

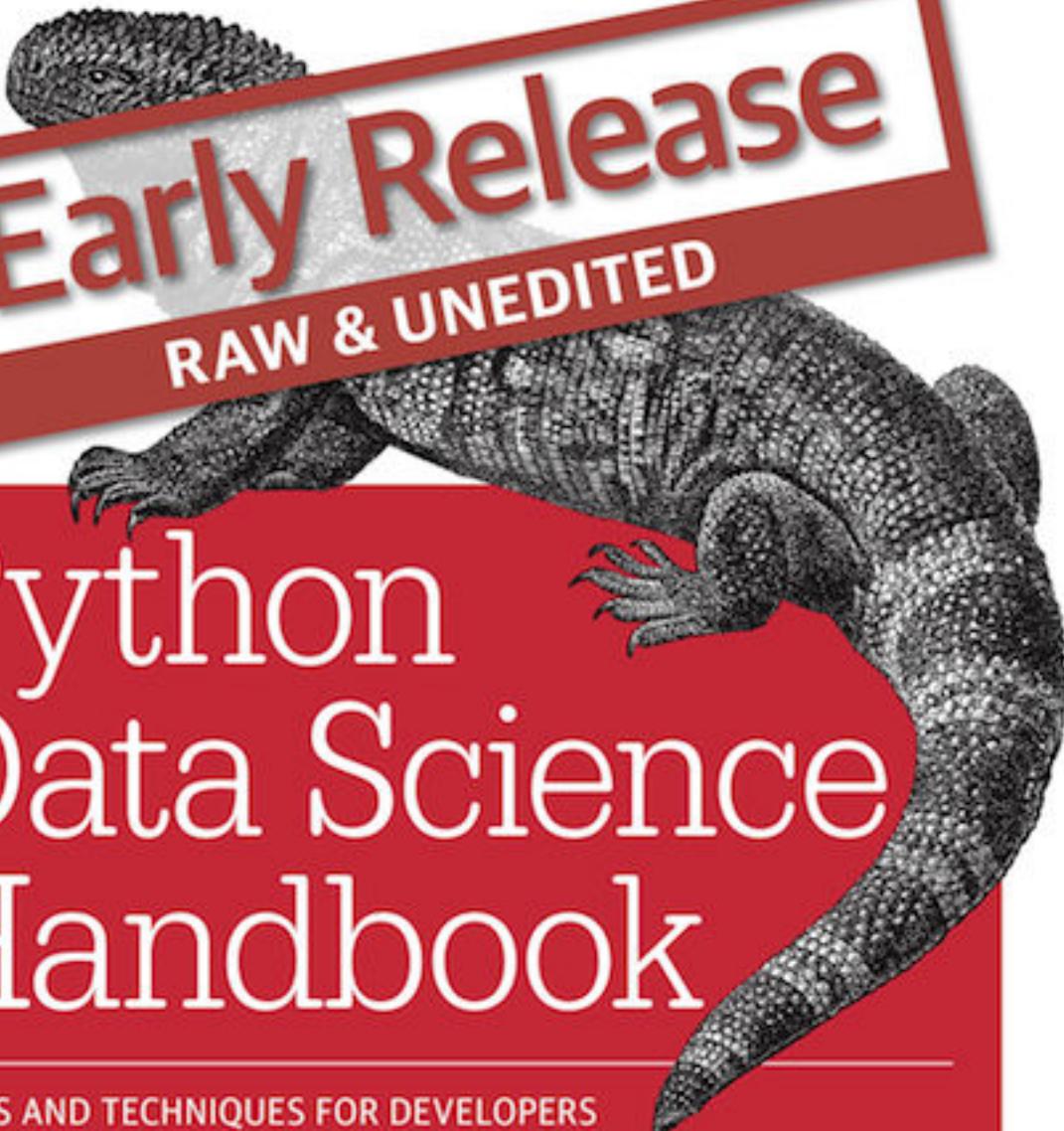
O'REILLY®

Early Release

RAW & UNEDITED

Python Data Science Handbook

TOOLS AND TECHNIQUES FOR DEVELOPERS



Jake VanderPlas

Python Data Science Handbook

Jake VanderPlas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Python Data Science Handbook

by Jake VanderPlas

Copyright © 2015 Jake VanderPlas. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meghan Blanchette

Production Editor: FILL IN PRODUCTION EDITOR
TOR

Copyeditor: FILL IN COPYEDITOR

Proofreader: FILL IN PROOFREADER

Indexer: FILL IN INDEXER

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

March 2016: First Edition

Revision History for the First Edition

2015-08-05: First Release

2015-12-01: Second Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491912058> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Python Data Science Handbook*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author(s) have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author(s) disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-91205-8

[FILL IN]

Table of Contents

Preface.....	xv
1. A Whirlwind Tour of the Python Language.....	23
Python is Glue	23
The Zen of Python	24
How to Run Python Code	24
The Python Interpreter	25
A Quick Tour of Python Language Syntax	27
Comments are marked by #	27
End-of-line Terminates a Statement	28
Semicolon can Optionally Terminate a Statement	28
Indentation: Whitespace Matters!	28
Whitespace <i>Within</i> Lines Does Not Matter	30
Parentheses are for Grouping or Calling	30
Finishing Up and Learning More	31
Sidebar: Note on the <code>print()</code> Function	31
Basic Python Semantics: Variables and Objects	32
Python Variables are Pointers	32
Everything is an Object	34
Basic Python Semantics: Operators	35
Arithmetic Operations	35
Bitwise Operations	36
Assignment Operations	37
Comparison Operations	38
Boolean Operations	39
Identity and Membership Operators	39
Summary	41
Built-in Types: Simple Values	41

Numeric Types	41
String Type	45
None Type	46
Boolean Type	46
Built-in Data Structures	47
Lists	47
Tuples	51
Dictionaries	52
Sets	53
More Specialized Data Structures	54
Control Flow	54
Conditional Statements: <code>if-elif-else:</code>	54
<code>for</code> loops	55
<code>while</code> loops	56
<code>break</code> and <code>continue</code> : Fine Tuning Your Loops	56
Loops With an <code>else</code> Block	57
Defining and Using Functions	57
Using Functions	58
Defining Functions	58
Default Argument Values	59
<code>*args</code> and <code>**kwargs</code> : Flexible Arguments	59
Anonymous (<code>lambda</code>) Functions	60
Errors and Exceptions	61
Runtime Errors	61
Catching Exceptions: <code>try</code> and <code>except</code>	63
Raising Exceptions: <code>raise</code>	64
Advanced Topics	66
<code>try...except...else...finally</code>	67
Iterators	70
Iterating over lists	70
<code>range()</code> : A List is Not Always a List	71
Useful Iterators	72
Advanced Iterators: <code>itertools</code>	74
List Comprehensions	75
Basic List Comprehensions	75
Multiple Iteration	76
Conditionals on the Iterator	76
Conditionals on the Value	76
Other Types of Comprehensions	77
Dict Comprehension	78
Generator Expressions	78
Generators	79

List Comprehensions vs Generator Expressions	79
Generator Functions: <code>yield</code>	81
Example: Prime Number Generator	82
Modules and Packages	83
Loading Modules: the <code>import</code> Statement	83
Python's Standard Library	85
Third-party modules	86
String Manipulation and Regular Expressions	86
Simple String Manipulation in Python	86
Format Strings	90
Flexible Pattern Matching with Regular Expressions	92
Further Python Resources	98
More Advanced Python Language Features	99
More Built-in Modules	99
More Third-Party Modules	99
2. IPython: Beyond Normal Python.....	101
Shell or Notebook?	102
Launching the IPython Shell	102
Launching the IPython Notebook	102
Help and Documentation in IPython	103
Accessing Documentation with "?"	104
Accessing Source Code with "??"	105
Exploring Modules with Tab-Completion	106
Keyboard Shortcuts in the IPython Shell	108
Navigation shortcuts	108
Text Entry Shortcuts	109
Command History Shortcuts	109
Miscellaneous Shortcuts	110
IPython Magic Commands	111
Pasting Code Blocks: <code>%paste</code> and <code>%cpaste</code>	111
Running External Code: <code>%run</code>	112
Timing Code Execution: <code>%timeit</code>	113
Help on Magic Functions: <code>?</code> , <code>%magic</code> , and <code>%lsmagic</code>	113
Input and Output History	114
IPython's <code>In</code> and <code>Out</code> Objects	114
Underscore Shortcuts and Previous Outputs	115
Suppressing Output	115
Related Magic Commands	116
IPython and Shell Commands	116
Quick Introduction to the Shell	116
Shell Commands in IPython	118

Passing Values To and From the Shell	118
Shell-related Magic Commands	119
Errors and Debugging	120
Controlling Exceptions: <code>%xmode</code>	120
Debugging: When Reading Tracebacks is Not Enough	122
Profiling and Timing Code	125
Timing Code Snippets: <code>%timeit</code> and <code>%time</code>	125
Profiling Full Scripts: <code>%prun</code>	127
Line-by-line Profiling with <code>%lprun</code>	128
Profiling Memory Use: <code>%memit</code> and <code>%mprun</code>	129
More IPython Resources	130
Web Resources	130
Books	131
3. Introduction to NumPy.....	133
Reminder about Built-in Documentation	134
Understanding Data Types in Python	134
A Python Integer is More than just an Integer	135
A Python List is More than just a List	137
Fixed-type arrays in Python	138
Creating Arrays from Python Lists	139
Creating arrays from scratch	139
NumPy Standard Data Types	141
The Basics of NumPy Arrays	142
NumPy Array Attributes	142
Array Indexing: Accessing Single Elements	143
Array Slicing: Accessing Subarrays	144
Reshaping of Arrays	147
Array Concatenation and Splitting	148
Summary	150
Random Number Generation	150
Understanding a Simple “Random” Sequence	151
Built-in tools: Python’s <code>random</code> module	153
Efficient Random Numbers: <code>numpy.random</code>	153
Simultaneously Using Multiple Chains	156
Random Numbers: Further Resources	157
Computation on NumPy Arrays: Universal Functions	157
The Slowness of Loops	157
Introducing UFuncs	158
Exploring NumPy’s UFuncs	159
Advanced Ufunc Features	164
Finding More	165

Aggregations: Min, Max, and Everything In Between	166
Examples of NumPy Aggregates	166
Example: How Tall is the Average US President?	169
Computation on Arrays: Broadcasting	171
Introducing Broadcasting	171
Rules of Broadcasting	173
Broadcasting in Practice	176
Utility Routines for Broadcasting	178
Comparisons, Masks, and Boolean Logic	179
Example: Counting Rainy Days	179
Comparison Operators as ufuncs	181
Working with Boolean Arrays	182
Returning to Seattle's Rain	184
Boolean Arrays as Masks	185
Sidebar: "&" vs. "and"...	186
Fancy Indexing	187
Exploring Fancy Indexing	188
Combined Indexing	189
Generating Indices: <code>np.where</code>	190
Example: Selecting Random Points	191
Modifying values with Fancy Indexing	192
Example: Binning data	194
Numpy Indexing Tricks	196
<code>np.mgrid</code> : Convenient Multi-dimensional Mesh Grids	197
<code>np.ogrid</code> : Convenient Open Grids	200
<code>np.ix_</code> : Open Index Grids	202
<code>np.r_</code> : concatenation along rows	203
<code>np.c_</code> : concatenation along columns	204
Why Index Tricks?	205
Sorting Arrays	205
Sidebar: Big-O Notation	206
Fast Sorts in Python	207
Fast Sorts in NumPy: <code>np.sort</code> and <code>np.argsort</code>	207
Partial Sorts: Partitioning	209
Example: K Nearest Neighbors	209
Searching and Counting Values In Arrays	213
Python Standard Library Tools	213
Searching for Values in NumPy Arrays	214
Counting and Binning	215
Structured Data: NumPy's Structured Arrays	216
Creating Structured Arrays	218
More Advanced Compound Types	219

RecordArrays: Structured Arrays with a Twist	219
On to Pandas	220
4. Introduction to Pandas.....	221
Installing and Using Pandas	222
Reminder about Built-in Documentation	222
Introducing Pandas Objects	222
Pandas Series	223
Pandas DataFrame	226
Pandas Index	230
Looking Forward	231
Data Indexing and Selection	232
Data Selection in Series	232
Data Selection in DataFrame	235
Operations in Pandas	240
Ufuncs: Index Preservation	240
UFuncs: Index Alignment	241
Ufuncs: Operations between DataFrame and Series	244
Summary	245
Handling Missing Data	245
Tradeoffs in Missing Data Conventions	246
Missing Data in Pandas	246
Operating on Null Values	249
Summary	254
Hierarchical Indexing	254
A Multiply-Indexed Series	254
Aside: Panel Data	258
Methods of MultiIndex Creation	258
Indexing and Slicing a MultiIndex	261
Rearranging Multi-Indices	264
Data Aggregations on Multi-Indices	267
Summary	268
Combining Datasets: Concat & Append	269
Recall: Concatenation of NumPy Arrays	270
Simple Concatenation with pd.concat	270
Combining Datasets: Merge and Join	278
Relational Algebra	278
Categories of Joins	278
Specification of the Merge Key	282
Specifying Set Arithmetic for Joins	288
Overlapping Column Names: The suffixes Keyword	290
Example: US States Data	292

Aggregation and Grouping	297
Planets Data	297
Simple Aggregation in Pandas	298
Group By: Split, Apply, Combine	300
Pivot Tables	312
Motivating Pivot Tables	312
Pivot Tables By Hand	312
Pivot Table Syntax	313
Example: Birthrate Data	316
Vectorized String Operations	321
Introducing Pandas String Operations	321
Tables of Pandas String Methods	322
Further Information	327
Example: Recipe Database	327
Working with Time Series	331
Dates and Times in Python	332
Pandas TimeSeries: Indexing by Time	335
Pandas TimeSeries Data Structures	336
Frequencies and Offsets	338
Resampling, Shifting, and Windowing	339
Where to Learn More	345
Example: Visualizing Seattle Bicycle Counts	346
High-Performance Pandas: eval() and query()	353
Motivating query() and eval(): Compound Expressions	353
pandas.eval() for Efficient Operations	354
DataFrame.eval() for Column-wise Operations	356
DataFrame.query() Method	358
Performance: When to Use these functions	359
Learning More	361
Further Resources	361
5. Introduction to Matplotlib.....	363
General matplotlib tips	364
Importing Matplotlib	364
show() or no show()? How to Display your Plots	364
Saving Figures to File	366
Learning More about matplotlib	367
Sidebar: Two Interfaces for the Price of One	368
MatLab-style Interface	368
Simple Line Plots	370
Adjusting the Plot: Line Colors and Styles	374
Adjusting the Plot: Axes limits	377

Labeling Plots	382
Sidebar: Gotchas	384
Simple Scatter Plots	385
Scatter Plots with <code>plt.plot</code>	385
Scatter Plots with <code>plt.scatter</code>	389
A Note on Efficiency	392
Visualizing Errors	393
Basic Errorbars	393
Continuous Errors	395
Density and Contour Plots	397
Visualizing a 3D function	397
Histograms and Binnings	402
Two-dimensional Histograms and Binnings	405
Binned Statistics	408
Customizing Legends	409
Choosing Elements for the Legend	412
Faking the Legend	414
Multiple Legends	416
Customizing Colorbars	417
Customizing Colorbars	418
Example: Hand-written Digits	422
Multiple Subplots	425
<code>plt.axes: subplots by-hand</code>	425
<code>plt.subplot: simple grids of subplots</code>	427
<code>plt.subplots: the whole grid in one go</code>	429
Text and Annotation	433
The Cost of Storage over Time	434
Arrows and Annotation	441
Customizing Ticks	445
Removing Ticks or Labels	447
Reducing or Increasing the Number of Ticks	448
Fancy Tick Formats	450
Summary of Formatters and Locators	453
Customizing Matplotlib: Configurations and Style Sheets	454
Plot Customization By Hand	454
Changing the Defaults: <code>rcParams</code>	456
Stylesheets	459
Three-dimensional Plotting in Matplotlib	463
3D Points and Lines	464
3D Contour Plots	466
Wireframes and Surface Plots	467
Surface Triangulations	470

Geographic Data with Basemap	474
Map Projections	476
Drawing a Map Background	482
Plotting Data On Maps	483
Example: California Cities	484
Example: Surface Temperature Data	486
Visualization With Seaborn	488
Seaborn vs. Matplotlib	489
Exploring Seaborn Plots	491
Example: Exploring New York City Marathon Data	505
6. Machine Learning.....	515
What Is Machine Learning?	516
Machine Learning vs. Statistical Modeling	516
Categories of Machine Learning	517
Qualitative Examples of Machine Learning Applications	517
Summary	527
Figure Code	528
Introducing Scikit-Learn	534
Data Representation in Scikit-Learn	535
Scikit-Learn’s Estimator API	538
Application: Exploring Hand-written Digits	546
Hyperparameters and Model Validation	553
Thinking About Model Validation	554
Selecting the Best Model	558
Learning Curves	565
Validation in Practice: Grid Search	569
Summary	571
Figure Code	571
In Depth: Naive Bayes Classification	575
Bayesian Classification	575
Gaussian Naive Bayes	576
Multinomial Naive Bayes	580
When to Use Naive Bayes	583
Figures	584
In Depth: Linear Regression	585
Simple Linear Regression	585
Basis Function Regression	588
Regularization	592
Example: Predicting Bicycle Traffic	597
Figures	602
In-Depth: Support Vector Machines	602

Motivating Support Vector Machines	603
Support Vector Machines: Maximizing the <i>Margin</i>	605
Example: Face Recognition	614
Support Vector Machine Summary	619
In-Depth: Decision Trees and Random Forests	619
Motivating Random Forests: Decision Trees	620
Ensembles of Estimators: Random Forests	626
Random Forest Regression	628
Example: Random Forest for Classifying Digits	630
Summary of Random Forests	633
Figure Code	634
In Depth: Principal Component Analysis	637
Introducing Principal Component Analysis	638
PCA as Noise Filtering	646
Example: Eigenfaces	648
Principal Component Analysis Summary	651
Figures	652
In-Depth: Manifold Learning	654
Manifold Learning: “HELLO”	654
Multidimensional Scaling (MDS)	656
MDS as Manifold Learning	659
Nonlinear Embeddings: Where MDS Fails	661
Nonlinear Manifolds: Locally Linear Embedding	663
Some Thoughts on Manifold Methods	665
Example: Isomap on Faces	666
Example: Visualizing Structure in Digits	670
Figures	673
In Depth: K-Means Clustering	674
Introducing K-Means	675
K-Means Algorithm: Expectation Maximization	677
In Depth: Gaussian Mixture Models	694
Motivating GMM: Weaknesses of K Means	695
Generalizing E-M: Gaussian Mixture Models	699
GMM as <i>Density Estimation</i>	704
Example: GMM for Generating New Data	709
Figures	713
Covariance Type	713
In-Depth: Kernel Density Estimation	714
Motivating KDE: Histograms	714
Kernel Density Estimation in Practice	719
Example: KDE on a Sphere	721
Example: Not-So-Naive Bayes	725

Feature Engineering: Working with Images	730
HOG features	730
HOG in Action: A Simple Face Detector	731
Caveats and Improvements	737
Learning More	738
Machine Learning in Python	739
General Machine Learning	739

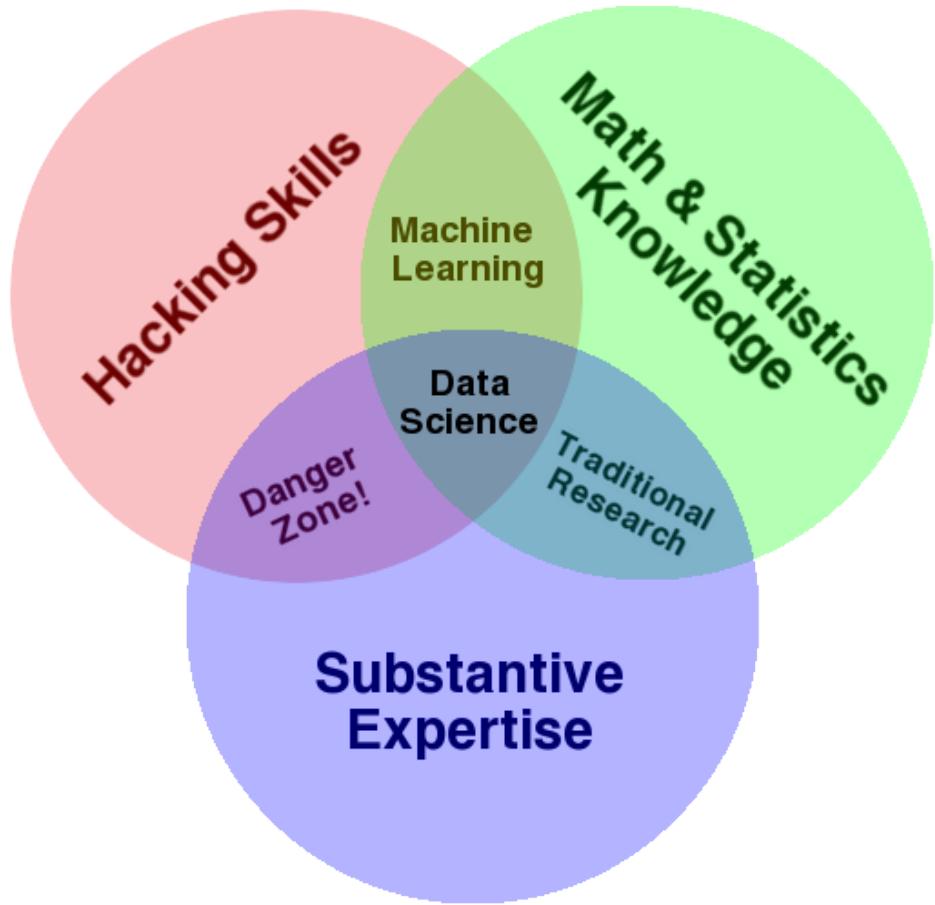
Preface

Jake VanderPlas, Spring 2015

What Is Data Science?

This is a book about doing data science with Python, which immediately begs the question: what is *data science*? It's a surprisingly hard definition to nail down, especially given how ubiquitous the term has become. Vocal critics have variously dismissed the term as a superfluous label – after all, what science doesn't involve data? – or a simple buzzword which only exists to salt resumes and catch the eye of overzealous tech recruiters.

In my mind, these dismissals miss something important. Data science, despite its hype-laden veneer, is perhaps the best label we have for the cross-disciplinary set of skills that are becoming increasingly important in many applications across industry and academia. This cross-disciplinary piece is key: in my mind, the best existing definition of data science is illustrated in the *Data Science Venn Diagram* created by Drew Conway in 2010:



(Source: [Drew Conway](#). Used by permission.)

While some of the intersection labels are a bit tongue-in-cheek, this diagram captures the essence of what I think people mean when they say “Data Science”: it is fundamentally an *interdisciplinary* subject. Data Science comprises three distinct and overlapping sets of skills: 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.

With this in mind, I would encourage you to think of data science not as a new domain of knowledge to learn, but a new set of skills that you can apply within your current area of expertise. Whether you are reporting election results, forecasting

stock returns, optimizing online ad clicks, identifying microorganisms in microscope photos, seeking new classes of astronomical objects, or working with data in any other field, my goal is that the content of this book would give you the ability to ask and answer new questions about your chosen subject area.

Who Is This Book For?

In my teaching both at the University and at various tech-focused conferences and meetups, one of the most common questions I have heard is this: “how should I learn Python?” The people asking are generally technically-minded students, developers, or researchers, often with an already strong background in writing code and using computational and numerical tools. Most of these folks don’t want to learn Python *per se*, but want to learn the language with the aim of using it as a tool for data-intensive and computational science. While a large patchwork of videos, blog posts, and tutorials for this audience is available online, I’ve long been frustrated by the lack of a single good answer to this question; that is what inspired this book.

This book is about how to do data-intensive and computational science and analysis using the open source ecosystem of tools available in Python. It is not meant to be an introduction to programming in general; I assume the reader has familiarity with some coding language, and has some level of experience in defining functions, assigning variables, controlling the flow of a program, and other basic computer science tasks. The first sections, however, do offer content which will be helpful to the reader who has never experienced the joys of programming in Python in particular.

I recognize that many readers will come to this book already experienced on one or another side of the Venn diagram. You as the reader know your own background, and should feel free to focus on the parts of the book that will be most useful to you: for example, a developer new to Python might focus their reading on the first few sections detailing the use of Python tools, while an experienced Python user new to data science might skim these chapters and instead focus on the latter sections which use these Python tools to explore statistical approaches to data. My hope is that as the reader gains expertise in these overlapping areas, the organization of this book will allow him or her to continue using it as a reference, pulling it off the shelf often to learn or recall the basics of different Python data analysis tools and patterns.

What to Expect from This Book

The mental model of data science advocated above consists of the overlap of three broad parts: the **computational expertise**, **statistical expertise**, and **domain expertise** required to tackle modern data-rich analysis. The first four sections of the book focus on the computational component: they are a walk-through of the extensive and mature ecosystem of data-focused tools available in the Python programming lan-

guage. The remaining sections of this book tackle the statistical component: the fundamental concepts of statistics and mathematics useful in analyzing datasets of a variety of size. The goal is that by the end, readers will be poised to use these Python tools process, describe, model, and draw inferences from the various data they encounter.

The final component – the domain expertise – is necessarily left up to the reader. It is up to you to take the fundamental tools and concepts presented here and apply them to data in your own research, work, or field of interest, whether that data consists of server logs, click rates, detector output, economic indicators, election results, traffic volumes, stock prices, social media posts, or anything in between.

Why Python?

This book focuses on recipes and techniques for doing data science using the Python programming language. Python has emerged over the last couple decades as a first-class tool for scientific computing tasks, including the analysis and visualization of large datasets. This may have come as a surprise to early proponents of the Python language: the language itself was not specifically 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: particularly *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 tools that will be mentioned in the following pages.

Python 2 vs Python 3

This book uses the syntax of Python 3, which contains language enhancements which are not compatible with the 2.x series of Python. Though Python 3.0 was first released in 2008, adoption has been relatively slow, particularly in the scientific and web development communities. This is primarily because it took some time for many of the essential packages and toolkits to be made compatible with the new language internals. Since early 2014, however, stable releases of the most important tools in the data science ecosystem have been fully-compatible with both Python 2 and 3, and so this book will use the newer Python 3 syntax. Even though that is the case, the vast majority of code snippets in this book will also work without modification in Python 2: in cases where a Py2-incompatible syntax is used, I will make every effort to note it explicitly.

Other Miscellany

In a few places through this book, shell commands are used. These shell commands are standard ones that you'll see on Linux, Unix, Mac OSX, and other *nix-based systems. Unfortunately, the default Windows shell is different from these; if you're using a Windows system some of these commands may not work. It has been my experience that because of this difference and others, most users of Python for data-intensive computing tend to avoid Windows PCs. For readers using Windows, the vast majority of this book still applicable; just be aware that when the book uses shell commands (marked by "!" in IPython), they may not work as advertised.

Setting Up Your Computer

This section covers how to set up your computer for using Python, including installation of the latest scientific packages.

Installation and packaging of scientific packages has long been a pain-point for users. Fortunately, the last few years has seen the rise of several solutions to the long-standing problems with packaging and distribution of Python's scientific tools.

So why is installation such a challenge? Primarily, this challenge derives from the fact that scientific packages tend not to be written in pure Python: they also call-out to other languages (such as C or Fortran) which have their own system dependencies outside the Python environment itself. Additionally, graphical tools like matplotlib interact with various third-party graphical backends that vary from system to system. For this reason, installing scientific tools is a much more involved process than installing pure Python packages.

Here we'll briefly list the approaches that different users take, along with the advantages and disadvantages, before going into detail on our recommended method. Spoiler: we recommend Anaconda for most users. If you're impatient, feel free to skip the other options and go straight to the instructions at the end of this section.

Installing from Source

Nearly all relevant Python packages have their source code available on Github. Traditionally, package authors more-or-less expected that users would download the source code, possibly adjust the installation settings for their own system, and run the installer.

Advantages:

- This exercise gives the user a greater understanding of the details and dependencies of the software they use.

- Using system-specific settings in a custom install can increase the performance of some packages (e.g. NumPy and SciPy, which use BLAS and LAPACK for linear algebra, can be linked to optimized installations of these libraries on your system).
- You can brag to your friends about being able to install SciPy from source on a mac: a feat accomplished by perhaps only a dozen people in history.
- If desired, you can keep your installations on the bleeding-edge, and update packages from source before new features make it in to a release.

Disadvantages:

- This takes a lot of patience and investment to do successfully.
- You must keep track of complicated dependency chains yourself: this can make your setup very fragile to seemingly unrelated system changes

Using the `pip`: the Python Package Index

The Python Package Index (<http://pypi.python.org>) is the official repository for all Python packages. Built-in to Python 3.4 and above (and also available for earlier Python versions) is the `pip` tool, which aids in downloading and installing released distributions from the PyPI repository. As of 2014, `pip` supports not just Python source code, but also pre-built binaries for various systems. Thus, you can type `pip install scipy` and quickly install the latest SciPy release. There is also a built-in tool for managing *virtual environments*, a way to quickly switch between environments with multiple versions of various packages.

Advantages:

- Quick and easy, and should work similarly across platforms (Mac, Linux, Windows)
- PyPI is the official Python repository, and most new releases of packages will hit here first.
- One-line download and installation of packages without the need for other tools.

Disadvantages:

- pre-built binaries may not be optimized for your particular system.
- does not handle non-Python system dependencies
- `pip` is not a dependency manager: upgrading any single package might break something that depends on it.
- `virtualenv` was originally designed with pure Python code in mind; it sometimes has problems with linking of compiled packages.

Using System Distributions

If you are on Linux, most tools mentioned in this book can be installed with package managers like `apt` and `yum`, which come included with the operating system. Mac users can find similar distribution tools in *Macports* and *Homebrew*, and Windows users might try *Chocolatey* (though I have not used this myself).

Advantages:

- Quick and easy
- Automatic handling of even non-Python dependencies

Disadvantages:

- Platform-dependent: instructions will be different on linux, mac, and windows
- Lack of flexibility: for stability, these are generally a few versions behind for most packages
- Installations may not be optimized for your system
- Generally does not support multiple concurrent installs of different package versions

Third-party Distributions

Responding to the disadvantages of the above options, several companies, organizations, and individuals have gone to the trouble of creating packaged-distributions. While there are many of these in existence, I'll mention two here:

- **Anaconda** (Mac, Linux, Windows) is a distribution created by Continuum Analytics (more on this below).
- **Enthought Canopy** (Mac, Linux, Windows) is a distribution created and maintained by Enthought, Inc.
- **Python(X,Y)** (Windows only) is a Python distribution built and maintained by volunteers.

You can read more about the features of each of these on their project webpages.

Advantages:

- Such distributions are a one-stop shop: they install all the tools you need.
- Canopy and Anaconda in particular are platform-independent.

Disadvantages:

- Installations are from binary, and may not be optimized for your particular machine.

- Depending on the distribution, you may not be able to use the most recent software releases.
- Installations do not affect the system Python installation (though some may actually view this as an advantage).

My Recommendation: Anaconda & conda

Of all the options above, this author has settled on Anaconda as the best option. The killer feature of Anaconda is a piece that hasn't yet been mentioned: the open-source `conda` command-line tool. `conda` in many ways combines the best features of many of the above methods:

- Like platform-dependent package managers, it tracks dependencies in Python and beyond and links to pre-compiled binary distributions of packages.
- Like `pip`, it can download and install the latest version of packages in a simple command
- Like `virtualenv`, it has the notion of multiple environments so that you can cleanly switch between different versions of packages.
- Unlike any of the above tools, it cleanly installs, tracks, and switches between multiple versions of Python and multiple versions of Python packages and all their dependencies.

To get started with `conda`, first download the `miniconda` package and install it (see <http://conda.pydata.org/miniconda.html> – make sure to get a version with Python 3).

Once it is installed, open a terminal and run the command

```
[~]$ conda update conda
```

to make sure `conda`'s list of available packages is updated. Finish by running the command

```
[~]$ conda install numpy pandas scipy matplotlib scikit-learn ipython-notebook
```

to install the most important among the packages used in this book.

That's all it takes to get your system up and ready to follow along with this book! For more information on `conda`, including information about creating and using `conda` environments, refer to the `conda` package documentation linked at the above page.

A Whirlwind Tour of the Python Language

This is a book about doing data science with Python. In order to do that, it helps to first understand a bit about the Python language itself. The intent of this chapter is to get the reader up-to-speed with the features of the Python language that we'll make use of in the remaining chapters. It is aimed at people who are familiar with programming in some language, but may not be familiar with the syntax and semantics of Python in particular.

As such, this chapter in no way aims to be an introduction to programming, or a full introduction to the language; if that is what you are looking for, you might check out one of the recommended references listed in Section X.X. Instead, this chapter will provide a quick overview of some of the essential syntax, built-in data types and structures, function definitions, control flow statements, and other aspects of the Python language that we will find useful further on. Readers who are familiar with the basics of the Python language can safely skip this chapter, and move-on to data science specific material in the later chapters.

Python is Glue

Before starting, we should make one thing clear: Python was not designed as a language for large-scale data analysis. Unlike more specialized computational languages like R, Stata, Matlab, Julia, and others, Python does not have numerical and statistical tools baked-in to the language: those are provided by third-party packages and modules which will be the subject of the rest of this book. For this reason, the relevance of Python syntax to the data science task might not be completely obvious. But it is absolutely relevant, in this sense: **Python syntax is the glue that holds your data science code together**. As many scientists and statisticians have found, Python excels in that role because it is powerful, intuitive, quick to write, fun to use, and above all extremely useful in day-to-day data science tasks.

Because we cannot hope to cover everything about Python in detail, the final section of the chapter lists suggestions and resources to learn more.

The Zen of Python

One of the key aspects of Python is its intuitive simplicity. This is perhaps best captured in Tim Peters' *Zen of Python*, a succinct list which captures the spirit of the Python language. It's so central to the language that it is included with every Python installation. You can find it by closing your eyes, meditating for a few moments, and then importing this:

```
import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

With that, let's start looking at the Python language.

How to Run Python Code

Python is a flexible language, and there are several ways to use it depending on your particular task. One thing that distinguishes Python from other programming languages is that it is *interpreted* rather than *compiled*. This means that it is executed line-by-line, which allows programming to be interactive in a way that is not possible with compiled languages like Fortran, C, or Java. Here we'll describe four primary ways you can run Python code: the **Python Interpreter**, the **IPython Interpreter**, via **Self-contained Scripts**, or in the **IPython notebook**.

The Python Interpreter

The most basic way to execute Python code is line-by-line within the *Python interpreter*. The Python Interpreter can be started by installing the Python language and typing `python` at the command prompt (look for the Terminal on Mac/OSX and Unix/Linux systems, or the Command Prompt application in Windows). Once you do the following:

```
$ python
Python 3.3.5 |Anaconda 1.6.1 (x86_64)| (default, Jan 2 2015, 13:57:31)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

With the interpreter running, you can begin to type and execute code snippets. Here we'll use the interpreter as a simple calculator, performing calculations and assigning values to variables:

```
>>> 1 + 1
2
>>> x = 5
>>> x * 3
15
```

The interpreter makes it very convenient to try-out small snippets of Python code and to experiment with short sequences of operations.

The IPython Interpreter

If you spend much time with the basic Python interpreter, you'll find that it lacks many of the features of a full-fledged interactive development environment. An alternative interpreter called *IPython* (for Interactive Python) is bundled with the Anaconda distribution, and includes a host of convenient enhancements to the basic Python interpreter. It can be started by typing `ipython` at the command prompt:

```
$ ipython
Python 3.3.5 |Anaconda 1.6.1 (x86_64)| (default, Jan 2 2015, 13:57:31)
Type "copyright", "credits" or "license" for more information.

IPython 2.2.0 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
In [1]:
```

The main aesthetic difference between the Python interpreter and the enhanced IPython interpreter lies in the command prompt: Python uses `>>>` by default, while IPython uses numbered commands (e.g. `In [1]:`). Regardless, we can execute code line-by-line just as we did above:

```
In [1]: 1 + 1  
Out[1]: 2
```

```
In [2]: x = 5
```

```
In [3]: x * 3  
Out[3]: 15
```

Note that just as the input is numbered, the output of each command is numbered as well. IPython makes available a wide array of useful features, which we will cover more fully in Chapter 2.

Self-contained Python Scripts

Running Python snippets line-by-line is useful in some cases, but for more complicated programs it is more convenient to save code to file, and execute it all at once. By convention, Python scripts are saved in files with a .py extension. For example, let's create a script called `test.py` which contains the following:

```
# file: test.py  
print("Running test.py")  
x = 5  
print("Result is", 3 * x)
```

To run this file, we make sure it is in the current directory and type `python filename` at the command prompt:

```
$ python test.py  
Running test.py  
Result is 15
```

For more complicated programs, creating self-contained scripts like this one is a must.

The IPython Notebook

A nice meeting of the interactive terminal and the self-contained script is the *IPython notebook*, a document format which allows executable code, formatted text, graphics, and even interactive features to be combined into a single document. Though the notebook began as a Python-only format, it has since been made compatible with a large number of programming languages, and is now part of the rapidly evolving *Jupyter Project*. The notebook is useful both as a development environment, and as a means of sharing work via rich computational and data-driven narratives that mix together code, figures, data, and text. As an example of what can be accomplished in the IPython notebook, I will refer you to the book you are reading: this entire text was composed in a series of IPython notebooks, some of which can be viewed, executed, and downloaded online at the O'Reilly website [TODO: add link to book website].

A Quick Tour of Python Language Syntax

Python was originally developed as a teaching language, but its ease-of-use and clean syntax have led it to be embraced by beginners and experts alike. The cleanliness of Python's syntax has led some to call it “executable pseudocode”, and indeed my own experience has been that it is much easier to read a Python script than to read a similar script written in C. Here we'll begin to discuss the main features of Python's syntax.

Syntax refers to the structure of the language: i.e. what constitutes a correctly-formed program. For the time being we'll not focus on the semantics – the meaning of the words and symbols within the syntax – but will return to this at a later point.

Consider the following code example:

```
# set the midpoint
midpoint = 5

# make two empty lists
lower = []; upper = []

# split the numbers into lower and upper
for i in range(10):
    if (i < midpoint):
        lower.append(i)
    else:
        upper.append(i)

print("lower:", lower)
print("upper:", upper)

lower: [0, 1, 2, 3, 4]
upper: [5, 6, 7, 8, 9]
```

This script is a bit silly, but it compactly illustrates several of the important aspects of Python syntax. Let's walk through it and discuss some of the syntactical features of Python

Comments are marked by

The script starts with a comment:

```
# set the midpoint
```

Comments in Python are indicated by a pound sign (#), and anything on the line following the pound sign is ignored by the interpreter. This means, for example, that you can have stand-alone comments like the one above, as well as inline comments that follow a statement. For example:

```
x += 2 # shorthand for x = x + 2
```

Python does not have any syntax for multi-line comments, like the `/* ... */` syntax used in C and C++, though multi-line strings are often used as a replacement for multi-line comments (more on this in section X.X).

End-of-line Terminates a Statement

The next line in the script is

```
midpoint = 5
```

This is an assignment operation, where we've created a variable named `midpoint` and assigned it the value 5. Notice that the end of this statement is simply marked by the end of the line. This is in contrast to languages like C and C++, where every statement must end with a semicolon (`;`).

In Python, if you'd like a statement to continue to the next line, it is possible to use the `"\"` marker to indicate this:

```
x = 1 + 2 + 3 + 4 +\
    5 + 6 + 7 + 8
```

It is also possible to continue expressions on the next line within parentheses, without using the `"\"` marker:

```
x = (1 + 2 + 3 + 4 +
      5 + 6 + 7 + 8)
```

Most Python style-guides recommend the second version of line continuation (within parentheses) to the first (use of the `"\"` marker).

Semicolon can Optionally Terminate a Statement

Sometimes it can be useful to put multiple statements on a single line. The next portion of the script is

```
lower = []; upper = []
```

This shows the example of how the semicolon (`;`) familiar in C can be used optionally in Python to put two statements on a single line. Functionally, this is entirely equivalent to writing

```
lower = []
upper = []
```

Using a semicolon to put multiple statements on a single line is generally discouraged by most Python style guides, though occasionally it proves convenient.

Indentation: Whitespace Matters!

Next we get to the main block of code:

```
for i in range(10):
    if i < midpoint:
        lower.append(i)
    else:
        upper.append(i)
```

This is a compound control-flow statement including a loop and a conditional – we'll look at these types of statements in a moment. For now, consider that this demonstrates what is perhaps the most controversial feature of Python's syntax: whitespace is meaningful!

In programming languages, a *block* of code is a set of statements which should be treated as a unit. In C, for example, code blocks are denoted by curly braces:

```
// C code
for(int i=0; i<100; i++)
{
    // curly braces indicate code block
    total += i;
}
```

In Python, code blocks are denoted by *indentation*:

```
for i in range(100):
    # indentation indicates code block
    total += i
```

in Python, indented code blocks are always preceded by a colon (:) on the previous line.

The use of indentation helps to enforce the uniform, readable style that many find appealing in Python code. But it might be confusing to the uninitiated; for example, the following two snippets will produce different results:

```
>>> if x < 4:
...     y = x * 2
...     print(x)      ... print(x)
```

In the snippet on the left, `print(x)` is in the indented block, and will be executed only if `x` is less than 4. In the snippet on the right `print(x)` is outside the block, and will be executed regardless of the value of `x`!

Python's use of meaningful whitespace often is surprising to programmers who are used to other languages, but in practice it can lead to much more consistent and readable code than languages which do not enforce indentation of code blocks. If you find Python's use of whitespace disagreeable, I'd encourage you to give it a try: as I did, you may find that you come to appreciate it.

Finally, you should be aware that the *amount* of whitespace used for indenting code blocks is up to the user, as long as it is consistent throughout the script. By convention, most style guides recommend to **indent code blocks by four spaces**, and that is

the convention we will follow in this book. Note that many text editors like emacs and vim contain Python modes which do four-space indentation automatically.

Whitespace *Within* Lines Does Not Matter

While the mantra of *meaningful whitespace* holds true for white space *before* lines (which indicate a code block), white space *within* lines of Python code does not matter. For example, all three of these expressions are equivalent:

```
x=1+2  
x = 1 + 2  
x           =     1     +          2
```

Abusing this flexibility can lead to issues with code readability – in fact, abusing white space is often one of the primary means of intentionally obfuscating code (which some people do for sport). Using whitespace effectively can lead to much more readable code, especially in cases where operators follow each other: compare the following two expressions for exponentiating by a negative number:

```
x=10**-2
```

to

```
x = 10 ** -2
```

I find the second version with spaces much more easily readable at a single glance. Most Python style guides recommend using a single space around binary operators, and no space around unary operators. We'll discuss Python's operators further in the following chapter.

Parentheses are for Grouping or Calling

In the above code snippet, we see two uses of parentheses. First, they can be used in the typical way to group statements or mathematical operations:

```
2 * (3 + 4)
```

```
14
```

They can also be used to indicate that a *function* is being called. In the above snippet, the `print()` function is used to display the contents of a variable (see the sidebar). The function call is indicated by a pair of open and close parentheses, with the *arguments* to the function contained within:

```
print('first value:', 1)  
print('second value:', 2)  
  
first value: 1  
second value: 2
```

Some functions can be called with no arguments at all, in which case the open and close parentheses still must be used to indicate a function evaluation. An example of this is the `sort` method of lists:

```
L = [4,2,3,1]
L.sort()
print(L)

[1, 2, 3, 4]
```

The "`()`" after `sort` indicates that the function should be executed, and is required even if no arguments are necessary.

Finishing Up and Learning More

This has been a very brief exploration of the essential features of Python syntax; its purpose is to give you a good frame of reference for when you're reading the code in later sections. Several times we've mentioned Python "style guides", which can help teams to write similar-style code. The most widely-used style guide in Python is known as PEP8, and can be found at <https://www.python.org/dev/peps/pep-0008/>. As you begin to write more Python code, it would be useful to read through this! The style suggestions contain the wisdom of many Python gurus, and most suggestions go beyond simple pedantry: they are experience-based recommendations for avoiding mistakes and bugs in your code.

Sidebar: Note on the `print()` Function

Above we used the example of the `print()` function. The `print()` function is one piece that has changed between Python 2.x and Python 3.x. In Python 2, `print` behaved as a statement: that is, you could write

```
# Python 2 only!
>> print "first value:", 1
first value: 1
```

For various reasons, the language maintainers decided that in Python 3 `print()` should become a function, so we now write

```
# Python 3 only!
>>> print("first value:", 1)
first value: 1
```

This is one of the many backward-incompatible constructs between Python 2 and 3. As of the writing of this book, it is common to find examples written in both versions of Python, and the presence of the `print` statement rather than the `print()` function is often one of the first signs that you're looking at Python 2 code.

Basic Python Semantics: Variables and Objects

This section will begin to cover the basic semantics of the Python language. As opposed to the *syntax* covered in the previous section, the *semantics* of a language involve the meaning of the statements. As with the syntax section, here we'll preview a few of the essential semantic constructions in Python to give you a better frame of reference for understanding the code in the following sections.

This section will cover the semantics of *Variables* and *Objects*, which are the main ways you store, reference, and operate on data within a Python script.

Python Variables are Pointers

Assigning variables in Python is as easy as putting a variable name to the left of the equal (=) sign:

```
# assign 4 to the variable x
x = 4
```

This may seem straightforward, but if you have the wrong mental model of what this operation does, the way Python works may seem confusing. We'll briefly dig into that here.

In many programming languages, variables are best thought of as containers or buckets into which you put data. So in C, for example, when you write

```
// C code
int x = 4;
```

you are essentially defining a *memory bucket* named x, and putting the value 4 into it. In Python, by contrast, variables are best thought of not as containers but as pointers. So in Python, when you write

```
x = 4
```

you are essentially defining a *pointer* named x which points to some other bucket containing the value 4. Note one consequence of this: because Python variables just point to various objects, there is no need to "declare" the variable, or even require the variable to always point to information of the same type! This is the sense in which people say Python is *dynamically-typed*: variable names can point to objects of any type. So in Python, you can do things like this:

```
x = 1      # x is an integer
x = 'hello' # now x is a string
x = [1, 2, 3] # now x is a list
```

While users of statically-typed languages might miss the type-safety that comes with declarations like C's declarations,

```
int x = 4;
```

this dynamic typing is one of the pieces that makes Python so quick to write and easy to read.

There is a consequence of this “variable as pointer” approach that users need to be aware of. If we have two variable names pointing to the same *mutable* object, then changing one will change the other as well! For example, let’s create and modify a list:

```
x = [1, 2, 3]
y = x
```

We’ve created two variables `x` and `y` which both point to the same bucket (or more precisely, the same *object*). Because of this, if we modify the list via one of the pointers, we’ll see that the “other” list will be modified as well:

```
print(y)
x.append(4) # add 4 to the end of x
print(y) # y is modified as well!

[1, 2, 3]
[1, 2, 3, 4]
```

This behavior might seem confusing if you’re wrongly thinking of variables as bins-which-contain-data. But if you’re correctly thinking of variables as pointers-to-objects, then this behavior makes sense.

Note also that if we use `=` to assign another value to `x`, this will not affect the value of `y`: assignment is simply a change of what object the variable points to:

```
x = 'something else'
print(y) # y is unchanged

[1, 2, 3, 4]
```

Again, this makes perfect sense if you think of `x` and `y` as pointers, and the `=` operator as an operation which changes what the name points to.

You might wonder if all of this pointer stuff makes arithmetic operations in Python difficult to track, but Python is set up so that this is not an issue. Numbers, strings, and other *simple types* are immutable: you can’t change their value – you can only change what values the variables point to. So, for example, it’s perfectly safe to do operations like the following:

```
x = 10
y = 10
x += 5 # add 5 to x
print("x =", x)
print("y =", y)

x = 15
y = 10
```

When we call `x += 5`, we are not modifying the value of the `5` object pointed to by `x`, but rather we are changing the object to which `x` points. For this reason, the value of `y` is not affected by the operation.

Everything is an Object

Python is an object-oriented programming language, and in Python everything is an object.

Let's flesh-out what this means. Above we saw that variables are simply pointers, and the variable names themselves have no attached type information. This leads some to claim erroneously that Python is a type-free language. But this is not the case! Consider the following:

```
x = 4
type(x)

int

x = 'hello'
type(x)

str

x = 3.14159
type(x)

float
```

Python has types; however, the types are linked not to the variable names but **to the objects themselves**.

In object-oriented programming languages like Python, an *object* is an entity which contains data along with associated metadata and/or functionality. In Python everything is an object, which means every entity has this type of metadata (called *attributes*) and associated functionality (called *methods*). These attributes and methods are accessed via the `". "` syntax.

For example, above we saw that lists have an `append` method, which adds an item to the list, and is accessed via the `". "` syntax:

```
L = [1, 2, 3]
L.append(100)
print(L)

[1, 2, 3, 100]
```

While it might be expected for compound objects like lists to have attributes and methods, what is sometimes unexpected is that in Python even simple types have attached attributes and methods. For example, numerical types have a `real` and `imag` attribute which returns the real and imaginary part of the value, if viewed as a complex number:

```
x = 4.5
print(x.real, "+", x.imag, 'i')
4.5 + 0.0 i
```

Methods are like attributes, except they are functions which you can call using open and closed parentheses. For example, floating point numbers have a method called `is_integer` that checks whether the value is an integer:

```
x = 4.5
x.is_integer()
False

x = 4.0
x.is_integer()

True
```

When we say that everything in Python is an object, we really mean that *everything* is an object: even the attributes and methods of objects are themselves objects with their own `type` information:

```
type(x.is_integer)
builtin_function_or_method
```

We'll find that the everything-is-object design choice of Python allows for some very convenient language constructs.

Basic Python Semantics: Operators

In the previous section we began to look at the semantics of Python variables and objects; here we'll dig into the semantics the various *operators* included in the language. By the end of this section, you'll have the basic tools to begin comparing and operating on data in Python.

Arithmetic Operations

Python implements seven basic binary arithmetic operators, two of which can double as unary operators. They are summarized in the following table:

Operator	Name	Description
a + b	Addition	sum of a and b
a - b	Subtraction	difference of a and b
a * b	Multiplication	product of a and b
a / b	True Division	quotient of a and b
a // b	Floor Division	quotient of a and b, removing fractional parts
a % b	Modulus	integer remainder after division of a by b

Operator	Name	Description
a ** b	Exponentiation	a raised to the power of b
-a	Negation	the negative of a
+a	Unary Plus	a unchanged (rarely used)

These operators can be used and combined in intuitive ways, using standard parentheses to group operations. For example:

```
# addition, subtraction, multiplication
(4 + 8) * (6.5 - 3)
42.0
```

Floor division is true division with fractional parts truncated:

```
# True Division and Floor Division
print(11 / 2)
print(11 // 2)

5.5
5
```

The floor division operator was added in Python 3; you should be aware if working in Python 2 that the standard division operator (/) acts like floor division for integers and like true division for floating point numbers.

Finally, I'll mention that there is an eighth arithmetic operator which will be added in Python 3.5, set for final release in late 2015: this is the a @ b operator which is meant to indicate the *matrix product* of a and b, for use in various linear algebra packages.

Bitwise Operations

In addition to the standard numerical operations, Python includes operators to perform bitwise logical operations on integers. These are much less commonly used than the standard arithmetic operations, but it's useful to know that they exist. The six bitwise operators are summarized in the following table:

Operator	Name	Description
a & b	Bitwise And	bits defined in both a and b
a b	Bitwise Or	bits defined in a or b or both
a ^ b	Bitwise XOr	bits defined in a or b but not both
a << b	Bit Shift Left	shift bits of a left by b units
a >> b	Bit Shift Right	shift bits of a right by b units
~a	Bitwise Not	Bitwise Negation of a

These bitwise operators only make sense in terms of the binary representation of numbers, which you can see using the built-in `bin` function:

```
bin(10)  
'0b1010'
```

The result is prefixed with '`0b`', which indicates a binary representation. The rest of the digits indicate that the number 10 is expressed as the sum $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$. Similarly, we can write:

```
bin(4)  
'0b100'
```

Now, using bitwise or, we can find the number which combines the bits of 4 and 10:

```
4 | 10  
14  
bin(4 | 10)  
'0b1110'
```

Bitwise operators will prove extremely useful later, when we take a look at the uses of masking operations on arrays of data (see Section X.X).

