10)
        plt.ylim(-1.5, 1.5);
```

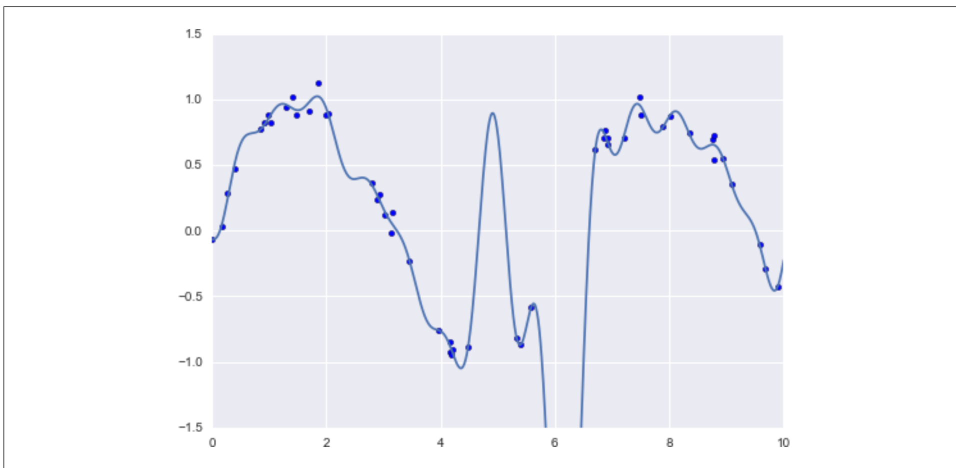


*Figure 5-47. An overly complex basis function model that **overfits** the data*

With the data projected to the 30-dimensional basis, the model has far too much flex-ibility and goes to extreme values between locations where it is constrained by data. We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations (Figure 5-48):

```
In[11]: def basis_plot(model, title=None):
            fig, ax = plt.subplots(2, sharex=True)
            model.fit(x[:, np.newaxis], y)
            ax[0].scatter(x, y)
```

```
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:

        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)

    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))

model = make_pipeline(GaussianFeatures(30), LinearRegression())
basis_plot(model)
```

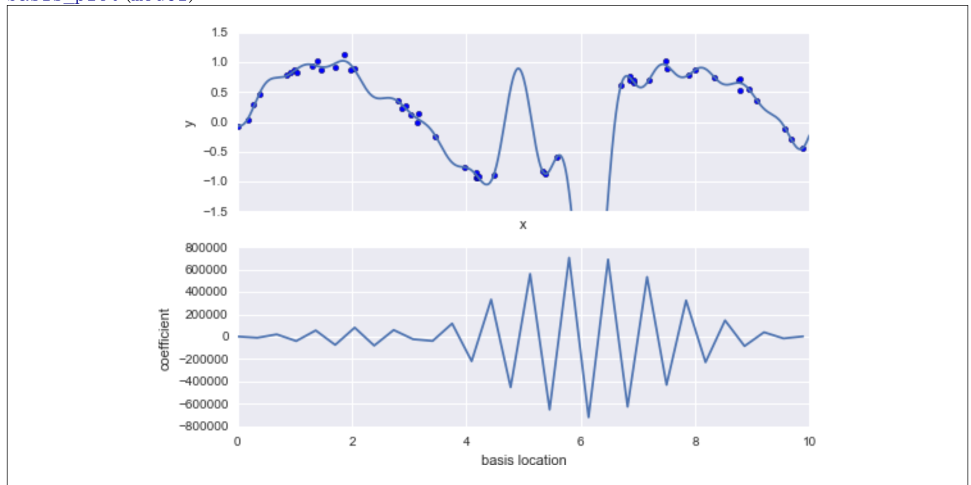


*Figure 5-48. **The coefficients** of the Gaussian bases in the overly complex model*

The lower panel in Figure 5-48 shows the amplitude of the basis function at each location. This is typical overfitting behavior when basis functions overlap: the coeffi-cients of adjacent basis functions blow up and cancel each other out. We know that such behavior is problematic, and it would be nice if we could limit such spikes explicitly in the model by penalizing large values of the model parameters. Such a penalty is known as *regularization*, and comes in several forms.

Ridge regression (L_2 regularization)

Perhaps the most common form of regularization is known as *ridge regression* or L_2 *regularization*, sometimes also called *Tikhonov regularization*. This proceeds by penal- izing the sum of squares (2-norms) of the model coefficients; in this case, the penaltyon the model fit would be:

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

where α is a free parameter that controls the strength of the penalty. This type of penalized model is built into Scikit-Learn with the `Ridge` estimator (Figure 5-49):

```
In[12]: from sklearn.linear_model import Ridge

        model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
        basis_plot(model, title='Ridge Regression')
```

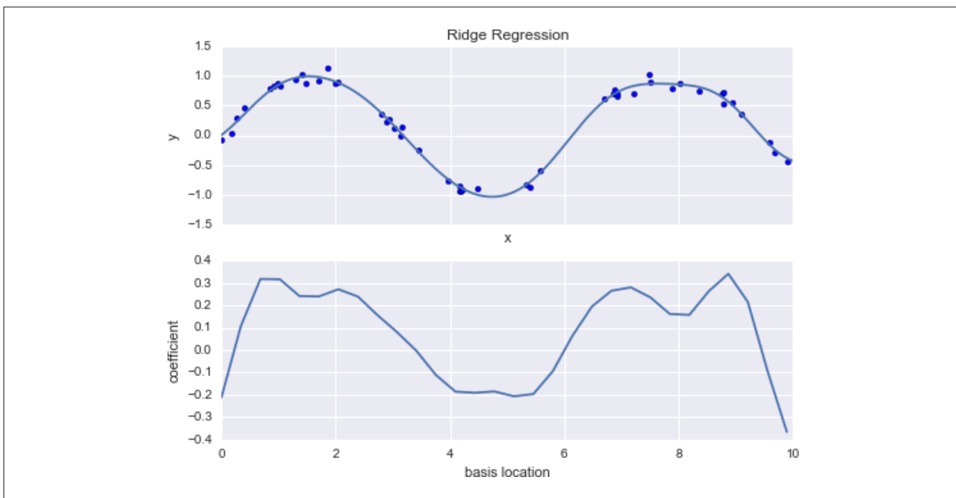


Figure 5-49. Ridge (L_2) regularization applied to the overly complex model (compare to Figure 5-48)

The α parameter is essentially a knob controlling the complexity of the resulting model. In the limit $\alpha \to 0$, we recover the standard linear regression result; in the limit $\alpha \to \infty$, all model responses will be suppressed. One advantage of ridge regres- sion in particular is that it can be computed very efficiently—at hardly more compu- tational cost than the original linear regression model.

Lasso regularization (L_1)

Another very common type of regularization is known as lasso, and involves penaliz-ing the sum of absolute values (1-norms) of regression coefficients:

$$P = \alpha \sum_{n=1}^{N} \left| \theta_n \right|$$

Though this is conceptually very similar to ridge regression, the results can differ sur-prisingly: for example, due to geometric reasons lasso regression tends to favor *sparsemodels* where possible; that is, it preferentially sets model coefficients to exactly zero.

We can see this behavior in duplicating the plot shown in Figure 5-49, but using L1-normalized coefficients (Figure 5-50):

```
In[13]: from sklearn.linear_model import Lasso

        model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
        basis_plot(model, title='Lasso Regression')
```

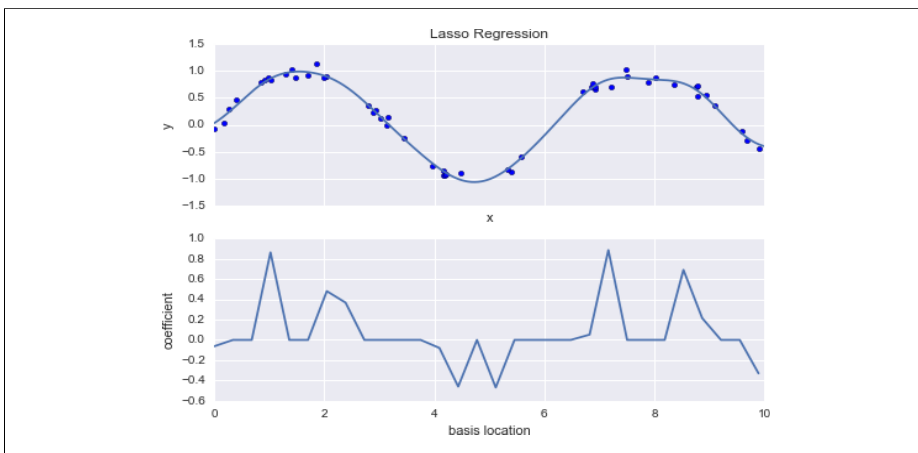


Figure 5-50. Lasso (L_1) regularization applied to the overly complex model (compare to Figure 5-48)

With the lasso regression penalty, the majority of the coefficients are exactly zero, with the functional behavior being modeled by a small subset of the available basis functions. As with ridge regularization, the α parameter tunes the strength of the penalty, and should be determined via, for example, cross-validation (refer back to "Hyperparameters and Model Validation" on page 359 for a discussion of this).

Example: Predicting Bicycle Traffic

As an example, let's take a look at whether we can predict the number of bicycle tripsacross Seattle's Fremont Bridge based on weather, season, and other factors. We haveseen this data already in "Working with Time Series" on page 188.

In this section, we will join the bike data with another dataset, and try to determine the extent to which weather and seasonal factors—temperature, precipitation, and daylight hours—affect the volume of bicycle traffic through this corridor. Fortunately,the NOAA makes available their daily weather station data (I used station ID USW00024233) and we can easily use Pandas to join the two data sources. We will perform a simple linear regression to relate weather and other information to bicycle counts, in order to estimate how a change in any one of these parameters affects the number of riders on a given day.

In particular, this is an example of how the tools of Scikit-Learn can be used in a stat-istical modeling framework, in which the parameters of the model are assumed to have interpretable meaning. As discussed previously, this is not a standard approach within machine learning, but such interpretation is possible for some models.

Let's start by loading the two datasets, indexing by date:

```
In[14]:

import pandas as pd

counts = pd.read_csv('fremont_hourly.csv', index_col='Date', parse_dates=True)
weather = pd.read_csv('599021.csv', index_col='DATE', parse_dates=True)
```

Next we will compute the total daily bicycle traffic, and put this in its own `DataFrame`:

```
In[15]: daily = counts.resample('d', how='sum')
        daily['Total'] = daily.sum(axis=1)

        daily = daily[['Total']] # remove other columns
```

We saw previously that the patterns of use generally vary from day to day; let's account for this in our data by adding binary columns that indicate the day of the week:

```
In[16]: days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

        for i in range(7):

            daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

Similarly, we might expect riders to behave differently on holidays; let's add an indica-tor of this as well:

```
In[17]: from pandas.tseries.holiday import USFederalHolidayCalendar
        cal = USFederalHolidayCalendar()

        holidays = cal.holidays('2012', '2016')

        daily = daily.join(pd.Series(1, index=holidays, name='holiday'))
        daily['holiday'].fillna(0, inplace=True)
```

We also might suspect that the hours of daylight would affect how many people ride; let's use the standard astronomical calculation to add this information (Figure 5-51):

```
In[18]: def hours_of_daylight(date, axis=23.44, latitude=47.61):
            """Compute the hours of daylight for the given  date"""
            days = (date - pd.datetime(2000, 12, 21)).days

            m = (1. - np.tan(np.radians(latitude))

                 * np.tan(np.radians(axis) * np.cos(days * 2 * np.pi / 365.25)))

            return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.


        daily['daylight_hrs'] = list(map(hours_of_daylight, daily.index))
        daily[['daylight_hrs']].plot();
```

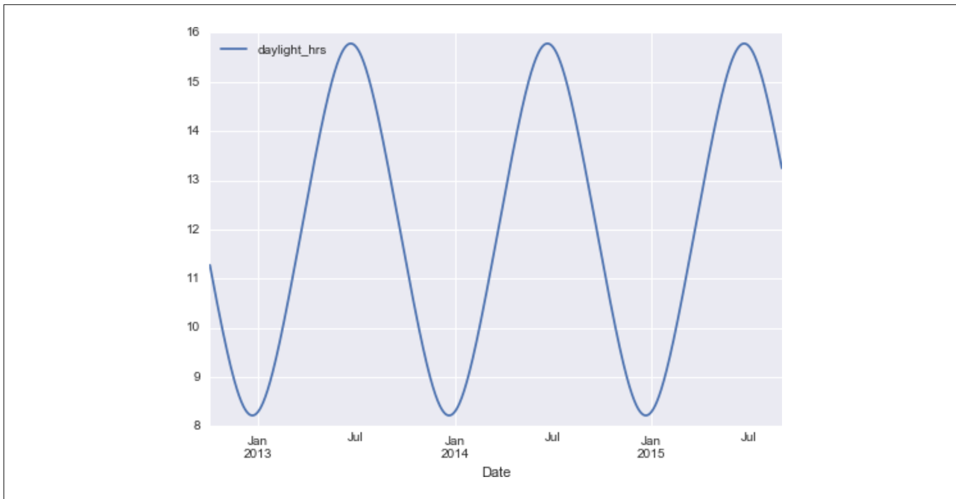


Figure 5-51. Visualization of hours of daylight in Seattle

We can also add the average temperature and total precipitation to the data. In addi- tion to the inches of precipitation, let's add a flag that indicates whether a day is dry (has zero precipitation):

```
In[19]: # temperatures are in 1/10 deg C; convert to C
        weather['TMIN'] /= 10
        weather['TMAX'] /= 10
        weather['Temp (C)'] = 0.5 * (weather['TMIN'] + weather['TMAX'])

        # precip is in 1/10 mm; convert to inches
        weather['PRCP'] /= 254
        weather['dry day'] = (weather['PRCP'] == 0).astype(int)

        daily = daily.join(weather[['PRCP', 'Temp (C)', 'dry day']])
```

Finally, let's add a counter that increases from day 1, and measures how many years have passed. This will let us measure any observed annual increase or decrease in daily crossings:

```
In[20]: daily['annual'] = (daily.index - daily.index[0]).days / 365.
```

Now our data is in order, and we can take a look at it:

```
In[21]: daily.head()

Out[21]:
             Tota Mon  Tue  Wed  Thu  Fri  Sat  Sun  holida    daylight_hrs
Date          l                                       y                  \\
```

2012-10-03	3521	0	0	1	0	0	0	0	0	11.277359
2012-10-04	3475	0	0	0	1	0	0	0	0	11.219142
2012-10-05	3148	0	0	0	0	1	0	0	0	11.161038
2012-10-06	2006	0	0	0	0	0	1	0	0	11.103056
2012-10-07	2142	0	0	0	0	0	0	1	0	11.045208

Date	PRCP	Temp (C)	dry day	annual
2012-10-03	0	13.35	1	0.000000
2012-10-04	0	13.60	1	0.002740
2012-10-05	0	15.30	1	0.005479
2012-10-06	0	15.85	1	0.008219
2012-10-07	0	15.85	1	0.010959

With this in place, we can choose the columns to use, and fit a linear regressionmodel to our data. We will set `fit_intercept = False`, because the daily flags essen- tially operate as their own day-specific intercepts:

```
In[22]:

column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'holiday',
                'daylight_hrs', 'PRCP', 'dry day', 'Temp (C)', 'annual']

X = daily[column_names]
y = daily['Total']


model = LinearRegression(fit_intercept=False)
model.fit(X, y)

daily['predicted'] = model.predict(X)
```

Finally, we can compare the total and predicted bicycle traffic visually (Figure 5-52):In[23]: daily[['Total', 'predicted']].plot(alpha=0.5);

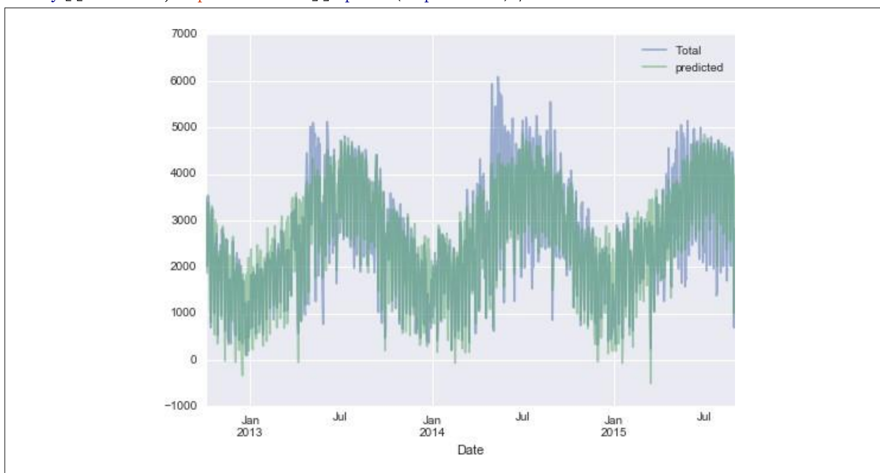


Figure 5-52. Our model's prediction of bicycle ***traffic***

It is evident that we have missed some key features, especially during the summer time. Either our features are not complete (i.e., people decide whether to ride to workbased on more than just these) or there are some nonlinear relationships that we have failed to take into account (e.g., perhaps people ride less at both high and low temper-atures). Nevertheless, our rough approximation is enough to give us some insights, and we can take a look at the coefficients of the linear model to estimate how much each feature contributes to the daily bicycle count:

```
In[24]: params = pd.Series(model.coef_, index=X.columns)
        params
```

```
Out[24]: Mon             503.797330
         Tue             612.088879
         Wed             591.611292
         Thu             481.250377
         Fri             176.838999
         Sat            -1104.321406
         Sun            -1134.610322
         holiday        -1187.212688
         daylight_hrs    128.873251
         PRCP           -665.185105
         dry day         546.185613
         Temp (C)         65.194390
```

```
annual      27.865349
dtype:
float64
```

These numbers are difficult to interpret without some measure of their uncertainty. We can compute these uncertainties quickly using bootstrap resamplings of the data:

```
In[25]: from sklearn.utils import resample
        np.random.seed(1)

        err = np.std([model.fit(*resample(X, y)).coef_

                      for i in range(1000)], 0)
```

With these errors estimated, let's again look at the results:

```
In[26]: print(pd.DataFrame({'effect': params.round(0),

                            'error': err.round(0)}))
```

```
             effec  error
             t
Mon          504    85
Tue          612    82
Wed          592    82
Thu          481    85
Fri          177    81
Sat          -1104  79
Sun          -1135  82
holiday      -1187  164
daylight_h   129    9
rs
PRCP         -665   62
dry day      546    33
Temp (C)     65     4
annual       28     18
```

We first see that there is a relatively stable trend in the weekly baseline: there are many more riders on weekdays than on weekends and holidays. We see that for each additional hour of daylight, 129 ± 9 more people choose to ride; a temperature increase of one degree Celsius encourages 65 ± 4 people to grab their bicycle; a dry day means an average of 546 ± 33 more riders; and each inch of precipitation means 665 ± 62 more people leave their bike at home. Once all these effects are accounted for, we see a modest increase of 28 ± 18 new daily riders each year.

Our model is almost certainly missing some relevant information. For example, non- linear effects (such as effects of precipitation *and* cold temperature) and nonlinear trends within each variable (such as disinclination to ride at very cold and very hot temperatures) cannot be accounted for in this model. Additionally, we have thrown away some of the finer-grained information (such as the difference between a rainy morning and a rainy afternoon), and we have ignored correlations between days(such as the possible effect of a rainy Tuesday on Wednesday's numbers, or the effectof an unexpected sunny day after a streak of rainy days). These are all potentially interesting effects, and you now have the tools to begin exploring them if you wish!

Day-04: Support Vector Machines

Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression. In this section, we will develop the intuition behind support vector machines and their use in classification problems. We begin with the standard imports:

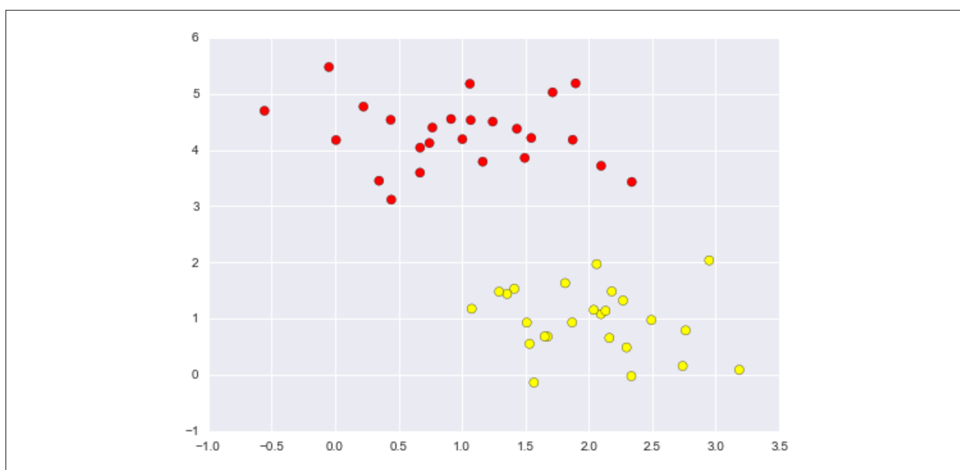
```
In[1]: %matplotlib inline

       import numpy as np

       import matplotlib.pyplot as plt
       from scipy import stats


       # use Seaborn plotting defaults

       import seaborn as sns; sns.set()
```

Motivating Support Vector Machines

As part of our discussion of Bayesian classification (see "In Depth: Naive Bayes Clas-sification" on page 382), we learned a simple model describing the distribution of each underlying class, and used these generative models to probabilistically deter- mine labels for new points. That was an example of *generative classification*; here wewill consider instead *discriminative classification*: rather than modeling each class, wesimply find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

As an example of this, consider the simple case of a classification task, in which the two classes of points are well separated (Figure 5-53):

```
In[2]: from sklearn.datasets.samples_generator import make_blobs
       X, y = make_blobs(n_samples=50,
       centers=2,random_state=0, cluster_std=0.60)
```




```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

*Figure 5-53. Simple data for **classification***

A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification. For two-dimensional data like that shown here, this is a task we could do by hand. But immediately we seea problem: there is more than one possible dividing line that can perfectly discrimi- nate between the two classes!

We can draw them as follows (Figure 5-54):

```
In[3]: xfit = np.linspace(-1, 3.5)

       plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

       plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2, markersize=10)

       for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:

           plt.plot(xfit, m * xfit + b, '-k')

       plt.xlim(-1, 3.5);
```

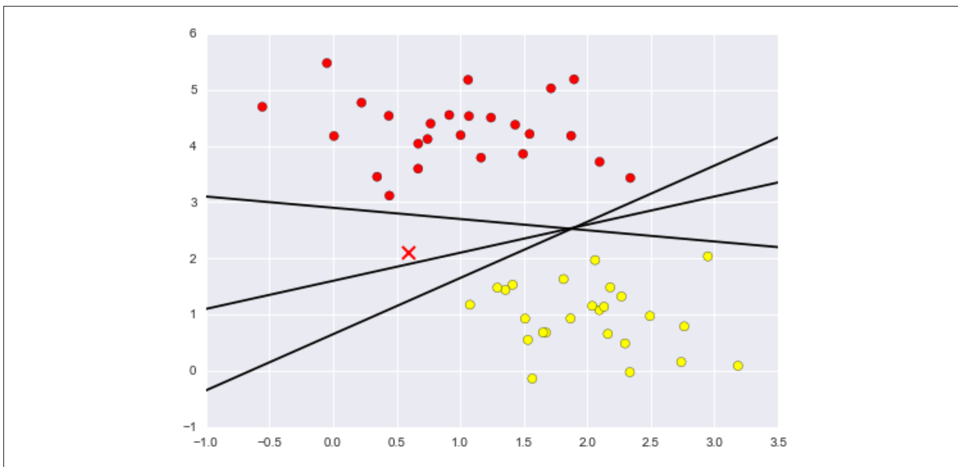


*Figure 5-54. **Three** perfect linear discriminative **classifiers** for our data*

These are three *very* different separators that, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (e.g., the one marked by the "X" in Figure 5-54) will be assigned a different label! Evidently oursimple intuition of "drawing a line between classes" is not enough, and we need to think a bit deeper.

Support Vector Machines: Maximizing the Margin

Support vector machines offer one way to improve on this. The intuition is this:rather than simply drawing a zero-width line between the classes, we can drawaround each line a *margin* of some width, up to the nearest point. Here is an example of how this might

look (Figure 5-55):

```
In[4]:

xfit = np.linspace(-1, 3.5)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:

    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')

    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none', color='#AAAAAA',
                     alpha=0.4)


plt.xlim(-1, 3.5);
```

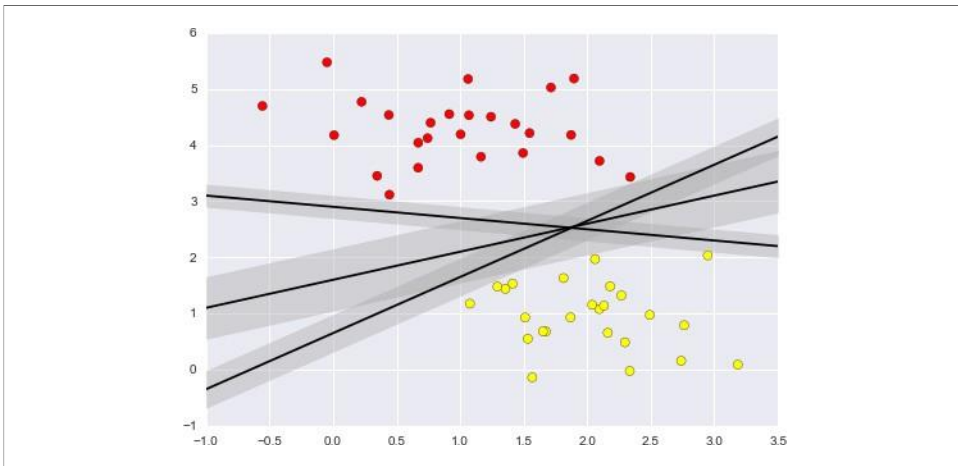


Figure 5-55. Visualization of "margins" within discriminative ***classifiers***

In support vector machines, the line that maximizes this margin is the one we will choose as the optimal model. Support vector machines are an example of such a *max-imum margin* estimator.

Fitting a support vector machine

Let's see the result of an actual fit to this data: we will use Scikit-Learn's support vector classifier to train an SVM model on this data. For the time being, we will use a linearkernel and set the C parameter to a very large number (we'll discuss the meaning of these in more depth momentarily):

```
In[5]: from sklearn.svm import SVC # "Support vector classifier"
```

```
        model = SVC(kernel='linear', C=1E10)
        model.fit(X, y)
```

```
Out[5]: SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0,
          decision_function_shape=None, degree=3, gamma='auto', kernel='linear',
          max_iter=-1, probability=False, random_state=None, shrinking=True,
          tol=0.001, verbose=False)
```

To better visualize what's happening here, let's create a quick convenience function that will plot SVM decision boundaries for us (Figure 5-56):

```
In[6]: def plot_svc_decision_function(model, ax=None, plot_support=True):
           """Plot the decision function for a two-dimensional SVC"""
           if ax is None:
               ax = plt.gca()
           xlim = ax.get_xlim()
           ylim = ax.get_ylim()

           # create grid to evaluate model
           x = np.linspace(xlim[0], xlim[1], 30)
           y = np.linspace(ylim[0], ylim[1], 30)Y, X = np.meshgrid(y, x)
           xy = np.vstack([X.ravel(), Y.ravel()]).T
           P = model.decision_function(xy).reshape(X.shape)

           # plot decision boundary and margins
           ax.contour(X, Y, P, colors='k',
                      levels=[-1, 0, 1], alpha=0.5,
                      linestyles=['--', '-', '--'])

           # plot support vectors
           if plot_support:
               ax.scatter(model.support_vectors_[:, 0],
                          model.support_vectors_[:, 1],
                          s=300, linewidth=1, facecolors='none');
           ax.set_xlim(xlim)
           ax.set_ylim(ylim)
```

```
In[7]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
       plot_svc_decision_function(model);
```

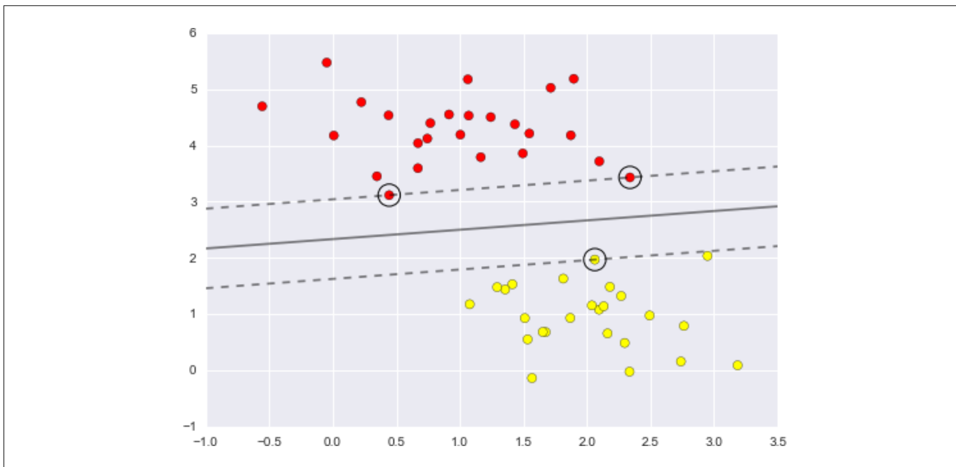


*Figure 5-56. A support vector machine **classifier fit** to the data, with margins (dashed lines) and support vectors (circles) shown*

This is the dividing line that maximizes the margin between the two sets of points. Notice that a few of the training points just touch the margin; they are indicated bythe black circles in Figure 5-56. These points are the pivotal elements of this fit, and are known as the *support vectors*, and give the algorithm its name. In Scikit-Learn, theidentity of these points is stored in the `support_vectors_` attribute of the classifier:

```
In[8]: model.support_vectors_

Out[8]: array([[          3.11530945]
0.44359863,               ,
[ 2.33812285,             3.43116792]
                          ,
[ 2.06156753,             1.96918596]
                          ])
```

A key to this classifier's success is that for the fit, only the position of the support vec-tors matters; any points further from the margin that are on the correct side do not modify the fit! Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

We can see this, for example, if we plot the model learned from the first 60 points andfirst 120 points of this dataset (Figure 5-57):

```
In[9]: def plot_svm(N=10, ax=None):

           X, y = make_blobs(n_samples=200, centers=2,

                             random_state=0, cluster_std=0.60)

           X = X[:N]
```

```
        y = y[:N]

        model = SVC(kernel='linear', C=1E10)
        model.fit(X, y)

        ax = ax or plt.gca()

        ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        ax.set_xlim(-1, 4)

        ax.set_ylim(-1, 6)
        plot_svc_decision_function(model, ax)

    fig, ax = plt.subplots(1, 2, figsize=(16, 6))
    fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
    for axi, N in zip(ax, [60, 120]):

        plot_svm(N, axi)

        axi.set_title('N = {0}'.format(N))
```

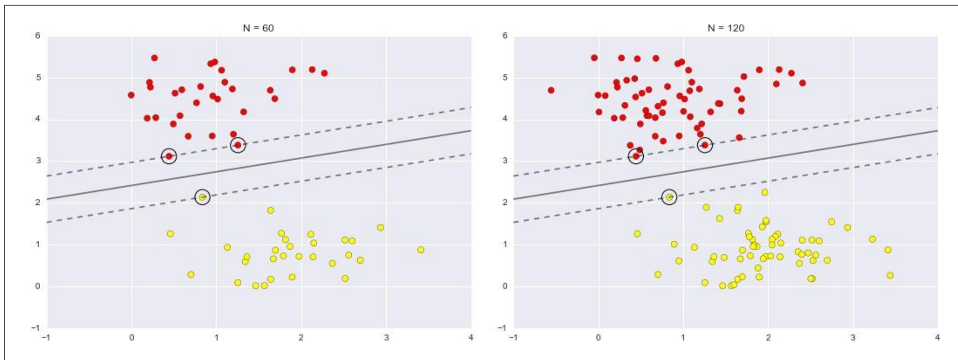


Figure 5-57. The influence of new training points on the SVM model

In the left panel, we see the model and the support vectors for 60 training points. In the right panel, we have doubled the number of training points, but the model hasnot changed: the three support vectors from the left panel are still the support vectors from the right panel. This insensitivity to the exact behavior of distant points is one ofthe strengths of the SVM model.

If you are running this notebook live, you can use IPython's interactive widgets toview this feature of the SVM model interactively (Figure 5-58):

```
In[10]: from ipywidgets import interact, fixed
        interact(plot_svm, N=[10, 200], ax=fixed(None));
```

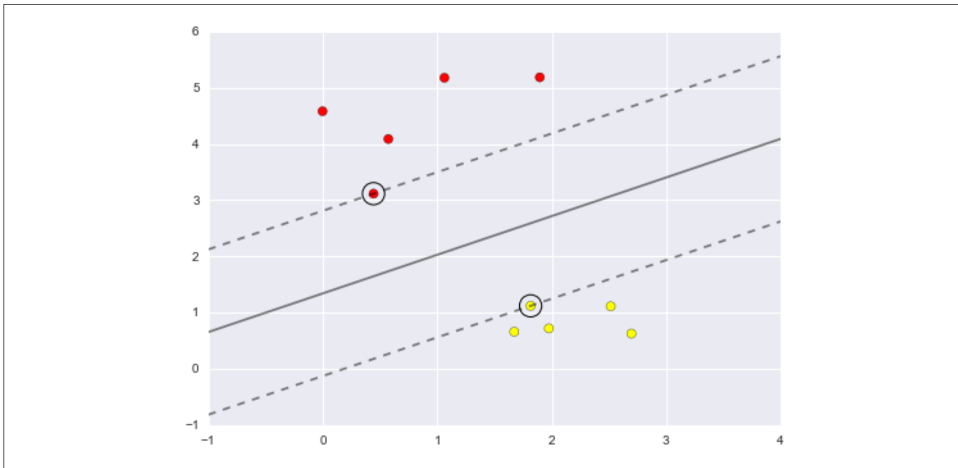


*Figure 5-58. **The first** frame of the interactive SVM visualization (see the online appen-dix for the full version)*

Beyond linear boundaries: Kernel SVM

Where SVM becomes extremely powerful is when it is combined with *kernels*. We have seen a version of kernels before, in the basis function regressions of "In Depth: Linear Regression" on page 390. There we projected our data into higher-dimensionalspace defined by polynomials and Gaussian basis functions, and thereby were able tofit for nonlinear relationships with a linear classifier.

In SVM models, we can use a version of the same idea. To motivate the need for ker-nels, let's look at some data that is not linearly separable (Figure 5-59):

```
In[11]: from sklearn.datasets.samples_generator import make_circles
        X, y = make_circles(100, factor=.1, noise=.1)


        clf = SVC(kernel='linear').fit(X, y)


        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(clf, plot_support=False);
```

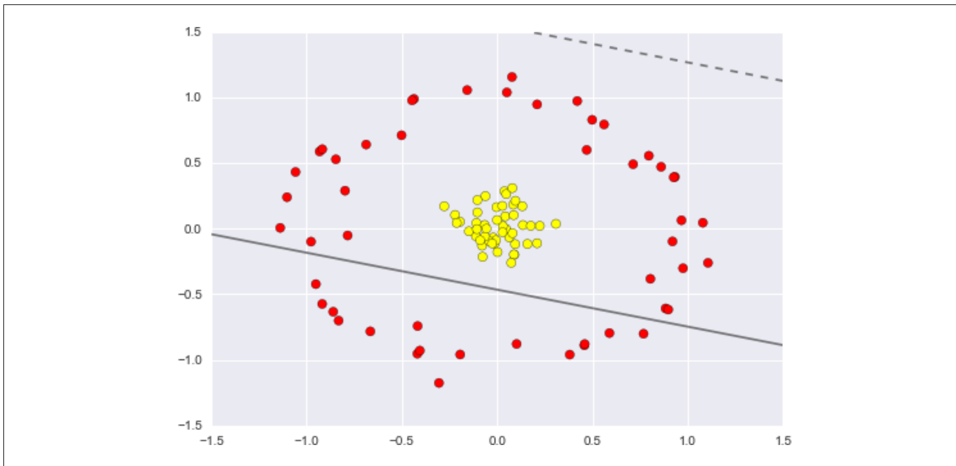


*Figure 5-59. A linear **classifier** performs poorly for nonlinear boundaries*

It is clear that no linear discrimination will *ever* be able to separate this data. But we can draw a lesson from the basis function regressions in "In Depth: Linear Regres- sion" on page 390, and think about how we might project the data into a higher dimension such that a linear separator *would* be sufficient. For example, one simple projection we could use would be to compute a *radial basis function* centered on the middle clump:

```
In[12]: r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot—if you arerunning this notebook live, you will be able to use the sliders to rotate the plot(Figure 5-60):

```
In[13]: from mpl_toolkits import mplot3d


        def plot_3D(elev=30, azim=30, X=X, y=y):
            ax = plt.subplot(projection='3d')

            ax.scatter3D(X[:, 0], X[:, 1], r, c=y, s=50, cmap='autumn')
            ax.view_init(elev=elev, azim=azim)

            ax.set_xlabel('x')
            ax.set_ylabel('y')
            ax.set_zlabel('r')


        interact(plot_3D, elev=[-90, 90], azip=(-180, 180),
                 X=fixed(X), y=fixed(y));
```

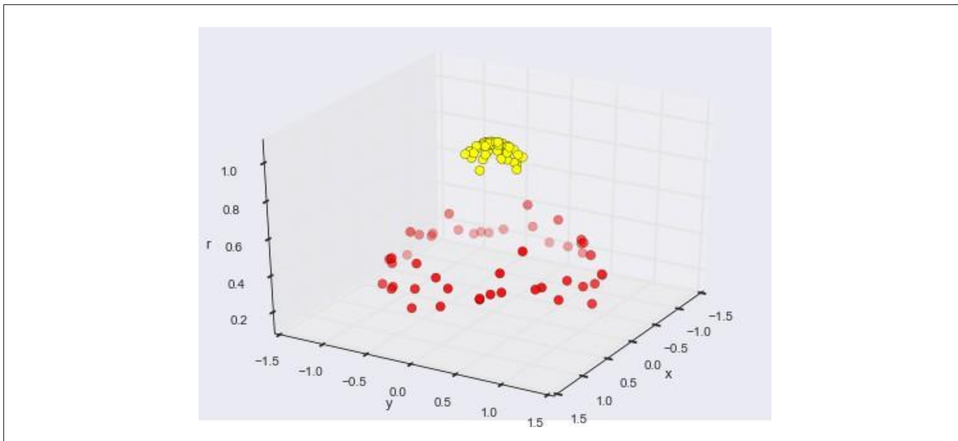


Figure 5-60. A third dimension added to the data allows for linear separation

We can see that with this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, *r*=0.7.

Here we had to choose and carefully tune our projection; if we had not centered our radial basis function in the right location, we would not have seen such clean, linearlyseparable results. In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use.

One strategy to this end is to compute a basis function centered at *every* point in the dataset, and let the SVM algorithm sift through the results. This type of basis functiontransformation is known as a *kernel transformation*, as it is based on a similarity rela-tionship (or kernel) between each pair of points.

A potential problem with this strategy—projecting *N* points into *N* dimensions—is that it might become very computationally intensive as *N* grows large. However, because of a neat little procedure known as the *kernel trick*, a fit on kernel- transformed data can be done implicitly—that is, without ever building the full *N*- dimensional representation of the kernel projection! This kernel trick is built into the SVM, and is one of the reasons the method is so powerful.

In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel toan RBF (radial basis function) kernel, using the `kernel` model hyperparameter (Figure 5-61):

```
In[14]: clf = SVC(kernel='rbf', C=1E6)
        clf.fit(X, y)

Out[14]: SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape=None, degree=3, gamma='auto', kernel='rbf',
           max_iter=-1, probability=False, random_state=None, shrinking=True,
           tol=0.001, verbose=False)

In[15]: plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
        plot_svc_decision_function(clf)
```

```
plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1],
            s=300, lw=1, facecolors='none');
```

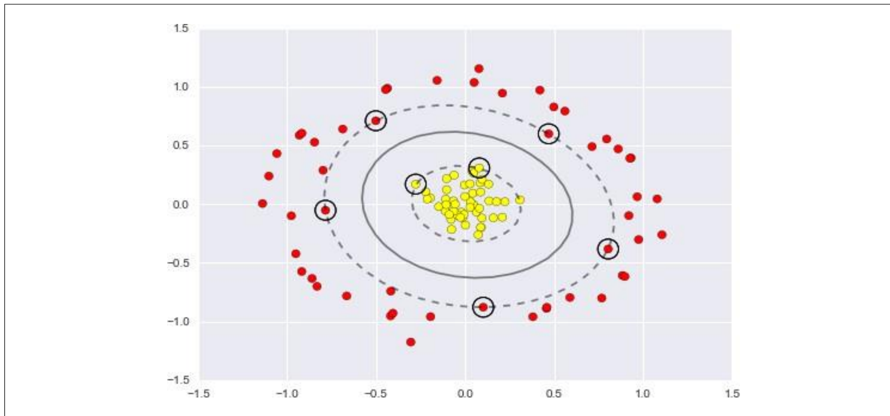


Figure 5-61. Kernel SVM fit to the data

Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning toturn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used.