Assignment Operations

We've seen that variables can be assigned with the "`=`" operator, and the values stored for later use. For example:

```
a = 24  
print(a)  
24
```

We can use these variables in expressions with any of the operators mentioned above. For example, to add 2 to `a` we write:

```
a + 2  
26
```

We might want to update the variable `a` with this new value; in this case we could combine the addition and the assignment and write `a = a + 2`. Because this type of combined operation and assignment is so common, Python includes built-in update operators for all of the arithmetic operations:

```
a += 2 # equivalent to a = a + 2  
print(a)  
26
```

There is an augmented assignment operator corresponding to each of the binary operators listed above; in brief they are:

```
a += b  a -= b  a *= b  a /= b  
a //= b  a %= b  a **= b  a &= b  
a |= b  a ^= b  a <<= b  a >>= b
```

Each one is equivalent to the corresponding operation followed by assignment: that is, for any operator " ", the expression `a = b` is equivalent to `a = a b`, with a slight catch. We'll see that for more complicated objects like the arrays and dataframes discussed in later chapters, these augmented assignment operations are actually subtly different than their more verbose counterparts: they modify the contents of the original object rather than creating a new object to store the result.

Comparison Operations

Another type of operation which can be very useful is comparison of different values. For this, Python implements standard comparison operators, which return Boolean values `True` and `False`. The comparison operations are listed in the following table:

Operation	Description	Operation	Description
<code>a == b</code>	<code>a</code> equal to <code>b</code>	<code>a != b</code>	<code>a</code> not equal to <code>b</code>
<code>a < b</code>	<code>a</code> less than <code>b</code>	<code>a > b</code>	<code>a</code> greater than <code>b</code>
<code>a <= b</code>	<code>a</code> less than or equal to <code>b</code>	<code>a >= b</code>	<code>a</code> greater than or equal to <code>b</code>

These comparison operators can be combined with the above arithmetic and bitwise operators to express a virtually limitless range of tests for the numbers. For example, we can check if a number is odd by checking that the modulus with 2 returns 1:

```
# 25 is odd  
25 % 2 == 1  
  
True  
  
# 66 is odd  
66 % 2 == 1  
  
False
```

We can string-together multiple comparisons to check more complicated relationships:

```
# check if a is between 15 and 30  
a = 25  
15 < a < 30  
  
True
```

And, just to make your head hurt a bit, take a look at this comparison:

```
-1 == ~0
```

```
True
```

If you're curious as to why this is, look up the *Two's Complement* integer encoding scheme, which is what Python uses to encode signed integers, and think about happens when you start flipping all the bits of this encoding.

Boolean Operations

Finally, Python contains operators to combine boolean values using the standard concepts of “and”, “or”, and “not”. Predictably, these operators are expressed using the words `and`, `or`, and `not`:

```
x = 4
(x < 6) and (x > 2)
```

```
True
```

```
(x > 10) or (x % 2 == 0)
```

```
True
```

```
not (x < 6)
```

```
False
```

Boolean algebra experts will notice that the “`xor`” operator is not included; this can of course be constructed in several ways from a compound statement of the other operators. Otherwise, a clever trick you can use for XOR of boolean values is the following:

```
# (x < 6) xor (x > 2)
(x > 1) != (x < 10)
```

```
False
```

These sorts of boolean operations will become extremely useful when we begin discussing *control flow statements* such as conditionals and loops.

One sometimes confusing thing about the language is when to use boolean operators (`and`, `or`, `not`), and when to use bitwise operations (`&`, `|`, `~`). The answer lies in their names: boolean operators should be used when you want to compute *boolean values* (*i.e. truth or falsehood*) of *entire statements*. Bitwise operations should be used when you want to *operate on individual bits or components of the objects in question*.

Identity and Membership Operators

For completeness here, we'll mention that Python also includes operators to check for identity and membership. They are the following:

Operator	Description
a is b	True if a and b are identical objects
a is not b	True if a and b are not identical objects
a in b	True if a is a member of b
a not in b	True if a is not a member of b

We'll discuss each of these here:

Identity Operators

The identity operators check for *object identity*. Object identity is different than equality, as we can see here:

```
a = [1, 2, 3]
b = [1, 2, 3]

a == b
True

a is b
False

a is not b
True
```

What do identical objects look like? Here is an example:

```
a = [1, 2, 3]
b = a
a is b
True
```

The difference between the two cases here is that in the first, a and b point to *different objects*, while in the second they point to the *same object*. As we saw in the previous section, Python variables are pointers. The **is** operator checks whether the two variables are pointing to the same container (object), rather than referring to what the container contains. In general, unless you know what you're doing and you really have reason to use **is**, you'll usually want to use **==** instead.

Membership Operators

Membership operators check for membership within compound objects. So, for example, we can write:

```
1 in [1, 2, 3]
True

2 not in [1, 2, 3]
```

```
False
```

These membership operations are an example of what makes Python so easy to use compared to, say, C. In C, you would have to construct a loop over your list and manually check for equality in each value. In Python, you just type what you want to know, in a manner that looks almost like English.

Summary

Here we've discussed the semantics of *operators* in Python; this concludes our brief tour of Python's syntax and semantics. From this foundation of knowledge about the language, we'll start building up some of the constructs we'll need to do data science in practice.

Built-in Types: Simple Values

When discussing Python variables and objects (section X.X), we mentioned the fact that all Python objects have type information attached. Here we'll briefly walk through the built-in simple types offered by Python. We say "simple types" because there are also several compound types, which will be discussed in Section X.X.

Python's simple types are summarized in the following table:

Table 1-1. Python Scalar Types

Type	Example	Description
int	x = 1	integers: i.e. whole numbers
float	x = 1.0	floating-point: i.e. real numbers
complex	x = 1 + 2j	complex: i.e. numbers with real and imaginary part
bool	x = True	boolean: True/False values
str	x = 'abc'	string: characters or text
NoneType	x = None	special object indicating nulls

We'll take a quick look at each of these in turn.

Numeric Types

Integers

The most basic numerical type is the integer. Any number without a decimal point is an integer:

```
x = 1
type(x)

int
```

Python integers are actually quite a bit more sophisticated than integers in languages like C. These are fixed-precision, and usually overflow at some value (often near 2^{31} or 2^{63} , depending on your system). Python integers are variable-precision, so you can do computations that would overflow in other languages:

```
2 ** 200  
1606938044258990275541962092341162602522202993782792835301376
```

Another convenient feature of Python integers is that by default, division up-casts to floating-point type:

```
5 / 2  
2.5
```

Note that this upcasting is a feature of Python 3; in Python 2, like in statically-typed languages such as C, integer division truncates any decimal and always returns an integer:

```
# Python 2 behavior  
>>> 5 / 2  
2
```

To recover this behavior in Python 3, you can use the floor-division operator:

```
5 // 2  
2
```

Finally, note that although Python 2.x had both an `int` and `long` type, Python 3 combines the behavior of these two into a single `int` type.

Floating Point Numbers

The floating-point type can store fractional numbers. They can be defined either in standard decimal notation, or in exponential notation:

```
x = 0.000005  
y = 5e-6  
print(x == y)  
  
True  
  
x = 1400000.00  
y = 1.4e6  
print(x == y)  
  
True
```

In the exponential notation, the `e` or `E` can be read “...times ten to the...”, so that `1.4e6` is interpreted as 1.4×10^6 .

An integer can be explicitly converted to a float with the `float` constructor:

```
float(1)
```

```
1.0
```

One convenient method of the `float` type is the `is_integer()` method, which (predictably) tells you whether the value is an integer:

```
x = 4.0
x.is_integer()
```

```
True
```

```
y = 3.14
y.is_integer()
```

```
False
```

Sidebar: Floating Point Precision. One thing to be aware of with floating point arithmetic is that its precision is limited, which can cause equality tests to be unstable. For example:

```
0.1 + 0.2 == 0.3
```

```
False
```

Why is this the case? It turns out that it is not a behavior unique to Python, but is due to the fixed-precision format of the binary floating-point storage used by most, if not all, scientific computing platforms. All programming languages using floating-point numbers store them in a fixed number of bits, and this leads some numbers to be represented only approximately. We can see this by printing the three values to high precision:

```
print("0.1 = {:.17f}".format(0.1))
print("0.2 = {:.17f}".format(0.2))
print("0.3 = {:.17f}".format(0.3))

0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.3 = 0.29999999999999999
```

We're used to thinking of numbers in decimal (base-10) notation, so that each fraction must be expressed as a sum of powers of 10:

$$1/8 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

In the familiar base-10 representation, we represent this in the familiar decimal expression: 0.125.

Computers store values in binary notation, so that each number is expressed as a sum of powers of 2:

$$1/8 = 0 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

In a base-2 representation, we can write this 0.001_2 , where the subscript 2 indicates binary notation. The value $0.125 = 0.001_2$ happens to be one number which both binary and decimal notation can represent in a finite number of digits.

In the familiar base-10 representation of numbers, you are probably familiar with numbers which can't be expressed in a finite number of digits. For example, dividing 1 by 3 gives, in standard decimal notation:

$$1/3 = 0.33333333\cdots$$

The 3s go on forever: that is, to truly represent this quotient, the number of required digits is infinite!

Similarly, there are numbers for which binary representations require an infinite number of digits. For example:

$$1/10 = 0.00011001100110011\cdots_2$$

Just as decimal notation requires an infinite number of digits to perfectly represent $1/3$, binary notation requires an infinite number of digits to represent $1/10$. Python internally truncates these representations at 52 bits beyond the first nonzero bit on most systems.

This rounding error for floating point values is a necessary evil of working with floating point numbers. The best way to deal with it is to always keep in mind that floating point arithmetic to be approximate, and **never** use exact equality tests with floating point values.

Complex Numbers

Complex numbers are numbers with real and imaginary (floating point) parts. We've seen integers and real numbers above; we can use these to construct a complex number:

```
complex(1, 2)  
(1+2j)
```

Alternatively, we can use the "j" suffix in expressions to indicate the imaginary part:

```
1 + 2j  
(1+2j)
```

Complex numbers have a variety of interesting attributes and methods, which we'll briefly demonstrate here:

```
c = 3 + 4j  
c.real # real part  
3.0  
c.imag # imaginary part
```

```
4.0
c.conjugate() # complex conjugate
(3-4j)
abs(c) # magnitude, i.e. sqrt(c.real ** 2 + c.imag ** 2)
5.0
```

String Type

Strings in Python are created with single or double quotes:

```
message = "what do you like?"
response = 'spam'
```

Python has many extremely useful string functions and methods; here are a few of them:

```
# length of string
len(response)

4

# Make upper-case. See also str.lower()
response.upper()

'SPAM'

# Capitalize. See also str.title()
message.capitalize()

'What do you like?'

# concatenation with +
message + response

'what do you like?spam'

# multiplication is multiple concatenation
5 * response

'spamspamspamspamspam'

# Access individual characters (zero-based indexing)
message[0]

'w'
```

For more discussion of indexing in Python, see the next section.

These examples only scrape the surface of the string methods and functions available; because manipulating strings is such an important topic when working with data, we cover them more completely in Section X.X.

None Type

Python includes a special type, the `NoneType` which has only a single possible value: `None`.

```
type(None)
```

```
NoneType
```

You'll see `None` used in many places, but perhaps most commonly it is used as the default return value of a function. For example, the `print()` function in Python 3 does not return anything, but we can still catch its value:

```
ret = print('abc')
```

```
abc
```

```
print(ret)
```

```
None
```

Likewise, any function in Python with no return value is, in reality, returning `None`.

Boolean Type

The boolean type is a simple type with two possible values: `True` and `False`, and is returned by comparison operators discussed previously:

```
result = (4 < 5)
```

```
result
```

```
True
```

```
type(result)
```

```
bool
```

Keep in mind that the boolean values are case-sensitive: unlike some other languages, `True` and `False` must be capitalized!

```
print(True, False)
```

```
True False
```

Booleans can also be constructed using the `bool()` object constructor: values of any other type can be converted to boolean via predictable rules. For example, any numeric type is `False` if equal to zero, and `True` otherwise:

```
bool(2014)
```

```
True
```

```
bool(0)
```

```
False
```

```
bool(3.1415)
```

```
True
```

The boolean conversion of `None` is always False:

```
bool(None)  
False
```

For strings, `bool(s)` is False for empty strings and True otherwise:

```
bool("")  
False  
bool("abc")  
True
```

For sequences, which we'll see in the next section, the boolean representation is False for empty sequences and True for any other sequences

```
bool([1, 2, 3])  
True  
bool([])  
False
```

Built-in Data Structures

The previous section discussed Python's simple types: `int`, `float`, `complex`, `bool`, `str`, etc. Python also has several built-in compound types, which act as containers for other types. These compound types are:

Type Name	Example	Description
list	[1, 2, 3]	ordered collection
tuple	(1, 2, 3)	immutable ordered collection
dict	{'a':1, 'b':2, 'c':3}	unordered key,value mapping
set	{1, 2, 3}	unordered collection

As you can see, round, square, and curly brackets have distinct meanings when it comes to the type of collection produced. We'll take a quick tour of these data structures here.

Lists

Lists are the basic *ordered* and *mutable* data collection type in Python. They can be defined with comma-separated values between square brackets; for example, here is a list of the first several prime numbers

```
L = [2, 3, 5, 7]
```

Lists have a number of useful properties and methods available to them. Here we'll take a quick look at some of the more common and useful ones:

```
# Length of a list
len(L)
4

# Append a value to the end
L.append(11)
print(L)

[2, 3, 5, 7, 11]

# Addition concatenates lists
L + [13, 17, 19]

[2, 3, 5, 7, 11, 13, 17, 19]

# sort() method sorts in-place
L = [2, 5, 1, 6, 3, 4]
L.sort()
L

[1, 2, 3, 4, 5, 6]
```

There are many more built-in list methods than we have space to cover here. Fortunately, they are well-covered in Python's online documentation.

While we've been demonstrating lists containing values of a single type, one of the powerful features of Python's compound objects is that they can contain objects of *any* type, or even a mix of types. For example:

```
L = [1, 'two', 3.14, [0, 3, 5]]
```

This flexibility is a consequence of Python's dynamic type system. Creating such a mixed list in a statically-typed language like C can be much more of a headache! We see that lists can even contain other lists as elements. Such type flexibility is an essential piece of what makes Python code relatively quick and easy to write.

So far we've been considering manipulations of lists as a whole; another essential piece is the accessing of individual elements. This is done in Python via *indexing* and *slicing*, which we'll explore next.

List Indexing and Slicing

Python provides access to elements in compound types through *indexing* for single elements, and *slicing* for multiple elements. As we'll see, both are indicated by a square-bracket syntax. Suppose we return to our list of the first several primes:

```
L = [2, 3, 5, 7, 11]
```

Python uses *zero-based* indexing, so we can access the first and second element in using the following syntax:

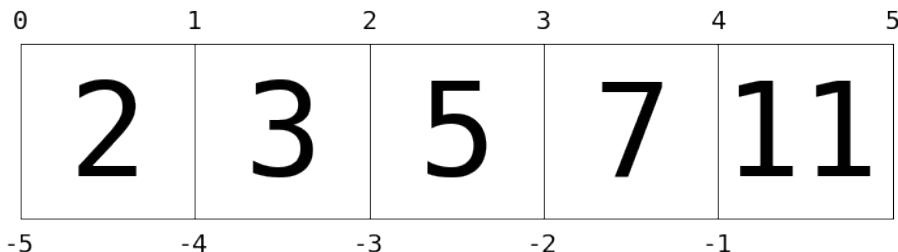
```
L[0]  
2  
L[1]  
3
```

Elements at the end of the list can be accessed with negative numbers, starting from -1:

```
L[-1]  
11  
L[-2]  
7
```

You can visualize this indexing scheme this way:

```
# TODO: how to hide this code?  
L = [2, 3, 5, 7, 11]  
  
%matplotlib inline  
import matplotlib.pyplot as plt  
  
fig = plt.figure(figsize=(10, 4))  
ax = fig.add_axes([0, 0, 1, 1], xticks=[], yticks=[], frameon=False,  
                  aspect='equal')  
  
for i in range(5):  
    ax.add_patch(plt.Rectangle([i - 0.5, -0.5], 1, 1, fc='none', ec='black'))  
    ax.text(i, -0.05, L[i], size=100,  
            ha='center', va='center', family='monospace')  
  
for i in range(6):  
    ax.text(i - 0.5, 0.55, str(i), size=20,  
            ha='center', va='bottom', family='monospace')  
  
for i in range(5):  
    ax.text(i - 0.5, -0.58, str(-5 + i), size=20,  
            ha='center', va='top', family='monospace')  
  
ax.axis([-0.7, 4.7, -0.7, 0.7]);
```



Here values in the list are represented by large numbers in the squares; list indices are represented by small numbers above and below. In this case `L[2]` returns 5, because that is the next value at index 2.

Where *indexing* is a means of fetching a single value from the list, *slicing* is a means of accessing multiple values in sub-lists. It uses a colon to indicate the start point (inclusive) and end point (non-inclusive) of the sub-array. For example, to get the first three elements of the list, we can write

```
L[0:3]
```

```
[2, 3, 5]
```

Notice where 0 and 3 lie in the above diagram, and how the slice takes just the values between the indices. If we leave out the first index, 0 is assumed, so we can equivalently write

```
L[:3]
```

```
[2, 3, 5]
```

Similarly, if we leave out the last index, it defaults to the length of the list. Thus the last three elements can be accessed as follows

```
L[-3:]
```

```
[5, 7, 11]
```

Finally, it is possible to specify a third integer which represents the step size; for example, to select every second element of the list, we can write:

```
L[::-2] # equivalent to L[0:len(L):2]
```

```
[2, 5, 11]
```

A particularly useful version of this is to specify a negative step, which will reverse the array:

```
L[::-1]
```

```
[11, 7, 5, 3, 2]
```

Both indexing and slicing can be used to set elements as well as access them. The syntax is as you would expect:

```
L[0] = 100
print(L)
[100, 3, 5, 7, 11]
L[1:3] = [55, 56]
print(L)
[100, 55, 56, 7, 11]
```

Get used to this slicing syntax: it will become extremely important in later chapters when we begin looking at NumPy arrays and Pandas Dataframes!

Now that we have seen Python lists and how to access elements in ordered compound types, let's take a look at the other three standard compound data types mentioned above.

Tuples

Tuples are in many ways similar to lists, but they are defined with parentheses rather than square brackets:

```
t = (1, 2, 3)
```

They can also be defined without any brackets at all:

```
t = 1, 2, 3
print(t)
(1, 2, 3)
```

Like the lists discussed above, tuples have a length, and individual elements can be extracted using square-bracket indexing:

```
len(t)
3
t[0]
1
```

The main distinguishing feature of tuples is that they are **immutable**: this means that once they are created, their size and contents cannot be changed:

```
t[1] = 4
```

```
-----
TypeError                                         Traceback (most recent call last)

<ipython-input-24-141c76cb54a2> in <module>()
      1 t[1] = 4

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

```
t.append(4)

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-25-e8bd1632f9dd> in <module>()
      1 t.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

Tuples are often used in a Python program; a particularly common case is in functions which have multiple return values. For example, the `as_integer_ratio()` method of floating-point objects returns a numerator and a denominator; this dual return value comes in the form of a tuple:

```
(0.125).as_integer_ratio()
(1, 8)
```

These multiple return values can be individually assigned as follows:

```
numerator, denominator = (0.125).as_integer_ratio()
print(numerator / denominator)

0.125
```

All of the indexing and slicing logic covered above works for tuples as well. In addition, there are a host of other methods available. Refer to the online Python documentation for a more complete list of these.

Dictionaries

Dictionaries are extremely flexible mappings of keys to values, and form the basis of much of Python's internal implementation. They can be created via a comma-separated list of `key:value` pairs within curly braces:

```
numbers = {'one':1, 'two':2, 'three':3}
```

Items are accessed and set via the indexing syntax used for lists and tuples, except here the index is not a zero-based order but valid key in the dictionary:

```
# Access a value via the key
numbers['two']

2
```

New items can be added to the dictionary using indexing as well:

```
# Set a new key, value pair
numbers['ninety'] = 90
print(numbers)

{'one': 1, 'two': 2, 'ninety': 90, 'three': 3}
```

As seen here, dictionaries do not maintain any sense of order for the input parameters; this is by design. This lack of ordering allows dictionaries to be implemented very efficiently, so that random element access is very fast, regardless of the size of the dictionary (if you're curious how this works, look up the concept of a *hash table*). For more information on the methods and functions available to dictionaries, you can refer to the Python documentation.

Sets

Though they are not used quite as often as lists, tuples, and dictionaries, for completeness, we'll briefly mention sets here. Sets are unordered collections of unique items. They are defined much like lists and tuples, except they use the curly brackets of dictionaries:

```
primes = {2, 3, 5, 7}  
odds = {1, 3, 5, 7, 9}
```

If you're familiar with the mathematics of sets, you'll be familiar with operations like the union, intersection, difference, symmetric difference, and others. Python's sets have all of these operations built-in, via methods or operators. For each, we'll show the two equivalent methods:

```
# union: items appearing in either  
primes | odds      # with an operator  
primes.union(odds) # equivalently with a method  
  
{1, 2, 3, 5, 7, 9}  
  
# intersection: items appearing in both  
primes & odds      # with an operator  
primes.intersection(odds) # equivalently with a method  
  
{3, 5, 7}  
  
# difference: items in primes but not in odds  
primes - odds      # with an operator  
primes.difference(odds) # equivalently with a method  
  
{2}  
  
# symmetric difference: items appearing in only one set  
primes ^ odds      # with an operator  
primes.symmetric_difference(odds) # equivalently with a method  
  
{1, 2, 9}
```

Many more set methods and operations are available. You've probably already guessed what I'll say next: refer to Python's online documentation for a complete reference.

More Specialized Data Structures

Python contains several other data structures which you might find useful; these can generally be found in the built-in `collections` module. The `collections` module is fully-documented in Python's online documentation, and you can read more about the various objects available there.

In particular, I've found the following very useful on occasion:

- `collections.namedtuple`: like a tuple, but each value has a name
- `collections.defaultDict`: like a dictionary, but unspecified keys have a user-specified default value
- `collections.OrderedDict`: like a dictionary, but the order of keys is maintained

Once you've seen the standard built-in types, the use of these extended functionalities is very intuitive.

Control Flow

Control flow is where the rubber really meets the road in programming. Without it, a program is simply a list of statements that are sequentially executed. With control flow, you can execute certain code blocks conditionally and/or repeatedly: these basic building blocks can be combined to create surprisingly sophisticated programs!

Here we'll cover **conditional statements** including `if`, `elif`, `else`, and the `pass`, as well as **loop statements** including `for` and `while` and the accompanying `break`, `continue`, and `pass`.

Conditional Statements: `if-elif-else`:

Conditional statements, often referred to as *if-then* statements, allow the programmer to execute certain pieces of code depending on some boolean condition. A basic example of a Python conditional statement is this:

```
x = -15

if x == 0:
    print(x, "is zero")
elif x > 0:
    print(x, "is positive")
elif x < 0:
    print(x, "is negative")
else:
    print(x, "is unlike anything I've ever seen...")

-15 is negative
```

Note especially the use of colons (:) and whitespace to denote separate blocks of code.

Python adopts the `if` and `else` often used in other languages; its more unique keyword is `elif`, a contraction of “else if”. Note that while `if` and `elif` must precede a boolean conditional, the `else` clause is a catch-all that has no condition and may only appear at the end. In these conditional clauses, `elif` and `else` blocks are optional; additionally, you can optionally include as few or as many `elif` statements as you would like.

for loops

Loops in Python are a way to repeatedly execute some code statement. So, for example, if we'd like to print each of the items in a list, we can use a `for` loop:

```
for N in [2, 3, 5, 7]:  
    print(N, end=' ') # print all on same line  
  
2 3 5 7
```

Notice the simplicity of the `for` loop: we specify the variable we want to use, the sequence we want to loop over, and use the `in` operator to link them together in an intuitive way. More precisely, the object to the right of the `in` can be any Python *iterator*. An iterator can be thought of as a generalized sequence, and we'll discuss them further in section X.X.

For example, one of the most commonly-used iterators in Python is the `range` object, which generates a sequence of numbers:

```
for i in range(10):  
    print(i, end=' ')  
  
0 1 2 3 4 5 6 7 8 9
```

Note that the range starts at zero by default, and that by convention the top of the range is not included in the output. Range objects can also have more complicated values:

```
# range from 5 to 10  
list(range(5, 10))  
  
[5, 6, 7, 8, 9]  
  
# range from 0 to 10 by 2  
list(range(0, 10, 2))  
  
[0, 2, 4, 6, 8]
```

You might notice that meaning of `range` arguments is very similar to the slicing syntax that we covered in Section X.X, a fact which is not accidental.

Note that `range()` is one of the small differences between Python 2 and Python 3: in Python 2, `range()` produces a list, while in Python 3, `range()` produces an iterable object.

while loops

The other type of loop in Python is a `while` loop, which iterates until some condition is met:

```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1

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

The argument of the `while` loop is evaluated as a boolean statement, and the loop is executed until the statement evaluates to False.

break and continue: Fine Tuning Your Loops

There are two useful statements that can be used within loops to fine-tune how they are executed:

- the `break` statement breaks-out of the loop entirely
- the `continue` statement skips the remainder of the current loop, and goes to the next iteration

These can be used in both `for` and `while` loops.

Here is an example of using `continue` to print a string of even numbers. In this case the result could be accomplished just as well with an `if-else` statement, but sometimes the `continue` statement can be a more convenient way to express the idea you have in mind:

```
for n in range(20):
    # check if n is even
    if n % 2 == 0:
        continue
    print(n, end=' ')

1 3 5 7 9 11 13 15 17 19
```

Here is an example of a `break` statement used for a less trivial task. This loop will fill a list with all Fibonacci numbers up to a certain value:

```
a, b = 0, 1
amax = 100
L = []
```

```
while True:  
    (a, b) = (b, a + b)  
    if a > amax:  
        break  
    L.append(a)  
  
print(L)  
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Notice that we use a `while True` loop, which will loop forever unless we have a `break` statement!

Loops With an `else` Block

One rarely used pattern available in Python is the `else` statement as part of a `for` or `while` loop. We know the `else` block from above: it executes if all the `if` and `elif` statements evaluate to `False`. This is perhaps one of the more confusingly-named statements in Python; I prefer to think of it as a `nobreak` statement: that is, the `else` block is executed only if the loop ends naturally, without a `break` statement.

As an example of where this might be useful, consider the following (non-optimized) implementation of the Sieve of Eratosthenes, a well-known algorithm for finding primes:

```
L = []  
nmax = 30  
  
for n in range(2, nmax):  
    for factor in L:  
        if n % factor == 0:  
            break  
    else: # no break  
        L.append(n)  
print(L)  
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

It might be useful to think of the `else` block in loops as rather a `nobreak`: statement: it is only executed if there is no `break` in the loop. The `else` statement works similarly with the `while` loop.

Defining and Using Functions

So far our scripts have been simple, single-use code blocks. One way to organize our Python code and to make it more readable and reusable is to factor-out useful pieces into reusable *functions*. Here we'll cover two ways of creating functions: the `def` statement, useful for any type of function, and the `lambda` statement, useful for creating short anonymous functions.

Using Functions

Functions are groups of code which have a name, and can be called using parentheses. We've seen functions before. For example, `print` in Python 3 is a function:

```
print('abc')
```

```
abc
```

Here `print` is the function name, and '`abc`' is the function's *argument*.

In addition to arguments, there are *keyword arguments* which are specified by name. One available keyword argument for the `print()` function (in Python 3) is `sep`, which tells what character or characters should be used to separate multiple items:

```
print(1, 2, 3)
```

```
1 2 3
```

```
print(1, 2, 3, sep='--')
```

```
1--2--3
```

When non-keyword arguments are used together with keyword arguments, the keyword arguments must come at the end.

Defining Functions

Where functions get really useful is when we begin to define our own, organizing functionality to be used in multiple places. In Python, functions are defined with the `def` statement. For example, we can encapsulate a version of our Fibonacci sequence code from the previous section as follows:

```
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

Now we have a function named `fibonacci` which takes a single argument `N`, does something with this argument, and `returns` a value; in this case a list of the first `N` fibonacci numbers:

```
fibonacci(10)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

If you're used to strongly-typed languages like C, you'll immediately notice that there is no type information associated with the function. Python functions can return any Python object, simple or compound, which means constructs that may be difficult in other languages are dead simple in Python.

For example, multiple return values are simply put in a tuple, which is indicated by commas:

```
def real_imag_conj(val):
    return val.real, val.imag, val.conjugate()

r, i, c = real_imag_conj(3 + 4j)
print(r, i, c)
3.0 4.0 (3-4j)
```

Default Argument Values

Often when defining a function, there are certain values that we want the function to use **most** of the time, but we'd also like to give the user some flexibility. In this case, we can use *default values* for arguments. Consider the `fibonacci` function from above. What if we would like the user to be able to play with the starting values? We could do that as follows:

```
def fibonacci(N, a=0, b=1):
    L = []
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

With a single argument, the result of the function call is identical to before:

```
fibonacci(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

But now we can use the function to explore new things, by changing those values:

```
fibonacci(10, 0, 2)
[2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```

The values can also be specified by name if desired, in which case the order of the named values does not matter:

```
fibonacci(10, b=2, a=0)
[2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```

*args and **kwargs: Flexible Arguments

Sometimes you might wish to write a function in which you don't initially know how many arguments the user will pass. In this case, you can use the special form `*args` and `**kwargs` to catch all arguments that are passed. Here is an example:

```
def catch_all(*args, **kwargs):
    print("args =", args)
    print("kwargs = ", kwargs)
```

```
catch_all(1, 2, 3, a=4, b=5)

args = (1, 2, 3)
kwargs = {'b': 5, 'a': 4}

catch_all('a', keyword=2)

args = ('a',)
kwargs = {'keyword': 2}
```

Here it is not the names `args` and `kwargs` which are important, but the `*` characters preceding them. `args` and `kwargs` are just the variable names often used by convention, and are short for “arguments” and “key word arguments”. The operative difference is the asterisk characters: a single `*` before a variable means “expand this sequence, while a double `**` before a variable means “expand this dictionary”. In fact, this syntax can be used not only with the function definition, but with the function call as well!

```
inputs = (1, 2, 3)
keywords = {'pi': 3.14}

catch_all(*inputs, **keywords)

args = (1, 2, 3)
kwargs = {'pi': 3.14}
```

Finally, keep in mind that this `*/**`-expansion syntax is only valid in particular places like function calls and definitions.

Anonymous (`lambda`) Functions

Above we quickly covered the most common way of defining functions, the `def` statement. You’ll likely come across another way of defining short, one-off functions with the `lambda` statement. It looks something like this:

```
add = lambda x, y: x + y
add(1, 2)

3
```

This lambda function is roughly equivalent to

```
def add(x, y):
    return x + y
```

So why would you ever want to use such a thing? Primarily, it comes down to the fact that *everything is an object* in Python, even functions themselves! That means that functions can be passed as arguments to functions.

As an example of this, suppose we have some data stored in a list of lists:

```
data = [{"first": "Guido", "last": "Van Rossum", "YOB": 1956},  
        {"first": "Grace", "last": "Hopper", "YOB": 1906},  
        {"first": "Alan", "last": "Turing", "YOB": 1912}]
```

Now suppose we want to sort this data. Python has a `sorted` function which does this:

```
sorted([2,4,3,5,1,6])  
[1, 2, 3, 4, 5, 6]
```

But dictionaries are not orderable: we need a way to tell the function *how* to sort our data. We can do this by specifying the `key` function, a function which given an item returns the sorting key for that item:

```
# sort alphabetically by first name  
sorted(data, key=lambda item: item['first'])  
[{'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},  
 {'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},  
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]  
  
# sort by year of birth  
sorted(data, key=lambda item: item['YOB'])  
[{'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},  
 {'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},  
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]
```

While these key functions could certainly be created by the normal, `def` syntax, the `lambda` syntax is convenient for such short one-off functions like these.

Errors and Exceptions

One important thing to realize is that no matter how good a coder you are, you will make mistakes. These mistakes come in three basic flavors:

- **Syntax Errors:** Errors where the code is not valid Python (generally easy to fix)
- **Runtime Errors:** Errors where syntactically valid code fails to execute, perhaps due to invalid user input (sometimes easy to fix)
- **Semantic Errors:** Errors in logic: code executes without a problem, but the result is not what you expect (often very difficult to track-down and fix)

Here we're going to focus on how to deal cleanly with the middle one: **runtime errors**. As we'll see, Python handles runtime errors via its *exception handling* framework.

Runtime Errors

If you've done any coding in Python, you've likely come across runtime errors. They can happen in a lot of ways.

For example, if you try to reference an undefined variable:

```
print(Q)
```

```
-----  
NameError                                 Traceback (most recent call last)  
  
<ipython-input-1-e796bdcf24ff> in <module>()  
      1 print(Q)  
  
NameError: name 'Q' is not defined
```

Or if you try an operation that's not defined:

```
1 + 'abc'
```

```
-----  
TypeError                                Traceback (most recent call last)  
  
<ipython-input-2-aab9e8ede4f7> in <module>()  
      1 1 + 'abc'  
  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Or you might be trying another type of undefined operation:

```
2 / 0
```

```
-----  
ZeroDivisionError                          Traceback (most recent call last)  
  
<ipython-input-3-ae0c5d243292> in <module>()  
      1 2 / 0  
  
ZeroDivisionError: division by zero
```

Or maybe you're trying to access a sequence element that doesn't exist:

```
L = [1, 2, 3]  
L[1000]
```

```
-----  
IndexError                                Traceback (most recent call last)  
  
<ipython-input-4-06b6eb1b8957> in <module>()  
      1 L = [1, 2, 3]  
      2 L[1000]
```

```
IndexError: list index out of range
```

Note that in each case, Python is kind enough to not simply indicate that an error happened, but to spit out a *meaningful* exception which includes information about what exactly went wrong, along with the exact line of code where the error happened. Having access to meaningful errors like this is immensely useful when trying to trace the root of problems in your code.

Catching Exceptions: `try` and `except`

The main tool Python gives you for handling runtime exceptions is the `try...except` clause. Its basic structure is this:

```
try:  
    print("this gets executed first")  
except:  
    print("this gets executed only if there is an error")  
  
this gets executed first
```

Note that the second block here did not get executed: this is because the first block did not return an error. Let's put a problematic statement in the `try` block and see what happens:

```
try:  
    print("let's try something:")  
    x = 1 / 0 # ZeroDivisionError  
except:  
    print("something bad happened!")  
  
let's try something:  
something bad happened!
```

Here we see that when the error was raised in the `try` statement (in this case a `ZeroDivisionError`), the error was caught, and the `except` statement was executed.

One way this is often used is to check user input within a function or another piece of code. For example, we might wish to have a function which catches zero-division and returns some other value, perhaps a suitably large number like 10^{100}

```
def safe_divide(a, b):  
    try:  
        return a / b  
    except:  
        return 1E100  
  
safe_divide(1, 2)  
0.5  
safe_divide(2, 0)
```

```
1e+100
```

There is a subtle problem with this code, though: what happens when another type of exception comes up? For example, this is probably not what we intended:

```
safe_divide(1, '2')
```

```
1e+100
```

Dividing an integer and a string raises a `TypeError`, which our over-zealous code caught and assumed was a `ZeroDivisionError`! For this reason, it's nearly always a better idea to catch exceptions *explicitly*:

```
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return 1E100

safe_divide(1, 0)
1e+100
safe_divide(1, '2')

-----
```

```
TypeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-13-2331af6a0acf> in <module>()
----> 1 safe_divide(1, '2')
```

```
<ipython-input-11-10b5f0163af8> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 1E100
```

```
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

We're now catching zero-division errors only, and letting all other errors pass through un-modified.

Raising Exceptions: `raise`

We've seen how valuable it is to have informative exceptions when using parts of the Python language. It's equally valuable to make use of informative exceptions within the code you write, so that users of your code (foremost yourself!) can figure out what caused their errors.

The way you raise your own exceptions is with the `raise` statement. For example:

```
raise RuntimeError("my error message")
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-14-c6a4c1ed2f34> in <module>()
      1 raise RuntimeError("my error message")
RuntimeError: my error message
```

As an example of where this might be useful, let's return to our `fibonacci` function that we defined previously:

```
def fibonacci(N):
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
```

One potential problem here is that the input value could be negative. This will not currently cause any error in our function, but we might want to let the user know that a negative `N` is not supported. Errors in values passed to function should, by convention, lead to a `ValueError` being raised:

```
def fibonacci(N):
    if N < 0:
        raise ValueError("N must be non-negative")
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L

fibonacci(10)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
fibonacci(-10)
-----
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-18-3d291499cf7> in <module>()
      1 fibonacci(-10)

<ipython-input-16-01d0cf168d63> in fibonacci(N)
      1 def fibonacci(N):
```

```
2     if N < 0:  
----> 3         raise ValueError("N must be non-negative")  
4     L = []  
5     a, b = 0, 1
```

```
ValueError: N must be non-negative
```

Now the user knows exactly why the input is invalid, and could even use a `try...except` block to handle it!

```
N = -10  
try:  
    print("trying this...")  
    print(fibonacci(N))  
except ValueError:  
    print("Bad value: need to do something else")  
  
trying this...  
Bad value: need to do something else
```

Advanced Topics

Briefly, I want to mention here some other concepts you might run into. I'll not go into detail on these concepts and how and why to use them, but instead simply show you the syntax so you can explore more on your own.

Accessing the Error Message

Sometimes in a `try...except` statement, you would like to be able to work with the error message itself. This can be done with the `as` keyword:

```
try:  
    x = 1 / 0  
except ZeroDivisionError as err:  
    print("Error class is: ", type(err))  
    print("Error message is:", err)  
  
Error class is: <class 'ZeroDivisionError'>  
Error message is: division by zero
```

With this pattern, you can further customize the exception handling of your function.

Defining Custom Exceptions

In addition to built-in exceptions, it is possible to define custom exceptions through *class inheritance*. For instance, if you want a special kind of `ValueError`, you can do this:

```
class MySpecialError(ValueError):  
    pass
```

```
raise MySpecialError("here's the message")
-----
MySpecialError                                     Traceback (most recent call last)

<ipython-input-21-92c36e04a9d0> in <module>()
      2     pass
      3
----> 4 raise MySpecialError("here's the message")

MySpecialError: here's the message
```

This would allow you to use a `try...except` block which only catches this type of error:

```
try:
    print("do something")
    raise MySpecialError("[informative error message here]")
except MySpecialError:
    print("do something else")

do something
do something else
```

You might find this useful as you develop more complicated and specialized code.

try...except...else...finally

In addition to `try` and `except`, you can use the `else` and `finally` keywords to further tune your code's handling of exceptions. The basic structure is this:

```
try:
    print("try something here")
except:
    print("this happens only if it fails")
else:
    print("this happens only if it succeeds")
finally:
    print("this happens no matter what")

try something here
this happens only if it succeeds
this happens no matter what
```

The utility of `else` here is clear, but what's the point of `finally`? Well, the `finally` clause really is executed *no matter what*, and this can be useful for running some type of cleanup code after the `return` statement in a function.

For an example of where this might be useful, we'll have to look ahead a bit and preview some of the more complicated data structures we'll come across later in the

book. As we'll discuss in Section X.X, representation of missing data is often a contentious question, and different libraries might represent missing data in different ways.

Imagine, for example, we have a dataset in which missing data is represented by the sentinel value 9999. Using the NumPy array syntax which will be introduced more fully in Section X.X, this data might look something like this:

```
import numpy
data = numpy.array([1.0, 3.5, 9999, 2.1, 4.4, 9999])
```

Now suppose we'd like to compute the mean of the defined values, without copying the data (because, say, the data is so large that you cannot hold two copies of it in memory). Using Numpy's built-in `mean()` function doesn't work: it treats the 9999 as part of the array:

```
numpy.mean(data)
3334.8333333333335
```

Numpy has a `nanmean()` function which can compute a mean while ignoring missing data, but the data must be marked with the special `numpy.nan` (Not A Number) value:

```
data2 = numpy.array([1.0, 3.5, numpy.nan, 2.1, 4.4, numpy.nan])
numpy.nanmean(data2)
2.75
```

So how do we proceed? We might start with functions that convert between the two representations. These use the Pandas library (see Section X.X) and NumPy's *masking* functionality (see Section X.X):

```
import pandas

def convert_to_nan_rep(data):
    data[data == 9999] = numpy.nan

def convert_to_9999_rep(data):
    data[pandas.isnull(data)] = 9999
```

we can now do a bit of a pipeline: convert our data inplace, use the `nanmean` function, then convert back:

```
def safe_mean(data):
    convert_to_nan_rep(data)
    ret = numpy.nanmean(data)
    convert_to_9999_rep(data)
    return ret

safe_mean(data)
2.75
```

Now we've computed the result we want efficiently, and at the end of the operation the data is back to its original form. There is a problem here, though: what if the `nanmean` function leads to an error? Then the data conversion may not be properly undone, which could cause problems later in the program. For example, if the data contains a string, the `nanmean` function will fail:

```
bad_data = numpy.array([1, 2, 9999, 3, 'dog'], dtype=object)
try:
    safe_mean(bad_data)
except:
    print("failed")
failed
```

Our function failed, but it had a side-effect: the data has been modified!

```
bad_data
array([1, 2, nan, 3, 'dog'], dtype=object)
```

This could cause problems later in your program if you are not expecting the missing data to be in this form!

Situations like these are where a `try...finally` clause can be useful. Here's how we could modify our `safe_mean` function to *always reset the data*, even if `nanmean` leads to an error:

```
def safe_mean(data):
    convert_to_nan_rep(data)
    try:
        return np.nanmean(data)
    finally:
        convert_to_9999_rep(data)
```

Now, *no matter what happens in the try block*, the data re-conversion will be run after it was run. For example:

```
bad_data = numpy.array([1, 2, 9999, 3, 'dog'], dtype=object)

try:
    safe_mean(bad_data)
except:
    print('failed')
bad_data
failed
```

```
array([1, 2, 9999, 3, 'dog'], dtype=object)
```

Our `bad_data` array is back to how it started, even though the operation failed.

Hopefully this gives you an idea of how the `finally` statement could be useful in a real-world data science context.

Iterators

Often an important piece of data analysis is repeating a similar calculation, over and over, in an automated fashion. For example, you may have a table of names which you'd like to split into first and last, or perhaps of dates that you'd like to convert to some standard format. One of Python's answers to this is the *iterator* syntax. We've seen this already in Section X.X. when we discussed `for` loops:

```
for i in range(10):
    print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
```

Here we're going to dig a bit deeper into what is happening here. It turns out that in Python 3, `range` is not a list, but is something called an *iterator*, and learning how it works is a key to understanding a wide class of very useful Python functionality.

Iterating over lists

Iterators are perhaps most easily understood in the concrete case of iterating through a list. Consider the following:

```
for value in [2, 4, 6, 8, 10]:
    # do some operation
    print(value + 1, end=' ')
3 5 7 9 11
```

The familiar "`for x in y`" syntax allows us to repeat some operation for each value in the list. The fact that the syntax of the code is so close to its English description ("*for [each] value in [the] list*") is just one of the syntactic choices that makes Python such an intuitive language to learn and use.

But the face-value behavior is not what's *really* happening. When you write something like "`for val in L`", the Python interpreter checks whether it has an *iterator* interface, which you can check yourself with the built-in `iter` function:

```
iter([2, 4, 6, 8, 10])
<list_iterator at 0x102789250>
```

It is this iterator object which provides the functionality required by the `for` loop. The `iter` object is a container that gives you access to the next object for as long as it's valid, which can be seen with the built-in function `next`:

```
I = iter([2, 4, 6, 8, 10])
print(next(I))
```

```
print(next(I))
print(next(I))

2
4
6
```

What is the purpose of this level of indirection? Well, it turns out this is incredibly useful, because it allows Python to treat things as lists that are *not actually lists*.

range(): A List is Not Always a List

Perhaps the best known example of this indirect iteration is the `range()` function in Python 3 (named `xrange()` in Python 2), which returns not a list, but a special `range()` object:

```
range(10)
range(0, 10)
```

`range`, like a list, exposes an iterator:

```
iter(range(10))
<range_iterator at 0x1025a51e0>
```

So Python knows to treat it *as if* it's a list:

```
for i in range(10):
    print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
```

The benefit of the iterator indirection is that *the full list is never explicitly created!* We can see this by doing a range calculation that would overwhelm our system memory if we actually instantiated it (note that in Python 2, `range` creates a list, so running the following will not lead to good things!):

```
for i in range(10 ** 12):
    if i >= 10: break
    print(i, end=', ')
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

If `range` were to actually create that list of one trillion values, it would occupy tens of terabytes of machine memory: a waste, given the fact that we're throwing away all but the first ten values!

In fact, there's no reason that iterators ever have to end at all! Python's `itertools` library contains a `count` function which acts as an infinite list:

```
from itertools import count
for i in count():
    if i >= 10:
```

```
        break
    print(i, end=' ')
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Had we not thrown-in a loop break here, it would go on happily counting until the process is interrupted or killed.

Useful Iterators

This iterator syntax is used nearly universally in Python built-in types as well as the more data-science-specific object we'll explore in later chapters. Here we'll cover some of the more useful iterators in the Python language:

`enumerate`

Often you need to iterate not only the values in an array, but also keep track of the index. You might be tempted to do things this way:

```
L = [2, 4, 6, 8, 10]
for i in range(len(L)):
    print(i, L[i])

0 2
1 4
2 6
3 8
4 10
```

While this works, Python provides a cleaner syntax using the `enumerate` iterator:

```
for i, val in enumerate(L):
    print(i, val)

0 2
1 4
2 6
3 8
4 10
```

This is the more “Pythonic” way to enumerate the indices and values in a list.

`zip`

Other times, you may have multiple lists that you want to iterate over simultaneously. You could certainly iterate over the index as in the non-Pythonic example above, but it is better to use the `zip` iterator, which zips together iterables:

```
L = [2, 4, 6, 8, 10]
R = [3, 6, 9, 12, 15]
for lval, rval in zip(L, R):
    print(lval, rval)
```

```
2 3  
4 6  
6 9  
8 12  
10 15
```

Any number of iterables can be zipped together, and if they are different lengths, the shortest will determine the length of the `zip`.

map and filter

The `map` iterator takes a function and applies it to the values in an iterator:

```
# find the first ten square numbers  
square = lambda x: x ** 2  
for val in map(square, range(10)):  
    print(val, end=' ')  
  
0 1 4 9 16 25 36 49 64 81
```

The `filter` iterator looks similar, except it only passes-through values for which the filter function evaluates to True:

```
# find values up to 10 for which x % 2 is zero  
is_even = lambda x: x % 2 == 0  
for val in filter(is_even, range(10)):  
    print(val, end=' ')  
  
0 2 4 6 8
```

The `map` and `filter` functions, along with the `reduce` function (which lives in Python's `functools` module) are fundamental components of the *functional programming* style, which, while not a dominant programming style in the Python world, has its outspoken proponents (see for example the `pytoolz` library).

Iterators as function arguments

We saw in section X.X. that `*args` and `**kwargs` can be used to pass sequences and dictionaries to functions. It turns out that the `*args` syntax works not just with sequences, but with any iterator:

```
print(*range(10))  
  
0 1 2 3 4 5 6 7 8 9
```

So, for example, we can get tricky and compress the `map` example above into the following:

```
print(*map(lambda x: x ** 2, range(10)))  
  
0 1 4 9 16 25 36 49 64 81
```

Using this trick lets us answer the age-old question that comes up in Python learners' forums: why is there no `unzip()` function which does the opposite of `zip()`? If you lock yourself in a dark closet and think about it for a while, you might realize that the opposite of `zip()` is... `zip()!` The key is that `zip()` can zip-together any number of iterators or sequences. Observe:

```
L1 = (1, 2, 3, 4)
L2 = ('a', 'b', 'c', 'd')

z = zip(L1, L2)
print(*z)

(1, 'a') (2, 'b') (3, 'c') (4, 'd')

z = zip(L1, L2)
new_L1, new_L2 = zip(*z)
print(new_L1, new_L2)

(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

Think on this for a while. If you understand why it works, you'll have come a long way in understanding Python iterators!

Advanced Iterators: `itertools`

Above we briefly looked at the infinite `range` iterator, `itertools.count`. The `itertools` module contains a whole host of useful iterators; it's well worth your while to take explore the module to see what's available. As an example, consider the `itertools.permutations` function, which iterates over all permutations of a sequence:

```
from itertools import permutations
p = permutations(range(3))
print(*p)

(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1, 0)
```

Similarly, the `itertools.combinations` function which iterates over all unique combinations of N values within a list:

```
from itertools import combinations
c = combinations(range(4), 2)
print(*c)

(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
```

Somewhat related is the `product` iterator, which iterates over all sets of pairs between two or more iterables:

```
from itertools import product
p = product('ab', range(3))
print(*p)

('a', 0) ('a', 1) ('a', 2) ('b', 0) ('b', 1) ('b', 2)
```

In section X.X, this `product` iterator will become very useful when we iterate over a grid of machine learning model parameters to find the best model for our data.

Many more useful iterators exist in `itertools`: the full list can be found, along with some examples, in Python's online documentation.

List Comprehensions

If you read enough Python code, you'll eventually come across the extremely terse and efficient construction known as a *list comprehension*. This is one feature of Python I expect you will fall in love with if you've not used it before; it looks something like this:

```
[i for i in range(20) if i % 3 > 0]  
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The result of this is a list of numbers which excludes multiples of 3. While this example may seem a bit confusing at first, as your Python skills grow, reading and writing list comprehensions will become second nature.

Basic List Comprehensions

List comprehensions are simply a way to compress a list-building for-loop into a single short line. For example, here is a loop which constructs a list of the first twelve square integers:

```
L = []  
for n in range(12):  
    L.append(n ** 2)  
L  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

The list comprehension equivalent of this is the following:

```
[n ** 2 for n in range(12)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

As with many Python statements, you can almost read-off the meaning of this statement in plain English: “construct a list consisting of the square of `n` for each `n` up to 12”.

This basic syntax, then, is `[expr for var in iterable]`, where `expr` is any valid expression, `var` is a variable name, and `iterable` is any iterable Python object.

Multiple Iteration

Sometimes you want to build a list not just from one value, but from two. To do this, simply add another `for` expression in the comprehension:

```
[(i, j) for i in range(2) for j in range(3)]  
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Notice that the second `for` expression acts as the interior index, varying the fastest in the resulting list. This type of construction can be extended to 3, 4, or more iterators within the comprehension, though at some point code readability will suffer!

Conditionals on the Iterator

You can further control the iteration by adding a conditional to the end of the expression. In the first example of the section, we iterated over all numbers from 1 to 20, but left-out multiples of 3. Look at this again, and notice the construction:

```
[val for val in range(20) if val % 3 > 0]  
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

The expression `(i % 3 > 0)` evaluates to `True` when unless `val` is divisible by 3. Again, the English language meaning can be immediately read off: “Construct a list of values for each value up to 20, but only if the value is not divisible by 3”. Once you are used to it, this is much easier to write – and to understand at a glance – than the equivalent loop syntax:

```
L = []  
for val in range(20):  
    if val % 3:  
        L.append(val)  
L  
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Conditionals on the Value

If you’ve programmed in C, you might be used to the single-line conditional enabled by the `?` operator:

```
int absval = (val < 0) ? -val : val
```

Python has something very similar to this, which is most often used within list comprehensions, `lambda` functions, and other places where a simple expression is desired:

```
val = -10  
val if val >= 0 else -val  
10
```

We see that this simply duplicates the functionality of the built-in `abs()` function, but the construction lets you do some really interesting things within list comprehensions. This is getting pretty complicated now, but you could do something like this:

```
[val if val % 2 else -val  
  for val in range(20) if val % 3]  
  
[1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16, 17, 19]
```

Note the line break within the list comprehension before the `for` expression: this is valid in Python, and is often a nice way to break-up long list comprehensions for greater readability. Look this over: what we're doing is constructing a list, leaving out multiples of 3, and negating all mutliples of 2.

With this, you now know everything you need to know to solve the “Fizz Buzz Test” in a single line of Python. Fizz-buzz is a classic question to test basic coding ability:

Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.

In drawn-out code, it might look like this for the first fifteen items:

```
for i in range(1, 16):  
    if i % 3 == 0:  
        if i % 5 == 0:  
            print('FizzBuzz', end=' ')  
        else:  
            print('Fizz', end=' ')  
    elif i % 5 == 0:  
        print('Buzz', end=' ')  
    else:  
        print(i, end=' ')  
  
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz
```

See if you can use a list comprehension with conditional expressions to write Fizz Buzz in one line. It may not be pretty, but it's a good way to test how well you can do list comprehensions!

Other Types of Comprehensions

Once you understand the dynamics of list comprehensions, it's straightforward to move on to other types of comprehensions. The syntax is largely, the same, the only difference is the type of bracket you use.

Let's go back to our list of squares:

```
[n ** 2 for n in range(12)]  
  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Set Comprehension

We can create a `set` rather than a `list` by using curly braces:

```
{n**2 for n in range(12)}  
{0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121}
```

Recall that a `set` is a collection which contains no duplicates. The set comprehension respects this rule, and eliminates any duplicate entries:

```
{a % 3 for a in range(1000)}  
{0, 1, 2}
```

Dict Comprehension

In Python 3, you can also directly create a dictionary with the comprehension syntax. The key is to use curly braces, and use a colon (`:`) to separate keys and values:

```
{n:n**2 for n in range(6)}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

In Python 2, dict comprehensions are not supported, but you can do something very similar with the `dict` constructor:

```
dict((n, n**2) for n in range(6))  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

This is actually an example of a *generator expression*.

Generator Expressions

Finally, if you use standard parentheses rather than square brackets, you get what's called a generator expression:

```
(n**2 for n in range(12))  
<generator object <genexpr> at 0x1027a5a50>
```

A generator expression is essentially a lazily-evaluated list comprehension. Just as we saw that `range()` doesn't construct a list, but provides an interface to iterate over values as if it's a list, a generator expression behaves like a list comprehension, but produces the elements on demand rather than all at once. So, for example, you can use the `* print` pattern covered in the previous section to see the items:

```
a = (n**2 for n in range(12))  
print(*a)  
0 1 4 9 16 25 36 49 64 81 100 121
```

Generators and generator expressions are an incredibly powerful language feature, and we'll explore them more fully in the next section.

Generators

We finished the last section showing how a tiny change in type of brace can turn a *list comprehension* into a *generator expression*. Here we'll dig a bit further into Generator expressions, as well as *generator functions*, another way to produce generators.

List Comprehensions vs Generator Expressions

The difference between list comprehensions and generator expressions is sometimes confusing; here we'll quickly list the differences between them:

1. List comprehensions use square brackets, while generator expressions use parentheses

This is a representative list comprehension:

```
[n ** 2 for n in range(12)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

While this is a representative generator expression:

```
(n ** 2 for n in range(12))  
<generator object <genexpr> at 0x1027a3230>
```

Notice that printing the generator expression does not print the contents; one way to print the contents of a generator expression is to pass it to the `list` constructor:

```
G = (n ** 2 for n in range(12))  
list(G)  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

2. A list is a collection of values, while a generator is a recipe for producing values

This is important to keep in mind: when you create a list, you are actually building a collection of values, and there is some memory cost associated with that. When you create a generator, you are not building a collection of values, but a recipe for producing those values. Both expose the same iterator interface, as we can see here:

```
L = [n ** 2 for n in range(12)]  
for val in L:  
    print(val, end=' ')  
  
0 1 4 9 16 25 36 49 64 81 100 121  
  
G = (n ** 2 for n in range(12))  
for val in G:  
    print(val, end=' ')  
  
0 1 4 9 16 25 36 49 64 81 100 121
```

The difference is that a generator expression does not actually compute the values until they are needed. This not only leads to memory efficiency, but to computational

efficiency as well! This also means that while the size of a list is limited by available memory, the size of a generator expression is unlimited!

An example of an infinite generator expression is the `count` iterator defined in `itertools`:

```
from itertools import count
count()
count(0)

for i in count():
    print(i, end=' ')
    if i >= 10: break

0 1 2 3 4 5 6 7 8 9 10
```

The `count` iterator will go on happily counting forever until you tell it to stop; this makes it convenient to create generators which will also go on forever:

```
factors = [2, 3, 5, 7]
G = (i for i in count(10) if all(i % n > 0 for n in factors))
for val in G:
    print(val, end=' ')
    if val > 40: break

11 13 17 19 23 29 31 37 41
```

You might see what we're getting at here: if we were to expand the list of factors appropriately, what we would have the beginnings of is a prime number generator, using the well-known Sieve of Eratosthenes. We'll explore this more below.

3. A list can be iterated multiple times; a generator expression is single-use

This is one of those potential gotchas of generator expressions. With a list, we can straightforwardly do this:

```
L = [n ** 2 for n in range(12)]
for val in L:
    print(val, end=' ')
print()

for val in L:
    print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

A generator expression, on the other hand, is used-up after one iteration:

```
G = (n ** 2 for n in range(12))
print(list(G))
print(list(G))
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[]
```

This can be very useful because it means iteration can be stopped and started

```
G = (n**2 for n in range(12))
for n in G:
    print(n, end=' ')
    if n > 30: break

print("\ndoing something in between")

for n in G:
    print(n, end=' ')

0 1 4 9 16 25 36
doing something in between
49 64 81 100 121
```

One place I've found this useful is when working with collections of data files on disk; it means that you can quite easily analyze them in batches, letting the generator keep track of which ones you have yet to see.

Generator Functions: `yield`

We saw in the previous section that list comprehensions are best used to create relatively simple lists, while using a normal `for` loop can be better in more complicated situations. The same is true of generator expressions: we can make more complicated generators using *generator functions*, which make use of the `yield` statement.

Here we have two ways of constructing the same list:

```
L1 = [n ** 2 for n in range(12)]

L2 = []
for n in range(12):
    L2.append(n ** 2)

print(L1)
print(L2)

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Similarly, here we have two ways of constructing equivalent generators:

```
G1 = (n ** 2 for n in range(12))

def gen():
    for n in range(12):
        yield n ** 2

G2 = gen()
```

```

print(*G1)
print(*G2)

0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121

```

A generator function is a function which, rather than using `return` to return a value once, uses `yield` to yield a (potentially infinite) string of values. Just as in generator expressions, the state of the generator is preserved between partial iterations, but if we want a fresh copy of the generator we can just call the function again!

Example: Prime Number Generator

Here I'll show my favorite example of a generator function: a function to generate an unbounded series of prime numbers. A classic algorithm for this is the *Sieve of Eratosthenes*, which works something like this:

```

# Generate a list of candidates
L = [n for n in range(2, 40)]
print(L)

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]

# Remove all multiples of the first value
L = [n for n in L if n == L[0] or n % L[0] > 0]
print(L)

[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39]

# Remove all multiples of the second value
L = [n for n in L if n == L[1] or n % L[1] > 0]
print(L)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37]

# Remove all multiples of the third value
L = [n for n in L if n == L[2] or n % L[2] > 0]
print(L)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]

```

If we repeat this procedure enough times on a large enough list, we can generate as many primes as we wish!

Let's encapsulate this logic in a generator function:

```

def gen_primes(N):
    """Generate primes up to N"""
    primes = set()
    for n in range(2, N):
        if all(n % p > 0 for p in primes):
            primes.add(n)
            yield n

print(*gen_primes(100))

```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

That's all there is to it! While this is certainly not the most computationally efficient implementation of the Sieve of Eratosthenes, it illustrates how convenient the generator function syntax can be for building more complicated sequences.

As an exercise, see if you can modify this generator expression so that rather than returning primes with values less than N , it returns the first N prime numbers. You might want to use the `itertools.count` function we used above, as well as realizing that once the loop ends and no more `yield` statements are encountered, the generator will be finished.

Once you've done this, it's a short leap to write an *infinite prime number generator*, which will go on generating primes until you tell it to stop.

Modules and Packages

One feature of Python that makes it useful for a wide range of tasks is the fact that it comes “batteries included” – that is, the Python standard library contains useful tools for a wide range of tasks. On top of this, there is a broad ecosystem of third party tools and packages which can be easily installed with a variety of tools. Here we'll take a look at importing standard library modules, tools for installing third-party modules, and a description of how you can make your own modules.

Loading Modules: the `import` Statement

For loading built-in and third-party modules, Python provides the `import` statement. There are a few ways to use the statement, which we will mention briefly here, from most recommended to least recommended.

1. Explicit module import

Explicit import of a module preserves the module's content in a namespace. The namespace is then used to refer to its contents with a `."` between them. For example, here we'll import the built-in `math` module and compute the sine of pi:

```
import math
math.cos(math.pi)

-1.0
```

2. Explicit module import by alias

For longer module names, it's not convenient to use the full module name each time you access some element. For this reason, we'll commonly use the `"import ... as ..."` pattern to create a shorter alias for the namespace. For example, the `numpy`

package, which will be the subject of chapter X.X, is by convention imported under the alias np:

```
import numpy as np
np.cos(np.pi)

-1.0
```

3. Explicit import of module contents

Sometimes rather than importing the module namespace, you would just like to import a few particular items from the module. This can be done with the "from ... import ..." pattern. For example, we can import just the cos function and the pi constant from the math module:

```
from math import cos, pi
cos(pi)

-1.0
```

4. Implicit import of module contents

Finally, it is sometimes useful to import the entirety of the module contents into the local namespace. This can be done with the "from ... import *" pattern:

```
from math import *
sin(pi) ** 2 + cos(pi) ** 2

1.0
```

This pattern should be used sparingly, if at all. The problem is that such imports can sometimes overwrite function names that you do not intend to overwrite, and the implicitness of the statement makes it difficult to determine what has changed.

For example, Python has a built-in sum function which can be used for various operations:

```
help(sum)
Help on built-in function sum in module builtins:

sum(...)
    sum(iterable[, start]) -> value

    Return the sum of an iterable of numbers (NOT strings) plus the value
    of parameter 'start' (which defaults to 0). When the iterable is
    empty, return start.
```

We can use this to compute the sum of a sequence, starting with a certain value (here, we'll start with -1):

```
sum(range(5), -1)

9
```

Now observe what happens if we make the *exact same function call* after importing * from numpy:

```
from numpy import *
sum(range(5), -1)
10
```

The result is off by one! The reason for this is that the `import *` statement *overwrites* the built-in `sum` function with the `numpy.sum` function, which has a different call signature:

```
sum(a, axis=None, dtype=None, out=None, keepdims=False)
    Sum of array elements over a given axis.

Parameters
-----
a : array_like
    Elements to sum.
axis : None or int or tuple of ints, optional
    Axis or axes along which a sum is performed.
```

In this case, the second argument is not a starting value, but an axis argument (in this case, the first axis from the end – see section X.X for details on numpy array sums). This type of bug can cause some really subtle and unexpected issues in your code!

This is the type of situation which may arise if you are not careful when using "`import *`". For this reason, it is best to avoid this unless you know exactly what you are doing.

Python's Standard Library

Python's standard library contains many useful built-in modules, which you can read about fully in [Python's documentation](#). Any of these can be imported with the `import` statement, and then explored using IPython's help features discussed in section X.X. Here is an extremely incomplete list of some of the modules you might wish to explore and learn about:

- `os` and `sys`: tools for interfacing with the operating system, including navigating file directory structures and executing shell commands
- `math` and `cmath`: mathematical functions and operations on real and complex numbers
- `itertools`: tools for constructing and interacting with iterators and generators
- `functools`: tools which assist with functional programming
- `random`: tools for generating pseudorandom numbers
- `pickle`: tools for object persistence: saving objects to and loading objects from disk

- `json` and `csv`: tools for reading JSON-formatted and CSV-formatted files.
- `urllib`: tools for doing HTTP and other web requests.

You can find information on these, and many more, in the Python standard library documentation: <https://docs.python.org/3/library/>.

Third-party modules

One of the things that makes Python the most useful, especially within the world of data science, is its ecosystem of third-party modules. These can be imported just as the built-in modules above, but first the modules must be installed on your system. The standard registry for such modules is the Python Package Index (*PyPI* for short) found on the web at <http://pypi.python.org/>. For convenience, Python comes with a program called `pip` (a recursive acronym meaning “`pip` installs packages”) which will automatically fetch packages released and listed on PyPI (if you use Python version 2, `pip` must be installed separately). For example, if you’d like to install the `supersmooth` package that I wrote, all that is required is to type at the command line

```
$ pip install supersmooth
```

The source code for the package will be automatically downloaded from the PyPI repository, and the package installed in the standard Python path (assuming you have permission to do so on the computer you’re using).

For more information about PyPI and the `pip` installer, refer to the documentation at <http://pypi.python.org/>.

String Manipulation and Regular Expressions

One place where the Python language really shines is in the manipulation of strings. This section will cover some of Python’s built-in string methods and formatting operations, before moving on to a quick guide to the extremely useful subject of *regular expressions*. Such string manipulation patterns come up often in the context of data science work, and we will return to these ideas in Section X.X when we discuss string manipulation with the Pandas package.

Simple String Manipulation in Python

For basic manipulation of strings, Python’s built-in string methods can be extremely convenient. If you have a background working in C or another low-level language, you will likely find the simplicity of Python’s methods extremely refreshing. We introduced Python’s string type and a few of these methods in Section X.X; here we’ll dive a bit deeper.

Formatting Strings: Adjusting Case

Python makes it quite easy to adjust the case of a string. Here we'll look at the `upper()`, `lower()`, `capitalize()`, `title()`, and `swapcase()` methods, using the following messy string as an example:

```
fox = "tHe qUICk bROWN fOX."
```

To convert the entire string into upper-case or lower-case, you can use the `upper()` or `lower()` methods respectively:

```
fox.upper()  
'THE QUICK BROWN FOX.'  
fox.lower()  
'the quick brown fox.'
```

A common formatting need is to capitalize just the first letter of each word, or perhaps the first letter of each sentence. This can be done with the `title()` and `capitalize()` methods:

```
fox.title()  
'The Quick Brown Fox.'  
fox.capitalize()  
'The quick brown fox.'
```

The cases can be swapped using the `swapcase()` method:

```
fox.swapcase()  
'ThE QuicK BrowN FoX.'
```

Formatting Strings: Adding and Removing Spaces

Another common need is to remove spaces (or other characters) from the beginning or end of the string. The basic method of removing characters is the `strip()` method, which strips whitespace from the beginning and end of the line:

```
line = '      this is the content      '  
line.strip()  
'this is the content'
```

To remove just space to the right or left, use `rstrip()` or `lstrip()` respectively:

```
line.rstrip()  
'      this is the content'  
line.lstrip()  
'this is the content      '
```

To remove characters other than spaces, you can pass the desired character to the `strip()` method:

```
num = "00000000000435"
num.strip('0')
'435'
```

The opposite of this operation, adding spaces or other characters, can be accomplished using the `center()`, `ljust()`, and `rjust()` methods.

For example, we can use the `center()` method to center a given string within a given number of spaces:

```
line = "this is the content"
line.center(30)
'      this is the content      '
```

Similarly, `ljust()` and `rjust()` will left-justify or right-justify the string within spaces of a given length:

```
line.ljust(30)
'this is the content          '
line.rjust(30)
'          this is the content'
```

All these methods additionally accept any character which will be used to fill the space. For example:

```
'435'.rjust(10, '0')
'0000000435'
```

Because zero-filling is such a common need, Python also provides `zfill()`, which is a special method to right-pad a string with zeros:

```
'435'.zfill(10)
'0000000435'
```

Finding and Replacing Substrings

If you want to find occurrences of a certain character in a string, the `find()`/`rfind()`, `index()`/`rindex()`, and `replace()` methods are the best built-in methods.

`find()` and `index()` are very similar, in that they search for the first occurrence of a character or substring within a string, and return the index of the substring:

```
line = 'the quick brown fox jumped over a lazy dog'
line.find('fox')
```

```
line.index('fox')
```

```
16
```

The only difference between `find()` and `index()` is their behavior when the search string is not found; `find()` returns `-1`, while `index()` raises a `ValueError`:

```
line.find('bear')  
-1  
try:  
    line.index('bear')  
except ValueError as e:  
    print("ValueError:", e)  
  
ValueError: substring not found
```

The related `rfind()` and `rindex()` work similarly, except they search for the first occurrence from the end rather than the beginning of the string:

```
line.rfind('a')  
35
```

For the special case of checking for a substring at the beginning or end of a string, Python provides the `startswith()` and `endswith()` methods:

```
line.endswith('dog')  
True  
line.startswith('fox')  
False
```

To go one step further and replace a given substring with a new string, you can use the `replace()` method. Here, let's replace `'brown'` with `'red'`:

```
line.replace('brown', 'red')  
'the quick red fox jumped over a lazy dog'
```

The `replace()` function returns a new string, and will replace all occurrences of the input:

```
line.replace('o', '--')  
'the quick br--wn f--x jumped --ver a lazy d--g'
```

For a more flexible approach to this `replace()` functionality, see the discussion of regular expressions below.

Splitting and Partitioning Strings

If you would like to find a substring *and then* split the string based on its location, the `partition()` and/or `split()` methods are what you're looking for. Both will return a sequence of substrings.

The `partition()` method returns a tuple with three elements: the substring before the first instance of the split-point, the split-point itself, and the substring after:

```
line.partition('fox')
('the quick brown ', 'fox', ' jumped over a lazy dog')
```

The `rpartition()` method is similar, but searches from the right of the string.

The `split()` method is perhaps more useful; it finds *all* instances of the split-point and returns the substrings in between. The default is to split on any whitespace, returning a list of the individual words in a string:

```
line.split()
['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
```

A related method is `splitlines()`, which splits on newline characters. Let's do this with a Haiku, popularly attributed to the 17th-century poet Matsuo Bashō:

```
haiku = """matsushima-ya
aah matsushima-ya
matsushima-ya"""
haiku.splitlines()
['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

Note that if you would like to undo a `split()`, you can use the `join()` method, which returns a string built from a splitpoint and an iterable:

```
'--'.join(['1', '2', '3'])
'1--2--3'
```

A common pattern is to use the special character "\n" (newline) to join together lines that have been previously split, and recover the input:

```
print("\n".join(['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']))
matsushima-ya
aah matsushima-ya
matsushima-ya
```

Format Strings

In the above methods we have learned how to extract values from strings, and to manipulate strings themselves into desired formats. Another use of string methods is

to manipulate string *representations* of values of other types. Of course, string representations can always be found using the `str()` function; for example:

```
pi = 3.14159
str(pi)
'3.14159'
```

For more complicated formats, you might be tempted to use string arithmetic as outlined in Section X.X:

```
"The value of pi is " + str(pi)
'The value of pi is 3.14159'
```

A more flexible way to do this is to use *format strings*, which are strings with special markers (noted by curly braces) into which string-formatted values will be inserted. Here is a basic example:

```
"The value of pi is {}".format(pi)
'The value of pi is 3.14159'
```

Inside the `{}` marker you can also include information on exactly *what* you would like to appear there. If you include a number, it will refer to the index of the argument to insert:

```
"""First letter: {0}. Last letter: {1}""".format('A', 'Z')
'First letter: A. Last letter: Z.'
```

If you include a string, it will refer to the key of any keyword argument:

```
"""First letter: {first}. Last letter: {last}""".format(last='Z', first='A')
'First letter: A. Last letter: Z.'
```

Finally, for numerical inputs, you can include format codes which control how the value is converted to a string. For example, to print a number as a floating point with three digits after the decimal point, you can use the following:

```
"pi = {:.3f}".format(pi)
pi = 3.142'
```

Here the "`0`", as above, refers to the index of the value to be inserted. The ":" marks that format codes will follow. The ".3f" encodes the desired precision: three digits beyond the decimal point, floating-point format.

This style of format specification is very flexible, and the examples here barely scratch the surface of the formatting options available. For more information on the syntax of these format strings, see the [Format Specification](#) section of Python's online documentation.

Flexible Pattern Matching with Regular Expressions

The methods of Python's `str` type give you a powerful set of tools for formatting, splitting, and manipulating string data. But even more powerful tools are available in Python's built-in *regular expression* module. Regular expressions are a huge topic; there are [entire books](#) written on the topic, so it will be hard to do justice within just a single subsection.

My goal here is to give you an idea of the types of problems which might be addressed using regular expressions, as well as a basic idea of how to use them in Python. Below I'll suggest some references for learning more.

Fundamentally, regular expressions are a means of *flexible pattern matching* in strings. If you frequently use the command-line, you are probably familiar with this type of flexible matching with the "*" character, which acts as a wildcard. For example, we can list all the IPython notebooks (i.e. files with extension `.ipynb`) with "Python" in their filename by using the "*" wildcard to match any characters in between:

```
!ls *Python*.ipynb
01.00-Whirlwind-Tour-of-Python.ipynb 01.02-Basic-Python-Syntax.ipynb
01.01-How-to-Run-Python-Code.ipynb
```

Regular expressions generalize this "wildcard" idea to a wide range of flexible string-matching syntaxes. The Python interface to regular expressions is contained in the built-in `re` module; as a simple example, let's use it to duplicate the functionality of the string `split()` method:

```
import re
regex = re.compile('\s+')
regex.split(line)

['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'a', 'lazy', 'dog']
```

Here we've first *compiled* a regular expression, then used it to *split* a string. Just as Python's `split()` method returns a list of all substrings between whitespace, the regular expression `split()` method returns a list of all substrings between matches to the input pattern.

In this case, the input is "`\s+`": "`\s`" is a special character which matches any whitespace (space, tab, newline, etc.), and the "+" is a character which indicates *one or more* of the entity preceding it. Thus, the regular expression matches any substring consisting of one or more spaces.

The `split()` method here is basically a convenience routine built upon this *pattern matching* behavior; more fundamental is the `match()` method, which will tell you whether the beginning of a string matches the pattern:

```
for s in ["    ", "abc ", " abc"]:
    if regex.match(s):
```

```
    print(repr(s), "matches")
else:
    print(repr(s), "does not match")

'      matches
'abc  does not match
' abc' matches
```

Like `split()`, there are similar convenience routines to find the first match (like `str.index()` or `str.find()`) or to find and replace (like `str.replace()`). We'll again use the line from above:

```
line = 'the quick brown fox jumped over a lazy dog'
```

With this, we can see that the `regex.search()` method operates a lot like `str.index()` or `str.find()`:

```
line.index('fox')
16

regex = re.compile('fox')
match = regex.search(line)
match.start()
16
```

Similarly, the `regex.sub()` method operates much like `str.replace()`:

```
line.replace('fox', 'BEAR')
'the quick brown BEAR jumped over a lazy dog'
regex.sub('BEAR', line)
'the quick brown BEAR jumped over a lazy dog'
```

With a bit of thought, other native string operations can also be cast as regular expressions.

A More Sophisticated Example

But, you might ask, why would you want to use the more complicated and verbose syntax of regular expressions rather than the more intuitive and simple string methods? The advantage is that regular expressions offer *far* more flexibility.

Here we'll consider a more complicated example: the common task of matching email addresses. I'll start by simply writing a (somewhat indecipherable) regular expression, and then walk through what is going on. Here it goes:

```
email = re.compile('\w+@\w+\.[a-z]{3}')
```

Using this, if we're given a line from a document, we can quickly extract all email addresses

```
text = "To email Guido, try guido@python.org or the older address guido@google.com."
email.findall(text)

['guido@python.org', 'guido@google.com']
```

(Note that these addresses are entirely made up; there are probably better ways to get in touch with Guido).

We can do further operations, like replacing these email addresses with another string, perhaps to hide addresses in the output:

```
email.sub('--@----', text)

'To email Guido, try --@---- or the older address --@----.'
```

Finally, note that if you really want to match *any* email address, the above regular expression is far too simple. For example, it only allows addresses made of alphanumeric characters which end in one of several common domain suffixes. So, for example, the period used here means that we only find part of the address:

```
email.findall('barack.obama@whitehouse.gov')

['obama@whitehouse.gov']
```

This goes to show how unforgiving regular expressions can be if you're not careful! If you search around online, you can find some suggestions for regular expressions which will match *all* valid emails, but beware: they are much more involved than the simple expression used above!

Basics of Regular Expression Syntax

The Syntax of regular expressions is much too large a topic for this short section. Still, a bit of familiarity can go a long way: I will walk through some of the basic constructs here, and then list some more complete resources from which you can learn more. My hope is that the following quick primer will enable you to use these resources effectively.

1. Simple strings are matched directly. This almost goes without saying. If you build a regular expression on a simple string of characters or digits, it will match that exact string

```
regex = re.compile('ion')
regex.findall('Great Expectations')

['ion']
```

2. Some characters have special meanings. While simple letters or numbers are direct matches, there are a handful of characters which have special meanings within regular expressions. They are:

```
. ^ $ * + ? { } [ ] \ | ( )
```

We will discuss the meaning of some of these below. In the meantime, you should know that if you'd like to match any of these characters directly, you can *escape* them with a back-slash:

```
regex = re.compile(r'\$')
regex.findall("the cost is $20")
['$']
```

The `r` preface in `r'\$'` indicates a *raw string*; in standard Python strings, the back-slash is used to indicate special characters. For example, a tab is indicated by "`\t`":

```
print('a\tb\tc')
a    b    c
```

Such substitutions are not made in a raw string:

```
print(r'a\tb\tc')
a\tb\tc
```

For this reason, whenever you use backslashes in a regular expression, it is good practice to use a raw string.

3. Special characters can match character groups. Just as the "`\`" character within regular expressions can escape special characters, turning them into normal characters, it can also be used to give normal characters special meaning. These special characters match specified groups of characters, and we've seen them before. In the email address regexp above, we used the character "`\w`", which is a special marker matching *any alphanumeric character*. Similarly, in the simple `split()` example we also saw "`\s`", a special marker indicating *any whitespace character*.

Putting these together, we can create a regular expression that will match *any two letters/digits with whitespace between them*:

```
regex = re.compile(r'\w\s\w')
regex.findall('the fox is 9 years old')
['e f', 'x i', 's 9', 's o']
```

This example begins to hint at the power and flexibility of regular expressions.

The following table lists a few of these characters that are commonly useful:

Character	Description	Character	Description
<code>\d</code>	match any digit	<code>\D</code>	match any non-digit
<code>\s</code>	match any whitespace	<code>\S</code>	match any non-whitespace
<code>\w</code>	match any alphanumeric char	<code>\W</code>	match any non-alphanumeric char

This is **not** a comprehensive list or description; for more details see Python's [regular expression syntax](#) documentation.

4. Square brackets match custom character groups. If the built-in character groups above aren't specific enough for you, you can use square brackets to specify any set of characters you're interested in. For example, the following will match any lower-case vowel:

```
regex = re.compile('[aeiou]')
regex.split('consequential')
['c', 'ns', 'q', '', 'nt', '', 'l']
```

Similarly, you can use a dash to specify a range: for example "[a-z]" will match any lower-case letter, and "[1-3]" will match any of "1", "2", or "3". For example, you may need to extract specific numerical codes in a document which consist of a capital letter followed by a digit. You could do this as follows:

```
regex = re.compile('[A-Z][0-9]')
regex.findall('1043879, G2, H6')
['G2', 'H6']
```

5. Wildcards match repeated characters. If you would like to match a string with, say, three alphanumeric characters in a row, it is possible to write, e.g. "\w\w\w". Because this is such a common need, there is a specific syntax to match repetitions: curly braces with a number:

```
regex = re.compile(r'\w{3}')
regex.findall('The quick brown fox')
['The', 'qui', 'bro', 'fox']
```

There are also markers available to match any number of repetitions: for example, the "+" character will match *one or more* repetitions of what precedes it:

```
regex = re.compile(r'\w+')
regex.findall('The quick brown fox')
['The', 'quick', 'brown', 'fox']
```

The following is a table of the repetition markers available for use in regular expressions:

Character	Description	Example
?	match zero or one repetitions of preceding	"ab?" matches "a" or "ab"
*	match zero or more repetitions of preceding	"ab*" matches "a", "ab", "abb", "abbb"...
+	match one or more repetitions of preceding	"ab+" matches "ab", "abb", "abbb"... but not "a"
{n}	match n repetitions of preceding	"ab{2}" matches "abb"

Character	Description	Example
{m,n}	match between m and n repetitions of preceding	"ab{2,3}" matches "abb" or "abbb"

With the above basics in mind, let's return to our email address matcher:

```
email = re.compile(r'\w+@\w+\.[a-z]{3}')
```

We can now understand what this means: we want one or more alphanumeric character ("`\w+`") followed by the *at sign* ("`@`"), followed by one or more alphanumeric character ("`\w+`"), followed by a period ("`\.`" – note the need for a backslash escape), followed by exactly three lower-case letters.

If we want to now modify this so that the obama email address matches, we can do so using the square-bracket notation:

```
email2 = re.compile(r'[\w.]+@\w+\.[a-z]{3}')
email2.findall('barack.obama@whitehouse.gov')
['barack.obama@whitehouse.gov']
```

We have changed "`\w+`" to "`[\w.]+`", so we will match any alphanumeric character *or* a period. With this more flexible expression, we can match a wider range of email addresses (though still not all – can you identify other shortcomings of this expression?)

6. Parentheses indicate *group*s to extract. For compound regular expressions like our email matcher, we often want to extract their components rather than the full match. This can be done using "`()`" to *group* the results:

```
email3 = re.compile(r'([\w.]+)@(\w+)\.([a-z]{3})')
text = "To email Guido, try guido@python.org or the older address guido@google.com."
email3.findall(text)
[('guido', 'python', 'org'), ('guido', 'google', 'com')]
```

As we see, this grouping actually extracts a list of the sub-components of the email address.

We can go a bit further and *name* the extracted components using the "`(?P<name>)`" syntax, in which case the groups can be extracted as a Python dictionary:

```
email4 = re.compile(r'(?P<user>[\w.]+)@(?P<domain>\w+)\.(?P<suffix>[a-z]{3})')
match = email4.match('guido@python.org')
match.groupdict()
{'domain': 'python', 'suffix': 'org', 'user': 'guido'}
```

Combining these ideas (as well as some of the powerful regexp syntax that we have not covered here) allows you to flexibly and quickly extract information from strings in Python.

Regular Expressions: Further Resources

The above discussion is just a quick – and incomplete – treatment of this large topic. If you'd like to learn more, I recommend the following resources:

- **Python's re package Documentation:** I, for one, promptly forget how to use regular expressions just about every time I use them. Once I got the basics down, I have found this page to be an incredibly valuable resource to recall what each specific character or sequence means within a regular expression.
- **Python's official regular expression HOWTO:** a more narrative approach to regular expressions in Python.
- **Mastering Regular Expressions (O'Reilly, 2006)** is a 500+ page book on this subject. If you want a really complete treatment of this topic, this is the resource for you.

For some examples of string manipulation and regular expressions in action at a larger scale, see Section X.X, where we look at applying these sorts of expressions across *tables* of string data within the *Pandas* package.

Further Python Resources

This concludes our whirlwind tour of the Python language. My hope is that if you read this far, you have an idea of the essential syntax and semantics of the Python language, and some idea of the range of tools and code constructs that you can explore further.

I have tried to cover the pieces and patterns in the Python language which will be most useful to a data scientist using Python, but this has by no means been a complete introduction. If you'd like to go deeper in understanding the Python language itself and how to use it effectively, here are a couple resources I'd recommend:

- **Dive Into Python** <http://www.diveintopython.net/> This is a free online book that provides a ground-up introduction to the Python language.
- **Learn Python the Hard Way** by Zed Shaw. This book follows a “learn by trying” approach, and deliberately emphasizes developing what may be the most useful skill a programmer can learn: Googling things you don't understand.
- **Python Essential Reference** by Dave Beazley. This 700 page monster is well-written, and covers virtually everything there is to know about the Python language and its built-in libraries. For a more application-focused Python walk-through, see Beazley's **Python Cookbook**
- **O'Reilly Python Resources:** <http://shop.oreilly.com/category/browse-subjects/programming/python.do> O'Reilly features a number of excellent books on Python itself and specialized topics in the Python world.

- **PyCon Talks and Tutorials:** The PyCon conference series draws thousands of attendees each year, and has turned into an incredible resource for learning about the language. Talk selection has become very competitive, and tutorials in particular are generally very well-done. Videos from past years are available online: search the web for “pycon tutorial videos”, or visit <http://pyvideo.org/> and start watching.

More Advanced Python Language Features

These are topics that we'll not cover in this book. They're only rarely useful in data science applications, but they may be interesting to those who want to dive deeper into the abilities of the Python language. We'll list them here mainly so you can know the correct terminology to use as you search for online resources to learn more.

- creating packages and modules (`import`-ing your own code!)
- defining and using classes (creating your own types!)
- decorators (what's with the `@` sign anyway?)
- context managers (the `with` statement)
- coroutines (The `yield` statement as a two-way street!)
- metaclasses (classes which define classes!)
- properties and descriptors (attributes which hide computation!)

In my opinion, some of the best material on the more advanced Python topics comes from Dave Beazley, within his books, blog posts, and video-recorded talks. Search his name along with the topic you're interested in, and you're likely to find a detailed and entertaining discussion.

More Built-in Modules

In section X.X we covered some of the built-in “batteries included” modules that come along with Python. You can see a full list in the online Python documentation, found at <https://docs.python.org/3/library/>. As you explore them, remember to use IPython's tab completion and help features to learn what's available.

More Third-Party Modules

The bread-and-butter of data science with Python are the third-party modules that people develop for specific application areas. The next several chapters of this book will discuss some of the most important of these in some detail, but there are more tools out there than we can possibly hope to discuss, and more being developed every day. Interested in astronomy? check out <http://astropy.org/>. Neuro Imaging? <http://nipy.org/>. Biological science? <http://biopython.org/>. Looking for some of the statistical models found in R? Try <http://statsmodels.sourceforge.net/>. Network analysis? <http://networkx.github.io/> is a great place to start.

And this just scrapes the surface many, many tools that Python users are developing and making public. Browse <http://pypi.python.org> for a nearly complete (but mostly unreviewed) list of packages people are releasing, or search the web for mailing list threads, blog posts, tweets, and other discussions which might point you to tools and resources for the application you're interested in. Chances are, someone has been thinking about the same things and has already put their code out there. If not, then who better than you to start it?

IPython: Beyond Normal Python

There are many options for IDEs (Interactive Development Environments) for Python, and I'm often asked which one I prefer in my own work. My answer sometimes surprises people: my preferred IDE is IPython plus a text editor (in my case, emacs). IPython, short for Interactive Python, was started in 2001 by Fernando Perez as an enhanced Python interpreter, and has since grown into a project aiming to provide, in Perez's words, "Tools for the entire life-cycle of Research computing." If Python is the engine of our data science task, you might think of IPython as the interactive control panel.

As well as being a useful interactive interface to Python, IPython also provides a number of useful syntactic additions to the language; we'll cover the most useful of these additions here. In addition, IPython is closely tied with the Jupyter project, which provides a browser-based notebook which is useful for development, collaboration, sharing, and even publication of data science results. The IPython notebook is actually a special case of the broader Jupyter notebook structure, which encompasses notebooks for Julia, R, and other programming languages. As an example of the usefulness of the notebook format, look no further than the page you are reading: this entire text was composed as a set of IPython notebooks.

IPython is about using Python effectively for interactive scientific and data-intensive computing. This chapter will start by stepping through some of the features of IPython which are useful to the practice of data science, focusing especially on the syntax it offers beyond the standard features of Python. Next, we will go into a bit more depth on some of the more useful "magic commands" that can speed-up common tasks in creating and using data science code. Finally, we will touch on some of the features of the notebook that make it useful in understanding data and sharing results.

Shell or Notebook?

There are two primary means of using IPython that we'll discuss in this chapter: the IPython shell and the IPython notebook. The bulk of the material in this chapter is relevant to both, and the examples will switch between them depending on what is most convenient. In the few sections which are relevant to just one or the other, we will explicitly state that fact. Before we start, some words on how to launch the IPython shell and IPython notebook.

Launching the IPython Shell

This chapter, like most of this book, is not designed to be absorbed passively. I recommend that as you read through it, you follow along and experiment with the tools and syntax we cover: the muscle-memory you build through doing this will be far more useful than the simple act of reading about it. Start by launching the IPython interpreter by typing `ipython` on the command-line; alternatively, if you've installed a distribution like Anaconda or EPD, there may be a launcher specific to your system (for more information, refer back to the brief discussion in Section 1.1).

Once you do this, you should see a prompt like the following:

```
IPython 2.2.0 -- An enhanced Interactive Python.  
?          -> Introduction and overview of IPython's features.  
%quickref -> Quick reference.  
help       -> Python's own help system.  
object?    -> Details about 'object', use 'object??' for extra details.  
In [1]:
```

With that, you're ready to follow along.

Launching the IPython Notebook

The IPython notebook is a browser-based graphical interface to the IPython shell, and builds 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 them and execute the code on their own systems.

Though the IPython notebook is viewed and edited through your web browser window, it must connect to a running Python process in order to execute code. This process (known as a “kernel”) can be started by running the following command in your system shell:

```
$ ipython notebook
```

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

```
$ ipython notebook  
[NotebookApp] Using existing profile dir: '/home/jake/.ipython/profile_default'  
[NotebookApp] Serving notebooks from local directory: /home/jake/notebooks/  
[NotebookApp] The IPython Notebook is running at: http://localhost:8888/  
[NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation)
```

At 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 copy this address (here `http://localhost:8888/`) manually.

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 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 the answer as much as knowing how to quickly find an unknown answer. In data science it's the same: searchable web resources like online documentation, mailing-list threads, and StackOverflow answers contain a wealth of information, even (especially?) if its a topic you've found yourself searching before. Being an effective practitioner of data science is less about memorizing the tool or command you should use for every possible situation, and more about learning to effectively find the information you don't know, whether through a web search engine or another means.

One of the most useful functions of IPython is to shorten the gap between the Python user 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 object have?

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 the `<TAB>` character for auto-completion.

Accessing Documentation with "?"

The Python language and its data science ecosystem is 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 *doc string*, which in most cases will contain a concise summary of the object and how to use it. Python has a built-in `help()` function which accesses this information and prints the results. For example, to see the documentation of the built-in `len` function, 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_method
String form: <built-in function len>
Namespace:  Python builtin
Docstring:
len(object) -> integer

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

This notation works for just about anything, including object methods:

```
In [3]: L = [1, 2, 3]
In [4]: L.insert?
Type:       builtin_function_or_method
String form: <built-in method insert of list object at 0x1024b8ea8>
Docstring:  L.insert(index, object) -- insert object before index
```

or even objects themselves, with the documentation from their type:

```
In [5]: L?
Type:       list
String form: [1, 2, 3]
Length:    3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Importantly, this will even work for functions or other objects you create yourself! Here we'll define a small function with a docstring:

```
In [6]: def square(a):
....:     """Return the square of a."""
....:     return a ** 2
....:
```

Note that to create a docstring for our function, we simply placed a string literal in the first line. Since doc strings are usually multiple lines, by convention we used Python's triple-quote notation for multi-line strings.

Now we'll use the ? mark to find this doc string:

```
In [7]: square?
Type:      function
String form: <function square at 0x103713cb0>
Definition: square(a)
Docstring:  Return the square of a.
```

this quick access to documentation via docstrings is one reason you should get in the habit of always adding such inline documentation to the code you write!

Accessing Source Code with "??"

Because the Python language is so easily readable, another level of insight can usually be gained by reading the source code of the object you're curious about. IPython provides 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 into the details under-the-hood.

If you play with this much, you'll notice that sometimes the ?? suffix doesn't display any source code: this is generally because the object in question is not implemented in Python, 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 len from above:

```
In [9]: len??
Type:      builtin_function_or_method
String 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 auto-completion and exploration of the contents of objects, modules, and name-spaces. In all the examples below, we'll use the characters `<TAB>` to indicate pressing the tab key.

Tab-completion of Object Contents

Every Python object has various attributes and methods associated with it. Like with the `help` function above, Python has a built-in `dir` function which returns a list of these, but the tab-completion interface is much easier to use in practice. To see a list of all available attributes of an object, you can type the name of the object followed by a `.` character and the tab key:

```
In [10]: L.<TAB>  
L.append  L.copy    L.extend  L.insert  L.remove  L.sort  
L.clear   L.count   L.index   L.pop     L.reverse
```

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, the tab character will complete the line for you. For example,

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

will instantly be replaced with `L.count`.

Though Python has no strictly-enforced distinction between public/external attributes and private/internal attributes, by convention a preceding underscore is used to denote such methods. For clarity, these private methods and special methods are left out of the list by default, but it's possible to list them by explicitly typing the underscore:

```
In [10]: L._<TAB>  
L.__add__        L.__gt__       L.__reduce__  
L.__class__      L.__hash__     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); you can read more about these in the references mentioned in section X.X.

Tab Completion when Importing

Tab completion is also useful when importing objects from packages. Here we'll use it to find all possible imports in the `itertools` package which start with the letter `c`:

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

Similarly, you can use tab-completion to see what imports are available on your system (this will change depending on what 3rd-party scripts and modules are visible to your Python session):

```
In [10]: import <TAB>
Display all 399 possibilities? (y or n)
Crypto          dis          py_compile
Cython          distutils    pyclbr
...
difflib         pwd          zmq
In [10]: import h<tab>
hashlib         hmac         http
heapq          html         husl
```

(Note that for brevity, we did not print 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 attribute you'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 matching for names using the `*` character.

For example, we can use this to list every object in the namespace which ends with `Warning`:

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

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

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

```
In [10]: str.*find*?  
str.find  
str.rfind
```

I find this type of flexible wildcard search can be very useful to search for a particular command when getting to know a new tool or reacquainting myself with a familiar one.

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 the `Ctrl-C` and `Ctrl-V` for copying and pasting in a wide variety of programs and systems. Power-users tend to go even further: popular text editors like emacs, vim, and others provide users an incredible range of operations through intricate combinations of keystrokes.

The IPython shell doesn't go this far, but does provide a number of keyboard shortcuts for fast navigation while typing commands. These shortcuts are not in fact provided by IPython itself, but through its dependency on the GNU readline package: as such, some of the following shortcuts may differ depending on your system configuration. 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 used 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 following will be very familiar. We'll divide these into a few sections: *Navigation Shortcuts*, *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 which don't require moving your hands from the "home" keyboard position:

Keystroke	Action
<code>Ctrl-a</code>	Move cursor to the beginning of the line
<code>Ctrl-e</code>	Move cursor to the end of the line
<code>Ctrl-b</code> or <code>left-arrow</code>	Move cursor back one character
<code>Ctrl-f</code> or <code>right-arrow</code>	Move cursor forward one character

Text Entry Shortcuts

While everyone is familiar with `backspace` to delete the previous character, reaching 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 removing 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 typing `Ctrl-b` `Ctrl-d` instead of `backspace` to delete the previous character!

Keystroke	Action
<code>backspace</code>	Delete previous character in line
<code>Ctrl-d</code>	Delete next character in line
<code>Ctrl-k</code>	Cut text from cursor to end of line
<code>Ctrl-u</code>	Cut all text in line
<code>Ctrl-y</code>	Yank (i.e. Paste) text which was previously cut
<code>Ctrl-t</code>	Transpose (i.e. switch) previous two characters

Command History Shortcuts

Perhaps most impactful shortcuts discussed here are IPython's shortcuts to navigate the command history. This command history goes beyond your current 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 arrows to step through the history, but other options exist as well:

Keystroke	Action
<code>Ctrl-p</code> or up-arrow	Access previous command in history
<code>Ctrl-n</code> or down-arrow	Access next command in history
<code>Ctrl-r</code>	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 new IPython 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` again to search further for another command which 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 the `Ctrl-p/Ctrl-n` or up/down arrows can also be used to search through history, but only by matching characters at the beginning of the line. That is, if you type `def<Ctrl-p>` it would find the most recent command (if any) in your history which begins with the characters `def`.

Miscellaneous Shortcuts

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

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

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

Though some of the shortcuts discussed above 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 missing them in other contexts. If you come to enjoy this style of shortcut, you may want to try out the Emacs text editor, which contains many of these shortcuts, along with a near endless variety of others in the same style.

IPython Magic Commands

The previous two sections showed how IPython lets you use and explore Python efficiently 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*, denoted by a single % prefix and which operate on a single line of input, and *cell magics*, denoted by a double %% prefix and which operate on multiple lines of input. We'll demonstrate and discuss a few brief examples here, and come back to more focused discussion of several useful magic commands later in the chapter.

Pasting Code Blocks: %paste and %cpaste

When working in the IPython interpreter, one common gotcha is that pasting multi-line code blocks can lead to unexpected errors, especially when indentation and interpreter markers are involved. A common case is that you find some example code on 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-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
```

In the direct paste, the interpreter is confused by the additional prompt characters. But never fear: IPython's `%paste` magic function is designed to handle this exact type of multi-line, 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 is ready 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'll see, make available functionality that would be 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 in both IPython for interactive exploration, as well as a text editor to store code that you want to re-use. Rather than running this code in a new window, it can be convenient to run it within your IPython session. This can be done with the `%run` magic.

For example, imagine you've created the following file in `myscript.py`

```
#-----
# 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 available for 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 documentation 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 which 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 multiple runs in order to attain more robust results. For multi-line statements, adding a second `%` sign will turn this into a cell magic which can handle multiple lines of input. For example, here's the equivalent construction with a `for`-loop:

```
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 other approaches to timing and profiling code in section X.X.

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

One useful thing to know is that the IPython magic functions have docstrings like normal Python functions, and this documentation can be accessed in the standard manner. So, for example, to read the documentation of the `%timeit` magic simply type

```
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

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

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

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

Finally, I'll mention that it is quite straightforward to define your own magic functions if you wish. We won't discuss it here, but if you are interested, see the references listed at the end of this chapter.

Input and Output History

Previously we saw that the IPython shell allows you to access previous commands with the up-arrow/down-arrow keys, or equivalently the `Ctrl-p/Ctrl-n` shortcuts. Additionally, in both the shell and the notebook, IPython exposes several ways to obtain the output of previous commands, as well as string versions of the commands themselves. We'll explore those here.

IPython's In and Out Objects

By now I imagine you're quite used to the `In [1]:/Out[1]:` style prompts used by IPython. But it turns out that these are not just pretty decoration: they give a clue as to how you can access previous inputs and outputs in your current session. Imagine you start a session which looks like this:

```
In [1]: import math  
  
In [2]: math.sin(2)  
Out[2]: 0.9092974268256817  
  
In [3]: math.cos(2)  
Out[3]: -0.4161468365471424
```

We've imported the built-in `math` package, then computed the sine and the cosine of the number 2. These inputs and outputs are displayed in the shell with `In/Out` labels, but there's more: IPython actually creates some Python variables called `In` and `Out` which are automatically updated to reflect this history:

```
In [4]: print(In)  
[], 'import math', 'math.sin(2)', 'math.cos(2)', 'print(In)'  
  
In [5]: Out  
Out[5]: {2: 0.9092974268256817, 3: -0.4161468365471424}
```

The `In` object is a list, which keeps track of the commands in order (the first item in the list is a place-holder so that `In[1]` can refer to the first command):

```
In [6]: print(In[1])  
import math
```

The `Out` object is not a list but a dictionary mapping input numbers to their outputs (if any):

```
In [7]: print(Out[2])  
0.9092974268256817
```

Note that not all operations have outputs: for example, `import` statements and `print` statements don't affect the `Output`. The latter may be surprising, but makes sense if

you consider that `print` is a function which returns `None`; for brevity, any command which returns `None` is not added to `Out`.

Where this can be useful is if you want to interact with past results. For example, let's check the sum of `sin(2) ** 2` and `cos(2) ** 2` using the previously-computed results:

```
In [8]: Out[2] ** 2 + Out[3] ** 2
Out[8]: 1.0
```

The result is `1.0` as we'd expect from the well-known trigonometric identity. In this case, using these previous results probably is not necessary, but it can become very handy if you execute a very expensive computation and want to reuse the result!

Underscore Shortcuts and Previous Outputs

The standard Python shell contains just one simple shortcut for accessing previous output; the variable `_` (i.e. a single underscore) is kept updated with the previous output; this works in IPython as well:

```
In [9]: print(_)
1.0
```

But IPython takes this a bit further: you can use a double underscore to access the second-to-last output, and a triple underscore to access the third-to-last output (skipping any commands with no output):

```
In [10]: print(__)
-0.4161468365471424
```

```
In [11]: print(__)
0.9092974268256817
```

IPython stops there: more than three underscores starts to get a bit hard to count, and at that point it's easier to refer to the output by line number as above.

There is one more set of shortcuts we should mention, however: a shorthand for `Out[X]` is `_X`, a single underscore followed by the line number:

```
In [12]: Out[2]
Out[12]: 0.9092974268256817
```

```
In [13]: _2
Out[13]: 0.9092974268256817
```

Suppressing Output

Sometimes you might wish to suppress the output of a statement; this is perhaps most common with the plotting commands that we'll explore in chapter X.X. Or perhaps the command you're executing produces a result you'd not like to store in your output

history, perhaps so that it can be deallocated when other references are removed. The easiest way to suppress the output of a command is to add a semicolon to the end of the line:

```
In [14]: math.sin(2) + math.cos(2);
```

Note that the result is computed silently, and the output is neither displayed on the screen or stored in the Out dictionary:

```
In [15]: 14 in Out  
Out[15]: False
```

Related Magic Commands

For accessing a batch of previous inputs at once, the %history magic command is very helpful. Here is how you can print the first four inputs:

```
In [16]: %history -n 1-4  
1: import math  
2: math.sin(2)  
3: math.cos(2)  
4: print(In)
```

As usual, you can type %history? for more information and a description of options available. Other similar magic commands are %rerun, which will re-execute some portion of the command history and %save, which saves some set of the command history to a file. For more information, I suggest exploring these using the ? help functionality discussed in the previous sections.

IPython and Shell Commands

When working interactively with the standard Python interpreter, one of the frustrations is the need to switch between multiple windows to access Python tools and system command-line tools. IPython bridges this gap, and gives you a syntax for executing shell commands directly from within the IPython terminal. The magic happens with the exclamation point: anything appearing after "!" on a line will be executed not by the Python kernel, but by the system command-line.

The following assumes you're on a Unix-like system, such as Linux or Mac OSX. Some of the examples below will fail on Windows, which uses a different type of shell by default.

Quick Introduction to the Shell

A full intro to using the shell/terminal/command-line is well beyond the scope of this section, but for the uninitiated we will offer a quick introduction here. The shell is a way to interact textually with your computer. Ever since the mid 1980s, when Micro-

soft and Apple introduced the first versions of their now ubiquitous graphical operating systems, most computer users have interacted with their operating system through familiar clicking of menus and drag-and-drop movements. But operating systems existed before these graphical elements did, and were primarily controlled through sequences of text input: at the prompt, the user would type a command, and the computer would do what the user told it to. Those early prompt systems are the precursors of the shells and terminals that most active data scientists still use today.

Someone unfamiliar with the shell might ask why you would bother with this, when many results can be accomplished by simply clicking on icons and menus. A shell user might reply with another question: why hunt icons and click menus when you can accomplish things much more easily by typing? While it might sound like a typical tech preference impasse, when moving beyond basic tasks it quickly becomes clear that the shell offers much more control of advanced tasks, though admittedly the learning curve can intimidate the average computer user.

As an example, here is a sample of a Linux/OSX shell session where a user explores, creates, and modifies directories and files on their system. The prompt text looks like `mac:~ $`; everything after the `$` sign is the typed command. Text after `#` is meant just as description, rather than something you would actually type in:

```
osx:~ $ echo "hello world"          # echo is like Python's print function
hello world

osx:~ $ pwd                         # pwd = print working directory
/home/jake                           # this is the "path" that we're sitting in

osx:~ $ ls                           # ls = list working directory contents
notebooks  projects

osx:~ $ cd projects/                 # cd = change directory

osx:projects $ pwd                   # pwd = print working directory
/home/jake/projects

osx:projects $ ls                    # ls = list working directory contents
datasci_book  mpld3  myproject.txt

osx:projects $ mkdir myproject      # mkdir = make new directory

osx:projects $ cd myproject/
osx:myproject $ mv ../myproject.txt . / # mv = move file. Here we're moving the
                                         # file myproject.txt from one directory up (../)
                                         # to the current directory (./)

osx:myproject $ ls
myproject.txt
```

Notice that all of this is just a compact way to do familiar operations (navigating a directory structure, creating a directory, moving a file, etc.) by typing commands rather than clicking icons and menus. Note that with just a few commands (here `pwd`, `ls`, `cd`, `mkdir`, `cp`) you can do many of the most common file operations. It's when you go beyond these basics that the shell approach becomes really powerful.

Shell Commands in IPython

Any command that works at the command-line can be used in IPython preceded by the the "!" character. Let's continue the above session and start a new IPython interpreter, and execute the `ls` command again from there:

```
osx:myproject jakevdp$ ipython
IPython 2.3.1 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?     -> Details about 'object', use 'object??' for extra details.