Tuning the SVM: Softening margins

Our discussion so far has centered on very clean datasets, in which a perfect decision boundary exists. But what if your data has some amount of overlap? For example, youmay have data like this (Figure 5-62):

```
In[16]: X, y = make_blobs(n_samples=100, centers=2,

                          random_state=0, cluster_std=1.2)
        plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```

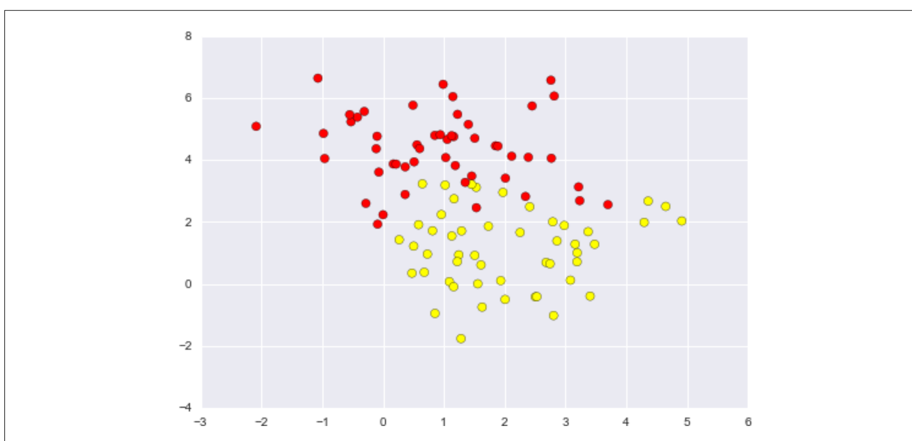


Figure 5-62. Data with some level of overlap

To handle this case, the SVM implementation has a bit of a fudge-factor that "softens"the margin; that is, it allows some of the points to creep into the margin if that allows a better fit. The hardness of the margin is controlled by a tuning parameter, mostoften known as *C*. For very large *C*, the margin is hard, and points cannot lie in it. Forsmaller *C*, the margin is softer, and can grow to encompass some points.

The plot shown in Figure 5-63 gives a visual picture of how a changing *C* parameter affects the final fit, via the softening of the margin:

```
In[17]: X, y = make_blobs(n_samples=100, centers=2,
                          random_state=0, cluster_std=0.8)

        fig, ax = plt.subplots(1, 2, figsize=(16, 6))
        fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

        for axi, C in zip(ax, [10.0, 0.1]):
            model = SVC(kernel='linear', C=C).fit(X, y)
            axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
            plot_svc_decision_function(model, axi)
            axi.scatter(model.support_vectors_[:, 0],
                        model.support_vectors_[:, 1],
                        s=300, lw=1, facecolors='none');
            axi.set_title('C = {0:.1f}'.format(C), size=14)
```

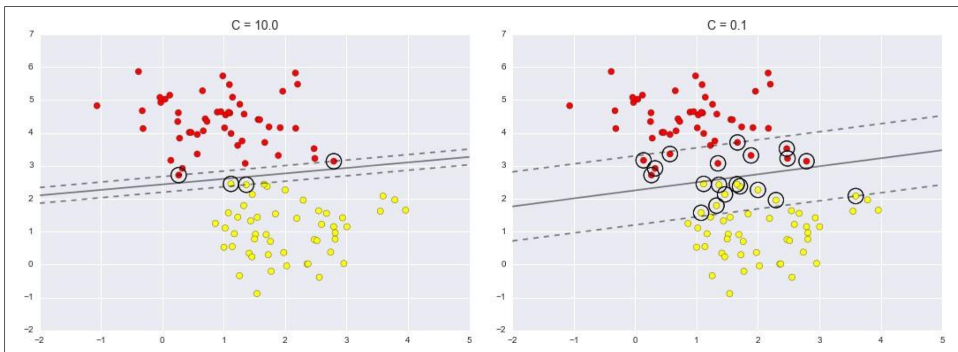


*Figure 5-63. **The effect** of the C parameter on the support vector **fit***

The optimal value of the *C* parameter will depend on your dataset, and should be tuned via cross-validation or a similar procedure (refer back to "Hyperparametersand Model Validation" on page 359 for further information).

Example: Face Recognition

As an example of support vector machines in action, let's take a look at the facial rec-ognition problem. We will use the Labeled Faces in the Wild dataset, which consists of several

thousand collated photos of various public figures. A fetcher for the datasetis built into Scikit-Learn:

```
In[18]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=60)
        print(faces.target_names)
        print(faces.images.shape)

['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Let's plot a few of these faces to see what we're working with (Figure 5-64):

```
In[19]: fig, ax = plt.subplots(3, 5)

        for i, axi in enumerate(ax.flat):
            axi.imshow(faces.images[i], cmap='bone')
            axi.set(xticks=[], yticks=[],

                    xlabel=faces.target_names[faces.target[i]])
```

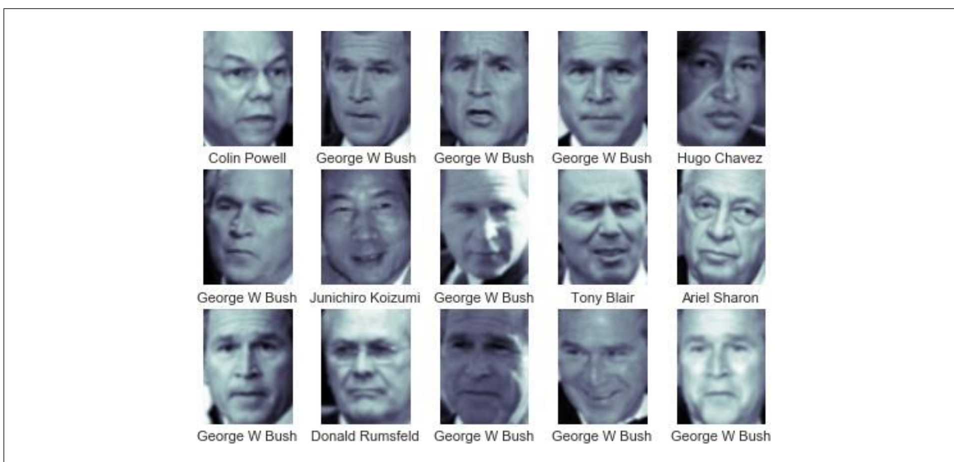


Figure 5-64. Examples from the Labeled Faces in the Wild dataset

Each image contains [62×47] or nearly 3,000 pixels. We could proceed by simply using each pixel value as a feature, but often it is more effective to use some sort of preprocessor to extract more meaningful features; here we will use a principal com- ponent analysis (see "In Depth: Principal Component Analysis" on page 433) to extract 150 fundamental components to feed into our support vector machine classi- fier. We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

```
In[20]: from sklearn.svm import SVC

        from sklearn.decomposition import RandomizedPCA

        from sklearn.pipeline import make_pipeline
```

```
pca = RandomizedPCA(n_components=150, whiten=True, random_state=42)
svc = SVC(kernel='rbf', class_weight='balanced')

model = make_pipeline(pca, svc)
```

For the sake of testing our classifier output, we will split the data into a training and testing set:

```
In[21]: from sklearn.cross_validation import train_test_split

        Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data, faces.target,

                                                        random_state=42)
```

Finally, we can use a grid search cross-validation to explore combinations of parame-ters. Here we will adjust `C` (which controls the margin hardness) and `gamma` (which controls the size of the radial basis function kernel), and determine the best model:

```
In[22]: from sklearn.grid_search import GridSearchCV
        param_grid = {'svc_C': [1, 5, 10, 50],

                      'svc__gamma': [0.0001, 0.0005, 0.001, 0.005]}

        grid = GridSearchCV(model, param_grid)

        %time grid.fit(Xtrain, ytrain)

        print(grid.best_params_)

CPU times: user 47.8 s, sys: 4.08 s, total: 51.8 s
Wall time: 26 s

{'svc__gamma': 0.001, 'svc__C': 10}
```

The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

Now with this cross-validated model, we can predict the labels for the test data, whichthe model has not yet seen:

```
In[23]: model = grid.best_estimator_
        yfit = model.predict(Xtest)
```

Let's take a look at a few of the test images along with their predicted values(Figure 5-65):

```
In[24]: fig, ax = plt.subplots(4, 6)

        for i, axi in enumerate(ax.flat):
            axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
            axi.set(xticks=[], yticks=[])
            axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
```

```
                                color='black' if yfit[i] == ytest[i] else 'red')
        fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```

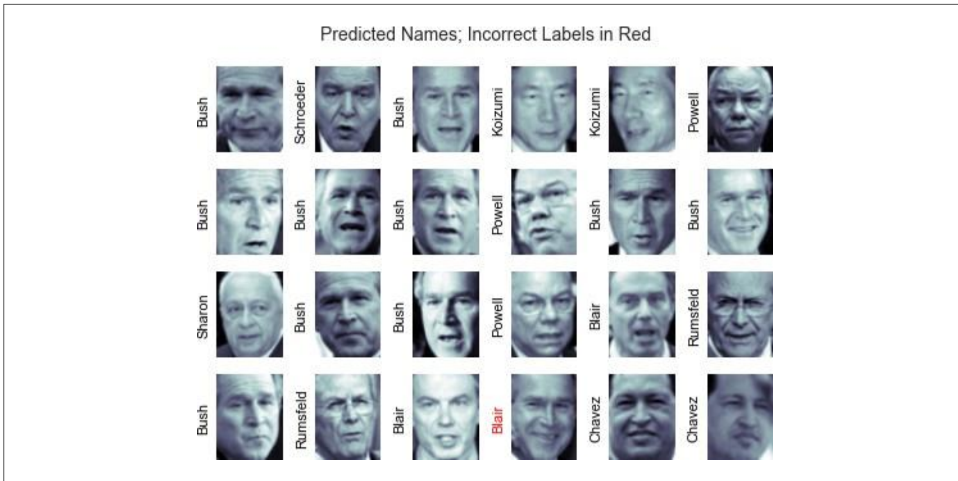


Figure 5-65. Labels predicted by our model

Out of this small sample, our optimal estimator mislabeled only a single face (Bush'sface in the bottom row was mislabeled as Blair). We can get a better sense of our esti-mator's performance using the classification report, which lists recovery statistics label by label:

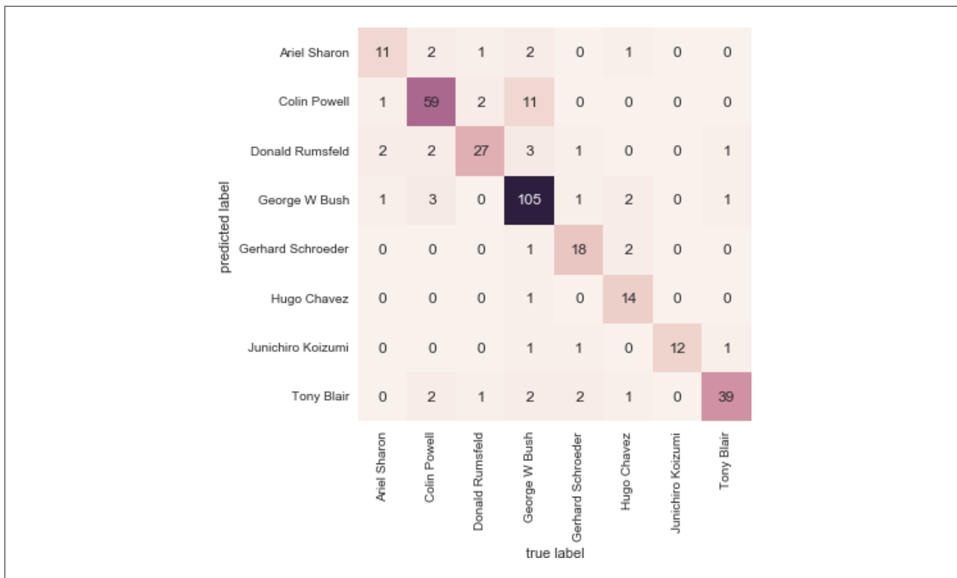
```
In[25]: from sklearn.metrics import classification_report

        print(classification_report(ytest, yfit,

                                    target_names=faces.target_names))
```

```
                   precision    recall  f1-score   support

     Ariel Sharon       0.65      0.73      0.69        15
     Colin Powell       0.81      0.87      0.84        68
  Donald Rumsfeld       0.75      0.87      0.81        31
    George W Bush       0.93      0.83      0.88       126
Gerhard Schroeder       0.86      0.78      0.82        23
      Hugo Chavez       0.93      0.70      0.80        20
Junichiro Koizumi       0.80      1.00      0.89        12
       Tony Blair       0.83      0.93      0.88        42

      avg / total       0.85      0.85      0.85       337
```

We might also display the confusion matrix between these classes (Figure 5-66):

```
In[26]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(ytest, yfit)
```

```
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                xticklabels=faces.target_names,
```




```
                yticklabels=faces.target_names)plt.xlabel('true
                label') plt.ylabel('predicted label');
```

Figure 5-66. Confusion matrix for the faces data

This helps us get a sense of which labels are likely to be confused by the estimator.

For a real-world facial recognition task, in which the photos do not come precropped into nice grids, the only difference in the facial classification scheme is the feature selection: you would need to use a more sophisticated algorithm to find the faces, andextract features that are independent of the pixellation. For this kind of application, one good option is to make use of OpenCV, which among other things, includes pre- trained implementations of state-of-the-art feature extraction tools for images in gen- eral and faces in particular.

Day-05: Decision Trees and Random Forests

Previously we have looked in depth at a simple generative classifier (naive Bayes; see"In Depth: Naive Bayes Classification" on page 382) and a powerful discriminativeclassifier (support vector machines; see "In-Depth: Support Vector Machines" onpage 405). Here we'll take a look at motivating another powerful algorithm—a non-parametric algorithm called *random forests*. Random forests are an example of an *ensemble* method, a method that relies on aggregating the results of an ensemble ofsimpler estimators. The somewhat surprising result with such ensemble methods isthat the sum can be greater than the parts; that is, a majority vote among a number ofestimators can end up being better than any of the individual estimators doing thevoting! We will see examples of this in the following sections. We begin with the stan-dard imports:

```
In[1]: %matplotlib inline
```

```
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Motivating Random Forests: Decision Trees

Random forests are an example of an *ensemble learner* built on decision trees. For thisreason we'll start by discussing decision trees themselves.

Decision trees are extremely intuitive ways to classify or label objects: you simply aska series of questions designed to zero in on the classification. For example, if you wanted to build a decision tree to classify an animal you come across while on a hike,you might construct the one shown in Figure 5-67.

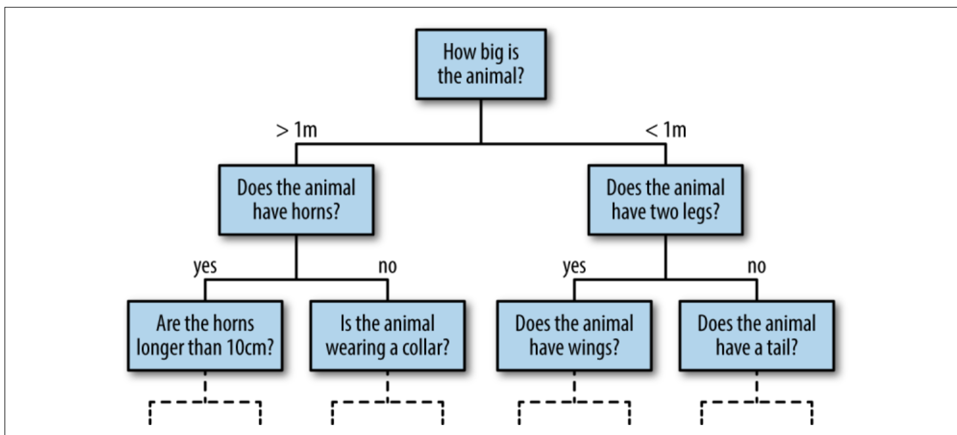


Figure 5-67. An example of a binary decision tree

The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrow- ing the options even among a large number of classes. The trick, of course, comes in deciding which questions to ask at each step. In machine learning implementations ofdecision trees, the questions generally take the form of axis-aligned splits in the data; that is, each node in the tree splits the data into two groups using a cutoff valuewithin one of the features. Let's now take a look at an example.

Creating a decision tree

Consider the following two-dimensional data, which has one of four class labels (Figure 5-68):

```
In[2]: from sklearn.datasets import make_blobs
```

```
X, y = make_blobs(n_samples=300, centers=4,random_state=0, cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

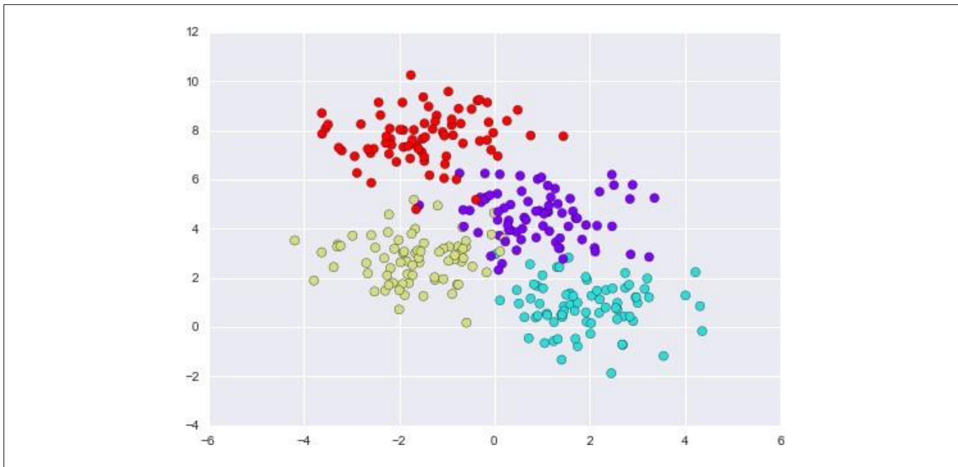


*Figure 5-68. Data for the decision tree **classifier***

A simple decision tree built on this data will iteratively split the data along one or theother axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it. Figure 5-69 presents a visualization of the first four levels of a decision tree classifier for this data.

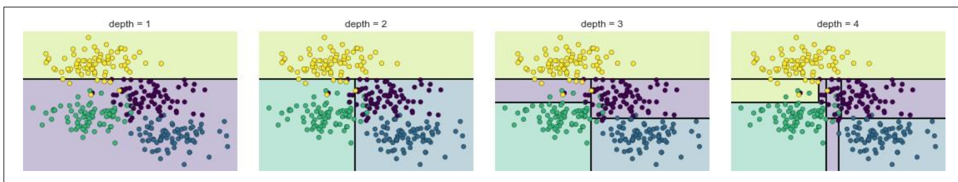


Figure 5-69. Visualization of how the decision tree splits the data

Notice that after the first split, every point in the upper branch remains unchanged,so there is no need to further subdivide this branch. Except for nodes that contain all of one color, at each level *every* region is again split along one of the two features.

This process of fitting a decision tree to our data can be done in Scikit-Learn with the `DecisionTreeClassifier` estimator:

```
In[3]: from sklearn.tree import DecisionTreeClassifier
       tree = DecisionTreeClassifier().fit(X, y)
```

Let's write a quick utility function to help us visualize the output of the classifier:

```
In[4]: def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
           ax = ax or plt.gca()

           # Plot the training points
```

```
ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,

           clim=(y.min(), y.max()), zorder=3)
ax.axis('tight')

ax.axis('off')

xlim = ax.get_xlim()
ylim = ax.get_ylim()


# fit the estimator

model.fit(X, y)

xx, yy = np.meshgrid(np.linspace(*xlim, num=200),

                     np.linspace(*ylim, num=200))

Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)


# Create a color plot with the results

n_classes = len(np.unique(y))

contours = ax.contourf(xx, yy, Z, alpha=0.3,

                       levels=np.arange(n_classes + 1) - 0.5,
                       cmap=cmap, clim=(y.min(), y.max()),
                       zorder=1)


ax.set(xlim=xlim, ylim=ylim)
```

Now we can examine what the decision tree classification looks like (Figure 5-70):

```
In[5]: visualize_classifier(DecisionTreeClassifier(), X, y)
```

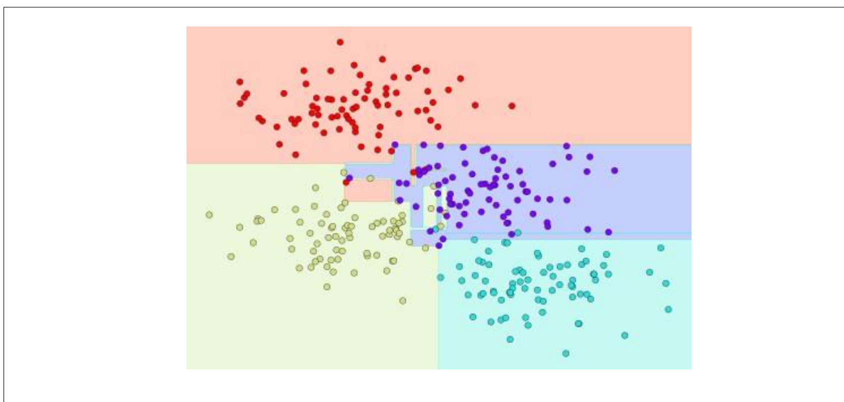


*Figure 5-70. Visualization of a decision tree **classification***

If you're running this notebook live, you can use the helpers script included in the online

appendix to bring up an interactive visualization of the decision tree building process (Figure 5-71):

```
In[6]: # helpers_05_08 is found in the online appendix
       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.plot_tree_interactive(X, y);
```

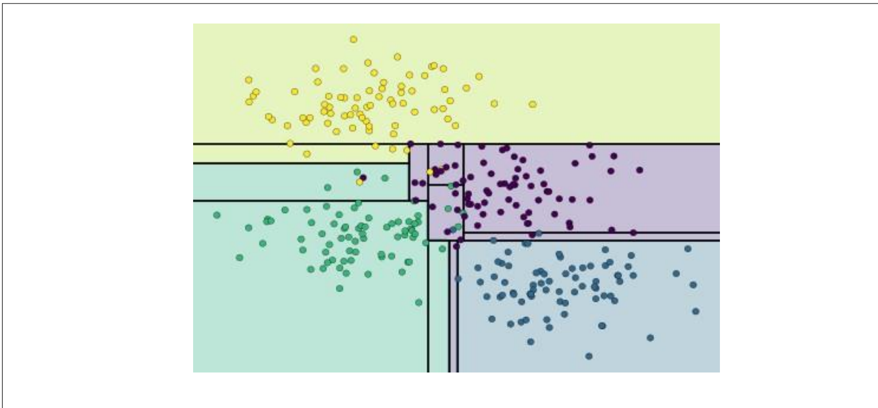


Figure 5-71. First frame of the interactive decision tree widget; for the full version, see the online appendix

Notice that as the depth increases, we tend to get very strangely shaped classificationregions; for example, at a depth of five, there is a tall and skinny purple region between the yellow and blue regions. It's clear that this is less a result of the true, intrinsic data distribution, and more a result of the particular sampling or noise prop- erties of the data. That is, this decision tree, even at only five levels deep, is clearly overfitting our data.

Decision trees and overfitting

Such overfitting turns out to be a general property of decision trees; it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions they are drawn from. Another way to see this overfitting is to look at models trained on different subsets of the data—for example, in Figure 5-72 we train two different trees, each on half of the original data.

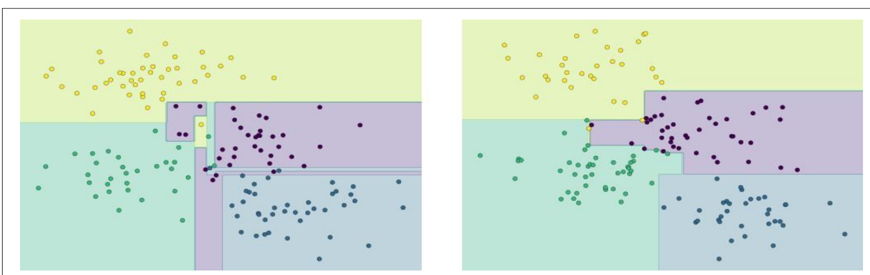


Figure 5-72. An example of two randomized decision trees

It is clear that in some places, the two trees produce consistent results (e.g., in thefour corners), while in other places, the two trees give very different classifications (e.g., in the regions between any two clusters). The key observation is that the incon- sistencies tend to happen where the classification is less certain, and thus by using information from *both* of these trees, we might come up with a better result!

If you are running this notebook live, the following function will allow you to interac- tively display the fits of trees trained on a random subset of the data (Figure 5-73):

```
In[7]: # helpers_05_08 is found in the online appendix

       # (https://github.com/jakevdp/PythonDataScienceHandbook)
       import helpers_05_08
       helpers_05_08.randomized_tree_interactive(X, y)
```

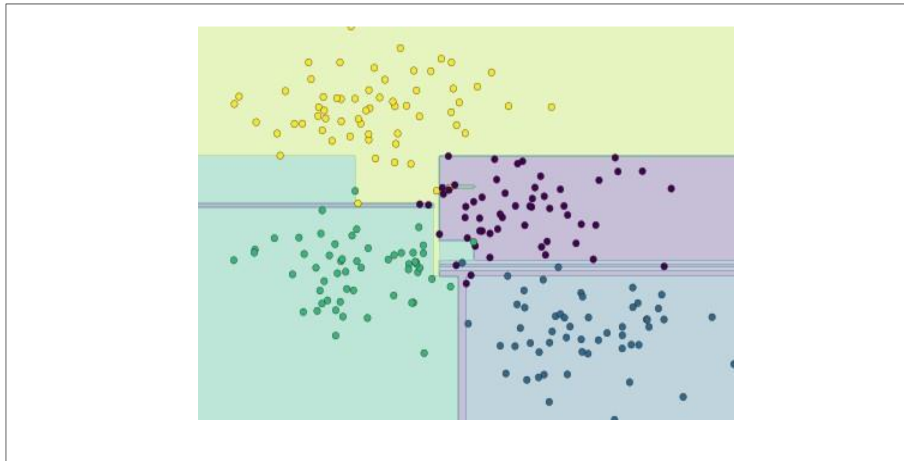


Figure 5-73. First frame of the interactive randomized decision tree widget; for the full version, see the online appendix

Just as using information from two trees improves our results, we might expect thatusing information from many trees would improve our results even further.

Ensembles of Estimators: Random Forests

This notion—that multiple overfitting estimators can be combined to reduce theeffect of this overfitting—is what underlies an ensemble method called *bagging*. Bag-ging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which overfits the data, and averages the results to find a better classification. An ensemble of randomized decision trees is known as a *random forest*.

We can do this type of bagging classification manually using Scikit-Learn's `Bagging Classifier` meta-estimator as shown here (Figure 5-74):

```
In[8]: from sklearn.tree import DecisionTreeClassifier

       from sklearn.ensemble import BaggingClassifier
```

```
tree = DecisionTreeClassifier()

bag = BaggingClassifier(tree, n_estimators=100, max_samples=0.8,
                        random_state=1)


bag.fit(X, y)
visualize_classifier(bag, X, y)
```

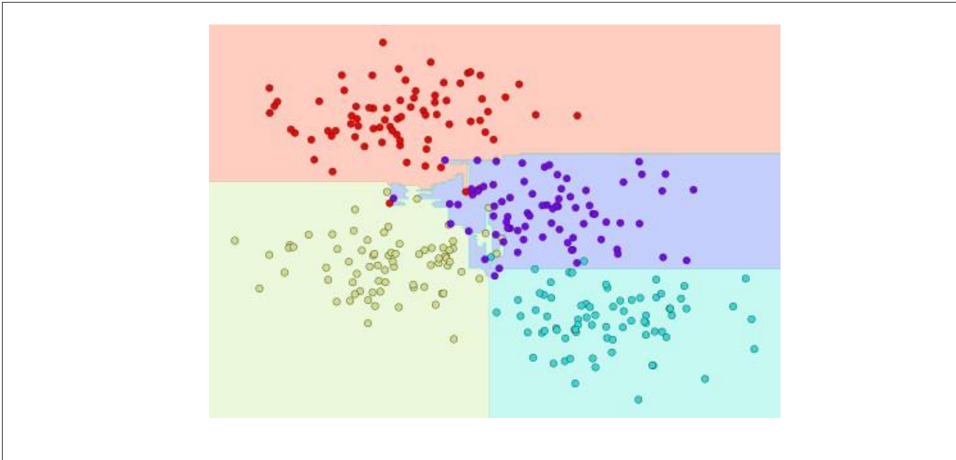


Figure 5-74. Decision boundaries for an ensemble of random decision trees

In this example, we have randomized the data by fitting each estimator with a ran- dom subset of 80% of the training points. In practice, decision trees are more effec- tively randomized when some stochasticity is injected in how the splits are chosen; this way, all the data contributes to the fit each time, but the results of the fit still havethe desired randomness. For example, when determining which feature to split on,the randomized tree might select from among the top several features. You can read more technical details about these randomization strategies in the Scikit-Learn docu- mentation and references within.

In Scikit-Learn, such an optimized ensemble of randomized decision trees is imple- mented in the `RandomForestClassifier` estimator, which takes care of all the ran- domization automatically. All you need to do is select a number of estimators, and it will very quickly (in parallel, if desired) fit the ensemble of trees (Figure 5-75):

```
In[9]: from sklearn.ensemble import RandomForestClassifier


       model = RandomForestClassifier(n_estimators=100, random_state=0)
       visualize_classifier(model, X, y);
```

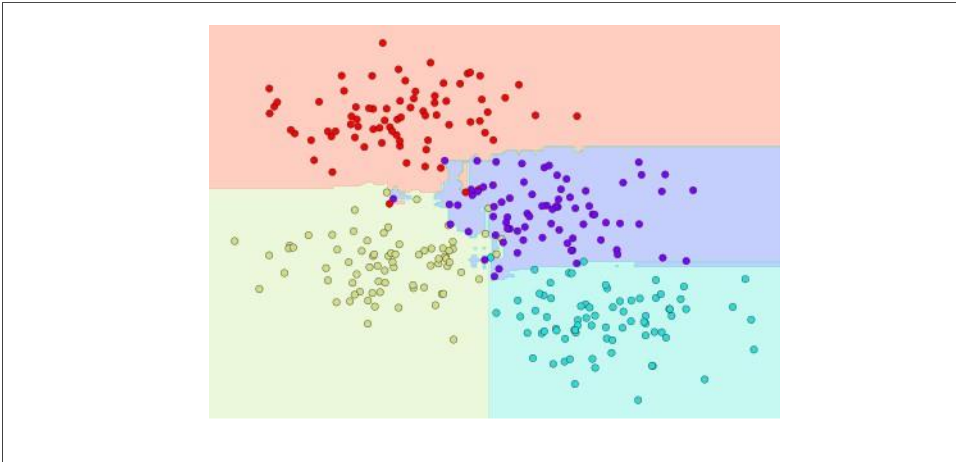


Figure 5-75. Decision boundaries for a random forest, which is an optimized ensemble of decision trees

We see that by averaging over 100 randomly perturbed models, we end up with an overall model that is much closer to our intuition about how the parameter space should be split.

Random Forest Regression

In the previous section we considered random forests within the context of classifica-tion. Random forests can also be made to work in the case of regression (that is, con- tinuous rather than categorical variables). The estimator to use for this is the `RandomForestRegressor`, and the syntax is very similar to what we saw earlier.

Consider the following data, drawn from the combination of a fast and slow oscilla-tion (Figure 5-76):

```
In[10]: rng = np.random.RandomState(42)
        x = 10 * rng.rand(200)


        def model(x, sigma=0.3):
            fast_oscillation = np.sin(5 * x)
            slow_oscillation = np.sin(0.5 * x)
            noise = sigma * rng.randn(len(x))


            return slow_oscillation + fast_oscillation + noise

        y = model(x)

        plt.errorbar(x, y, 0.3, fmt='o');
```

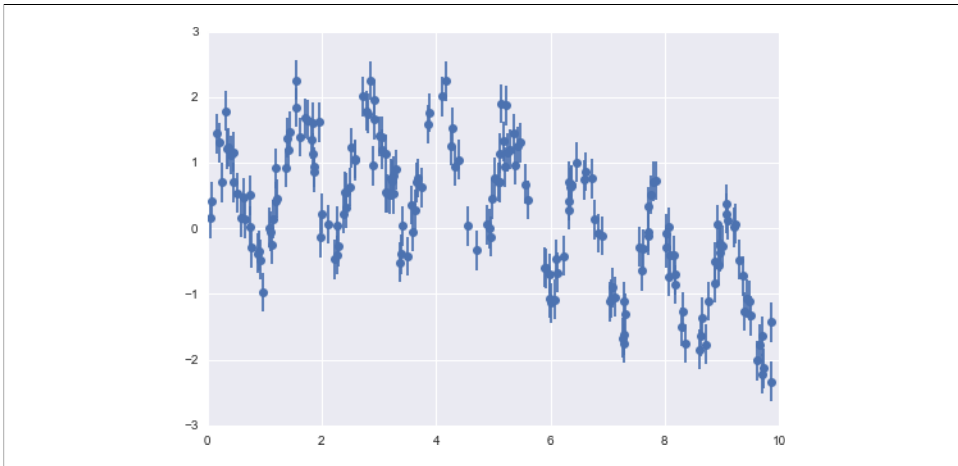


Figure 5-76. Data for random forest regression

Using the random forest regressor, we can find the best-fit curve as follows (Figure 5-77):

```
In[11]: from sklearn.ensemble import RandomForestRegressor
        forest = RandomForestRegressor(200)
        forest.fit(x[:, None], y)


        xfit = np.linspace(0, 10, 1000)

        yfit = forest.predict(xfit[:, None])
        ytrue = model(xfit, sigma=0)


        plt.errorbar(x, y, 0.3, fmt='o', alpha=0.5)
        plt.plot(xfit, yfit, '-r');

        plt.plot(xfit, ytrue, '-k', alpha=0.5);
```

Here the true model is shown by the smooth curve, while the random forest model is shown by the jagged curve. As you can see, the nonparametric random forest modelis flexible enough to fit the multiperiod data, without us needing to specify a multi- period model!

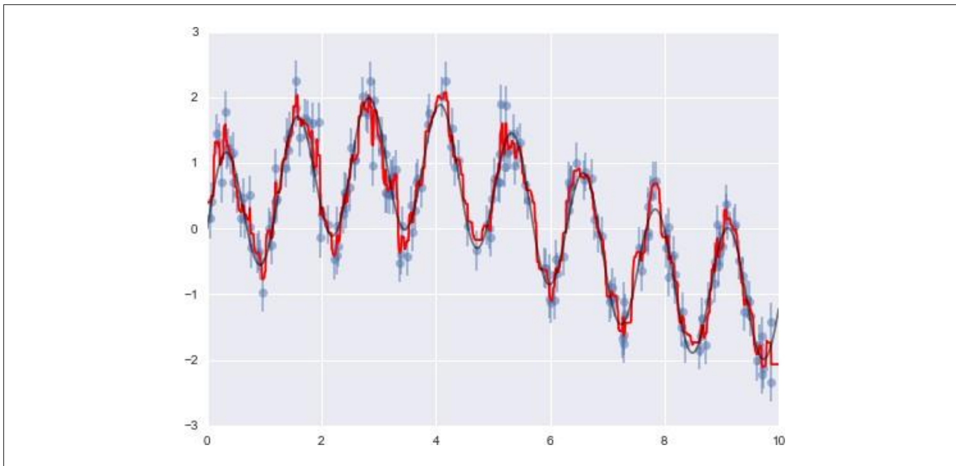


Figure 5-77. Random forest model fit to the data

Example: Random Forest for Classifying Digits

Earlier we took a quick look at the handwritten digits data (see "Introducing Scikit- Learn" on page 343). Let's use that again here to see how the random forest classifiercan be used in this context.

```
In[12]: from sklearn.datasets import load_digits
        digits = load_digits()

        digits.keys()

Out[12]: dict_keys(['target', 'data', 'target_names', 'DESCR', 'images'])
```

To remind us what we're looking at, we'll visualize the first few data points (Figure 5-78):

```
In[13]:

# set up the figure

fig = plt.figure(figsize=(6, 6))  # figure size in inches

fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)


# plot the digits: each image is 8x8 pixels

for i in range(64):

    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')


    # label the image with the target value

    ax.text(0, 7, str(digits.target[i]))
```

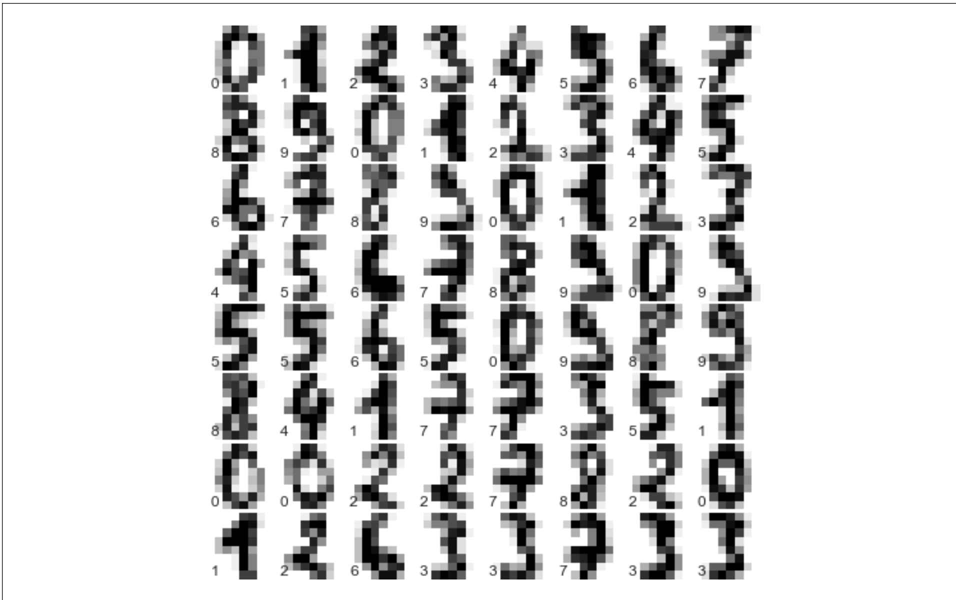


Figure 5-78. Representation of the digits data

We can quickly classify the digits using a random forest as follows (Figure 5-79):

```
In[14]:

from sklearn.cross_validation import train_test_split


Xtrain, Xtest, ytrain, ytest = train_test_split(digits.data, digits.target,

                                                random_state=0)
model = RandomForestClassifier(n_estimators=1000)
model.fit(Xtrain, ytrain)

ypred = model.predict(Xtest)
```

We can take a look at the classification report for this classifier:

```
In[15]: from sklearn import metrics

        print(metrics.classification_report(ypred, ytest))
```

```
            precisio  recall  f1-     support
            n                 score
0           1.00      0.97    0.99    38
1           1.00      0.98    0.99    44
2           0.95      1.00    0.98    42
3           0.98      0.96    0.97    46
4           0.97      1.00    0.99    37
5           0.98      0.96    0.97    49
6           1.00      1.00    1.00    52
7           1.00      0.96    0.98    50
```

```
8          0.94       0.98     0.96     46
9          0.96       0.98     0.97     46

avg /      0.98       0.98     0.98     450
total
```

And for good measure, plot the confusion matrix (Figure 5-79):

```
In[16]: from sklearn.metrics import confusion_matrix
        mat = confusion_matrix(ytest, ypred)

        sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False)
        plt.xlabel('true label')

        plt.ylabel('predicted label');
```

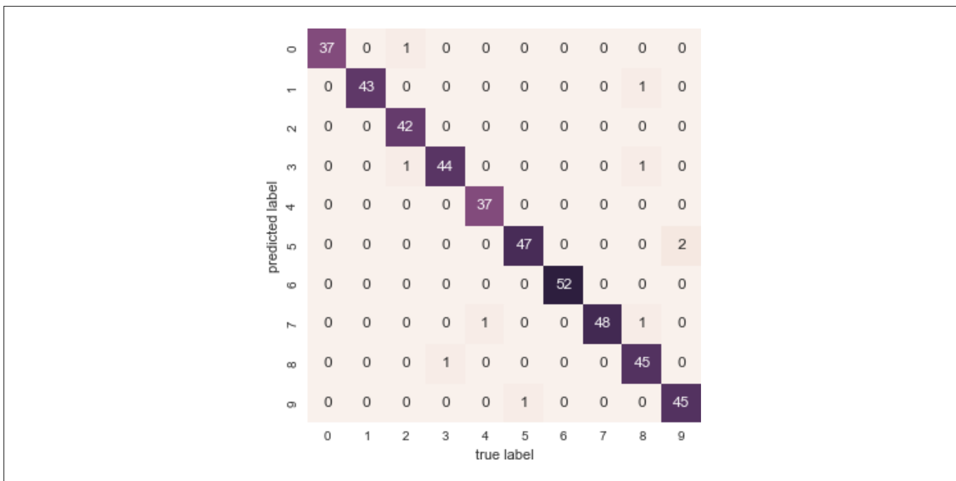


Figure 5-79. Confusion matrix for digit ***classification*** *with random forests*

We find that a simple, untuned random forest results in a very accurate classification of the digits data.

Summary of Random Forests

This section contained a brief introduction to the concept of *ensemble estimators*, andin particular the random forest model—an ensemble of randomized decision trees. Random forests are a powerful method with several advantages:

- Both training and prediction are very fast, because of the simplicity of the underlying decision trees. In addition, both tasks can be straightforwardly parallelized, because the individual trees are entirely independent entities.
- The multiple trees allow for a probabilistic classification: a majority vote among estimators gives an estimate of the probability (accessed in Scikit-Learn with the `predict_proba()` method).

- The nonparametric model is extremely flexible, and can thus perform well on tasks that are underfit by other estimators.

A primary disadvantage of random forests is that the results are not easily interpreta- ble; that is, if you would like to draw conclusions about the *meaning* of the classifica-tion model, random forests may not be the best choice.

In Depth: Principal Component Analysis

Up until now, we have been looking in depth at supervised learning estimators: those estimators that predict labels based on labeled training data. Here we begin looking atseveral unsupervised estimators, which can highlight interesting aspects of the data without reference to any known labels.

In this section, we explore what is perhaps one of the most broadly used of unsuper- vised algorithms, principal component analysis (PCA). PCA is fundamentally a dimensionality reduction algorithm, but it can also be useful as a tool for visualiza- tion, for noise filtering, for feature extraction and engineering, and much more. After a brief conceptual discussion of the PCA algorithm, we will see a couple examples ofthese further applications. We begin with the standard imports:

```
In[1]: %matplotlib inline

       import numpy as np

       import matplotlib.pyplot as plt
       import seaborn as sns; sns.set()
```

Introducing Principal Component Analysis

Principal component analysis is a fast and flexible unsupervised method for dimen- sionality reduction in data, which we saw briefly in "Introducing Scikit-Learn" on page 343. Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points (Figure 5-80):

```
In[2]: rng = np.random.RandomState(1)

       X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T

       plt.scatter(X[:, 0], X[:, 1])
       plt.axis('equal');
```

By eye, it is clear that there is a nearly linear relationship between the x and y vari- ables. This is reminiscent of the linear regression data we explored in "In Depth: Lin-ear Regression" on page 390, but the problem setting here is slightly different: rather than attempting to *predict* the *y* values from the *x* values, the unsupervised learning problem attempts to learn about the *relationship* between the *x* and *y* values.

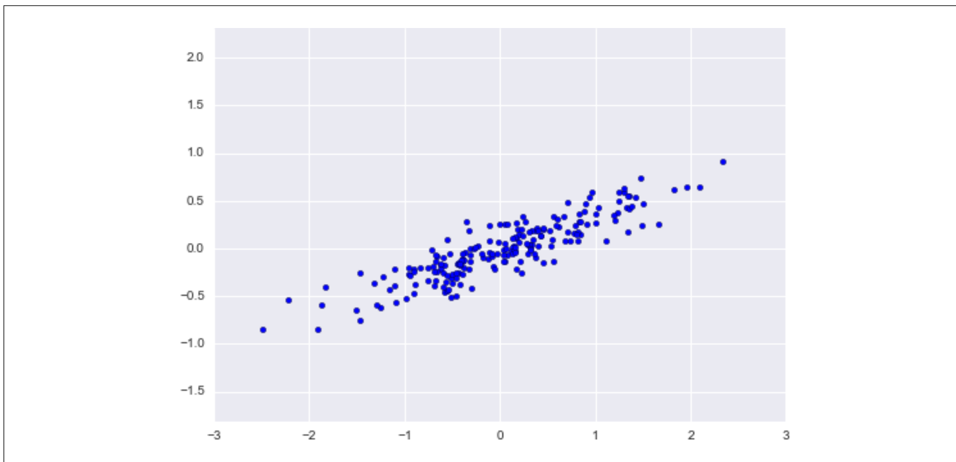


Figure 5-80. Data for demonstration of PCA

In principal component analysis, one quantifies this relationship by finding a list of the *principal axes* in the data, and using those axes to describe the dataset. Using Scikit-Learn's PCA estimator, we can compute this as follows:

```
In[3]: from sklearn.decomposition import PCA
       pca = PCA(n_components=2)

       pca.fit(X)

Out[3]: PCA(copy=True, n_components=2, whiten=False)
```

The fit learns some quantities from the data, most importantly the "components" and "explained variance":

```
In[4]: print(pca.components_)

[[ 0.94446029 0.32862557]

 [ 0.32862557 -0.94446029]]

In[5]: print(pca.explained_variance_)

[ 0.75871884 0.01838551]
```

To see what these numbers mean, let's visualize them as vectors over the input data, using the "components" to define the direction of the vector, and the "explained var- iance" to define the squared-length of the vector (Figure 5-81):

```
In[6]: def draw_vector(v0, v1, ax=None):
           ax = ax or plt.gca()

           arrowprops=dict(arrowstyle='->',

                           linewidth=2,
                           shrinkA=0, shrinkB=0)
```

```
    ax.annotate('', v1, v0, arrowprops=arrowprops)

# plot data

plt.scatter(X[:, 0], X[:, 1], alpha=0.2)

for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)

    draw_vector(pca.mean_, pca.mean_ + v)
plt.axis('equal');
```

Figure 5-81. Visualization of the principal axes in the data

These vectors represent the *principal axes* of the data, and the length shown inFigure 5-81 is an indication of how "important" that axis is in describing the distribu-tion of the data—more precisely, it is a measure of the variance of the data when pro-jected onto that axis. The projection of each data point onto the principal axes are the "principal components" of the data.

If we plot these principal components beside the original data, we see the plots shownin Figure 5-82.

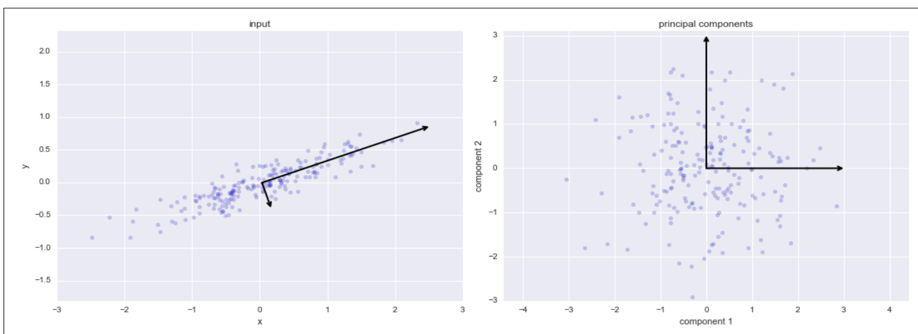


Figure 5-82. Transformed principal axes in the data

This transformation from data axes to principal axes is as an *affine transformation*, which basically means it is composed of a translation, rotation, and uniform scaling.

While this algorithm to find principal components may seem like just a mathematicalcuriosity, it turns out to have very far-reaching applications in the world of machine learning and data exploration.

PCA as dimensionality reduction

Using PCA for dimensionality reduction involves zeroing out one or more of the smallest principal components, resulting in a lower-dimensional projection of thedata that preserves the maximal data variance.

Here is an example of using PCA as a dimensionality reduction transform:

```
In[7]: pca = PCA(n_components=1)
       pca.fit(X)

       X_pca = pca.transform(X)

       print("original  shape:    ", X.shape)

       print("transformed shape:", X_pca.shape)

original shape:    (200, 2)

transformed shape: (200, 1)
```

The transformed data has been reduced to a single dimension. To understand the effect of this dimensionality reduction, we can perform the inverse transform of this reduced data and plot it along with the original data (Figure 5-83):

```
In[8]: X_new = pca.inverse_transform(X_pca)
       plt.scatter(X[:, 0], X[:, 1], alpha=0.2)

       plt.scatter(X_new[:, 0], X_new[:, 1], alpha=0.8)
       plt.axis('equal');
```

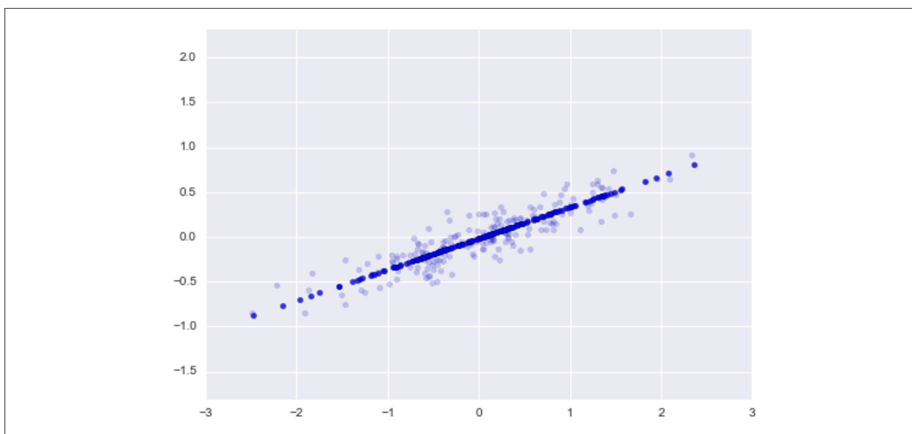


Figure 5-83. Visualization of PCA as dimensionality reduction

The light points are the original data, while the dark points are the projected version. This makes clear what a PCA dimensionality reduction means: the information along the least important principal axis or axes is removed, leaving only the component(s) of the data with the highest variance. The fraction of variance that is cut out (propor- tional to the spread of points about the line formed in Figure 5-83) is roughly a meas-ure of how much "information" is discarded in this reduction of dimensionality.

This reduced-dimension dataset is in some senses "good enough" to encode the most important relationships between the points: despite reducing the dimension of thedata by 50%, the overall relationship between the data points is mostly preserved.

PCA for visualization: Handwritten digits

The usefulness of the dimensionality reduction may not be entirely apparent in only two dimensions, but becomes much more clear when we look at high-dimensional data. To see this, let's take a quick look at the application of PCA to the digits data wesaw in "In-Depth: Decision Trees and Random Forests" on page 421.

We start by loading the data:

```
In[9]: from sklearn.datasets import load_digits
       digits = load_digits()

       digits.data.shape

Out[9]:

(1797, 64)
```

Recall that the data consists of 8×8 pixel images, meaning that they are 64-dimensional. To gain some intuition into the relationships between these points, we can use PCA to project them to a more manageable number of dimensions, say two:

```
In[10]: pca = PCA(2)  # project from 64 to 2 dimensions
        projected = pca.fit_transform(digits.data)
        print(digits.data.shape)

        print(projected.shape)

(1797, 64)

(1797, 2)
```

We can now plot the first two principal components of each point to learn about the data (Figure 5-84):

```
In[11]: plt.scatter(projected[:, 0], projected[:, 1],

                    c=digits.target, edgecolor='none', alpha=0.5,
                    cmap=plt.cm.get_cmap('spectral', 10))

        plt.xlabel('component 1') plt.ylabel('component 2')plt.colorbar();
```

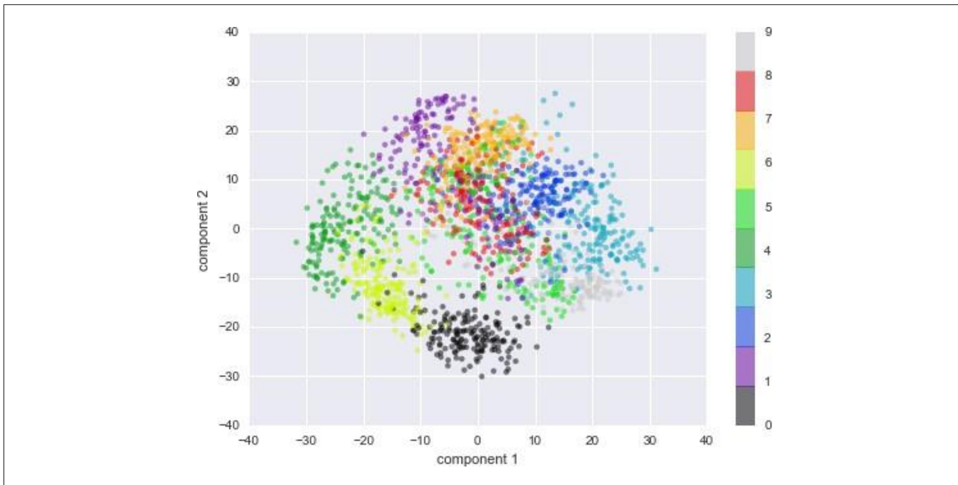


Figure 5-84. PCA applied to the handwritten digits data

Recall what these components mean: the full data is a 64-dimensional point cloud, and these points are the projection of each data point along the directions with the largest variance. Essentially, we have found the optimal stretch and rotation in 64- dimensional space that allows us to see the layout of the digits in two dimensions, andhave done this in an unsupervised manner—that is, without reference to the labels.

What do the components mean?

We can go a bit further here, and begin to ask what the reduced dimensions *mean*. This meaning can be understood in terms of combinations of basis vectors. For example, each image in the training set is defined by a collection of 64 pixel values, which we will call the vector x:

$$x = [x_1, x_2, x_3 \cdots x_{64}]$$

One way we can think about this is in terms of a pixel basis. That is, to construct the image, we multiply each element of the vector by the pixel it describes, and then add the results together to build the image:

$$\text{image}(x) = x_1 \cdot (\text{pixel 1}) + x_2 \cdot (\text{pixel 2}) + x_3 \cdot (\text{pixel 3}) \cdots x_{64} \cdot (\text{pixel 64})$$

One way we might imagine reducing the dimension of this data is to zero out all but afew of these basis vectors. For example, if we use only the first eight pixels, we get aneight-dimensional projection of the data (Figure 5-85), but it is not very reflective of the whole image: we've thrown out nearly 90% of the pixels!

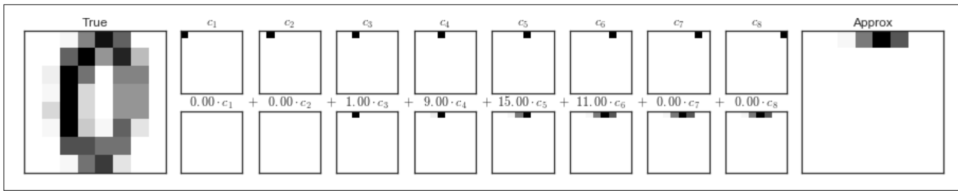


Figure 5-85. A naive dimensionality reduction achieved by discarding pixels

The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64- pixel image. Were we to continue this sequence and use all 64 pixels, we would recover the original image.

But the pixel-wise representation is not the only choice of basis. We can also use otherbasis functions, which each contain some predefined contribution from each pixel, and write something like:

$$image(x) = \text{mean} + x_1 \cdot (\text{basis } 1) + x_2 \cdot (\text{basis } 2) + x_3 \cdot (\text{basis } 3) \cdots$$

PCA can be thought of as a process of choosing optimal basis functions, such that adding together just the first few of them is enough to suitably reconstruct the bulk ofthe elements in the dataset. The principal components, which act as the low- dimensional representation of our data, are simply the coefficients that multiply each of the elements in this series. Figure 5-86 is a similar depiction of reconstructing this digit using the mean plus the first eight PCA basis functions.

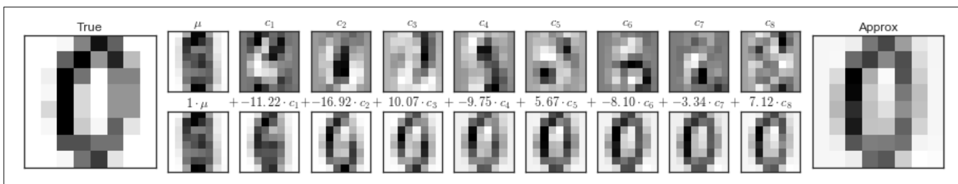


Figure 5-86. A more sophisticated dimensionality reduction achieved by discarding the least important principal components (compare to Figure 5-85)

Unlike the pixel basis, the PCA basis allows us to recover the salient features of the input image with just a mean plus eight components! The amount of each pixel in each component is the corollary of the orientation of the vector in our two- dimensional example. This is the sense in which PCA provides a low-dimensional representation of the data: it discovers a set of basis functions that are more efficient than the native pixel-basis of the input data.

Choosing the number of components

A vital part of using PCA in practice is the ability to estimate how many components are needed to describe the data. We can determine this by looking at the cumulative *explained variance ratio* as a function of the number of components (Figure 5-87):

```
In[12]: pca = PCA().fit(digits.data)
        plt.plot(np.cumsum(pca.explained_variance_ratio_))
        plt.xlabel('number of components')
        plt.ylabel('cumulative explained variance');
```

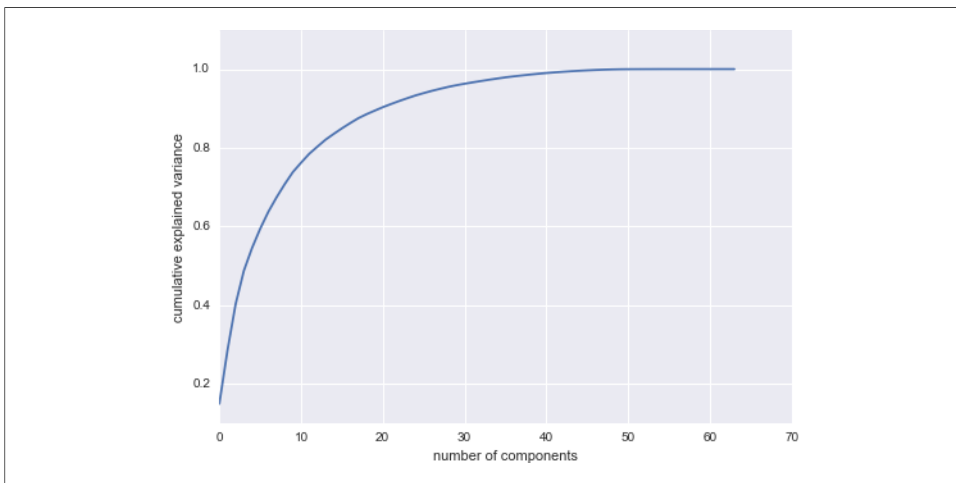


*Figure 5-87. **The** cumulative explained variance, which measures how well PCA pre-serves the content of the data*

This curve quantifies how much of the total, 64-dimensional variance is contained within the first *N* components. For example, we see that with the digits the first 10 components contain approximately 75% of the variance, while you need around 50 components to describe close to 100% of the variance.

Here we see that our two-dimensional projection loses a lot of information (as meas- ured by the explained variance) and that we'd need about 20 components to retain 90% of the variance. Looking at this plot for a high-dimensional dataset can help youunderstand the level of redundancy present in multiple observations.

PCA as Noise Filtering

PCA can also be used as a filtering approach for noisy data. The idea is this: any com-ponents with variance much larger than the effect of the noise should be relatively unaffected by the noise. So if you reconstruct the data using just the largest subset of principal components, you should be preferentially keeping the signal and throwing out the noise.

Let's see how this looks with the digits data. First we will plot several of the inputnoise-free data (Figure 5-88):

```
In[13]: def plot_digits(data):

            fig, axes = plt.subplots(4, 10, figsize=(10, 4),

                                     subplot_kw={'xticks':[], 'yticks':[]},
                                     gridspec_kw=dict(hspace=0.1, wspace=0.1))

            for i, ax in enumerate(axes.flat):
                ax.imshow(data[i].reshape(8, 8),

                          cmap='binary', interpolation='nearest',
                          clim=(0, 16))

        plot_digits(digits.data)
```

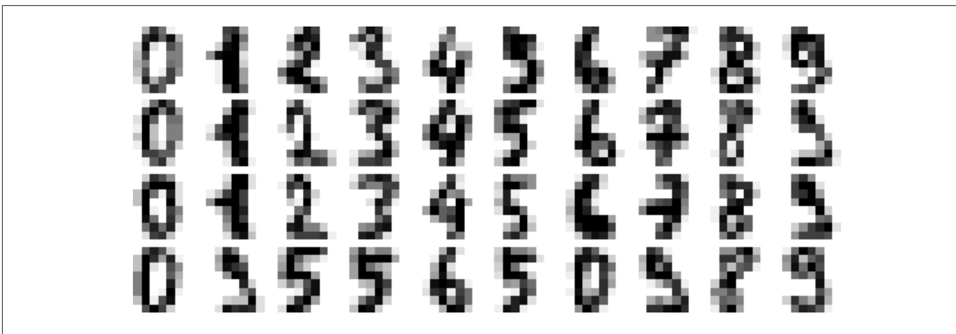


Figure 5-88. Digits without noise

Now let's add some random noise to create a noisy dataset, and replot it(Figure 5-89):

```
In[14]: np.random.seed(42)

        noisy = np.random.normal(digits.data, 4)
        plot_digits(noisy)
```

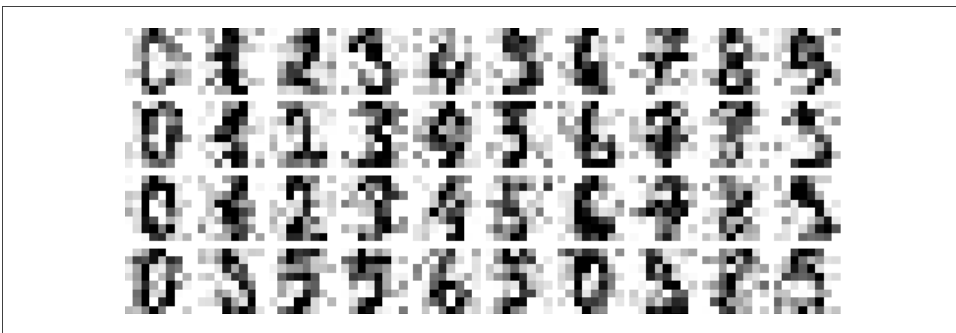


Figure 5-89. Digits with Gaussian random noise added

It's clear by eye that the images are noisy, and contain spurious pixels. Let's train aPCA on the noisy data, requesting that the projection preserve 50% of the variance:

```
In[15]: pca = PCA(0.50).fit(noisy)
        pca.n_components_
```

```
Out[15]: 12
```

Here 50% of the variance amounts to 12 principal components. Now we compute these components, and then use the inverse of the transform to reconstruct the fil- tered digits (Figure 5-90):

```
In[16]: components = pca.transform(noisy)

        filtered = pca.inverse_transform(components)
        plot_digits(filtered)
```

Figure 5-90. Digits "denoised" using PCA

This signal preserving/noise filtering property makes PCA a very useful feature selec-tion routine—for example, rather than training a classifier on very high-dimensional data, you might instead train the classifier on the lower-dimensional representation, which will automatically serve to filter out random noise in the inputs.

Example: Eigenfaces

Earlier we explored an example of using a PCA projection as a feature selector for facial recognition with a support vector machine ("In-Depth: Support Vector Machines" on page 405). Here we will take a look back and explore a bit more of whatwent into that. Recall that we were using the Labeled Faces in the Wild dataset made available through Scikit-Learn:

```
In[17]: from sklearn.datasets import fetch_lfw_people
        faces = fetch_lfw_people(min_faces_per_person=60)
        print(faces.target_names)
        print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Let's take a look at the principal axes that span this dataset. Because this is a large dataset, we will use RandomizedPCA—it contains a randomized method to approxi-

mate the first *N* principal components much more quickly than the standard PCA esti-mator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3,000). We will take a look at the first 150 components:

```
In[18]: from sklearn.decomposition import RandomizedPCA
        pca = RandomizedPCA(150)

        pca.fit(faces.data)

Out[18]: RandomizedPCA(copy=True, iterated_power=3, n_components=150,
                random_state=None, whiten=False)
```

In this case, it can be interesting to visualize the images associated with the first sev- eral principal components (these components are technically known as "eigenvec- tors," so these types of images are often called "eigenfaces"). As you can see in Figure 5-91, they are as creepy as they sound:

```
In[19]: fig, axes = plt.subplots(3, 8, figsize=(9, 4),

                                 subplot_kw={'xticks':[], 'yticks':[]},
                                 gridspec_kw=dict(hspace=0.1, wspace=0.1))

        for i, ax in enumerate(axes.flat):
            ax.imshow(pca.components_[i].reshape(62, 47), cmap='bone')
```

Figure 5-91. A visualization of eigenfaces learned from the LFW dataset

The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking out certainfeatures, such as eyes, noses, and lips. Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserv- ing (Figure 5-92):

```
In[20]: plt.plot(np.cumsum(pca.explained_variance_ratio_))
        plt.xlabel('number of components')
        plt.ylabel('cumulative explained variance');
```

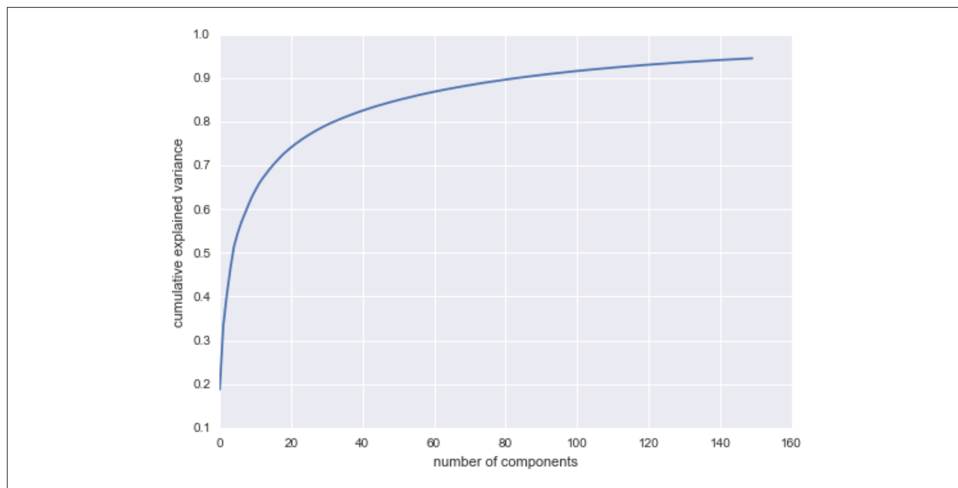


Figure 5-92. Cumulative explained variance for the LFW data

We see that these 150 components account for just over 90% of the variance. That would lead us to believe that using these 150 components, we would recover most of the essential characteristics of the data. To make this more concrete, we can compare the input images with the images reconstructed from these 150 components (Figure 5-93):

```
In[21]: # Compute the components and projected faces
        pca = RandomizedPCA(150).fit(faces.data)
        components = pca.transform(faces.data)
        projected = pca.inverse_transform(components)

In[22]: # Plot the results

        fig, ax = plt.subplots(2, 10, figsize=(10, 2.5),

                               subplot_kw={'xticks':[], 'yticks':[]},
                               gridspec_kw=dict(hspace=0.1, wspace=0.1))

        for i in range(10):

            ax[0, i].imshow(faces.data[i].reshape(62, 47), cmap='binary_r')
            ax[1, i].imshow(projected[i].reshape(62, 47), cmap='binary_r')


        ax[0, 0].set_ylabel('full-dim\ninput')

        ax[1, 0].set_ylabel('150-dim\nreconstruction');
```

Figure 5-93. 150-dimensional PCA reconstruction of the LFW data

The top row here shows the input images, while the bottom row shows the recon- struction of the images from just 150 of the ~3,000 initial features. This visualization makes clear why the PCA feature selection used in "In-Depth: Support Vector Machines" on page 405 was so successful: although it reduces the dimensionality of the data by nearly a factor of 20, the projected images contain enough information that we might, by eye, recognize the individuals in the image. What this means is thatour classification algorithm needs to be trained on 150-dimensional data rather than 3,000-dimensional data, which depending on the particular algorithm we choose, can lead to a much more efficient classification.

Week 7: Creating Reports and Dashboards

Day-01 : Introduction to Dashboards

A dashboard for data analytics is a tool used to multi-task, organize, visualize, analyze, and track data. The overall purpose of a data analytics dashboard is to make it easier for data analysts, decision makers, and average users to understand their data, gain deeper insights, and make better data-driven decisions.

Data dashboards are designed to connect and help extract important information from a wide variety of different data sources, services, and APIs. This information is displayed in a single, unified view via visuals such as charts, figures, graphs, and tables. An organization can have a different customizable dashboard for each department and even a dashboard for each individual project, which helps provide granular monitoring of very specific KPIs.

“Smart” data analytics dashboard software uses AI and Machine Learning to save time and automate processes like data collection, discovery, preparation, replication, and reporting, which is crucial for big data sets where manual processing is impractical. Advanced interactive dashboards will provide compelling storytelling through attractive designs and real-time, interactive dynamic data visualizations that empower team members to quickly and easily reveal hidden insights and draw valuable conclusions that can help answer business questions and informa business decisions.

Data Analytics Dashboard Benefits

There are many different benefits to be gained from the many different kinds of data analytics dashboards. Some of the most common benefits include: data visibility and accessibility, measuring performance, business forecasting abilities, and agile responses:

Visibility and Accessibility: One of the primary benefits of a dashboard is its ability to display all of the most relevant, important data in a way that is intuitive, digestible, and useful for the average user. Dashboards should be a place where users can easily access key metrics and insights in a unified space so that anyone in the organization can derive value from it.

Measuring Performance: Dashboards will help measure and keep track of the performance of different teams, departments, products, and services. When analyzing the performance of an organization as a whole, it is crucial to set KPIs and have access to specific performance data in order to be able to hone in on processes that are creating inefficiencies and develop new strategies.

Agility: Dashboards help users detect changes in data quickly, in turn empowering users to react quickly. Real-time updates enable users to immediately correct course in the moment, or even get a jump on forthcoming trends.

Forecasting: AI and machine learning algorithms take historical data and current, real-time data in order to identify trends and anomalies, and forecast potential issues before they become problems. Forecasting can help direct things like demand planning, financial operations, future production, risk reduction, and digital marketing operations.

What are some Data Analytics Dashboard Examples?

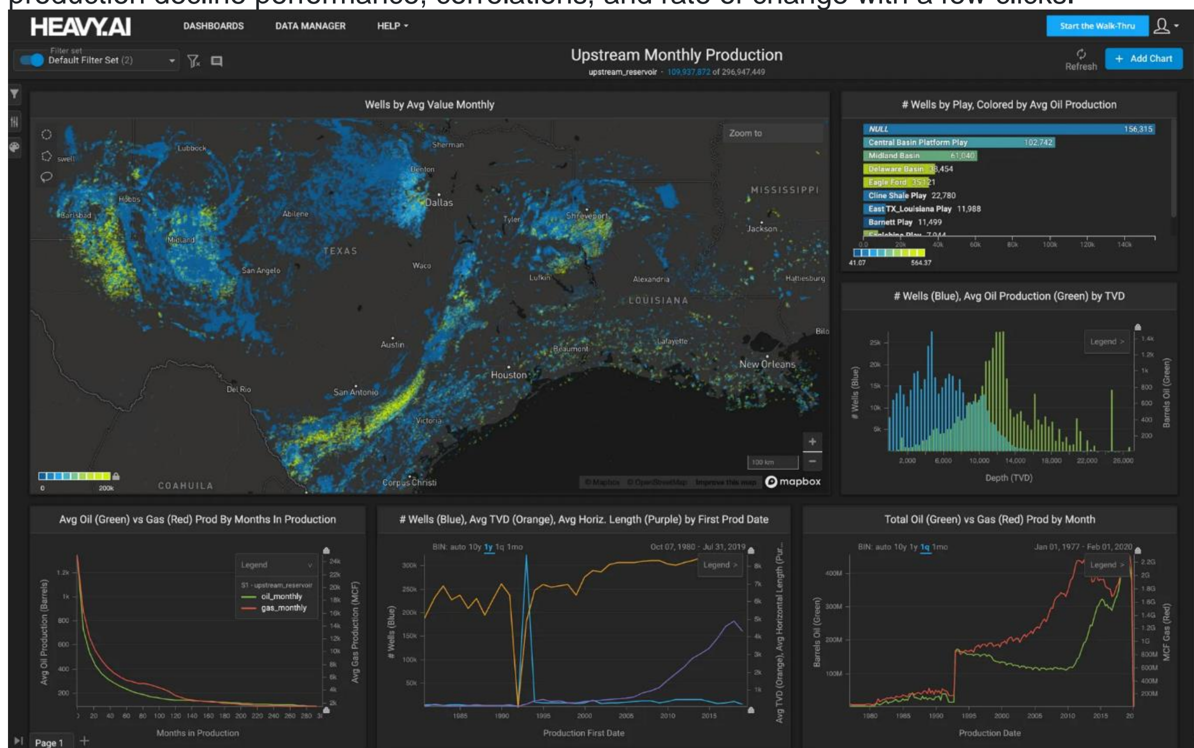
Big data analytics is leveraged in nearly every modern industry. Some big data analytics examples include retail, manufacturing, oil and gas, government, healthy industries, education, sports, sciences, airlines, banking, business analytics dashboards, and marketing analytics dashboards. All of these industries can benefit enormously from data analytics dashboards tailored to their specific needs. Read on to see some data analytics dashboard examples and data analytics demos.

Example1: Oil and Gas Data Analytics Dashboard

Companies throughout the oil and gas industry can derive enormous value from big data analytics dashboards. Industry professionals can interact with spatiotemporal data analytics in energy to determine things like productivity drivers, assess suitable land, and understand benchmark performance.

Oil industry professionals can visually analyze data and conclude why wells are over or underperforming, forecast their estimated potential, compare daily drill and well performance, manage fleets, and identify production trends across basins. Dashboards for data analytics can help renewable energy industry professionals visualize and interact with massive multi-sourced datasets to determine where their customers should make wind, solar, biomass, hydroelectric, or geothermal energy investments.

In this oil and gas demo, visualizations for 250 million well production records across the entire United States are available for analysis. Research scenarios and quickly analyze production decline performance, correlations, and rate of change with a few clicks.



*Example2: **Covid-19 Pandemic Data Analytics Dashboard***

Covid-19 data maps updated with real-time information were crucial for tracking the spread of the pandemic, recovery rates, and monitoring the effectiveness of quarantine orders and mask mandates. Covid-19 data analytics dashboards provided a simple, unified view of cases around the world filtered across location and time, informing decisions made by hospital administrators and lawmakers, such as office, school and business closure orders; mask mandated spaces; travel bans; PPE inventory forecasts; and more.

Government data analytics dashboards compile data from a wide variety of sources, like hospitals, government agencies, the CDC, World Health Organization, and make it easier for users to quickly identify patterns and draw conclusions.

In this Covid-19 demo, visualize the spread of the virus using maps and charts, compare the growth of cases across various countries and US states, and analyze the recovery rate in various regions of the world.

What are the Best Analytics Dashboard Tools?

The quality, variety, and volume of data analytics dashboard tools has increased in recent years. The best option for your organization depends on a number of factors, such as budget, deployment, client, and the specific goals and objectives of the project at hand. There are three main types of dashboard software: operational, strategic, tactical, and analytical. Analytical dashboard software functionality is prevalent in many business intelligence tools as they provide the greatest value to data analysts and data scientists. Learn how to make the most out of your business intelligence dashboard here.

The best data analytics dashboard tools will offer: the ability to connect your data from multiple sources, embedding capabilities, self-service reporting, automated real-time updates, streaming and predictive analytics driven by AI, filtering across time and location, interactive visual analytics, full customization, and at-a-click exploration. Some examples of popular enterprise analytics dashboard software include: HEAVY's visual analytics platform, Izenda, Periscope Data, Dundas BI analytics dashboard, Microsoft Power BI, IBM Cognos, TIBCO Spotfire, Looker, and Sisense.

Every data analytics dashboard will look different depending on each different project's goals and objectives. The best option will be one that empowers you to be at one with your data and to interact with it instantly and effortlessly. See the key capabilities that OmniSci's converged analytics platform provides to help users achieve insights from your largest datasets at the speed of curiosity.

Building interactive dashboards with libraries like Dash or Streamlit

What's a real-time live dashboard?

A real-time live dashboard is a web app used to display Key Performance Indicators (KPIs).

If you want to build a dashboard to monitor the stock market, IoT Sensor Data, AI Model Training, or anything else with streaming data, then this tutorial is for you.

1. How to import the required libraries and read input data

Here are the libraries that you'll need for this dashboard:

Streamlit (st). As you might've guessed, you'll be using Streamlit for building the web app/dashboard.

Time, NumPy (np). Because you don't have a data source, you'll need to simulate a live data feed. Use NumPy to generate data and make it live (looped) with the Time library (unless you already have a live data feed).

Pandas (pd). You'll use pandas to read the input data source. In this case, you'll use a Comma Separated Values (CSV) file.

Go ahead and import all the required libraries:

```
import time  # to simulate a real time data, time loop
import numpy as np  # np mean, np random
import pandas as pd  # read csv, df manipulation
import plotly.express as px  # interactive charts
import streamlit as st  # data web app development
```

You can read your input data in a CSV by using pd.read_csv(). But remember, this data source could be streaming from an API, a JSON or an XML object, or even a CSV that gets updated at regular intervals.

Next, add the pd.read_csv() call within a new function get_data() so that it gets properly cached.

What's caching? It's simple. Adding the decorator @st.experimental_memo will make the function get_data() run once. Then every time you rerun your app, the data will stay memoized! This way you can avoid downloading the dataset again and again. Read more about caching in Streamlit docs.

```
dataset_url = "https://raw.githubusercontent.com/Lexie88rus/bank-marketing-analysis/master/bank.csv"
# read csv from a URL
```

```
@st.experimental_memo
def get_data() -> pd.DataFrame:
    return pd.read_csv(dataset_url)
df = get_data()
table-1
```

2. How to do a basic dashboard setup

Now let's set up a basic dashboard. Use st.set_page_config() with parameters serving the following purpose:

The web app title page_title in the HTML tag <title> and in the browser tab

The favicon that uses the argument page_icon (also in the browser tab)

The layout = "wide" that renders the web app/dashboard with a wide-screen layout

```
st.set_page_config(
    page_title="Real-Time Data Science Dashboard",
    page_icon="✅",
    layout="wide",
)
```

3. How to design a user interface

A typical dashboard contains the following basic UI design components:

- A page title
- A top-level filter
- KPIs/summary cards
- Interactive charts
- A data table

Let's drill into them in detail.

Page title

The title is rendered as the <h1> tag. To display the title, use st.title(). It'll take the string "Real-Time / Live Data Science Dashboard" and display it in the Page Title.

```
# dashboard title
st.title("Real-Time / Live Data Science Dashboard")
```

Top-level filter

First, create the filter by using st.selectbox(). It'll display a dropdown with a list of options. To generate it, take the unique elements of the job column from the dataframe df. The selected item is saved in an object named job_filter:

```
# top-level filters
job_filter = st.selectbox("Select the Job", pd.unique(df["job"]))
```

Now that your filter UI is ready, use job_filter to filter your dataframe df.

```
# dataframe filter
df = df[df["job"] == job_filter]
```

KPIs/summary cards

Before you can design your KPIs, divide your layout into a 3 column layout by using st.columns(3). The three columns are kpi1, kpi2, and kpi3. st.metric() helps you create a KPI card. Use it to fill one KPI in each of those columns.

st.metric()'s label helps you display the KPI title. The value **is the argument that helps you show the actual metric (value) and add-ons like delta to compare the KPI value with the KPI goal.

```
# create three columns
kpi1, kpi2, kpi3 = st.columns(3)
# fill in those three columns with respective metrics or KPIs
kpi1.metric(
  label="Age ⏳",
  value=round(avg_age),
  delta=round(avg_age) - 10,
)

kpi2.metric(
  label="Married Count 💍",
  value=int(count_married),
  delta=-10 + count_married,
)

kpi3.metric(
  label="A/C Balance ＄",
  value=f"$ {round(balance,2)} ",
  delta=-round(balance / count_married) * 100,
)
```

Interactive charts

Split your layout into 2 columns and fill them with charts. Unlike the metric above, use the with clause to fill the interactive charts in the respective columns:

Density_heatmap in fig_col1

Histogram in fig_col2

```
# create two columns for charts
fig_col1, fig_col2 = st.columns(2)

with fig_col1:
  st.markdown("### First Chart")
  fig = px.density_heatmap(
    data_frame=df, y="age_new", x="marital"
  )
  st.write(fig)

with fig_col2:
  st.markdown("### Second Chart")
  fig2 = px.histogram(data_frame=df, x="age_new")
  st.write(fig2)
```

Data table

Use st.dataframe() to display the data frame. Remember, your data frame gets filtered based on the filter option selected at the top:

```
st.markdown("### Detailed Data View")
st.dataframe(df)
```

4. How to refresh the dashboard for real-time or live data feed

Since you don't have a real-time or live data feed yet, you're going to simulate your existing data frame (unless you already have a live data feed or real-time data flowing in).

To simulate it, use a for loop from 0 to 200 seconds (as an option, on every iteration you'll have a second sleep/pause):

```
for seconds in range(200):
    df["age_new"] = df["age"] * np.random.choice(range(1, 5))
    df["balance_new"] = df["balance"] * np.random.choice(range(1, 5))
    time.sleep(1)
```

Inside the loop, use NumPy's random.choice to generate a random number between 1 to 5. Use it as a multiplier to randomize the values of age and balance columns that you've used for your metrics and charts.

5. How to auto-update components

Now you know how to do a Streamlit web app!

To display the live data feed with auto-updating KPIs/Metrics/Charts, put all these components inside a single-element container using st.empty(). Call it placeholder:

```
# creating a single-element container.
placeholder = st.empty()
```

Put your components inside the placeholder by using a with clause. This way you'll replace them in every iteration of the data update. The code below contains the placeholder.container() along with the UI components you created above:

```
with placeholder.container():

    # create three columns
    kpi1, kpi2, kpi3 = st.columns(3)

    # fill in those three columns with respective metrics or KPIs
    kpi1.metric(
        label="Age ⏳",
        value=round(avg_age),
        delta=round(avg_age) - 10,
    )

    kpi2.metric(
        label="Married Count 💍",
        value=int(count_married),
        delta=-10 + count_married,
    )

    kpi3.metric(
        label="A/C Balance ＄",
        value=f"$ {round(balance,2)} ",
```

```
        delta=-round(balance / count_married) * 100,
    )

    # create two columns for charts
    fig_col1, fig_col2 = st.columns(2)

    with fig_col1:
        st.markdown("### First Chart")
        fig = px.density_heatmap(
            data_frame=df, y="age_new", x="marital"
        )
        st.write(fig)

    with fig_col2:
        st.markdown("### Second Chart")
        fig2 = px.histogram(data_frame=df, x="age_new")
        st.write(fig2)

    st.markdown("### Detailed Data View")
    st.dataframe(df)
    time.sleep(1)
```

And...here is the full code!

```
import time  # to simulate a real time data, time loop
import numpy as np  # np mean, np random
import pandas as pd  # read csv, df manipulation
import plotly.express as px  # interactive charts
import streamlit as st  # 🎈 data web app development

st.set_page_config(
    page_title="Real-Time Data Science Dashboard",
    page_icon="✅",
    layout="wide",
)

# read csv from a github repo
dataset_url = "https://raw.githubusercontent.com/Lexie88rus/bank-marketing-
analysis/master/bank.csv"

# read csv from a URL
@st.experimental_memo
def get_data() -> pd.DataFrame:
    return pd.read_csv(dataset_url)

df = get_data()

# dashboard title
st.title("Real-Time / Live Data Science Dashboard")

# top-level filters
job_filter = st.selectbox("Select the Job", pd.unique(df["job"]))

# creating a single-element container
placeholder = st.empty()

# dataframe filter
df = df[df["job"] == job_filter]
```

```
# near real-time / live feed simulation
for seconds in range(200):

    df["age_new"] = df["age"] * np.random.choice(range(1, 5))
    df["balance_new"] = df["balance"] * np.random.choice(range(1, 5))

    # creating KPIs
    avg_age = np.mean(df["age_new"])

    count_married = int(
        df[(df["marital"] == "married")]["marital"].count()
        + np.random.choice(range(1, 30))
    )

    balance = np.mean(df["balance_new"])

    with placeholder.container():

        # create three columns
        kpi1, kpi2, kpi3 = st.columns(3)

        # fill in those three columns with respective metrics or KPIs
        kpi1.metric(
            label="Age ⏳",
            value=round(avg_age),
            delta=round(avg_age) - 10,
        )

        kpi2.metric(
            label="Married Count 💍",
            value=int(count_married),
            delta=-10 + count_married,
        )

        kpi3.metric(
            label="A/C Balance ＄",
            value=f"$ {round(balance,2)} ",
            delta=-round(balance / count_married) * 100,
        )

        # create two columns for charts
        fig_col1, fig_col2 = st.columns(2)
        with fig_col1:
            st.markdown("### First Chart")
            fig = px.density_heatmap(
                data_frame=df, y="age_new", x="marital"
            )
            st.write(fig)
              with fig_col2:
            st.markdown("### Second Chart")
            fig2 = px.histogram(data_frame=df, x="age_new")
            st.write(fig2)

        st.markdown("### Detailed Data View")
        st.dataframe(df)
        time.sleep(1)
```

To run this dashboard on your local computer:

- Save the code as a single monolithic app.py.
- Open your Terminal or Command Prompt in the same path where the app.py is stored.
- Execute streamlit run app.py for the dashboard to start running on your localhost and the link would be displayed in your Terminal and also opened as a new Tab in your default browser.

Day-02: Develop Data Visualization Interfaces in Python With Dash

Dash gives data scientists the ability to showcase their results in interactive web applications. You don't need to be an expert in web development. In an afternoon, you can build and deploy a Dash app to share with others.

Here you'll learn how to:

- Create a **Dash application**
- Use Dash **core components** and **HTML components**
- **Customize the style** of your Dash application
- Use **callbacks** to build interactive applications
- Deploy your application on **PythonAnywhere**

You can download the source code, data, and resources for the sample application that you'll make in this tutorial by clicking the link below:

What Is Dash?

Dash is an open-source framework for building data visualization interfaces. Released in 2017 as a Python library, it's grown to include implementations for R, Julia, and F#. Dash helps data scientists build analytical web applications without requiring advanced web development knowledge.

Three technologies constitute the core of Dash:

1. **Flask** supplies the web server functionality.
2. **React.js** renders the user interface of the web page.
3. **Plotly.js** generates the charts used in your application.

But you don't have to worry about making all these technologies work together. Dash will do that for you. You just need to write Python, R, Julia, or F# and sprinkle in a bit of CSS.

Plotly, a Canada-based company, built Dash and supports its development. You may know the company from the popular graphing libraries that share its name. The company released Dash as open source under an MIT license, so you can use Dash at no cost.

Plotly also offers a commercial companion to Dash called Dash Enterprise. This paid service provides companies with support services such as hosting, deploying, and handling authentication on Dash applications. But these features live outside of Dash's open-source ecosystem.

Dash will help you build dashboards quickly. If you're used to analyzing data or building data visualizations using Python, then Dash will be a useful addition to your toolbox. Here are a few examples of what you can make with Dash:

1. A dashboard showing object detection for self-driving cars
2. A visualization of millions of Uber rides
3. An interactive tool for analyzing soccer match data

This is just a tiny sample. If you'd like to see other interesting use cases, then go check out the Dash App Gallery.

Note: You don't need advanced knowledge of web development to follow this manual, but some familiarity with HTML and CSS won't hurt.

You should know the basics of the following topics, though:

- Python graphing libraries such as Plotly, Bokeh, and Matplotlib
- HTML and the structure of an HTML file CSS and style sheets

Get Started With Dash in Python

You'll go through the end-to-end process of building a dashboard using Dash. If you follow along with the examples, then you'll go from a bare-bones dashboard on your local machine to a styled dashboard deployed on PythonAnywhere.

To build the dashboard, you'll use a dataset of sales and prices of avocados in the United States between 2015 and 2018. Justin Kiggins compiled this dataset using data from the Hass Avocado Board.

How to Set Up Your Local Environment

To develop your app, you'll need a new directory to store your code and data. You'll also need a clean Python virtual environment. To create those, execute the commands below, choosing the version that matches your operating system:

```
PS> mkdir avocado_analytics
PS> cd avocado_analytics
PS> python -m venv venv
PS> venv\Scripts\activate
```

The first two commands create a directory for your project and move your current location there. The next command creates a virtual environment in that location. The last command activates the virtual environment.

Next, you need to install the required libraries. You can do that using pip inside your virtual environment. Install the libraries as follows:

```
(venv) $ python -m pip install dash==2.8.1 pandas==1.5.3
```

This command will install Dash and pandas in your virtual environment. You'll use specific versions of these packages to make sure that you have the same environment as the one used throughout this tutorial. Alongside Dash, pandas will help you handle reading and wrangling the data that you'll use in your app.

Save the data as avocado.csv in the root directory of the project. By now, you should have a virtual environment with the required libraries and the data in the root folder of your project. Your project's structure should look like this:

```
avocado_analytics/
venv/
avocado.csv
```

Now you'll build your first Dash application.

How to Build a Dash Application

For development purposes, it's useful to think of the process of building a Dash application in three steps:

1. Define the **content** of your application using the app's layout.
2. Style the **looks** of your app with CSS or styled components.
3. Use callbacks to determine which parts of your app are **interactive** and what they react to.

Initializing Your Dash Application

Create an empty file named app.py in the root directory of your project, then review the code of app.py in this section. To make it easier for you to copy the full code, you'll find the entire contents of app.py at the end of this section.

Here are the first few lines of `app.py`:

```
1# app.py
2
3import pandas as pd
4from dash import Dash, dcc, html
5
6data = (
7    pd.read_csv("avocado.csv")
8    .query("type == 'conventional' and region == 'Albany'")
9    .assign(Date=lambda data: pd.to_datetime(data["Date"], format="%Y-%m-%d"))
10    .sort_values(by="Date")
11)
12
13app = Dash(__name__)
```

On lines 3 and 4, you import the required libraries: pandas and dash. You'll use pandas to read and organize the data. You're importing the following elements from dash:

- **`Dash`** helps you initialize your application.
- **`html`**, also called Dash HTML Components, lets you access HTML tags.
- **`dcc`**, short for Dash Core Components, allows you to create interactive components like graphs, dropdowns, or date ranges.

On lines 6 to 11, you read the data and preprocess it for use in the dashboard. You filter some of the data because your dashboard isn't interactive yet, and the plotted values wouldn't make sense otherwise.

On line 13, you create an instance of the `Dash` class you use `Dash(__name__)`.

Defining the Layout of Your Dash Application

Next, you'll define the layout property of your application. This property dictates the content of your app. In this case, you'll use a heading with a description immediately below it, followed by two graphs. Here's how you define it:

```
1# app.py
2
3# ...
4
5app.layout = html.Div(
6    children=[
7        html.H1(children="Avocado Analytics"),
8        html.P(
9            children=(
10                "Analyze the behavior of avocado prices and the number"
11                " of avocados sold in the US between 2015 and 2018"
12            ),
13        ),
14        dcc.Graph(
15            figure={
16                "data": [
17                    {
18                        "x": data["Date"],
19                        "y": data["AveragePrice"],
20                        "type": "lines",
21                    },
22                ],
23                "layout": {"title": "Average Price of Avocados"},
24            },
25        ),
26        dcc.Graph(
27            figure={
28                "data": [
29                    {
30                        "x": data["Date"],
```

```
31                          "y": data["Total Volume"],
32                          "type": "lines",
33                      },
34                  ],
35                  "layout": {"title": "Avocados Sold"},
36              },
37          ),
38      ]
39)
```

With this code, you define the .layout property of the app object. This property determines the content of your application using a tree structure made of Dash components.

Dash components come prepackaged in Python libraries. Some of them come with Dash when you install it. You have to install the rest separately. You'll see two sets of components in almost every app:

1. The Dash HTML Components module provides you with Python wrappers for HTML elements. For example, you could use Dash HTML Components to create elements such as paragraphs, headings, or lists.
2. The Dash Core Components module provides you with Python abstractions for creating interactive user interfaces. You can use these components to create interactive elements such as graphs, sliders, or dropdowns.

On lines 5 to 13, you can see the Dash HTML components in practice. You start by defining the parent component, html.Div. Then you add two more elements, a heading (html.H1) and a paragraph (html.P), as its children.

These components are equivalent to the <div>, <h1>, and <p> HTML tags. You can use the components' arguments to modify attributes or the content of the tags. For example, to specify what goes inside the <div> tag, you use the children argument in html.Div.

There are also other arguments in the components, such as style, className, and id, that refer to attributes of the HTML tags. You'll see how to use some of these properties to style your dashboard in the next section.

The part of the layout shown on lines 5 to 13 will get transformed into the following HTML code:

```
<div>
  <h1>Avocado Analytics</h1>
  <p>
    Analyze the behavior of avocado prices and the number
    of avocados sold in the US between 2015 and 2018
  </p>
  <!-- Rest of the app -->
```

```
</div>
```

This HTML code is rendered when you open your application in the browser. It follows the same structure as your Python code, with a <div> tag containing an <h1> and a <p> element.

On lines 14 and 26 in the layout code snippet, you can see the graph component from Dash Core Components in practice. There are two dcc.Graph components in app.layout. The first one plots the average prices of avocados during the period of study, and the second plots the number of avocados sold in the United States during the same period.

Under the hood, Dash uses Plotly.js to generate graphs. The dcc.Graph components expect a figure object or a Python dictionary containing the plot's data and layout. In this case, you provide the latter.

Finally, these two lines of code help you run your application:

```
# app.py

# ...

if __name__ == "__main__":
    app.run_server(debug=True)
```

These lines make it possible to run your Dash application locally using Flask's built-in server. The debug=True parameter enables the **hot-reloading** option in your application. This means that when you make a change to your app, it reloads automatically, without you having to restart the server.

This is the code for your bare-bones dashboard. It includes all the snippets of code that you reviewed earlier in this section.

Now it's time to run your application. Open a terminal inside your project's root directory with the project's virtual environment activated. Run python app.py, then go to http://localhost:8050 using your preferred browser.

Note: Install dash by typing followin command on Shell:

```
conda install dash dash-core-components dash-html-components dash-renderer -c conda-forge
```

and the py -m pip install dash

Your dashboard should look like this:

The dashboard is far from visually pleasing, and you still need to add some interactivity to it.

Style Your Dash Application

Dash provides you with a lot of flexibility to customize the look of your application. You can use your own CSS or JavaScript files, set a **favicon**—the small icon shown on tabs in the web browser—and embed images, among other advanced options.