In [1]: !ls
myproject.txt
```

It works!

```
In [2]: !pwd
/home/jake/projects/myproject

In [3]: !echo "printing from the shell"
printing from the shell
```

Passing Values To and From the Shell

Not only can shell commands be called from IPython, but they can be made to interact with the IPython namespace. For example, you can save the output of any shell command to a Python list using the assignment operator:

```
In [4]: contents = !ls

In [5]: print(contents)
['myproject.txt']

In [6]: directory = !pwd

In [7]: print(directory)
[/Users/jakevdp/notebooks/tmp/myproject']
```

Note that these results are not returned as lists, but as a special shell return type defined in IPython:

```
In [8]: type(directory)
IPython.utils.text.SList
```

This looks and acts a lot like a Python list, but has additional functionality, (such as the `grep` and `fields` methods, and the `s`, `n`, and `p` properties) allow you to search, filter, and display the results in convenient ways. For more information on these, you can use IPython's built-in help features.

Communication is also possible in the other direction, by using the `{varname}` syntax:

```
In [9]: message = "hello from Python"
```

```
In [10]: !echo {message}
hello from Python
```

The curly braces contain the variable name which is replaced by the variable's contents in the shell command.

Shell-related Magic Commands

If you play with IPython's shell commands for a while, you might notice that you cannot use `!cd` to navigate the filesystem:

```
In [11]: !pwd
/home/jake/projects/myproject
```

```
In [12]: !cd ..
```

```
In [13]: !pwd
/home/jake/projects/myproject
```

The reason is that shell commands in the notebook are executed in a temporary sub-shell. If you'd like to change the working directory in a more enduring way, you can use the `%cd` magic command:

```
In [14]: %cd ..
/home/jake/projects
```

In fact, by default you can even use this without the `%` sign:

```
In [15]: cd myproject
/home/jake/projects/myproject
```

This is known as an `automagic` function, and this behavior can be toggled with the `%automagic` magic function.

Beside `%cd`, other available shell-like magic functions are `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm`, and `%rmdir`, any of which can be used without the `%` sign if `automagic` is on. This makes it so that you can almost treat the IPython prompt as if it's a normal shell:

```
In [16]: mkdir tmp  
In [17]: ls  
myproject.txt  tmp/  
In [18]: cp myproject.txt tmp/  
In [19]: ls tmp  
myproject.txt  
In [20]: rm -r tmp
```

This access to the shell from within the same terminal window as your Python session means that there is a lot less switching back and forth between interpreter and shell as you write your Python code.

Errors and Debugging

Code development and data analysis always requires a bit of trial and error, and IPython contains tools to streamline this process. This section will briefly cover some options for controlling Python's exception reporting, followed by exploring tools for debugging errors in code.

Controlling Exceptions: %xmode

Most of the time when a Python script fails, it will raise an `Exception`, which we covered in section X.X. When the interpreter hits one of these exceptions, information about the cause of the error can be found in the `traceback`, which can be accessed from within Python. With the `%xmode` magic function, IPython allows you to control the amount of information printed when the exception is raised. Consider the following code:

```
def func1(a, b):  
    return a / b  
  
def func2(x):  
    a = x  
    b = x - 1  
    return func1(a, b)  
  
func2(1)  
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-2-b2e110f6fc8f> in <module>()  
      1 func2(1)  
  
<ipython-input-1-d849e34d61fb> in func2(x)
```

```
5     a = x
6     b = x - 1
----> 7     return func1(a, b)

<ipython-input-1-d849e34d61fb> in func1(a, b)
    1 def func1(a, b):
----> 2     return a / b
    3
    4 def func2(x):
    5     a = x

ZeroDivisionError: division by zero
```

Calling `func2` results in an error, and reading the printed trace lets us see exactly what happened. By default, this trace includes several lines showing the context of each step that led to the error. Using the `%xmode` magic function (short for Exception mode), we can change what information is printed.

`%xmode` takes a single argument, the mode, and there are three possibilities: `Plain`, `Context`, and `Verbose`. The default is `Context`, and gives an output like above. `Plain` is more compact and gives less information:

```
%xmode Plain
Exception reporting mode: Plain
func2(1)
Traceback (most recent call last):

File "<ipython-input-4-b2e110f6fc8f>", line 1, in <module>
  func2(1)

File "<ipython-input-1-d849e34d61fb>", line 7, in func2
  return func1(a, b)

File "<ipython-input-1-d849e34d61fb>", line 2, in func1
  return a / b

ZeroDivisionError: division by zero
```

The `Verbose` mode adds some extra information, including the arguments to any functions that are called:

```
%xmode Verbose
Exception reporting mode: Verbose
```

```

func2(1)
-----
ZeroDivisionError                                Traceback (most recent call last)

<ipython-input-6-b2e110f6fc8f> in <module>()
----> 1 func2(1)
        global func2 = <function func2 at 0x103729320>

<ipython-input-1-d849e34d61fb> in func2(x=1)
      5     a = x
      6     b = x - 1
----> 7     return func1(a, b)
        global func1 = <function func1 at 0x1037294d0>
      a = 1
      b = 0

<ipython-input-1-d849e34d61fb> in func1(a=1, b=0)
      1 def func1(a, b):
----> 2     return a / b
        a = 1
        b = 0
      3
      4 def func2(x):
      5     a = x

ZeroDivisionError: division by zero

```

The added information can help narrow-in on why the exception is being raised. So why not use the `Verbose` mode all the time? As code gets complicated, this kind of traceback can get extremely long. Depending on the context, sometimes the brevity of `Default` mode is easier to work with.

Debugging: When Reading Tracebacks is Not Enough

The standard Python tool for interactive debugging is `pdb`, the Python debugger. This debugger lets the user step through the code line-by-line in order to see what might be causing a more difficult error. The ipython-enhanced version of this is `ipdb`, the IPython debugger.

There are many ways to launch and use both these debuggers; we won't cover them fully here. Refer to the online documentation of these two utilities to learn more.

In IPython, perhaps the most convenient interface to debugging is the `%debug` magic command. If you call it after hitting an exception, it will automatically open an interactive debugging prompt at the point of the exception. The `ipdb` prompt lets you

explore the current state of the stack, explore the available variables, and even run Python commands!

Let's look at the most recent exception, then do some basic tasks: print the values of `a` and `b`, and type `quit` to quit the debugging session:

```
%debug
> \u001b[0;32m<ipython-input-1-d849e34d61fb>\u001b[0m(2)\u001b[0;36mfunc1\u001b[0;34m()\u001b[0m
 \u001b[0;32m      1 \u001b[0;31m\u001b[0;32mdef\u001b[0m \u001b[0mfunc1\u001b[0m\u001b[0;34m(\u001b[0m
 \u001b[0m\u001b[0;32m----> 2 \u001b[0;31m      \u001b[0;32mreturn\u001b[0m \u001b[0ma\u001b[0m \u001b[0m
 \u001b[0m\u001b[0;32m      3 \u001b[0;31m\u001b[0;34m\u001b[0m\u001b[0m\u001b[0m
 \u001b[0m
ipdb> print(a)
1
ipdb> print(b)
0
ipdb> quit
```

The interactive debugger allows much more than this, though: we can even step up and down through the stack and explore the values of variables there:

```
%debug
> \u001b[0;32m<ipython-input-1-d849e34d61fb>\u001b[0m(2)\u001b[0;36mfunc1\u001b[0;34m()\u001b[0m
 \u001b[0;32m      1 \u001b[0;31m\u001b[0;32mdef\u001b[0m \u001b[0mfunc1\u001b[0m\u001b[0;34m(\u001b[0m
 \u001b[0m\u001b[0;32m----> 2 \u001b[0;31m      \u001b[0;32mreturn\u001b[0m \u001b[0ma\u001b[0m \u001b[0m
 \u001b[0m\u001b[0;32m      3 \u001b[0;31m\u001b[0;34m\u001b[0m\u001b[0m\u001b[0m
 \u001b[0m
ipdb> up
> \u001b[0;32m<ipython-input-1-d849e34d61fb>\u001b[0m(7)\u001b[0;36mfunc2\u001b[0;34m()\u001b[0m
 \u001b[0;32m      5 \u001b[0;31m      \u001b[0ma\u001b[0m \u001b[0;34m=\u001b[0m \u001b[0mx\u001b[0m
 \u001b[0m\u001b[0;32m      6 \u001b[0;31m      \u001b[0mb\u001b[0m \u001b[0m \u001b[0;34m=\u001b[0m \u001b[0mb\u001b[0m
 \u001b[0m\u001b[0;32m----> 7 \u001b[0;31m      \u001b[0;32mreturn\u001b[0m \u001b[0ma\u001b[0mfunc1\u001b[0m\u001b[0m
 \u001b[0m
ipdb> print(x)
1
ipdb> up
> \u001b[0;32m<ipython-input-6-b2e110f6fc8f>\u001b[0m(1)\u001b[0;36m<module>\u001b[0;34m()\u001b[0m
 \u001b[0;32m----> 1 \u001b[0;31m\u001b[0mfunc2\u001b[0m \u001b[0m \u001b[0;34m(\u001b[0m \u001b[0m \u001b[0;36m1\u001b[0m
 \u001b[0m
ipdb> down
> \u001b[0;32m<ipython-input-1-d849e34d61fb>\u001b[0m(7)\u001b[0;36mfunc2\u001b[0;34m()\u001b[0m
 \u001b[0;32m      5 \u001b[0;31m      \u001b[0ma\u001b[0m \u001b[0m \u001b[0;34m=\u001b[0m \u001b[0mx\u001b[0m
 \u001b[0m\u001b[0;32m      6 \u001b[0;31m      \u001b[0mb\u001b[0m \u001b[0m \u001b[0;34m=\u001b[0m \u001b[0mb\u001b[0m
 \u001b[0m\u001b[0;32m----> 7 \u001b[0;31m      \u001b[0;32mreturn\u001b[0m \u001b[0ma\u001b[0mfunc1\u001b[0m\u001b[0m
 \u001b[0m
```

This allows you to quickly find out not only what caused the error, but what function calls led up to the error.

If you'd like the debugger to launch automatically whenever an exception is raised, you can use the `%pdb` magic function to turn on this automatic behavior:

```
%xmode Plain
%pdb on
func2(1)

Exception reporting mode: Plain
Automatic pdb calling has been turned ON

Traceback (most recent call last):

File "<ipython-input-9-569a67d2d312>", line 3, in <module>
    func2(1)

File "<ipython-input-1-d849e34d61fb>", line 7, in func2
    return func1(a, b)

File "<ipython-input-1-d849e34d61fb>", line 2, in func1
    return a / b

ZeroDivisionError: division by zero

> \u001b[0;32m<ipython-input-1-d849e34d61fb>\u001b[0m(2)\u001b[0;36mfunc1\u001b[0;34m()\u001b[0m
\u001b[0;32m      1 \u001b[0;31m\u001b[0;32mdef\u001b[0m \u001b[0mfunc1\u001b[0m\u001b[0;34m(\u001b[0m
\u001b[0m\u001b[0;32m----> 2 \u001b[0;31m      \u001b[0;32mreturn\u001b[0m \u001b[0ma\u001b[0m \u001b[0m
\u001b[0m\u001b[0;32m      3 \u001b[0;31m\u001b[0;34m\u001b[0m\u001b[0m\u001b[0m
\u001b[0m
ipdb> print(b)
0
ipdb> quit
```

Finally, if you have a script that you'd like to run from the beginning in interactive mode, you can run it with the command `%run -d`, and use the `next` command to step through the lines of code interactively.

Partial List of Debugging Commands

There are many more available commands for interactive debugging than we've listed here; the following table contains a description of some of the more common and useful ones:

Command	Description
<code>list</code>	show the current location in the file
<code>h(elp)</code>	show a list of commands, or find help on specific command

Command	Description
q(uit)	quit the debugger and the program
c(ontinue)	quit the debugger, continue in the program
n(ext)	go to the next step of the program
<enter>	repeat the previous command
p(rint)	print variables
s(tep)	step into a subroutine
return	return out of a subroutine

For more information, use the `help` command in the debugger, or take a look at the online documentation for `pdb` and `ipdb`.

Profiling and Timing Code

In the process of developing code and creating data processing pipelines, there are often tradeoffs 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 commands; other times it’s useful to dig into a multiple-line process and determine where the bottleneck 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 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 latter commands are not built-in to IPython, but require the `line_profiler` and `memory_profiler` extensions, which we will discuss below.

Timing Code Snippets: `%timeit` and `%time`

We saw the `%timeit` line-magic and `%%timeit` cell-magic in the introduction to magic functions in section X.X; it can be used to time the repeated execution of snippets of code:

```
%timeit sum(range(100))
100000 loops, best of 3: 1.54 µs per loop
```

Note that because this operation is so fast, `%timeit` automatically does a large number of repetitions. For slower commands, `%timeit` will automatically adjust and perform fewer repetitions:

```
%%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:

```
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 for longer-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:

```
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

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 pre-sorted list is to sort, but notice also how much longer the timing takes with `%time` vs `%timeit` above, even for the pre-sorted list! This is a result of the fact that `%timeit` does some clever things under the hood to prevent system calls from interfering with the timing. For example, it prevents cleanup of unused Python objects (known as *garbage collection*) happens during the timing. For this reason, `%timeit` results are usually noticeably faster than `%time` results.

For `%time` as well as `%timeit`, using the double-percent-sign cell magic syntax allows timing of multiple-line scripts:

```
%%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`, `%timeit`, and 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 statements in context is more important than timing them on their own. Python contains a built-in code profiler which you can read about in the Python documentation; IPython has a magic function which is a much more convenient way to use this profiler.

By way of example, we'll define a couple simple functions which do some calculations:

```
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 `%prun` with a function call to see the profiled results:

```
%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 list, in order of total time on each function call, which indicates where the execution is spending the most time. In this case, the bulk of execution time is in

the list comprehension inside `make_list`. From here, we could start thinking about what changes we might make to improve the performance in the algorithm.

For more information on `%prun` and its available options, use the IPython help functionality, i.e. type `%prun?` at the IPython prompt.

Line-by-line Profiling with `%lprun`

The function-by-function profiling of `%prun` is useful, but sometimes it's more convenient to have a profile report line-by-line. This is not built-in to Python or IPython, but there is a `line_profiler` package available for installation which can do this. Start by `pip` installing the `line_profiler` package by running

```
$ pip install line_profiler
```

(For more information on `pip`, see section X.X). Next, you can use IPython to load the `line_profiler` IPython extension, offered as part of this package:

```
%load_ext line_profiler
```

Now the `%lprun` command will do a line-by-line profiling of any function: in this case we need to tell it explicitly which functions we're interested in profiling:

```
%lprun -f sum_of_lists sum_of_lists(5000)
```

As above, the notebook sends the result to the pager, but it looks something like this:

```
Timer unit: 1e-06 s

Total time: 0.009382 s
File: <ipython-input-19-fa2be176cc3e>
Function: sum_of_lists at line 1

Line #      Hits       Time  Per Hit   % Time  Line Contents
=====      ===      =====  ======   =====
    1                      def sum_of_lists(N):
    2      1            2      2.0      0.0      total = 0
    3      6            8      1.3      0.1      for i in range(5):
    4      5  9001     1800.2    95.9
    5      5            371    74.2      4.0      total += sum(L)
    6      1            0      0.0      0.0      return total
```

The information at the top gives us the key to reading the results: the time is reported in microseconds and we can see where the program is spending the most time. At this point, we may be able to use this information to modify aspects of the script and make it perform better for our desired use case.

For more information on `%lprun` and its available options, use the IPython help functionality, i.e. type `%lprun?` at the IPython prompt.

Profiling Memory Use: %memit and %mprun

Another aspect of profiling is the amount of memory an operation uses. This can be evaluated with another IPython extension, the `memory_profiler`. As with the `line_profiler` above, we start by pip-installing the extension:

```
$ pip install memory_profiler
```

Then we can use IPython to load the extension:

```
%load_ext memory_profiler
```

The memory profiler extension contains two useful magic functions: the `%memit` magic, which offers a memory-measuring equivalent of `%timeit`, and the `%mprun` function, which offers a memory-measuring equivalent of `%lprun`. The `%memit` function can be used rather simply:

```
%memit sum_of_lists(1000000)
peak memory: 100.08 MiB, increment: 61.36 MiB
```

We see that this function uses about 100 megabytes of memory.

For a line-by-line description of memory use, we can use the `%mprun` magic. Unfortunately, this magic works only for functions defined in separate modules rather than the notebook itself, so we'll start by using the `%%file` magic to create a simple module called `mprun_demo.py` which contains our `sum_of_lists` function, with one addition that will make our memory profiling results more clear:

```
%%file mprun_demo.py
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
        del L # remove reference to L
    return total
```

Overwriting mprun_demo.py

We can now import the new version of this function and run the memory line profiler.

```
from mprun_demo import sum_of_lists
%mprun -f sum_of_lists sum_of_lists(1000000)
```

The result, printed to the pager, gives us a summary of the memory use of the function, and looks something like this:

```
Filename: ./mprun_demo.py

Line #      Mem usage      Increment   Line Contents
=====
```

```
4      71.9 MiB      0.0 MiB          L = [j ^ (j >> i) for j in range(N)]
```

Filename: ./mprun_demo.py

Line #	Mem usage	Increment	Line Contents
1	39.0 MiB	0.0 MiB	def sum_of_lists(N):
2	39.0 MiB	0.0 MiB	total = 0
3	46.5 MiB	7.5 MiB	for i in range(5):
4	71.9 MiB	25.4 MiB	L = [j ^ (j >> i) for j in range(N)]
5	71.9 MiB	0.0 MiB	total += sum(L)
6	46.5 MiB	-25.4 MiB	del L # remove reference to L
7	39.1 MiB	-7.4 MiB	return total

Here the `increment` column tells us how much each line affects the total memory budget: observe that when we create and delete the list `L`, we are adding about 25 megabytes of memory usage. This is on top of the background memory usage from the Python interpreter itself.

For more information on `%memit`, `%mprun`, and their available options, use the IPython help functionality, i.e. type `%memit?` at the IPython prompt.

More IPython Resources

In this chapter we've just scratched the surface of using IPython to enable data science tasks. Much more information is available both in print and on the web, and here we'll list some other resources that you may find helpful.

Web Resources

- <http://ipython.org>: The official IPython documentation. It includes examples, tutorials, detailed documentation, and links to a variety of other resources.
- <http://nbviewer.ipython.org/>: The nbviewer site shows static renderings of any IPython notebook available on the internet. The front page features some example notebooks that you can browse to see what other folks are using IPython for!
- <http://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks/>: This ever-growing list of notebooks, powered by nbviewer, shows the depth and breadth of numerical analysis you can do with IPython. Included in the list are everything from short examples and tutorials to full-blown courses and books composed in the notebook format!
- <http://pyvideo.org/>: search especially for some of the excellent video tutorials by Fernando Perez, the creator of IPython and leader of the IPython team.

Books

- *Python for Data Analysis* by Wes McKinney (O'Reilly Publishing, 2013) has a chapter which covers using IPython as a data scientist. Though much of the material overlaps what we've discussed here, another perspective is always helpful.
- *Learning IPython for Interactive Computing and Data Visualization* by Cyrille Rossant (Packt Publishing, 2013) is a short introductory book on IPython for data analysis.
- *IPython Interactive Computing and Visualization Cookbook* by Cyrille Rossant (Packt Publishing, 2014) is a longer and more advanced treatment of using IPython for data science. Despite its name, it's not just about IPython, but also goes into some depth on a broad range of data science topics.

Finally, a reminder that you can find help on your own: IPython's ?-based help functionality discussed in section X.X can be very useful if you use it well and use it often. As you go through the examples here and elsewhere, this can be used to familiarize yourself with all the tools that IPython has to offer.

CHAPTER 3

Introduction to NumPy

The next two chapters outline techniques for effectively working with data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats: they could be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

For example, images, and in particular digital images, can be thought of as simply a two dimensional array of numbers representing pixel brightness across the area. Sound clips can be thought of as one-dimensional arrays of intensity versus time. Text can be converted in various ways into numerical representations, perhaps binary digits representing the frequency of certain words or pairs of words. No matter what the data is, the first step in making it analyzable will be to transform it into arrays of numbers.

For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science. Here we'll take a look at the specialized tools that Python has to handle such numerical arrays: the *NumPy* package, and the *Pandas* package.

This chapter will cover NumPy in detail. NumPy, short for “Numerical Python”, provides an efficient interface to store and operate on dense data buffers. In many ways, NumPy arrays are like Python's built-in `list` type, but NumPy arrays provide much more efficient storage and data operations as the size of the arrays grow larger. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice from the preface and installed the Anaconda stack, you have NumPy installed and ready to go. If you're more the do-it-yourself type, you can navigate to <http://www.numpy.org/> and follow the installation instructions found there. Once you do, you can import numpy and double-check the version:

```
import numpy  
numpy.__version__  
'1.9.1'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import numpy using np as an alias:

```
import numpy as np
```

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

Reminder about Built-in Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the "?" character).

For example, you can type

```
In [3]: np.<TAB>
```

to display all the contents of the numpy namespace, and

```
In [4]: np?
```

to display NumPy's built-in documentation. More detailed documentation, along with tutorials and other resources, can be found at <http://www.numpy.org>.

Understanding Data Types in Python

Effective data-driven science and computation requires understanding how data is stored and manipulated. This section outlines and contrasts how arrays of data are handled in the Python language itself, and how NumPy improves on this. Understanding this difference is fundamental to understanding much of the material throughout the rest of the book.

Users of Python are often drawn-in by its ease of use, one piece of which is dynamic typing. While a statically-typed language like C or Java requires each variable to be explicitly declared, a dynamically-typed language like Python skips this specification.

For example, in C you might specify a particular operation as follows:

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be written this way:

```
# Python code
result = 0
for i in range(100):
    result += i
```

Notice the main difference: in C, the data types of each variable are explicitly declared, while in Python the types are dynamically inferred. This difference certainly contributes to the ease Python provides for translating algorithms to code, but to really understand data in Python we must first understand what is happening under the hood.

As we have mentioned several times, Python variables are dynamically typed. This means, for example, that we can assign any kind of data to any variable:

```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an integer to a string.

The same thing in C would lead (depending on compiler settings) to a compilation error or other unintended consequences:

```
/* C code */
int x = 4;
x = "four"; // FAILS
```

This sort of flexibility is one piece that makes Python and other dynamically-typed languages convenient and easy to use. Understanding *how* this works is an important piece of learning to analyze data efficiently and effectively with Python. But what this type-flexibility also points to is the fact that Python variables are more than just their value; they also contain extra information about the type of the value. We'll explore this more below.

A Python Integer is More than just an Integer

The standard Python implementation is written in C. This means that every Python object is simply a cleverly-disguised C structure, which contains not only its value, but other information as well.

For example, when we define an integer in Python,

```
x = 10000
```

`x` is not just a “raw” integer. It’s actually a pointer to a compound C structure, which contains several values. Looking through the Python 3.4 source code, we find that the long integer type definition effectively looks like this (once the C macros are expanded):

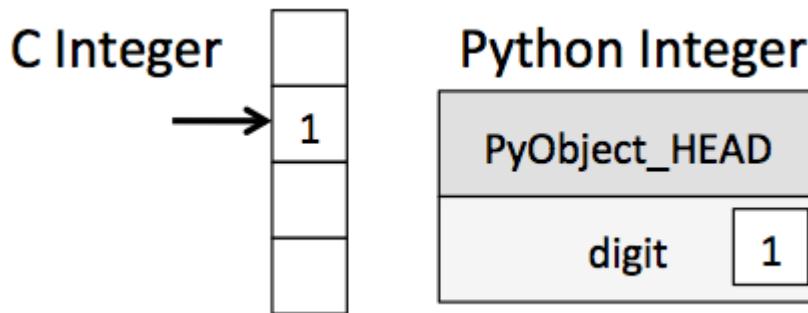
```
struct _longobject {  
    long ob_refcnt;  
    PyTypeObject *ob_type;  
    size_t ob_size;  
    long ob_digit[1];  
};
```

A single integer in Python 3.4 actually contains four pieces:

- `ob_refcnt`, a reference count which helps Python silently handle memory allocation and deallocation
- `ob_type`, which encodes the type of the variable
- `ob_size`, which specifies the size of the following data members
- `ob_digit`, which contains the actual integer value that we expect the Python variable to represent.

This means that there is some overhead in storing an integer in Python as compared to an integer in a compiled language like C.

We can visualize this as follows:



Here `PyObject_HEAD` is the part of the structure containing the reference count, type code, and other pieces mentioned above.

Notice the difference here: a C integer is essentially a label for a position in memory whose bytes encode an integer value. A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes which contain the integer value. This extra information in the Python integer structure is what allows Python to be coded so freely and dynamically. All this additional infor-

mation in Python types comes at a cost, however, which becomes especially apparent in structures which combine many of these objects.

A Python List is More than just a List

Let's consider now what happens when we use a Python data structure which holds many Python objects. The standard mutable multi-element container in Python is the list. We can create a list of integers as follows:

```
L = list(range(10))
L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
type(L[0])
int
```

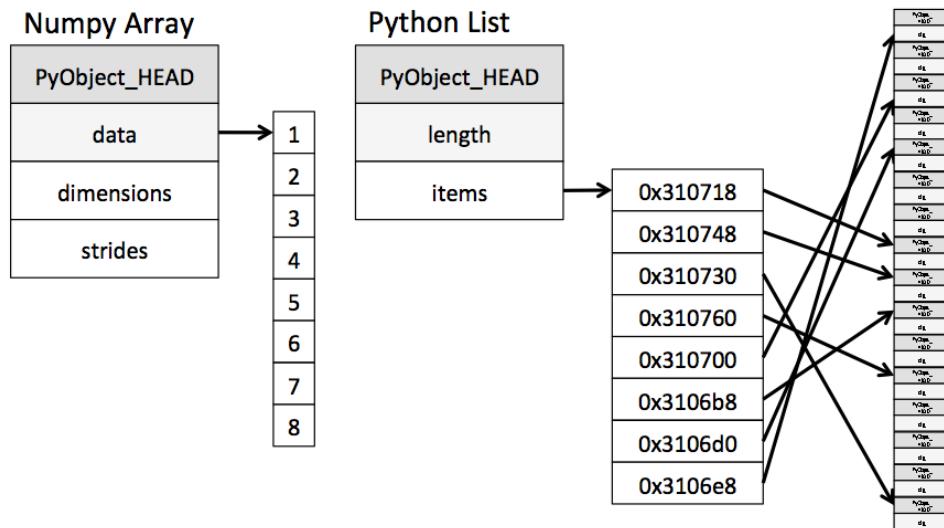
Or similarly a list of strings:

```
L2 = [str(c) for c in L]
L2
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
type(L2[0])
str
```

Because of Python's dynamic typing, we can even create heterogeneous lists:

```
L3 = [True, "2", 3.0, 4]
[type(item) for item in L3]
[bool, str, float, int]
```

But this flexibility comes at a cost: to allow these flexible types, each item in the list must contain its own type info, reference count, and other information; that is, each item is a complete Python object. In the special case that all variables are of the same type, much of this information is redundant: it can be much more efficient to store data in a fixed-type array. The difference between a dynamic-type list and a fixed-type (NumPy-style) array is illustrated in the following figure:



At the implementation level, the array essentially contains a single pointer to one contiguous block of data. The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw above. Again, the advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

Fixed-type arrays in Python

Python offers several different options for storing data in efficient, fixed-type data buffers. Built-in since Python 3.3 is the `array` module, which can be used to create dense arrays of a uniform type:

```
import array
L = list(range(10))
A = array.array('i', L)
A

array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Here '`i`' is a type code indicating the contents are integers. Much more useful, however, is the `ndarray` object of the NumPy package. While Python's `array` object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data. We will explore these operations in later sections; here we'll demonstrate several ways of creating a numpy array.

We'll start with the standard NumPy import, under the alias `np`:

```
import numpy as np
```

Creating Arrays from Python Lists

First, we can use `np.array` to create arrays from Python lists:

```
# integer array:  
np.array([1, 4, 2, 5, 3])  
array([1, 4, 2, 5, 3])
```

Remember that unlike Python lists, NumPy is constrained to arrays which all contain the same type. If types do not match, NumPy will upcast if possible (here, integers are up-cast to floating point):

```
np.array([3.14, 4, 2, 3])  
array([ 3.14, 4., 2., 3.])
```

If we want to explicitly set the datatype of the resulting array, we can use the `dtype` keyword:

```
np.array([1, 2, 3, 4], dtype='float32')  
array([ 1., 2., 3., 4.], dtype=float32)
```

Finally, unlike Python lists, NumPy arrays can explicitly be multi-dimensional; here's one way of initializing a multidimensional array using a list of lists:

```
# nested lists result in multi-dimensional arrays  
np.array([range(i, i + 3) for i in [2, 4, 6]])  
array([[2, 3, 4],  
[4, 5, 6],  
[6, 7, 8]])
```

The inner lists are treated as rows of the resulting two-dimensional array.

Creating arrays from scratch

Especially for larger arrays, it is more efficient to create arrays from scratch using routines built-in to NumPy. Here are several examples:

```
# Create a length-10 integer array filled with zeros  
np.zeros(10, dtype=np.int)  
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])  
  
# Create a (3 x 5) floating-point array filled with ones  
np.ones((3, 5), dtype=float)  
array([[ 1., 1., 1., 1., 1.],  
[ 1., 1., 1., 1., 1.],  
[ 1., 1., 1., 1., 1.]])  
  
# Create a (3 x 5) array filled with 3.14  
np.full((3, 5), 3.14)
```

```

array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]]))

# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
np.arange(0, 20, 2)

array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

# Create an array of 5 values evenly spaced between 0 and 1
np.linspace(0, 1, 5)

array([ 0. ,  0.25,  0.5 ,  0.75,  1. ])

# Create a (3 x 3) array of uniformly-distributed
# random values between 0 and 1
np.random.random((3, 3))

array([[ 0.57553592,  0.91443811,  0.50999268],
       [ 0.59732101,  0.10454524,  0.23510437],
       [ 0.10431603,  0.0562055 ,  0.47216143]]))

# Create a (3 x 3) array of normally-distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))

array([[ 0.16103055, -0.97851038, -4.40762392],
       [-0.55868781,  0.87953933,  1.95440309],
       [ 0.19103472,  0.61948762,  0.19737634]]))

# Create a (3 x 3) array of random integers in the interval
# [0, 10)
np.random.randint(0, 10, (3, 3))

array([[1, 3, 0],
       [6, 0, 3],
       [5, 9, 7]])

# Create a (3 x 3) identity matrix
np.eye(3)

array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]]) 

# Create an uninitialized array of 3 integers
# The values will be whatever happens to already exist at that memory location
np.empty(3)

array([ 1.,  1.,  1.])

```

NumPy Standard Data Types

Because NumPy arrays contain values of a single type, it is important to have detailed knowledge of those types and their limitations. Because NumPy is built in C, the types will be familiar to users of C, Fortran, and other related languages.

The standard numpy data types are listed in the following table. Note that when constructing an array, they can be specified using a string, e.g.

```
np.zeros(10, dtype='int16')
```

or using the associated numpy object, e.g.

```
np.zeros(10, dtype=np.int16)
```

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

More advanced type specification is possible, such as specifying big or little endian numbers; for more information please refer to the numpy documentation at <http://numpy.org/>. NumPy also supports compound data types, which will be covered in section X.X.

The Basics of NumPy Arrays

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (Chapter X.X) are built around the NumPy array. This section will give several examples of manipulation of NumPy arrays to access data and subarrays, and to split, reshape, and join the arrays. While the types of operations shown 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 go over some useful array attributes. We'll start by defining three random arrays, a 1-dimensional, 2-dimensional, and 3-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value: this will ensure that the same random arrays are generated each time this code is run.

```
import numpy as np
np.random.seed(0) # seed for reproducibility

x1 = np.random.randint(10, size=6) # 1D array
x2 = np.random.randint(10, size=(3, 4)) # 2D array
x3 = np.random.randint(10, size=(3, 4, 5)) # 3D array
```

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

```
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`, short for the data type of the array, which we discussed in the previous section:

```
print("dtype:", x3.dtype)
```

```
dtype: int64
```

(See the table of built-in data types in Section X.X).

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

```
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

We saw previously that individual items in Python lists can be accessed with square brackets and a zero-based integer index; NumPy arrays use similar notation with some additional enhancements for arrays with multiple dimensions.

In a one-dimensional array, the i^{th} can be accessed by specifying the desired index in square brackets, just as with Python lists:

```
x1
array([5, 0, 3, 3, 7, 9])
x1[0]
5
x1[4]
7
```

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

```
x1[-1]
9
x1[-2]
7
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

```
x2
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
x2[0, 0]
3
```

```
x2[2, 0]  
1  
x2[2, -1]  
7
```

Values can also be modified using any of the above index notation:

```
x2[0, 0] = 12  
x2  
array([[12,  5,  2,  4],  
       [ 7,  6,  8,  8],  
       [ 1,  6,  7,  7]])
```

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

```
x1[0] = 3.14159 # this will be truncated!  
x1  
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 use them 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 of an array `x`, use

```
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 sub-arrays in one dimension and in multiple dimensions:

One-dimensional Subarrays

```
x = np.arange(10)  
x  
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  
x[:5] # first five elements  
array([0, 1, 2, 3, 4])  
x[5:] # elements after index 5  
array([5, 6, 7, 8, 9])  
x[4:7] # middle sub-array  
array([4, 5, 6])
```

```
x[::2] # every other element
array([0, 2, 4, 6, 8])
x[1::2] # every other element, starting at index 1
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:

```
x[::-1] # all elements, reversed
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
x[5::-2] # reversed every other from index 5
array([5, 3, 1])
```

Multi-dimensional Subarrays

Multi-dimensional slices work in the same way, except multiple slices can be specified. For example:

```
x2
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
x2[:2, :3] # two rows, three columns
array([[12,  5,  2],
       [ 7,  6,  8]])
x2[:, ::2] # all rows, every other column
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

```
x2[::-1, ::-1]
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

Accessing Array Rows and Columns. One commonly-needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (`:`):

```
print(x2[:, 0]) # first column of x2
[12  7  1]
```

```
print(x2[0, :]) # first row of x2  
[12 5 2 4]
```

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

```
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. (If you’re familiar with Python lists, keep in mind that this is different behavior: In lists, slices are copies by default) Consider our two-dimensional array from above:

```
print(x2)  
[[12 5 2 4]  
 [ 7 6 8 8]  
 [ 1 6 7 7]]
```

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

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

Now if we modify this sub-array, we’ll see that the original array is changed! Observe:

```
x2_sub[0, 0] = 99  
print(x2_sub)  
[[99 5]  
 [ 7 6]]  
  
print(x2)  
[[99 5 2 4]  
 [ 7 6 8 8]  
 [ 1 6 7 7]]
```

This is actually an extremely useful default behavior: it means that when we work with large datasets, we can access and process pieces of these datasets without copying the underlying data buffer. This works because NumPy arrays have a very flexible internal representation, which we’ll explore in more detail in section X.X.

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:

```
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:

```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)

[[42  5]
 [ 7  6]]

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 3x3 grid, you can do the following:

```
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 non-contiguous memory buffers this is not always the case.

Another common reshaping pattern is the conversion of a 1D array into a 2D row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

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

# row vector via reshape
x.reshape((1, 3))

array([[1, 2, 3]])

# row vector via newaxis
x[np.newaxis, :]

array([[1, 2, 3]])

# column vector via reshape
x.reshape((3, 1))
```

```
array([[1,
       [2],
       [3]]])

# column vector via newaxis
x[:, np.newaxis]

array([[1],
       [2],
       [3]])
```

We'll make use of these types of transformations often through the remainder of the text.

Array Concatenation and Splitting

All of the above 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 using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `Concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

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

array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))

[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

```
grid = np.array([[1, 2, 3],
                [4, 5, 6]])

# concatenate along the first axis
np.concatenate([grid, grid])

array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])

# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

```
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:

```
x = np.array([1, 2, 3])  
grid = np.array([[9, 8, 7],  
                [6, 5, 4]])  
  
# vertically stack the arrays  
np.vstack([x, grid])  
  
array([[1, 2, 3],  
       [9, 8, 7],  
       [6, 5, 4]])  
  
# horizontally stack the arrays  
y = np.array([[99],  
              [99]])  
np.hstack([grid, y])  
  
array([[ 9,  8,  7, 99],  
       [ 6,  5,  4, 99]])
```

Similary, `np.dstack` will stack three-dimensional 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:

```
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, leads to $N + 1$ subarrays. The related functions `np.hsplit` and `np.vsplit` are similar:

```
grid = np.arange(16).reshape((4, 4))  
grid  
  
array([[ 0,  1,  2,  3],  
       [ 4,  5,  6,  7],  
       [ 8,  9, 10, 11],  
       [12, 13, 14, 15]])  
  
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]]

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.dsplit` will split arrays along the third axis.

Summary

This section has covered many of the basic patterns used in NumPy to examine the content of arrays, to access elements and portions of arrays, to reshape arrays, and to join and split arrays. These operations are the building-blocks of more sophisticated recipes that we'll see later in the book. They may seem a bit dry and pedantic, but these patterns are a bit like the conjugations and grammar rules that must be memorized when first learning a foreign language: an absolutely essential foundation for the more interesting material to come. Get to know these patterns well!

Random Number Generation

Often in computational science it is useful to be able to generate sequences of random numbers. This might seem simple, but after scratching at the surface it reveals itself to be a much more subtle problem: computers are entirely deterministic machines: can a computer algorithm ever be said to generate anything truly random? Further, how might we even define what traits a random sequence would have?

Given these questions, many purists will make fine distinctions between random numbers, pseudo-random numbers, quasi-random numbers, etc. Here we'll follow Press et al. [REF] and ignore such subtleties in favor of a more practical (if perhaps less satisfying) definition:

the deterministic program that produces a random sequence should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that uses its output.

That is, a sequence of random numbers is random if it's random enough for the purposes we're using it for. This is an admittedly circular definition, but ends up being very useful in practice. This section outlines how to generate random numbers and arrays of random numbers within Python. After exploring a simple Python imple-

mentation of a deterministic pseudorandom number generator, it covers the efficient random number tools available in Python and NumPy.

Understanding a Simple “Random” Sequence

Before exploring the tools available in Python and NumPy, we’ll briefly code from-scratch a simple pseudorandom number generator. The code in this section is for example only, and should not be used in practice: below we’ll see examples of the much more efficient and robust random number generators available in Python. Here the goal is to give some insight into the principles behind these algorithms and the resulting strengths and weaknesses.

While it is possible to obtain sequences of truly random numbers through physical means (by, e.g. monitoring decay events of a radioactive substance) most algorithmic random number generators used in practice are simple deterministic algorithms designed to produce a sequence of pseudo-random numbers with suitable properties. The simplest of these are *linear congruential generators*, which generate a sequence of integers r_i using the following recursive form:

$$r_{i+1} = (ar_i + c) \text{ MOD } m$$

where a , c , and m are chosen such that the resulting numbers have useful properties. For example, we can follow Knuth and Lewis [REF] who suggest generating 32-bit uniform deviates using $m = 2^{32}$, $a = 1664525$, and $c = 1013904223$. We can implement this step simply in Python:

```
def LCG_next(r, a=1664525, c=1013904223, m=2 ** 32):
    """
    Generate the next pseudorandom number
    using a Linear Congruential Generator
    """
    return (a * r + c) % m

seed = 0
for i in range(5):
    seed = LCG_next(seed)
    print(seed)

1013904223
1196435762
3519870697
2868466484
1649599747
```

Often, rather than creating a sequence of integers, it is more convenient to create a sequence of *uniform deviates*: that is, numbers distributed uniformly in the half-open interval $[0, 1)$. Here we’ll implement this by making use of Python’s convenient generator syntax (see section X.X). We’ll also add code which will automatically seed the sequence based on the current microseconds in the system clock.

```

from datetime import datetime
from itertools import islice

def LCG_generator(seed=None, a=1664525, c=1013904223, m=2 ** 32):
    """
    Linear Congruential generator of pseudorandom numbers
    """
    if seed is None:
        # If seed is not provided, use current microseconds
        seed = datetime.now().microsecond

    while True:
        seed = LCG_next(seed, a=a, c=c, m=m)
        yield float(seed) / m

def simple_uniform_deviate(N, seed=None):
    """
    return a list of N pseudorandom numbers
    in the interval [0, 1)
    """
    gen = LCG_generator(seed)
    return list(islice(gen, N))

# Print a list of 5 random numbers
simple_uniform_deviate(5)

[0.7377541123423725,
 0.3999146604910493,
 0.18632183666341007,
 0.591240135487169,
 0.22258975286968052]

```

Again, the above code is *not* an ideal way to generate sequences of random numbers; it is included to give you a bit of insight into *how* pseudorandom numbers are generated in practice. We've started with a seed value, which defaults to some changing value or other source of pseudo-randomness available in the operating system. From this seed, we construct an algorithm which creates a deterministic sequence of values which are sufficiently random for our purposes. All pseudorandom number generators share this determinism.

Below we'll see some more efficient and sophisticated random number generators built-in to Python and to NumPy. Despite their complexity, under the hood they are just deterministic algorithms which step from one seed to the next, albeit with more involved steps than we used above.

Built-in tools: Python's `random` module

Python has a built-in `random` module which uses the sophisticated Mersenne Twister algorithm [REF] to generate sequences of uniform pseudorandom numbers:

```
import random
[random.random() for i in range(5)]

[0.6307611680584317,
 0.4050472985131417,
 0.4218846620497734,
 0.7692476200181592,
 0.15093482641991562]
```

The seed is implicitly set from an operating system source such as the current time. Alternatively, we can set the seed explicitly from any hashable object:

```
random.seed(100)
[random.random() for i in range(5)]

[0.1456692551041303,
 0.45492700451402135,
 0.7707838056590222,
 0.705513226934028,
 0.7319589730332557]
```

Reseeding with the same value will lead to identical results, as we can confirm:

```
random.seed(100)
[random.random() for i in range(5)]

[0.1456692551041303,
 0.45492700451402135,
 0.7707838056590222,
 0.705513226934028,
 0.7319589730332557]
```

The built-in `random` module has many more functions and features; for more information refer to the Python documentation at <http://www.python.org/>. Rather than digging further into Python's `random` module, we'll instead move our focus to the random number tools included with NumPy. These are optimized to generate sequences of random numbers, and will end up being much more useful for the types of data science tasks we will tackle in this book.

Efficient Random Numbers: `numpy.random`

The `numpy.random` submodule, like Python's built-in `random` module, is based on the Mersenne Twister algorithm. Unlike the Python's built-in tools, it is optimized for generating large arrays of random numbers.

Uniform Deviates

The basic interface is the `rand` function, which generates uniform deviates:

```
import numpy as np  
np.random.rand()  
  
0.6065604039577245
```

Arrays of any size and shape can be created with this function:

```
np.random.rand(5)  
  
array([ 0.11763091,  0.68041723,  0.25315934,  0.41112628,  0.17916994])  
  
np.random.rand(3, 3)  
  
array([[ 0.47070662,  0.19456102,  0.96443832],  
       [ 0.13359021,  0.4849765 ,  0.69473637],  
       [ 0.04753502,  0.88342211,  0.60243267]])
```

Random Seed

Though we did not set the seed above, it was implicitly set based on a system-dependent source of randomness. As above, specifying an explicit seed leads to reproducible sequences:

```
np.random.seed(2)  
print(np.random.rand(5))  
  
np.random.seed(2)  
print(np.random.rand(5))  
  
[ 0.4359949  0.02592623  0.54966248  0.43532239  0.4203678 ]  
[ 0.4359949  0.02592623  0.54966248  0.43532239  0.4203678 ]
```

We'll use explicit seeds like this throughout this book: they can be very useful for making demonstrations and results reproducible.

Random Integers

Random integers can be obtained using the `np.random.randint` function:

```
# 10 random integers in the interval [0, 10)  
np.random.randint(0, 10, size=10)  
  
array([2, 1, 5, 4, 4, 5, 7, 3, 6, 4])
```

Note that here the upper bound is exclusive: that is, the value 10 will never appear in the above list. A related function is `np.random.random_integers`, which instead implements an inclusive upper bound:

```
# 10 random integers in the interval [0, 10]  
np.random.random_integers(0, 10, size=10)  
  
array([10,  3,  7,  6, 10,  1, 10,  3,  5,  8])
```

Permutations and Selections

Both of the above random integer functions return lists containing repeats. If instead you'd like a random permutation of non-repeating integers, you can use `np.random.permutation`:

```
x = np.arange(10)
np.random.permutation(x)

array([5, 9, 8, 0, 2, 1, 3, 7, 6, 4])
```

A related function is `np.random.choice`, which allows you to select N random values from any array, with or without replacement:

```
np.random.choice(x, 10, replace=False)

array([5, 3, 8, 6, 7, 0, 1, 9, 4, 2])
np.random.choice(x, 10, replace=True)

array([9, 8, 7, 1, 6, 8, 5, 9, 9, 9])
```

Neither of these functions modify the original array; if we'd like to shuffle the array in-place we can use the `np.random.shuffle` function:

```
print(x)
np.random.shuffle(x)
print(x)

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

Normally-distributed Values

While uniform deviates and collections of integers are useful, there are many other distribution functions that are useful in practice. Perhaps the best-known is the normal distribution, where the probability density function is

$$p(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-(x-\mu)^2}{2\sigma^2}\right]$$

where μ is the mean, and σ is the standard deviation. Values can be drawn from a *standard normal*, which has $(\mu, \sigma) = (0, 1)$, using the `np.random.randn` function:

```
# Sequence of normally-distributed values
np.random.randn(10)

array([-0.52715869,  1.11385518, -1.75408685, -0.24978025,  0.54959138,
       0.52170749, -1.52343212,  0.08266405, -0.43774428,  1.57694963])
```

To specify different values of μ and σ , you can use the `np.random.normal` function:

```
# mean=10, stdev=2, size=(3, 3)
np.random.normal(10, 2, (3, 3))
```

```
array([[ 13.8069827 ,  10.05675847,   9.65668028],
       [ 13.7363806 ,   9.83213452,  11.04451775],
       [ 12.24302322,   8.483193  ,  10.34135175]])
```

Other Distributions

There are numerous other distribution functions available in the `np.random` submodule, more than we can cover here.

For example, you can create a sequence of poisson-distributed integers:

```
# poisson distribution for lambda=5
np.random.poisson(5, size=10)

array([ 6,  5,  9, 10,  3, 10,  7,  0,  4,  2])
```

You can create a sequence of exponentially-distributed values:

```
# exponential distribution with scale=1
np.random.exponential(1, size=10)

array([ 2.037982 ,  0.32218776,  0.15689987,  0.70402945,  0.4081106 ,
       0.1763738 ,  0.24755148,  1.81105024,  1.27420941,  0.05286104])
```

For information on further available random distributions refer to the documentation of the `np.random` submodule, and of the `scipy.stats.distributions` submodule.

Simultaneously Using Multiple Chains

Sometimes it is useful to have multiple random number sequences available concurrently; `numpy.random` provides the `RandomState` class for this purpose. Under the hood, the above functions are simply making use of a single global instance of this class. It can be instantiated and used as follows:

```
# instantiate a random number generator
# if seed is not specified, it will be seeded
# with a system-dependent source of randomness
rng = np.random.RandomState(seed=2)
```

Once this class instance is created, many of the above functions can be used as a method of the class. For example:

```
rng.rand(3, 3)
array([[ 0.4359949 ,  0.02592623,  0.54966248],
       [ 0.43532239,  0.4203678 ,  0.33033482],
       [ 0.20464863,  0.61927097,  0.29965467]])

rng.randint(0, 10, size=5)
array([4, 4, 5, 7, 3])

rng.randn(3, 3)
```

```
array([[ 2.6460672 , -0.04386375, -0.96561968],  
       [ 0.87866389, -2.24587483,  1.11957525],  
       [-1.054368 , -1.0088915 , -0.06752199]])
```

Random Numbers: Further Resources

This section has been a quick introduction to some of the pseudorandom number generators available in Python. For more information, refer to the following sources:

- <http://numpy.org>: the official NumPy documentation has much more information on the routines available in the `np.random` package
- <http://scipy.org>: the `scipy.stats.distributions` submodule contains implementations of many more advanced and obscure statistical distributions, including the ability to draw random samples from these distributions.
- Press et al. [REF] contains a more thorough discussion of (pseudo)random number generation, including common algorithms and their strengths and weaknesses.
- Ivezic et al. [REF] contains a Python-driven discussion of various distributions implemented in NumPy and SciPy, as well as examples of applications which depend on these tools.

Computation on NumPy Arrays: Universal Functions

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

The Slowness of Loops

Python's default implementation (known as CPython) does some operations very slowly. This is in part due to its dynamic nature: the fact that types are not specified, so that sequences of operations cannot be compiled-down to efficient machine code, as they are in languages like C and Fortran. Recently there have been various attempts to address this weakness: well-known examples are the [PyPy](#) project, a just-in-time compiled implementation of Python; the [Cython](#) project, which converts Python code to compilable C code; and the [Numba](#) project, which converts snippets of Python code to fast LLVM bytecode. Each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.

The slowness of Python generally manifests itself in situations where many small operations are being repeated: i.e. looping over arrays to operate on each element. For example, imagine we have an array of values and we'd like to compute the reciprocals of each. A straightforward approach might look like this:

```
import numpy as np
np.random.seed(0)

def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1 / values[i]
    return output

values = np.random.randint(1, 10, size=5)
compute_reciprocals(values)

array([ 0.16666667,  1.          ,  0.25        ,  0.25        ,  0.125       ])
```

This implementation probably feels fairly natural to someone from, say, a C or Java background. But if we measure the execution time of this code for a large input, we see that this operation is very slow, perhaps surprisingly so! Here we'll use IPython's `%timeit` magic function, which was introduced in sections X.X and X.X:

```
big_array = np.random.randint(1, 100, size=1E6)
%timeit compute_reciprocals(big_array)

1 loops, best of 3: 292 ms per loop
```

It takes significant fraction of a second to compute these million operations and to store the result! When even cell phones have processing speeds measured in Giga-FLOPS (i.e. billions of numerical operations per second), this seems almost absurdly slow. It turns out that the bottleneck here is not the operations themselves, but the dynamic type-checking that CPython must do at each cycle of the loop. Each time the reciprocal is computed, Python first examines the object's type and does a dynamic lookup of the correct function to use for that type. If we were working in compiled code instead, this type specification would be known before the code executes and the result could be computed much more efficiently.

Introducing UFuncs

For many types of operations, NumPy provides a convenient interface into just this kind of statically-typed, compiled routine. This is known as a *vectorized* operation. This can be accomplished by simply performing an operation on the array, which will then be applied to each element. The loop over each operation can then be pushed into the compiled layer of numpy, leading to much faster execution.

Compare the results of the following two:

```

print(compute_reciporicals(values))
print(1 / values)

[ 0.16666667  1.          0.25          0.25          0.125         ]
[ 0.16666667  1.          0.25          0.25          0.125         ]

```

Looking at the execution time for our big array, we see that it completes orders of magnitude faster than the Python loop:

```

%timeit (1 / big_array)
100 loops, best of 3: 8.09 ms per loop

```

This vectorized operation is known as a *ufunc* in NumPy, short for “Universal Function”. The main purpose of ufuncs is to quickly execute repeated operations by pushing the loops down into fast compiled code. They are extremely flexible: above we saw an operation between a scalar and an array; we can also operate between two arrays:

```

np.arange(5) / np.arange(1, 6)
array([ 0.          ,  0.5          ,  0.66666667,  0.75          ,
       0.8          ])

```

And they are not limited to one-dimensional arrays: they can also act on multi-dimensional arrays as well:

```

x = np.arange(9).reshape((3, 3))
2 ** x
array([[ 1,   2,   4],
       [ 8,  16,  32],
       [64, 128, 256]])

```

Vectorized operations such as ufuncs are nearly always more efficient than their counterpart implemented using Python loops. Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

Ufuncs exist in two flavors: *unary ufuncs*, which operate on a single input, and *binary ufuncs*, which operate on two inputs. We'll see examples of both these types of functions below.

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:

```

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)

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, and a `**` operator for exponentiation, and a `%` operator for modulus:

```

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 order of operations is respected:

```

-(0.5*x + 1) ** 2
array([-1. , -2.25, -4. , -6.25])

```

Each of these arithmetic operations are simply convenient wrappers around specific functions built-in to NumPy; for example, the `+` operator is simply a wrapper for the `add` function:

```

np.add(x, 2)
array([2, 3, 4, 5])

```

The following is a table of the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
<code>+</code>	<code>np.add</code>	addition (e.g. <code>1 + 1 = 2</code>)
<code>-</code>	<code>np.subtract</code>	subtraction (e.g. <code>3 - 2 = 1</code>)
<code>-</code>	<code>np.negative</code>	unary negation (e.g. <code>-2</code>)
<code>*</code>	<code>np.multiply</code>	multiplication (e.g. <code>2 * 3 = 6</code>)
<code>/</code>	<code>np.divide</code>	division (e.g. <code>3 / 2 = 1.5</code>)
<code>//</code>	<code>np.floor_divide</code>	floor division (e.g. <code>3 // 2 = 1</code>)
<code>**</code>	<code>np.power</code>	exponentiation (e.g. <code>2 ** 3 = 8</code>)
<code>%</code>	<code>np.mod</code>	modulus/remainder (e.g. <code>9 % 4 = 1</code>)

Here we are not covering boolean/bitwise operators: for a similar table of boolean operators and ufuncs, see section X.X.

Absolute value

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

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

array([2, 1, 0, 1, 2])
```

The corresponding NumPy ufunc is `np.absolute`, which is also available as `np.abs`:

```
np.absolute(x)

array([2, 1, 0, 1, 2])

np.abs(x)

array([2, 1, 0, 1, 2])
```

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

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

array([ 5.,  5.,  2.,  1.])
```

Trigonometric Functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

```
theta = np.linspace(0, np.pi, 3)
```

Now we can compute some trigonometric functions on these values:

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

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

```

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) = [-1.57079633  0.           1.57079633]
arccos(x) = [ 3.14159265  1.57079633  0.           ]
arctan(x) = [-0.78539816  0.           0.78539816]

```

Exponents and Logarithms

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

```

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   7.3890561   20.08553692]
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:

```

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 which are useful for maintaining precision with very small input:

```

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 to be 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 can reveal many interesting functionality.

Another excellent source for more specialized and obscure ufuncs is the `scipy.special` submodule. There are far too many functions here to list them all, but we'll go over some highlights:

```
from scipy import special

# Bessel functions:
x = [0, 1, 2]
print("J0(x) =", special.j0(x))
print("J1(x) =", special.j1(x))
print("J4(x) =", special.jn(x, 4))

J0(x) = [ 1.          0.76519769  0.22389078]
J1(x) = [ 0.          0.44005059  0.57672481]
J4(x) = [-0.39714981 -0.06604333  0.36412815]
```

See also `special.in` (modified Bessel functions) and `special.kn` (modified Bessel functions of the second kind).

```
# Gamma functions (generalized factorials) & 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+00  2.40000000e+01  3.62880000e+05]
ln|gamma(x)|  = [ 0.          3.17805383  12.80182748]
beta(x, 2)    = [ 0.5          0.03333333  0.00909091]

# 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

Ufuncs are very flexible beasts; they implement several interesting and useful features, which we'll quickly demonstrate below.

Specifying Output

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

```
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:

```
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 `y` array. This doesn't make much of a difference for such a small computation, but for very large arrays the memory savings from careful use of the `out` argument can be significant.

Aggregates

For binary ufuncs, there are some interesting aggregates that can be computed from Ufuncs. For example, if we'd like to *reduce* an array with a particular operation, we can use the `reduce` method of any ufunc. A `reduce` repeatedly applies a given operation to the elements of an array until only a single result is left.

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

```
x = np.arange(1, 6)
np.add.reduce(x)
```

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

```
np.multiply.reduce(x)
```

120

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

```
np.add.accumulate(x)
```

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

```
np.multiply.accumulate(x)
```

```
array([ 1,  2,  6, 24, 120])
```

Note that for these particular cases, there are dedicated NumPy functions to compute the results (`np.sum`, `np.prod`, `np.cumsum`, `np.cumprod`). We'll cover these and other aggregation functions in section X.X.

Outer Products

Finally, any ufunc can compute the output of all pairs of two different inputs using the `outer` method. This allows you, in one line, to do things like create a multiplication table:

```
x = np.arange(1, 13)
np.multiply.outer(x, x)
```

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24],
       [ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30, 33, 36],
       [ 4,  8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48],
       [ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60],
       [ 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72],
       [ 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84],
       [ 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96],
       [ 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99, 108],
       [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120],
       [11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132],
       [12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132, 144]])
```

Other extremely useful methods of ufuncs are the `ufunc.at` and `ufunc.reduceat` methods, which we'll explore when we cover *fancy indexing* in section X.X.

Finding More

More information on universal functions (including the full list of available functions) can be found on the NumPy and SciPy documentation websites:

- NumPy: <http://www.numpy.org>

- SciPy: <http://www.scipy.org>

Recall that you can also access information directly from within IPython by importing the above packages and using IPython's help utility:

```
In [32]: numpy?
```

or

```
In [33]: scipy.special?
```

There is another extremely useful aspect of ufuncs that this recipe did not cover: the important subject of **broadcasting**. We'll take a detailed look at this topic in section X.X.

Aggregations: Min, Max, and Everything In Between

Often when faced with a large amount of data, a first step is to compute summary 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” values in a dataset, but other aggregates are useful as well (the sum, product, median, min and max, quantiles, etc.)

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

Examples of NumPy Aggregates

Here we'll give a few examples of NumPy's aggregates

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 `sum` function

```
import numpy as np  
  
L = np.random.random(100)  
sum(L)  
47.48211649355256
```

The syntax is quite similar to that of NumPy's `sum` function, and the result is the same.

```
np.sum(L)  
47.482116493552553
```

Because the NumPy version executes the operation in compiled code, however, NumPy's version of the operation is computed much more quickly:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)

10 loops, best of 3: 93.4 ms per loop
1000 loops, best of 3: 597 µs per loop
```

Be careful, though: the `sum` function and the `np.sum` function are not identical, which can sometimes lead to confusion! In particular, their optional arguments have different meanings: `np.sum` is aware of multiple array dimensions, as we will see below.

Minimum and Maximum

Similarly, Python has built-in `min` and `max` functions, used to find the minimum value and maximum value of any given array:

```
min(big_array), max(big_array)
(2.3049172436229171e-06, 0.99999789905734671)
```

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

```
np.min(big_array), np.max(big_array)
(2.3049172436229171e-06, 0.99999789905734671)

%timeit min(big_array)
%timeit np.min(big_array)

10 loops, best of 3: 69 µs per loop
1000 loops, best of 3: 444 µs per loop
```

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

```
big_array.min(), big_array.max(), big_array.sum()
(2.3049172436229171e-06, 0.99999789905734671, 500500.73894520913)
```

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

Multi-dimensional 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:

```
M = np.random.random((3, 4))
print(M)

[[ 0.87592335  0.05691319  0.05410134  0.81738248]
 [ 0.60644247  0.5832756   0.91279565  0.38541931]
 [ 0.89088002  0.68781086  0.52309224  0.71095617]]
```

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

```
M.sum()  
7.1049926640949987
```

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

```
M.min(axis=0)  
array([ 0.60644247,  0.05691319,  0.05410134,  0.38541931])
```

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

Similarly, we can find the maximum value within each row:

```
M.max(axis=1)  
array([ 0.87592335,  0.91279565,  0.89088002])
```

The way the axis is specified here can be confusing to users coming from other languages. The `axis` keyword specifies the *dimension of the array which 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.

Other Aggregation Functions

NumPy provides many other aggregation functions which we won't discuss in detail. Additionally, most aggregates have a NaN-safe counterpart, which computes the result while ignoring missing values, which are marked by the special IEEE floating-point NaN value. Some of these NaN-safe functions were added NumPy 1.8-1.9, so they will not be available in older NumPy versions.

The following gives a table of useful aggregation functions available in numpy.

Function Name	Nan-safe Version	Description
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<code>N/A</code>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute median of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute variance
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmax</code>	<code>np.nanargmin</code>	Find index of minimum value

Function Name	NaN-safe Version	Description
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: How Tall is the Average US President?

Aggregates available in NumPy can be extremely useful for summarizing a set of values. As a simple example, let's consider the heights of all United States presidents. We have this data in the file `president_heights.csv`, which is a simple comma-separated list of labels and values:

```
!head -4 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 to read the file and extract this information; further information about Pandas and CSV files can be found in chapter X.X.

```
import pandas as pd
data = pd.read_csv('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]
```

Note that the heights are measured in centimeters. Now that we have this data array, we can compute a variety of summary statistics (note that all heights are measured in centimeters):

```
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:

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

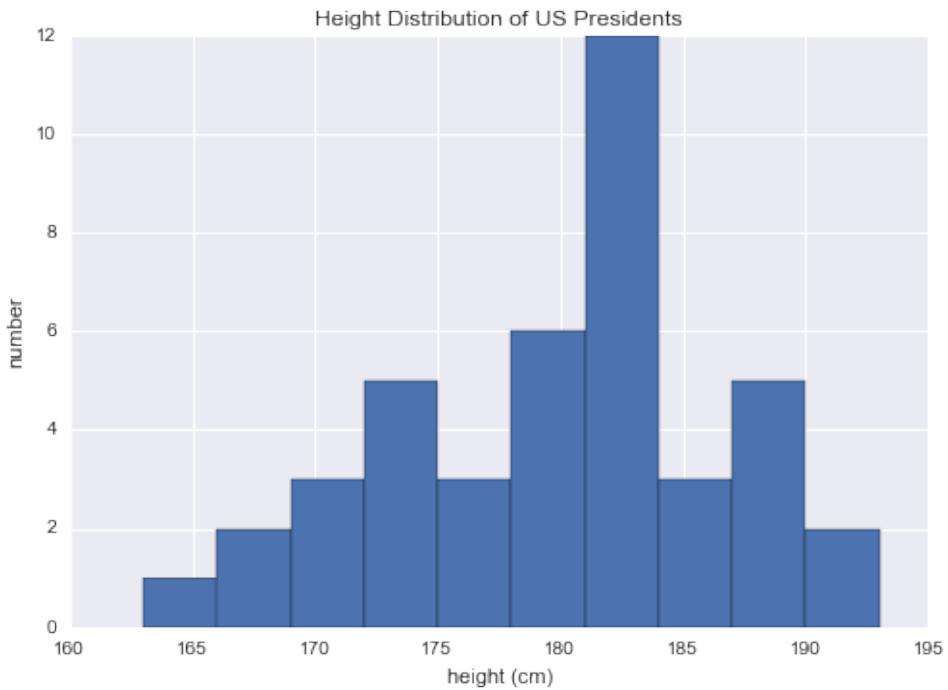
25th percentile:    174.25
Median:             182.0
75th percentile:   183.0
```

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

Of course, sometimes it's more useful to see a more visual representation of this data. We can do this using tools in Matplotlib, which will be discussed further in chapter X.X:

```
# See section X.X for a description of these imports
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot style

plt.hist(heights)
plt.title('Height Distribution of US Presidents')
plt.xlabel('height (cm)')
plt.ylabel('number');
```



These aggregates are some of the fundamental pieces of exploratory data analysis that we'll explore in more depth in later sections of the book.

Computation on Arrays: Broadcasting

We saw in the previous section how NumPy's Universal Functions can be used to *vectorize* operations and thereby remove slow Python loops. Another means of vectorizing operations is to use NumPy's *broadcasting* functionality. Broadcasting is simply a set of rules for applying universal functions like addition, subtraction, multiplication, and others 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:

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

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

```
a + 5  
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 that this duplication of values does not actually take place, but it's 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-dimensinoal array:

```
M = np.ones((3, 3))  
M  
array([[ 1.,  1.,  1.],  
       [ 1.,  1.,  1.],  
       [ 1.,  1.,  1.]])  
  
M + a  
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`.

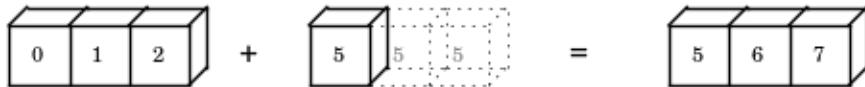
While these examples are relatively easy to understand, more complicated cases can involve broadcasting of both arrays. Consider the following example:

```
a = np.arange(3)  
b = np.arange(3)[:, np.newaxis]  
  
print(a)  
print(b)  
  
[0 1 2]  
[[0]  
 [1]  
 [2]]  
  
a + b  
array([[0, 1, 2],  
       [1, 2, 3],  
       [2, 3, 4]])
```

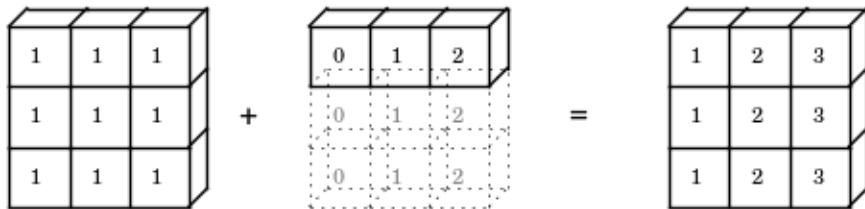
Just as above we stretched or broadcasted one value to match the shape of the other, here we've stretched *both* `a` and `b` to match a common shape, and the result is a two-

dimensional array! The geometry of the above examples is visualized in the following figure:

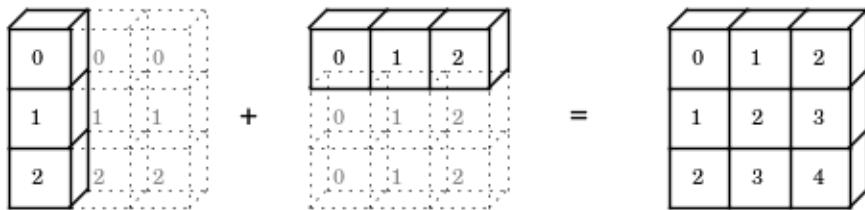
`np.arange(3) + 5`



`np.ones((3, 3)) + np.arange(3)`



`np.arange(3).reshape((3, 1)) + np.arange(3)`



The dotted boxes represent the broadcasted values: again, this extra memory is not actually allocated in the course of the operation, but it can be useful conceptually to imagine that it is.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

1. If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.
2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail:

Broadcasting Example 1:

Let's look at adding a two-dimensional array to a 1-dimensional array:

```
M = np.ones((2, 3))
a = np.arange(3)

print(M.shape)
print(a.shape)

(2, 3)
(3,)
```

Let's consider an operation on these two arrays. The shape of the arrays are

- `M.shape = (2, 3)`
- `a.shape = (3,)`

We see by rule 1 that the the array `a` has fewer dimensions, so we pad it on the left with ones:

- `M.shape -> (2, 3)`
- `a.shape -> (1, 3)`

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

- `M.shape -> (2, 3)`
- `a.shape -> (2, 3)`

The shapes match, and we see that the final shape will be `(2, 3)`:

```
M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

Broadcasting Example 2:

Let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

- `a.shape = (3, 1)`
- `b.shape = (3,)`

Rule 1 says we must pad the shape of `b` with ones:

- `a.shape -> (3, 1)`

- `b.shape` -> (1, 3)

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

- `a.shape` -> (3, 3)
- `b.shape` -> (3, 3)

Since the result matches, these shapes are compatible. We can see this here:

```
a + b  
array([[0, 1, 2],  
       [1, 2, 3],  
       [2, 3, 4]])
```

Broadcasting Example 3:

Now let's take a look at an example in which the two arrays are not compatible:

```
M = np.ones((3, 2))  
a = np.arange(3)  
  
print(M.shape)  
print(a.shape)  
  
(3, 2)  
(3,)
```

This is just a slightly different situation than in example 1: the matrix `M` is transposed. How does this affect the calculation? The shape of the arrays are

- `M.shape` = (3, 2)
- `a.shape` = (3,)

Again, rule 1 tells us that we must pad the shape of `a` with ones:

- `M.shape` -> (3, 2)
- `a.shape` -> (1, 3)

By rule 2, the first dimension of `a` is stretched to match that of `M`:

- `M.shape` -> (3, 2)
- `a.shape` -> (3, 3)

Now we hit rule 3: the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
M + a
```

```
ValueError                                Traceback (most recent call last)

<ipython-input-13-9e16e9f98da6> in <module>()
      1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the potential confusion here: you could imagine making `a` and `M` compatible by, say, padding the size of `a` with ones on the right rather than the left. But this is not how the broadcasting rules work! That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side-padding is what you'd like, you can do this explicitly by reshaping the array (we'll use the `np.newaxis` keyword that we introduce in section X.X):

```
a[:, np.newaxis].shape
(3, 1)
M + a[:, np.newaxis]
array([[ 1.,  1.],
       [ 2.,  2.],
       [ 3.,  3.]])
```

Also note that while we've been focusing on the "+" operator here, these broadcasting rules apply to *any* binary ufunc. For example, here is the `logaddexp(a, b)` function, which computes $\log(\exp(a) + \exp(b))$ with more precision than the naive approach

```
np.logaddexp(M, a[:, np.newaxis])
array([[ 1.31326169,  1.31326169],
       [ 1.69314718,  1.69314718],
       [ 2.31326169,  2.31326169]])
```

For more information on the many available universal functions, refer to section X.X.

Broadcasting in Practice

Broadcasting operations form the core of many examples we'll see throughout this book; here are a couple simple examples of where they can be useful:

Centering an Array

In the previous section we saw that ufuncs allow a NumPy user to remove the need to explicitly write slow Python loops. Broadcasting extends this ability. One commonly-seen example is when centering an array of data. Imagine you have an array of ten observations, each of which consists of three values. Using the standard convention, we'll store this in a 10×3 array:

```
X = np.random.random((10, 3))
```

We can compute the mean using the `mean` aggregate across the first dimension:

```
Xmean = X.mean(0)  
Xmean  
array([ 0.54579044,  0.61639938,  0.51815359])
```

And now we can center the `X` array by subtracting the mean (this is a broadcasting operation):

```
X_centered = X - Xmean
```

To double-check that we've done this correctly, we can check that the centered array has near zero mean:

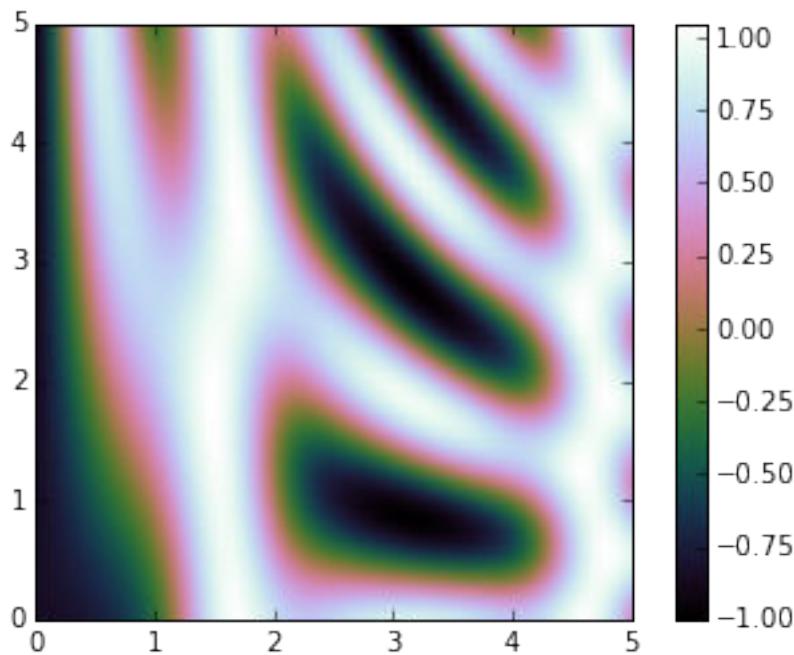
```
X_centered.mean(0)  
array([-1.11022302e-17,  6.66133815e-17, -8.88178420e-17])
```

To within machine precision, the mean is now zero.

Plotting a 2D function

One place that broadcasting is very useful is in displaying images based on 2D functions. If we want to define a function $z = f(x, y)$, broadcasting can be used to compute the function across the grid:

```
# x and y have 50 steps from 0 to 5  
x = np.linspace(0, 5, 50)  
y = np.linspace(0, 5, 50)[:, np.newaxis]  
  
z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)  
  
# Use of matplotlib is discussed further in section X.X  
%matplotlib inline  
import matplotlib.pyplot as plt  
  
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],  
           cmap=plt.cm.cubehelix)  
plt.colorbar();
```



More details on creating graphics with matplotlib can be found in Chapter X.X.

Utility Routines for Broadcasting

`np.broadcast` and `np.broadcast_arrays` are utility routines which can help with broadcasting.

`np.broadcast` will create a `broadcast` object which gives the shape of the broadcasted arrays, and allows iteration through all broadcasted sets of elements:

```
x = [['A'],
      ['B']]
y = [[1, 2, 3]]
xy = np.broadcast(x, y)

xy.shape
(2, 3)

for xi, yi in xy:
    print(xi, yi)

A 1
A 2
A 3
B 1
```

```
B 2  
B 3
```

`np.broadcast_arrays` takes input arrays and returns array views with the resulting shape and structure:

```
xB, yB = np.broadcast_arrays(x, y)  
print(xB)  
print(yB)  
  
[['A' 'A' 'A']  
 ['B' 'B' 'B']]  
[[1 2 3]  
 [1 2 3]]
```

These two functions will prove useful when working with arrays of different sizes; we will occasionally see them in action through the remainder of this text.

Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers which are above some threshold. In NumPy, boolean masking is often the most efficient way to accomplish these types of tasks.

Example: Counting Rainy Days

Imagine you have a series of data that represents the amount of precipitation each day for a year in a given city. For example, here we'll load the daily rainfall statistics for the city of Seattle in 2014, using the `pandas` tools covered in more detail in section X.X:

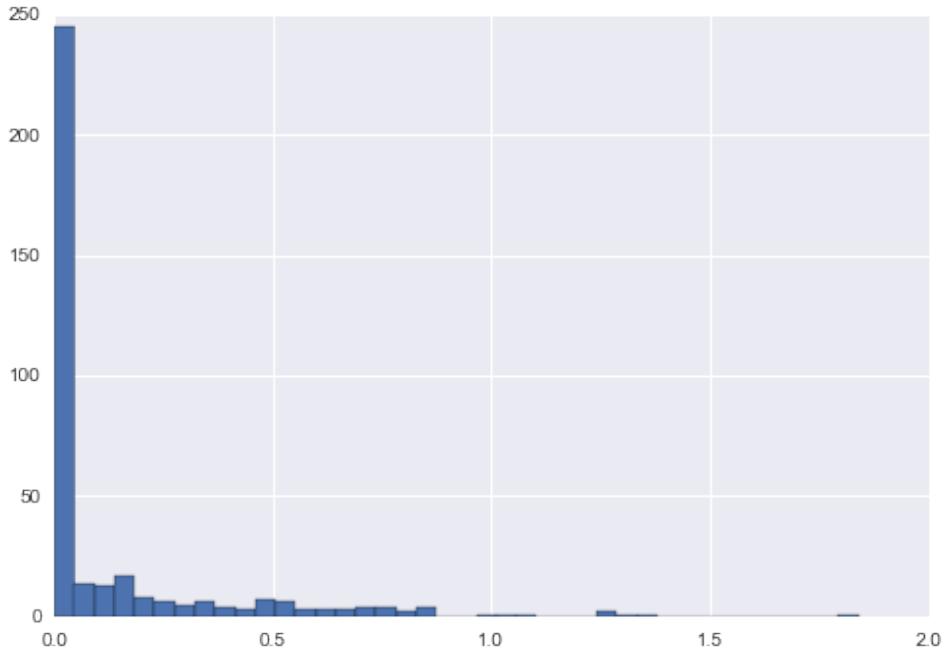
```
import numpy as np  
import pandas as pd  
  
# use Pandas extract rainfall inches as a NumPy array  
rainfall = pd.read_csv('Seattle2014.csv')['PRCP'].values  
inches = rainfall / 254 # 1/10mm -> inches  
inches.shape  
  
(365,)
```

The array contains 365 values, giving daily rainfall in inches from January 1 to December 31, 2014.

As a first quick visualization, Let's look at the histogram of rainy days:

```
# More information on matplotlib can be found in Chapter X.X  
%matplotlib inline
```

```
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # set plot styles
plt.hist(inches, 40);
```



This histogram gives us a general idea of what the data looks like: the vast majority of days in Seattle in 2014 (despite its reputation) saw near zero measured rainfall. But this doesn't do a good job of conveying some information we'd like to see: for example, how many rainy days were there in the year? What is the average precipitation on those rainy days? How many days were there with more than half an inch of rain?

Digging Into the Data

One approach to this would be to answer these questions by hand: loop through the data, incrementing a counter each time we see values in some desired range. For reasons discussed through this chapter, such an approach is very inefficient: both from the standpoint of both time writing code and time computing the result. We saw in section X.X that NumPy's *Universal Functions* can be used in place of loops to do fast elementwise arithmetic operations on arrays; in the same way, we can use other ufuncs to do elementwise *comparisons* over arrays, and we can then manipulate the results to answer the questions we have. We'll leave the data aside for right now, and discuss some general tools in NumPy to use *masking* to quickly answer these types of questions.

Comparison Operators as ufuncs

In section X.X we introduced NumPy's Universal Functions (ufuncs), and focused in particular on arithmetic operators. We saw that using `+`, `-`, `*`, `/`, and others on arrays leads to *elementwise* operations. For example, adding a number to an array adds that number to every element:

```
x = np.array([1, 2, 3, 4, 5])
x + 10
array([11, 12, 13, 14, 15])
```

NumPy also implements comparison operators such as `<` (less than) and `>` (greater than) as elementwise ufuncs. The result of these comparison operators is always an array with a boolean data type. All six of the standard comparison operations are available:

```
x < 3 # less than
array([ True,  True, False, False, False], dtype=bool)
x > 3 # greater than
array([False, False, False,  True,  True], dtype=bool)
x <= 3 # less than or equal
array([ True,  True,  True, False, False], dtype=bool)
x >= 3 # greater than or equal
array([False, False,  True,  True,  True], dtype=bool)
x != 3 # not equal
array([ True,  True, False,  True,  True], dtype=bool)
x == 3 # equal
array([False, False,  True, False, False], dtype=bool)
```

It is also possible to compare two arrays element-by-element, and to include compound expressions:

```
2 * x == x ** 2
array([False,  True, False, False, False], dtype=bool)
```

As in the case of arithmetic operators, the comparison operators are implemented as universal functions in NumPy; for example, when you write `x < 3`, internally NumPy uses `np.less(x, 3)`. A summary of the comparison operators and their equivalent ufunc is shown below:

Operator	Equivalent ufunc
<code>==</code>	<code>np.equal</code>

Operator	Equivalent ufunc
<code>!=</code>	<code>np.not_equal</code>
<code><</code>	<code>np.less</code>
<code><=</code>	<code>np.less_equal</code>
<code>></code>	<code>np.greater</code>
<code>>=</code>	<code>np.greater_equal</code>

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

```

rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
x

array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])

x < 6

array([[ True,  True,  True,  True],
       [False, False,  True,  True],
       [ True,  True, False, False]], dtype=bool)

```

In each case, the result is a boolean array. Working with boolean arrays is straightforward, and there are a few common patterns that we'll mention here:

Working with Boolean Arrays

Given a boolean array, there are a host of useful operations you can do. We'll work with `x`, the two-dimensional array we created above.

```

print(x)

[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]

```

Counting Entries

To count the number of `True` entries in a boolean array, `np.count_nonzero` is useful:

```

# how many values less than six?
np.count_nonzero(x < 6)

```

8

We see that there are eight array entries that are less than six. Another way to get at this information is to use `np.sum`; in this case `False` is interpreted as 0, and `True` is interpreted as 1:

```
np.sum(x < 6)
```

```
8
```

The benefit of `sum()` is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

```
# how many values less than six in each row?  
np.sum(x < 6, 1)  
  
array([4, 2, 2])
```

This counts the number of values less than six in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) `np.any` or `np.all`:

```
# are there any values greater than eight?  
np.any(x > 8)  
  
True  
  
# are there any values less than zero?  
np.any(x < 0)  
  
False  
  
# are all values less than ten?  
np.all(x < 10)  
  
True  
  
# are all values equal to six?  
np.all(x == 6)  
  
False
```

`np.all` and `np.any` can be used along particular axes as well. For example:

```
# are all values in each row less than four?  
np.all(x < 8, axis=1)  
  
array([ True, False,  True], dtype=bool)
```

Here all the elements in the first and third rows are less than eight, while this is not the case for the second row.

Finally, a quick warning: as mentioned in Section X.X, Python has built-in `sum()`, `any()`, and `all()` functions. These have a different syntax than the NumPy versions, and in particular will fail or produce unintended results when used on multi-dimensional arrays. Be sure that you are using `np.sum()`, `np.any()`, and `np.all()` for these examples!

Boolean Operators

Above we saw how to count, say, all days with rain less than four inches, or all days with rain greater than two inches. But what if we want to know about all days with rain less than four inches AND greater than one inch? This is accomplished through Python's *bitwise logic operators*, `&`, `|`, `^`, and `~`, first discussed in Section X.X. Like with the standard arithmetic operators, NumPy overloads these as ufuncs which work element-wise on (usually boolean) arrays.

For example, we can address this sort of compound question this way:

```
np.sum((inches > 0.5) & (inches < 1))
```

29

So we see that there are 29 days with rainfall between 0.5 and 1.0 inches.

Note that the parentheses here are important: because of operator precedence rules, with parentheses removed this expression would be evaluated as

```
inches > (1 & inches) < 4
```

which results in an error.

Using boolean identities, we can answer questions in terms of other boolean operators. Here, we answer the same question in a more convoluted way, using boolean identities:

```
np.sum(~( (inches <= 0.5) | (inches >= 1) ))
```

29

As you can see, combining comparison operators and boolean operators on arrays can lead to a wide range of possible logical operations on arrays.

The following table summarizes the bitwise boolean operators and their equivalent ufuncs:

Operator	Equivalent ufunc
<code>&</code>	<code>np.bitwise_and</code>
<code> </code>	<code>np.bitwise_or</code>
<code>^</code>	<code>np.bitwise_xor</code>
<code>~</code>	<code>np.bitwise_not</code>

Returning to Seattle's Rain

With these tools in mind, we can start to answer the types of questions we have about this data. Here are some examples of results we can compute when combining masking with aggregations:

```

print("Number days without rain:      ", np.sum(inches == 0))
print("Number days with rain:        ", np.sum(inches != 0))
print("Days with more than 0.5 inches:", np.sum(inches > 0.5))
print("Days with 0.2 to 0.5 inches:   ", np.sum((inches > 0.2) & (inches < 0.5)))

Number days without rain:      215
Number days with rain:        150
Days with more than 0.5 inches: 37
Days with 0.2 to 0.5 inches:   36

```

Boolean Arrays as Masks

Above we looked at aggregates computed directly on boolean arrays. A more powerful pattern is to use boolean arrays as masks, to select particular subsets of the data themselves. Returning to our `x` array from above, suppose we want an array of all values in the array which are less than, say, 5.

```

x
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])

```

We can obtain a boolean array for this condition easily, as we saw above:

```

x < 5
array([[False,  True,  True,  True],
       [False, False,  True, False],
       [ True,  True, False, False]], dtype=bool)

```

Now to *select* these values from the array, we can simply index on this boolean array: this is known as a *masking* operation:

```

x[x < 5]
array([0, 3, 3, 3, 2, 4])

```

What is returned is a one-dimensional array filled with all the values which meet this condition; in other words, all the values in positions at which the mask array is `True`. We are then free do operate on these values as we wish. For example, we can compute some relevant statistics on the data:

```

# construct a mask of all rainy days
rainy_days = (inches > 0)

# construct a mask of all summer days (June 21st is the 172nd day)
summer = (np.arange(365) - 172 < 90)

print("Median daily precip on rainy days in 2014 (inches):", np.median(inches[rainy_days]))
print("Median daily precip overall, summer 2014 (inches):", np.median(inches[summer]))
print("Maximum daily precip, summer 2014 (inches):", np.max(inches[summer]))
print("Median precip on all non-summer rainy days (inches):", np.median(inches[rainy_days] & ~summer))

```

```
Median daily precip on rainy days in 2014 (inches): 0.194881889764
Median daily precip overall, summer 2014 (inches): 0.0
Maximum daily precip, summer 2014 (inches): 1.83858267717
Median precip on all non-summer rainy days (inches): 0.224409448819
```

By combining boolean operations, masking operations, and aggregates, we can very quickly answer these sorts of questions for our dataset.

Sidebar: "&" vs. "and"...

One common point of confusion is the difference between the keywords "and" and "or", and the operators & and |. When would you use one versus the other?

The difference is this: "and" and "or" guage to the truth or falsehood of *entire object*, while & and | refer to *portions of each object*.

When you use "and" or "or", it's equivalent to asking Python to treat the object as a single boolean entity. In Python, all nonzero integers will evaluate as True. Thus:

```
bool(42), bool(27)
(True, True)

bool(42 and 27)
True

bool(42 or 27)
True
```

When you use & and | on integers, the expression operates on the bits of the element, applying the *and* or the *or* to the individual bits making up the number:

```
print(bin(42))
print(bin(59))

0b101010
0b111011

print(bin(42 & 59))

0b101010
bin(42 | 59)
'0b111011'
```

Notice that the corresponding bits (right to left) of the binary representation are compared in order to yield the result.

When you have an array of boolean values in NumPy, this can be thought of as a string of bits where 1 = `True` and 0 = `False`, and the result of & and | operates similarly to above:

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B

array([ True,  True,  True, False,  True,  True], dtype=bool)
```

Using `or` on these arrays will try to evaluate the truth or falsehood of the entire array object, which is not a well-defined value:

```
A or B
```

```
-----
```

ValueError	Traceback (most recent call last)
<ipython-input-37-5d8e4f2e21c0> in <module>()	
----> 1 A or B	

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.
```

Similarly, when doing a boolean expression on a given array, you should use `|` or `&` rather than `or` or `and`:

```
x = np.arange(10)
(x > 4) & (x < 8)

array([False, False, False, False, False,  True,  True,  True, False, False], dtype=bool)
```

Trying to evaluate the truth or falsehood of the entire array will give the same `ValueError` we saw above:

```
(x > 4) and (x < 8)
```

```
-----
```

ValueError	Traceback (most recent call last)
<ipython-input-39-3d24f1ffd63d> in <module>()	
----> 1 (x > 4) and (x < 8)	

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.
```

So remember this: "and" and "or" perform a single boolean evaluation on an entire object, while `&` and `|` perform multiple boolean evaluations on the content (the individual bits or bytes) of an object. For boolean NumPy arrays, the latter is nearly always the desired operation.

Fancy Indexing

In the previous section we saw how to access and modify portions of arrays using simple indices (e.g. `arr[0]`), slices (e.g. `arr[:5]`), and boolean masks (e.g. `arr >`

0]). In this section we'll look at another style of array indexing, known as *fancy indexing*. Fancy indexing is like the simple indexing above, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it simply means passing an array of indices to access multiple array elements at once. For example, consider the following array:

```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)

[51 92 14 71 60 20 82 86 74 74]
```

Suppose we want to access three different elements. We could do it like this:

```
[x[3], x[7], x[2]]

[71, 86, 14]
```

Alternatively, we can pass a single list or array of indices to obtain the same result:

```
ind = [3, 7, 4]
x[ind]

array([71, 86, 60])
```

When using fancy indexing, the shape of the result reflects the shape of the *index arrays* rather than the shape of the *array being indexed*:

```
ind = np.array([[3, 7],
               [4, 5]])
x[ind]

array([[71, 86],
       [60, 20]])
```

Fancy indexing also works in multiple dimensions. Consider the following array:

```
X = np.arange(12).reshape((3, 4))
X

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Like with standard indexing, the first index refers to the row, and the second to the column:

```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

```
array([ 2,  5, 11])
```

Notice that the first value in the result is $X[0, 2]$, the second is $X[1, 1]$, and the third is $X[2, 3]$. The pairing of indices in fancy indexing is even more powerful than this: it follows all the broadcasting rules that were mentioned in section X.X. So, for example, if we combine a column vector and a row vector within the indices, we get a two-dimensional result:

```
X[row[:, np.newaxis], col]  
array([[ 2,  1,  3],  
       [ 6,  5,  7],  
       [10,  9, 11]])
```

Here, each row value is matched with each column vector, exactly as we saw in broadcasting of arithmetic operations. For example:

```
row[:, np.newaxis] * col  
array([[0, 0, 0],  
       [2, 1, 3],  
       [4, 2, 6]])
```

It is always important to remember with fancy indexing that the return value reflects the *shape of the indices*, rather than the shape of the array being indexed.

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen.

```
print(X)  
[[ 0  1  2  3]  
 [ 4  5  6  7]  
 [ 8  9 10 11]]
```

We can combine fancy and simple indices:

```
X[2, [2, 0, 1]]  
array([10,  8,  9])
```

We can also combine fancy indexing with slicing:

```
X[1:, [2, 0, 1]]  
array([[ 6,  4,  5],  
       [10,  8,  9]])
```

And we can combine fancy indexing with masking:

```
mask = np.array([1, 0, 1, 0], dtype=bool)  
X[row[:, np.newaxis], mask]
```

```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

All of these indexing options combined lead to a very flexible set of operations for accessing and modifying array values.

Generating Indices: `np.where`

One commonly-seen pattern is to use `np.where` (or the very similar `np.nonzero`) to generate indices to use within fancy indexing. We saw previously that you can use boolean masks to select certain elements of an array. Here, let's create a random array and select all the even elements:

```
X = rand.randint(10, size=(3, 4))
X

array([[7, 4, 3, 7],
       [7, 2, 5, 4],
       [1, 7, 5, 1]])

evens = X[X % 2 == 0]
evens

array([4, 2, 4])
```

Equivalently, you might use the `np.where` function:

```
X[np.where(X % 2 == 0)]
array([4, 2, 4])
```

What does `np.where` do? In this case, we have given it a boolean mask, and it has returned a set of indices:

```
i, j = np.where(X % 2 == 0)
print(i)
print(j)

[0 1 1]
[1 1 3]
```

These indices, like the ones we saw above, are interpreted in pairs: (i.e. the first element is `X[0, 1]`, the second is `X[0, 2]`, etc.) Note here that the computation of these indices is an extra step, and thus using `np.where` in this manner will generally be less efficient than simply using the boolean mask itself. So why might you use `np.where`? Many use it because they have come from a language like IDL or MatLab where such constructions are familiar. But `np.where` can be useful in itself when the indices themselves are of interest, and also has other functionality which you can read about in its documentation.

Example: Selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. By convention, the values of N points in D dimensions are often represented by a 2-dimensional, $N \times D$ array. We'll generate some points from a two-dimensional multivariate normal distribution:

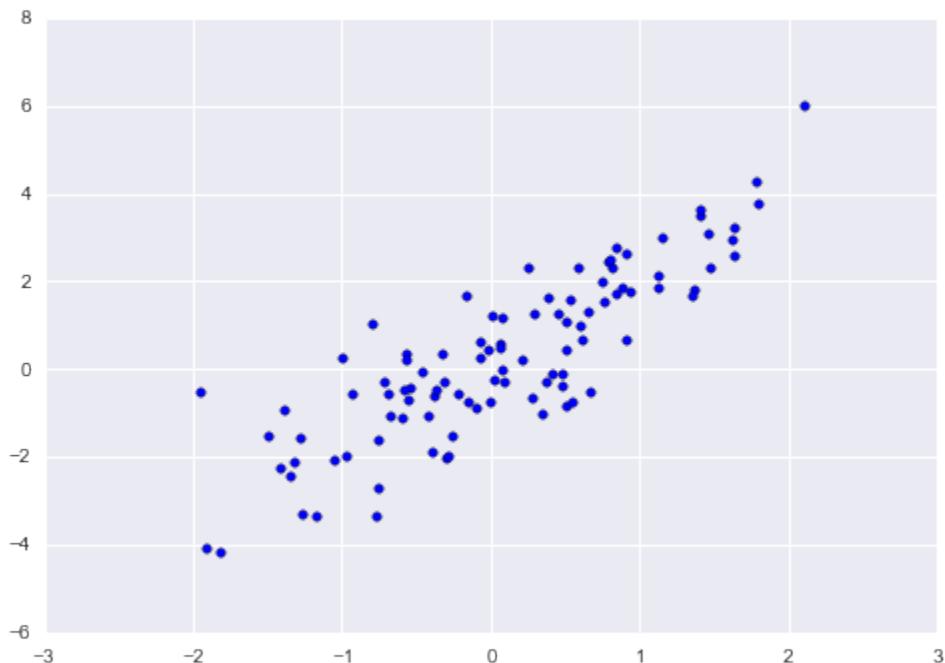
```
mean = [0, 0]
cov = [[1, 2],
       [2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape
```

(100, 2)

Using the plotting tools we will discuss in chapter X.X, we can visualize these points:

```
%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 20 random indices with no repeats, and use these indices to select a portion of the original array:

```
indices = np.random.choice(X.shape[0], 20, replace=False)
selection = X[indices] # fancy indexing here
selection.shape
(20, 2)
```

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

```
plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1],
            facecolor='none', s=200);
```



This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models (see section X.X). We'll see further uses of fancy indexing throughout the book. More information on plotting with matplotlib is available in chapter X.X.

Modifying values with Fancy Indexing

Above we saw how to access parts of an array with fancy indexing. Fancy indexing can also be used to modify parts of an array.

For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value:

```
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:

```
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:

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

[ 6.  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:

```
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x

array([ 6.,  0.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

You might expect that $x[3]$ would contain the value 2, and $x[3]$ 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 $x[i] += 1$ is meant as a short-hand of $x[i] = x[i] + 1$. $x[i] + 1$ is evaluated, and then the result is assigned to the indices in x . With this in mind, it is not the augmentation that happens multiple times, but the assignment, which leads to the rather non-intuitive results.

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

```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)

[ 0.  0.  1.  2.  3.  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 which 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 quickly bin data to create a histogram. For example, imagine we have 1000 values and would like to quickly find where they fall within an array of bins. We could compute it using `ufunc.at` like this:

```
np.random.seed(42)
x = np.random.randn(100)

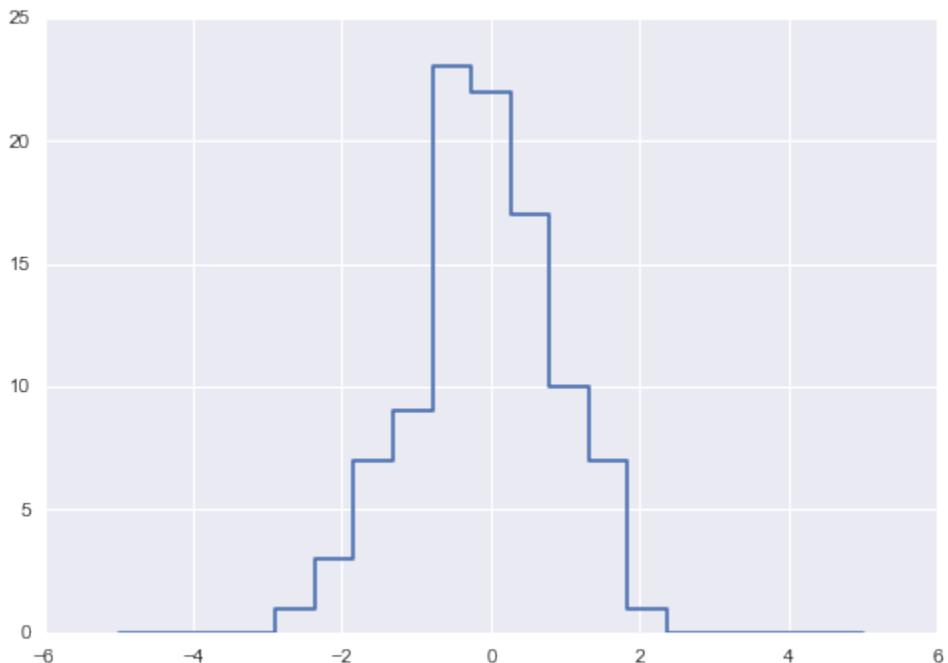
# compute a histogram by-hand
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)

# 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)
```

You might notice that what we've done is to simply compute the contents of a histogram:

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



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');
```



To compute the binning, matplotlib uses the `np.histogram` function, which does a very similar computation to what we did above. Let's compare the two here:

```
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:
The slowest run took 4.70 times longer than the fastest. This could mean that an intermediate resu
10000 loops, best of 3: 69.6 µs per loop
Custom routine:
100000 loops, best of 3: 19.2 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:

```
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: 65.1 ms per loop
Custom routine:
10 loops, best of 3: 129 ms per loop
```

What this comparison shows is that algorithmic efficiency is almost never a simple question. An algorithm efficient for large datasets will not always be the best choice for small datasets, and vice versa (see *big-O notation* in section X.X). But the advantage of coding this algorithm yourself is that with an understanding of these basic methods, you could use these building blocks to extend this to do some very interesting custom behaviors. The key to efficiently using Python in data-intensive applications is knowing about general convenience routines like `np.histogram` and when they're appropriate, but also knowing how to implement your own efficient custom algorithms for times that you need more pointed behavior.

Numpy Indexing Tricks

This section goes over some of NumPy's *indexing tricks*. They are quick and powerful constructs to build certain types of arrays very quickly and easily, but the very terseness which makes them useful can make them difficult to understand. For this reason, even many experienced NumPy users have never used or even seen these; for most use cases, I'd recommend avoiding these in favor of more standard indexing operations. Nevertheless, this is an interesting enough topic that we'll cover it briefly here, if only to give you a means to brazenly confuse and befuddle your collaborators.

All the constructs below come from the module `numpy.lib.index_tricks`. Here is a listing of the tricks we'll go over:

- `numpy.mgrid`: construct dense multi-dimensional mesh grids
- `numpy.ogrid`: construct open multi-dimensional mesh grids
- `numpy.ix_`: construct an open multi-index grid
- `numpy.r_`: translate slice objects to concatenations along rows
- `numpy.c_`: translate slice objects to concatenations along columns

All of these index tricks are marked by the fact that rather than providing an interface through function calls (e.g. `func(x, y)`), they provide an interface through indexing and slicing syntax (e.g. `func[x, y]`). This type of re-purposing of slicing syntax is,

from what I've seen, largely unique in the Python world, and is why some purists would consider these tricks dirty hacks which should be avoided at all costs. Consider yourself warned.

Because these tricks are a bit overly terse and uncommon, we'll also include some recommendations for how to duplicate the behavior of each with more commonly-seen and easy to read NumPy constructions. Some functionality with a similar spirit is provided by the objects `numpy.s_` and `numpy.index_exp`: these are primarily useful as utility routines which convert numpy-style indexing into tuples of Python `slice` objects which can then be manipulated independently. We won't cover these two utilities here: for more information refer to the NumPy documentation.

`np.mgrid`: Convenient Multi-dimensional Mesh Grids

The primary use of `np.mgrid` is the quick creation multi-dimensional grids of values. For example, say you want to visualize the function

$$f(x, y) = \text{sinc}(x^2 + y^2)$$

Where $\text{sinc}(x) = \sin(x)/x$. To plot this, we'll first need to create two two-dimensional grids of values for x and y . Using common NumPy broadcasting constructs, we might do something like this:

```
import numpy as np

# Create the 2D arrays
x = np.zeros((3, 4), int)
y = x.copy()

# Fill the arrays with values
x += np.arange(3)[:, np.newaxis]
y += np.arange(4)

print(x)
print(y)

[[0 0 0 0]
 [1 1 1 1]
 [2 2 2 2]]
[[0 1 2 3]
 [0 1 2 3]
 [0 1 2 3]]
```

With `np.mgrid`, we can do the same thing in one line using a slice syntax

```
x, y = np.mgrid[0:3, 0:4]
print(x)
print(y)

[[0 0 0 0]
 [1 1 1 1]]
```

```
[2 2 2 2]  
[[0 1 2 3]  
 [0 1 2 3]  
 [0 1 2 3]]
```

Notice what this object does: it converts a slice syntax (such as `[0:3]`) into a range syntax (such as `range(0, 3)`) and automatically fills the results in a multi-dimensional array. The `x` value increases along the first axis, while the `y` value increases along the second axis.

Above we used the default step of 1, but we could just as well use a different step size:

```
x, y = np.mgrid[-2:2:0.1, -2:2:0.1]  
print(x.shape, y.shape)  
(40, 40) (40, 40)
```

The third slice value gives the step size for the range: 40 even steps of 0.1 between -2 and 2.

Deeper down the rabbit hole: imaginary steps

`np.mgrid` would be useful enough if it stopped there. But what if you prefer to use `np.linspace` rather than `np.arange`? That is, what if you'd like to specify not the step size, but the *number* of steps between the start and end points? A drawn-out way of doing this might look as follows:

```
# Create the 2D arrays  
x = np.zeros((3, 5), float)  
y = x.copy()  
  
# Fill the arrays with values  
x += np.linspace(-1, 1, 3)[ :, np.newaxis ]  
y += np.linspace(-1, 1, 5)  
  
print(x)  
print(y)  
  
[[-1. -1. -1. -1. -1.]  
 [ 0.  0.  0.  0.  0.]  
 [ 1.  1.  1.  1.  1.]]  
[[-1. -0.5  0.   0.5  1. ]  
 [-1. -0.5  0.   0.5  1. ]  
 [-1. -0.5  0.   0.5  1. ]]
```

`np.mgrid` allows you to specify these `linspace` arguments by passing *imaginary* values to the slice. If the third value is an imaginary number, the integer part of the imaginary component will be interpreted as a number of steps. Recalling that the imaginary values in Python are expressed by appending a `j`, we can write:

```

x, y = np.mgrid[-1:1:3j, -1:1:5j]
print(x)
print(y)

[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.]
 [ 1.  1.  1.  1.]]
[[-1. -0.5  0.  0.5  1.]
 [-1. -0.5  0.  0.5  1.]
 [-1. -0.5  0.  0.5  1.]]

```

Using this now, we can use `np.mgrid` to quickly build-up a grid of values for plotting our 2D function:

```

# grid of 50 steps from -3 to 3
x, y = np.mgrid[-3: 3: 50j,
                  -3: 3: 50j]
f = np.sinc(x ** 2 + y ** 2)

```

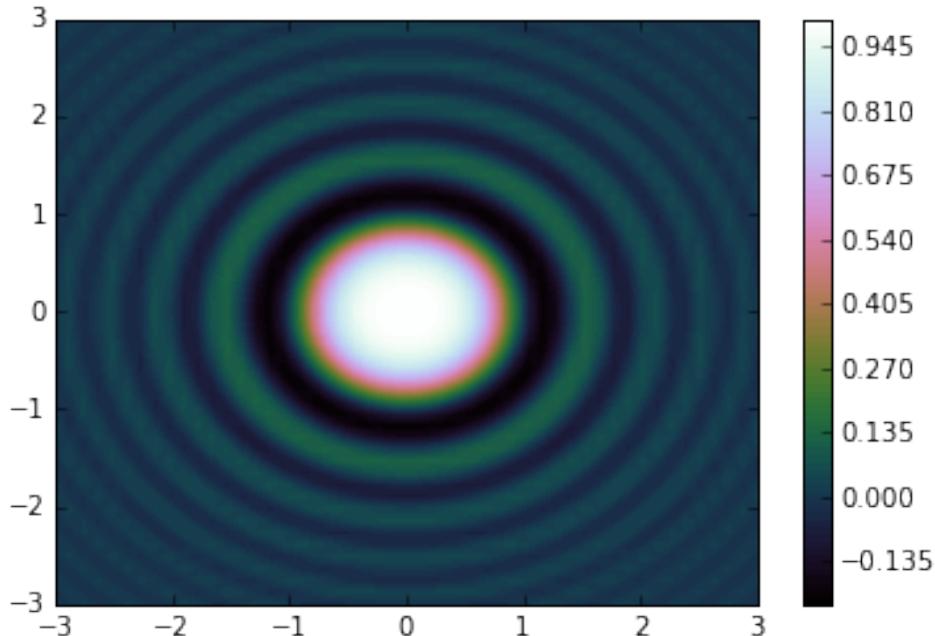
We'll plot this with a contour function; for more information on contour plots, see section X.X:

```

%matplotlib inline
import matplotlib.pyplot as plt

plt.contourf(x, y, f, 100, cmap='cubehelix')
plt.colorbar();

```



The Preferred Alternative: `np.meshgrid`

Because of the non-standard slicing syntax of `np.mgrid`, it should probably not be your go-to method in code that other people will read and use. A more readable alternative is to use the `np.meshgrid` function: though it's a bit less concise, it's much easier for the average reader of your code to understand what's happening.