Now you'll see how to show off your own style with CSS. There are several packages on PyPI that provide styled Dash components. For example, dash-bootstrap-components are Bootstrap themed.

Apply custom styles to components, and then you'll style the dashboard that you built in the previous section.

How to Apply a Custom Style to Your Components

You can style components in two ways:

- Using the style argument of individual components
- Providing an external CSS file

Using the style argument to customize your dashboard is straightforward. This argument takes a Python dictionary with key-value pairs consisting of the names of CSS properties and the values that you want to set.

When specifying CSS properties in the style argument, you should use mixedCase syntax instead of hyphen-separated words. For example, to change the background color of an element, you should use backgroundColor and not background-color.

If you wanted to change the size and color of the H1 element in app.py, then you could set the element's style argument as follows:

```
html.H1(
    children="Avocado Analytics",
    style={"fontSize": "48px", "color": "red"},
),
```

If you want to include your own local CSS or JavaScript files, then you need to create a folder called assets/ in the root directory of your project and save the files that you want to add there. By default, Dash automatically serves any file included in assets/. This will also work for adding a favicon or embedding images, as you'll see in a bit.

Then you can use the className or id arguments of the components to adjust their styles using CSS. These arguments correspond with the class and id attributes when they're transformed into HTML tags.

If you wanted to adjust the font size and text color of the H1 element in app.py, then you could use the className argument as follows:

```
html.H1(
    children="Avocado Analytics",
    className="header-title",
),
```

Setting the className argument will define the class attribute for the <h1> element. You could then use a CSS file in the assets folder to specify how you want it to look:

```
.header-title {
  font-size: 48px;
  color: red;
}
```

You use a class selector to format the heading in your CSS file. This selector will adjust the heading format. You could also use it with another element that needs to share the format by setting className="header-title".

How to Improve the Looks of Your Dashboard

You just covered the basics of styling in Dash. Now, you'll learn how to customize your dashboard's looks. You'll make these improvements:

- Add a favicon and title to the page.
- Change the font family of your dashboard.
- Use an external CSS file to style Dash components.

You'll start by learning how to use external assets in your application. That'll allow you to add a favicon, a custom font family, and a CSS style sheet. Then you'll learn how to use the className argument to apply custom styles to your Dash components.

Adding External Assets to Your Application

Create a folder called assets/ in your project's root directory. Download a favicon from the Twemoji open-source project and save it as favicon.ico in assets/. Finally, create a CSS file in assets/ called style.css and add the code in the collapsible section below:

style.cssShow/Hide

The assets/style.css file contains the styles that you'll apply to components in your application's layout. By now, your project structure should look like this:

```
avocado_analytics/
|
├── assets/
|   ├── favicon.ico
|   └── style.css
|
├── venv/
|
├── app.py
└── avocado.csv
```

Once you start the server, Dash will automatically serve the files located in assets/. You include two files, favicon.ico and style.css, in assets/. To set a default favicon, you don't have to take any additional steps. To use the styles that you defined in style.css, you'll need to use the className argument in Dash components.

You need to make a few changes in app.py. You'll include an external style sheet, add a title to your dashboard, and style the components using the style.css file. Review the changes below. Then, in the last part of this section, you'll find the full code for your updated version of app.py.

Here's how you include an external style sheet and add a title to your dashboard:

```
# app.py
```

```python
# ...

external_stylesheets = [
    {
        "href": (
            "https://fonts.googleapis.com/css2?"
            "family=Lato:wght@400;700&display=swap"
        ),
        "rel": "stylesheet",
    },
]
app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
app.title = "Avocado Analytics: Understand Your Avocados!"

# ...
```

In these code lines, you specify an external CSS file containing a font family, which you want to load in your application. You add external files to the head tag of your application, so they load before the body of your application loads. You use the external_stylesheets argument for adding external CSS files or external_scripts for external JavaScript files like Google Analytics.

You also set the title of your application. This is the text that appears in the title bar of your web browser, in Google's search results, and in social media cards when you share your site.

Customizing the Styles of Components

To use the styles in style.css, you'll need to use the className argument in Dash components. The code below adds a className with a corresponding class selector to each of the components in the header of your dashboard:

```python
# app.py

# ...

app.layout = html.Div(
    children=[
        html.Div(
            children=[
                html.P(children="🥑", className="header-emoji"),
                html.H1(
                    children="Avocado Analytics", className="header-title"
                ),
                html.P(
```

```
                    children=(
                        "Analyze the behavior of avocado prices and the number"
                        " of avocados sold in the US between 2015 and 2018"
                    ),
                    className="header-description",
                ),
            ],
            className="header",


        # ...
```
In the highlighted lines, you can see that you've made three changes to the initial version of the dashboard:

1. There's a new `<div>` element that wraps all the header components.
2. There's a new paragraph element with an avocado emoji, 🥑, that'll serve as a logo on the page.
3. There's a `className` argument in each component. These class names match a class selector in `style.css`, which defines the looks of each component.

For example, the header-description class assigned to the paragraph component starting with "Analyze the behavior of avocado prices" has a corresponding selector in style.css. In that file, you'll see the following:

```
.header-description {
    color: #CFCFCF;
    margin: 4px auto;
    text-align: center;
    max-width: 384px;
}
```
These lines define the format for the header-description class selector. They'll change the color, margin, alignment, and maximum width of any component with className="header-description". All the components have corresponding class selectors in the CSS file.

The other significant change is in the graphs. Here's the new code for the price chart:

```
1# app.py
2
3# ...
4
5app.layout = html.Div(
6    children=[
7        # ...
8
```

```
9           html.Div(
10              children=[
11                  html.Div(
12                      children=dcc.Graph(
13                          id="price-chart",
14                          config={"displayModeBar": False},
15                          figure={
16                              "data": [
17                                  {
18                                      "x": data["Date"],
19                                      "y": data["AveragePrice"],
20                                      "type": "lines",
21                                      "hovertemplate": (
22                                          "$%{y:.2f}<extra></extra>"
23                                      ),
24                                  },
25                              ],
26                              "layout": {
27                                  "title": {
28                                      "text": "Average Price of Avocados",
29                                      "x": 0.05,
30                                      "xanchor": "left",
31                                  },
32                                  "xaxis": {"fixedrange": True},
33                                  "yaxis": {
34                                      "tickprefix": "$",
35                                      "fixedrange": True,
36                                  },
37                                  "colorway": ["#17b897"],
38                              },
39                          },
40                      ),
41                      className="card",
42                  ),
43
44                  # ...
45
46              ],
```

```
47            className="wrapper",
48        ),
49    ]
50)
51
52# ...
```
In this code, you define a `className` and a few customizations for the `config` and `figure` parameters of your chart. Here are the changes:

- **Line 14**: You remove the floating toolbar that Plotly shows by default.
- **Lines 21 to 23:** You set the hover template so that when users hover over a data point, it shows the price in dollars. Instead of `2.5`, it'll show as `$2.5`.
- **Lines 26 to 38:** You adjust the axes, the color of the figure, and the title format in the layout section of the graph.
- **Lines 11 and 41:** You wrap the graph in a `<div>` element with a `"card"` class. This will give the graph a white background and add a small shadow below it.
- **Lines 9 and 47:** You add a `<div>` element that wraps the graph components with a `wrapper` class.

There are similar adjustments to the sales and volume charts. You can see those in the full code for the updated `app.py` in the collapsible section below:

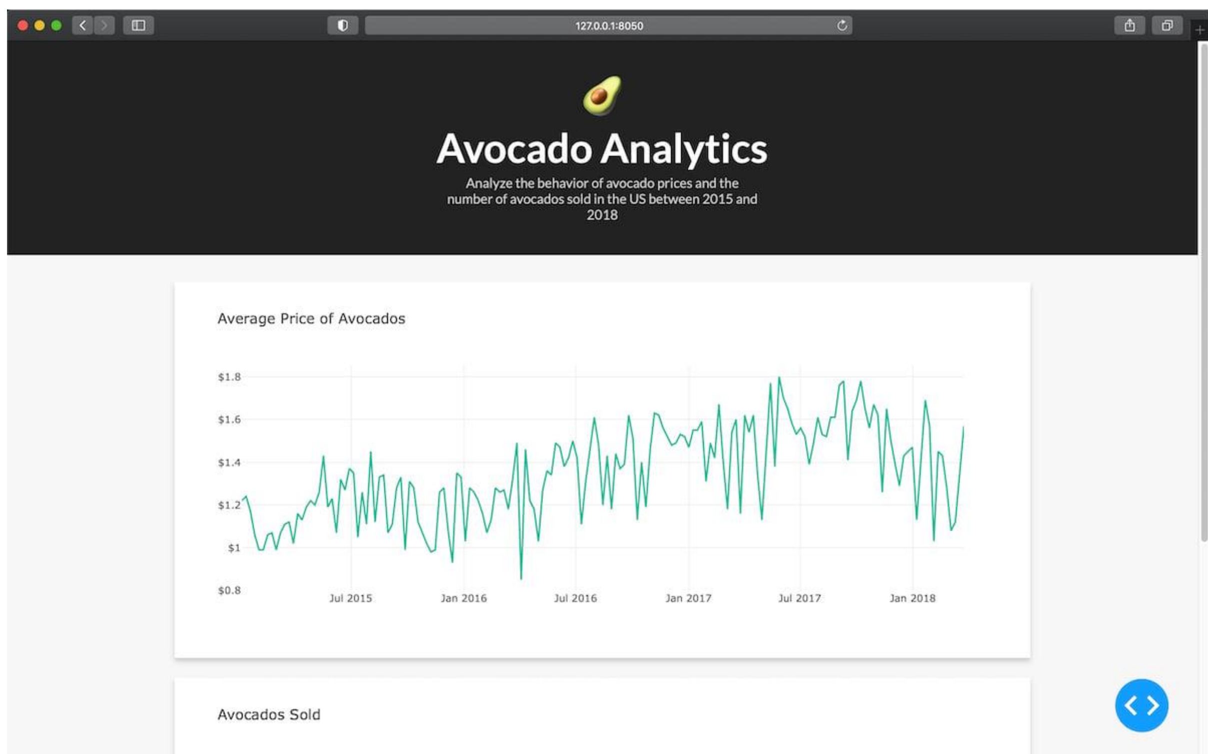
```
# app.py

import pandas as pd
from dash import Dash, dcc, html

data = (
    pd.read_csv("avocado.csv")
    .query("type == 'conventional' and region == 'Albany'")
    .assign(Date=lambda data: pd.to_datetime(data["Date"], format="%Y-%m-%d"))
    .sort_values(by="Date")
)

external_stylesheets = [
    {
        "href": (
            "https://fonts.googleapis.com/css2?"
            "family=Lato:wght@400;700&display=swap"
        ),
        "rel": "stylesheet",
    },
```

```
]
app = Dash(__name__, external_stylesheets=external_stylesheets)
app.title = "Avocado Analytics: Understand Your Avocados!"

app.layout = html.Div(
    children=[
        html.Div(
            children=[
                html.P(children="🥑", className="header-emoji"),
                html.H1(
                    children="Avocado Analytics", className="header-title"
                ),
                html.P(
                    children=(
                        "Analyze the behavior of avocado prices and the number"
                        " of avocados sold in the US between 2015 and 2018"
                    ),
                    className="header-description",
                ),
            ],
            className="header",
        ),
        html.Div(
            children=[
                html.Div(
                    children=dcc.Graph(
                        id="price-chart",
                        config={"displayModeBar": False},
                        figure={
                            "data": [
                                {
                                    "x": data["Date"],
                                    "y": data["AveragePrice"],
                                    "type": "lines",
                                    "hovertemplate": (
                                        "$%{y:.2f}<extra></extra>"
                                    ),
                                },
```

```
                ],
                "layout": {
                    "title": {
                        "text": "Average Price of Avocados",
                        "x": 0.05,
                        "xanchor": "left",
                    },
                    "xaxis": {"fixedrange": True},
                    "yaxis": {
                        "tickprefix": "$",
                        "fixedrange": True,
                    },
                    "colorway": ["#17b897"],
                },
            },
        ),
        className="card",
    ),
    html.Div(
        children=dcc.Graph(
            id="volume-chart",
            config={"displayModeBar": False},
            figure={
                "data": [
                    {
                        "x": data["Date"],
                        "y": data["Total Volume"],
                        "type": "lines",
                    },
                ],
                "layout": {
                    "title": {
                        "text": "Avocados Sold",
                        "x": 0.05,
                        "xanchor": "left",
                    },
                    "xaxis": {"fixedrange": True},
                    "yaxis": {"fixedrange": True},
```

```
                                "colorway": ["#E12D39"],
                            },
                        },
                    ),
                    className="card",
                ),
            ],
            className="wrapper",
        ),
    ]
)


if __name__ == "__main__":
    app.run_server(debug=True)
```

This is the updated version of `app.py`. It has the required changes in the code to add a favicon and a page title, update the font family, and use an external CSS file. After these changes, your dashboard should look like this:



Add Interactivity to Your Dash Apps Using Callbacks

In this section, you'll learn how to add interactive elements to your dashboard.

Dash's interactivity is based on a reactive programming paradigm. This means that you can link components with elements of your app that you want to update. If a user interacts with an input component like a dropdown or a range slider, then the output, such as a graph, will react automatically to the changes in the input.

Now you're going to make your dashboard interactive. This new version of your dashboard will allow the user to interact with the following filters:

- Region
- Type of avocado
- Date range

The collapsible boxes below contain the full source code that you'll be exploring in this section. Start by replacing your local `app.py` with the new version in the collapsible section below:

app.pyShow/Hide
Next, replace `style.css` with the code in the collapsible section below:

style.cssShow/Hide
Now you're ready to explore the interactive components that you've added to your application!

How to Create Interactive Components

First, you'll learn how to create components that users can interact with. For that, you'll include a new `<div>` element above your charts. It'll include two dropdowns and a date range selector that the user can use to filter the data and update the graphs.

You start by changing how you process your data. You no longer filter the data when you read them. Instead you find the regions and avocado types that are present in your data:

```
# app.py

# ...

data = (
    pd.read_csv("avocado.csv")
    # Remove .query(...)
    .assign(Date=lambda data: pd.to_datetime(data["Date"], format="%Y-%m-%d"))
    .sort_values(by="Date")
)
regions = data["region"].sort_values().unique()
avocado_types = data["type"].sort_values().unique()
```

```
# ...
```

Next, you'll use `regions` and `avocado_types` to populate a few dropdowns. Here's how that looks in `app.py`:

```
1# app.py
2
3# ...
4
5app.layout = html.Div(
6    children=[
7
8        # ...
9
10        html.Div(
11            children=[
12                html.Div(
13                    children=[
14                        html.Div(children="Region", className="menu-title"),
15                        dcc.Dropdown(
16                            id="region-filter",
17                            options=[
18                                {"label": region, "value": region}
19                                for region in regions
20                            ],
21                            value="Albany",
22                            clearable=False,
23                            className="dropdown",
24                        ),
25                    ]
26                ),
27                html.Div(
28                    children=[
29                        html.Div(children="Type", className="menu-title"),
30                        dcc.Dropdown(
31                            id="type-filter",
32                            options=[
33                                {
34                                    "label": avocado_type.title(),
```

```
35                                      "value": avocado_type,
36                                  }
37                                  for avocado_type in avocado_types
38                              ],
39                              value="organic",
40                              clearable=False,
41                              searchable=False,
42                              className="dropdown",
43                          ),
44                      ],
45                  ),
46                  html.Div(
47                      children=[
48                          html.Div(
49                              children="Date Range", className="menu-title"
50                          ),
51                          dcc.DatePickerRange(
52                              id="date-range",
53                              min_date_allowed=data["Date"].min().date(),
54                              max_date_allowed=data["Date"].max().date(),
55                              start_date=data["Date"].min().date(),
56                              end_date=data["Date"].max().date(),
57                          ),
58                      ]
59                  ),
60              ],
61              className="menu",
62          ),
63
64          # ...
```

On lines 10 to 62, you define a `<div>` element above your graphs, consisting of two dropdowns and a date range selector. It'll serve as a menu that the user will use to interact with the data:

Region
Albany
Type
organic
Date Range
01/04/2015 → 03/25/2018

The first component in the menu is the Region dropdown. Focus on the code for that component:

```
html.Div(
    children=[
        html.Div(children="Region", className="menu-title"),
        dcc.Dropdown(
            id="region-filter",
            options=[
                {"label": region, "value": region}
                for region in regions
            ],
            value="Albany",
            clearable=False,
            className="dropdown",
        ),
    ]
),
```

Here, you define the dropdown that users will use to filter the data by region. In addition to the title, it has a `dcc.Dropdown` component. Here's what each of the parameters means:

- `id` is the **identifier** of this element.
- `options` indicates the **options** shown when the dropdown is selected. It expects a dictionary with labels and values.
- `value` is the **default value** when the page loads.
- `clearable` allows the user to **leave this field empty** if set to `True`.
- `className` is a CSS **class selector** used for applying styles.

The Type and Date Range selectors follow the same structure as the Region dropdown. Feel free to review them on your own.

Next, take a look at the `dcc.Graphs` components:

```
# app.py

# ...

app.layout = html.Div(
    children=[
```

```python
        # ...

        html.Div(
            children=[
                html.Div(
                    children=dcc.Graph(
                        id="price-chart",
                        config={"displayModeBar": False},
                    ),
                    className="card",
                ),
                html.Div(
                    children=dcc.Graph(
                        id="volume-chart",
                        config={"displayModeBar": False},
                    ),
                    className="card",
                ),
            ],
            className="wrapper",
        ),
    ]
)

# ...
```

In this part of the code, you define the `dcc.Graph` components. You may have noticed that, compared to the previous version of the dashboard, the components are missing the `figure` argument. That's because a callback function will now generate the `figure` argument using the inputs that the user sets using the Region, Type, and Date Range selectors.

How to Define Callbacks

You've defined how the user will interact with your application. Now you need to make your application react to user interactions. For that, you'll use **callback functions.**

Dash's callback functions are regular Python functions with an `app.callback` decorator. In Dash, when an input changes, a callback function is triggered. The function performs some predetermined operations, like filtering a dataset, and returns an output to the application. In essence, callbacks link inputs and outputs in your app.

Here's the callback function that's used for updating the graphs:

```
1# app.py
2
3# ..
4
5@app.callback(
6    Output("price-chart", "figure"),
7    Output("volume-chart", "figure"),
8    Input("region-filter", "value"),
9    Input("type-filter", "value"),
10   Input("date-range", "start_date"),
11   Input("date-range", "end_date"),
12)
13def update_charts(region, avocado_type, start_date, end_date):
14   filtered_data = data.query(
15       "region == @region and type == @avocado_type"
16       " and Date >= @start_date and Date <= @end_date"
17   )
18   price_chart_figure = {
19       "data": [
20           {
21               "x": filtered_data["Date"],
22               "y": filtered_data["AveragePrice"],
23               "type": "lines",
24               "hovertemplate": "$%{y:.2f}<extra></extra>",
25           },
26       ],
27       "layout": {
28           "title": {
29               "text": "Average Price of Avocados",
30               "x": 0.05,
31               "xanchor": "left",
32           },
33           "xaxis": {"fixedrange": True},
34           "yaxis": {"tickprefix": "$", "fixedrange": True},
35           "colorway": ["#17B897"],
36       },
37   }
```

```
38
39    volume_chart_figure = {
40        "data": [
41            {
42                "x": filtered_data["Date"],
43                "y": filtered_data["Total Volume"],
44                "type": "lines",
45            },
46        ],
47        "layout": {
48            "title": {"text": "Avocados Sold", "x": 0.05, "xanchor": "left"},
49            "xaxis": {"fixedrange": True},
50            "yaxis": {"fixedrange": True},
51            "colorway": ["#E12D39"],
52        },
53    }
54    return price_chart_figure, volume_chart_figure
55
56# ...
```

On lines 6 to 11, you define the inputs and outputs inside the `app.callback` decorator.

First, you define the outputs using `Output` objects. They take two arguments:

1. The identifier of the element that they'll modify when the function executes
2. The property of the element to be modified

For example, `Output("price-chart", "figure")` will update the `figure` property of the `"price-chart"` element.

Then you define the inputs using `Input` objects. They also take two arguments:

1. The identifier of the element that they'll be watching for changes
2. The property of the watched element that they'll be watching for changes

So, `Input("region-filter", "value")` will watch the `"region-filter"` element and its `value` property for changes. The argument passed on to the callback function will be the new value of `region-filter.value`.

Note: The `Input` object that you're using here is imported directly from `dash`. Be careful not to confuse it with the `Input` component coming from `dcc`. These objects aren't interchangeable, and they have different purposes.

On line 13, you define the function that'll be applied when an input changes. It's worth noticing that the arguments of the function will correspond with the order of

the `Input` objects supplied to the callback. There's no explicit relationship between the names of the arguments in the function and the values specified in the Input objects.

Finally, on lines 14 to 54, you define the body of the function. In this case, the function takes the inputs (region, type of avocado, and date range), filters the data, and generates the figure objects for the price and volume charts.

That's all! If you've followed along to this point, then your dashboard should look like this:

Way to go! That's the final version of your dashboard. In addition to making it look beautiful, you also made it interactive. The only missing step is making it public so you can share it with others.

Deploy Your Dash Application to PythonAnywhere

You're done building your application, and you have a beautiful, fully interactive dashboard. Now you'll learn how to deploy it.

Dash apps are Flask apps, so both share the same deployment options. In this section, you'll deploy your app on PythonAnywhere, which offers a free tier for hosting Python web applications in the cloud.

Day-03: Host, run, and code Python in the cloud!

PythonAnywhere by anaconda

How to Create a Free PythonAnywhere Account

Before you get started, make sure you've signed up for a PythonAnywhere **beginner account**, which is completely free of charge and doesn't require you to provide any payment details. That said, it comes with a few limitations that you should be aware of. The most important ones will prevent you from doing the following:

- Running more than one web application at a time
- Defining a custom domain name
- Exceeding the available disk quota (512 MB)
- Using the CPU for longer than 100 seconds per day
- Making unrestricted HTTP requests from your app

For this tutorial, though, you won't need any of that!

If you're based in Europe, then consider signing up through eu.pythonanywhere.com instead of the www.pythonanywhere.com. It'll ensure GDPR compliance for your data, which PythonAnywhere will store on servers in Germany. Because of that, you may also experience slightly faster response times. Finally, if you decide to become a paid customer one day, then you'll be charged in euros instead of US dollars.

Feel free to follow either of the two PythonAnywhere links above if you don't care about any of these features at the moment. Note, however, that once you register a username on one domain, then you won't be able to reuse it on the other!

Another reason to think carefully about your username is that it must be unique, as it'll become a part of your very own domain name, such as in these examples:

```
http://realpython.pythonanywhere.com/
http://realpython.eu.pythonanywhere.com/
```

Once you register a new account on PythonAnywhere, you must confirm your email address so that you can reset the password if you forget it. Also, it might be a good idea to enable two-factor authentication on the *Security* tab in your *Account* settings as an extra security measure.

If you've just created a new account, then you're already good to go. But if you registered a PythonAnywhere account a while ago, then you might need to change your system image to a newer one, which comes with a more recent Python version and newer third-party libraries. At the time of writing, the latest image, called *haggis*, shipped with Python 3.10.5, pandas 1.3.5, and Dash 2.4.1.

Note: You can always check the available batteries for a given image and Python version.

With that out of the way, it's time to create your first web app on PythonAnywhere!

How to Deploy Your Avocado Analytics App

Because Dash apps are Flask apps with some extra frills, you can take advantage of PythonAnywhere's excellent support for this popular Python web framework.

When you're logged in to your PythonAnywhere account, create a new Bash shell console, either from the *Dashboard* or the *Consoles* tab. This will throw you into an interactive prompt of the virtual server, letting you remotely execute commands straight from your web browser.

There are already several useful programs installed for you, including a Git client, which you'll use to get your project's source code into PythonAnywhere. You can also upload files in other ways, but using Git seems the most convenient. If you haven't made your own repository yet, then you might clone Real Python's materials repository with your sample Dash application in it:

```
$ git clone --depth=1 https://github.com/realpython/materials.git
```

The `--depth=1` option tells Git only to clone the latest commit, which saves time and disk space. Note that if you don't want to configure SSH keys for your PythonAnywhere machine, then you'll have to clone a *public* repository using the HTTPS protocol. Since August 2021, cloning private repositories has been possible only after configuring a personal access token in GitHub.

When the repository is cloned, you can move and rename a subfolder with the finished avocado app to your home folder on PythonAnywhere, and then remove the rest of the materials:

```
$ mv materials/python-dash/avocado_analytics_3/ ~/avocado_analytics
$ rm -rf materials/
```

Remember that you only have 512 megabytes of disk space on the free tier at your disposal, and the materials take up a significant portion of that!

At this point, your home folder should look like this:

```
home/realpython/
|
└── avocado_analytics/
    |
    ├── assets/
    |   ├── favicon.ico
    |   └── style.css
    |
    ├── app.py
    └── avocado.csv
```

Of course, the username `realpython` will be different on your account, but the overall folder structure should remain the same.

Now, go the *Web* tab and click the button labeled *Add a new web app*. This will open a wizard, asking you a few questions. First, select **Flask** as the Python web framework of your choice:

Create new web app

Select a Python Web framework

...or select "Manual configuration" if you want detailed control.

» **Django**
» **web2py**
» **Flask**
» **Bottle**
» **Manual configuration** (including virtualenvs)

What other frameworks should we have here? Send us some feedback using the link at the top of the page!

Cancel « Back Next »

Next, you'll see a specific Flask version running on top of the given Python interpreter. Select the latest version available:

Create new web app

Select a Python version

» **Python 3.7** (Flask 2.1.2)
» **Python 3.8** (Flask 2.1.2)
» **Python 3.9** (Flask 2.1.2)
» **Python 3.10** (Flask 2.1.2)

Note: If you'd like to use a different version of Flask to the default version, you can use a virtualenv for your web app. There are instructions here.

Cancel « Back Next »

In the next step, you’ll need to update the file path leading up to the main Python module with your Flask app:

Create new web app ✖

Quickstart new Flask project

Enter a path for a Python file you wish to use to hold your Flask app. **If this file already exists, its contents will be overwritten with the new app.**

Path

/home/realpython/avocado_analytics/flask_app.py

Cancel | « Back | Next »

While you can change it later, it's much easier if you do it right now, so make sure to rename the default `mysite/` folder with `avocado_analytics/` to match your project's name. At the same time, you want to keep the suggested `flask_app.py` filename intact. PythonAnywhere will generate this file and populate it with a demo app, so if you renamed it to `app.py`, then the code that you cloned from GitHub would get overwritten!

Once this is done, you'll be presented with a number of configuration options for your new web app. First, you need to update the working directory of the app to be the same as the source code:

Code:

What your site is running.

Source code:	/home/realpython/avocado_analytics	Go to directory
Working directory:	/home/realpython/avocado_analytics	Go to directory
WSGI configuration file:	/var/www/realpython_pythonanywhere_com_wsgi.py	
Python version:	3.10	

This will ensure that Python can find your `avocado.csv` file at runtime and open it for reading.

Next, you'll need to tweak the default WSGI server configuration, which is slightly different for Dash apps than it is for Flask. PythonAnywhere uses the uWSGI server behind the scenes, which reads the configuration from a special Python module located in the `/var/www/` folder.

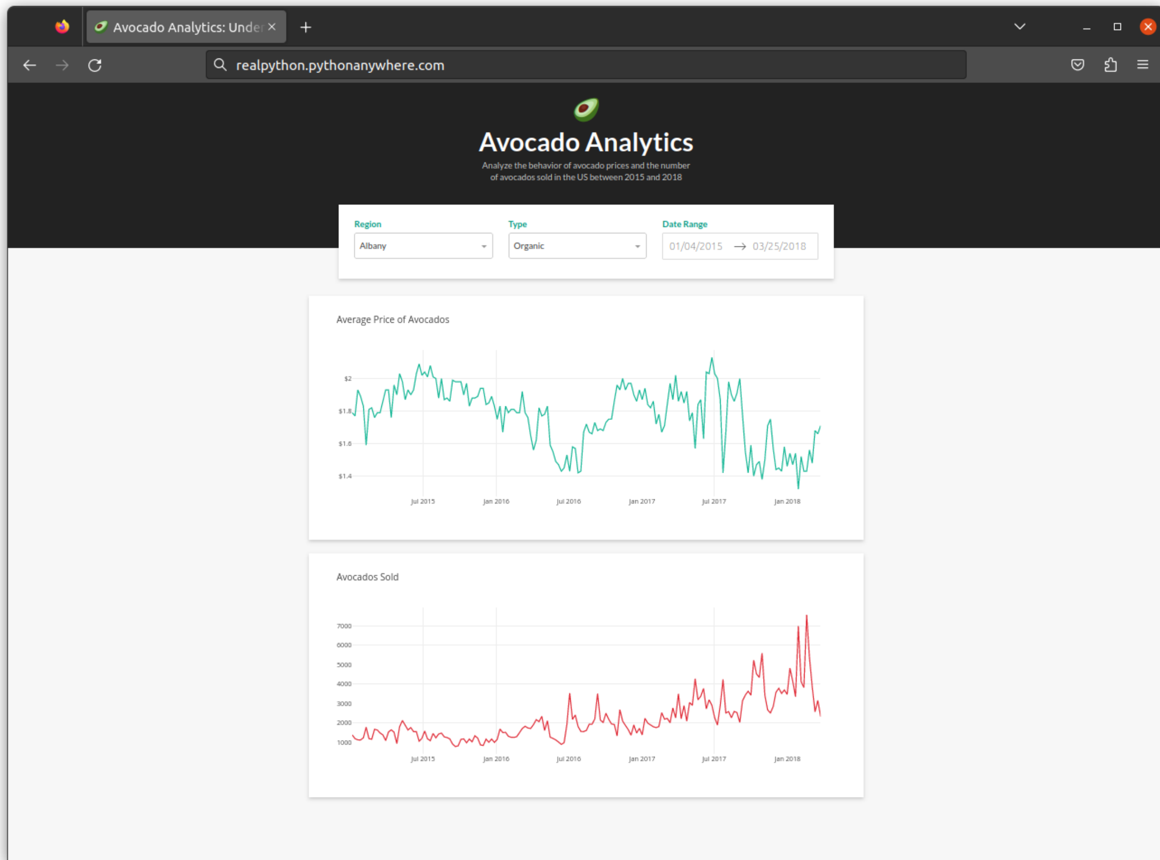
Click the *WSGI configuration file* option visible in the screenshot above to open it in an editor in your web browser:

```
# This file contains the WSGI configuration required to serve up your
# web application at http://<your-username>.pythonanywhere.com/
# It works by setting the variable 'application' to a WSGI handler of some
# description.
#
# The below has been auto-generated for your Flask project

import sys

# add your project directory to the sys.path
project_home = '/home/realpython/avocado_analytics'
if project_home not in sys.path:
    sys.path = [project_home] + sys.path

# import flask app but need to call it "application" for WSGI to work
-from flask_app import app as application  # noqa
+from app import app
+application = app.server
```

You need to rename the `flask_app` module generated by the wizard to the actual `app` module that came with your avocado project. Besides that, you must expose the callable WSGI application through the Dash app's `.server` field, as described in the official help page on PythonAnywhere. You might as well double-check if the path in your `project_home` variable is correct.

Finally, save the file by hitting `Ctrl` + `S`, go back to the *Web* tab, and click the green button to reload your web app:

When you visit the corresponding URL of your web app deployed to PythonAnywhere, you should see the familiar interface:



Avocado Analytics Web App Deployed to PythonAnywhere

That's it! Note that you never installed Dash or pandas because they were already shipped with PythonAnywhere. Also, you didn't have to configure static resources, which are typically served by the web server rather than Flask, because Dash takes care of them automatically.

Note: If you need more control over the external library versions, then you can use virtualenvwrapper to create a virtual environment for the platform and manually install those dependencies. Unfortunately, doing so will likely consume all of your disk space and drain your CPU bandwidth to the point you'll end up in the tarpit.

You can now share your Dash apps with the world by deploying them to PythonAnywhere or other web hosting providers.

Day-04: Interactive Data Visualization in Python With Bokeh

Bokeh prides itself on being a library for interactive data visualization.

Unlike popular counterparts in the Python visualization space, like Matplotlib and Seaborn, Bokeh renders its graphics using HTML and JavaScript. This makes it a great candidate for building web-based dashboards and applications. However, it's an equally powerful tool for exploring and understanding your data or creating beautiful custom charts for a project or report.

Using a number of examples on a real-world dataset, the goal of this tutorial is to get you up and running with Bokeh.

- Transform your data into visualizations, using Bokeh
- Customize and organize your visualizations
- Add interactivity to your visualizations

Building a visualization with Bokeh involves the following steps:

- Prepare the data
- Determine where the visualization will be rendered
- Set up the figure(s)
- Connect to and draw your data
- Organize the layout
- Preview and save your beautiful data creation

Prepare the Data

Any good data visualization starts with—you guessed it—data. If you need a quick refresher on handling data in Python.

This step commonly involves data handling libraries like Pandas and Numpy and is all about taking the required steps to transform it into a form that is best suited for your intended visualization.

Determine Where the Visualization Will Be Rendered

At this step, you'll determine how you want to generate and ultimately view your visualization. In this tutorial, you'll learn about two common options that Bokeh provides: generating a static HTML file and rendering your visualization inline in a Jupyter Notebook.

Set up the Figure(s)

From here, you'll assemble your figure, preparing the canvas for your visualization. In this step, you can customize everything from the titles to the tick marks. You can also set up a suite of tools that can enable various user interactions with your visualization.

Connect to and Draw Your Data

Next, you'll use Bokeh's multitude of renderers to give shape to your data. Here, you have the flexibility to draw your data from scratch using the many available marker and shape options, all of which are easily customizable. This functionality gives you incredible creative freedom in representing your data.

Additionally, Bokeh has some built-in functionality for building things like stacked bar charts and plenty of examples for creating more advanced visualizations like network graphs and maps.

Organize the Layout

If you need more than one figure to express your data, Bokeh's got you covered. Not only does Bokeh offer the standard grid-like layout options, but it also allows you to easily organize your visualizations into a tabbed layout in just a few lines of code.

In addition, your plots can be quickly linked together, so a selection on one will be reflected on any combination of the others.

Preview and Save Your Beautiful Data Creation

Finally, it's time to see what you created.

Whether you're viewing your visualization in a browser or notebook, you'll be able to explore your visualization, examine your customizations, and play with any interactions that were added.

If you like what you see, you can save your visualization to an image file. Otherwise, you can revisit the steps above as needed to bring your data vision to reality.

That's it! Those six steps are the building blocks for a tidy, flexible template that can be used to take your data from the table to the big screen:

```
"""Bokeh Visualization Template

This template is a general outline for turning your data into a
visualization using Bokeh.
"""
# Data handling
import pandas as pd
import numpy as np

# Bokeh libraries
from bokeh.io import output_file, output_notebook
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource
from bokeh.layouts import row, column, gridplot
from bokeh.models.widgets import Tabs, Panel

# Prepare the data

# Determine where the visualization will be rendered
```

```
output_file('filename.html')  # Render to static HTML, or
output_notebook()  # Render inline in a Jupyter Notebook

# Set up the figure(s)
fig = figure()  # Instantiate a figure() object

# Connect to and draw the data

# Organize the layout

# Preview and save
show(fig)  # See what I made, and save if I like it
```

Some common code snippets that are found in each step are previewed above, and you'll see how to fill out the rest as you move through the rest of the tutorial!

Generating Your First Figure

There are multiple ways to output your visualization in Bokeh. In this tutorial, you'll see these two options:

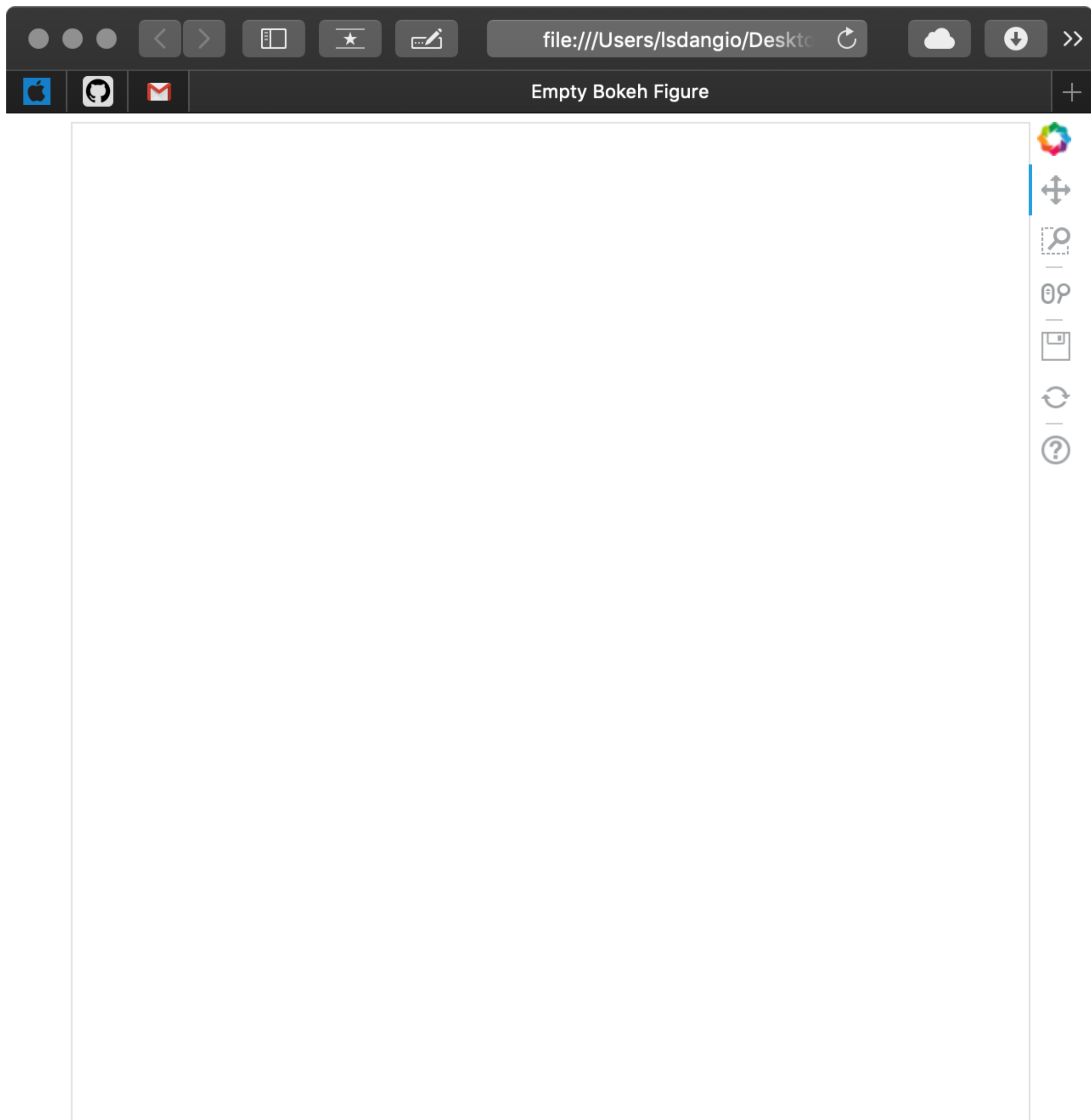
- **`output_file('filename.html')`** will write the visualization to a static HTML file.
- **`output_notebook()`** will render your visualization directly in a Jupyter Notebook.

It's important to note that neither function will actually show you the visualization. That doesn't happen until `show()` is called. However, they will ensure that, when `show()` is called, the visualization appears where you intend it to.

By calling both `output_file()` and `output_notebook()` in the same execution, the visualization will be rendered both to a static HTML file and inline in the notebook. However, if for whatever reason you run multiple `output_file()` commands in the same execution, only the last one will be used for rendering.

This is a great opportunity to give you your first glimpse at a default Bokeh `figure()` using `output_file()`:

```
# Bokeh Libraries
from bokeh.io import output_file
from bokeh.plotting import figure, show

# The figure will be rendered in a static HTML file called
output_file_test.html
output_file('output_file_test.html',
            title='Empty Bokeh Figure')

# Set up a generic figure() object
fig = figure()

# See what it looks like
show(fig)
```



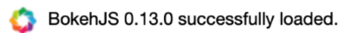
As you can see, a new browser window opened with a tab called *Empty Bokeh Figure* and an empty figure. Not shown is the file generated with the name *output_file_test.html* in your current working directory.

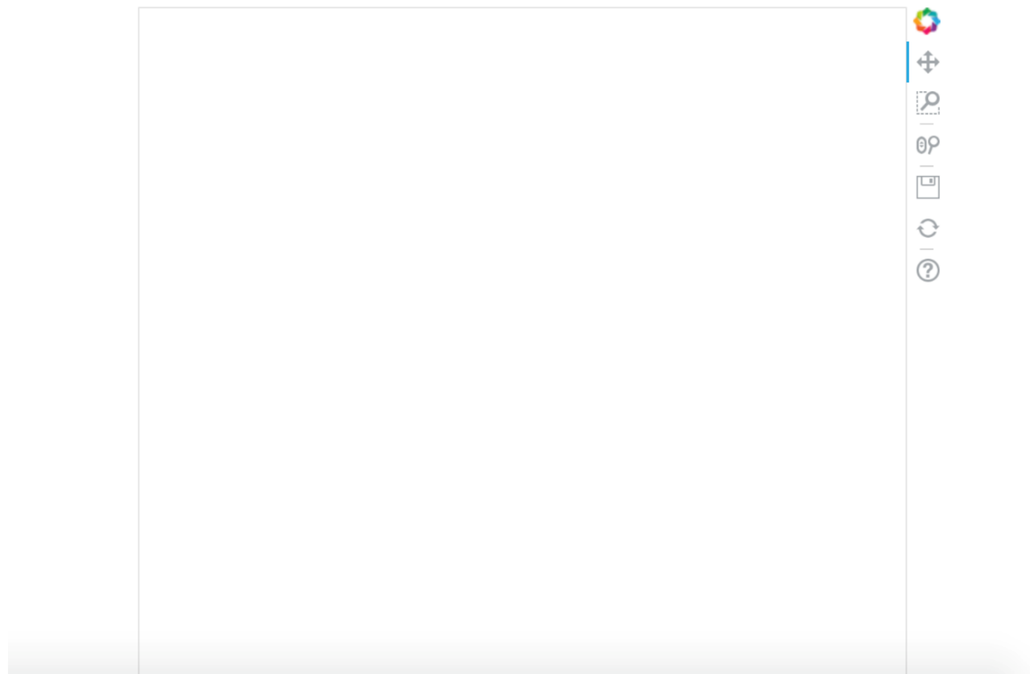
If you were to run the same code snippet with `output_notebook()` in place of `output_file()`, assuming you have a Jupyter Notebook fired up and ready to go, you will get the following:

```
# Bokeh Libraries
from bokeh.io import output_notebook
from bokeh.plotting import figure, show

# The figure will be right in my Jupyter Notebook
output_notebook()

# Set up a generic figure() object
```

```
fig = figure()

# See what it looks like
show(fig)
```

In [1]:
```
# Bokeh Libraries
from bokeh.io import output_notebook
from bokeh.plotting import figure, show

# The figure will be right in my Jupyter Notebook
output_notebook()

# Set up a generic figure() object
fig = figure()

# See what it looks like
show(fig)
```





As you can see, the result is the same, just rendered in a different location.

More information about both `output_file()` and `output_notebook()` can be found in the [Bokeh official docs].

Note: Sometimes, when rendering multiple visualizations sequentially, you'll see that past renders are not being cleared with each execution. If you experience this, import and run the following between executions:

```
# Import reset_output (only needed once)
from bokeh.plotting import reset_output

# Use reset_output() between subsequent show() calls, as needed
reset_output()
```

Before moving on, you may have noticed that the default Bokeh figure comes pre-loaded with a toolbar. This is an important sneak preview into the interactive elements of Bokeh that

come right out of the box. You'll find out more about the toolbar and how to configure it in the Adding Interaction section at the end of this tutorial.

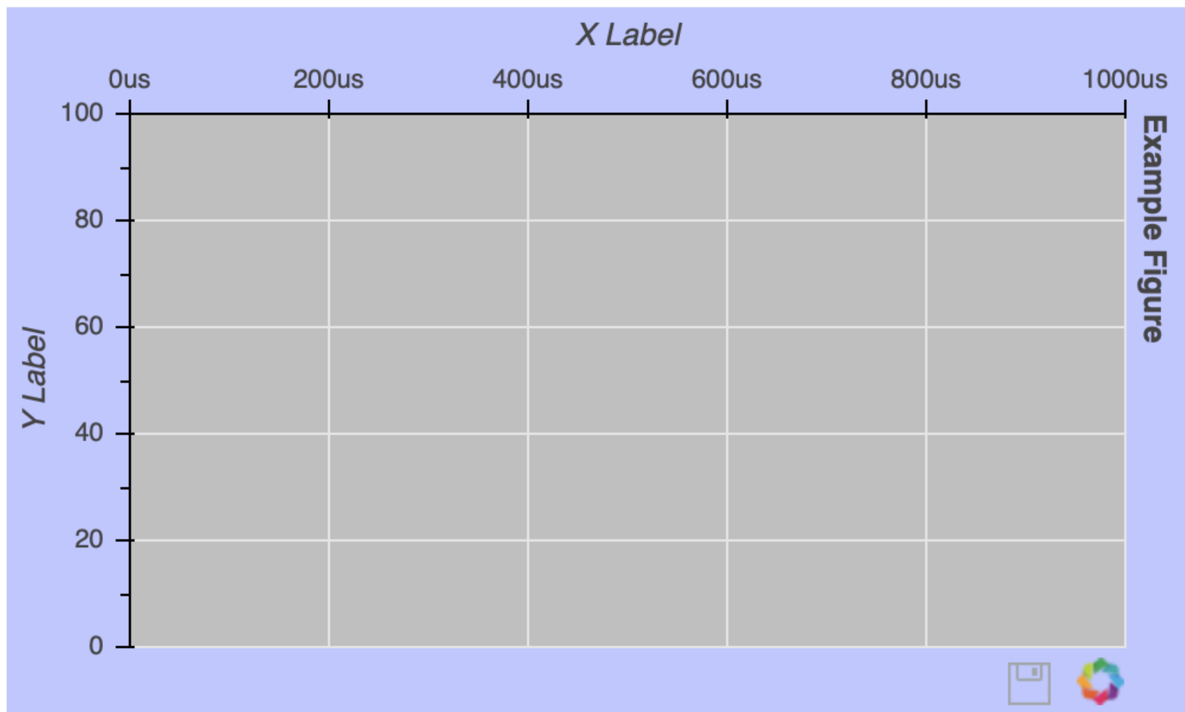
Getting Your Figure Ready for Data

Now that you know how to create and view a generic Bokeh figure either in a browser or Jupyter Notebook, it's time to learn more about how to configure the `figure()` object.

The `figure()` object is not only the foundation of your data visualization but also the object that unlocks all of Bokeh's available tools for visualizing data. The Bokeh figure is a subclass of the Bokeh Plot object, which provides many of the parameters that make it possible to configure the aesthetic elements of your figure.

To show you just a glimpse into the customization options available, let's create the ugliest figure ever:

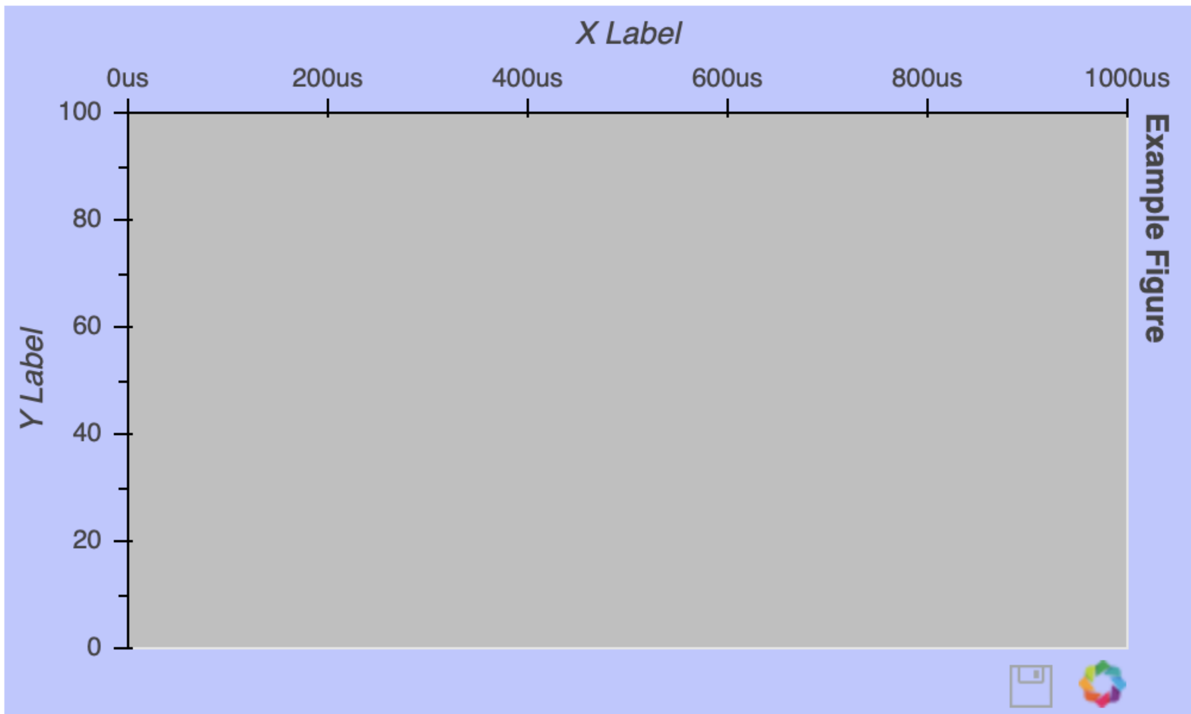
```
# Bokeh Libraries
from bokeh.io import output_notebook
from bokeh.plotting import figure, show

# The figure will be rendered inline in my Jupyter Notebook
output_notebook()

# Example figure
fig = figure(background_fill_color='gray',
             background_fill_alpha=0.5,
             border_fill_color='blue',
             border_fill_alpha=0.25,
             plot_height=300,
             plot_width=500,
             h_symmetry=True,
             x_axis_label='X Label',
             x_axis_type='datetime',
             x_axis_location='above',
             x_range=('2018-01-01', '2018-06-30'),
             y_axis_label='Y Label',
             y_axis_type='linear',
             y_axis_location='left',
             y_range=(0, 100),
             title='Example Figure',
             title_location='right',
             toolbar_location='below',
             tools='save')

# See what it looks like
show(fig)
```



Once the `figure()` object is instantiated, you can still configure it after the fact. Let's say you want to get rid of the gridlines:

```
# Remove the gridlines from the figure() object
fig.grid.grid_line_color = None

# See what it looks like
show(fig)
```

The gridline properties are accessible via the figure's `grid` attribute. In this case, setting `grid_line_color` to `None` effectively removes the gridlines altogether. More details about figure attributes can be found below the fold in the Plot class documentation.



Note: If you're working in a notebook or IDE with auto-complete functionality, this feature can definitely be your friend! With so many customizable elements, it can be very helpful in discovering the available options:

Otherwise, doing a quick web search, with the keyword *bokeh* and what you are trying to do, will generally point you in the right direction.

There is tons more I could touch on here, but don't feel like you're missing out. I'll make sure to introduce different figure tweaks as the tutorial progresses. Here are some other helpful links on the topic:

- The Bokeh Plot Class is the superclass of the `figure()` object, from which figures inherit a lot of their attributes.
- The Figure Class documentation is a good place to find more detail about the arguments of the `figure()` object.

Here are a few specific customization options worth checking out:

- Text Properties covers all the attributes related to changing font styles, sizes, colors, and so forth.
- TickFormatters are built-in objects specifically for formatting your axes using Python-like string formatting syntax.

Sometimes, it isn't clear how your figure needs to be customized until it actually has some data visualized in it, so next you'll learn how to make that happen.

Drawing Data With Glyphs

An empty figure isn't all that exciting, so let's look at glyphs: the building blocks of Bokeh visualizations. A glyph is a vectorized graphical shape or marker that is used to represent your data, like a circle or square. More examples can be found in the Bokeh gallery. After you create your figure, you are given access to a bevy of configurable glyph methods.

Let's start with a very basic example, drawing some points on an x-y coordinate grid:

```
# Bokeh Libraries
from bokeh.io import output_file
from bokeh.plotting import figure, show

# My x-y coordinate data
x = [1, 2, 1]
y = [1, 1, 2]
# Output the visualization directly in the notebook
output_file('first_glyphs.html', title='First Glyphs')


# Create a figure with no toolbar and axis ranges of [0,3]
fig = figure(title='My Coordinates',
             plot_height=300, plot_width=300,
             x_range=(0, 3), y_range=(0, 3),
             toolbar_location=None)
```

```
# Draw the coordinates as circles

fig.circle(x=x, y=y,

         color='green', size=10, alpha=0.5)

# Show plot

show(fig)
```

First Glyphs

Once your figure is instantiated, you can see how it can be used to draw the x-y coordinate data using customized circle glyphs.

Here are a few categories of glyphs:

Marker includes shapes like circles, diamonds, squares, and triangles and is effective for creating visualizations like scatter and bubble charts.

Line covers things like single, step, and multi-line shapes that can be used to build line charts.

Bar/Rectangle shapes can be used to create traditional or stacked bar (hbar) and column (vbar) charts as well as waterfall or gantt charts.

Information about the glyphs above, as well as others, can be found in Bokeh's Reference Guide.

These glyphs can be combined as needed to fit your visualization needs. Let's say I want to create a visualization that shows how many words I wrote per day to make this tutorial, with an overlaid trend line of the cumulative word count:

```
import numpy as np

# Bokeh libraries

from bokeh.io import output_notebook

from bokeh.plotting import figure, show



# My word count data

day_num = np.linspace(1, 10, 10)

daily_words = [450, 628, 488, 210, 287, 791, 508, 639, 397, 943]

cumulative_words = np.cumsum(daily_words)
```

```
# Output the visualization directly in the notebook

output_notebook()

# Create a figure with a datetime type x-axis

fig = figure(title='My Tutorial Progress',

            plot_height=400, plot_width=700,

            x_axis_label='Day Number', y_axis_label='Words Written',

            x_minor_ticks=2, y_range=(0, 6000),

            toolbar_location=None)

# The daily words will be represented as vertical bars (columns)

fig.vbar(x=day_num, bottom=0, top=daily_words,

        color='blue', width=0.75,

        legend='Daily')


# The cumulative sum will be a trend line

fig.line(x=day_num, y=cumulative_words,

        color='gray', line_width=1,

        legend='Cumulative')

# Put the legend in the upper left corner

fig.legend.location = 'top_left'


# Let's check it out

show(fig)
```

Multi-Glyph Example

To combine the columns and lines on the figure, they are simply created using the same figure() object.

Additionally, you can see above how seamlessly a legend can be created by setting the legend property for each glyph. The legend was then moved to the upper left corner of the plot by assigning 'top_left' to fig.legend.location.

You can check out much more info about styling legends. Teaser: they will show up again later in the tutorial when we start digging into interactive elements of the visualization.

A Quick Aside About Data

Anytime you are exploring a new visualization library, it's a good idea to start with some data in a domain you are familiar with. The beauty of Bokeh is that nearly any idea you have should be possible. It's just a matter of how you want to leverage the available tools to do so.

The remaining examples will use publicly available data from Kaggle, which has information about the National Basketball Association's (NBA) 2017-18 season, specifically:

2017-18_playerBoxScore.csv: game-by-game snapshots of player statistics

2017-18_teamBoxScore.csv: game-by-game snapshots of team statistics

2017-18_standings.csv: daily team standings and rankings

This data has nothing to do with what I do for work, but I love basketball and enjoy thinking about ways to visualize the ever-growing amount of data associated with it.

If you don't have data to play with from school or work, think about something you're interested in and try to find some data related to that. It will go a long way in making both the learning and the creative process faster and more enjoyable!

To follow along with the examples in the tutorial, you can download the datasets from the links above and read them into a Pandas DataFrame using the following commands:

```
import pandas as pd



# Read the csv files

player_stats = pd.read_csv('2017-18_playerBoxScore.csv', parse_dates=['gmDate'])

team_stats = pd.read_csv('2017-18_teamBoxScore.csv', parse_dates=['gmDate'])

standings = pd.read_csv('2017-18_standings.csv', parse_dates=['stDate'])
```

This code snippet reads the data from the three CSV files and automatically interprets the date columns as datetime objects.

It's now time to get your hands on some real data.

Using the ColumnDataSource Object

The examples above used Python lists and Numpy arrays to represent the data, and Bokeh is well equipped to handle these datatypes. However, when it comes to data in Python, you are most likely going to come across Python dictionaries and Pandas DataFrames, especially if you're reading in data from a file or external data source.

Bokeh is well equipped to work with these more complex data structures and even has built-in functionality to handle them, namely the ColumnDataSource.

You may be asking yourself, "Why use a ColumnDataSource when Bokeh can interface with other data types directly?"

For one, whether you reference a list, array, dictionary, or DataFrame directly, Bokeh is going to turn it into a ColumnDataSource behind the scenes anyway. More importantly, the ColumnDataSource makes it much easier to implement Bokeh's interactive affordances.

The ColumnDataSource is foundational in passing the data to the glyphs you are using to visualize. Its primary functionality is to map names to the columns of your data. This makes it easier for you to reference elements of your data when building your visualization. It also makes it easier for Bokeh to do the same when building your visualization.

The ColumnDataSource can interpret three types of data objects:

Python dict: The keys are names associated with the respective value sequences (lists, arrays, and so forth).

Pandas DataFrame: The columns of the DataFrame become the reference names for the ColumnDataSource.

Pandas groupby: The columns of the ColumnDataSource reference the columns as seen by calling groupby.describe().

Let's start by visualizing the race for first place in the NBA's Western Conference in 2017-18 between the defending champion Golden State Warriors and the challenger Houston Rockets. The daily win-loss records of these two teams is stored in a DataFrame named west_top_2:

```
>>> west_top_2 = (standings[(standings['teamAbbr'] == 'HOU') | (standings['teamAbbr'] == 'GS')]

...            .loc[:, ['stDate', 'teamAbbr', 'gameWon']]

...            .sort_values(['teamAbbr','stDate']))

>>> west_top_2.head()

       stDate teamAbbr  gameWon

9   2017-10-17      GS       0

39  2017-10-18      GS       0

69  2017-10-19      GS       0
```

99 2017-10-20   GS   1

129 2017-10-21   GS   1

From here, you can load this DataFrame into two ColumnDataSource objects and visualize the race:

```
# Bokeh libraries

from bokeh.plotting import figure, show

from bokeh.io import output_file

from bokeh.models import ColumnDataSource


# Output to file

output_file('west-top-2-standings-race.html',

            title='Western Conference Top 2 Teams Wins Race')


# Isolate the data for the Rockets and Warriors

rockets_data = west_top_2[west_top_2['teamAbbr'] == 'HOU']

warriors_data = west_top_2[west_top_2['teamAbbr'] == 'GS']


# Create a ColumnDataSource object for each team

rockets_cds = ColumnDataSource(rockets_data)

warriors_cds = ColumnDataSource(warriors_data)


# Create and configure the figure

fig = figure(x_axis_type='datetime',

             plot_height=300, plot_width=600,

             title='Western Conference Top 2 Teams Wins Race, 2017-18',

             x_axis_label='Date', y_axis_label='Wins',
```

```
            toolbar_location=None)

# Render the race as step lines
fig.step('stDate', 'gameWon',
         color='#CE1141', legend='Rockets',
         source=rockets_cds)
fig.step('stDate', 'gameWon',
         color='#006BB6', legend='Warriors',
         source=warriors_cds)

# Move the legend to the upper left corner
fig.legend.location = 'top_left'

# Show the plot
show(fig)
```

Rockets vs. Warriors

Notice how the respective ColumnDataSource objects are referenced when creating the two lines. You simply pass the original column names as input parameters and specify which ColumnDataSource to use via the source property.

The visualization shows the tight race throughout the season, with the Warriors building a pretty big cushion around the middle of the season. However, a bit of a late-season slide allowed the Rockets to catch up and ultimately surpass the defending champs to finish the season as the Western Conference number-one seed.

Note: In Bokeh, you can specify colors either by name, hex value, or RGB color code.

For the visualization above, a color is being specified for the respective lines representing the two teams. Instead of using CSS color names like 'red' for the Rockets and 'blue' for the Warriors, you might have wanted to add a nice visual touch by using the official team colors in the form of hex color codes. Alternatively, you could have used tuples representing RGB color codes: (206, 17, 65) for the Rockets, (0, 107, 182) for the Warriors.

Bokeh provides a helpful list of CSS color names categorized by their general hue. Also, htmlcolorcodes.com is a great site for finding CSS, hex, and RGB color codes.

ColumnDataSource objects can do more than just serve as an easy way to reference DataFrame columns. The ColumnDataSource object has three built-in filters that can be used to create views on your data using a CDSView object:

GroupFilter selects rows from a ColumnDataSource based on a categorical reference value

IndexFilter filters the ColumnDataSource via a list of integer indices

BooleanFilter allows you to use a list of boolean values, with True rows being selected

In the previous example, two ColumnDataSource objects were created, one each from a subset of the west_top_2 DataFrame. The next example will recreate the same output from one ColumnDataSource based on all of west_top_2 using a GroupFilter that creates a view on the data:

```
# Bokeh libraries

from bokeh.plotting import figure, show

from bokeh.io import output_file

from bokeh.models import ColumnDataSource, CDSView, GroupFilter


# Output to file

output_file('west-top-2-standings-race.html',

        title='Western Conference Top 2 Teams Wins Race')


# Create a ColumnDataSource

west_cds = ColumnDataSource(west_top_2)


# Create views for each team

rockets_view = CDSView(source=west_cds,

              filters=[GroupFilter(column_name='teamAbbr', group='HOU')])

warriors_view = CDSView(source=west_cds,
```

```
                filters=[GroupFilter(column_name='teamAbbr', group='GS')])

# Create and configure the figure
west_fig = figure(x_axis_type='datetime',
            plot_height=300, plot_width=600,
            title='Western Conference Top 2 Teams Wins Race, 2017-18',
            x_axis_label='Date', y_axis_label='Wins',
            toolbar_location=None)

# Render the race as step lines
west_fig.step('stDate', 'gameWon',
          source=west_cds, view=rockets_view,
          color='#CE1141', legend='Rockets')
west_fig.step('stDate', 'gameWon',
          source=west_cds, view=warriors_view,
          color='#006BB6', legend='Warriors')

# Move the legend to the upper left corner
west_fig.legend.location = 'top_left'

# Show the plot
show(west_fig)
```

Rockets vs. Warriors 2

Notice how the GroupFilter is passed to CDSView in a list. This allows you to combine multiple filters together to isolate the data you need from the ColumnDataSource as needed.

For information about integrating data sources, check out the Bokeh user guide's post on the ColumnDataSource and other source objects available.

The Western Conference ended up being an exciting race, but say you want to see if the Eastern Conference was just as tight. Not only that, but you'd like to view them in a single visualization. This is a perfect segue to the next topic: layouts.

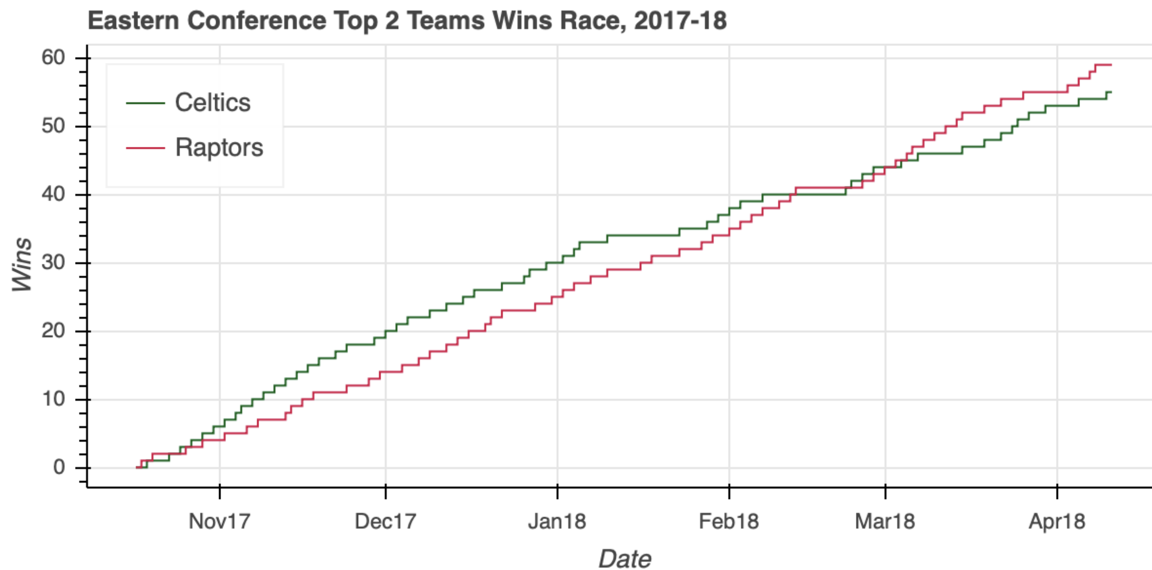
Day-05:Organizing Multiple Visualizations With Layouts

The Eastern Conference standings came down to two rivals in the Atlantic Division: the Boston Celtics and the Toronto Raptors. Before replicating the steps used to create `west_top_2`, let's try to put the `ColumnDataSource` to the test one more time using what you learned above.

In this example, you'll see how to feed an entire DataFrame into a `ColumnDataSource` and create views to isolate the relevant data:

```
# Bokeh libraries
from bokeh.plotting import figure, show
from bokeh.io import output_file
from bokeh.models import ColumnDataSource, CDSView, GroupFilter

# Output to file
output_file('east-top-2-standings-race.html',
            title='Eastern Conference Top 2 Teams Wins Race')

# Create a ColumnDataSource
standings_cds = ColumnDataSource(standings)

# Create views for each team
celtics_view = CDSView(source=standings_cds,
                      filters=[GroupFilter(column_name='teamAbbr',
                                           group='BOS')])
raptors_view = CDSView(source=standings_cds,
                      filters=[GroupFilter(column_name='teamAbbr',
                                           group='TOR')])

# Create and configure the figure
east_fig = figure(x_axis_type='datetime',
           plot_height=300, plot_width=600,
           title='Eastern Conference Top 2 Teams Wins Race, 2017-18',
           x_axis_label='Date', y_axis_label='Wins',
           toolbar_location=None)

# Render the race as step lines
east_fig.step('stDate', 'gameWon',
              color='#007A33', legend='Celtics',
              source=standings_cds, view=celtics_view)
east_fig.step('stDate', 'gameWon',
              color='#CE1141', legend='Raptors',
              source=standings_cds, view=raptors_view)

# Move the legend to the upper left corner
```

```
east_fig.legend.location = 'top_left'

# Show the plot
show(east_fig)
```



The `ColumnDataSource` was able to isolate the relevant data within a 5,040-by-39 `DataFrame` without breaking a sweat, saving a few lines of Pandas code in the process.

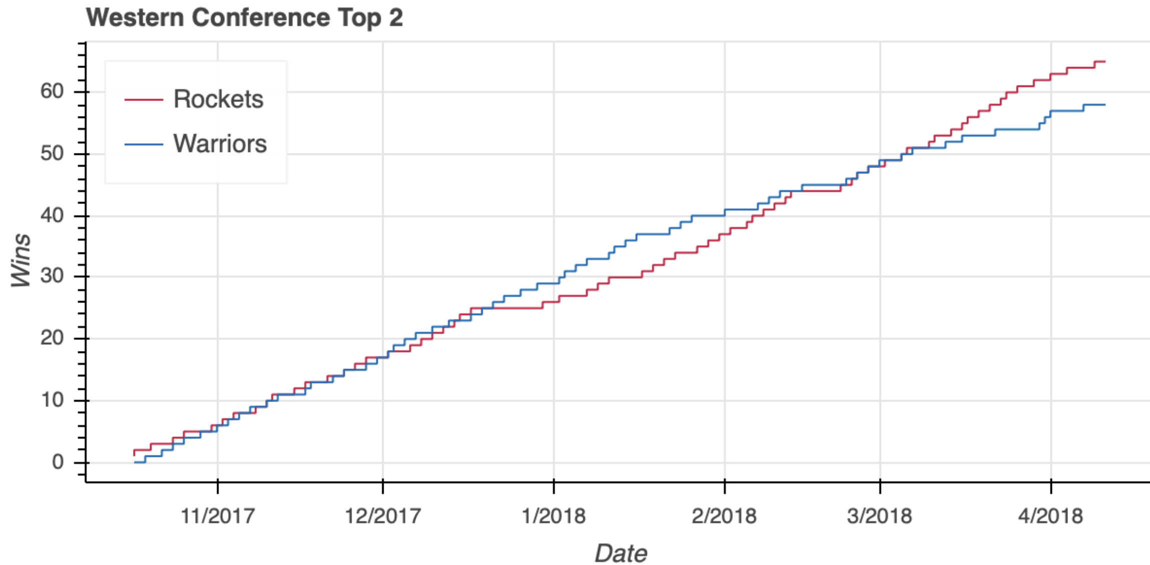
Looking at the visualization, you can see that the Eastern Conference race was no slouch. After the Celtics roared out of the gate, the Raptors clawed all the way back to overtake their division rival and finish the regular season with five more wins.

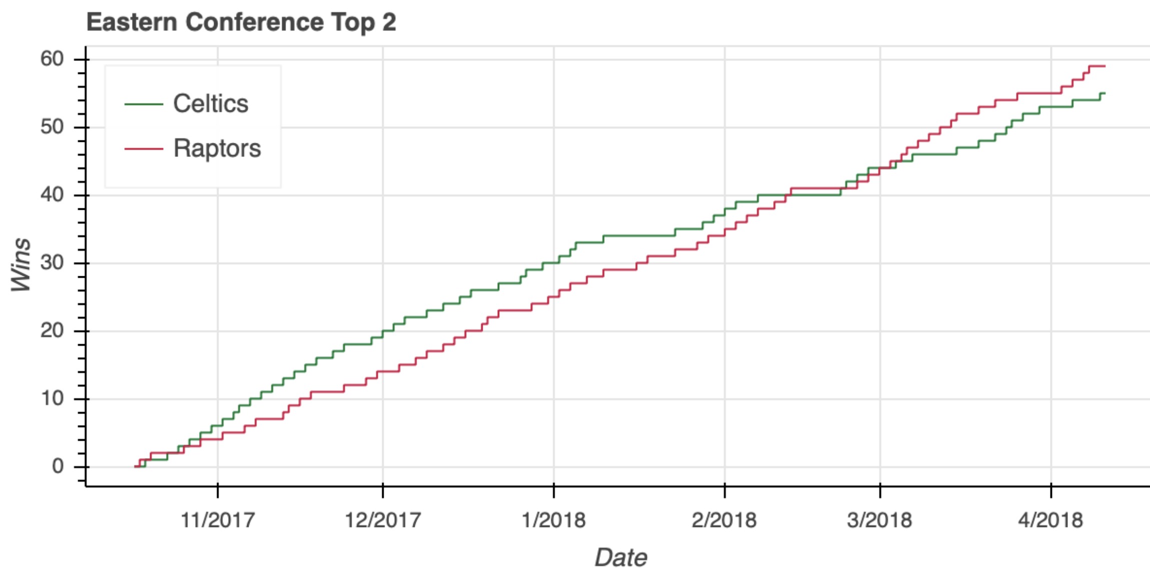
With our two visualizations ready, it's time to put them together.

Similar to the functionality of [Matplotlib's `subplot`], Bokeh offers the `column`, `row`, and `gridplot` functions in its `bokeh.layouts` module. These functions can more generally be classified as **layouts**.

The usage is very straightforward. If you want to put two visualizations in a vertical configuration, you can do so with the following:

```
# Bokeh library
from bokeh.plotting import figure, show
from bokeh.io import output_file
from bokeh.layouts import column

# Output to file
output_file('east-west-top-2-standings-race.html',
            title='Conference Top 2 Teams Wins Race')

# Plot the two visualizations in a vertical configuration
show(column(west_fig, east_fig))
```





I'll save you the two lines of code, but rest assured that swapping `column` for `row` in the snippet above will similarly configure the two plots in a horizontal configuration.

Note: If you're trying out the code snippets as you go through the tutorial, I want to take a quick detour to address an error you may see accessing `west_fig` and `east_fig` in the following examples. In doing so, you may receive an error like this:

```
WARNING:bokeh.core.validation.check:W-1004 (BOTH_CHILD_AND_ROOT): Models
should not be a document root...
```

This is one of many errors that are part of Bokeh's validation module, where `w-1004` in particular is warning about the re-use of `west_fig` and `east_fig` in a new layout.

To avoid this error as you test the examples, preface the code snippet illustrating each layout with the following:

```
# Bokeh libraries
```

```
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource, CDSView, GroupFilter

# Create a ColumnDataSource
standings_cds = ColumnDataSource(standings)

# Create the views for each team
celtics_view = CDSView(source=standings_cds,
                      filters=[GroupFilter(column_name='teamAbbr',
                                           group='BOS')])

raptors_view = CDSView(source=standings_cds,
                      filters=[GroupFilter(column_name='teamAbbr',
                                           group='TOR')])

rockets_view = CDSView(source=standings_cds,
                      filters=[GroupFilter(column_name='teamAbbr',
                                           group='HOU')])
warriors_view = CDSView(source=standings_cds,
                      filters=[GroupFilter(column_name='teamAbbr',
                                           group='GS')])

# Create and configure the figure
east_fig = figure(x_axis_type='datetime',
                  plot_height=300,
                  x_axis_label='Date',
                  y_axis_label='Wins',
                  toolbar_location=None)

west_fig = figure(x_axis_type='datetime',
                  plot_height=300,
                  x_axis_label='Date',
                  y_axis_label='Wins',
                  toolbar_location=None)

# Configure the figures for each conference
east_fig.step('stDate', 'gameWon',
              color='#007A33', legend='Celtics',
              source=standings_cds, view=celtics_view)
east_fig.step('stDate', 'gameWon',
              color='#CE1141', legend='Raptors',
              source=standings_cds, view=raptors_view)

west_fig.step('stDate', 'gameWon', color='#CE1141', legend='Rockets',
              source=standings_cds, view=rockets_view)
west_fig.step('stDate', 'gameWon', color='#006BB6', legend='Warriors',
              source=standings_cds, view=warriors_view)

# Move the legend to the upper left corner
east_fig.legend.location = 'top_left'
west_fig.legend.location = 'top_left'

# Layout code snippet goes here!
```

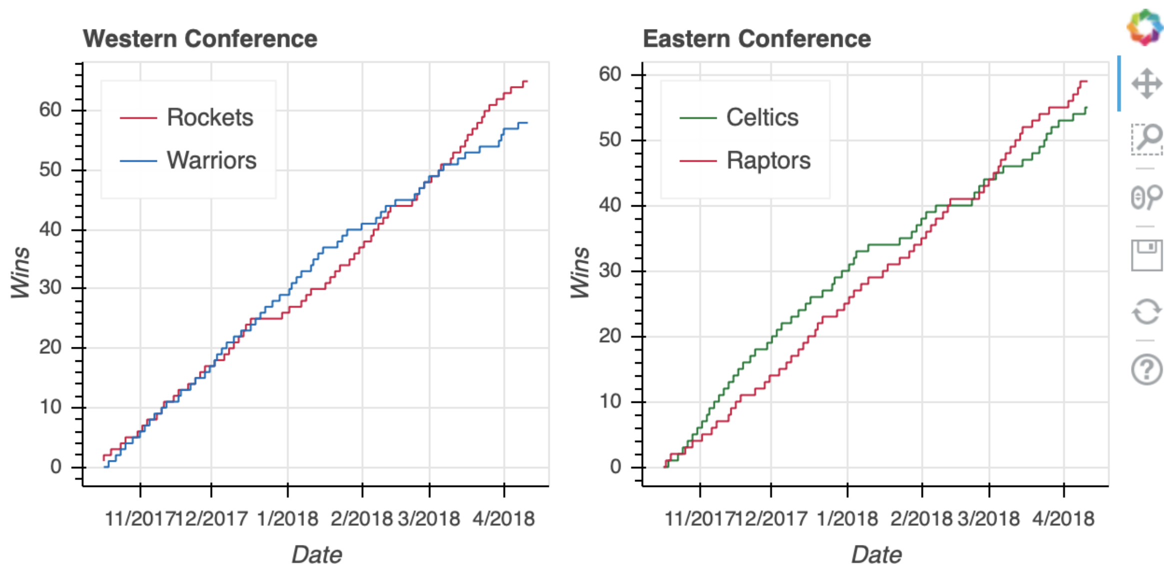
Doing so will renew the relevant components to render the visualization, ensuring that no warning is needed.

Instead of using `column` or `row`, you may want to use a `gridplot` instead.

One key difference of `gridplot` is that it will automatically consolidate the toolbar across all of its children figures. The two visualizations above do not have a toolbar, but if they did, then each figure would have its own when using `column` or `row`. With that, it also has its own `toolbar_location` property, seen below set to `'right'`.

Syntactically, you'll also notice below that `gridplot` differs in that, instead of being passed a tuple as input, it requires a list of lists, where each sub-list represents a row in the grid:

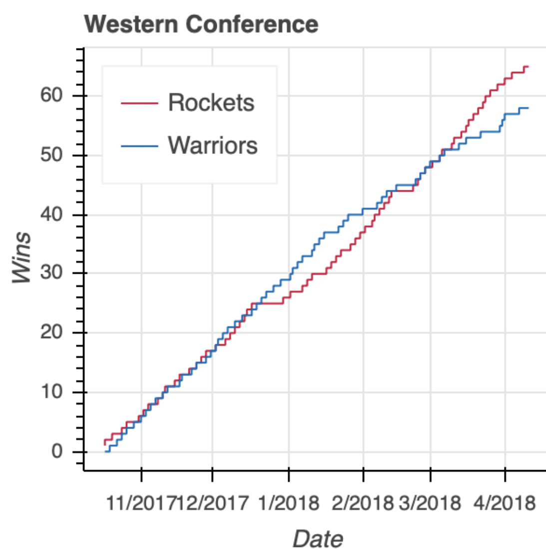
```
# Bokeh libraries
from bokeh.io import output_file
from bokeh.layouts import gridplot

# Output to file
output_file('east-west-top-2-gridplot.html',
            title='Conference Top 2 Teams Wins Race')

# Reduce the width of both figures
east_fig.plot_width = west_fig.plot_width = 300

# Edit the titles
east_fig.title.text = 'Eastern Conference'
west_fig.title.text = 'Western Conference'

# Configure the gridplot
east_west_gridplot = gridplot([[west_fig, east_fig]],
                              toolbar_location='right')

# Plot the two visualizations in a horizontal configuration
show(east_west_gridplot)
```

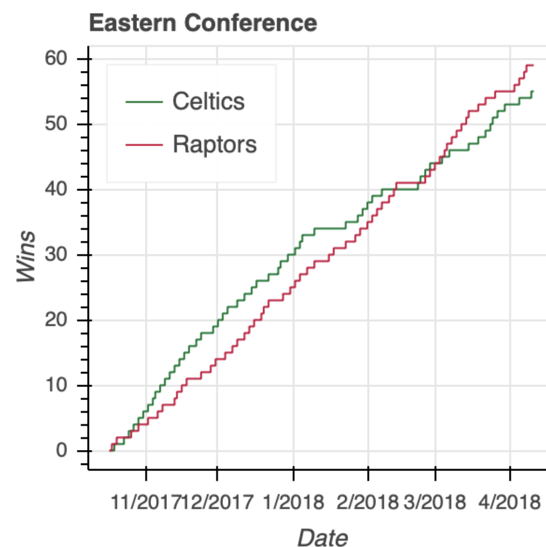


Lastly, `gridplot` allows the passing of `None` values, which are interpreted as blank subplots. Therefore, if you wanted to leave a placeholder for two additional plots, then you could do something like this:

```
# Bokeh libraries
from bokeh.io import output_file
from bokeh.layouts import gridplot
```

```
# Output to file
output_file('east-west-top-2-gridplot.html',
            title='Conference Top 2 Teams Wins Race')

# Reduce the width of both figures
east_fig.plot_width = west_fig.plot_width = 300

# Edit the titles
east_fig.title.text = 'Eastern Conference'
west_fig.title.text = 'Western Conference'

# Plot the two visualizations with placeholders
east_west_gridplot = gridplot([[west_fig, None], [None, east_fig]],
                              toolbar_location='right')

# Plot the two visualizations in a horizontal configuration
show(east_west_gridplot)
```

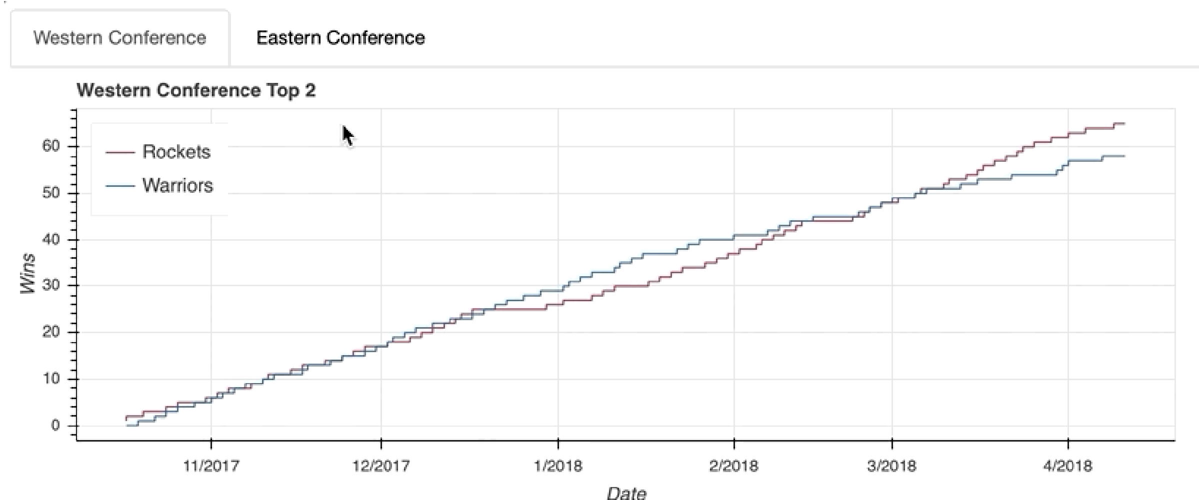




If you'd rather toggle between both visualizations at their full size without having to squash them down to fit next to or on top of each other, a good option is a tabbed layout.

A tabbed layout consists of two Bokeh widget functions: `Tab()` and `Panel()` from the `bokeh.models.widgets` sub-module. Like using `gridplot()`, making a tabbed layout is pretty straightforward:

```
# Bokeh Library
from bokeh.io import output_file
from bokeh.models.widgets import Tabs, Panel

# Output to file
output_file('east-west-top-2-tabbed_layout.html',
            title='Conference Top 2 Teams Wins Race')

# Increase the plot widths
east_fig.plot_width = west_fig.plot_width = 800

# Create two panels, one for each conference
east_panel = Panel(child=east_fig, title='Eastern Conference')
west_panel = Panel(child=west_fig, title='Western Conference')

# Assign the panels to Tabs
tabs = Tabs(tabs=[west_panel, east_panel])

# Show the tabbed layout
show(tabs)
```



The first step is to create a `Panel()` for each tab. That may sound a little confusing, but think of the `Tabs()` function as the mechanism that organizes the individual tabs created with `Panel()`.

Each `Panel()` takes as input a child, which can either be a single `figure()` or a layout. (Remember that a layout is a general name for a `column`, `row`, or `gridplot`.) Once your panels are assembled, they can be passed as input to `Tabs()` in a list.

Now that you understand how to access, draw, and organize your data, it's time to move on to the real magic of Bokeh: interaction! As always, check out Bokeh's User Guide for more information on layouts.

Adding Interaction

The feature that sets Bokeh apart is its ability to easily implement interactivity in your visualization. Bokeh even goes as far as describing itself as an **interactive visualization library**:

Bokeh is an interactive visualization library that targets modern web browsers for presentation. (Source)

In this section, we'll touch on five ways that you can add interactivity:

- Configuring the toolbar
- Selecting data points
- Adding hover actions
- Linking axes and selections
- Highlighting data using the legend

Implementing these interactive elements open up possibilities for exploring your data that static visualizations just can't do by themselves.

Configuring the Toolbar

As you saw all the way back in Generating Your First Figure, the default Bokeh `figure()` comes with a toolbar right out of the box. The default toolbar comes with the following tools (from left to right):

- Pan
- Box Zoom
- Wheel Zoom
- Save
- Reset
- A link to Bokeh's user guide for Configuring Plot Tools
- A link to the Bokeh homepage

The toolbar can be removed by passing `toolbar_location=None` when instantiating a `figure()` object, or relocated by passing any of `'above'`, `'below'`, `'left'`, or `'right'`.

Additionally, the toolbar can be configured to include any combination of tools you desire. Bokeh offers 18 specific tools across five categories:

- Pan/Drag: `box_select`, `box_zoom`, `lasso_select`, `pan`, `xpan`, `ypan`, `resize_select`
- Click/Tap: `poly_select`, `tap`
- Scroll/Pinch: `wheel_zoom`, `xwheel_zoom`, `ywheel_zoom`
- Actions: `undo`, `redo`, `reset`, `save`
- Inspectors: `crosshair`, `hover`

To geek out on tools , make sure to visit Specifying Tools. Otherwise, they'll be illustrated in covering the various interactions covered herein.

Selecting Data Points

Implementing selection behavior is as easy as adding a few specific keywords when declaring your glyphs.

The next example will create a scatter plot that relates a player's total number of three-point shot attempts to the percentage made (for players with at least 100 three-point shot attempts).

The data can be aggregated from the `player_stats` DataFrame:

```
# Find players who took at least 1 three-point shot during the season
three_takers = player_stats[player_stats['play3PA'] > 0]

# Clean up the player names, placing them in a single column
three_takers['name'] = [f'{p["playFNm"]} {p["playLNm"]}'
                        for _, p in three_takers.iterrows()]

# Aggregate the total three-point attempts and makes for each player
three_takers = (three_takers.groupby('name')
                            .sum()
                            .loc[:,['play3PA', 'play3PM']]
                            .sort_values('play3PA', ascending=False))

# Filter out anyone who didn't take at least 100 three-point shots
three_takers = three_takers[three_takers['play3PA'] >= 100].reset_index()

# Add a column with a calculated three-point percentage (made/attempted)
three_takers['pct3PM'] = three_takers['play3PM'] / three_takers['play3PA']
```

Here's a sample of the resulting `DataFrame`:

```
>>> three_takers.sample(5)
                   name  play3PA  play3PM    pct3PM
229        Corey Brewer      110       31  0.281818
78           Marc Gasol      320      109  0.340625
126      Raymond Felton      230       81  0.352174
127  Kristaps Porziņģis      229       90  0.393013
66      Josh Richardson      336      127  0.377976
```

Let's say you want to select a groups of players in the distribution, and in doing so mute the color of the glyphs representing the non-selected players:

```
# Bokeh Libraries
from bokeh.plotting import figure, show
from bokeh.io import output_file
from bokeh.models import ColumnDataSource, NumeralTickFormatter

# Output to file
output_file('three-point-att-vs-pct.html',
            title='Three-Point Attempts vs. Percentage')

# Store the data in a ColumnDataSource
three_takers_cds = ColumnDataSource(three_takers)

# Specify the selection tools to be made available
select_tools = ['box_select', 'lasso_select', 'poly_select', 'tap',
'reset']
```

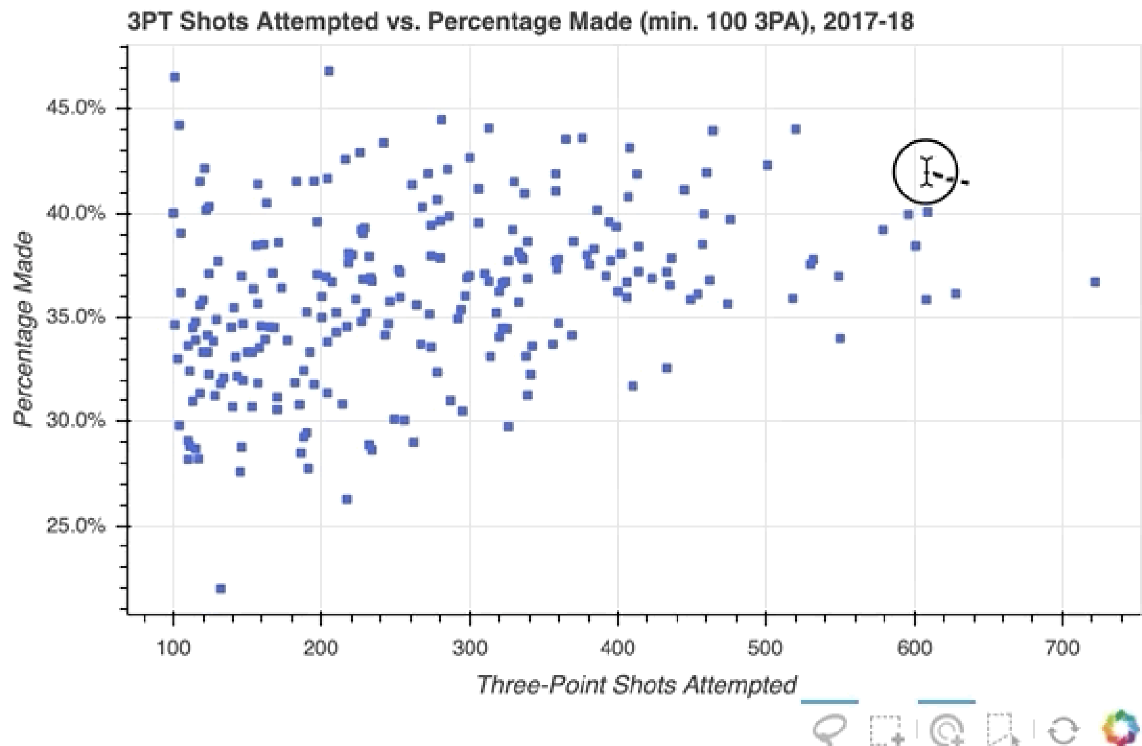
```
# Create the figure
fig = figure(plot_height=400,
             plot_width=600,
             x_axis_label='Three-Point Shots Attempted',
             y_axis_label='Percentage Made',
             title='3PT Shots Attempted vs. Percentage Made (min. 100 3PA),
2017-18',
             toolbar_location='below',
             tools=select_tools)

# Format the y-axis tick labels as percentages
fig.yaxis[0].formatter = NumeralTickFormatter(format='00.0%')

# Add square representing each player
fig.square(x='play3PA',
           y='pct3PM',
           source=three_takers_cds,
           color='royalblue',
           selection_color='deepskyblue',
           nonselection_color='lightgray',
           nonselection_alpha=0.3)

# Visualize
show(fig)
```

First, specify the selection tools you want to make available. In the example above, `'box_select'`, `'lasso_select'`, `'poly_select'`, and `'tap'` (plus a reset button) were specified in a list called `select_tools`. When the figure is instantiated, the toolbar is positioned `'below'` the plot, and the list is passed to `tools` to make the tools selected above available.