```
x, y = np.meshgrid(np.linspace(-1, 1, 3),
                    np.linspace(-1, 1, 5), indexing='ij')
print(x)
print(y)

[[[-1. -1. -1. -1. -1.]
 [ 0.  0.  0.  0.]
 [ 1.  1.  1.  1.]]
 [[-1. -0.5  0.  0.5  1.]
 [-1. -0.5  0.  0.5  1.]
 [-1. -0.5  0.  0.5  1.]]]
```

As we see, `np.meshgrid` is a functional interface that takes any two sequences as input. The `indexing` keyword specifies whether the `x` values will increase along columns or rows. This form is much more clear than the rather obscure `np.mgrid` shortcut, and is a good balance between clarity and brevity.

Though we've stuck with two-dimensional examples here, both `mgrid` and `meshgrid` can be used for any number of dimensions: in either case, simply add more range specifiers to the argument list.

`np.ogrid: Convenient Open Grids`

`ogrid` has a very similar syntax to `mgrid`, except it doesn't fill-in the full multi-dimensional array of repeated values. Recall that `mgrid` gives multi-dimensional outputs for each input:

```
x, y = np.mgrid[-1:2, 0:3]
print(x)
print(y)

[[[-1 -1 -1]
 [ 0  0  0]
 [ 1  1  1]]
 [[0 1 2]
 [0 1 2]
 [0 1 2]]]
```

The corresponding function call with `ogrid` returns a simple column and row array:

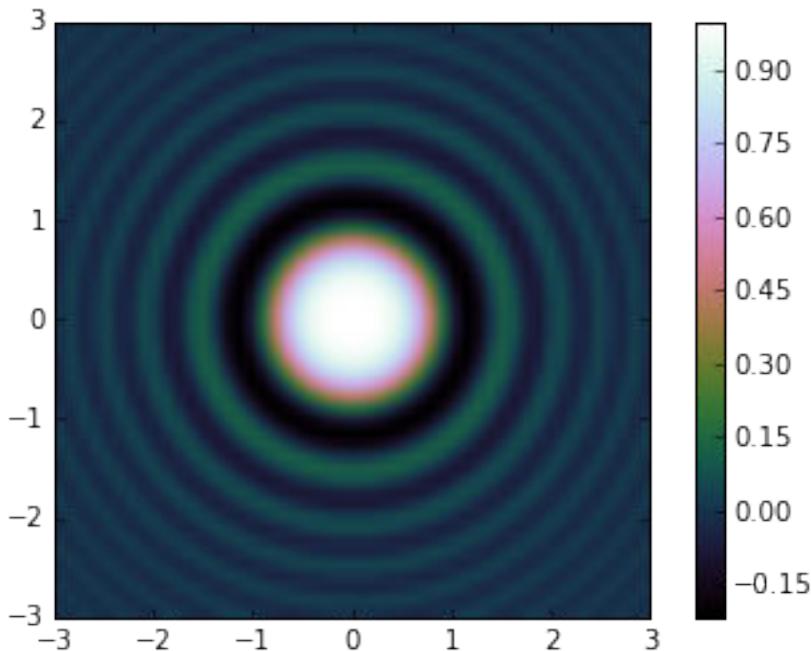
```
x, y = np.ogrid[-1:2, 0:3]
print(x)
print(y)
```

```
[[[-1]
 [ 0]
 [ 1]]
 [[0 1 2]]]
```

This is useful, because often you can use broadcasting tricks (see section X.X) to obviate the need for the full, dense array. Especially for large arrays, this can save a lot of memory overhead within your calculation.

For example, because `plt.imshow` does not require the dense `x` and `y` grids, if we wanted to use it rather than `plt.contourf` to visualize the 2-dimensional sinc function, we could use `np.ogrid`. NumPy broadcasting takes care of the rest:

```
# grid of 50 steps from -3 to 3
x, y = np.ogrid[-3: 3: 50j,
                 -3: 3: 50j]
f = np.sinc(x ** 2 + y ** 2)
plt.imshow(f, cmap='cubehelix',
           extent=[-3, 3, -3, 3])
plt.colorbar();
```



The preferred alternative: Manual reshaping

Like `mgrid`, `ogrid` can be confusing for somebody reading through code. For this type of operation, I prefer using `np.newaxis` to manually reshape input arrays. Compare the following:

```
x, y = np.ogrid[-1:2, 0:3]
print(x)
print(y)

[[-1]
 [ 0]
 [ 1]]
[[0 1 2]]


x = np.arange(-1, 2)[:, np.newaxis]
y = np.arange(0, 3)[np.newaxis, :]
print(x)
print(y)

[[-1]
 [ 0]
 [ 1]]
[[0 1 2]]
```

The average reader of your code is much more likely to have encountered `np.newaxis` than to have encountered `np.ogrid`.

np.ix_: Open Index Grids

`np.ix_` is an index trick which can be used in conjunction with Fancy Indexing (see section X.X). Functionally, it is very similar to `np.ogrid` in that it turns inputs into a multi-dimensional open grid, but rather than generating sequences of numbers based on the inputs, it simply reshapes the inputs:

```
i, j = np.ix_([0, 1], [2, 4, 3])
print(i)
print(j)

[[0]
 [1]
 [[2 4 3]]]
```

The result of `np.ix_` is most often used directly as a fancy index. For example:

```
M = np.arange(16).reshape((2, 8))
print(M)

[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]]

M[np.ix_([0, 1], [2, 4, 3])]

array([[ 2,  4,  3],
       [10, 12, 11]])
```

Notice what this did: it created a new array with rows specified by the first argument, and columns specified by the second. Like `mgrid` and `ogrid`, `ix_` can be used in any number of dimensions. Often, however, it can be cleaner to specify the indices directly. For example, to find the equivalent of the above result, we can alternatively mix slicing and fancy indexing to write

```
M[:, [2, 4, 3]]  
array([[ 2,  4,  3],  
       [10, 12, 11]])
```

For more complicated operations, we might instead follow the strategy above under `np.ogrid` and use a solution based on `np.newaxis`.

np.r_: concatenation along rows

`np.r_` is an index trick which allows concise concatenations of arrays. For example, imagine that we want to create an array of numbers which counts up to a value and then back down. We might use `np.concatenate` as follows:

```
np.concatenate([range(5), range(5, -1, -1)])  
array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0])
```

`np.r_` allows us to do this concisely using index notations similar to those in `np.mgrid` and `np.ogrid`:

```
np.r_[:5, 5:-1:-1]  
array([0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0])
```

Furthermore, slice notation (with the somewhat confusing imaginary steps discussed above) can be mixed with arrays and lists to create even more flexible concatenations:

```
np.r_[0:1:3j, 3*[5], range(4)]  
array([ 0. ,  0.5,  1. ,  5. ,  5. ,  5. ,  0. ,  1. ,  2. ,  3. ])
```

To make things even more confusing, if the first index argument is a string, it specifies the axis of concatenation. For example, for a two-dimensional array, the string “0” or “1” will indicate whether to stack horizontally or vertically:

```
x = np.array([[0, 1, 2],  
              [3, 4, 5]])  
np.r_["0", x, x]  
  
array([[0, 1, 2],  
       [3, 4, 5],  
       [0, 1, 2],  
       [3, 4, 5]])  
  
np.r_["1", x, x]
```

```
array([[0, 1, 2, 0, 1, 2],  
       [3, 4, 5, 3, 4, 5]])
```

Even more complicated options are available for the initial string argument. For example, we can turn one-dimensional arrays into two-dimensional arrays by putting another argument within the string:

```
np.r_[ "0,2", :3, 1:2:3j, [3, 4, 3]]  
  
array([[ 0. ,  1. ,  2. ],  
       [ 1. ,  1.5,  2. ],  
       [ 3. ,  4. ,  3. ]])
```

Roughly, this string argument says “make sure the arrays are two-dimensional, and then concatenate them along axis zero”.

Even more complicated axis specifications are available: you can refer to the documentation of `np.r_` for more information.

The Preferred Alternative: concatenation

As you might notice, `np.r_` can quickly become very difficult to parse. For this reason, it’s probably better to use `np.concatenate` for general concatenation, or `np.vstack`/`np.hstack`/`np.dstack` for specific cases of multi-dimensional concatenation. For example, the above expression could also be written with a few more keystrokes in a much clearer way using vertical stacking:

```
np.vstack([range(3),  
          np.linspace(1, 2, 3),  
          [3, 4, 3]])  
  
array([[ 0. ,  1. ,  2. ],  
       [ 1. ,  1.5,  2. ],  
       [ 3. ,  4. ,  3. ]])
```

np.c_: concatenation along columns

In case `np.r_` was not concise enough for you, you can also use `np.c_`. The expression `np.c_*` is simply shorthand for `np.r_['-1, 2, 0', *]`, where `*` can be replaced with any list of objects. The result is that one-dimensional arguments are stacked horizontally as columns of the two-dimensional result:

```
np.c_[:3, 1:4:3j, [1, 1, 1]]  
  
array([[ 0. ,  1. ,  1. ],  
       [ 1. ,  2.5,  1. ],  
       [ 2. ,  4. ,  1. ]])
```

This is useful because stacking vectors in columns is a common operation. As with `np.r_` above, this sort of operation can be more clearly expressed using some form of concatenation or stacking, along with a transpose:

```
np.vstack([np.arange(3),
           np.linspace(1, 4, 3),
           [1, 1, 1]]).transpose()

array([[ 0. ,  1. ,  1. ],
       [ 1. ,  2.5,  1. ],
       [ 2. ,  4. ,  1. ]])
```

Why Index Tricks?

In all the cases above, we've seen that the index trick functionality is extremely concise, but this comes along with potential confusion for anyone reading the code. Throughout we've recommended avoiding these and using slightly more verbose (and much more clear) alternatives. The reason we even took the time to cover these is that they *do* sometimes appear in the wild, and it's good to know that they exist – if only so that you can properly understand them and convert them to more readable code.

Sorting Arrays

This section covers algorithms related to sorting NumPy arrays. These algorithms are a favorite topic in introductory computer science 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:

```
import numpy as np

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

L = np.array([2, 1, 4, 3, 5])
selection_sort(L)

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

As any first-year Computer Science major will tell you, the selection sort is useful for its simplicity, but is much too slow to be used in practice. The reason is that as lists get big, it does not scale well. For a list of N values, it requires N loops, each of which does $\propto N$ comparisons to find the swap value. In terms of the “big-O” notation often used to characterize these algorithms, selection sort averages $\mathcal{O}[N^2]$: if you double the number of items in the list, the execution time will go up by about a factor of four.

Even selection sort, though, is much better than my all-time favorite sorting algorithms, the *bogosort*:

```
def bogosort(L):
    while np.any(L[:-1] > L[1:]):
        np.random.shuffle(L)
    return L

L = np.array([2, 1, 4, 3, 5])
bogosort(L)

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

This silly sorting method relies on pure chance: it shuffles the array repeatedly until the result happens to be sorted. With an average scaling of $\mathcal{O}[N \times N!]$, this should (quite obviously) never be used.

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

Sidebar: Big-O Notation

Big-O notation is a means of describing how the number of operations required for an algorithm scales as the size of the input grows. To use it correctly is to dive deeply into the realm of computer science theory, and to carefully distinguish it from the related small-o notation, big- θ notation, big- Ω notation, and probably many mutant hybrids thereof. These distinctions add precision to statements about algorithmic scaling. Outside computer science theory exams and the remarks of pedantic blog commenters, though, you'll rarely see such distinctions made in practice. Far more common in the data science world is a less rigid use of big-O notation: as a general (if imprecise) description of the scaling of an algorithm. With apologies to theorists and pedants, this is the interpretation we'll use throughout this book.

Big-O notation, in this loose sense, tells you how much time your algorithm will take as you increase the amount of data. If you have an $\mathcal{O}[N]$ (read “order N ”) algorithm which takes one second to operate on a list of length $N=1000$, then you should expect it to take about 5 seconds for a list of length $N=5000$. If you have an $\mathcal{O}[N^2]$ (read “order N squared”) algorithm which takes 1 second for $N=1000$, then you should expect it to take about 25 seconds for $N=5000$.

For our purposes, the N will usually indicate some aspect of the size of the data set: how many distinct objects we are looking at, how many features each object has, etc. When trying to analyze billions or trillions of samples, the difference between $\mathcal{O}[N]$ and $\mathcal{O}[N^2]$ can be far from trivial!

Notice that the big-O notation by itself tells you nothing about the actual wall-clock time of a computation, but only about its scaling as you change N . Generally, for example, an $\mathcal{O}[N]$ algorithm is considered to have better scaling than an $\mathcal{O}[N^2]$ algorithm, and for good reason. But for small datasets in particular the algorithm with better scaling might not be faster! For a particular problem, an $\mathcal{O}[N^2]$ algorithm might take 0.01sec, while a “better” $\mathcal{O}[N]$ algorithm might take 1 sec. Scale up N by a factor of 1000, though, and the $\mathcal{O}[N]$ algorithm will win out.

Even this loose version of Big-O notation can be very useful when comparing the performance of algorithms, and we’ll use this notation throughout the book when talking about how algorithms scale.

Fast Sorts in Python

Python has a `list.sort` function and a `sorted` function, both of which use the *Tim-sort* algorithm (named after its creator, Tim Peters) which has average and worst-case complexity $\mathcal{O}[N \log N]$. Python’s `sorted()` function will return a sorted version of any iterable without modifying the original:

```
L = [2, 1, 4, 3, 5]
sorted(L)
[1, 2, 3, 4, 5]
print(L)
[2, 1, 4, 3, 5]
```

While the `list.sort` method works sorts a list in place:

```
L.sort()
print(L)
[1, 2, 3, 4, 5]
```

There are additional arguments to each of these sorting routines which allow you to customize how items are compared; for more information, refer to the Python documentation.

Fast Sorts in NumPy: `np.sort` and `np.argsort`

Just as NumPy has a `np.sum` which is faster on arrays than Python’s built-in `sum`, NumPy also has a `np.sort` which is faster on arrays than Python’s built-in `sorted` function. NumPy’s version uses the *quicksort* algorithm by default, though you can specify whether you’d like to use *mergesort* or *heapsort* instead. All three are $\mathcal{O}[N \log N]$, and each has advantages and disadvantages that you can read about elsewhere. For most applications, the default quicksort is more than sufficient.

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

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

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

If you prefer to sort the array in-place, you can instead use the `array.sort` method:

```
x.sort()
print(x)

[1 2 3 4 5]
```

A related function is `argsort`, which instead of returning a sorted list returns the list of indices in sorted order:

```
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, etc. These indices can then be used to construct the sorted array if desired:

```
x[i]
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 multi-dimensional array using the `axis` argument. With it, we can sort along rows or columns of a two-dimensional array:

```
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]]

# sort each column of X
np.sort(X, axis=0)

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]])
```

```
# sort each row of X
np.sort(X, axis=1)

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 don't care about sorting the entire array, but simply care about finding the N 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 smallest K values to the left of the partition, and the remaining values to the right, in arbitrary order:

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

array([2, 1, 3, 4, 6, 5, 7])
```

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

Similarly to sorting, we can partition along an arbitrary axis of a multi-dimensional array:

```
np.partition(X, 2, axis=1)

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 values from that row, with the remaining values filling the remaining slots.

Finally, just as there is a `np.argsort` which computes indices of the sort, there is a `np.argpartition` which computes indices of the partition. We'll see this in action below.

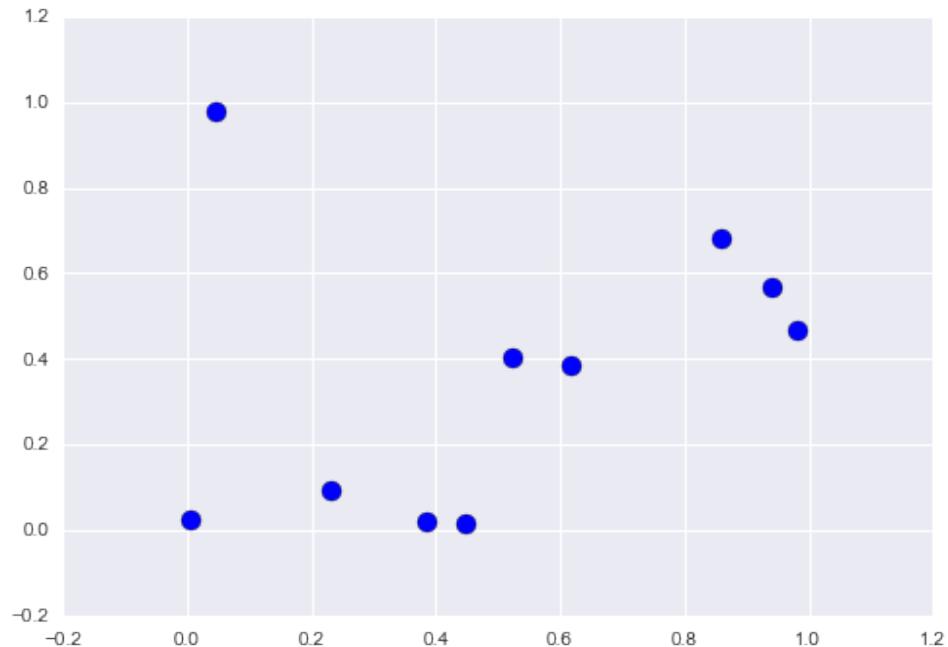
Example: K Nearest Neighbors

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 ten points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10×2 array:

```
X = rand.rand(10, 2)
```

To get an idea of how these points look, let's quickly scatter-plot them, using tools explored in chapter X.X:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # Plot styling
plt.scatter(X[:, 0], X[:, 1], s=100);
```



Now we'll compute the distance between each pair of points. Recall that the distance $D_{x,y}$ between a point x and a point y in d dimensions satisfies

$$D_{x,y}^2 = \sum_{i=1}^d (x_i - y_i)^2$$

We can keep in mind that sorting according to D^2 is equivalent to sorting according to D . Using the broadcasting rules covered in section X.X along with the aggregation routines from section X.X, we can compute the matrix of distances in a single line of code:

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

The above operation has a lot packed into it, and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules. When you come across code like this, it can be useful to mentally break it down into steps:

```

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

(10, 10, 2)

# square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape

(10, 10, 2)

# sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape

(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:

```

dist_sq.diagonal()

array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

```

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

```

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 zero through nine in order: this is due to the fact that each point's closest neighbor is itself! But by using a full sort, 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 three squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the `np.argpartition` function:

```

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 with lines representing the connections from each point to its two nearest neighbors:

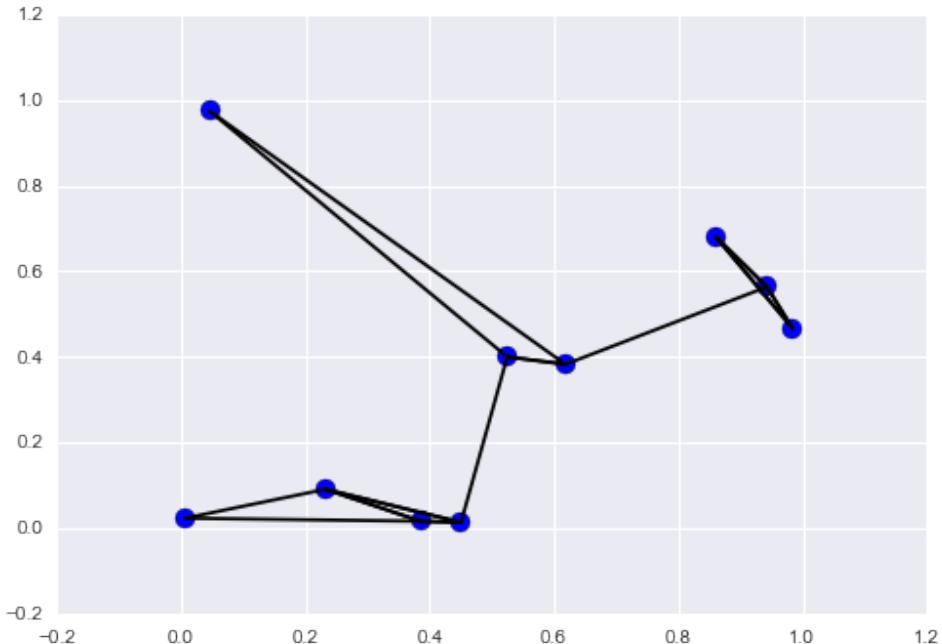
```

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

# draw lines from each point to its K 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')

```



Each point in the plot has lines drawn to its two nearest neighbors. At first glance, it might seem strange that some of the points have more than two lines coming out of them: this is due to the fact that if point A is one of the two nearest neighbors of point B, this does not necessarily imply that point B is one of the two nearest neighbors of point A.

Though the broadcasting and row-wise sorting of the above approach might seem less straightforward than writing a loop, it turns out to be a very efficient way of operating on this data in Python. You might be tempted to do the same type of operation by manually looping through the data and sorting each set of neighbors individually, but this would almost certainly lead to a slower algorithm than the vectorized version we used above. The beauty of the above approach is that it's written in a way that's agnostic to the size of the input data: we could just as easily compute the neighbors

among 100 or 1,000,000 points in any number of dimensions, and the code would look the same.

Finally, we should note that when doing very large nearest neighbor searches, there are tree-based algorithms or approximate algorithms that can scale as $\mathcal{O}[N \log N]$ or better rather than the $\mathcal{O}[N^2]$ of the brute-force algorithm above. For more information on these fast tree-based K-neighbors queries, see section X.X.

Searching and Counting Values In Arrays

This section covers ways of finding a particular value or particular values in an array of data. This may sound like it simply requires a linear scan of the data, but when you know the data is sorted, or if you're trying to find multiple values at the same time, there are faster ways to go about it.

Python Standard Library Tools

Because this is so important, the Python standard library has a couple solutions you should be aware of. Here we'll quickly go over the functions and methods Python contains for doing searches in unsorted and sorted lists, before continuing on to the more specialized tools in NumPy.

Unsorted Lists

For any arbitrary list of data, the only way to find whether an item is in the list is to scan through it. Rather than making you write this loop yourself, Python provides the `list.index` method, which scans the list looking for the first occurrence of a value, and returning its index:

```
L = [5, 2, 6, 1, 3, 6]
L.index(6)
2
```

Sorted Lists

If your list is sorted, there is a more sophisticated approach based on recursively bisecting the list, and narrowing-in on the half which contains the desired value. Python implements this $\mathcal{O}[N \log N]$ in the built-in `bisect` package:

```
import bisect
L = [2, 4, 5, 5, 7, 9]
bisect.bisect_left(L, 7)
4
```

Technically, `bisect` is not searching for the value itself, but for the *insertion index*: that is, the index at which the value should be inserted in order to keep the list sorted:

```
bisect.bisect(L, 4.5)
2
L.insert(2, 4.5)
L == sorted(L)
True
```

If your goal is to insert items into the list, the `bisect.insort` function is a bit more efficient. For more details on `bisect` and tools therein, use IPython's help features or refer to Python's online documentation.

Searching for Values in NumPy Arrays

NumPy has some similar tools and patterns which work for quickly locating values in arrays, but their use is a bit different than the Python standard library approaches. The patterns listed below have the benefit of operating much more quickly on NumPy arrays, and of scaling well as the size of the array grows.

Unsorted Arrays

For finding values in unsorted data, NumPy has no strict equivalent of `list.index`. Instead, it is typical to use a pattern based on masking and the `np.where` function (see section X.X)

```
import numpy as np
A = np.array([5, 2, 6, 1, 3, 6])
np.where(A == 6)
(array([2, 5]),)
```

`np.where` always returns a *tuple* of index arrays; even in the case of a one-dimensional array you should remember that the output is a one-dimensional tuple. To isolate the first index at which the value is found, then, you must use `[0]` to access the first item of the tuple, and `[0]` again to access the first item of the array of indices. This gives the equivalent result to `list.index`:

```
list(A).index(6) == np.where(A == 6)[0][0]
True
```

Note that this masking approach solves a different use-case than `list.index`: rather than finding the first occurrence of the value and stopping, it finds all occurrences of the value simultaneously. If `np.where` is a bottleneck and your code requires quickly finding the first occurrence of a value in an unsorted list, it will require writing a simple utility with Cython, Numba, or a similar tool; see chapter X.X.

Sorted Arrays

If your array is sorted, NumPy provides a fast equivalent of Python's `bisect` utilities with the `np.searchsorted` function:

```
A = np.array([2, 4, 5, 5, 7, 9])
np.searchsorted(A, 7)

4
```

Like `bisect`, `np.searchsorted` returns an *insertion index* for the given value:

```
np.searchsorted(A, 4.5)

2
```

Unlike `bisect`, it can search for insertion indices for multiple values in one call, without the need for an explicit loop:

```
np.searchsorted(A, [7, 4.5, 5, 10, 0])
array([4, 2, 2, 6, 0])
```

The `searchsorted` function has a few other options, which you can read about in the functions docstring.

Counting and Binning

A related set of functionality in NumPy is the built-in tools for counting and binning of values. For example, to count the occurrences of a value or other condition in an array, use `np.count_nonzero`:

```
A = np.array([2, 0, 2, 3, 4, 3, 4, 3])
np.count_nonzero(A == 4)

2
```

np.unique for counting

For counting occurrences of all values at once, you can use the `np.unique` function, which in NumPy versions 1.9 or later has a `return_count` option:

```
np.unique(A, return_counts=True)
(array([0, 2, 3, 4]), array([1, 2, 3, 2]))
```

The first return value is the list of unique values in the array, and the second return value is the list of associated counts for those values.

np.bincount. If your data consist of positive integers, a more compact way to get this information is with the `np.bincount` function:

```
np.bincount(A)
```

```
array([1, 0, 2, 3, 2])
```

If `counts` is the output of `np.bincount(A)` then `counts[val]` is the number of times value `val` occurs in the array `A`.

np.histogram. `np.bincount` can become cumbersome when your values are large, and it does not apply when your values are not integers. For this more general case, you can specify bins for your values with the `np.histogram` function:

```
counts, bins = np.histogram(A, bins=range(6))
print("bins:          ", bins)
print("counts in bin: ", counts)

bins:          [0 1 2 3 4 5]
counts in bin: [1 0 2 3 2]
```

Here the output is formatted to make clear that the bins give the *boundaries* of the range, and the counts indicate how many values fall within each of those ranges. For this reason, the counts array will have one fewer entry than the bins array. A related function to be aware of is `np.digitize`, which quickly computes index of the appropriate bin for a series of values.

Structured Data: NumPy's Structured Arrays

While often our data can be well-represented by a homogeneous array of values, sometimes this is not the case. This section demonstrates the use of NumPy's *structured arrays* and *record arrays*, which provide efficient storage for compound, heterogeneous data. While the patterns shown here are useful for simple operations, scenarios like this often lend themselves to the use of Pandas Dataframes, which we'll explore in the next chapter.

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a single structure to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

Recall that previously we created a simple array using an expression like this:

```
x = np.zeros(4, dtype=int)
```

We can similarly create a structured array using a compound data type specification:

```

# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                        'formats':('U10', 'i4', 'f8')})
print(data.dtype)

[('name', '<U10'), ('age', 'i4'), ('weight', '<f8')]

```

Here 'U10' translates to “unicode string of maximum length 10”, 'i4' translates to “4-byte (i.e. 32 bit) integer” and 'f8' translates to “8-byte (i.e. 64 bit) float”. We’ll discuss other options for these type codes below.

Now that we’ve created an empty container array, we can fill the array with our lists of values:

```

data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)

[('Alice', 25, 55.0) ('Bob', 45, 85.5) ('Cathy', 37, 68.0)
 ('Doug', 19, 61.5)]

```

As we had hoped, the data is now arranged together in one convenient block of memory.

The handy thing with structured arrays is that you can now refer to values either by index or by name:

```

# Get all names
data['name']

array(['Alice', 'Bob', 'Cathy', 'Doug'],
      dtype='<U10')

# Get first row of data
data[0]

('Alice', 25, 55.0)

# Get the name from the last row
data[-1]['name']

'Doug'

```

Using boolean masking, this even allows you to do some more sophisticated operations such as filtering on age:

```

# Get names where age is under 30
data[data['age'] < 30]['name']

array(['Alice', 'Doug'],
      dtype='<U10')

```

Note that if you’d like to do any operations which are much more complicated than these, you should probably consider the Pandas package, covered in the next section. Pandas provides a Dataframe object, which is a structure built on NumPy arrays that

offers a variety of useful data manipulation functionality similar to what we've shown above, as well as much, much more.

Creating Structured Arrays

Structured array data types can be specified in a number of ways. Above, we saw the dictionary method:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':('U10', 'i4', 'f8')})

dtype([('name', 'U10'), ('age', 'i4'), ('weight', 'f8')])
```

For clarity, numerical types can be specified using Python types or NumPy dtypes instead:

```
np.dtype({'names':('name', 'age', 'weight'),
          'formats':((np.str_, 10), int, np.float32)})

dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

A compound type can also be specified as a list of tuples:

```
np.dtype([(('name', 'S10'), ('age', 'i4'), ('weight', 'f8'))])

dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

If the names of the types do not matter to you, you can specify the types alone in a comma-separated string:

```
np.dtype('S10,i4,f8')

dtype([('f0', 'S10'), ('f1', 'i4'), ('f2', 'f8')])
```

The shortened string format codes may seem confusing, but they are built on simple principles. The first (optional) character is < or >, which means “little endian” or “big endian” respectively, and specifies the ordering convention for significant bits. The next character specifies the type of data: characters, bytes, ints, floating points, etc. (see the table below). The last character or characters represents the size of the object in bytes.

character	description	example
'b'	byte	np.dtype('b')
'i'	(signed) integer	np.dtype('i4') == np.int32
'u'	unsigned integer	np.dtype('u1') == np.uint8
'f'	floating point	np.dtype('f8') == np.int64
'c'	complex floating point	np.dtype('c16') == np.complex128
'S', 'a'	string	np.dtype('S5')
'U'	unicode string	np.dtype('U') == np.str_

character	description	example
'V'	raw data (void)	<code>np.dtype('V') == np.void</code>

More Advanced Compound Types

It is possible to define even more advanced compound types. For example, you can create a type where each element contains an array or matrix of values. Here, we'll create a data type with a `mat` component consisting of a 3x3 floating point matrix:

```
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])

(0, [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
```

Now each element in the `X` array consists of an id and a 3x3 matrix. Why would you use this rather than a simple multi-dimensional array, or perhaps a Python dictionary? The reason is that this numpy dtype directly maps onto a C structure definition, so the buffer containing the array content can be accessed directly within an appropriately written C program. We'll see examples of this type of pattern in section X.X, when we discuss Cython, a C-enabled extension of the Python language.

RecordArrays: Structured Arrays with a Twist

NumPy also provides the `np.recarray` class, which is almost identical to the structured arrays described above with one additional feature: fields can be accessed as attributes rather than as dictionary keys. Recall that we previously accessed the ages by writing:

```
data['age']
array([25, 45, 37, 19], dtype=int32)
```

If we view our data as a record array instead, we can access this with slightly fewer keystrokes:

```
data_rec = data.view(np.recarray)
data_rec.age
array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

```
%timeit data['age']
%timeit data_rec['age']
%timeit data_rec.age
```

```
1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

Whether the more convenient notation is worth the additional overhead will depend on your own application.

On to Pandas

This section on structured and record arrays is purposely at the end of the NumPy section, because it leads so well into the next chapter: Pandas. Structured arrays like the ones above are good to know about for certain situations, especially in case you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language. For day-to-day use of structured data, the Pandas package is a much better choice. We'll take a look at that package next.

Introduction to Pandas

In the last section we dove into detail on NumPy and its `ndarray` object, which provides efficient storage and manipulation of dense typed arrays in Python. Here we'll build on this knowledge by looking in detail at the data structures provided by the Pandas library. Pandas is a newer package built on top of NumPy, and provides an efficient implementation of a *Data Frame*. Data frames are essentially multi-dimensional 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 flexibility (such as attaching labels to data, working with missing data, etc.) and when attempting operations which do not map well to element-wise broadcasting (such as groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available 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 access to these sorts of "data munging" tasks that occupy most of a data scientist's time.

In this chapter, we will focus on the mechanics of using the `Series`, `DataFrame`, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus. Like the previous chapter on NumPy, the primary purpose of this chapter is to serve as a comprehensive introduction to the Python tools that will enable the more in-depth analyses and discussions of the second half of the book.

Installing and Using Pandas

Installation of Pandas on your system requires a previous install of NumPy, as well as 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: <http://pandas.pydata.org/>. If you followed the advice in the introduction and used the Anaconda stack, you will already have Pandas installed.

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

```
import pandas  
pandas.__version__  
'0.16.1'
```

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

```
import pandas as pd
```

This import convention will be used throughout the remainder of this book.

Reminder about Built-in Documentation

As you read through this chapter, don't forget that IPython gives you the ability to quickly explore the contents of a package (by using the tab-completion feature), as well as the documentation of various functions (using the "?" character).

For example, you can type

```
In [3]: pd.<TAB>
```

to display all the contents of the pandas namespace, and

```
In [4]: pd?
```

to display Pandas's built-in documentation. More detailed documentation, along with tutorials and other resources, can be found at <http://pandas.pydata.org/>.

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 through the rest of this chapter, Pandas provides a host of useful tools, methods, and functionality on top of these data structures, but nearly everything that follows will require an understanding of what these structures are. This first section will cover the three fundamental Pandas data structures: the Series, DataFrame, and Index.

Just as the standard alias for importing numpy is np, the standard alias for importing pandas is pd:

```
import numpy as np
import pandas as pd
```

Pandas Series

A pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

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

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

As we see in the output above, the series has 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:

```
data.values
array([ 0.25,  0.5 ,  0.75,  1. ])
```

while the `index` is an array-like object of type `pd.Index`, which we'll discuss in more detail below.

```
data.index
Int64Index([0, 1, 2, 3], dtype='int64')
```

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

```
data[1]
0.5
data[1:3]
1    0.50
2    0.75
dtype: float64
```

As we will see, though, the Pandas series is 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 interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an *implicitly defined* integer index used to 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:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                 index=['a', 'b', 'c', 'd'])  
  
data  
  
a    0.25  
b    0.50  
c    0.75  
d    1.00  
dtype: float64
```

and the item access works as expected:

```
data['b']  
  
0.5
```

We can even use non-contiguous or non-sequential indices

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],  
                 index=[2, 5, 3, 7])  
  
data  
  
2    0.25  
5    0.50  
3    0.75  
7    1.00  
dtype: float64  
  
data[5]  
  
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 which maps arbitrary keys to a set of arbitrary values, and a series is a structure which maps *typed* keys to a set of *typed* values. This typing 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 certain operations.

The series-as-dict analogy can be made even more clear by constructing a `Series` object directly from a Python dictionary:

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

California    38332521
Florida       19552860
Illinois      12882135
New York      19651127
Texas         26448193
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:

```
population['California']
38332521
```

Unlike a dictionary, though, the Series also supports array-style operations such as slicing:

```
population['California':'Illinois']

California    38332521
Florida       19552860
Illinois      12882135
dtype: int64
```

We'll discuss some of the quirks of Pandas indexing and slicing in Section X.X.

Constructing Series Objects

We've already seen a few ways of constructing a Pandas `Series` from scratch; all of them are some version of the following,

```
>>> pd.Series(data, index=index)
```

where `index` is an optional argument, and `data` can be one of many entities.

For example, `data` can be a list or NumPy array, in which case `index` defaults to an integer sequence:

```
pd.Series([2, 4, 6])

0    2
1    4
2    6
dtype: int64
```

`data` can be a scalar, which is broadcast to fill the specified index:

```
pd.Series(5, index=[100, 200, 300])  
100    5  
200    5  
300    5  
dtype: int64
```

`data` can be a dictionary, in which `index` defaults to the sorted dictionary keys:

```
pd.Series({2:'a', 1:'b', 3:'c'})  
1    b  
2    a  
3    c  
dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])  
3    c  
2    a  
dtype: object
```

Notice that here we explicitly identified the particular indices to be included from the dictionary.

Pandas DataFrame

The next fundamental structure in Pandas is the `DataFrame`. Like the `Series` above, the `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll discuss these views below.

DataFrame as a Generalized NumPy array

If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is 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 the same index.

To demonstrate this, let's first construct a new `Series` listing the area of each of the five states mentioned above:

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297, 'Florida': 170312, 'Illino...  
area = pd.Series(area_dict)  
area  
California    423967  
Florida        170312
```

```
Illinois      149995  
New York     141297  
Texas        695662  
dtype: int64
```

Now that we have this along with the population Series from above, we can use a dictionary to construct a single two-dimensional object containing this information:

```
states = pd.DataFrame({'population': population,  
                      'area': area})  
states
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Like the Series object, the DataFrame has an `index` attribute which gives access to the index labels:

```
states.index  
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

Additionally, the DataFrame has a `columns` attribute which is an Index object holding the column labels:

```
states.columns  
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 accessing the data.

DataFrame as Specialized Dictionary

Similarly, we can think of a dataframe as a specialization of a dictionary. Where a dictionary maps a key to a value, a data frame maps a column name to a Series of column data. For example, asking for the 'area' attribute returns the Series object containing the areas we saw above:

```
states['area']  
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 generalized dictionaries rather than generalized arrays, though both ways of looking at the situation can be useful. We'll explore more flexible means of indexing DataFrames in Section X.X.

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, and a single-column DataFrame can be constructed from a single series:

```
pd.DataFrame(population, columns=['population'])
```

	population
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:

```
data = [{‘a’: i, ‘b’: 2 * i}  
        for i in range(3)]  
pd.DataFrame(data)
```

	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:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

	a	b	c
0	1	2	NaN
1	NaN	3	4

From a dictionary of Series objects. We saw this above, but a DataFrame can be constructed from a dictionary of Series objects works as well:

```
pd.DataFrame({'population': population,
              'area': area})
```

	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 dataframe with any specified column and index names. If left out, an integer index will be used for each.

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

	foo	bar
a	0.201512	0.332724
b	0.905656	0.281871
c	0.455450	0.530707

From a numpy structured array. We covered structured arrays in section X.X. A Pandas dataframe operates much like a structured array, and can be created directly from one:

```
A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
A
array([(0, 0.0), (0, 0.0), (0, 0.0)],
      dtype=[('A', '<i8'), ('B', '<f8')])

pd.DataFrame(A)
```

	A	B
0	0	0
1	0	0
2	0	0

Pandas Index

Above we saw that both the `Series` and `DataFrame` contain an explicit `index` which 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*. Those views have some interesting consequences in the operations available on `Index` objects. As a simple example, let's construct an index from a list of integers:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```



```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as Immutable Array

The index in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
ind[1]  
3  
ind[::-2]  
Int64Index([2, 5, 11], dtype='int64')
```

Index objects also have many of the attributes familiar from NumPy arrays:

```
print(ind.size, ind.shape, ind.ndim, ind.dtype)
```

One difference between `Index` objects and NumPy arrays is that indices are immutable: that is, they cannot be modified via the normal means:

```
ind[1] = 0
-----
TypeError Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0

/Users/jakevdp/anaconda/envs/py3k/lib/python3.3/site-packages/pandas/core/i
 1046
```

```
1047     def __setitem__(self, key, value):
-> 1048         raise TypeError("Indexes does not support mutable operations")
1049
1050     def __getitem__(self, key):
```

TypeError: Indexes does not support mutable operations

This immutability makes it safer to share indices between multiple DataFrames and arrays, without the potential for nasty 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. Recall that Python has a built-in `set` object, which we explored in section X.X. The `Index` object follows many of the conventions of this built-in set object, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])

indA & indB # intersection
Int64Index([3, 5, 7], dtype='int64')

indA | indB # union
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

indA - indB # difference
Int64Index([-1, 0, 0, 0, -2], dtype='int64')

indA ^ indB # symmetric difference
Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods, e.g. `indA.intersection(indB)`. For more information on the variety of set operations implemented in Python, see section X.X. Nearly every syntax listed there, with the exception of operations which modify the set, can also be performed on `Index` objects.

Looking Forward

Above we saw the basics of the `Series`, `DataFrame`, and `Index` objects, which form the foundation of data-oriented computing with Pandas. We saw how they are similar to and different from other Python data structures, and how they can be created from scratch from these more familiar objects. Through this chapter, we'll go more into more detail about creation of these structures (including very useful interfaces for creating them from various file types) and manipulating data within these structures.

Just as understanding the effective use of NumPy arrays is fundamental to effective numerical computing in Python, understanding the effective use of Pandas structures is fundamental to the data munging required for data science in Python.

Data Indexing and Selection

In the previous chapter, we looked in detail at methods and tools to access, set, and modify values 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 mentioned above, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional Series object, and then move on to the more complicated two-dimensional DataFrame object.

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 we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data['b']
```

0.5

We can also use dictionary-like Python expressions and methods to examine the keys/indices and values:

```
'a' in data
True
data.keys()
Index(['a', 'b', 'c', 'd'], dtype='object')
list(data.items())
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`DataFrame` objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a series by assigning to a new index value:

```
data['e'] = 1.25
data

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 1D Array

A `Series` builds on this dictionary-like interface and provides array-style item selection via *slices*, *masking*, and *fancy indexing*, examples of which can be seen below:

```
# slicing by explicit index
data['a':'c']

a    0.25
b    0.50
c    0.75
dtype: float64

# slicing by implicit integer index
data[0:2]

a    0.25
b    0.50
dtype: float64

# masking
data[(data > 0.3) & (data < 0.8)]

b    0.50
c    0.75
dtype: float64

# fancy indexing
data[['a', 'e']]

a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e. `data['a':'c']`), the final index is *included* in the slice,

while when slicing with an implicit index (i.e. `data[0:2]`), the final index is *excluded* from the slice.

Indexers: `loc`, `iloc`, and `ix`

The slicing and indexing conventions above can be a source of confusion. For example, if your series has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
1    a
3    b
5    c
dtype: object

# explicit index when indexing
data[1]
'a'

# implicit index when slicing
data[1:3]
3    b
5    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special *indexer* attributes which explicitly access certain indexing schemes. These are not functional methods, but attributes which expose a particular slicing interface to the data in the Series.

First, the `loc` attribute allows indexing and slicing which always references the explicit index:

```
data.loc[1]
'a'

data.loc[1:3]
1    a
3    b
dtype: object
```

The `iloc` attribute allows indexing and slicing which always references the implicit Python-style index:

```
data.iloc[1]
'b'
```

```
data.iloc[1:3]  
3    b  
5    c  
dtype: object
```

A third indexing attribute, `ix`, is a hybrid of the two, and for Series objects is equivalent to standard `[]`-based indexing. The purpose of the `ix` indexer will become more apparent in the context of DataFrame objects, below.

One guiding principle of Python code (see the Zen of Python, section X.X) is that “explicit is better than implicit”. The explicit nature of `loc` and `iloc` make them very useful in maintaining clean and readable 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 the mixed indexing/slicing convention.

Data Selection in DataFrame

Recall that a DataFrame acts in many ways like a two-dimensional or structured array, and acts in many 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:

```
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
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

The individual Series which make up the columns of the dataframe can be accessed via dictionary-style indexing of the column name:

```
data['area']
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 which are strings:

```
data.area
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:

```
data.area is data['area']
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 example, the DataFrame has a `pop` method, so `data.pop` will point to this rather than the "pop" column:

```
data.pop is data['pop']
False
```

Like with the Series objects above, this dictionary-style syntax can also be used to modify the object, in this case adding a new column:

```
data['density'] = data['pop'] / data['area']
data
```

	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 section X.X.

DataFrame as Two-dimensional Array

As mentioned, we can also view the dataframe as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

```
data.values  
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, many familiar array-like observations can be done on the dataframe itself. For example, we can transpose the full dataframe to swap rows and columns:

```
data.transpose()
```

	California	Florida	Illinois	New York	Texas
area	423967.000000	170312.000000	149995.000000	141297.000000	695662.000000
pop	38332521.000000	19552860.000000	12882135.000000	19651127.000000	26448193.000000
density	90.413926	114.806121	85.883763	139.076746	38.01874

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:

```
data.values[0]  
array([ 4.23967000e+05,  3.83325210e+07,  9.04139261e+01])
```

While passing a single “index” to a dataframe accesses a column:

```
data['area']  
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 above. 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 in the result:

```
data.iloc[:3, :2]
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Similarly, using the `loc` indexer we can index the underlying data in an array-like style but using the explicit index and column names:

```
data.loc[:, 'Illinois', : 'pop']
```

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

The `ix` indexer allows a hybrid of these two approaches:

```
data.ix[:, 3, : 'pop']
```

	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 `Series` objects above.

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

```
data.loc[data.density > 100, ['pop', 'density']]
```

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Keep in mind also that 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 used to from NumPy:

```
data.iloc[0, 2] = 90  
data
```

	area	pop	density
California	423967	38332521	90.000000
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

To built-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 which might seem a bit inconsistent with the above discussion, but nevertheless can be very useful in practice. First, while direct integer *indices* are not allowed on DataFrames, direct integer *slices* are allowed, and are taken on rows rather than on columns as you might expect:

```
data[1:3]
```

	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:

```
data[data.density > 100]
```

	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 while these may not precisely fit the mold of the above conventions they are nevertheless quite useful in practice.

Summary

Here we have discussed the various ways to access and modify values within the basic Pandas data structures. With this, we're slowly building-up our fluency with manipulating and operating on labeled data within Pandas. In the next section, we'll take this a bit farther and begin to examine the types of *operations* that you can do on Pandas Series and DataFrame objects.

Operations in Pandas

One of the essential pieces of NumPy is the ability to perform quick elementwise operations, both with basic arithmetic (addition, subtraction, multiplication, etc.) and with more sophisticated operations (trigonometric functions, exponential and logarithmic functions, etc.). Pandas inherits much of this functionality from NumPy, and the *universal functions* (ufuncs for short) which we introduced in section X.X 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 the output, 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 potentially error-prone tasks with raw NumPy arrays – become essentially foolproof with Pandas. We will additionally see that there are well-defined operations between one-dimensional Series structures and two-dimensional DataFrame structures.

Ufuncs: Index Preservation

Because Pandas is designed to work with NumPy, any NumPy ufunc will work on pandas Series and DataFrame objects. Lets start by defining a simple Series and DataFrame on which to demonstrate this:

```
import pandas as pd
import numpy as np

rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser

0    6
1    3
2    7
3    4
dtype: int64

df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	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 Pandas object *with the indices preserved*:

```
np.exp(ser)
0      403.428793
1      20.085537
2     1096.633158
3      54.598150
dtype: float64
```

Or, for a slightly more complex calculation:

```
np.sin(df * np.pi / 4)
```

	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 Section X.X can be used in a similar manner.

UFuncs: Index Alignment

For binary operations on two Series or DataFrame objects, Pandas will align indices in the process of performing the operation. This is very convenient when working with incomplete data, as we'll see in some of the examples below.

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*:

```
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:

```
population / area
```

```
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 could be determined using standard Python set arithmetic on these indices:

```
area.index | population.index
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Any item for which one or the other does not have an entry is marked by `NaN`, or “Not a Number”, which is how Pandas marks missing data (see further discussion of missing data in Section X.X). This index matching is implemented this way for any of Python’s built-in arithmetic expressions; any missing values are filled-in with `NaN` by default:

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B

0    NaN
1      5
2      9
3    NaN
dtype: float64
```

If filling-in `NaN` values is not the desired behavior, the fill value can be modified 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 value:

```
A.add(B, fill_value=0)

0    2
1    5
2    9
3    5
dtype: float64
```

Index Alignment in DataFrame

A similar type of alignment takes place for *both* columns and indices when performing operations on dataframes:

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                  columns=list('AB'))
A
```

	A	B
0	1	11
1	5	1

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                  columns=list('BAC'))
B
```

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

A + B

	A	B	C
0	1	15	NaN
1	13	6	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. Similarly to the case of the Series, we can use the associated object's arithmetic method and pass any desired `fill_value` to be used in place of missing entries:

```
A.add(B, fill_value=np.mean(A.values))
```

	A	B	C
0	1.0	15.0	13.5
1	13.0	6.0	4.5
2	6.5	13.5	10.5

A table of Python operators and their equivalent Pandas object methods follows:

Operator	Pandas Method(s)
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()

Operator	Pandas Method(s)
%	mod()
**	pow()

Ufuncs: Operations between DataFrame and Series

When 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 2D and 1D NumPy array. Consider one common operation, where we find the difference of a 2D array and one of its rows:

```
A = rng.randint(10, size=(3, 4))
A

array([[3, 8, 2, 4],
       [2, 6, 4, 8],
       [6, 1, 3, 8]])

A - A[0]

array([[ 0,  0,  0,  0],
       [-1, -2,  2,  4],
       [ 3, -7,  1,  4]])
```

According to NumPy's broadcasting rules (see Section X.X), subtraction between a two-dimensional array and one of its rows is applied row-wise.

In Pandas, the convention similarly operates row-wise by default:

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

	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 above, while specifying the `axis` keyword:

```
df.subtract(df['R'], axis=0)
```

	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 above, will automatically align indices between the two elements:

```
halfrow = df.iloc[0, ::2]
halfrow
```

```
Q    3
S    2
Name: 0, dtype: int64
df - halfrow
```

	Q	R	S	T
0	0	NaN	0	NaN
1	-1	NaN	2	NaN
2	3	NaN	1	NaN

This preservation and alignment of indices and columns means that operations on data in Pandas will always maintain the data context, which prevents the types of silly errors that might come up when working with heterogeneous data in raw NumPy arrays.

Summary

We've shown that standard NumPy ufuncs will operate element-by-element on Pandas objects, with some additional useful functionality: they preserve index and column names, and automatically align different sets of indices and columns. Like the basic indexing and selection operations we saw in the previous section, these types of element-wise operations on Series and DataFrames form the building blocks of many more sophisticated data processing examples to come. The index alignment operations, in particular, sometimes lead to a state where values are missing from the resulting arrays. In the next section we will discuss in detail how Pandas chooses to handle such missing values.

Handling Missing Data

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

In this section, 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 missing data in general as “null”, “NaN”, or “NA” values.

Tradeoffs in Missing Data Conventions

There are a number of schemes that have been developed to indicate the presence of missing data in an array of data. Generally, they revolve around one of two strategies: using a *mask* which globally indicates missing values, or choosing a *sentinel value* which indicates a 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 tradeoffs: 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 which can be represented, and may require extra (often non-optimized) logic in CPU & GPU arithmetic. 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 patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell which indicates a NA state.

Missing Data in Pandas

Pandas' choice for how to handle missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for non-floating-point datatypes.

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 in Pandas' case. 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 an unwieldy amount of overhead in special-casing various operations for various types, and the implementation would probably require a new fork of the NumPy package.

NumPy does have support for masked arrays – i.e. arrays which have a separate boolean mask array attached which marks data as “good” or “bad”. Pandas could have

derived from this, but the overhead in both storage, computation, and code maintenance 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`. `None` is a Python singleton object which is often used for missing data in Python code. Because it 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):

```
import numpy as np
import pandas as pd

vals1 = np.array([1, None, 3, 4])
vals1

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 for arrays with native types.

```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()

dtype = object
10 loops, best of 3: 73.3 ms per loop

dtype = int
100 loops, best of 3: 3.08 ms per loop
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error.