Each player is initially represented by a royal blue square glyph, but the following configurations are set for when a player or group of players is selected:

- Turn the selected player(s) to `deepskyblue`
- Change all non-selected players' glyphs to a `lightgray` color with `0.3` opacity

That's it! With just a few quick additions, the visualization now looks like this:



For even more information about what you can do upon selection, check out Selected and Unselected Glyphs.

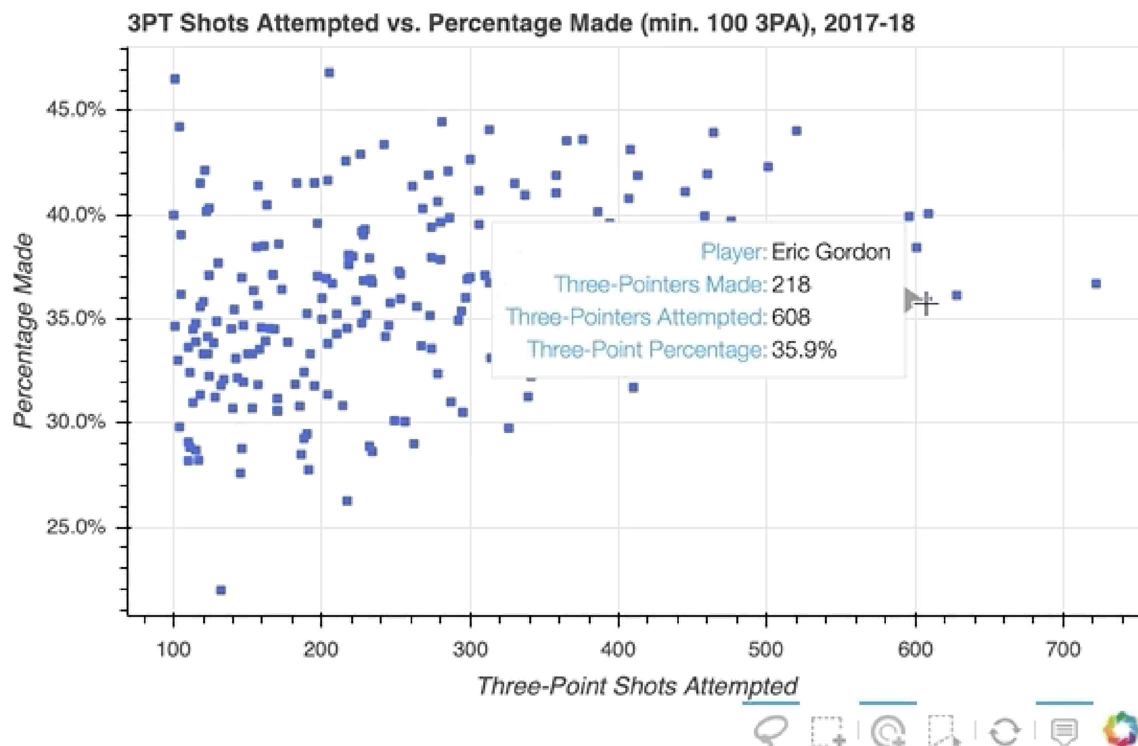
Adding Hover Actions

So the ability to select specific player data points that seem of interest in my scatter plot is implemented, but what if you want to quickly see what individual players a glyph represents? One option is to use Bokeh's `HoverTool()` to show a tooltip when the cursor crosses paths with a glyph. All you need to do is append the following to the code snippet above:

```
# Bokeh Library
from bokeh.models import HoverTool

# Format the tooltip
tooltips = [
             ('Player','@name'),
             ('Three-Pointers Made', '@play3PM'),
             ('Three-Pointers Attempted', '@play3PA'),
             ('Three-Point Percentage','@pct3PM{00.0%}'),
            ]

# Add the HoverTool to the figure
fig.add_tools(HoverTool(tooltips=tooltips))

# Visualize
show(fig)
```

The `HoverTool()` is slightly different than the selection tools you saw above in that it has properties, specifically `tooltips`.

First, you can configure a formatted tooltip by creating a list of tuples containing a description and reference to the `ColumnDataSource`. This list was passed as input to the `HoverTool()` and then simply added to the figure using `add_tools()`. Here's what happened:



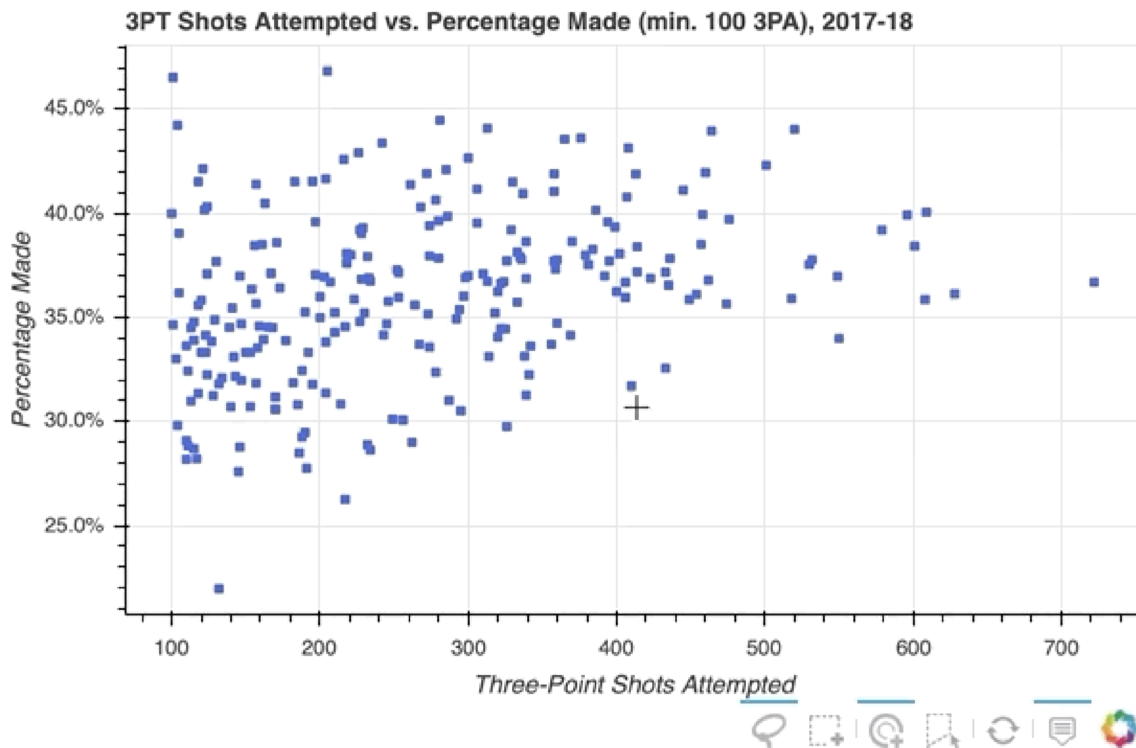
Notice the addition of the *Hover* button to the toolbar, which can be toggled on and off.

If you want to even further emphasize the players on hover, Bokeh makes that possible with hover inspections. Here is a slightly modified version of the code snippet that added the tooltip:

```
# Format the tooltip
tooltips = [
            ('Player','@name'),
            ('Three-Pointers Made', '@play3PM'),
            ('Three-Pointers Attempted', '@play3PA'),
            ('Three-Point Percentage','@pct3PM{00.0%}'),
           ]

# Configure a renderer to be used upon hover
hover_glyph = fig.circle(x='play3PA', y='pct3PM', source=three_takers_cds,
                         size=15, alpha=0,
                         hover_fill_color='black', hover_alpha=0.5)

# Add the HoverTool to the figure
fig.add_tools(HoverTool(tooltips=tooltips, renderers=[hover_glyph]))

# Visualize
show(fig)
```

This is done by creating a completely new glyph, in this case circles instead of squares, and assigning it to `hover_glyph`. Note that the initial opacity is set to zero so that it is invisible until the cursor is touching it. The properties that appear upon hover are captured by setting `hover_alpha` to `0.5` along with the `hover_fill_color`.

Now you will see a small black circle appear over the original square when hovering over the various markers:



To further explore the capabilities of the `HoverTool()`, see the [HoverTool](#) and [Hover Inspections](#) guides.

Linking Axes and Selections

Linking is the process of syncing elements of different visualizations within a layout. For instance, maybe you want to link the axes of multiple plots to ensure that if you zoom in on one it is reflected on another. Let’s see how it is done.

For this example, the visualization will be able to pan to different segments of a team’s schedule and examine various game stats. Each stat will be represented by its own plot in a two-by-two `gridplot()` .

The data can be collected from the `team_stats` DataFrame, selecting the Philadelphia 76ers as the team of interest:

```
# Isolate relevant data
phi_gm_stats = (team_stats[(team_stats['teamAbbr'] == 'PHI') &
                           (team_stats['seasTyp'] == 'Regular')]
                 .loc[:, ['gmDate',
                          'teamPTS',
```

```
                         'teamTRB',
                         'teamAST',
                         'teamTO',
                         'opptPTS',]]
                .sort_values('gmDate'))

# Add game number
phi_gm_stats['game_num'] = range(1, len(phi_gm_stats)+1)

# Derive a win_loss column
win_loss = []
for _, row in phi_gm_stats.iterrows():

    # If the 76ers score more points, it's a win
    if row['teamPTS'] > row['opptPTS']:
        win_loss.append('W')
    else:
        win_loss.append('L')

# Add the win_loss data to the DataFrame
phi_gm_stats['winLoss'] = win_loss
```

Here are the results of the 76ers' first 5 games:

```
>>> phi_gm_stats.head()
        gmDate  teamPTS  teamTRB  teamAST  teamTO  opptPTS  game_num
winLoss
10  2017-10-18      115       48       25      17      120         1
L
39  2017-10-20       92       47       20      17      102         2
L
52  2017-10-21       94       41       18      20      128         3
L
80  2017-10-23       97       49       25      21       86         4
W
113 2017-10-25      104       43       29      16      105         5
L
```

Start by importing the necessary Bokeh libraries, specifying the output parameters, and reading the data into a `ColumnDataSource`:

```
# Bokeh Libraries
from bokeh.plotting import figure, show
from bokeh.io import output_file
from bokeh.models import ColumnDataSource, CategoricalColorMapper, Div
from bokeh.layouts import gridplot, column

# Output to file
output_file('phi-gm-linked-stats.html',
                title='76ers Game Log')

# Store the data in a ColumnDataSource
gm_stats_cds = ColumnDataSource(phi_gm_stats)
```

Each game is represented by a column, and will be colored green if the result was a win and red for a loss. To accomplish this, Bokeh's `CategoricalColorMapper` can be used to map the data values to specified colors:

```
# Create a CategoricalColorMapper that assigns a color to wins and losses
win_loss_mapper = CategoricalColorMapper(factors = ['W', 'L'],
                                         palette=['green', 'red'])
```

For this use case, a list specifying the categorical data values to be mapped is passed to `factors` and a list with the intended colors to `palette`. For more on the `CategoricalColorMapper`, see the Colors section of Handling Categorical Data on Bokeh's User Guide.

There are four stats to visualize in the two-by-two `gridplot`: points, assists, rebounds, and turnovers. In creating the four figures and configuring their respective charts, there is a lot of redundancy in the properties. So to streamline the code a `for` loop can be used:

```
# Create a dict with the stat name and its corresponding column in the data
stat_names = {'Points': 'teamPTS',
              'Assists': 'teamAST',
              'Rebounds': 'teamTRB',
              'Turnovers': 'teamTO',}

# The figure for each stat will be held in this dict
stat_figs = {}

# For each stat in the dict
for stat_label, stat_col in stat_names.items():

    # Create a figure
    fig = figure(y_axis_label=stat_label,
                 plot_height=200, plot_width=400,
                 x_range=(1, 10), tools=['xpan', 'reset', 'save'])

    # Configure vbar
    fig.vbar(x='game_num', top=stat_col, source=gm_stats_cds, width=0.9,
             color=dict(field='winLoss', transform=win_loss_mapper))

    # Add the figure to stat_figs dict
    stat_figs[stat_label] = fig
```

As you can see, the only parameters that needed to be adjusted were the `y-axis-label` of the figure and the data that will dictate `top` in the `vbar`. These values were easily stored in a `dict` that was iterated through to create the figures for each stat.

You can also see the implementation of the `CategoricalColorMapper` in the configuration of the `vbar` glyph. The `color` property is passed a `dict` with the field in the `ColumnDataSource` to be mapped and the name of the `CategoricalColorMapper` created above.

The initial view will only show the first 10 games of the 76ers' season, so there needs to be a way to pan horizontally to navigate through the rest of the games in the season. Thus configuring the toolbar to have an `xpan` tool allows panning throughout the plot without having to worry about accidentally skewing the view along the vertical axis.

Now that the figures are created, `gridplot` can be setup by referencing the figures from the `dict` created above:

```
# Create layout
grid = gridplot([[stat_figs['Points'], stat_figs['Assists']],
                 [stat_figs['Rebounds'], stat_figs['Turnovers']]])
```

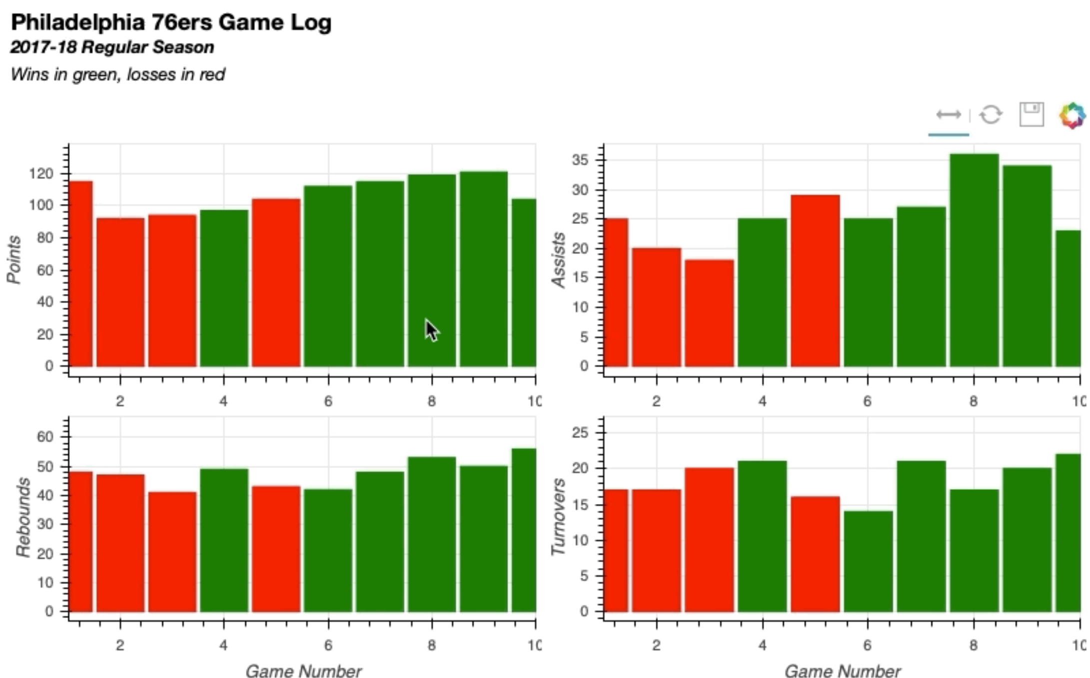
Linking the axes of the four plots is as simple as setting the `x_range` of each figure equal to one another:

```
# Link together the x-axes
stat_figs['Points'].x_range = \
    stat_figs['Assists'].x_range = \
    stat_figs['Rebounds'].x_range = \
    stat_figs['Turnovers'].x_range
```

To add a title bar to the visualization, you could have tried to do this on the points figure, but it would have been limited to the space of that figure. Therefore, a nice trick is to use Bokeh's ability to interpret HTML to insert a `Div` element that contains the title information. Once that is created, simply combine that with the `gridplot()` in a `column` layout:

```
# Add a title for the entire visualization using Div
html = """<h3>Philadelphia 76ers Game Log</h3>
<b><i>2017-18 Regular Season</i>
<br>
</b><i>Wins in green, losses in red</i>
"""
sup_title = Div(text=html)

# Visualize
show(column(sup_title, grid))
```

Putting all the pieces together results in the following:



Similarly you can easily implement linked selections, where a selection on one plot will be reflected on others.

To see how this works, the next visualization will contain two scatter plots: one that shows the 76ers' two-point versus three-point field goal percentage and the other showing the 76ers' team points versus opponent points on a game-by-game basis.

The goal is to be able to select data points on the left-side scatter plot and quickly be able to recognize if the corresponding datapoint on the right scatter plot is a win or loss.

The DataFrame for this visualization is very similar to that from the first example:

```
# Isolate relevant data
phi_gm_stats_2 = (team_stats[(team_stats['teamAbbr'] == 'PHI') &
                             (team_stats['seasTyp'] == 'Regular')]
                  .loc[:, ['gmDate',
                           'team2P%',
                           'team3P%',
                           'teamPTS',
                           'opptPTS']]
                  .sort_values('gmDate'))

# Add game number
phi_gm_stats_2['game_num'] = range(1, len(phi_gm_stats_2) + 1)

# Derive a win_loss column
win_loss = []
for _, row in phi_gm_stats_2.iterrows():

    # If the 76ers score more points, it's a win
    if row['teamPTS'] > row['opptPTS']:
        win_loss.append('W')
    else:
        win_loss.append('L')

# Add the win_loss data to the DataFrame
phi_gm_stats_2['winLoss'] = win_loss
```

Here's what the data looks like:

```
>>> phi_gm_stats_2.head()
        gmDate  team2P%  team3P%  teamPTS  opptPTS  game_num winLoss
10  2017-10-18   0.4746   0.4286      115      120         1       L
39  2017-10-20   0.4167   0.3125       92      102         2       L
52  2017-10-21   0.4138   0.3333       94      128         3       L
80  2017-10-23   0.5098   0.3750       97       86         4       W
113 2017-10-25   0.5082   0.3333      104      105         5       L
```

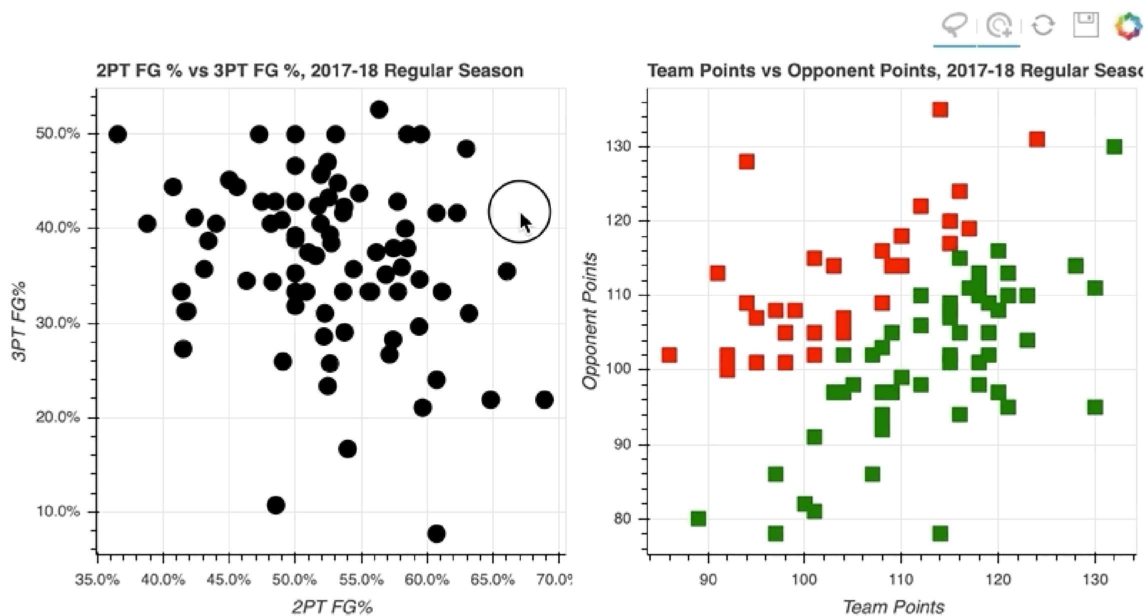
The code to create the visualization is as follows:

```
# Bokeh Libraries
from bokeh.plotting import figure, show
from bokeh.io import output_file
from bokeh.models import ColumnDataSource, CategoricalColorMapper,
NumeralTickFormatter
from bokeh.layouts import gridplot

# Output inline in the notebook
output_file('phi-gm-linked-selections.html',
            title='76ers Percentages vs. Win-Loss')
```

```
# Store the data in a ColumnDataSource
gm_stats_cds = ColumnDataSource(phi_gm_stats_2)

# Create a CategoricalColorMapper that assigns specific colors to wins and
losses
win_loss_mapper = CategoricalColorMapper(factors = ['W', 'L'],
palette=['Green', 'Red'])

# Specify the tools
toolList = ['lasso_select', 'tap', 'reset', 'save']

# Create a figure relating the percentages
pctFig = figure(title='2PT FG % vs 3PT FG %, 2017-18 Regular Season',
                plot_height=400, plot_width=400, tools=toolList,
                x_axis_label='2PT FG%', y_axis_label='3PT FG%')

# Draw with circle markers
pctFig.circle(x='team2P%', y='team3P%', source=gm_stats_cds,
              size=12, color='black')

# Format the y-axis tick labels as percenages
pctFig.xaxis[0].formatter = NumeralTickFormatter(format='00.0%')
pctFig.yaxis[0].formatter = NumeralTickFormatter(format='00.0%')

# Create a figure relating the totals
totFig = figure(title='Team Points vs Opponent Points, 2017-18 Regular
Season',
                plot_height=400, plot_width=400, tools=toolList,
                x_axis_label='Team Points', y_axis_label='Opponent Points')

# Draw with square markers
totFig.square(x='teamPTS', y='opptPTS', source=gm_stats_cds, size=10,
              color=dict(field='winLoss', transform=win_loss_mapper))

# Create layout
grid = gridplot([[pctFig, totFig]])

# Visualize
show(grid)
```

This is a great illustration of the power in using a `ColumnDataSource`. As long as the glyph renderers (in this case, the `circle` glyphs for the percentages, and `square` glyphs for the wins and losses) share the same `ColumnDataSource`, then the selections will be linked by default.

Here’s how it looks in action, where you can see selections made on either figure will be reflected on the other:



By selecting a random sample of data points in the upper right quadrant of the left scatter plot, those corresponding to both high two-point and three-point field goal percentage, the data points on the right scatter plot are highlighted.

Similarly, selecting data points on the right scatter plot that correspond to losses tend to be further to the lower left, lower shooting percentages, on the left scatter plot.

All the details on linking plots can be found at [Linking Plots] in the Bokeh User Guide.

Highlighting Data Using the Legend

That brings us to the final interactivity example in this tutorial: interactive legends.

In the [Drawing Data With Glyphs] section, you saw how easy it is to implement a legend when creating your plot. With the legend in place, adding interactivity is merely a matter of assigning a `click_policy`. Using a single line of code, you can quickly add the ability to either `hide` or `mute` data using the legend.

In this example, you'll see two identical scatter plots comparing the game-by-game points and rebounds of LeBron James and Kevin Durant. The only difference will be that one will use a `hide` as its `click_policy`, while the other uses `mute`.

The first step is to configure the output and set up the data, creating a view for each player from the `player_stats` DataFrame:

```
# Bokeh Libraries
from bokeh.plotting import figure, show
from bokeh.io import output_file
from bokeh.models import ColumnDataSource, CDSView, GroupFilter
from bokeh.layouts import row

# Output inline in the notebook
output_file('lebron-vs-durant.html',
```

```
            title='LeBron James vs. Kevin Durant')

# Store the data in a ColumnDataSource
player_gm_stats = ColumnDataSource(player_stats)

# Create a view for each player
lebron_filters = [GroupFilter(column_name='playFNm', group='LeBron'),
                  GroupFilter(column_name='playLNm', group='James')]
lebron_view = CDSView(source=player_gm_stats,
                      filters=lebron_filters)

durant_filters = [GroupFilter(column_name='playFNm', group='Kevin'),
                  GroupFilter(column_name='playLNm', group='Durant')]
durant_view = CDSView(source=player_gm_stats,
                      filters=durant_filters)
```

Before creating the figures, the common parameters across the figure, markers, and data can be consolidated into dictionaries and reused. Not only does this save redundancy in the next step, but it provides an easy way to tweak these parameters later if need be:

```
# Consolidate the common keyword arguments in dicts
common_figure_kwargs = {
    'plot_width': 400,
    'x_axis_label': 'Points',
    'toolbar_location': None,
}
common_circle_kwargs = {
    'x': 'playPTS',
    'y': 'playTRB',
    'source': player_gm_stats,
    'size': 12,
    'alpha': 0.7,
}
common_lebron_kwargs = {
    'view': lebron_view,
    'color': '#002859',
    'legend': 'LeBron James'
}
common_durant_kwargs = {
    'view': durant_view,
    'color': '#FFC324',
    'legend': 'Kevin Durant'
}
```

Now that the various properties are set, the two scatter plots can be built in a much more concise fashion:

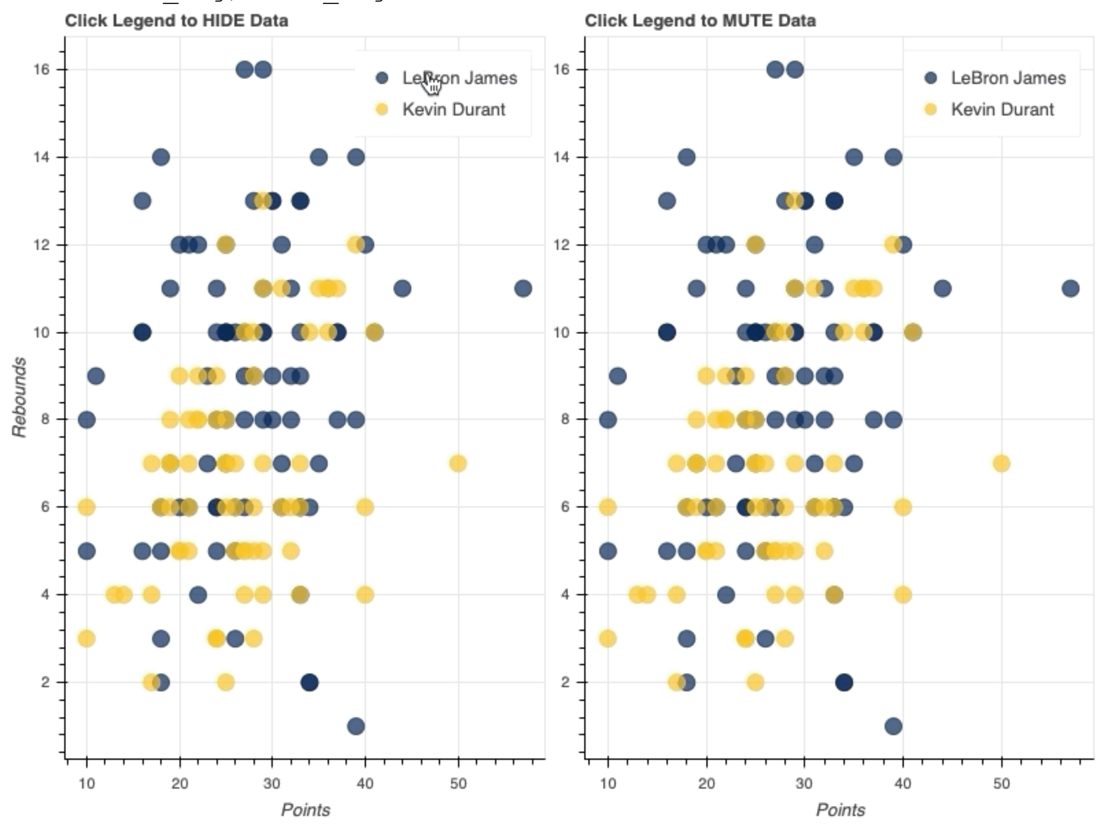
```
# Create the two figures and draw the data
hide_fig = figure(**common_figure_kwargs,
                  title='Click Legend to HIDE Data',
                  y_axis_label='Rebounds')
hide_fig.circle(**common_circle_kwargs, **common_lebron_kwargs)
hide_fig.circle(**common_circle_kwargs, **common_durant_kwargs)

mute_fig = figure(**common_figure_kwargs, title='Click Legend to MUTE
Data')
mute_fig.circle(**common_circle_kwargs, **common_lebron_kwargs,
                muted_alpha=0.1)
mute_fig.circle(**common_circle_kwargs, **common_durant_kwargs,
```

```
                muted_alpha=0.1)
```

Note that `mute_fig` has an extra parameter called `muted_alpha`. This parameter controls the opacity of the markers when `mute` is used as the `click_policy`.

Finally, the `click_policy` for each figure is set, and they are shown in a horizontal configuration:

```
# Add interactivity to the legend
hide_fig.legend.click_policy = 'hide'
mute_fig.legend.click_policy = 'mute'

# Visualize
show(row(hide_fig, mute_fig))
```



Once the legend is in place, all you have to do is assign either `hide` or `mute` to the figure's `click_policy` property. This will automatically turn your basic legend into an interactive legend.

Also note that, specifically for `mute`, the additional property of `muted_alpha` was set in the respective `circle` glyphs for LeBron James and Kevin Durant. This dictates the visual effect driven by the legend interaction.

For more on all things interaction in Bokeh, Adding Interactions in the Bokeh User Guide is a great place to start.

Week 8: Text analysis and sentiment analysis

Day- 01 & 02: NLTK libaray for text analysis

Natural language processing (NLP) is a field that focuses on making natural human language usable by computer programs. NLTK, or Natural Language Toolkit, is a Python package that you can use for NLP.

A lot of the data that you could be analyzing is unstructured data and contains human-readable text. Before you can analyze that data programmatically, you first need to preprocess it. In this tutorial, you'll take your first look at the kinds of text preprocessing tasks you can do with NLTK so that you'll be ready to apply them in future projects. You'll also see how to do some basic text analysis and create visualizations.

Steps:

Find text to analyze

Preprocess your text for analysis

Analyze your text

The first thing you need to do is make sure that you have Python installed. For this tutorial, you'll be using Python 3.9. If you don't yet have Python installed, then check out Python 3 Installation & Setup Guide to get started. In shell type following commamd

$ python -m pip install nltk==3.5

python -m pip install numpy matplotlib

Once you have that dealt with, your next step is to install NLTK with pip. It's a best practice to install it in a virtual environment. To learn more about virtual environments, check out Python Virtual Environments: A Primer.

Tokenizing

By tokenizing, you can conveniently split up text by word or by sentence. This will allow you to work with smaller pieces of text that are still relatively coherent and meaningful even outside of the context of the rest of the text. It's your first step in turning unstructured data into structured data, which is easier to analyze.

When you're analyzing text, you'll be tokenizing by word and tokenizing by sentence. Here's what both types of tokenization bring to the table:

Tokenizing by word: Words are like the atoms of natural language. They're the smallest unit of meaning that still makes sense on its own. Tokenizing your text by word allows you to identify words that come up particularly often. For example, if you were analyzing a group of job ads, then you might find that the word "Python" comes up often. That could suggest high demand for Python knowledge, but you'd need to look deeper to know more.

Tokenizing by sentence: When you tokenize by sentence, you can analyze how those words relate to one another and see more context. Are there a lot of negative words around the word "Python" because the hiring manager doesn't like Python? Are there more terms from the domain of herpetology than the domain of software development, suggesting that you may be dealing with an entirely different kind of python than you were expecting?

from nltk.tokenize import sent_tokenize, word_tokenize

You can use sent_tokenize() to split up example_string into sentences:

```
>>> sent_tokenize(example_string)

>>> word_tokenize(example_string)
```

Filtering Stop Words

Stop words are words that you want to ignore, so you filter them out of your text when you're processing it. Very common words like 'in', 'is', and 'an' are often used as stop words since they don't add a lot of meaning to a text in and of themselves.

Here's how to import the relevant parts of NLTK in order to filter out stop words:

```
>>> nltk.download("stopwords")

>>> from nltk.corpus import stopwords

>>> from nltk.tokenize import word_tokenize

>>> worf_quote = "Sir, I protest. I am not a merry man!"

>>> words_in_quote = word_tokenize(worf_quote)

>>> words_in_quote

['Sir', ',', 'protest', '.', 'merry', 'man', '!']
```

You have a list of the words in worf_quote, so the next step is to create a set of stop words to filter words_in_quote. For this example, you'll need to focus on stop words in "english":

```
>>> stop_words = set(stopwords.words("english"))

>>> stop_words = set(stopwords.words("english"))
```

Next, create an empty list to hold the words that make it past the filter:

```
>>>

>>> filtered_list = []
```

You created an empty list, filtered_list, to hold all the words in words_in_quote that aren't stop words. Now you can use stop_words to filter words_in_quote:

```
>>>

>>> for word in words_in_quote:

...    if word.casefold() not in stop_words:

...        filtered_list.append(word)
```

You iterated over words_in_quote with a for loop and added all the words that weren't stop words to filtered_list. You used .casefold() on word so you could ignore whether the letters in word were uppercase or lowercase. This is worth doing because stopwords.words('english') includes only lowercase versions of stop words.

Alternatively, you could use a list comprehension to make a list of all the words in your text that aren't stop words:

```
>>>

>>> filtered_list = [

...     word for word in words_in_quote if word.casefold() not in stop_words

... ]
```

When you use a list comprehension, you don't create an empty list and then add items to the end of it. Instead, you define the list and its contents at the same time. Using a list comprehension is often seen as more Pythonic.

Take a look at the words that ended up in filtered_list:

```
>>>

>>> filtered_list

['Sir', ',', 'protest', '.', 'merry', 'man', '!']
```

You filtered out a few words like 'am' and 'a', but you also filtered out 'not', which does affect the overall meaning of the sentence. (Worf won't be happy about this.)

Words like 'I' and 'not' may seem too important to filter out, and depending on what kind of analysis you want to do, they can be. Here's why:

'I' is a pronoun, which are context words rather than content words:

Content words give you information about the topics covered in the text or the sentiment that the author has about those topics.

Context words give you information about writing style. You can observe patterns in how authors use context words in order to quantify their writing style. Once you've quantified their writing style, you can analyze a text written by an unknown author to see how closely it follows a particular writing style so you can try to identify who the author is.

'not' is technically an adverb but has still been included in NLTK's list of stop words for English. If you want to edit the list of stop words to exclude 'not' or make other changes, then you can download it.

So, 'I' and 'not' can be important parts of a sentence, but it depends on what you're trying to learn from that sentence.

Stemming

Stemming is a text processing task in which you reduce words to their root, which is the core part of a word. For example, the words "helping" and "helper" share the root "help." Stemming allows you to zero in on the basic meaning of a word rather than all the details of how it's being used. NLTK has more than one stemmer, but you'll be using the Porter stemmer.

Here's how to import the relevant parts of NLTK in order to start stemming:

```
>>>

>>> from nltk.stem import PorterStemmer
```

```
>>> from nltk.tokenize import word_tokenize
```

Now that you're done importing, you can create a stemmer with PorterStemmer():

```
>>>
>>> stemmer = PorterStemmer()
```

The next step is for you to create a string to stem. Here's one you can use:

```
>>>
>>> string_for_stemming = """
... The crew of the USS Discovery discovered many discoveries.
... Discovering is what explorers do."""
```

Before you can stem the words in that string, you need to separate all the words in it:

```
>>>
>>> words = word_tokenize(string_for_stemming)
```

Now that you have a list of all the tokenized words from the string, take a look at what's in words:

```
>>>
>>> words
['The',
 'crew',
 'of',
 'the',
 'USS',
 'Discovery',
 'discovered',
 'many',
 'discoveries',
 '.',
 'Discovering',
 'is',
 'what',
 'explorers',
 'do',
 '.']
```

Create a list of the stemmed versions of the words in words by using stemmer.stem() in a list comprehension:

```
>>>

>>> stemmed_words = [stemmer.stem(word) for word in words]
```

Take a look at what’s in stemmed_words:

```
>>>

>>> stemmed_words

['the',
 'crew',
 'of',
 'the',
 'uss',
 'discoveri',
 'discov',
 'mani',
 'discoveri',
 '.',
 'discov',
 'is',
 'what',
 'explor',
 'do',
 '.']
```

Here’s what happened to all the words that started with 'discov' or 'Discov':

Original word	Stemmed version
'Discovery'	'discoveri'
'discovered'	'discov'
'discoveries'	'discoveri'
'Discovering'	'discov'

Those results look a little inconsistent. Why would 'Discovery' give you 'discoveri' when 'Discovering' gives you 'discov'?

Understemming and overstemming are two ways stemming can go wrong:

Understemming happens when two related words should be reduced to the same stem but aren't. This is a false negative.

Overstemming happens when two unrelated words are reduced to the same stem even though they shouldn't be. This is a false positive.

The Porter stemming algorithm dates from 1979, so it's a little on the older side. The Snowball stemmer, which is also called Porter2, is an improvement on the original and is also available through NLTK, so you can use that one in your own projects. It's also worth noting that the purpose of the Porter stemmer is not to produce complete words but to find variant forms of a word.

Fortunately, you have some other ways to reduce words to their core meaning, such as lemmatizing, which you'll see later in this tutorial. But first, we need to cover parts of speech.

Tagging Parts of Speech

Part of speech is a grammatical term that deals with the roles words play when you use them together in sentences. Tagging parts of speech, or POS tagging, is the task of labeling the words in your text according to their part of speech.

In English, there are eight parts of speech:

Part of speech	Role	Examples
Noun	Is a person, place, or thing	mountain, bagel, Poland
Pronoun	Replaces a noun	you, she, we
Adjective	Gives information about what a noun is like	efficient, windy, colorful
Verb	Is an action or a state of being	learn, is, go
Adverb	Gives information about a verb, an adjective, or another adverb	efficiently, always, very
Preposition	Gives information about how a noun or pronoun is connected to another word	from, about, at
Conjunction	Connects two other words or phrases	so, because, and
Interjection	Is an exclamation	yay, ow, wow

Some sources also include the category articles (like "a" or "the") in the list of parts of speech, but other sources consider them to be adjectives. NLTK uses the word determiner to refer to articles.

Here's how to import the relevant parts of NLTK in order to tag parts of speech:

```
>>>

>>> from nltk.tokenize import word_tokenize
```

Now create some text to tag. You can use this Carl Sagan quote:

```
>>>
>>> sagan_quote = """
... If you wish to make an apple pie from scratch,
... you must first invent the universe."""
```

Use word_tokenize to separate the words in that string and store them in a list:

```
>>>
>>> words_in_sagan_quote = word_tokenize(sagan_quote)
```

Now call nltk.pos_tag() on your new list of words:

```
>>>
>>> import nltk
>>> nltk.pos_tag(words_in_sagan_quote)
[('If', 'IN'),
 ('you', 'PRP'),
 ('wish', 'VBP'),
 ('to', 'TO'),
 ('make', 'VB'),
 ('an', 'DT'),
 ('apple', 'NN'),
 ('pie', 'NN'),
 ('from', 'IN'),
 ('scratch', 'NN'),
 (',', ','),
 ('you', 'PRP'),
 ('must', 'MD'),
 ('first', 'VB'),
 ('invent', 'VB'),
 ('the', 'DT'),
 ('universe', 'NN'),
 ('.', '.')]
```

All the words in the quote are now in a separate tuple, with a tag that represents their part of speech. But what do the tags mean? Here's how to get a list of tags and their meanings:

```
>>>
>>> nltk.help.upenn_tagset()
```

The list is quite long, but feel free to expand the box below to see it.

POS Tags and Their MeaningsShow/Hide

Here's a summary that you can use to get started with NLTK's POS tags:

Tags that start with	Deal with
JJ	Adjectives
NN	Nouns
RB	Adverbs
PRP	Pronouns
VB	Verbs

Now that you know what the POS tags mean, you can see that your tagging was fairly successful:

'pie' was tagged NN because it's a singular noun.

'you' was tagged PRP because it's a personal pronoun.

'invent' was tagged VB because it's the base form of a verb.

But how would NLTK handle tagging the parts of speech in a text that is basically gibberish? Jabberwocky is a nonsense poem that doesn't technically mean much but is still written in a way that can convey some kind of meaning to English speakers.

Make a string to hold an excerpt from this poem:

```
>>>
>>> jabberwocky_excerpt = """
... 'Twas brillig, and the slithy toves did gyre and gimble in the wabe:
... all mimsy were the borogoves, and the mome raths outgrabe."""
```

Use word_tokenize to separate the words in the excerpt and store them in a list:

```
>>>
>>> words_in_excerpt = word_tokenize(jabberwocky_excerpt)
```

Call nltk.pos_tag() on your new list of words:

```
>>>
>>> nltk.pos_tag(words_in_excerpt)
```

```
[("'T", 'NN'),
 ('was', 'VBD'),
 ('brillig', 'VBN'),
 (',', ','),
 ('and', 'CC'),
 ('the', 'DT'),
 ('slithy', 'JJ'),
 ('toves', 'NNS'),
 ('did', 'VBD'),
 ('gyre', 'NN'),
 ('and', 'CC'),
 ('gimble', 'JJ'),
 ('in', 'IN'),
 ('the', 'DT'),
 ('wabe', 'NN'),
 (':', ':'),
 ('all', 'DT'),
 ('mimsy', 'NNS'),
 ('were', 'VBD'),
 ('the', 'DT'),
 ('borogoves', 'NNS'),
 (',', ','),
 ('and', 'CC'),
 ('the', 'DT'),
 ('mome', 'JJ'),
 ('raths', 'NNS'),
 ('outgrabe', 'RB'),
 ('.', '.')]
```

Accepted English words like 'and' and 'the' were correctly tagged as a conjunction and a determiner, respectively. The gibberish word 'slithy' was tagged as an adjective, which is what a human English speaker would probably assume from the context of the poem as well. Way to go, NLTK!

Lemmatizing

Now that you're up to speed on parts of speech, you can circle back to lemmatizing. Like stemming, lemmatizing reduces words to their core meaning, but it will give you a complete English word that makes sense on its own instead of just a fragment of a word like 'discoveri'.

Note: A lemma is a word that represents a whole group of words, and that group of words is called a lexeme.

For example, if you were to look up the word "blending" in a dictionary, then you'd need to look at the entry for "blend," but you would find "blending" listed in that entry.

In this example, "blend" is the lemma, and "blending" is part of the lexeme. So when you lemmatize a word, you are reducing it to its lemma.

Here's how to import the relevant parts of NLTK in order to start lemmatizing:

```
>>>

>>> from nltk.stem import WordNetLemmatizer
```

Create a lemmatizer to use:

```
>>>

>>> lemmatizer = WordNetLemmatizer()
```

Let's start with lemmatizing a plural noun:

```
>>>

>>> lemmatizer.lemmatize("scarves")

'scarf'
```

"scarves" gave you 'scarf', so that's already a bit more sophisticated than what you would have gotten with the Porter stemmer, which is 'scarv'. Next, create a string with more than one word to lemmatize:

```
>>>

>>> string_for_lemmatizing = "The friends of DeSoto love scarves."
```

Now tokenize that string by word:

```
>>>

>>> words = word_tokenize(string_for_lemmatizing)
```

Here's your list of words:

```
>>>

>>> words

['The',

 'friends',

 'of',
```

```
 'DeSoto',
 'love'
 'scarves',
 '.']
```

Create a list containing all the words in words after they've been lemmatized:

```
>>>
>>> lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
```

Here's the list you got:

```
>>>
>>> lemmatized_words
['The',
 'friend',
 'of',
 'DeSoto',
 'love',
 'scarf',
 '.'
```

That looks right. The plurals 'friends' and 'scarves' became the singulars 'friend' and 'scarf'.

But what would happen if you lemmatized a word that looked very different from its lemma? Try lemmatizing "worst":

```
>>>
>>> lemmatizer.lemmatize("worst")
'worst'
```

You got the result 'worst' because lemmatizer.lemmatize() assumed that "worst" was a noun. You can make it clear that you want "worst" to be an adjective:

```
>>>
>>> lemmatizer.lemmatize("worst", pos="a")
'bad'
```

The default parameter for pos is 'n' for noun, but you made sure that "worst" was treated as an adjective by adding the parameter pos="a". As a result, you got 'bad', which looks very different from your original word and is nothing like what you'd get if you were stemming. This is because "worst" is the superlative form of the adjective 'bad', and lemmatizing reduces superlatives as well as comparatives to their lemmas.

Now that you know how to use NLTK to tag parts of speech, you can try tagging your words before lemmatizing them to avoid mixing up homographs, or words that are spelled the same but have different meanings and can be different parts of speech.

Chunking

While tokenizing allows you to identify words and sentences, chunking allows you to identify phrases.

Note: A phrase is a word or group of words that works as a single unit to perform a grammatical function. Noun phrases are built around a noun.

Here are some examples:

“A planet”

“A tilting planet”

“A swiftly tilting planet”

Chunking makes use of POS tags to group words and apply chunk tags to those groups. Chunks don’t overlap, so one instance of a word can be in only one chunk at a time.