```
vals1.sum()

-----
TypeError                                Traceback (most recent call last)

<ipython-input-4-749fd8ae6030> in <module>()
      1 vals1.sum()
```

```
/Users/jakevdp/anaconda/envs/py3k/lib/python3.3/site-packages/numpy/core/_methods.py in _sum(a, ax
    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 is because addition in Python between an integer and None is 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 that is recognized by all systems which use the standard IEEE floating-point representation.

```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype

dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array above, this array supports fast operations pushed into compiled code. You should be aware that NaN is a bit like a data virus which infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN:

```
1 + np.nan

nan

0 * np.nan

nan
```

Note that this means that the sum or maximum of the values is well-defined (it doesn't result in an error), but not very useful:

```
vals2.sum(), vals2.min(), vals2.max()

(nan, nan, nan)
```

Keep in mind that NaN is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

Examples

Each of the above sentinel representations has its place, and Pandas is built to handle the two of them nearly interchangeably, and will convert between the two sentinel values where appropriate:

```

data = pd.Series([1, np.nan, 2, None])
data

0    1
1    NaN
2    2
3    NaN
dtype: float64

```

Keep in mind, though, that because `None` is a Python object type and `NaN` is a floating-point type, there is *no in-type NA representation in Pandas for string, boolean, or integer values*. Pandas gets around this by type-casting in cases where NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be up-cast to a floating point type to accommodate the NA:

```

x = pd.Series(range(2), dtype=int)
x[0] = None
x

0    NaN
1    1
dtype: float64

```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value. Though 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/casting approach works well in practice and in my experience only rarely causes issues.

Here is a short table of the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Promotion when storing NAs	NA sentinel value
floating	no change	<code>np.nan</code>
object	no change	<code>None</code> or <code>np.nan</code>
integer	cast to <code>float64</code>	<code>np.nan</code>
boolean	cast to <code>object</code>	<code>None</code> or <code>np.nan</code>

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating 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 finish this section with a brief discussion 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:

```
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()

0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in section X.X, boolean masks can be used directly as a Series or DataFrame index:

```
data[data.notnull()]

0      1
2  hello
dtype: object
```

The `isnull()` and `notnull()` methods produce similar boolean results for DataFrames.

Dropping Null Values

In addition to the masking used above, there are the convenience methods, `dropna()` and `fillna()`, which respectively remove NA values and fill-in NA values. For a Series, the result is straightforward:

```
data.dropna()

0      1
2  hello
dtype: object
```

For a dataframe, there are more options. Consider the following dataframe:

```
df = pd.DataFrame([[1,       np.nan, 2],
                  [2,        3,      5],
                  [np.nan,   4,      6]])
df
```

	0	1	2
0	1	NaN	2
1	2	3	5
2	NaN	4	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:

```
df.dropna()
```

	0	1	2
1	2	3	5

Alternatively, you can drop NA values along a different axis: `axis=1` drops all columns containing a null value:

```
df.dropna(axis=1)
```

	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 specified through 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` keyword) containing a null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns which are *all* null values:

```
df[3] = np.nan
df
```

	0	1	2	3
0	1	NaN	2	NaN
1	2	3	5	NaN
2	NaN	4	6	NaN

```
df.dropna(axis=1, how='all')
```

	0	1	2
0	1	NaN	2
1	2	3	5
2	NaN	4	6

Keep in mind that to be a bit more clear, you can use `axis='rows'` rather than `axis=0` and `axis='columns'` rather than `axis=1`.

For finer-grained control, the `thresh` parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
df.dropna(thresh=3)
```

	0	1	2	3
1	2	3	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 using the `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 values replaced.

Consider the following Series:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
a    1
b    NaN
c    2
d    NaN
e    3
dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
```

```
a    1
b    0
```

```
c    2  
d    0  
e    3  
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill  
data.fillna(method='ffill')  
  
a    1  
b    1  
c    2  
d    2  
e    3  
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
# back-fill  
data.fillna(method='bfill')  
  
a    1  
b    2  
c    2  
d    3  
e    3  
dtype: float64
```

For DataFrames, the options are similar, but we can also specify an `axis` along which the fills take place:

```
df
```

	0	1	2	3
0	1	NaN	2	NaN
1	2	3	5	NaN
2	NaN	4	6	NaN

```
df.fillna(method='ffill', axis=1)
```

	0	1	2	3
0	1	1	2	2
1	2	3	5	5
2	NaN	4	6	6

Notice that if a previous value is not available during a forward fill, the NA value remains.

Summary

Here we have seen how Pandas handles null/NA values, and seen a few DataFrame and Series methods specifically designed to handle these missing values in a uniform way. Missing data is a fact of life in real-world datasets, and we'll see these tools often in the following chapters.

Hierarchical Indexing

Up to this point we've been focused primarily on one-dimensional and two-dimensional data, stored in Pandas Series and DataFrame objects respectively. But sometimes you'd like to go beyond this and store higher-dimensional data, that is, data indexed by more than one or two keys. While Pandas does provide Panel and Panel4D objects which natively handle 3D and 4D data (see below), 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 MultiIndex objects, considerations when indexing, slicing, and computing statistics across multiply-indexed data, and useful routines for converting between simple and hierarchically-indexed representations of your data.

```
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, consider a series of data where each point has a character and numerical key.

The Bad Way...

Consider a case in which you want to track data about states in two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
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)
```

```
(California, 2000)    33871648
(California, 2010)   37253956
(New York, 2000)    18976457
(New York, 2010)    19378102
(Texas, 2000)        20851820
(Texas, 2010)        25145561
dtype: int64
```

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```
pop[('California', 2010):('Texas', 2000)]
(California, 2010)   37253956
(New York, 2000)    18976457
(New York, 2010)    19378102
(Texas, 2000)        20851820
dtype: int64
```

but the convenience ends there. If you wish, for example, to select all 2010 values, you'll need to do some messy and slow munging to make it happen:

```
pop[[i for i in pop.index if i[1] == 2010]]
(California, 2010)   37253956
(New York, 2010)    19378102
(Texas, 2010)        25145561
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 poor-man's multi-index, and the Pandas `MultiIndex` type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

```
mindex = pd.MultiIndex.from_tuples(index)
mindex
MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
           labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the `MultiIndex` contains multiple *levels* of indexing, in this case the state names and the years, as well as multiple *labels* for each data point which encode these levels.

If we re-index our series with this `MultiIndex`, we see the Hierarchical representation of the data:

```
pop = pop.reindex(mindex)
pop
```

```
California  2000    33871648  
              2010    37253956  
New York    2000    18976457  
              2010    19378102  
Texas       2000    20851820  
              2010    25145561  
dtype: int64
```

Here the first two columns of the Series representation show the multiple index values, while the third column shows our data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

Now to access all data for which the second index is 2010, we can simply use Pandas' slicing notation:

```
pop[:, 2010]  
  
California    37253956  
New York      19378102  
Texas         25145561  
dtype: int64
```

The result is a singly-indexed array with just the keys we're interested in. This syntax is much more convenient (and the operation is much more efficient!) than the home-spun tuple-based multi-indexing solution that we started with. Below we'll further discuss this sort of indexing operation on hierarchically indexed data.

MultIndex as Extra Dimension

The astute reader 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:

```
pop_df = pop.unstack()  
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

Naturally, the `stack()` method provides the opposite operation:

```
pop_df.stack()  
  
California  2000    33871648  
              2010    37253956  
New York    2000    18976457
```

```
    2010    19378102  
Texas    2000    20851820  
          2010    25145561  
dtype: int64
```

Seeing this, you might wonder why we would bother with hierarchical indexing 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 represent three or higher-dimensional data 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 (say, population under 18) data for each state at each year; with a multiindex this is as easy as adding another column to the dataframe.

```
pop_df = pd.DataFrame({'total': pop,  
                      'under18': [9267089, 9284094,  
                                  4687374, 4318033,  
                                  5906301, 6879014]})  
pop_df
```

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

In addition, all the ufuncs and other functionality discussed in Section X.X work with hierarchical indices as well:

```
f_u18 = pop_df['under18'] / pop_df['total']  
print("fraction of population under 18:")  
f_u18.unstack()  
  
fraction of population under 18:
```

	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.

Aside: Panel Data

Pandas has a few other fundamental data structures that we have not yet discussed, namely the `pd.Panel` and `pd.Panel4D` objects. These can be thought of, respectively, as three-dimensional and four-dimensional generalizations of the (one-dimensional) Series and (two-dimensional) DataFrame structures. Once you are familiar with indexing and manipulation of data in a Series and DataFrame, the Panel and Panel4D are relatively straightforward to use. In particular, the `ix`, `loc`, and `iloc` indexers discussed in section X.X extend readily to these higher-dimensional structures.

We won't cover these panel structures further in this text, as I've found in the majority of cases that multi-indexing is a more useful and conceptually simple representation for higher-dimensional data. Additionally, panel data is fundamentally a dense data representation, while multiindexing is fundamentally a sparse data representation. As the number of dimensions grows, the dense representation becomes very inefficient for the majority of real-world datasets. For the occasional specialized application, however, these structures can be useful. If you'd like to read more about the Panel and Panel4D structures, see the references listed in section X.X.

Methods of MultiIndex Creation

The most straightforward way to construct a multiply-indexed Series or DataFrame is to simply pass a list of two index arrays to the constructor. For example:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])
```

df

		data1	data2
a	1	0.111677	0.887069
	2	0.195157	0.782505
b	1	0.921233	0.912892
	2	0.493172	0.736983

The work of creating the MultiIndex is done in the background.

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
```

```
('New York', 2010): 19378102}
pd.Series(data)

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 couple of these methods below.

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, in analogy to above, you can construct the MultiIndex from a simple list of arrays giving the index values within each level:

```
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
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:

```
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
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 unique index values:

```
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
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 which reference these labels:

```
pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
              labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Any of these objects can be passed as the `index` argument when creating a Series or Dataframe, or be passed to the `reindex` method of an existing Series or DataFrame.

MultilIndex Level Names

Sometimes it is convenient to name the levels of the MultiIndex. This can be accomplished by passing the `names` argument to any of the above MultiIndex constructors, or by setting the `names` attribute of the index after the fact:

```
pop.index.names = ['state', 'year']
pop

state      year
California 2000    33871648
              2010    37253956
New York   2000    18976457
              2010    19378102
Texas      2000    20851820
              2010    25145561
dtype: int64
```

With more involved data sets, this can be a useful way to keep track of the meaning of various index values.

MultilIndex for Columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```
# 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
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	42	35.3	35	36.1	43	36.1
	2	12	37.7	38	37.0	33	35.8

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2014	1	44	37.4	53	37.0	37	36.7
	2	46	37.9	26	36.0	27	35.3

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 DataFrame containing just that person's information:

```
health_data['Guido']
```

	type	HR	Temp
year	visit		
2013	1	35	36.1
	2	38	37.0
2014	1	53	37.0
	2	26	36.0

For complicated records containing multiple labeled measurements across multiple times for many subjects (be they 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 series of state populations we saw above:

```
pop
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:

```
pop['California', 2000]  
33871648
```

The MultiIndex also supports *partial indexing*, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained:

```
pop['California']  
year  
2000    33871648  
2010    37253956  
dtype: int64
```

Partial slicing is available as well, as long as the MultiIndex is sorted (see discussion below):

```
pop.loc['California':'New York']  
state      year  
California 2000    33871648  
                  2010    37253956  
New York   2000    18976457  
                  2010    19378102  
dtype: int64
```

With sorted indices, partial indexing can be performed on lower levels by passing an empty slice in the first index:

```
pop[:, 2000]  
state  
California    33871648  
New York     18976457  
Texas        20851820  
dtype: int64
```

Other types of indexing and selection discussed in section X.X work as well; for example selection based on boolean masks:

```
pop[pop > 22000000]  
state      year  
California 2000    33871648  
                  2010    37253956  
Texas      2010    25145561  
dtype: int64
```

Selection based on fancy indexing works as well:

```
pop[['California', 'Texas']]
```

```

state      year
California 2000    33871648
              2010    37253956
Texas       2000    20851820
              2010    25145561
dtype: int64

```

Multiply-indexed DataFrames

A multiply-indexed DataFrame behaves in a similar manner. Consider our toy medial dataframe from above:

```
health_data
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	42	35.3	35	36.1	43	36.1
	2	12	37.7	38	37.0	33	35.8
2014	1	44	37.4	53	37.0	37	36.7
	2	46	37.9	26	36.0	27	35.3

Remember that columns are primary in a dataframe, and the syntax used for multiply indexed Series above applies to the columns. For example, we can recover Guido's heartrate data with a simple operation:

```

health_data['Guido', 'HR']

year  visit
2013 1      35
      2      38
2014 1      53
      2      26
Name: (Guido, HR), dtype: float64

```

Also similarly to the single-index case, we can use the `loc`, `iloc`, and `ix` indexers introduced in section X.X. For example:

```
health_data.iloc[:, :2]
```

	subject	Bob	
	type	HR	Temp
year	visit		
2013	1	42	35.3
	2	12	37.7

these indexers provide an array-like view of the underlying two-dimensional data, but each individual index in `loc` or `iloc` can be passed a tuple of multiple indices. For example:

```
health_data.loc[:, ('Bob', 'HR')]

  year  visit
  2013    1      42
          2      12
  2014    1      44
          2      46
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:

```
health_data.loc[(:, 1), (:, 'HR')]

File "<ipython-input-32-8e3cc151e316>", line 1
    health_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 Pandas `IndexSlice` object, which is provided for precisely this situation. For example:

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

	subject	Bob	Guido	Sue
	type	HR	HR	HR
year	visit			
2013	1	42	35	43
2014	1	44	53	37

There are so many ways to interact with data in multiply-indexed Series and DataFrames, 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 effectively working with multiply-indexed data is knowing how to effectively transform the data. There are a number of operations which will preserve all the information in the dataset, but rearrange it for the purposes of various computations. We saw a brief example of this in the `stack()` and `unstack()` methods above, but there are many more ways to finely control the rearrangement of data between hierarchical indices and columns, and we'll explore them here.

Sorted and Unsorted Indices

We briefly mentioned a caveat above, but we should emphasize it more here. **Many of the 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*:

```
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

char  int
a      1    0.921624
       2    0.280299
c      1    0.459172
       2    0.586443
b      1    0.252360
       2    0.227359
dtype: float64
```

If we try to take a partial slice of this index, it will result in an error. For clarity here, we'll catch the error and print the message:

```
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)'
```

Though it is not entirely clear from the error message, this is the result of the fact that the MultiIndex is not sorted. For various reasons, partial slices and other similar operations 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:

```
data = data.sort_index()
data

char  int
a      1    0.921624
       2    0.280299
b      1    0.252360
       2    0.227359
c      1    0.459172
       2    0.586443
dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
data['a':'b']

char  int
a    1    0.921624
     2    0.280299
b    1    0.252360
     2    0.227359
dtype: float64
```

Stacking and Unstacking Indices

As we saw briefly above, it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation:

```
pop_df = pop.unstack()
pop_df
```

year	2000	2010
state		
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

The opposite of unstacking is stacking:

```
pop_df.stack()

state      year
California 2000    33871648
              2010    37253956
New York    2000    18976457
              2010    19378102
Texas       2000    20851820
              2010    25145561
dtype: int64
```

Each of these methods has keywords which can control the ordering of levels and dimensions in the output; see the method documentation for details.

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 population dictionary will result in a series 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:

```
pop_flat = pop.reset_index(name='population')
pop_flat
```

	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 working with data in the real world, the raw input data looks like this 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:

```
pop_flat.set_index(['state', 'year'])
```

		population
state	year	
California	2000	33871648
	2010	37253956
New York	2000	18976457
	2010	19378102
Texas	2000	20851820
	2010	25145561

In practice, I find this to be one of the most useful patterns with real-world datasets.

Data Aggregations on Multi-Indices

We've previously seen that Pandas has built-in data aggregation methods, such as `mean()`, `sum()`, `max()`, etc. For hierarchically indexed data, these can be passed a `level` parameter which controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
health_data
```

	subject	Bob		Guido		Sue	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	42	35.3	35	36.1	43	36.1
	2	12	37.7	38	37.0	33	35.8
2014	1	44	37.4	53	37.0	37	36.7
	2	46	37.9	26	36.0	27	35.3

Perhaps we'd like to average-out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the year:

```
data_mean = health_data.mean(level='year')
data_mean
```

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	27	36.50	36.5	36.55	38	35.95
2014	45	37.65	39.5	36.50	32	36.00

by further making use of the `axis` keyword, we can take the mean among levels on the columns as well:

```
data_mean.mean(axis=1, level='type')
```

type	HR	Temp
year		
2013	33.833333	36.333333
2014	38.833333	36.716667

Thus in two lines, we've been able to find the average heart rate and temperature measured among all subjects in all visits each year. While this is a toy example, many real-world datasets have similar hierarchical structure.

Summary

In this section we've covered many aspects of Hierarchical Indexing or Multi-Indexing, which is a convenient way to store, manipulate, and process multi-dimensional data. Probably the most important skill involved in this is the ability to transform data from hierarchical representations to flat representations to multi-dimensional representations. You'll find in working with real datasets that these

transformation operations are often the first step in preparing data for analysis. In a later section we'll see some examples of these tools in action.

Combining Datasets: Concat & Append

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges which correctly handle indices which might be shared between the datasets. Pandas series and dataframes are built with this type of operation in mind, and Pandas includes functions and methods which make this sort of data wrangling fast and straightforward.

In this section we'll take a look at simple concatenation of Series and DataFrames with the `pd.concat` function; in the following section we'll take a look at the more sophisticated in-memory merges and joins implemented in Pandas.

```
import pandas as pd
import numpy as np
```

For convenience, we'll define this function which creates a DataFrame of a particular form that will be useful below:

```
def make_df(cols, ind):
    """Quickly make a dataframe"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

In addition, we'll create a quick class which allows us to display multiple dataframes side-by-side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

```
class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
        <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
```

```

    self.args = args

def __repr_html__(self):
    return '\n'.join(self.template.format(a, eval(a).__repr_html__())
                    for a in self.args)

def __repr__(self):
    return '\n\n'.join(a + '\n' + repr(eval(a))
                      for a in self.args)

```

The use of this will become more clear below!

Recall: Concatenation of NumPy Arrays

Concatenation of Series and DataFrames is very similar to concatenation of Numpy arrays, which can be done via the `np.concatenate` function as discussed in Section X.X. Recall that with it, you can combine the contents of two arrays into a single array:

```

x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
np.concatenate([x, y, z])

array([1, 2, 3, 4, 5, 6, 7, 8, 9])

```

The first argument is a list or tuple of arrays to concatenate. Additionally, it takes an `axis` keyword that allows you to specify the axis along which the result will be concatenated:

```

x = [[1, 2],
      [3, 4]]
np.concatenate([x, x], axis=1)

array([[1, 2, 1, 2],
       [3, 4, 3, 4]])

```

Simple Concatenation with `pd.concat`

Pandas has a similar function, `pd.concat()`, which has a similar syntax, but which contains a number of options that we'll discuss below:

```
# Signature in Pandas v0.16
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False, copy=True)
```

`pd.concat()` can be used for a simple concatenation of Series or DataFrame object, just as `np.concatenate()` can be used for simple concatenations of arrays:

```

ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])

```

```
1    A  
2    B  
3    C  
4    D  
5    E  
6    F  
dtype: object
```

It also works to concatenate higher-dimensional objects, such as dataframes:

```
df1 = make_df('AB', [1, 2])  
df2 = make_df('AB', [3, 4])  
display('df1', 'df2', 'pd.concat([df1, df2])')
```

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

```
pd.concat([df1, df2])
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

By default, the concatenation takes place row-wise within the dataframe (i.e. `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

```
df3 = make_df('AB', [0, 1])  
df4 = make_df('CD', [0, 1])  
display('df3', 'df4', "pd.concat([df3, df4], axis='col')")
```

df3

	A	B
0	A0	B0
1	A1	B1

df4

	C	D
0	C0	D0
1	C1	D1

`pd.concat([df3, df4], axis='col')`

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

We could have equivalently specified `axis=1`; here we've used the more intuitive `axis='col'`.

Duplicate Indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation **preserves indices**, even if the result will have duplicate indices! Consider this simple example:

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
```

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y])
```

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

Notice the repeated indices in the result. While this is valid within DataFrames, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the Repeats as an Error. If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

```
try:  
    pd.concat([x, y], verify_integrity=True)  
except ValueError as e:  
    print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values: [0, 1]
```

Ignoring the Index. Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting Series:

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

```
x
```

	A	B
0	A0	B0
1	A1	B1

```
y
```

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y], ignore_index=True)
```

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Adding MultiIndex keys. Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")  
x
```

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

```
pd.concat([x, y], keys=['x', 'y'])
```

		A	B
x	0	A0	B0
	1	A1	B1
y	0	A2	B2
	1	A3	B3

The usefulness of this final version should be very apparent; it comes up often when combining data from different sources! The result is a multiply-indexed dataframe, and we can use the tools discussed in Section X.X to transform this data into the representation we're interested in.

Concatenation with Joins

When working with DataFrame objects, the concatenation takes place across one axis, and there are a few options for the set arithmetic to be used on the other columns.

Consider the concatenation of the following two dataframes, which have some (but not all!) columns in common:

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6])
```

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

Notice that by default, the entries for which no data is available are filled with NA values. To change this we can specify one of several options for the `join` and `join_axes` parameters of the concatenate function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'`:

```
display('df5', 'df6',
"pd.concat([df5, df6], join='inner')")
```

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

```
df6
```

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6], join='inner')
```

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

Another option is to directly specify the index of the remaining columns using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join_axes=[df5.columns])")
```

```
df5
```

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

```
df6
```

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

```
pd.concat([df5, df6], join_axes=[df5.columns])
```

	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	NaN	B3	C3
4	NaN	B4	C4

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data!

The `append()` Method

Because direct array concatenation is so common, Series and DataFrame objects have an `append` method which can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call

```
display('df1', 'df2', 'df1.append(df2)')  
df1
```

	A	B
1	A1	B1
2	A2	B2

`df2`

	A	B
3	A3	B3
4	A4	B4

```
df1.append(df2)
```

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Keep in mind that unlike the `append()` and `extend()` methods of Python lists, the pandas `append()` method does not modify the original object, but creates a new object with the combined data.

In the next section, we'll look at another more powerful approach to combining data from multiple sources, the database-style merges/joins implemented in `pd.merge`. For more information on `concat()`, `append()`, and related functionality, see the [Merge, Join, and Concatenate](#) section of the Pandas documentation.

Combining Datasets: Merge and Join

One extremely useful feature of 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` function, and we'll see few examples of how this can work in practice.

For convenience, we will start by re-defining the `display()` functionality that we used in the previous notebook:

```
import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                        for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                          for a in self.args)
```

Relational Algebra

The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases. The strength of the relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset. With this lexicon of fundamental operations implemented efficiently in a database or other program, a wide range of fairly complicated composite operations can be performed.

Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of Series and Dataframes. As we will see below, these let you efficiently link data from different sources.

Categories of Joins

The `pd.merge()` function implements a number of types of joins, which we'll briefly explore here: 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, as we will see below. 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 seen in Section X.X. As a concrete example, consider the following two dataframes which contain information on several employees in a company:

```
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]})
display('df1', 'df2')
```

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

To combine this information into a single dataframe, we can use the `pd.merge()` function:

```
display('df1', 'df2', "pd.merge(df1, df2)")
```

df1

	employee	group
0	Bob	Accounting
1	Jake	Engineering

	employee	group
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)

	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 DataFrame has 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 the above case, the order of the “employee” column differs between df1 and df2, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general **discards the index**, except in the special case of merges by index (see the `left_index` and `right_index` keywords, below).

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 duplicate entries as appropriate. Consider the following example of a many-to-one join.

```
df3 = pd.merge(df1, df2)
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', "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	Sue	HR	2014	Steve

The resulting DataFrame has 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 well-defined. If the key column in both the left and right array contains duplicates, then the 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:

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting', 'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux', 'spreadsheets', 'organization']}
display('df1', 'df5', "pd.merge(df1, df5)")
```

df1

	employee	group
0	Bob	Accounting

	employee	group
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	Sue	HR	spreadsheets
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 shown above. Below we'll consider some of the options provided by `pd.merge()` which enable you to tune how the join operations work.

Specification of the Merge Key

Above we see 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. 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` keyword, which takes a column name or a list of column names:

```
display('df1', 'df2', "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 column name.

The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is marked by “name” rather than “employee”. In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})

display('df1', 'df3', '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 which we can drop if desired, e.g. using the `drop()` method of DataFrames:

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

	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:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', '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()`:

```
display('df1a', 'df2a', "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 a merge which defaults to joining on indices:

```
display('df1a', 'df2a', 'df1a.join(df2a)')  
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

```
df2a
```

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

```
df1a.join(df2a)
```

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

```
display('df1a', 'df3', "pd.merge(df1a, df3, left_index=True, right_on='name')")  
df1a
```

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

```
df3
```

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

```
pd.merge(df1a, df3, left_index=True, right_on='name')
```

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this see the [Merge, Join, and Concatenate](#) section of the Pandas documentation.

Specifying Set Arithmetic for Joins

In all the above 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:

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'lamb', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
```

df6

	name	food
0	Peter	fish
1	Paul	lamb
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

```
pd.merge(df6, df7)
```

	name	food	drink
0	Mary	bread	wine

Here we have merged two datasets which 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”:

```
pd.merge(df6, df7, how='inner')
```

	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

```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")  
df6
```

	name	food
0	Peter	fish
1	Paul	lamb
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

```
pd.merge(df6, df7, how='outer')
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	lamb	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example,

```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")  
df6
```

	name	food
0	Peter	fish
1	Paul	lamb
2	Mary	bread

```
df7
```

	name	drink
0	Mary	wine
1	Joseph	beer

```
pd.merge(df6, df7, how='left')
```

	name	food	drink
0	Peter	fish	NaN
1	Paul	lamb	NaN
2	Mary	bread	wine

Notice here that the output rows correspond to the entries in the left input. Using `outer='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the above join types.

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:

```
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]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
```

```
df8
```

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

```
df9
```

	name	rank
0	Bob	3
1	Jake	1

	name	rank
2	Lisa	4
3	Sue	2

```
pd.merge(df8, df9, on="name")
```

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
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:

```
display('df8', 'df9', 'pd.merge(df8, df9, on="name", suffixes=["_L", "_R"]))  
df8
```

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

```
df9
```

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

	name	rank_L	rank_R
0	Bob	1	3

	name	rank_L	rank_R
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.

For more information on these patterns, see Section X.X where we dive a bit deeper into relational algebra. Also see the [Pandas Merge/Join documentation](#) for further discussion of these topics.

Example: US States Data

Merge and Join operations come up most often when 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/>

```
# 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 Pandas' `read_csv()` function:

```
pop = pd.read_csv('state-population.csv')
areas = pd.read_csv('state-areas.csv')
abbrevs = pd.read_csv('state-abbrevs.csv')

display('pop.head()', 'areas.head()', 'abbrevs.head()')

pop.head()
```

	state/region	ages	year	population
0	AL	under18	2012	1117489
1	AL	total	2012	4817528
2	AL	under18	2010	1130966
3	AL	total	2010	4785570
4	AL	under18	2011	1125763

```
areas.head()
```

	state	area (sq. mi)
0	Alabama	52423

	state	area (sq. mi)
1	Alaska	656425
2	Arizona	114006
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 & 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 figure it out.

We'll start with a many-to-one merge which will give us the full state name within the population dataframe. We want to merge based on the "state/region" column of pop, and the "abbreviation" column of abbrevs. We'll use `how='outer'` to make sure no data is thrown away due to mis-matched labels.

```
merged = pd.merge(pop, abbrevs, how='outer',
                   left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', 1) # drop duplicate info
merged.head()
```

	state/region	ages	year	population	state
0	AL	under18	2012	1117489	Alabama
1	AL	total	2012	4817528	Alabama
2	AL	under18	2010	1130966	Alabama
3	AL	total	2010	4785570	Alabama
4	AL	under18	2011	1125763	Alabama

Let's double-check whether there was any mis-matches here; we can do this by checking for rows with nulls:

```
merged.isnull().any()
```

```
state/region    False
ages           False
year            False
population     True
state          True
dtype: bool
```

Some of the "population" info is Null; let's figure out which these are!

```
merged[merged['population'].isnull()].head()
```

	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 "state" entries are also Null, which means that there was no corresponding entry in the abbrev key! Let's figure out which regions lack this match:

```
merged['state/region'][merged['state'].isnull()].unique()

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 the state abbreviation key. We can fix these quickly by filling-in appropriate entries:

```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()

state/region    False
ages           False
year            False
population     True
state          False
dtype: bool
```

No more NULLs in the state column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining what's above, we will want to join on the 'state' column in both:

```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489	Alabama	52423
1	AL	total	2012	4817528	Alabama	52423
2	AL	under18	2010	1130966	Alabama	52423
3	AL	total	2010	4785570	Alabama	52423
4	AL	under18	2011	1125763	Alabama	52423

Again, let's check for Nulls to see if there were any mis-matches.

```
final.isnull().any()

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:

```
final['state'][final['area (sq. mi)'].isnull()].unique()

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, e.g. the sum of all state areas), but in this case we'll just drop the null values because the population density of the entire US is not relevant to our current discussion:

```
final.dropna(inplace=True)
final.head()
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489	Alabama	52423
1	AL	total	2012	4817528	Alabama	52423
2	AL	under18	2010	1130966	Alabama	52423
3	AL	total	2010	4785570	Alabama	52423
4	AL	under18	2011	1125763	Alabama	52423

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2000, and the total population. We'll use the `query()` function to do this quickly (see Section X.X):

```
data2010 = final.query("year == 2010 & ages == 'total'")  
data2010.head()
```

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570	Alabama	52423
91	AK	total	2010	713868	Alaska	656425
101	AZ	total	2010	6408790	Arizona	114006
189	AR	total	2010	2922280	Arkansas	53182
197	CA	total	2010	37333601	California	163707

Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result.

```
data2010.set_index('state', inplace=True)  
density = data2010['population'] / data2010['area (sq. mi)']  
  
density.sort(ascending=False)  
density.head()  
  
state  
District of Columbia    8898.897059  
Puerto Rico            1058.665149  
New Jersey              1009.253268  
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 of their population density, in residents per square mile. We can see that Washington DC is by far the densest region in this dataset; the densest US state is New Jersey.

We can also check the end of the list:

```
density.tail()  
  
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, with just over one person per square mile.

This type of messy data merging is a common task when trying to answer questions based on real-world data. I hope that this example has given you an idea of the ways you can combine tools we've learned about to gain insight from real-world 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 number 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 *Group-By*. We'll see examples of these below.

For convenience, we'll use the same `display` magic function that we've seen in previous sections:

```
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                        for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                          for a in self.args)
```

Planets Data

Below we will use the *planets* dataset, available via the `seaborn` library. 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:

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape

(1035, 6)

planets.head()
```

	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 more than thousand planets discovered around other stars up to 2014.

Simple Aggregation in Pandas

In Section X.X, we explored some of the data aggregations available for NumPy arrays. For a Pandas Series, similarly to a one-dimensional NumPy array, the aggregates return a single value:

```

rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser

0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64

ser.sum()

2.8119254917081569

ser.mean()

0.56238509834163142

```

For a DataFrame, by default the aggregates return results within each column:

```

df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df

```

	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

```
df.mean()  
A    0.477888  
B    0.443420  
dtype: float64
```

By specifying the `axis` argument, you can instead aggregate along the columns instead:

```
df.mean(axis=1)  
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 Section X.X; in addition there is a convenience method `describe()` which computes several common aggregates for each column and returns the result. Let's use this on the planets data:

```
planets.describe()
```

	number	orbital_period	mass	distance	year
count	1035.000000	992.000000	513.000000	808.000000	1035.000000
mean	1.785507	2002.917596	2.638161	264.069282	2009.070531
std	1.240976	26014.728304	3.818617	733.116493	3.972567
min	1.000000	0.090706	0.003600	1.350000	1989.000000
25%	1.000000	5.442540	0.229000	32.560000	2007.000000
50%	1.000000	39.979500	1.260000	55.250000	2010.000000
75%	2.000000	526.005000	3.040000	178.500000	2012.000000
max	7.000000	730000.000000	25.000000	8500.000000	2014.000000

This can be a useful way to begin to understand a dataset. For example, we see in the `year` column that although exoplanets have been discovered since 1989, half of all known exoplanets were discovered since 2010! This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

Other built-in aggregations in Pandas are summarized in the following table:

Aggregation	Description
<code>count()</code>	total number of items
<code>first(), last()</code>	first or last item

Aggregation	Description
<code>mean()</code> , <code>median()</code>	mean or median
<code>min()</code> , <code>max()</code>	minimum or maximum
<code>std()</code> , <code>var()</code>	standard deviation or variance
<code>mad()</code>	mean absolute deviation
<code>prod()</code>	product of all items
<code>sum()</code>	sum of all items

These are all methods of DataFrame and Series objects.

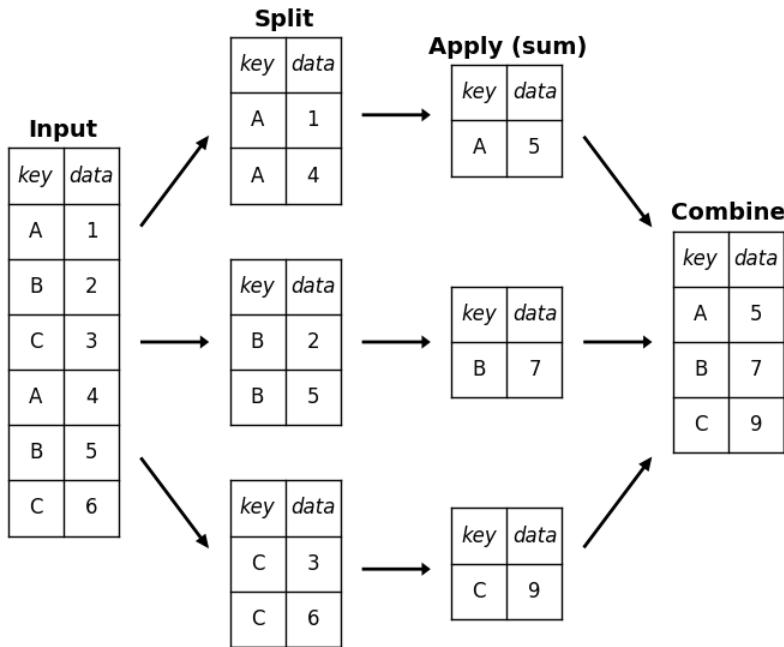
To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the *Group By* operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

Group By: 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 *Group By* 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

A canonical example of this split-apply-combine operation, where the “apply” is a summation aggregation, is illustrated below:



This figure makes abundantly clear what the GroupBy accomplishes:

- The **split** step involves breaking up and grouping a dataframe depending on the value of a particular key.
- The **apply** step involves computing some function, usually an aggregate, transformation, or filtering, over individual groups.
- The **combine** step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated*. It is quite easy to imagine creating a streaming algorithm which would simply update the sum, mean, count, min, or other aggregate for each group during a single pass over the data. The power of the GroupBy is that it abstracts-away these steps: a simple interface to this functionality wraps the more sophisticated computational decisions taking place under the hood.

As a concrete example, let's take a look at using Pandas for the above computation. We'll start by creating the input DataFrame:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

The most basic split-apply-combine operation can be computed with the `groupby()` method of dataframes, passing the name of the desired key column:

```
df.groupby('key')  
<pandas.core.groupby.DataFrameGroupBy object at 0x1024503d0>
```

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 does no computation until the aggregation is applied. This lazy evaluation approach means that common aggregates 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:

```
df.groupby('key').sum()
```

	data
key	
A	3
B	5
C	7

Notice that the *apply-combine* is accomplished within a single step. 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 below.

The GroupBy Object

The GroupBy object is a very flexible abstraction. In many ways, you can simply treat it as if it's a collection of DataFrames, and it does the difficult things under the hood.

Here are some examples, using the planets again:

Aggregate, Filter, Transform, Apply. The native operations built on GroupBy can be broadly split into *aggregation*, which we saw above, *filter*, which selects groups based on some criterion, *transform* which modifies groups more flexibly than aggregation, and *apply*, which is a general umbrella over operations on a group.

Because these are all very important and involved, we will discuss them in their own section further below.

Column Indexing. The GroupBy object supports column indexing in the same way as the DataFrame, and returns a modified GroupBy object. For example:

```
planets.groupby('method')
<pandas.core.groupby.DataFrameGroupBy object at 0x102450910>
planets.groupby('method')['orbital_period']
<pandas.core.groupby.SeriesGroupBy object at 0x102450f10>
```

Here we've selected a particular Series group from the original DataFrame group by reference to its column name. We can then compute any aggregate or other operation on this sub-GroupBy:

```
planets.groupby('method')['orbital_period'].median()

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
```

This gives an idea of the general scale of orbital periods (in days) that the methods are sensitive to.

Iteration over groups. The GroupBy object supports direct iteration over the groups, returning each group as a Series or DataFrame:

```
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 to use the built-in GroupBy functionality, discussed further below.

Dispatch Methods. Through some Python class magic, any method not explicitly implemented by the GroupBy object 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 which describe each group in the data.

```
planets.groupby('method')['year'].describe().unstack()
```

	count	mean	std	min	25%	50%	75%	max
method								
Astrometry	2	2011.500000	2.121320	2010	2010.75	2011.5	2012.25	2013
Eclipse Timing Variations	9	2010.000000	1.414214	2008	2009.00	2010.0	2011.00	2012
Imaging	38	2009.131579	2.781901	2004	2008.00	2009.0	2011.00	2013
Microlensing	23	2009.782609	2.859697	2004	2008.00	2010.0	2012.00	2013
Orbital Brightness Modulation	3	2011.666667	1.154701	2011	2011.00	2011.0	2012.00	2013
Pulsar Timing	5	1998.400000	8.384510	1992	1992.00	1994.0	2003.00	2011
Pulsation Timing Variations	1	2007.000000	NaN	2007	2007.00	2007.0	2007.00	2007
Radial Velocity	553	2007.518987	4.249052	1989	2005.00	2009.0	2011.00	2014
Transit	397	2011.236776	2.077867	2002	2010.00	2012.0	2013.00	2014
Transit Timing Variations	4	2012.500000	1.290994	2011	2011.75	2012.5	2013.25	2014

Looking at this table helps us to better understand the data: for example, the vast majority of planets have been discovered by the **Radial Velocity** and **Transit** methods, though the latter only became common (due to new, more accurate telescopes) in the last decade. The newest methods seem to be **Transit Timing Variation** and **Orbital Brightness Modulation**, which were not used to discover a new planet until 2011.

This is just one example of the utility of dispatch methods. Notice that they are applied to each *DataFrame/Series in the group*, and the results are then combined within GroupBy and returned. Again, any valid DataFrame/Series method can be used on the corresponding GroupBy object, which allows for some very flexible and powerful operations!

Aggregate, Filter, Transform, Apply

Above we have been focusing on aggregation for the combine operation, but there are more options available. In particular, GroupBy objects have an `aggregate()`, `filter()`, `transform()`, and `apply()` methods which efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the selections below, we'll use the following dataframe:

```
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
```

	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, but the `aggregate()` method allows for even more flexibility. It can take a string, a function, or a list thereof. Here is a quick example combining all these:

```
df.groupby('key').aggregate(['min', np.median, max])
```

	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 aggregation operations

```
df.groupby('key').aggregate({'data1': 'min',
                            'data2': 'max'})
```

	data2	data1
key		
A	5	0
B	7	1
C	9	2

Filtering. A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation meets some cut:

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()", "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()
```

	data1	data2
key		
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 filtering operation.

Transformation. While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. A common example is to center the data by subtracting the group-wise mean:

```
center = lambda x: x - x.mean()
df.groupby('key').transform(center)
```

	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

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 Pandas object (e.g. DataFrame, Series) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` which normalizes the first column by the sum of the second:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")

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').apply(norm_by_data2)
```

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

`apply()` within a GroupBy is extremely flexible: the only criterion is that the function takes a dataframe and returns a Pandas object or scalar; what you do in the middle is up to you!

Specifying Groups

In the above simple examples, we have split the DataFrame on a single column name. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here:

A list, array, series, or index providing the grouping keys. The key can be any appropriate series. For example:

```
L = [0, 1, 0, 1, 2, 0]
display('df', 'df.groupby(L).sum()')
```

```
df
```

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3

	key	data1	data2
3	A	3	3
4	B	4	7
5	C	5	9

```
df.groupby(L).sum()
```

	data1	data2
0	7	17
1	4	3
2	4	7

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from above:

```
display('df', "df.groupby(df['key']).sum()")  
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(df['key']).sum()
```

	data1	data2
key		
A	3	8
B	5	7
C	7	12

A dictionary or series mapping index to group. Another method is to provide a dictionary which maps index values to the group keys:

```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
```

```
df2
```

	data1	data2
key		
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

Any Python function. Similarly to the mapping, you can pass any Python function which will input the index value and output the group:

```
display('df2', 'df2.groupby(str.lower).mean()')
```

```
df2
```

	data1	data2
key		
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

```
df2.groupby(str.lower).mean()
```

	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

A list of valid keys. Further, any of the above key choices can be combined to group on a multi-index:

```
df2.groupby([str.lower, mapping]).mean()
```

		data1	data2
a	vowel	1.5	4.0
b	consonant	2.5	3.5
c	consonant	3.5	6.0

Grouping Example. As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0	0	0	2
Eclipse Timing Variations	0	0	5	10
Imaging	0	0	29	21
Microlensing	0	0	12	15
Orbital Brightness Modulation	0	0	0	5
Pulsar Timing	0	9	1	1
Pulsation Timing Variations	0	0	1	0
Radial Velocity	1	52	475	424
Transit	0	0	64	712
Transit Timing Variations	0	0	0	9

This shows the power of combining many of the operations we've discussed up to this point when looking at realistic datasets. We immediately gain a coarse understanding of when and how planets have been discovered over the past several decades!

Here I would suggest digging into the above few lines of code, and evaluating the individual steps to make sure you understand exactly what they are doing to the result. It's certainly a rather complicated example, but once you understand the pieces you'll understand the process!

Pivot Tables

In the previous section, we looked at grouping operations. A *pivot table* is a related operation which is commonly seen in spreadsheets and other programs which operate on tabular data. The Pivot Table takes simple column-wise data as input, and groups the entries into a two-dimensional table which provides a multi-dimensional 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 **multi-dimensional** 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:

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')

titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	Fal...
1	1	1	female	38	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	Fal...
2	1	3	female	26	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	Tru...
3	1	1	female	35	1	0	53.1000	S	First	woman	False	C	Southampton	yes	Fal...
4	0	3	male	35	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	Tru...

This contains a wealth of information on each passenger of that ill-fated voyage, including their gender, age, class, fare paid, and much more.

Pivot Tables By Hand

To start learning more about this data, we might want to like to group it by gender, survival, or some combination thereof. If you have read the previous section, you

might be tempted to apply a GroupBy operation to this data. For example, let's look at survival rate by gender:

```
titanic.groupby('sex')[['survived']].mean()
```

	survived
sex	
female	0.742038
male	0.188908

This immediately gives us some insight: overall three of every four females on board survived, while only one in five males survived!

This is an interesting insight, but we might like to go one step deeper and look at survival by both sex and, say, class. Using the vocabulary of GroupBy, we might proceed something like this: We *group by* class and gender, *select* survival, *apply* a mean aggregate, *combine* the resulting groups, and then *unstack* the hierarchical index to reveal the hidden multidimensionality. In code:

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

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 particularly easy to read or use. This type of operation is common enough that Pandas includes a convenience routine, `pivot_table`, which succinctly handles this type of multi-dimensional aggregation.

Pivot Table Syntax

Here is the equivalent to the above operation using the `pivot_table` method of dataframes:

```
titanic.pivot_table('survived', index='sex', columns='class')
```

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000

class	First	Second	Third
sex			
male	0.368852	0.157407	0.135447

This is eminently more readable than the equivalent GroupBy operation, 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 near certainty (hi Kate!), while only one in ten third-class men survived (sorry Leo!).

Multi-level Pivot Tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, 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.cut function:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', 'age'], 'class')
```

	class	First	Second	Third
sex	age			
female	(0, 18]	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	(0, 18]	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

we can do the same game with the columns; let's add info on the fare paid using pd.qcut to automatically compute quantiles:

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', 'age'], [fare, 'class'])
```

	fare	[0, 14.454]			(14.454, 512.329]		
	class	First	Second	Third	First	Second	Third
sex	age						
female	(0, 18]	NaN	1.000000	0.714286	0.909091	1.000000	0.318182
	(18, 80]	NaN	0.880000	0.444444	0.972973	0.914286	0.391304
male	(0, 18]	NaN	0.000000	0.260870	0.800000	0.818182	0.178571
	(18, 80]	0	0.098039	0.125000	0.391304	0.030303	0.192308

The result is a four-dimensional aggregation, shown in a grid which demonstrates the relationship between the values.

Additional Pivot Table Options

The full call signature of the `pivot_table` method of DataFrames is as follows:

```
DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean',
                      fill_value=None, margins=False, dropna=True)
```

Above we've seen examples of the first three arguments; here we'll take a quick look at the remaining arguments. 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 representing one of several common choices (e.g. `'sum'`, `'mean'`, `'count'`, `'min'`, `'max'`, etc.) or a function which implements an aggregation (e.g. `np.sum()`, `min()`, `sum()`, etc.). Additionally, it can be specified as dictionary mapping a column to any of the above desired options:

```
titanic.pivot_table(index='sex', columns='class',
                     aggfunc={'survived':sum, 'fare':'mean'})
```

	survived			fare		
class	First	Second	Third	First	Second	Third
sex						
female	91	70	72	106.125798	21.970121	16.118810
male	45	17	47	67.226127	19.741782	12.661633

Notice also here that we've omitted the `values` keyword; when 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:

```
titanic.pivot_table('survived', index='sex', columns='class', margins=True)
```

class	First	Second	Third	All
sex				
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%.

Example: Birthrate Data

As a more interesting example, let's take a look at the freely-available data on births in the USA, 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](#).

```
# shell command to download the data:  
# !curl -O https://raw.githubusercontent.com/jakevdp/data-CDCbirths/master/births.csv  
  
births = pd.read_csv('births.csv')
```

Taking a look at the data, we see that it's relatively simple: it contains the number of births grouped by date and gender:

```
births.head()
```

	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 decade column, and take a look at male and female births as a function of decade:

```
births['decade'] = 10 * (births['year'] // 10)  
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')
```

gender	F	M
decade		
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 Pandas' built-in plotting tools to visualize the total number of births by year (see Chapter X.X for a discussion of plotting with matplotlib):

```
%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');
```



With a simple pivot table and `plot()` method, we can immediately see the annual trend in births by gender. By eye, we find 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 interesting 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 mistyped dates (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:

```
# Some data is mis-reported; e.g. June 31st, etc.
# remove these outliers via robust sigma-clipping
quartiles = np.percentile(births['births'], [25, 50, 75])
```

```
mu = quartiles[1]
sig = 0.7413 * (quartiles[2] - quartiles[0])
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':

```
# 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 section X.X). This allows us to quickly compute the weekday corresponding to each row:

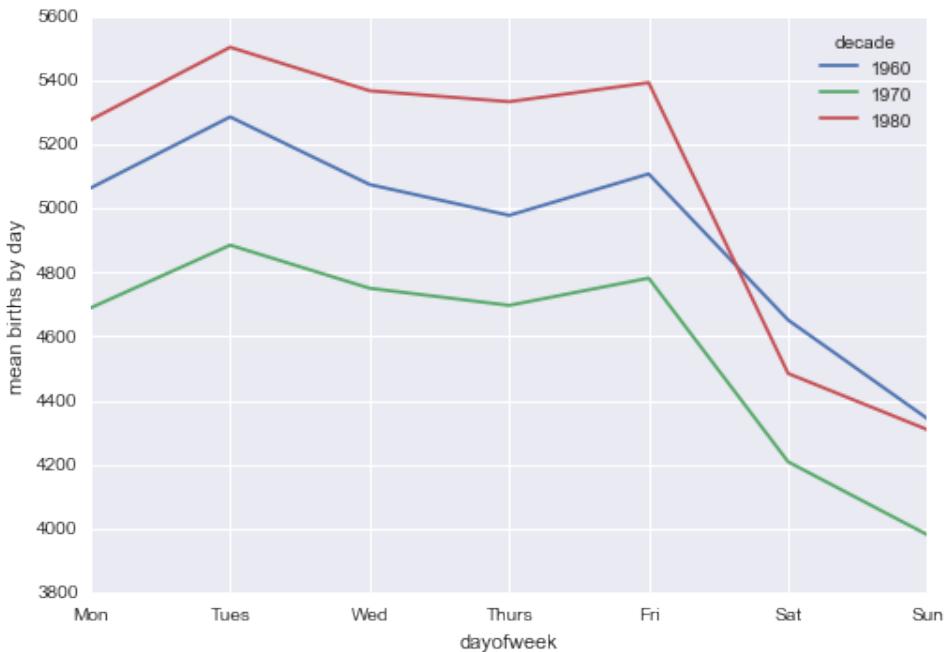
```
# create a datetime index from the year, month, day
births.index = pd.to_datetime(10000 * births.year +
                             100 * births.month +
                             births.day, format='%Y%m%d')

births['dayofweek'] = births.index.dayofweek
```

Using this we can plot births by weekday for several decades:

```
import matplotlib.pyplot as plt
import 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');
```



Apparently births are slightly less common on weekends than on weekdays! Note that the 1990s and 2000s are missing because the CDC stopped reports 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*. We can do this by constructing a datetime array for a particular year, making sure to choose a leap year so as to account for February 29th.