Here’s how to import the relevant parts of NLTK in order to chunk:

```
>>>

>>> from nltk.tokenize import word_tokenize
```

Before you can chunk, you need to make sure that the parts of speech in your text are tagged, so create a string for POS tagging. You can use this quote from The Lord of the Rings:

```
>>>

>>> lotr_quote = "It's a dangerous business, Frodo, going out your door."
```

Now tokenize that string by word:

```
>>>

>>> words_in_lotr_quote = word_tokenize(lotr_quote)

>>> words_in_lotr_quote

['It',

 "'s",

 'a',

 'dangerous',

 'business',

 ',',

 'Frodo',

 ',',

 'going',
```

```
 'out',
 'your',
 'door',
 '.']
```

Now you’ve got a list of all of the words in lotr_quote.

The next step is to tag those words by part of speech:

```
>>>
>>> nltk.download("averaged_perceptron_tagger")
>>> lotr_pos_tags = nltk.pos_tag(words_in_lotr_quote)
>>> lotr_pos_tags
[('It', 'PRP'),
 ("'s", 'VBZ'),
 ('a', 'DT'),
 ('dangerous', 'JJ'),
 ('business', 'NN'),
 (',', ','),
 ('Frodo', 'NNP'),
 (',', ','),
 ('going', 'VBG'),
 ('out', 'RP'),
 ('your', 'PRP$'),
 ('door', 'NN'),
 ('.', '.')]
```

You’ve got a list of tuples of all the words in the quote, along with their POS tag. In order to chunk, you first need to define a chunk grammar.

Note: A chunk grammar is a combination of rules on how sentences should be chunked. It often uses regular expressions, or regexes.

For this tutorial, you don’t need to know how regular expressions work, but they will definitely come in handy for you in the future if you want to process text.

Create a chunk grammar with one regular expression rule:

```
>>>
>>> grammar = "NP: {<DT>?<JJ>*<NN>}"
```

NP stands for noun phrase. You can learn more about noun phrase chunking in [Chapter 7](#) of Natural Language Processing with Python—Analyzing Text with the Natural Language Toolkit.

According to the rule you created, your chunks:

Start with an optional (?) determiner ('DT')

Can have any number (*) of adjectives (JJ)

End with a noun (<NN>)

Create a chunk parser with this grammar:

```
>>>

>>> chunk_parser = nltk.RegexpParser(grammar)
```

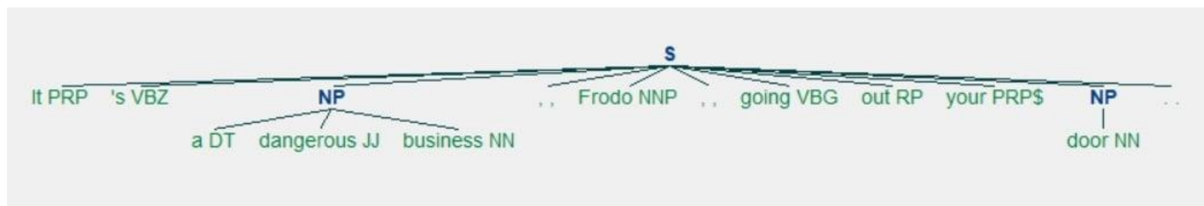
Now try it out with your quote:

```
>>>

>>> tree = chunk_parser.parse(lotr_pos_tags)
```

Here’s how you can see a visual representation of this tree:

```
>>>

>>> tree.draw()
```

This is what the visual representation looks like:



You got two noun phrases:

'a dangerous business' has a determiner, an adjective, and a noun.

'door' has just a noun.

Now that you know about chunking, it’s time to look at chinking.

Chinking

Chinking is used together with chunking, but while chunking is used to include a pattern, chinking is used to exclude a pattern.

Let’s reuse the quote you used in the section on chunking. You already have a list of tuples containing each of the words in the quote along with its part of speech tag:

```
>>>

>>> lotr_pos_tags

[('It', 'PRP'),
```

```
 ("'s", 'VBZ'),
 ('a', 'DT'),
 ('dangerous', 'JJ'),
 ('business', 'NN'),
 (',', ','),
 ('Frodo', 'NNP'),
 (',', ','),
 ('going', 'VBG'),
 ('out', 'RP'),
 ('your', 'PRP$'),
 ('door', 'NN'),
 ('.', '.')]
```

The next step is to create a grammar to determine what you want to include and exclude in your chunks. This time, you're going to use more than one line because you're going to have more than one rule. Because you're using more than one line for the grammar, you'll be using triple quotes ("""):

```
>>>
>>> grammar = """
... Chunk: {<.*>+}
...       }<JJ>{"""
```

The first rule of your grammar is {<.*>+}. This rule has curly braces that face inward ({}) because it's used to determine what patterns you want to include in you chunks. In this case, you want to include everything: <.*>+.

The second rule of your grammar is }<JJ>{. This rule has curly braces that face outward (}{) because it's used to determine what patterns you want to exclude in your chunks. In this case, you want to exclude adjectives: <JJ>.

Create a chunk parser with this grammar:

```
>>>
>>> chunk_parser = nltk.RegexpParser(grammar)
```

Now chunk your sentence with the chink you specified:

```
>>>
>>> tree = chunk_parser.parse(lotr_pos_tags)
```

You get this tree as a result:

```
>>>
```

>>> tree

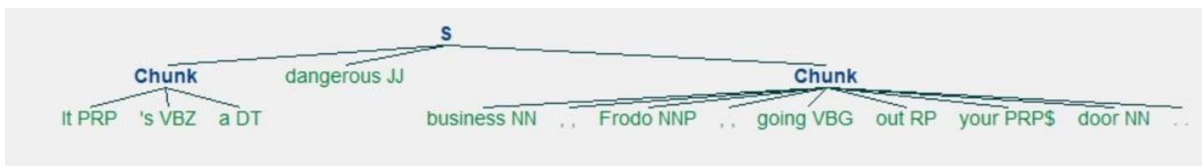
Tree('S', [Tree('Chunk', [('It', 'PRP'), ("'s", 'VBZ'), ('a', 'DT')]), ('dangerous', 'JJ'), Tree('Chunk', [('business', 'NN'), (',', ','), ('Frodo', 'NNP'), (',', ','), ('going', 'VBG'), ('out', 'RP'), ('your', 'PRP$'), ('door', 'NN'), ('.', '.')])])

In this case, ('dangerous', 'JJ') was excluded from the chunks because it's an adjective (JJ). But that will be easier to see if you get a graphic representation again:

>>>

>>> tree.draw()

You get this visual representation of the tree:



Here, you've excluded the adjective 'dangerous' from your chunks and are left with two chunks containing everything else. The first chunk has all the text that appeared before the adjective that was excluded. The second chunk contains everything after the adjective that was excluded.

Now that you know how to exclude patterns from your chunks, it's time to look into named entity recognition (NER).

Using Named Entity Recognition (NER)

Named entities are noun phrases that refer to specific locations, people, organizations, and so on. With named entity recognition, you can find the named entities in your texts and also determine what kind of named entity they are.

Here's the list of named entity types from the NLTK book:

NE type	Examples
ORGANIZATION	Georgia-Pacific Corp., WHO
PERSON	Eddy Bonte, President Obama
LOCATION	Murray River, Mount Everest
DATE	June, 2008-06-29
TIME	two fifty a m, 1:30 p.m.
MONEY	175 million Canadian dollars, GBP 10.40
PERCENT	twenty pct, 18.75 %
FACILITY	Washington Monument, Stonehenge
GPE	South East Asia, Midlothian

You can use nltk.ne_chunk() to recognize named entities. Let's use lotr_pos_tags again to test it out:

```
>>>
>>> nltk.download("maxent_ne_chunker")
>>> nltk.download("words")
>>> tree = nltk.ne_chunk(lotr_pos_tags)
```

Now take a look at the visual representation:

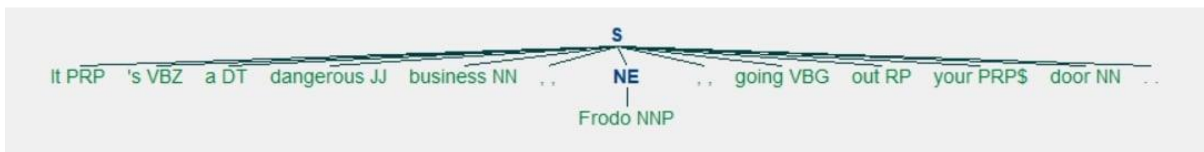
```
>>>
>>> tree.draw()
```

Here's what you get:

See how Frodo has been tagged as a PERSON? You also have the option to use the parameter binary=True if you just want to know what the named entities are but not what kind of named entity they are:

```
>>>
>>> tree = nltk.ne_chunk(lotr_pos_tags, binary=True)
>>> tree.draw()
```

Now all you see is that Frodo is an NE:



That's how you can identify named entities! But you can take this one step further and extract named entities directly from your text. Create a string from which to extract named entities. You can use this quote from [The War of the Worlds]:

```
>>>
>>> quote = """
... Men like Schiaparelli watched the red planet—it is odd, by-the-bye, that
... for countless centuries Mars has been the star of war—but failed to
... interpret the fluctuating appearances of the markings they mapped so well.
... All that time the Martians must have been getting ready.
...
... During the opposition of 1894 a great light was seen on the illuminated
```

```
... part of the disk, first at the Lick Observatory, then by Perrotin of Nice,
... and then by other observers. English readers heard of it first in the
... issue of Nature dated August 2."""
```

Now create a function to extract named entities:

```
>>>
>>> def extract_ne(quote):
...     words = word_tokenize(quote, language=language)
...     tags = nltk.pos_tag(words)
...     tree = nltk.ne_chunk(tags, binary=True)
...     return set(
...         " ".join(i[0] for i in t)
...         for t in tree
...         if hasattr(t, "label") and t.label() == "NE"
...     )
```

With this function, you gather all named entities, with no repeats. In order to do that, you tokenize by word, apply part of speech tags to those words, and then extract named entities based on those tags. Because you included binary=True, the named entities you'll get won't be labeled more specifically. You'll just know that they're named entities.

Take a look at the information you extracted:

```
>>>
>>> extract_ne(quote)
{'Lick Observatory', 'Mars', 'Nature', 'Perrotin', 'Schiaparelli'}
```

You missed the city of Nice, possibly because NLTK interpreted it as a regular English adjective, but you still got the following:

An institution: 'Lick Observatory'

A planet: 'Mars'

A publication: 'Nature'

People: 'Perrotin', 'Schiaparelli'

That's some pretty decent variety!

Getting Text to Analyze

Now that you've done some text processing tasks with small example texts, you're ready to analyze a bunch of texts at once. A group of texts is called a corpus. NLTK provides several corpora covering everything from novels hosted by Project Gutenberg to inaugural speeches by presidents of the United States.

In order to analyze texts in NLTK, you first need to import them. This requires nltk.download("book"), which is a pretty big download:

```
>>>

>>> nltk.download("book")

>>> from nltk.book import *

*** Introductory Examples for the NLTK Book ***

Loading text1, ..., text9 and sent1, ..., sent9

Type the name of the text or sentence to view it.

Type: 'texts()' or 'sents()' to list the materials.

text1: Moby Dick by Herman Melville 1851

text2: Sense and Sensibility by Jane Austen 1811

text3: The Book of Genesis

text4: Inaugural Address Corpus

text5: Chat Corpus

text6: Monty Python and the Holy Grail

text7: Wall Street Journal

text8: Personals Corpus

text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

You now have access to a few linear texts (such as Sense and Sensibility and Monty Python and the Holy Grail) as well as a few groups of texts (such as a chat corpus and a personals corpus). Human nature is fascinating, so let’s see what we can find out by taking a closer look at the personals corpus!

This corpus is a collection of personals ads, which were an early version of online dating. If you wanted to meet someone, then you could place an ad in a newspaper and wait for other readers to respond to you.

If you’d like to learn how to get other texts to analyze, then you can check out Chapter 3 of Natural Language Processing with Python – Analyzing Text with the Natural Language Toolkit.

Using a Concordance

When you use a concordance, you can see each time a word is used, along with its immediate context. This can give you a peek into how a word is being used at the sentence level and what words are used with it.

Let’s see what these good people looking for love have to say! The personals corpus is called text8, so we’re going to call .concordance() on it with the parameter "man":

```
>>>

>>> text8.concordance("man")
```

Displaying 14 of 14 matches:

to hearing from you all . ABLE young man seeks , sexy older women . Phone for

ble relationship . GENUINE ATTRACTIVE MAN 40 y . o ., no ties , secure , 5 ft .

ship , and quality times . VIETNAMESE MAN Single , never married , financially

ip . WELL DRESSED emotionally healthy man 37 like to meet full figured woman fo

nth subs LIKE TO BE MISTRESS of YOUR MAN like to be treated well . Bold DTE no

eeks lady in similar position MARRIED MAN 50 , attrac . fit , seeks lady 40 - 5

eks nice girl 25 - 30 serious rship . Man 46 attractive fit , assertive , and k

40 - 50 sought by Aussie mid 40s b / man f / ship r / ship LOVE to meet widowe

discreet times . Sth E Subs . MARRIED MAN 42yo 6ft , fit , seeks Lady for discr

woman , seeks professional , employed man , with interests in theatre , dining

tall and of large build seeks a good man . I am a nonsmoker , social drinker ,

lead to relationship . SEEKING HONEST MAN I am 41 y . o ., 5 ft . 4 , med . bui

quiet times . Seeks 35 - 45 , honest man with good SOH & similar interests , f

genuine , caring , honest and normal man for fship , poss rship . S / S , S /

Interestingly, the last three of those fourteen matches have to do with seeking an honest man, specifically:

SEEKING HONEST MAN

Seeks 35 - 45 , honest man with good SOH & similar interests

genuine , caring , honest and normal man for fship , poss rship

Let’s see if there’s a similar pattern with the word "woman":

```
>>>
>>> text8.concordance("woman")
```

Displaying 11 of 11 matches:

at home . Seeking an honest , caring woman , slim or med . build , who enjoys t

thy man 37 like to meet full figured woman for relationship . 48 slim , shy , S

rry . MALE 58 years old . Is there a Woman who would like to spend 1 weekend a

other interests . Seeking Christian Woman for fship , view to rship . SWM 45 D

ALE 60 - burly beared seeks intimate woman for outings n / s s / d F / ston / P

ington . SCORPIO 47 seeks passionate woman for discreet intimate encounters SEX

le dad . 42 , East sub . 5 " 9 seeks woman 30 + for f / ship relationship TALL

personal trainer looking for married woman age open for fun MARRIED Dark guy 37

rinker , seeking slim - medium build woman who is happy in life , age open . AC

. O . TERTIARY Educated professional woman , seeks professional , employed man

 real romantic , age 50 - 65 y . o . WOMAN OF SUBSTANCE 56 , 59 kg ., 50 , fit

The issue of honesty came up in the first match only:

Seeking an honest , caring woman , slim or med . build

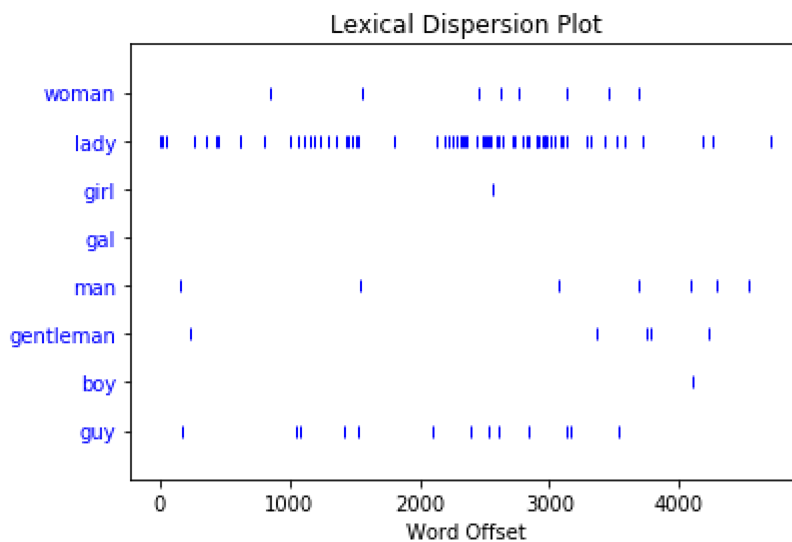
Dipping into a corpus with a concordance won't give you the full picture, but it can still be interesting to take a peek and see if anything stands out.

Making a Dispersion Plot

You can use a dispersion plot to see how much a particular word appears and where it appears. So far, we've looked for "man" and "woman", but it would be interesting to see how much those words are used compared to their synonyms:

```
>>>
>>> text8.dispersion_plot(
...     ["woman", "lady", "girl", "gal", "man", "gentleman", "boy", "guy"]
... )
```

Here's the dispersion plot you get:



Each vertical blue line represents one instance of a word. Each horizontal row of blue lines represents the corpus as a whole. This plot shows that:

"lady" was used a lot more than "woman" or "girl". There were no instances of "gal".

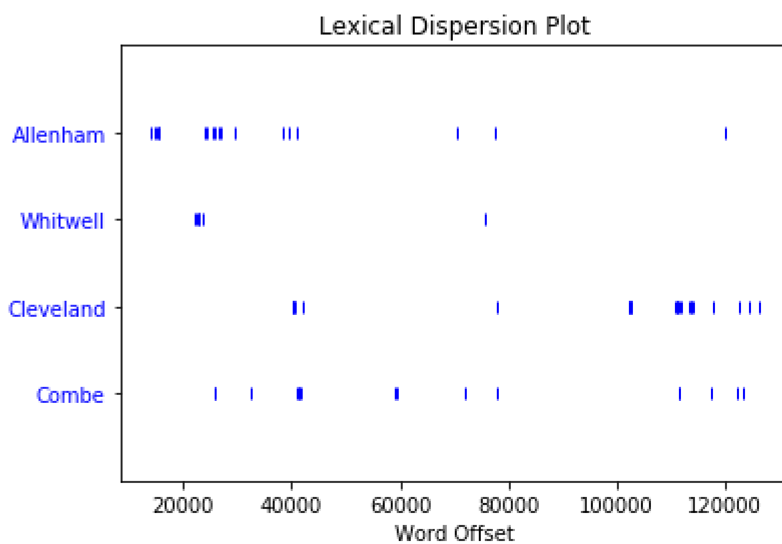
"man" and "guy" were used a similar number of times and were more common than "gentleman" or "boy".

You use a dispersion plot when you want to see where words show up in a text or corpus. If you're analyzing a single text, this can help you see which words show up near each other. If you're analyzing a corpus of texts that is organized chronologically, it can help you see which words were being used more or less over a period of time.

Staying on the theme of romance, see what you can find out by making a dispersion plot for Sense and Sensibility, which is text2. Jane Austen novels talk a lot about people's homes, so make a dispersion plot with the names of a few homes:

```
>>>
>>> text2.dispersion_plot(["Allenham", "Whitwell", "Cleveland", "Combe"])
```

Here's the plot you get:



Apparently Allenham is mentioned a lot in the first third of the novel and then doesn't come up much again. Cleveland, on the other hand, barely comes up in the first two thirds but shows up a fair bit in the last third. This distribution reflects changes in the relationship between Marianne and Willoughby:

Allenham is the home of Willoughby's benefactress and comes up a lot when Marianne is first interested in him.

Cleveland is a home that Marianne stays at after she goes to see Willoughby in London and things go wrong.

Dispersion plots are just one type of visualization you can make for textual data. The next one you'll take a look at is frequency distributions.

Making a Frequency Distribution

With a frequency distribution, you can check which words show up most frequently in your text. You'll need to get started with an import:

```
>>>
>>> from nltk import FreqDist
```

FreqDist is a subclass of collections.Counter. Here's how to create a frequency distribution of the entire corpus of personals ads:

```
>>>
>>> frequency_distribution = FreqDist(text8)
>>> print(frequency_distribution)
<FreqDist with 1108 samples and 4867 outcomes>
```

Since 1108 samples and 4867 outcomes is a lot of information, start by narrowing that down. Here's how to see the 20 most common words in the corpus:

```
>>>
>>> frequency_distribution.most_common(20)
[(',', 539),
 ('.', 353),
 ('/', 110),
 ('for', 99),
 ('and', 74),
 ('to', 74),
 ('lady', 68),
 ('-', 66),
 ('seeks', 60),
 ('a', 52),
 ('with', 44),
 ('S', 36),
 ('ship', 33),
 ('&', 30),
 ('relationship', 29),
 ('fun', 28),
 ('in', 27),
 ('slim', 27),
 ('build', 27),
 ('o', 26)]
```

You have a lot of stop words in your frequency distribution, but you can remove them just as you did earlier. Create a list of all of the words in text8 that aren't stop words:

```
>>>
>>> meaningful_words = [
```

```
...     word for word in text8 if word.casefold() not in stop_words
... ]
```

Now that you have a list of all of the words in your corpus that aren't stop words, make a frequency distribution:

```
>>>
>>> frequency_distribution = FreqDist(meaningful_words)
```

Take a look at the 20 most common words:

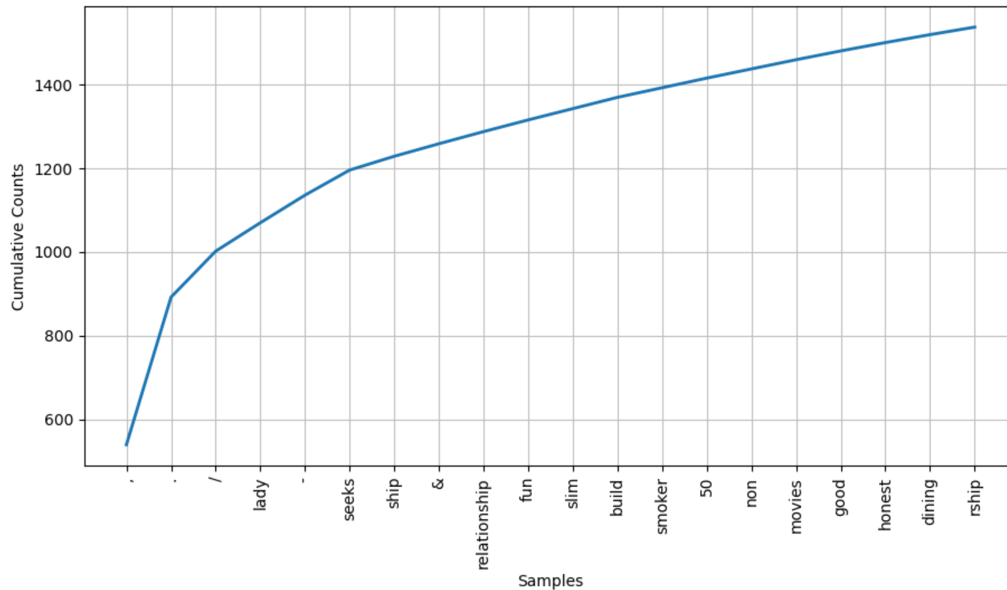
```
>>>
>>> frequency_distribution.most_common(20)
[(',', 539),
 ('.', 353),
 ('/', 110),
 ('lady', 68),
 ('-', 66),
 ('seeks', 60),
 ('ship', 33),
 ('&', 30),
 ('relationship', 29),
 ('fun', 28),
 ('slim', 27),
 ('build', 27),
 ('smoker', 23),
 ('50', 23),
 ('non', 22),
 ('movies', 22),
 ('good', 21),
 ('honest', 20),
 ('dining', 19),
 ('rship', 18)]
```

You can turn this list into a graph:

```
>>>
```

```
>>> frequency_distribution.plot(20, cumulative=True)
```

Here's the graph you get:



Some of the most common words are:

'lady'

'seeks'

'ship'

'relationship'

'fun'

'slim'

'build'

'smoker'

'50'

'non'

'movies'

'good'

'honest'

From what you've already learned about the people writing these personals ads, they did seem interested in honesty and used the word 'lady' a lot. In addition, 'slim' and 'build' both show up the same number of times. You saw slim and build used near each other when you were learning

about concordances, so maybe those two words are commonly used together in this corpus. That brings us to collocations!

Finding Collocations

A collocation is a sequence of words that shows up often. If you're interested in common collocations in English, then you can check out The BBI Dictionary of English Word Combinations. It's a handy reference you can use to help you make sure your writing is idiomatic. Here are some examples of collocations that use the word "tree":

Syntax tree

Family tree

Decision tree

To see pairs of words that come up often in your corpus, you need to call .collocations() on it:

```
>>>

>>> text8.collocations()

would like; medium build; social drinker; quiet nights; non smoker;

long term; age open; Would like; easy going; financially secure; fun

times; similar interests; Age open; weekends away; poss rship; well

presented; never married; single mum; permanent relationship; slim

build
```

slim build did show up, as did medium build and several other word combinations. No long walks on the beach though!

But what would happen if you looked for collocations after lemmatizing the words in your corpus? Would you find some word combinations that you missed the first time around because they came up in slightly varied versions?

If you followed the instructions earlier, then you'll already have a lemmatizer, but you can't call collocations() on just any data type, so you're going to need to do some prep work. Start by creating a list of the lemmatized versions of all the words in text8:

```
>>>

>>> lemmatized_words = [lemmatizer.lemmatize(word) for word in text8]
```

But in order for you to be able to do the linguistic processing tasks you've seen so far, you need to make an NLTK text with this list:

```
>>>

>>> new_text = nltk.Text(lemmatized_words)
```

Here's how to see the collocations in your new_text:

```
>>>
```

```
>>> new_text.collocations()
medium build; social drinker; non smoker; long term; would like; age
open; easy going; financially secure; Would like; quiet night; Age
open; well presented; never married; single mum; permanent
relationship; slim build; year old; similar interest; fun time; Photo
pls
```

Compared to your previous list of collocations, this new one is missing a few:

weekends away

poss rship

The idea of quiet nights still shows up in the lemmatized version, quiet night. Your latest search for collocations also brought up a few news ones:

year old suggests that users often mention ages.

photo pls suggests that users often request one or more photos.

That's how you can find common word combinations to see what people are talking about and how they're talking about it!

Conclusion

Congratulations on taking your first steps with NLP! A whole new world of unstructured data is now open for you to explore. Now that you've covered the basics of text analytics tasks, you can get out there are find some texts to analyze and see what you can learn about the texts themselves as well as the people who wrote them and the topics they're about.

Now you know how to:

Find text to analyze

Preprocess your text for analysis

Analyze your text

Create visualizations based on your analysis

For your next step, you can use NLTK to analyze a text to see whether the sentiments expressed in it are positive or negative. To learn more about sentiment analysis, check out Sentiment Analysis: First Steps With Python's NLTK Library. If you'd like to dive deeper into the nuts and bolts of NLTK, then you can work your way through Natural Language Processing with Python—Analyzing Text with the Natural Language Toolkit.

Day-03 & 04: Sentiment Analysis: First Steps With Python's NLTK Library

Sentiment Analysis

Sentiment analysis can help you determine the ratio of positive to negative engagements about a specific topic. You can analyze bodies of text, such as comments, tweets, and product reviews, to obtain insights from your audience. In this tutorial, you'll learn the important features of NLTK for

processing text data and the different approaches you can use to perform sentiment analysis on your data.

Steps for Sentiment Analysis

Split and filter text data in preparation for analysis

Analyze word frequency

Find concordance and collocations using different methods

Perform quick sentiment analysis with NLTK's built-in classifier

Define features for custom classification

Use and compare classifiers for sentiment analysis with NLTK

Getting Started With NLTK

The NLTK library contains various utilities that allow you to effectively manipulate and analyze linguistic data. Among its advanced features are text classifiers that you can use for many kinds of classification, including sentiment analysis.

Sentiment analysis is the practice of using algorithms to classify various samples of related text into overall positive and negative categories. With NLTK, you can employ these algorithms through powerful built-in machine learning operations to obtain insights from linguistic data.

import nltk

nltk.download()

NLTK will display a download manager showing all available and installed resources. Here are the ones you'll need to download for this task:

names: A list of common English names compiled by Mark Kantrowitz

stopwords: A list of really common words, like articles, pronouns, prepositions, and conjunctions

state_union: A sample of transcribed State of the Union addresses by different US presidents, compiled by Kathleen Ahrens

twitter_samples: A list of social media phrases posted to Twitter

movie_reviews: Two thousand movie reviews categorized by Bo Pang and Lillian Lee

averaged_perceptron_tagger: A data model that NLTK uses to categorize words into their part of speech

vader_lexicon: A scored list of words and jargon that NLTK references when performing sentiment analysis, created by C.J. Hutto and Eric Gilbert

punkt: A data model created by Jan Strunk that NLTK uses to split full texts into word lists

A quick way to download specific resources directly from the console is to pass a list to nltk.download():

>>>

```
>>> import nltk


>>> nltk.download([
...     "names",
...     "stopwords",
...     "state_union",
...     "twitter_samples",
...     "movie_reviews",
...     "averaged_perceptron_tagger",
...     "vader_lexicon",
...     "punkt",
... ])
```

his will tell NLTK to find and download each resource based on its identifier.

Should NLTK require additional resources that you haven't installed, you'll see a helpful LookupError with details and instructions to download the resource:

```
>>> import nltk


>>> w = nltk.corpus.shakespeare.words()
...
LookupError:
**********************************************************************
  Resource shakespeare not found.
  Please use the NLTK Downloader to obtain the resource:


  >>> import nltk
  >>> nltk.download('shakespeare')
```

Compiling Data

NLTK provides a number of functions that you can call with few or no arguments that will help you meaningfully analyze text before you even touch its machine learning capabilities. Many of NLTK's utilities are helpful in preparing your data for more advanced analysis.

Soon, you'll learn about frequency distributions, concordance, and collocations. But first, you need some data.

Start by loading the State of the Union corpus you downloaded earlier:

```
words = [w for w in nltk.corpus.state_union.words() if w.isalpha()]
```

Note that you build a list of individual words with the corpus's .words() method, but you use str.isalpha() to include only the words that are made up of letters. Otherwise, your word list may end up with "words" that are only punctuation marks.

Have a look at your list. You'll notice lots of little words like "of," "a," "the," and similar. These common words are called stop words, and they can have a negative effect on your analysis because they occur so often in the text. Thankfully, there's a convenient way to filter them out.

NLTK provides a small corpus of stop words that you can load into a list:

```
stopwords = nltk.corpus.stopwords.words("english")
```

Make sure to specify english as the desired language since this corpus contains stop words in various languages.

Now you can remove stop words from your original word list:

```
words = [w for w in words if w.lower() not in stopwords]
```

Since all words in the stopwords list are lowercase, and those in the original list may not be, you use str.lower() to account for any discrepancies. Otherwise, you may end up with mixedCase or capitalized stop words still in your list.

While you'll use corpora provided by NLTK for this tutorial, it's possible to build your own text corpora from any source. Building a corpus can be as simple as loading some plain text or as complex as labeling and categorizing each sentence. Refer to NLTK's documentation for more information on [how to work with corpus readers](#).

For some quick analysis, creating a corpus could be overkill. If all you need is a word list, there are simpler ways to achieve that goal. Beyond Python's own string manipulation methods, NLTK provides nltk.word_tokenize(), a function that splits raw text into individual words. While tokenization is itself a bigger topic (and likely one of the steps you'll take when creating a custom corpus), this tokenizer delivers simple word lists really well.

To use it, call word_tokenize() with the raw text you want to split:

```
>>>

>>> from pprint import pprint


>>> text = """

... For some quick analysis, creating a corpus could be overkill.

... If all you need is a word list,

... there are simpler ways to achieve that goal."""

>>> pprint(nltk.word_tokenize(text), width=79, compact=True)

['For', 'some', 'quick', 'analysis', ',', 'creating', 'a', 'corpus', 'could',

 'be', 'overkill', '.', 'If', 'all', 'you', 'need', 'is', 'a', 'word', 'list',
```

',', 'there', 'are', 'simpler', 'ways', 'to', 'achieve', 'that', 'goal', '.']

Now you have a workable word list! Remember that punctuation will be counted as individual words, so use str.isalpha() to filter them out later.

Creating Frequency Distributions

Now you're ready for frequency distributions. A frequency distribution is essentially a table that tells you how many times each word appears within a given text. In NLTK, frequency distributions are a specific object type implemented as a distinct class called FreqDist. This class provides useful operations for word frequency analysis.

To build a frequency distribution with NLTK, construct the nltk.FreqDist class with a word list:

```
words: list[str] = nltk.word_tokenize(text)

fd = nltk.FreqDist(words)
```

This will create a frequency distribution object similar to a Python dictionary but with added features.

Note: Type hints with generics as you saw above in words: list[str] = ... is a new feature in Python 3.9!

After building the object, you can use methods like .most_common() and .tabulate() to start visualizing information:

```
>>>

>>> fd.most_common(3)

[('must', 1568), ('people', 1291), ('world', 1128)]

>>> fd.tabulate(3)

  must people  world

  1568   1291   1128
```

These methods allow you to quickly determine frequently used words in a sample. With .most_common(), you get a list of tuples containing each word and how many times it appears in your text. You can get the same information in a more readable format with .tabulate().

In addition to these two methods, you can use frequency distributions to query particular words. You can also use them as iterators to perform some custom analysis on word properties.

For example, to discover differences in case, you can query for different variations of the same word:

```
>>>

>>> fd["America"]

1076

>>> fd["america"]  # Note this doesn't result in a KeyError

0

>>> fd["AMERICA"]

3
```

These return values indicate the number of times each word occurs exactly as given.

Since frequency distribution objects are iterable, you can use them within list comprehensions to create subsets of the initial distribution. You can focus these subsets on properties that are useful for your own analysis.

Try creating a new frequency distribution that's based on the initial one but normalizes all words to lowercase:

```
lower_fd = nltk.FreqDist([w.lower() for w in fd])
```

Now you have a more accurate representation of word usage regardless of case.

Think of the possibilities: You could create frequency distributions of words starting with a particular letter, or of a particular length, or containing certain letters. Your imagination is the limit!

Extracting Concordance and Collocations

In the context of NLP, a concordance is a collection of word locations along with their context. You can use concordances to find:

How many times a word appears

Where each occurrence appears

What words surround each occurrence

In NLTK, you can do this by calling .concordance(). To use it, you need an instance of the nltk.Text class, which can also be constructed with a word list.

Before invoking .concordance(), build a new word list from the original corpus text so that all the context, even stop words, will be there:

```
>>>

>>> text = nltk.Text(nltk.corpus.state_union.words())

>>> text.concordance("america", lines=5)

Displaying 5 of 1079 matches:

 would want us to do . That is what America will do . So much blood has already

ay , the entire world is looking to America for enlightened leadership to peace

beyond any shadow of a doubt , that America will continue the fight for freedom

 to make complete victory certain , America will never become a party to any pl

nly in law and in justice . Here in America , we have labored long and hard to
```

Note that .concordance() already ignores case, allowing you to see the context of all case variants of a word in order of appearance. Note also that this function doesn't show you the location of each word in the text.

Additionally, since .concordance() only prints information to the console, it's not ideal for data manipulation. To obtain a usable list that will also give you information about the location of each occurrence, use .concordance_list():

```
>>>
>>> concordance_list = text.concordance_list("america", lines=2)
>>> for entry in concordance_list:
...     print(entry.line)
...
 would want us to do . That is what America will do . So much blood has already
ay , the entire world is looking to America for enlightened leadership to peace
```

.concordance_list() gives you a list of ConcordanceLine objects, which contain information about where each word occurs as well as a few more properties worth exploring. The list is also sorted in order of appearance.

The nltk.Text class itself has a few other interesting features. One of them is .vocab(), which is worth mentioning because it creates a frequency distribution for a given text.

Revisiting nltk.word_tokenize(), check out how quickly you can create a custom nltk.Text instance and an accompanying frequency distribution:

```
>>>
>>> words: list[str] = nltk.word_tokenize(
...     """Beautiful is better than ugly.
...     Explicit is better than implicit.
...     Simple is better than complex."""
... )
>>> text = nltk.Text(words)
>>> fd = text.vocab()  # Equivalent to fd = nltk.FreqDist(words)
>>> fd.tabulate(3)
    is better   than
     3      3      3
```

.vocab() is essentially a shortcut to create a frequency distribution from an instance of nltk.Text. That way, you don’t have to make a separate call to instantiate a new nltk.FreqDist object.

Another powerful feature of NLTK is its ability to quickly find collocations with simple function calls. Collocations are series of words that frequently appear together in a given text. In the State of the Union corpus, for example, you’d expect to find the words United and States appearing next to each other very often. Those two words appearing together is a collocation.

Collocations can be made up of two or more words. NLTK provides classes to handle several types of collocations:

Bigrams: Frequent two-word combinations

Trigrams: Frequent three-word combinations

Quadgrams: Frequent four-word combinations

NLTK provides specific classes for you to find collocations in your text. Following the pattern you've seen so far, these classes are also built from lists of words:

```
words = [w for w in nltk.corpus.state_union.words() if w.isalpha()]

finder = nltk.collocations.TrigramCollocationFinder.from_words(words)
```

The TrigramCollocationFinder instance will search specifically for trigrams. As you may have guessed, NLTK also has the BigramCollocationFinder and QuadgramCollocationFinder classes for bigrams and quadgrams, respectively. All these classes have a number of utilities to give you information about all identified collocations.

One of their most useful tools is the ngram_fd property. This property holds a frequency distribution that is built for each collocation rather than for individual words.

Using ngram_fd, you can find the most common collocations in the supplied text:

```
>>>

>>> finder.ngram_fd.most_common(2)

[(('the', 'United', 'States'), 294), (('the', 'American', 'people'), 185)]

>>> finder.ngram_fd.tabulate(2)

 ('the', 'United', 'States') ('the', 'American', 'people')

                     294                        185
```

You don't even have to create the frequency distribution, as it's already a property of the collocation finder instance.

Now that you've learned about some of NLTK's most useful tools, it's time to jump into sentiment analysis!

Using NLTK's Pre-Trained Sentiment Analyzer

NLTK already has a built-in, pretrained sentiment analyzer called VADER (Valence Aware Dictionary and sEntiment Reasoner).

Since VADER is pretrained, you can get results more quickly than with many other analyzers. However, VADER is best suited for language used in social media, like short sentences with some slang and abbreviations. It's less accurate when rating longer, structured sentences, but it's often a good launching point.

To use VADER, first create an instance of nltk.sentiment.SentimentIntensityAnalyzer, then use .polarity_scores() on a raw string:

```
>>>

>>> from nltk.sentiment import SentimentIntensityAnalyzer

>>> sia = SentimentIntensityAnalyzer()
```

```
>>> sia.polarity_scores("Wow, NLTK is really powerful!")
{'neg': 0.0, 'neu': 0.295, 'pos': 0.705, 'compound': 0.8012}
```

You'll get back a dictionary of different scores. The negative, neutral, and positive scores are related: They all add up to 1 and can't be negative. The compound score is calculated differently. It's not just an average, and it can range from -1 to 1.

Now you'll put it to the test against real data using two different corpora. First, load the twitter_samples corpus into a list of strings, making a replacement to render URLs inactive to avoid accidental clicks:

```
tweets = [t.replace("://", "//") for t in nltk.corpus.twitter_samples.strings()]
```

Notice that you use a different corpus method, .strings(), instead of .words(). This gives you a list of raw tweets as strings.

Different corpora have different features, so you may need to use Python's help(), as in help(nltk.corpus.tweet_samples), or consult NLTK's documentation to learn how to use a given corpus.

Now use the .polarity_scores() function of your SentimentIntensityAnalyzer instance to classify tweets:

```
from random import shuffle


def is_positive(tweet: str) -> bool:
    """True if tweet has positive compound sentiment, False otherwise."""
    return sia.polarity_scores(tweet)["compound"] > 0


shuffle(tweets)
for tweet in tweets[:10]:
    print(">", is_positive(tweet), tweet)
```

In this case, is_positive() uses only the positivity of the compound score to make the call. You can choose any combination of VADER scores to tweak the classification to your needs.

Now take a look at the second corpus, movie_reviews. As the name implies, this is a collection of movie reviews. The special thing about this corpus is that it's already been classified. Therefore, you can use it to judge the accuracy of the algorithms you choose when rating similar texts.

Keep in mind that VADER is likely better at rating tweets than it is at rating long movie reviews. To get better results, you'll set up VADER to rate individual sentences within the review rather than the entire text.

Since VADER needs raw strings for its rating, you can't use .words() like you did earlier. Instead, make a list of the file IDs that the corpus uses, which you can use later to reference individual reviews:

```
positive_review_ids = nltk.corpus.movie_reviews.fileids(categories=["pos"])
```

```python
negative_review_ids = nltk.corpus.movie_reviews.fileids(categories=["neg"])

all_review_ids = positive_review_ids + negative_review_ids
```

.fileids() exists in most, if not all, corpora. In the case of movie_reviews, each file corresponds to a single review. Note also that you're able to filter the list of file IDs by specifying categories. This categorization is a feature specific to this corpus and others of the same type.

Next, redefine is_positive() to work on an entire review. You'll need to obtain that specific review using its file ID and then split it into sentences before rating:

```python
from statistics import mean


def is_positive(review_id: str) -> bool:

    """True if the average of all sentence compound scores is positive."""

    text = nltk.corpus.movie_reviews.raw(review_id)

    scores = [

        sia.polarity_scores(sentence)["compound"]

        for sentence in nltk.sent_tokenize(text)

    ]

    return mean(scores) > 0
```

.raw() is another method that exists in most corpora. By specifying a file ID or a list of file IDs, you can obtain specific data from the corpus. Here, you get a single review, then use nltk.sent_tokenize() to obtain a list of sentences from the review. Finally, is_positive() calculates the average compound score for all sentences and associates a positive result with a positive review.

You can take the opportunity to rate all the reviews and see how accurate VADER is with this setup:

```python
>>>

>>> shuffle(all_review_ids)

>>> correct = 0

>>> for review_id in all_review_ids:

...     if is_positive(review_id):

...         if review_id in positive_review_ids:

...             correct += 1

...     else:

...         if review_id in negative_review_ids:

...             correct += 1

...
```

>>> print(F"{correct / len(all_review_ids):.2%} correct")

64.00% correct

After rating all reviews, you can see that only 64 percent were correctly classified by VADER using the logic defined in is_positive().

A 64 percent accuracy rating isn't great, but it's a start. Have a little fun tweaking is_positive() to see if you can increase the accuracy.

In the next section, you'll build a custom classifier that allows you to use additional features for classification and eventually increase its accuracy to an acceptable level.

Customizing NLTK's Sentiment Analysis

NLTK offers a few built-in classifiers that are suitable for various types of analyses, including sentiment analysis. The trick is to figure out which properties of your dataset are useful in classifying each piece of data into your desired categories.

In the world of machine learning, these data properties are known as features, which you must reveal and select as you work with your data. While this tutorial won't dive too deeply into feature selection and feature engineering, you'll be able to see their effects on the accuracy of classifiers.

Selecting Useful Features

Since you've learned how to use frequency distributions, why not use them as a launching point for an additional feature?

By using the predefined categories in the movie_reviews corpus, you can create sets of positive and negative words, then determine which ones occur most frequently across each set. Begin by excluding unwanted words and building the initial category groups:

```
1unwanted = nltk.corpus.stopwords.words("english")
2unwanted.extend([w.lower() for w in nltk.corpus.names.words()])
3
4def skip_unwanted(pos_tuple):
5    word, tag = pos_tuple
6    if not word.isalpha() or word in unwanted:
7        return False
8    if tag.startswith("NN"):
9        return False
10   return True
11
12positive_words = [word for word, tag in filter(
13   skip_unwanted,
```

```
14    nltk.pos_tag(nltk.corpus.movie_reviews.words(categories=["pos"]))
15)]
16negative_words = [word for word, tag in filter(
17    skip_unwanted,
18    nltk.pos_tag(nltk.corpus.movie_reviews.words(categories=["neg"]))
19)]
```

This time, you also add words from the names corpus to the unwanted list on line 2 since movie reviews are likely to have lots of actor names, which shouldn't be part of your feature sets. Notice pos_tag() on lines 14 and 18, which tags words by their part of speech.

It's important to call pos_tag() before filtering your word lists so that NLTK can more accurately tag all words. skip_unwanted(), defined on line 4, then uses those tags to exclude nouns, according to NLTK's default tag set.

Now you're ready to create the frequency distributions for your custom feature. Since many words are present in both positive and negative sets, begin by finding the common set so you can remove it from the distribution objects:

```
positive_fd = nltk.FreqDist(positive_words)
negative_fd = nltk.FreqDist(negative_words)


common_set = set(positive_fd).intersection(negative_fd)


for word in common_set:
    del positive_fd[word]
    del negative_fd[word]


top_100_positive = {word for word, count in positive_fd.most_common(100)}
top_100_negative = {word for word, count in negative_fd.most_common(100)}
```

Once you're left with unique positive and negative words in each frequency distribution object, you can finally build sets from the most common words in each distribution. The amount of words in each set is something you could tweak in order to determine its effect on sentiment analysis.

This is one example of a feature you can extract from your data, and it's far from perfect. Looking closely at these sets, you'll notice some uncommon names and words that aren't necessarily positive or negative. Additionally, the other NLTK tools you've learned so far can be useful for building more features. One possibility is to leverage collocations that carry positive meaning, like the bigram "thumbs up!"

Here's how you can set up the positive and negative bigram finders:

```
unwanted = nltk.corpus.stopwords.words("english")
unwanted.extend([w.lower() for w in nltk.corpus.names.words()])


positive_bigram_finder = nltk.collocations.BigramCollocationFinder.from_words([
    w for w in nltk.corpus.movie_reviews.words(categories=["pos"])
    if w.isalpha() and w not in unwanted
])
negative_bigram_finder = nltk.collocations.BigramCollocationFinder.from_words([
    w for w in nltk.corpus.movie_reviews.words(categories=["neg"])
    if w.isalpha() and w not in unwanted
])
```

The rest is up to you! Try different combinations of features, think of ways to use the negative VADER scores, create ratios, polish the frequency distributions. The possibilities are endless!

Training and Using a Classifier

With your new feature set ready to use, the first prerequisite for training a classifier is to define a function that will extract features from a given piece of data.

Since you're looking for positive movie reviews, focus on the features that indicate positivity, including VADER scores:

```
def extract_features(text):
    features = dict()
    wordcount = 0
    compound_scores = list()
    positive_scores = list()


    for sentence in nltk.sent_tokenize(text):
        for word in nltk.word_tokenize(sentence):
            if word.lower() in top_100_positive:
                wordcount += 1
        compound_scores.append(sia.polarity_scores(sentence)["compound"])
        positive_scores.append(sia.polarity_scores(sentence)["pos"])


    # Adding 1 to the final compound score to always have positive numbers
```

```
    # since some classifiers you'll use later don't work with negative numbers.
    features["mean_compound"] = mean(compound_scores) + 1
    features["mean_positive"] = mean(positive_scores)
    features["wordcount"] = wordcount

    return features
```

extract_features() should return a dictionary, and it will create three features for each piece of text:

The average compound score

The average positive score

The amount of words in the text that are also part of the top 100 words in all positive reviews

In order to train and evaluate a classifier, you'll need to build a list of features for each text you'll analyze:

```
features = [
    (extract_features(nltk.corpus.movie_reviews.raw(review)), "pos")
    for review in nltk.corpus.movie_reviews.fileids(categories=["pos"])
]
features.extend([
    (extract_features(nltk.corpus.movie_reviews.raw(review)), "neg")
    for review in nltk.corpus.movie_reviews.fileids(categories=["neg"])
])
```

Each item in this list of features needs to be a tuple whose first item is the dictionary returned by extract_features and whose second item is the predefined category for the text. After initially training the classifier with some data that has already been categorized (such as the movie_reviews corpus), you'll be able to classify new data.

Training the classifier involves splitting the feature set so that one portion can be used for training and the other for evaluation, then calling .train():

```
>>>
>>> # Use 1/4 of the set for training
>>> train_count = len(features) // 4
>>> shuffle(features)
>>> classifier = nltk.NaiveBayesClassifier.train(features[:train_count])
>>> classifier.show_most_informative_features(10)
Most Informative Features
```

```
    wordcount = 2           pos : neg    =    4.1 : 1.0
    wordcount = 3           pos : neg    =    3.8 : 1.0
    wordcount = 0           neg : pos    =    1.6 : 1.0
    wordcount = 1           pos : neg    =    1.5 : 1.0
>>> nltk.classify.accuracy(classifier, features[train_count:])
0.668
```

Since you're shuffling the feature list, each run will give you different results. In fact, it's important to shuffle the list to avoid accidentally grouping similarly classified reviews in the first quarter of the list.

Adding a single feature has marginally improved VADER's initial accuracy, from 64 percent to 67 percent. More features could help, as long as they truly indicate how positive a review is. You can use classifier.show_most_informative_features() to determine which features are most indicative of a specific property.

To classify new data, find a movie review somewhere and pass it to classifier.classify(). You can also use extract_features() to tell you exactly how it was scored:

```
>>>
>>> new_review = ...
>>> classifier.classify(new_review)
>>> extract_features(new_review)
```

Was it correct? Based on the scoring output from extract_features(), what can you improve?

Feature engineering is a big part of improving the accuracy of a given algorithm, but it's not the whole story. Another strategy is to use and compare different classifiers.

Comparing Additional Classifiers

NLTK provides a class that can use most classifiers from the popular machine learning framework scikit-learn.

Many of the classifiers that scikit-learn provides can be instantiated quickly since they have defaults that often work well. In this section, you'll learn how to integrate them within NLTK to classify linguistic data.

Installing and Importing scikit-learn

Like NLTK, scikit-learn is a third-party Python library, so you'll have to install it with pip:

```
$ python3 -m pip install scikit-learn
```

After you've installed scikit-learn, you'll be able to use its classifiers directly within NLTK.

The following classifiers are a subset of all classifiers available to you. These will work within NLTK for sentiment analysis:

```
from sklearn.naive_bayes import (
    BernoulliNB,
```

```
    ComplementNB,
    MultinomialNB, )
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

With these classifiers imported, you'll first have to instantiate each one. Thankfully, all of these have pretty good defaults and don't require much tweaking.

To aid in accuracy evaluation, it's helpful to have a mapping of classifier names and their instances:

```
classifiers = {     "BernoulliNB": BernoulliNB(),
    "ComplementNB": ComplementNB(),
    "MultinomialNB": MultinomialNB(),
    "KNeighborsClassifier": KNeighborsClassifier(),
    "DecisionTreeClassifier": DecisionTreeClassifier(),
    "RandomForestClassifier": RandomForestClassifier(),
    "LogisticRegression": LogisticRegression(),
    "MLPClassifier": MLPClassifier(max_iter=1000),
    "AdaBoostClassifier": AdaBoostClassifier(), }
```

Now you can use these instances for training and accuracy evaluation.

Using scikit-learn Classifiers With NLTK

Since NLTK allows you to integrate scikit-learn classifiers directly into its own classifier class, the training and classification processes will use the same methods you've already seen, .train() and .classify().

You'll also be able to leverage the same features list you built earlier by means of extract_features(). To refresh your memory, here's how you built the features list:

```
features = [
    (extract_features(nltk.corpus.movie_reviews.raw(review)), "pos")
    for review in nltk.corpus.movie_reviews.fileids(categories=["pos"])
]
features.extend([
```

```
(extract_features(nltk.corpus.movie_reviews.raw(review)), "neg")
for review in nltk.corpus.movie_reviews.fileids(categories=["neg"]) ])
```

The features list contains tuples whose first item is a set of features given by extract_features(), and whose second item is the classification label from preclassified data in the movie_reviews corpus.

Since the first half of the list contains only positive reviews, begin by shuffling it, then iterate over all classifiers to train and evaluate each one:

```
>>>
>>> # Use 1/4 of the set for training
>>> train_count = len(features) // 4
>>> shuffle(features)
>>> for name, sklearn_classifier in classifiers.items():
...     classifier = nltk.classify.SklearnClassifier(sklearn_classifier)
...     classifier.train(features[:train_count])
...     accuracy = nltk.classify.accuracy(classifier, features[train_count:])
...     print(F"{accuracy:.2%} - {name}")
...
67.00% - BernoulliNB
66.80% - ComplementNB
66.33% - MultinomialNB
69.07% - KNeighborsClassifier
62.73% - DecisionTreeClassifier
66.60% - RandomForestClassifier
72.20% - LogisticRegression
73.13% - MLPClassifier
69.40% - AdaBoostClassifier
```

For each scikit-learn classifier, call nltk.classify.SklearnClassifier to create a usable NLTK classifier that can be trained and evaluated exactly like you’ve seen before with nltk.NaiveBayesClassifier and its other built-in classifiers. The .train() and .accuracy() methods should receive different portions of the same list of features.

Day-05: Labs and Practice activities for sentiments analysis on various datasets

This day is reserved for various data analysis and practice activities activities along with assessments.

Week 9: Time series analysis and forecasting

Day-01: Time Series – Introduction

NumPy

Numerical Python is a library used for scientific computing. It works on an N-dimensional array object and provides basic mathematical functionality such as size, shape, mean, standard deviation, minimum, maximum as well as some more complex functions such as linear algebraic functions and Fourier transform. You will learn more about these as we move ahead in this tutorial.

Pandas

This library provides highly efficient and easy-to-use data structures such as series, dataframes and panels. It has enhanced Python's functionality from mere data collection and preparation to data analysis. The two libraries, Pandas and NumPy, make any operation on small to very large dataset very simple. To know more about these functions,follow this tutorial.

SciPy

Science Python is a library used for scientific and technical computing. It provides functionalities for optimization, signal and image processing, integration, interpolation and linear algebra. This library comes handy while performing machine learning. We will discuss these functionalities as we move ahead in this tutorial.

Scikit Learn

This library is a SciPy Toolkit widely used for statistical modelling, machine learning and deep learning, as it contains various customizable regression, classification and clustering models. It works well with Numpy, Pandas and other libraries which makes it easier to use.

Statsmodels

Like Scikit Learn, this library is used for statistical data exploration and statistical modelling. It also operates well with other Python libraries.

Matplotlib

This library is used for data visualization in various formats such as line plot, bar graph, heat maps, scatter plots, histogram etc. It contains all the graph related functionalities required from plotting to labelling. We will discuss these functionalities as we move ahead in this tutorial.

Datetime

This library, with its two modules – datetime and calendar, provides all necessary datetime functionality for reading, formatting and manipulating time.

These libraries are very essential to start with machine learning with any sort of data.

Time Series – Data Processing and Visualization

Time Series is a sequence of observations indexed in equi-spaced time intervals. Hence, the order and continuity should be maintained in any time series. The dataset we will be using is a multi-variate time series having hourly data for approximately one year, for air quality in a significantly polluted Italian city. The dataset can be downloaded from the link given below: http://archive.ics.uci.edu/ml/datasets/air+quality

It is necessary to make sure that: • The time series is equally spaced, and • There are no redundant values or gaps in it. In case the time series is not continuous, we can upsample or downsample it.

Showing df.head()

```
import pandas

df = pandas.read_csv("AirQualityUCI.csv", sep = ";", decimal = ",") df = df.iloc[ : , 0:14]

len(df)

9471

df.head()

df.isna().sum()

df = df[df['Date'].notnull()]
```

For preprocessing the time series, we make sure there are no NaN(NULL) values in the dataset; if there are, we can replace them with either 0 or average or preceding or succeeding values. Replacing is a preferred choice over dropping so that the continuity of the time series is maintained. However, in our dataset the last few values seem to be NULL and hence dropping will not affect the continuity.

Dropping NaN(Not-a-Number)

```
df.isna().sum()

df = df[df['Date'].notnull()]

df = df[df['Date'].notnull()]
```

Converting to datetime object

```
df['DateTime'] = (df.Date) + ' ' + (df.Time)

print (type(df.DateTime[0]))

import datetime

df.DateTime = df.DateTime.apply(lambda x: datetime.datetime.strptime(x,'%d/%m/%Y %H.%M.%S'))
```

```
print (type(df.DateTime[0]))
```

Showing plots

```
df.index = df.DateTime

import matplotlib.pyplot as plt

plt.plot(df['T'])

plt.plot(df['C6H6(GT)'])
```

Box-plots are another useful kind of graphs that allow you to condense a lot of information about a dataset into a single graph. It shows the mean, 25% and 75% quartile and outliers of one or multiple variables. In the case when number of outliers is few and is very distant from the mean, we can eliminate the outliers by setting them to mean value or 75% quartile value.

Showing Boxplots

```
plt.boxplot(df[['T','C6H6(GT)']].values)
```

Time Series – Modeling

A time series has 4 components as given below:

- Level: It is the mean value around which the series varies.
- Trend: It is the increasing or decreasing behavior of a variable with time.
- Seasonality: It is the cyclic behavior of time series.
- Noise: It is the error in the observations added due to environmental factors.

Time Series Modeling Techniques

To capture these components, there are a number of popular time series modelling techniques. This section gives a brief introduction of each technique, however we will discuss about them in detail in the upcoming chapters:

Naïve Methods

These are simple estimation techniques, such as the predicted value is given the value equal to mean of preceding values of the time dependent variable, or previous actual value. These are used for comparison with sophisticated modelling techniques.

Auto Regression

Auto regression predicts the values of future time periods as a function of values at previous time periods. Predictions of auto regression may fit the data better than that of naïve methods, but it may not be able to account for seasonality.

ARIMA Model

An Auto-Regressive Integrated Moving-Average(ARIMA) models the value of a variable as a linear function of previous values and residual errors at previous time steps of a stationary time series. However, the real world data may be non-stationary and have seasonality, thus Seasonal-ARIMA and Fractional-ARIMA were developed. ARIMA works on univariate time series, to handle multiple variables VARIMA was introduced.

Exponential Smoothing

It models the value of a variable as an exponential weighted linear function of previous values. This statistical model can handle trend and seasonality as well.

LSTM

Long Short-Term Memory model (LSTM) is a recurrent neural network which is used for time series to account for long term dependencies. It can be trained with large amount of data to capture the trends in multi-variate time series.

Time Series – Parameter Calibration

Any statistical or machine learning model has some parameters which greatly influence how the data is modeled. For example, ARIMA has p, d, q values. These parameters are to be decided such that the error between actual values and modeled values is minimum.

Parameter calibration is said to be the most crucial and time-consuming task of model fitting. Hence, it is very essential for us to choose optimal parameters.

Methods for Calibration of Parameters

There are various ways to calibrate parameters. This section talks about some of them in

detail.

Hit-and-try

One common way of calibrating models is hand calibration, where you start by visualizing the time-series and intuitively try some parameter values and change them over and over until you achieve a good enough fit. It requires a good understanding of the model we are trying. For ARIMA model, hand calibration is done with the help of auto-correlation plot for 'p' parameter, partial auto-correlation plot for 'q' parameter and ADF-test to confirm the stationarity of time-series and setting 'd' parameter. We will discuss all these in detail in the coming chapters.

Grid Search

Another way of calibrating models is by grid search, which essentially means you try building a model for all possible combinations of parameters and select the one with minimum error. This is time-consuming and hence is useful when number of parameters to be calibrated and range of values they take are fewer as this involves multiple nested for loops.

Genetic Algorithm

Genetic algorithm works on the biological principle that a good solution will eventually evolve to the most 'optimal' solution. It uses biological operations of mutation, cross-over and selection to finally reach to an optimal solution.

For further knowledge you can read about other parameter optimization techniques like

Bayesian optimization and Swarm optimization.

Time Series – Naïve Methods

Naïve Methods such as assuming the predicted value at time 't' to be the actual value of the variable at time 't-1' or rolling mean of series, are used to weigh how well do the statistical models and machine learning models can perform and emphasize their need. In this chapter, let us try these models on one of the features of our time-series data. First we shall see the mean of the 'temperature' feature of our data and the deviation around it. It is also useful to see maximum and minimum temperature values. We can use the functionalities of numpy library here.

Showing statistics

import numpy print ('Mean: ',numpy.mean(df['T']), '; Standard Deviation: ',numpy.std(df['T']),'; \nMaximum Temperature: ',max(df['T']),'; Minimum Temperature: ',min(df['T']))

We have the statistics for all 9357 observations across equi-spaced timeline which are useful for us to understand the data. Now we will try the first naïve method, setting the predicted value at present time equal to actual value at previous time and calculate the root mean squared error(RMSE) for it to quantify the performance of this method.

Showing 1st naïve method

Before executing following commands first install scikit-learn in notebook:

```
!pip install scikit-learn
```

df['T'] df['T_t-1'] = df['T'].shift(1)

df_naive = df[['T','T_t-1']][1:]

from sklearn import metrics

from math import sqrt

df['T_rm'] = df['T'].rolling(3).mean().shift(1)

df_naive = df[['T','T_rm']].dropna()

true = df_naive['T']

prediction = df_naive['T_t-1']

error = sqrt(metrics.mean_squared_error(true,prediction))

print ('RMSE for Naive Method 1: ', error)

Let us see the next naïve method, where predicted value at present time is equated to the mean of the time periods preceding it. We will calculate the RMSE for this method too.

Showing 2nd naïve method

```
df['T_rm'] = df['T'].rolling(3).mean().shift(1)

df_naive = df[['T','T_rm']].dropna()


true = df_naive['T']

prediction = df_naive['T_rm']

error = sqrt(metrics.mean_squared_error(true,prediction))

print ('RMSE for Naive Method 2: ', error)
```

Here, you can experiment with various number of previous time periods also called 'lags' you want to consider, which is kept as 3 here. In this data it can be seen that as you increase the number of lags and error increases. If lag is kept 1, it becomes same as the naïve method used earlier.

Points to Note

- You can write a very simple function for calculating root mean squared error. Here, we have used the mean squared error function from the package 'sklearn' and then taken its square root.

- In pandas df['column_name'] can also be written as df.column_name, however for this dataset df.T will not work the same as df['T'] because df.T is the function for transposing a dataframe. So use only df['T'] or consider renaming this column before using the other syntax.

Time Series – Auto Regression

For a stationary time series, an auto regression models sees the value of a variable at time 't' as a linear function of values 'p' time steps preceding it. Mathematically it can be written as:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \epsilon_t$$

Where, 'p' is the auto-regressive trend parameter

ϵ_t is white noise, and

$yt{-}1$, $yt{-}2$ …$yt{-}p$ denote the value of variable at previous time periods.

The value of p can be calibrated using various methods. One way of finding the apt value of 'p' is plotting the auto-correlation plot.

Note: We should separate the data into train and test at 8:2 ratio of total data available prior to doing any analysis on the data because test data is only to find out the accuracy of our model and assumption is, it is not available to us until after predictions have been made. In case of time series, sequence of data points is very essential so one should keep in mind not to lose the order during splitting of data. An auto-correlation plot or a correlogram shows the relation of a variable with itself at prior time steps. It makes use of Pearson's correlation and shows the correlations within 95% confidence interval. Let's see how it looks like for 'temperature' variable of our data.

Showing ACP

```
split = len(df) - int(0.2*len(df))

train, test = df['T'][0:split], df['T'][split:]

from statsmodels.graphics.tsaplots import plot_acf

plot_acf(train, lags = 100)

plt.show()
```

All the lag values lying outside the shaded blue region are assumed to have a correlation.

Time Series – Moving Average

For a stationary time series, a moving average model sees the value of a variable at time 't' as a linear function of residual errors from 'q' time steps preceding it. The residual error is calculated by comparing the value at the time 't' to moving average of the values preceding. Mathematically it can be written as:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \ldots + \phi_p y_{t-p} + \epsilon_t$$

Where 'q' is the moving-average trend parameter

ϵ_t is white noise, and $\epsilon t-1$, $\epsilon t-2$ … $\epsilon t-q$are the error terms at previous time periods. Value of 'q' can be calibrated using various methods. One way of finding the apt value of 'q' is plotting the partial auto-correlation plot. A partial auto-correlation plot shows the relation of a variable with itself at prior time steps with indirect correlations removed, unlike auto-correlation plot which shows direct as well as indirect correlations, let's see how it looks like for 'temperature' variable of our data.

Showing PACP

```
from statsmodels.graphics.tsaplots import plot_pacf

plot_pacf(train, lags = 100)

plt.show()
```

A partial auto-correlation is read in the same way as a correlogram.

Time Series – ARIMA

We have already understood that for a stationary time series a variable at time 't' is a linear function of prior observations or residual errors. Hence it is time for us to combine the two and have an Auto-regressive moving average (ARMA) model.

However, at times the time series is not stationary, i.e the statistical properties of a series like mean, variance changes over time. And the statistical models we have studied so far assume the time series to be stationary, therefore, we can include a pre-processing step of differencing the time series to make it stationary. Now, it is important for us to find out whether the time series we are dealing with is stationary or not.

Various methods to find the stationarity of a time series are looking for seasonality or trend in the plot of time series, checking the difference in mean and variance for various time periods, Augmented Dickey-Fuller (ADF) test, KPSS test, Hurst's exponent etc. Let us see whether the 'temperature' variable of our dataset is a stationary time series or not using ADF test.

```
from statsmodels.tsa.stattools import adfuller

result = adfuller(train)

print('ADF Statistic: %f' % result[0])

print('p-value: %f' % result[1])

print('Critical Values:')

for key, value In result[4].items()

        print('\t%s: %.3f' % (key, value))
```

Now that we have run the ADF test, let us interpret the result. First we will compare the ADF Statistic with the critical values, a lower critical value tells us the series is most likely non-stationary. Next, we see the p-value. A p-value greater than 0.05 also suggests that the time series is non-stationary. Alternatively, p-value less than or equal to 0.05, or ADF Statistic less than critical values suggest the time series is stationary.

Hence, the time series we are dealing with is already stationary. In case of stationary time series, we set the 'd' parameter as 0. We can also confirm the stationarity of time series using Hurst exponent.

```
import hurst

H, c,data = hurst.compute_Hc(train)

print("H = {:.4f}, c = {:.4f}".format(H,c))
```

The value of H0.5 shows persistent behavior or a trending series. H=0.5 shows random walk/Brownian motion. The value of H< 0.5, confirming that our series is stationary.

For non-stationary time series, we set ‘d’ parameter as 1. Also, the value of the autoregressive trend parameter ‘p’ and the moving average trend parameter ‘q’, is calculated on the stationary time series i.e by plotting ACP and PACP after differencing the time series. ARIMA Model, which is characterized by 3 parameter, (p,d,q) are now clear to us, so let us model our time series and predict the future values of temperature.

```
from statsmodels.tsa.arima_model import ARIMA

model = ARIMA(train.values, order=(5, 0, 2))

model_fit = model.fit(disp=False)

predictions = model_fit.predict(len(test))

test_ = pandas.DataFrame(test)

test_['predictions'] = predictions[0:1871]

plt.plot(df['T'])

plt.plot(test_.predictions)

plt.show()

error = sqrt(metrics.mean_squared_error(test.values,predictions[0:1871]))

print ('Test RMSE for ARIMA: ', error)
```

Time Series – Variations of ARIMA

In the previous chapter, we have now seen how ARIMA model works, and its limitations that it cannot handle seasonal data or multivariate time series and hence, new models were introduced to include these features.

A glimpse of these new models is given here:

Vector Auto-Regression (VAR)

It is a generalized version of auto regression model for multivariate stationary time series.

It is characterized by ‘p’ parameter.

Vector Moving Average (VMA)

It is a generalized version of moving average model for multivariate stationary time series.

It is characterized by ‘q’ parameter.

Vector Auto Regression Moving Average (VARMA)

It is the combination of VAR and VMA and a generalized version of ARMA model for multivariate stationary time series. It is characterized by 'p' and 'q' parameters. Much like, ARMA is capable of acting like an AR model by setting 'q' parameter as 0 and as a MA model by setting 'p' parameter as 0, VARMA is also capable of acting like an VAR model by setting 'q' parameter as 0 and as a VMA model by setting 'p' parameter as 0.

```
from statsmodels.tsa.statespace.varmax import VARMAX

model = VARMAX(train_multi, order = (2,1))

model_fit = model.fit()

plt.plot(train_multi['T'])

plt.plot(test_multi['T'])

plt.plot(predictions_multi.iloc[:,0:1], '--')

plt.show()

plt.plot(train_multi['C6H6(GT)'])

plt.plot(test_multi['C6H6(GT)'])

plt.plot(predictions_multi.iloc[:,1:2], '--')

plt.show()
```

The above code shows how VARMA model can be used to model multivariate time series, although this model may not be best suited on our data.

VARMA with Exogenous Variables (VARMAX)

It is an extension of VARMA model where extra variables called covariates are used to model the primary variable we are interested it.

Seasonal Auto Regressive Integrated Moving Average (SARIMA)

This is the extension of ARIMA model to deal with seasonal data. It divides the data into seasonal and non-seasonal components and models them in a similar fashion. It is characterized by 7 parameters, for non-seasonal part (p,d,q) parameters same as for ARIMA model and for seasonal part (P,D,Q,m) parameters where 'm' is the number of seasonal periods and P,D,Q are similar to parameters of ARIMA model. These parameters can be calibrated using grid search or genetic algorithm.

SARIMA with Exogenous Variables (SARIMAX)

This is the extension of SARIMA model to include exogenous variables which help us to model the variable we are interested in.

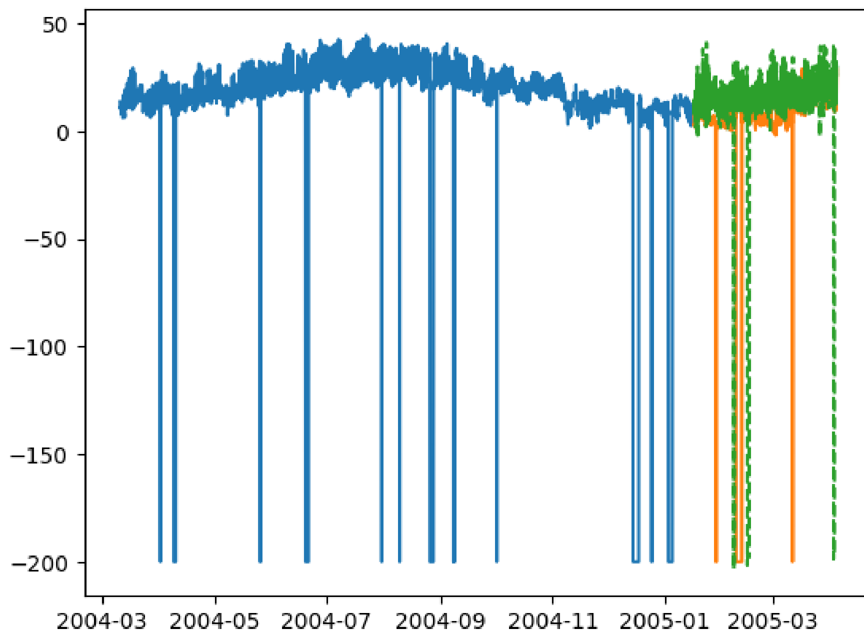
It may be useful to do a co-relation analysis on variables before putting them as exogenous

variables.

```
from scipy.stats.stats import pearsonr

x=train_multi['T'].values

y=train_multi['C6H6(GT)'].values

corr , p = pearsonr(x,y)

print ('Corelation Coefficient =', corr,'\nP-Value =',p)
```

Pearson's Correlation shows a linear relation between 2 variables, to interpret the results, we first look at the p-value, if it is less that 0.05 then the value of coefficient is significant, else the value of coefficient is not significant. For significant p-value, a positive value of correlation coefficient indicates positive correlation, and a negative value indicates a negative correlation. Hence, for our data, 'temperature' and 'C6H6' seem to have a highly positive correlation. Therefore, we will be modelling temperature and will give 'C6H6' as exogenous variable to SARIMAX model.

```
from statsmodels.tsa.statespace.sarimax import SARIMAX



model = SARIMAX(x, exog = y, order = (2, 0, 2), seasonal_order = (2, 0, 1, 4),enforce_stationarity=False,

enforce_invertibility = False)

model_fit = model.fit(disp = False)

y_ = test_multi['C6H6(GT)'].values

predicted = model_fit.predict(exog=y_)

test_multi_ = pandas.DataFrame(test)

test_multi_['predictions'] = predicted[0:1871]

plt.plot(train_multi['T'])

plt.plot(test_multi_['T'])

plt.plot(test_multi_.predictions, '--')
```



The predictions here seem to take larger variations now as opposed to univariate ARIMA modelling. Needless to say, SARIMAX can be used as an ARX, MAX, ARMAX or ARIMAX model by setting only the corresponding parameters to non-zero values.

Fractional Auto Regressive Integrated Moving Average (FARIMA) At times, it may happen that our series is not stationary, yet differencing with 'd' parameter taking the value 1 may over-difference it. So, we need to difference the time series using a fractional value.

In the world of data science there is no one superior model, the model that works on your data depends greatly on your dataset. Knowledge of various models allows us to choose one that work on our data and experimenting with that model to achieve the best results. And results should be seen as plot as well as error metrics, at times a small error may also be bad, hence, plotting and visualizing the results is essential.

Day-02:Time Series – Exponential Smoothing

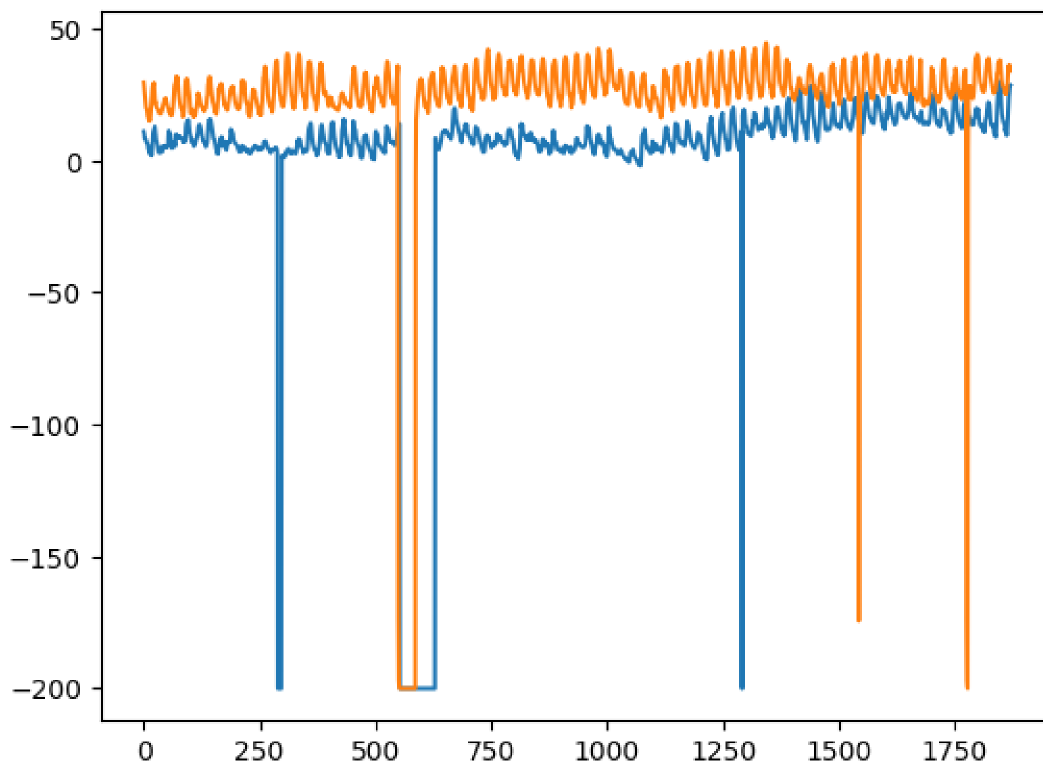
Simple Exponential Smoothing Exponential Smoothing is a technique for smoothing univariate time-series by assigning exponentially decreasing weights to data over a time period. Mathematically, the value of variable at time 't+1' given value at time t, y_(t+1|t) is defined as:

$$y_{t+1|t} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \ldots + y_1$$

where, 0≤ α ≤1 is the smoothing parameter, and y1,...,yt are previous values of network traffic at times 1, 2, 3, … ,t. This is a simple method to model a time series with no clear trend or seasonality. But exponential smoothing can also be used for time series with trend and seasonality. Triple Exponential Smoothing Triple Exponential Smoothing (TES) or Holt's Winter method, applies exponential smoothing three times - level smoothing lt , trend smoothing bt , and seasonal smoothing st , with α, $\beta*$ and γ as smoothing parameters with 'm' as the frequency of the seasonality, i.e. the number of seasons in a year. According to the nature of the seasonal component, TES has two categories: • Holt-Winter's Additive Method: When the seasonality is additive in nature. • Holt-Winter's Multiplicative Method: When the seasonality is multiplicative in nature. For non-

seasonal time series, we only have trend smoothing and level smoothing, which is called Holt's Linear Trend Method. Let's try applying triple exponential smoothing on our data.

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

model = ExponentialSmoothing(train.values )

model_fit = model.fit()

predictions_ = model_fit.predict(len(test))

plt.plot(test.values)

plt.plot(predictions_[1:1871])
```



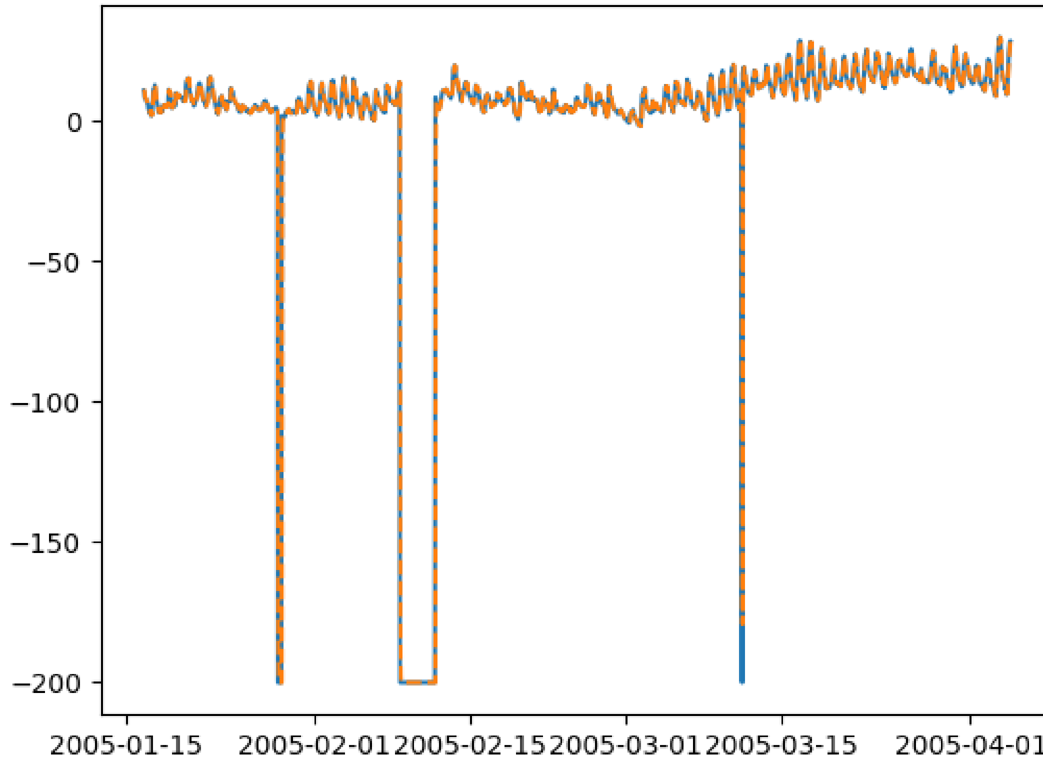
Here, we have trained the model once with training set and then we keep on making predictions. A more realistic approach is to re-train the model after one or more time step(s). As we get the prediction for time 't+1' from training data 'til time 't', the next prediction for time 't+2' can be made using the training data 'til time 't+1' as the actual value at 't+1' will be known then. This methodology of making predictions for one or more future steps and then re-training the model is called rolling forecast or walk forward validation.

Time Series – Walk Forward Validation

In time series modelling, the predictions over time become less and less accurate and hence it is a more realistic approach to re-train the model with actual data as it gets available for further predictions. Since training of statistical models are not time consuming, walk-forward

validation is the most preferred solution to get most accurate results. Let us apply one step walk forward validation on our data and compare it with the results we got earlier.

```
import numpy

prediction = []

data = train.values

for t in test.values:

    model = (ExponentialSmoothing(data).fit())

    y = model.predict()

    prediction.append(y[0])

    data = numpy.append(data, t)

test_ = pandas.DataFrame(test)

test_['predictionswf'] = prediction

plt.plot(test_['T'])

plt.plot(test_.predictionswf, '--')

plt.show()
```



error = sqrt(metrics.mean_squared_error(test.values,prediction))

print ('Test RMSE for Triple Exponential Smoothing with Walk-Forward Validation: ', error)

We can see that our model performs significantly better now. In fact, the trend is followed so closely that on the plot predictions are overlapping with the actual values. You can try applying walk-forward validation on ARIMA models too.

Day-03:Time Series – LSTM Model

Now, we are familiar with statistical modelling on time series, but machine learning is all the rage right now, so it is essential to be familiar with some machine learning models as well. We shall start with the most popular model in time series domain – Long Short-term Memory model.

LSTM is a class of recurrent neural network. So before we can jump to LSTM, it is essential to understand neural networks and recurrent neural networks.

Neural Networks

An artificial neural network is a layered structure of connected neurons, inspired by biological neural networks. It is not one algorithm but combinations of various algorithms

which allows us to do complex operations on data.

Recurrent Neural Networks

It is a class of neural networks tailored to deal with temporal data. The neurons of RNN have a cell state/memory, and input is processed according to this internal state, which is achieved with the help of loops with in the neural network. There are recurring module(s) of 'tanh' layers in RNNs that allow them to retain information. However, not for a long time, which is why we need LSTM models.

LSTM

It is special kind of recurrent neural network that is capable of learning long term dependencies in data. This is achieved because the recurring module of the model has a combination of four layers interacting with each other.

An LSTM module has a cell state and three gates which provides them with the power to selectively learn, unlearn or retain information from each of the units. The cell state in LSTM helps the information to flow through the units without being altered by allowing only a few linear interactions. Each unit has an input, output and a forget gate which can add or remove the information to the cell state. The forget gate decides which information from the previous cell state should be forgotten for which it uses a sigmoid function. The input gate controls the information flow to the current cell state using a point-wise multiplication operation of 'sigmoid' and 'tanh' respectively. Finally, the output gate decides which information should be passed on to the next hidden state. Now that we have understood the internal working of LSTM model, let us implement it. To understand the implementation of LSTM, we will start with a simple example – a straight line. Let us see, if LSTM can learn the relationship of a straight line and predict it. First let us create the dataset depicting a straight line.

```
x = numpy.arange (1,500,1)

y = 0.4 * x + 30

plt.plot(x,y)

trainx, testx = x[0:int(0.8*(len(x)))], x[int(0.8*(len(x))):]

trainy, testy = y[0:int(0.8*(len(y)))], y[int(0.8*(len(y))):]

train = numpy.array(list(zip(trainx,trainy)))

test = numpy.array(list(zip(trainx,trainy)))
```

```
def create_dataset(n_X, look_back):
    dataX, dataY = [], []
    for i in range(len(n_X)-look_back):
        a = n_X[i:(i+look_back), ]
        dataX.append(a)
        dataY.append(n_X[i + look_back, ])
    return numpy.array(dataX), numpy.array(dataY)
look_back = 1
trainx,trainy = create_dataset(train, look_back)
testx,testy = create_dataset(test, look_back)
trainx = numpy.reshape(trainx, (trainx.shape[0], 1, 2))
testx = numpy.reshape(testx, (testx.shape[0], 1, 2))
```

Now we will train our model

Small batches of training data are shown to network, one run of when entire training data is shown to the model in batches and error is calculated is called an epoch. The epochs are to be run ‘til the time the error is reducing.

Note: First Install Keras and Tensorflow libaries

```
!pip install keras
!anaconda create -n tensorflow python=3.11
!activate tensorflow
!pip install --ignore-installed --upgrade tensorflow
```

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
model = Sequential()
model.add(LSTM(256, return_sequences=True, input_shape=(trainx.shape[1], 2)))
```

```
model.add(LSTM(128,input_shape=(trainx.shape[1], 2)))
model.add(Dense(2))
model.compile(loss='mean_squared_error', optimizer = 'adam')
model.fit(trainx, trainy, epochs=2000, batch_size=10, verbose=2, shuffle=False)
model.save_weights('LSTMBasic1.h5')
```

Now, we should try and model a sine or cosine wave in a similar fashion. You can run the code given below and play with the model parameters to see how the results change.

```
model.load_weights('LSTMBasic1.h5')
predict = model.predict(testx)
```

Now let's see what our predictions look like.

```
x = numpy.arange (1,500,1)
y = numpy.sin(x)
plt.plot(x,y)
trainx, testx = x[0:int(0.8*(len(x)))], x[int(0.8*(len(x))):]
trainy, testy = y[0:int(0.8*(len(y)))], y[int(0.8*(len(y))):]
train = numpy.array(list(zip(trainx,trainy)))
test = numpy.array(list(zip(trainx,trainy)))
look_back = 1
trainx,trainy = create_dataset(train, look_back)
testx,testy = create_dataset(test, look_back)
trainx = numpy.reshape(trainx, (trainx.shape[0], 1, 2))
testx = numpy.reshape(testx, (testx.shape[0], 1, 2))
model = Sequential()
model.add(LSTM(512, return_sequences = True, input_shape = (trainx.shape[1],
2)))
```

```
model.add(LSTM(256,input_shape = (trainx.shape[1], 2)))

model.add(Dense(2))

model.compile(loss = 'mean_squared_error', optimizer = 'adam')

model.fit(trainx, trainy, epochs = 2000, batch_size = 10, verbose = 2, shuffle
= False)

model.save_weights('LSTMBasic2.h5')

model.load_weights('LSTMBasic2.h5')

predict = model.predict(testx)

plt.plot(trainx.reshape(398,2)[:,0:1], trainx.reshape(398,2)[:,1:2])

plt.plot(predict[:,0:1], predict[:,1:2])
```

Day-04: Time Series – Error Metrics

It is important for us to quantify the performance of a model to use it as a feedback and comparison. In this tutorial we have used one of the most popular error metric root mean squared error. There are various other error metrics available. This chapter discusses them in brief.

Mean Square Error

It is the average of square of difference between the predicted values and true values. Sklearn provides it as a function. It has the same units as the true and predicted values squared and is always positive.

$$MSE = \frac{1}{n}\sum_{t=1}^{n}(y'_t - y_t)^2$$

Where $y't$ is the predicted value,

yt is the actual value, and

n is the total number of values in test set.

It is clear from the equation that MSE is more penalizing for larger errors, or the outliers.

Root Mean Square Error

It is the square root of the mean square error. It is also always positive and is in the range of the data.

Root Mean Square Error

It is the square root of the mean square error. It is also always positive and is in the range of the data.

$$RMSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(y'_t - y_t)^2}$$

Where, y'_t is predicted value

y_t is actual value, and

n is total number of values in test set.

It is in the power of unity and hence is more interpretable as compared to MSE. RMSE is also more penalizing for larger errors. We have used RMSE metric in our tutorial.

Mean Absolute Error

It is the average of absolute difference between predicted values and true values. It has the same units as predicted and true value and is always positive.

$$MAE = \frac{1}{n}\sum_{t=1}^{t=n}|y'_t - y_t|$$

Where, y'_t is predicted value,

y_t is actual value and n is total number of values in test set.

However, the disadvantage of using this error is that the positive error and negative errors can offset each other. Hence mean absolute percentage error is used.

Mean Absolute Percentage Error

It is the percentage of average of absolute difference between predicted values and true values, divided by the true value.

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\frac{|y'_t - y_t|}{y_t} * 100\ \%$$

Where $y't$ is predicted value
yt is actual value, and
n is total number of values in test set.

Day-05: Time Series – Applications

We discussed time series analysis in this tutorial, which has given us the understanding that time series models first recognize the trend and seasonality from the existing observations and then forecast a value based on this trend and seasonality. Such analysis is useful in various fields such as:

- **Financial Analysis:** It includes sales forecasting, inventory analysis, stock market analysis, price estimation.
- **Weather Analysis:** It includes temperature estimation, climate change, seasonal shift recognition, weather forecasting.
- **Network Data Analysis:** It includes network usage prediction, anomaly or intrusion detection, predictive maintenance.
- **Healthcare Analysis:** It includes census prediction, insurance benefits prediction, patient monitoring.

Time Series – Further Scope

Machine learning deals with various kinds of problems. In fact, almost all fields have a scope to be automatized or improved with the help of machine learning. A few such problems on which a great deal of work is being done are given below.

Time Series Data

This is the data which changes according to time, and hence time plays a crucial role in it, which we largely discussed in this tutorial.

Non-Time Series Data

It is the data independent of time, and a major percentage of ML problems are on non time series data. For simplicity, we shall categorize it further as:

- **Numerical Data:** Computers, unlike humans, only understand numbers, so all kinds of data ultimately is converted to numerical data for machine learning, for example, image data is converted to (r,b,g) values, characters are converted to ASCII codes or words are indexed to numbers, speech data is converted to mfcc files containing numerical data.
- **Image Data:** Computer vision has revolutionized the world of computers, it has
- various applications in the field of medicine, satellite imaging etc.
- **Text Data:** Natural Language Processing (NLP) is used for text classification, paraphrase detection and language summarization. This is what makes Google and Facebook smart.
- **Speech Data:** Speech Processing involves speech recognition and sentimental understanding. It plays a crucial role in imparting to computers human-like qualities.

Now, Stdents are required to explore the projects related to time series analysis and forecasting and presents their proposal for further discussion and refinement.

Week 10 & 11: Data Analysis Projects

In these two weeks there will be discussion and working on Advanced Data Analysis Techniques and working on project in group of (2-3) students. Topics listed below will be discussed based on the profile of the students, learning and coverage. The following topics will be discussed, and notes will be shared with students in soft format. The projects will be developed using google labs for collaborative working of groups.

Advanced Analytics is the autonomous or semi-autonomous examination of data or content using sophisticated techniques and tools, typically beyond those of traditional business intelligence (BI), to discover deeper insights, make predictions, or generate recommendations.

Advanced analytic techniques include data/text mining, machine learning, pattern matching, forecasting, visualization, semantic analysis, sentiment analysis, network and cluster analysis, multivariate statistics, graph analysis, simulation, complex event processing, and neural networks.

List of resources & Acknoledgements:

1. Starting Out with Python[4th Globa lED] by Tony Gaddis
2. Python for Data Analysis by Wes McKinney
3. Python Data Science Handbook, Essential Tools for Working with Data by Beijing Boston
4. Data Science by Lillian Pierson 3rd Edition
5. Python Data Visualization Cookbook by Igor Milovanović
6. Natural Language Processing with Python by Steven Bird, Ewan Klein, and Edward Loper
7. https://realpython.com/python-data-visualization-bokeh/
8. https://github.com/osanchez2323/Portfolio/blob/master/NBA%20Draft%20Analysis/
9. https://docs.bokeh.org/en/latest/
10. https://pandas.pydata.org/
11. https://matplotlib.org/
12. https://dash.plotly.com/