```
# Choose a leap year to display births by date
dates = [pd.datetime(2012, month, day)
         for (month, day) in zip(births['month'], births['day'])]
```

We can now group by the data by day of year and plot the results. We'll additionally annotate the plot with the location of several US holidays:

```
# Plot the results
fig, ax = plt.subplots(figsize=(8, 6))
births.pivot_table('births', dates).plot(ax=ax)

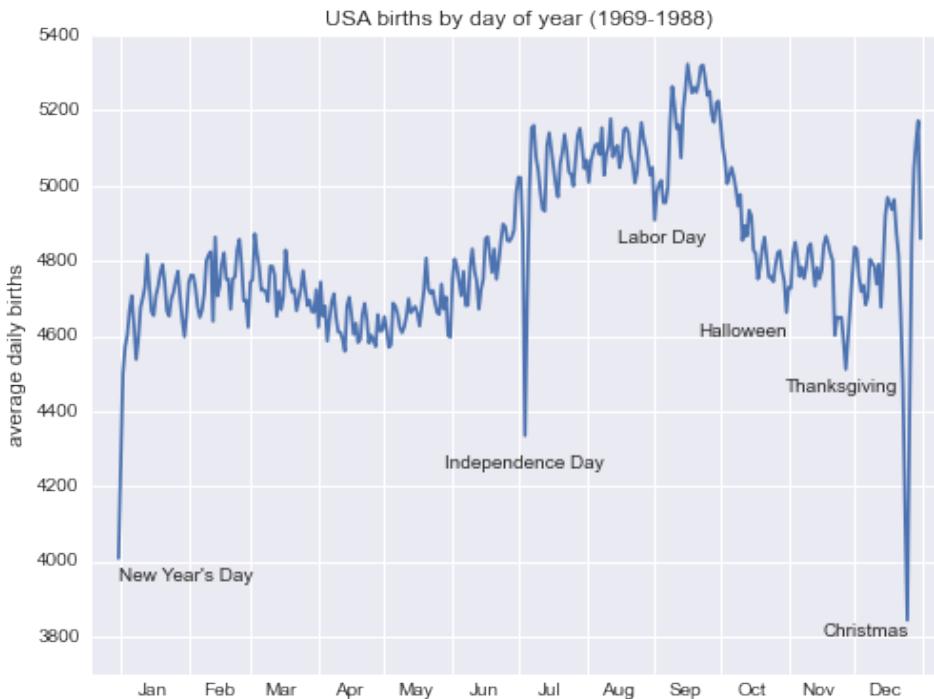
# Label the plot
ax.text('2012-1-1', 3950, "New Year's Day")
ax.text('2012-7-4', 4250, "Independence Day", ha='center')
ax.text('2012-9-4', 4850, "Labor Day", ha='center')
ax.text('2012-10-31', 4600, "Halloween", ha='right')
ax.text('2012-11-25', 4450, "Thanksgiving", ha='center')
ax.text('2012-12-25', 3800, "Christmas", ha='right')
ax.set(title='USA births by day of year (1969-1988)',
```

```

ylabel='average daily births',
xlim=('2011-12-20','2013-1-10'),
ylim=(3700, 5400);

# Format the x axis with centered month labels
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%h'));

```



The lower birthrate on holidays is striking, but is likely the result of selection for scheduled/induced births rather than any deep psychosomatic causes. For more discussion on this trend, see the discussion and links in [Andrew Gelman's blog posts](#) on the subject.

This short example should give you a good idea of how many of the Pandas tools we've seen to this point can be put together and used to gain insight from a variety of datasets. We will see some more sophisticated analysis of this data, and other datasets like it, in future sections!

Vectorized String Operations

Python has a very nice set of built-in operations for manipulating strings; we covered some of the basics in Section X.X. Pandas builds on this and provides a comprehensive set of *vectorized string operations* which become an essential piece of the type of munging required when working with real-world data.

In this section, we'll walk through some of the Pandas string operations, and then take a look at using them to partially clean-up a very messy dataset of recipes collected from the internet.

Introducing 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:

```
import numpy as np
x = np.array([2, 3, 5, 7, 11, 13])
x * 2

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

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]

['Peter', 'Paul', 'Mary', 'Guido']
```

This is perhaps sufficient to work with data, but it will break if there are any missing values. For example:

```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
[s.capitalize() for s in data]

-----
AttributeError                                Traceback (most recent call last)

<ipython-input-3-fc1d891ab539> in <module>()
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
----> 2 [s.capitalize() for s in data]

<ipython-input-3-fc1d891ab539> in <listcomp>(.0)
      1 data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
```

```
----> 2 [s.capitalize() for s in data]
```

```
AttributeError: 'NoneType' object has no attribute 'capitalize'
```

Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data through a bit of Python attribute magic. Pandas does this using the `str` attribute of Pandas Series, DataFrames, and Index objects. So, for example, if we create a Pandas Series with this data,

```
import pandas as pd
names = pd.Series(data)
names

0    peter
1     Paul
2     None
3    MARY
4   gUIDO
dtype: object
```

We can now call a single method which will capitalize all the entries, while skipping over any missing values:

```
names.str.capitalize()

0    Peter
1     Paul
2     None
3    Mary
4   Guido
dtype: object
```

This `str` attribute of Pandas objects is a special attribute which contains 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 available methods; we will start with that here, before diving deeper into a few of the subtleties.

Methods Similar to Python String Methods

Nearly all Python string methods have an equivalent Pandas vectorized string method; see Section X.X where a few of these are discussed.

Method	Description	Method	Description
<code>len()</code>	Equivalent to <code>len(str)</code> on each element	<code>ljust()</code>	Equivalent to <code>str.ljust()</code> on each element

Method	Description	Method	Description
rjust()	Equivalent to str.rjust() on each element	center()	Equivalent to str.center() on each element
zfill()	Equivalent to str.zfill() on each element	strip()	Equivalent to str.strip() on each element
rstrip()	Equivalent to str.rstrip() on each element	lstrip()	Equivalent to str.lstrip() on each element
lower()	Equivalent to str.lower() on each element	upper()	Equivalent to str.upper() on each element
find()	Equivalent to str.find() on each element	rfind()	Equivalent to str.rfind() on each element
index()	Equivalent to str.index() on each element	rindex()	Equivalent to str.rindex() on each element
capitalize()	Equivalent to str.capitalize() on each element	swapcase()	Equivalent to str.swapcase() on each element
translate()	Equivalent to str.translate() on each element	startswith()	Equivalent to str.startswith() on each element
endswith()	Equivalent to str.endswith() on each element	isalnum()	Equivalent to str.isalnum() on each element
isalpha()	Equivalent to str.isalpha() on each element	isdigit()	Equivalent to str.isdigit() on each element
isspace()	Equivalent to str.isspace() on each element	istitle()	Equivalent to str.istitle() on each element
islower()	Equivalent to str.islower() on each element	isupper()	Equivalent to str.isupper() on each element
isnumeric()	Equivalent to str.isnumeric() on each element	isdecimal()	Equivalent to str.isdecimal() on each element
split()	Equivalent to str.split() on each element	rsplit()	Equivalent to ``str.rsplit() on each element
partition()	Equivalent to str.partition() on each element	rpartition()	Equivalent to str.rpartition() on each element

Note that some of these, such as `capitalize()` above, return a series of strings:

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])

monte.str.lower()

0    graham chapman
1    john cleese
2    terry gilliam
3    eric idle
4    terry jones
```

```
5    michael palin
dtype: object
```

others return numbers:

```
monte.str.len()

0    14
1    11
2    13
3     9
4    11
5    13
dtype: int64
```

others return boolean values:

```
monte.str.startswith('T')

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:

```
monte.str.split()

0    [Graham, Chapman]
1    [John, Cleese]
2    [Terry, Gilliam]
3    [Eric, Idle]
4    [Terry, Jones]
5    [Michael, Palin]
dtype: object
```

The series-of-lists return value is the one that might give you pause: we'll take a look at this in more detail below.

Methods using Regular Expressions

In addition, there are several methods which accept regular expression to examine the content of each string element:

Method	Description	Method	Description
match()	Call <code>re.match()</code> on each element, returning a boolean.	extract()	Call <code>re.match()</code> on each element, returning matched groups as strings.
findall()	Call <code>re.findall()</code> on each element	replace()	Replace occurrences of pattern with some other string

Method	Description	Method	Description
contains()	Call <code>re.search()</code> on each element, returning a boolean	count()	Count occurrences of pattern
split()	Equivalent to <code>str.split()</code> , but accepts regexps	rsplit()	Equivalent to <code>str.rsplit()</code> , 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:

```
monte.str.extract('([A-Za-z]+)')

0    Graham
1    John
2    Terry
3    Eric
4    Terry
5    Michael
dtype: object
```

Or we can do something more complicated, like finding all names which start and end with a consonant, making use of the the start-of-string (^) and end-of-string (\$) regular expression characters:

```
monte.str.findall(r'^[AEIOU].*[aeiou]$')

0    [Graham Chapman]
1    []
2    [Terry Gilliam]
3    []
4    [Terry Jones]
5    [Michael Palin]
dtype: object
```

The ability to apply regular expressions across Series or Dataframe entries opens up many possibilities for analysis and cleaning of data.

Miscellaneous Methods

Finally, there are some miscellaneous methods that enable other convenient operations:

Method	Description	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

Method	Description	Method	Description
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 separator	get_dummies()	extract dummy variables as a dataframe

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 behavior is also available through Python's normal indexing syntax; e.g. `df.str.slice(0, 3)` is equivalent to `df.str[0:3]`:

```
monte.str[0:3]
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 likewise similar.

These `get()` and `slice()` methods also let you access elements of arrays returned by, e.g. `split()`. For example, to extract the last name of each entry, we can combine `split()` and `get()`:

```
monte.str.split().str.get(1)
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 indicator variables. For example, we might have a dataset which contains information in the form of codes, such as A="born in America", B="born in the United Kingdom", C="likes cheese", D="likes spam":

```
full_monte = pd.DataFrame({'name': monte,
                           'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C', 'B|C|D']})
full_monte
```

	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:

```
full_monte['info'].str.get_dummies('|')
```

	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 array of string processing procedures when cleaning your data.

Further Information

Above we saw just a brief survey of Pandas vectorized string methods. For more discussion of Python's string methods and basics of regular expressions, see Section X.X. For further examples of Pandas specialized string syntax, you can refer to Pandas' online documentation on [Working with Text Data](#).

Example: Recipe Database

These vectorized string operations become most useful in the process of cleanin-up messy real-world data. Here I'll walk through an example of that, using an open recipe database compiled from various sources on the web. Our goal will be to parse the recipe data into ingredient lists, so we can quickly find a recipe based on some ingredients we have on hand.

The scripts used to compile this can be found at <https://github.com/fictivekin/open-recipes>, and the link to the current version of the database is found there as well.

As of July 2015, this database is about 30 MB, and can be downloaded and unzipped with these commands:

```
# !curl -O http://openrecipes.s3.amazonaws.com/recipeitems-latest.json.gz
# !gunzip recipeitems-latest.json.gz
```

The database is in JSON format, so we will try `pd.read_json` to read it

```
try:
    recipes = pd.read_json('recipeitems-latest.json')
except ValueError as e:
    print("ValueError:", e)

ValueError: Trailing data
```

Oops! We get a `ValueError` mentioning that there is “trailing data”. Searching for this error on the internet, it seems that it’s due to using a file in which *each line* is itself a valid JSON, but the full file is not. Let’s check if this interpretation is true:

```
with open('recipeitems-latest.json') as f:
    line = f.readline()
    pd.read_json(line).shape

(2, 12)
```

Yes, apparently each line is a valid JSON, so we’ll need to string them together. One way we can do this is to actually construct a string representation containing all these JSON entries, and then load the whole thing with `pd.read_json`:

```
# read the entire file into a python array
with open('recipeitems-latest.json', 'r') as f:
    # Extract each line
    data = (line.strip() for line in f)
    # Reformat so each line is the element of a list
    data_json = "[{}].format('.join(data))"
    # read the result as a JSON
    recipes = pd.read_json(data_json)

recipes.shape

(173278, 17)
```

We see there are nearly 200,000 recipes, and we have 17 columns. Let’s take a look at one row to see what we have:

```
recipes.iloc[0]

_id                               {'$oid': '5160756b96cc62079cc2db15'}
cookTime                         PT30M
creator                           NaN
dateModified                      NaN
datePublished                     2013-03-11
description                       Late Saturday afternoon, after Marlboro Man ha...
image                             http://static.thepioneerwoman.com/cooking/file...
ingredients                      Biscuits\n3 cups All-purpose Flour\n2 Tablespo...
```

```
name           Drop Biscuits and Sausage Gravy
prepTime        PT10M
recipeCategory   NaN
recipeInstructions  NaN
recipeYield       12
source          thepioneerwoman
totalTime        NaN
ts                  {'$date': 1365276011104}
url      http://thepioneerwoman.com/cooking/2013/03/dro...
Name: 0, dtype: object
```

There is a lot of information there, but much of it is in a very messy form, as is typical of data scraped from the web. In particular, the ingredient list is in string format; we're going to have to carefully extract the information we're interested in. Let's start by taking a closer look at the ingredients:

```
recipes.ingredients.str.len().describe()

count    173278.000000
mean     244.617926
std      146.705285
min      0.000000
25%    147.000000
50%    221.000000
75%    314.000000
max     9067.000000
Name: ingredients, dtype: float64
```

The ingredient lists average 250 characters long, with a minimum of 0 and a maximum of nearly 10000 characters!

Just out of curiosity, let's see which recipe has the longest ingredient list:

```
recipes.name[np.argmax(recipes.ingredients.str.len())]

'Carrot Pineapple Spice & Brownie Layer Cake with Whipped Cream & Cream Cheese Frosting and'

That certainly looks like a complicated recipe.
```

We can do other aggregate explorations; for example, let's see how many of the recipes are for breakfast food:

```
recipes.description.str.contains('breakfast').sum()

3442
```

Or how many of the recipes list cinnamon as an ingredient:

```
recipes.ingredients.str.contains('[Cc]innamon').sum()

10526
```

We could even look to see whether any recipes mis-spell cinnamon with just a single "n":

```
recipes.ingredients.str.contains('[Cc]inamon').sum()
```

This is the type of essential data exploration that is possible with Pandas string tools. It is data munging like this that Python really excels at.

Building a Recipe Recommender

Let's go a bit further, and start working on a simple recipe recommendation system. The task is straightforward: given a list of ingredients, find a recipe which uses all those ingredients. While conceptually straightforward, the task is complicated by the heterogeneity of the data: there is no easy operation, for example, to extract a clean list of ingredients from each row. So we will cheat a bit: we'll start with a list of common ingredients, and simply search to see whether they are in each recipe's ingredient list. For simplicity, let's just stick with herbs and spices for the time being:

```
spice_list = ['salt', 'pepper', 'oregano', 'sage', 'parsley',
              'rosemary', 'tarragon', 'thyme', 'paprika', 'cumin']
```

We can then build a boolean dataframe consisting of True and False values, indicating whether this ingredient appears in the list:

```
import re
spice_df = pd.DataFrame(dict((spice, recipes.ingredients.str.contains(spice, re.IGNORECASE))
                               for spice in spice_list))
spice_df.head()
```

	cumin	oregano	paprika	parsley	pepper	rosemary	sage	salt	tarragon	thyme
0	False	False	False	False	False	False	True	False	False	False
1	False	False	False	False	False	False	False	False	False	False
2	True	False	False	False	True	False	False	True	False	False
3	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False

Now, as an example, let's say we'd like to find a recipe which uses parsley, paprika, and tarragon. We can compute this very quickly using the `query()` method of dataframes, discussed in Section X.X:

```
selection = spice_df.query('parsley & paprika & tarragon')
len(selection)
```

10

We find only ten recipes with this combination; let's use the index returned by this selection to discover the names of the recipes which have this combination:

```
recipes.name[selection.index]
2069      All cremat with a Little Gem, dandelion and wa...
74964     Lobster with Thermidor butter
```

```
93768      Burton's Southern Fried Chicken with White Gravy
113926          Mijo's Slow Cooker Shredded Beef
137686          Asparagus Soup with Poached Eggs
140530          Fried Oyster Po'boys
158475          Lamb shank tagine with herb tabbouleh
158486          Southern fried chicken in buttermilk
163175          Fried Chicken Sliders with Pickles + Slaw
165243          Bar Tartine Cauliflower Salad
Name: name, dtype: object
```

Now that we have narrowed-down our recipe selection by a factor of almost 20,000, we are in a position to make a more informed decision about what recipe we'd like!

Going Further with Recipes

I hope the above example gives you a bit of a flavor (ha!) for the types of data cleaning operations that are efficiently enabled by Pandas' string methods. Of course, to build a very robust recipe recommendation system would require a *lot* more work! Extracting full ingredient lists from each recipe would be an important piece of the task; unfortunately the wide variety formats used makes this a very difficult and time-consuming process. My goal with this example was not to do a complete, robust cleaning of the data, but to give you a taste (ha! again!) for how you might use these Pandas string tools in practice.

Working with Time Series

Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data. Date and time data comes in a few flavors, which we will discuss here:

- *Time stamps* reference particular moments in time; for example, July 4th, 2015 at 7:00am.
- *Time intervals* and *Periods* reference a length of time between a particular beginning and end point; for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (for example, 24 hour-long periods in a day).
- *Time deltas* or *Durations* reference an exact length of time; for example, a duration of 22.56 seconds.

In this section we will introduce how to work with each of these types of date/time data in Pandas. This short section is by no means a complete guide to the timeseries tools available in Python or Pandas, but instead is intended as a broad overview of how you as a user should approach working with timeseries. We will start with a brief discussion of tools for dealing with dates and times in Python, before moving more specifically to a discussion of the tools provided by Pandas. After listing some resour-

ces that go into more depth, we will go through some short examples of working with timeseries data in Pandas.

Dates and Times in Python

The Python world has a number of 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 quickly perform a host of useful functionalities on dates and times. For example, you can manually build a date using the `datetime` type:

```
from datetime import datetime
datetime(year=2015, month=7, day=4)

datetime.datetime(2015, 7, 4, 0, 0)
```

Or, using the `dateutil` module, you can parse dates from a variety of string formats:

```
from dateutil import parser
date = parser.parse("4th of July, 2015")
date

datetime.datetime(2015, 7, 4, 0, 0)
```

Once you have a `datetime` object, you can do things like printing the day of the week:

```
date.strftime('%A')
'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 tools for working with the most migrane-inducing piece of timeseries data: time zones.

The power of `datetime` and `dateutil` lie 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 suboptimal compared to NumPy-style typed numerical arrays, lists of Python `datetime` objects 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 timeseries data type to NumPy. The `datetime64` dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly. The `datetime64` requires a very specific input format:

```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
array(datetime.date(2015, 7, 4), dtype='datetime64[D]')
```

Once we have this date formatted, however, we can quickly do vectorized operations on it:

```
date + np.arange(12)
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 can be 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, `datetime64` imposes a tradeoff 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:

```
np.datetime64('2015-07-04')
numpy.datetime64('2015-07-04')
```

Here is a minute-based datetime:

```
np.datetime64('2015-07-04 12:00')
numpy.datetime64('2015-07-04T12:00-0700')
```

(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 format codes; for example, here we'll force a nanosecond-based time:

```
np.datetime64('2015-07-04 12:59:59.50', 'ns')
numpy.datetime64('2015-07-04T12:59:59.500000000-0700')
```

The following table, drawn from the [NumPy datetime64 documentation](#), lists the available format codes along with the relative and absolute timespans that they can encode:

Code	Meaning	Time span (relative)	Time span (absolute)
Y	year	$\pm 9.2\text{e}18$ years	[9.2e18 BC, 9.2e18 AD]
M	month	$\pm 7.6\text{e}17$ years	[7.6e17 BC, 7.6e17 AD]
W	week	$\pm 1.7\text{e}17$ years	[1.7e17 BC, 1.7e17 AD]
D	day	$\pm 2.5\text{e}16$ years	[2.5e16 BC, 2.5e16 AD]
h	hour	$\pm 1.0\text{e}15$ years	[1.0e15 BC, 1.0e15 AD]
m	minute	$\pm 1.7\text{e}13$ years	[1.7e13 BC, 1.7e13 AD]
s	second	$\pm 2.9\text{e}12$ years	[2.9e9 BC, 2.9e9 AD]
ms	millisecond	$\pm 2.9\text{e}9$ years	[2.9e6 BC, 2.9e6 AD]
us	microsecond	$\pm 2.9\text{e}6$ 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 it can 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 deficiencies of the built-in Python `datetime` type, it lacks many of the convenient methods 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 above tools 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` which 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:

```
import pandas as pd
date = pd.to_datetime("4th of July, 2015")
date
```

```
Timestamp('2015-07-04 00:00:00')
date.strftime('%A')
'Saturday'
```

Additionally, we can do NumPy-style vectorized operations directly on this same object:

```
date + pd.to_timedelta(np.arange(12), 'D')

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, tz=None)
```

Below we will take a closer look at manipulating timeseries data with these tools provided by Pandas.

Pandas TimeSeries: Indexing by Time

Where the Pandas timeseries tools really become useful is when you begin to *index data by timestamps*. For example, we can construct a Series object which has time-indexed data:

```
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

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 Series indexing patterns we discussed in previous sections, passing values which can be coerced into dates:

```
data['2014-07-04':'2015-07-04']

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:

```
data['2015']

2015-07-04    2
2015-08-04    3
dtype: int64
```

We will see further examples below of the convenience of dates-as-indices. But first, a closer look at the available TimeSeries data structures.

Pandas TimeSeries 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 above, it is essentially a replacement for Python's native `datetime`, but is based on the more efficient `numpy.datetime64` data type. The associated Index structure is **DatetimeIndex**.
- For *Time Periods*, Pandas provides the **Period** type. This encodes a fixed-frequency interval based on `numpy.datetime64`. The associated index structure is **PeriodIndex**.
- For *Time deltas or Durations*, Pandas provides the **Timedelta** type. Timedelta is a more 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. Passing a single date to `pd.to_datetime()` yields a `Timestamp`; passing a series of dates by default yields a `DatetimeIndex`:

```
dates = pd.to_datetime([datetime(2015, 7, 3), '4th of July, 2015',
                      '2015-Jul-6', '07-07-2015', '20150708'])

dates
DatetimeIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
                '2015-07-08'],
               dtype='datetime64[ns]', freq=None, tz=None)
```

Any `DatetimeIndex` can be converted to a `PeriodIndex` with the `to_period()` function with the addition of a frequency code; here we'll use '`D`' to indicate daily frequency:

```
dates.to_period('D')
PeriodIndex(['2015-07-03', '2015-07-04', '2015-07-06', '2015-07-07',
            '2015-07-08'],
            dtype='int64', freq='D')
```

A `TimedeltaIndex` is created, for example, when a date is subtracted from another:

```
dates - dates[0]
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()` for periods, 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 stepsize into a sequence. Similarly, `pd.date_range()` accepts a start date, an end date, and an optional frequency code to create a regular sequence of dates. By default, the frequency is one day:

```
pd.date_range('2015-07-03', '2015-07-10')

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', tz=None)
```

Optionally, the date range can be specified not with a start end end-point, but with a start-point and a number of periods:

```
pd.date_range('2015-07-03', periods=8)

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', tz=None)
```

The spacing can be modified by altering the `freq` argument, which defaults to D. For example, here we will construct a range of hourly timestamps:

```
pd.date_range('2015-07-03', periods=8, freq='H')

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', tz=None)
```

For more discussion of frequency options, see the Frequency Codes section below.

To create regular sequences of Period or Timedelta values, the very similar `pd.period_range()` and `pd.timedelta_range()` functions are useful. Here are some monthly periods:

```
pd.period_range('2015-07', periods=8, freq='M')

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:

```
pd.timedelta_range(0, periods=10, freq='H')

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 understanding of Pandas frequency codes, which we'll summarize in the next section.

Frequencies and Offsets

Fundamental to these Pandas Timeseries tools is the concept of a frequency or date offset. Just as we saw the "D" (day) and "H" (hour) codes above, we can use such codes to specify any desired frequency spacing. Following is a summary of the main codes available:

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	milliseconds		
U	microseconds		
N	nanoseconds	.	

The monthly, quarterly, annual frequencies are all marked at the end of the specified period. By adding an "S" suffix to any of these, they instead will be marked at the beginning:

Code	Description	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 code by 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, the split-point of the weekly frequency can be modified 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. For example, for a frequency of 2 hours 30 minutes, we can combine the hour ("H") and minute ("T") codes as follows:

```
pd.timedelta_range(0, periods=9, freq="2H30T")
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, which can be found in the `pd.tseries.offsets` module. For example, we can create a business day offset directly as follows:

```
from pandas.tseries.offsets import BDay
pd.date_range('2015-07-01', periods=5, freq=BDay())
DatetimeIndex(['2015-07-01', '2015-07-02', '2015-07-03', '2015-07-06',
                '2015-07-07'],
               dtype='datetime64[ns]', freq='B', tz=None)
```

For more discussion of the use of frequencies and offsets, see the Pandas online [DateOffset 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.) certainly apply, but Pandas also provides several timeseries-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 specific tools for financial data. For example, Pandas has built-in tool for reading available financial indices, the `DataReader` function. This function knows how to import financial data from a number of available sources, including Yahoo finance, Google Finance, and others. Here we will load Google's closing price history using Pandas:

```
from pandas.io.data import DataReader
goog = DataReader('GOOG', start='2004', end='2015',
                  data_source='google')
goog.head()
```

	Open	High	Low	Close	Volume
Date					
2004-08-19	49.96	51.98	47.93	50.12	NaN

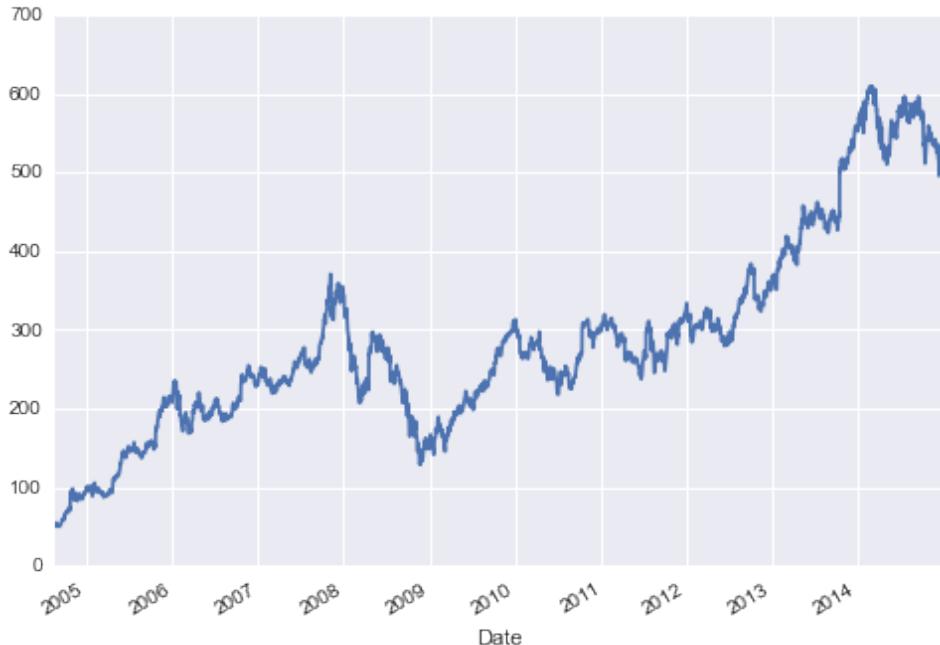
	Open	High	Low	Close	Volume
Date					
2004-08-20	50.69	54.49	50.20	54.10	NaN
2004-08-23	55.32	56.68	54.47	54.65	NaN
2004-08-24	55.56	55.74	51.73	52.38	NaN
2004-08-25	52.43	53.95	51.89	52.95	NaN

for simplicity, we'll use just the closing price:

```
goog = goog['Close']
```

We can visualize this using the `plot()` method, after the normal matplotlib setup boilerplate:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
goog.plot();
```



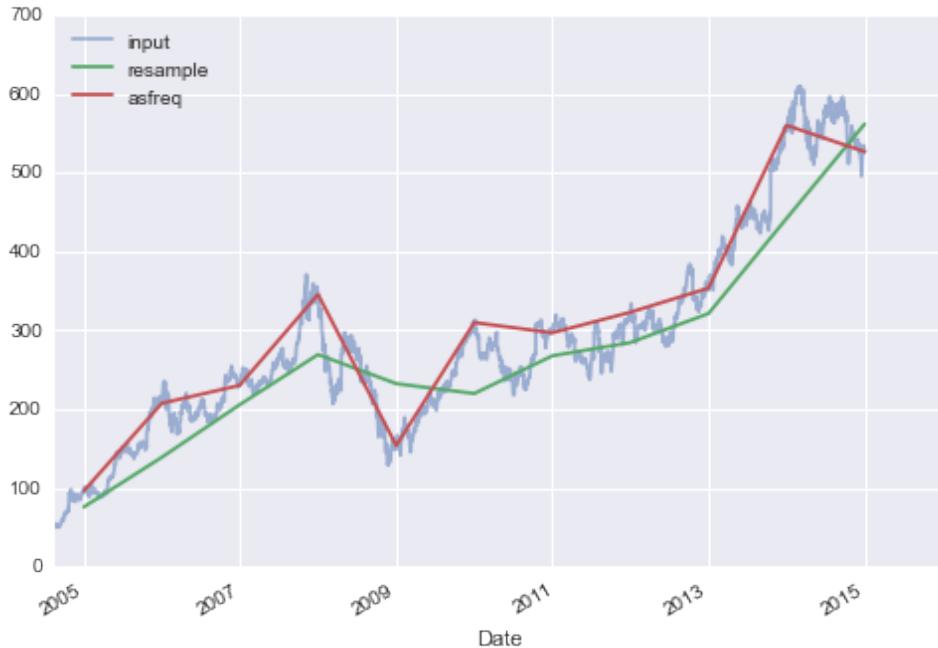
Resampling and Converting Frequencies

One common need for time series data is resampling at a higher or lower frequency. This can be done using the `resample()` method, or the much simpler `asfreq()`

method. The primary difference between the two is that `resample()` is fundamentally a *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:

```
goog.plot(alpha=0.5)
goog.resample('BA', how='mean').plot()
goog.asfreq('BA').plot();
plt.legend(['input', 'resample', 'asfreq'],
          loc='upper left');
```



Notice the difference: at each point, `resample` reports the *average of the previous year*, while `asfreq` reports the *value at the end of the year*.

For up-sampling, `resample()` and `asfreq()` are largely equivalent, though `resample` has many more options available. In this case, the default for both methods is to leave the up-sampled points empty, that is, filled with NA values. Just as with the `pd.fillna()` function discussed previously, `asfreq()` accepts a `method` argument to specify how values are imputed. Here, we will resample the business day data at a daily frequency (i.e. including weekends):

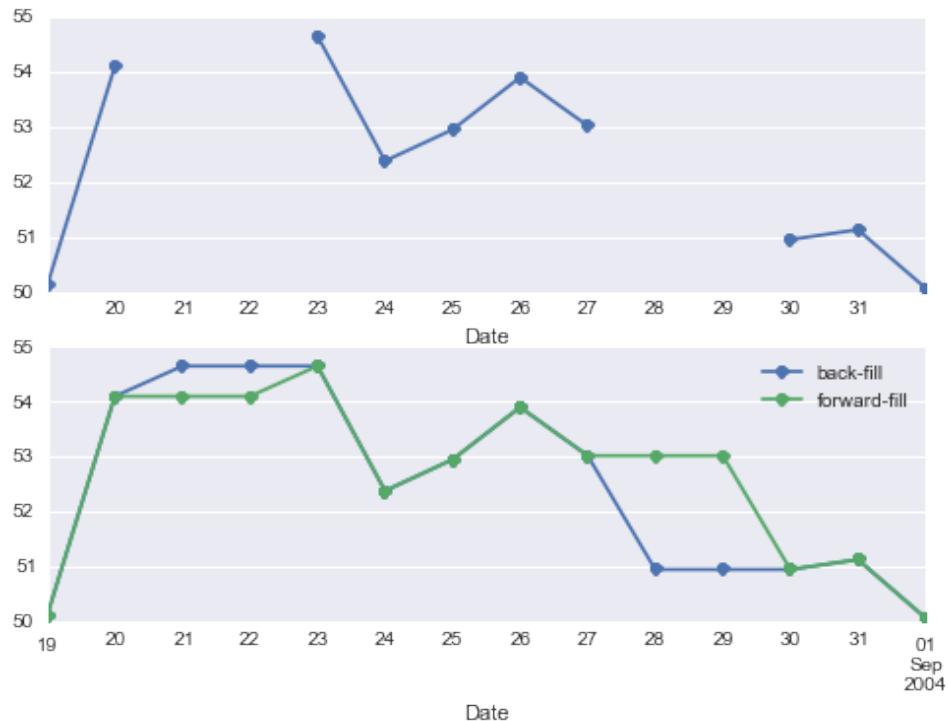
```
fig, ax = plt.subplots(2, sharex=True)
data = goog.iloc[:10]

data.asfreq('D').plot(ax=ax[0], marker='o')
```

```

data.asfreq('D', method='bfill').plot(ax=ax[1], marker='o')
data.asfreq('D', method='ffill').plot(ax=ax[1], marker='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 strategies for filling the gaps: forward-filling and backward-filling.

Time-shifts

Another common timeseries-specific operation is shifting of data in time. Pandas has 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 1000 days;

```

fig, ax = plt.subplots(3, sharey=True)

# apply a frequency to the data
goog = goog.asfreq('D', method='pad')

```

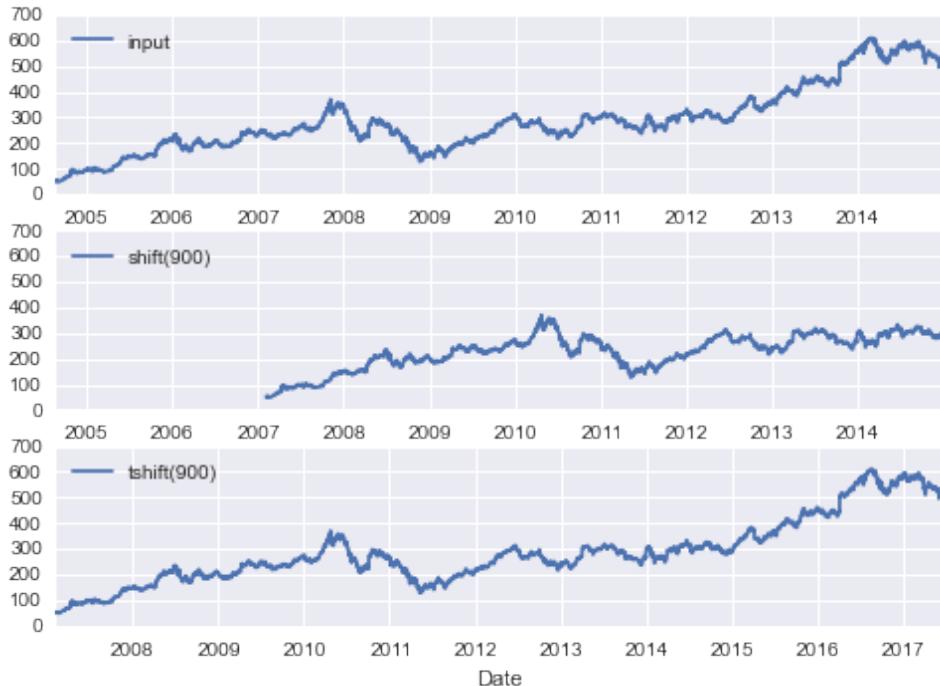
```

goog.plot(ax=ax[0])
ax[0].legend(['input'], loc=2)

goog.shift(900).plot(ax=ax[1])
ax[1].legend(['shift(900)'], loc=2)

goog.tshift(900).plot(ax=ax[2])
ax[2].legend(["tshift(900)"], loc=2);

```



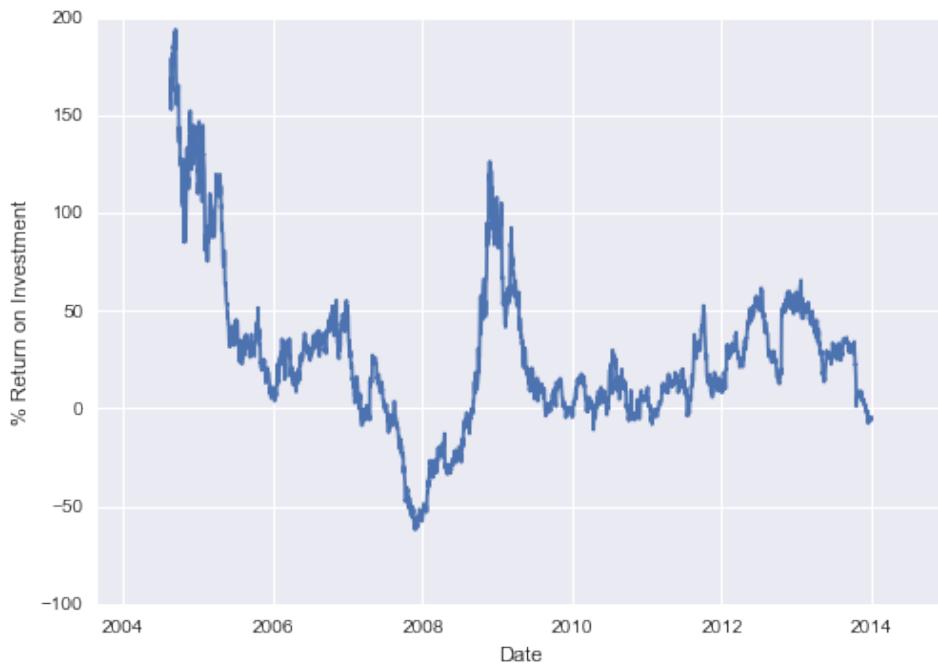
We see here visually that the `shift(900)` shifts the data by 900 days, pushing some of it off the end of the graph (and leaving NA values at the other end). On the other hand, carefully examining the x labels, we see that `tshift(900)` leaves the data in place while shifting the time index itself by 900 days.

A common context for this type of shift is in 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:

```

ROI = 100 * (goog.tshift(-365) / goog - 1)
ROI.plot()
plt.ylabel('% Return on Investment');

```



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 timeseries-specific operation implemented by Pandas. These can be accomplished via one of several functions, such as `pd.rolling_mean()`, `pd.rolling_sum()`, `pd.rolling_min()`, etc. Just about every pandas aggregation function (see Section X.X) has an associated rolling function.

The syntax of all of these is very similar: for example, here is the one-year centered rolling mean of the Google stock prices:

```
rmean = pd.rolling_mean(goog, 365, freq='D', center=True)
rstd = pd.rolling_std(goog, 365, freq='D', center=True)

data = pd.DataFrame({'input': goog, 'one-year rolling_mean': rmean, 'one-year rolling_std': rstd})
ax = data.plot()
ax.lines[0].set_alpha(0.3)
```



Along with the rolling versions of standard aggregates, there are also the more flexible functions `pd.rolling_window()` and `pd.rolling_apply()`. For details, see the documentation of these functions, or the example below.

Where to Learn More

The above is only a brief summary of some of the most essential features of time series tools provided by Pandas; for a more complete discussion you can refer to [Pandas Time Series Documentation](#).

Another excellent resource is the textbook [Python for Data Analysis](#) by Wes McKinney (O'Reilly, 2012). Though it is now a few years old, it is an invaluable resource on the 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 calendars, time zones, and related topics.

As usual, you can also use the IPython help functionality to explore and try further options available to the functions and methods discussed above: 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 bicycle 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 2015, the CSV can be downloaded as follows:

```
# !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:

```
data = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
data.head()
```

	Fremont Bridge West Sidewalk	Fremont Bridge East Sidewalk
Date		
2012-10-03 00:00:00	4	9
2012-10-03 01:00:00	4	6
2012-10-03 02:00:00	1	1
2012-10-03 03:00:00	2	3
2012-10-03 04:00:00	6	1

For convenience, we'll further process this dataset by shortening the column names and adding a "Total" column:

```
data.columns = ['West', 'East']
data['Total'] = data.eval('West + East')
```

Now let's take a look at the summary statistics for this data:

```
data.describe()
```

	West	East	Total
count	24017.000000	24017.000000	24017.000000
mean	55.452180	52.088646	107.540825
std	70.721848	74.615127	131.327728
min	0.000000	0.000000	0.000000
25%	7.000000	7.000000	16.000000
50%	31.000000	27.000000	62.000000

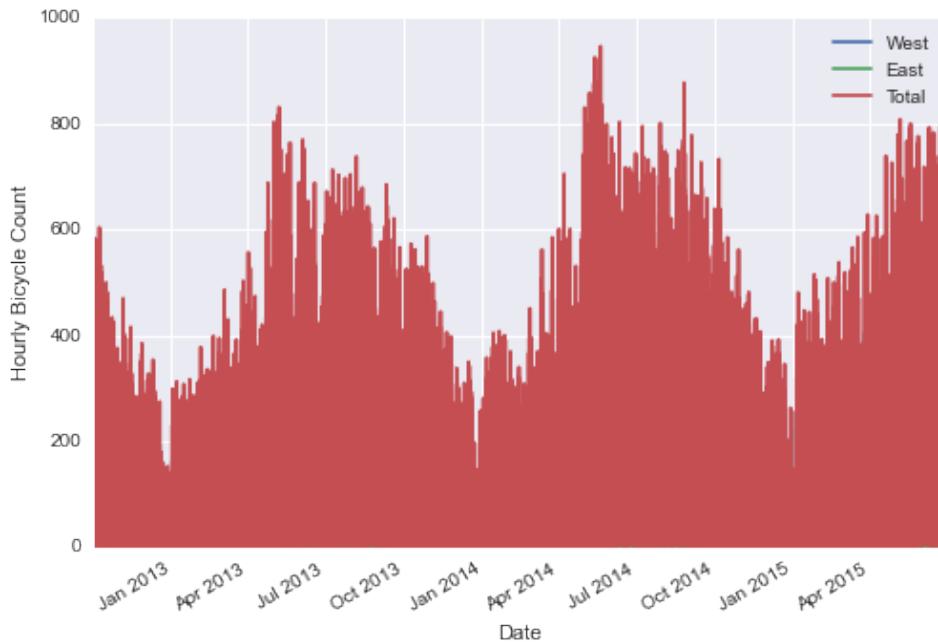
	West	East	Total
75%	74.000000	65.000000	143.000000
max	698.000000	667.000000	946.000000

Visualizing the Data

We can gain some insight into the dataset by visualizing it. Let's start by plotting the raw data:

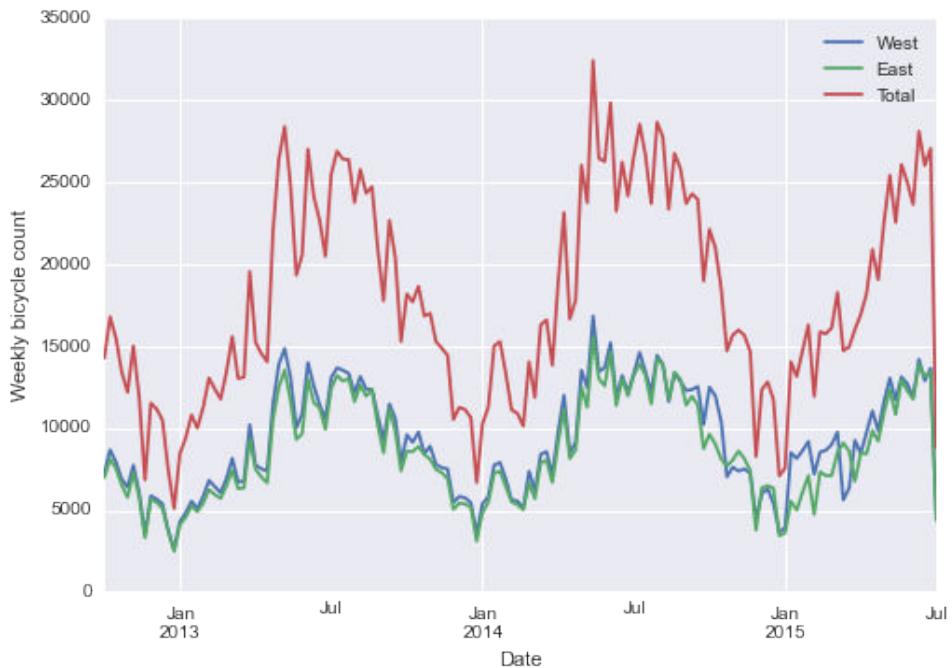
```
%matplotlib inline
import seaborn; seaborn.set()

data.plot()
plt.ylabel('Hourly Bicycle Count');
```



The ~25000 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:

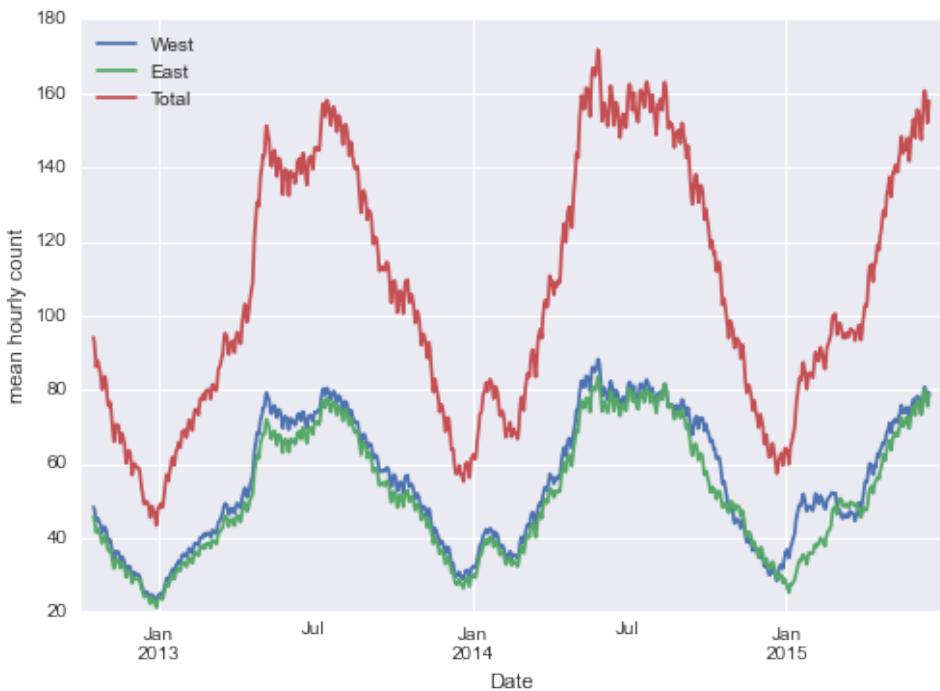
```
data.resample('W', how='sum').plot()
plt.ylabel('Weekly bicycle count');
```



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 (likely dependent on weather; see Section X.X where we explore this further).

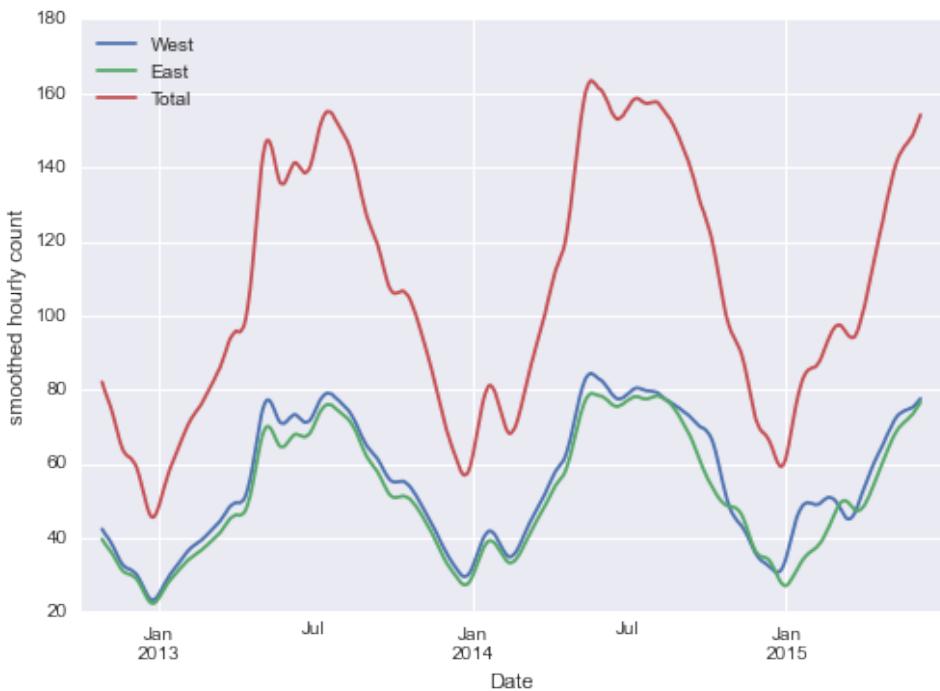
Another useful way to aggregate the data is to use a rolling mean, using the `pd.rolling_mean()` function. Here we'll do a 30 day rolling mean of our data, making sure to center the window:

```
pd.rolling_mean(data, 30, freq='D', center=True).plot()  
plt.ylabel('mean hourly count');
```



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 Gaussian window. Here we need to specify both the width of the window (we choose 50 days) and the width of the Gaussian within the window (we choose 10 days):

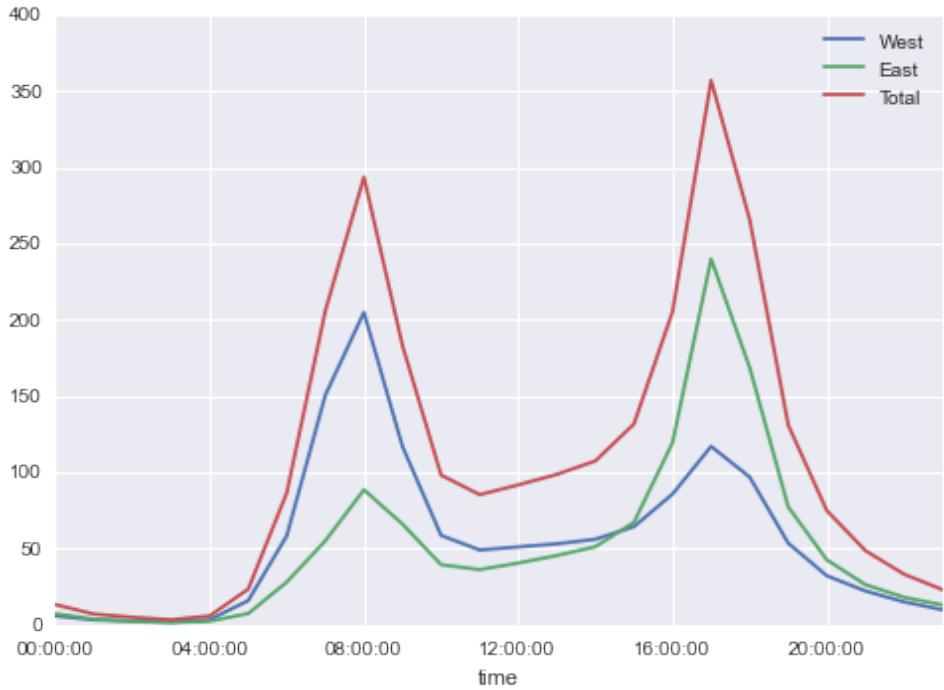
```
pd.rolling_window(data, 50, freq='D', center=True,
                   win_type='gaussian', std=10).plot()
plt.ylabel('smoothed hourly count')
<matplotlib.text.Text at 0x10c8d9250>
```



Digging Into the Data

While these smoothed data views 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 Section X.X:

```
by_time = data.groupby(data.index.time).mean()  
hourly_ticks = 4 * 60 * 60 * np.arange(6)  
by_time.plot(xticks=hourly_ticks);
```



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.

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:

```
by_weekday = data.groupby(data.index.dayofweek).mean()
by_weekday.index = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun']
by_weekday.plot();
```



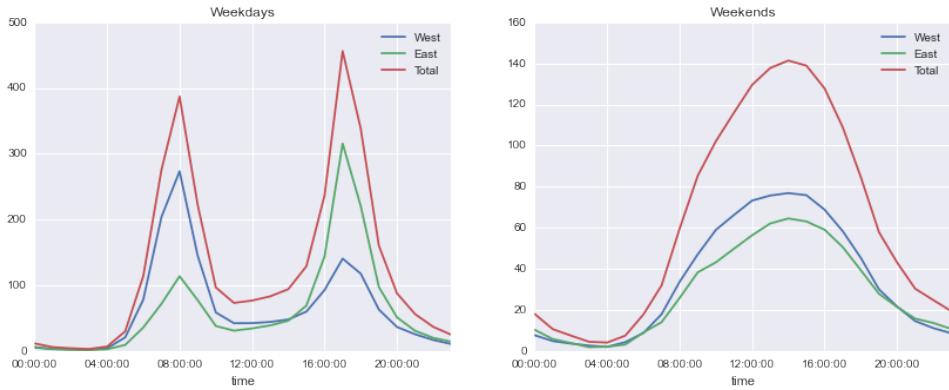
This shows a strong distinction between weekday and weekend totals, with around twice as many average riders crossing the bridge on Monday-Friday than on Saturday and Sunday.

With this in mind, let's do a compound GroupBy and look at the hourly trend on weekdays vs weekends. We'll start by grouping by both a flag marking the weekend, and the time of day:

```
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 Section X.X to plot two panels side-by-side:

```
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)
by_time.ix['Weekend'].plot(ax=ax[1], title='Weekends', xticks=hourly_ticks);
```



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, etc. on people's commuting patterns. I did this a bit in a blog post using a subset of this data; you can find that discussion [on my blog](#). We will also revisit this dataset in the context of modeling in Section X.X.

High-Performance Pandas: eval() and query()

As we've seen in the previous chapters, the power of the PyData stack lies in the ability of NumPy and Pandas to push basic operations into C via an intuitive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use-cases, they often rely on the creation of temporary intermediate objects which can cause undue overhead in computational time and memory use. Many of the Python performance solutions explored in chapter X.X are designed to address these deficiencies, and we'll explore these in more detail at that point.

As of version 0.13 (released January 2014), Pandas includes some experimental tools which allow you to directly access C-speed operations without costly allocation of intermediate arrays. These are the `eval()` and `query()` functions, which rely on the `numexpr` package (discussed more fully in section X.X). 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; for example, when adding the elements of two arrays:

```
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: 3.57 ms per loop
```

As discussed in Section X.X, this is much faster than doing the addition via a Python loop or comprehension

```
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)), dtype=x.dtype, count=len(x))

1 loops, best of 3: 232 ms per loop
```

But this abstraction can become less efficient when computing compound expressions. For example, consider the following expression:

```
mask = (x > 0.5) & (y < 0.5)
```

Because NumPy evaluates each subexpression, this is roughly equivalent to the following

```
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 overhead. 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. More details on numexpr are given in section X.X, but for the time being it is sufficient to say that the library accepts a *string* giving the NumPy-style expression you'd like to compute:

```
import numexpr
mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
np.allclose(mask, mask_numexpr)

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, especially for large arrays.

The Pandas eval() and query() tools discussed below 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 operations using dataframes. For example, consider the following dataframes:

```

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 can just write the sum:

```

%timeit df1 + df2 + df3 + df4
10 loops, best of 3: 88.6 ms per loop

```

The same result can be computed via `pd.eval` by constructing the expression as a string:

```

%timeit pd.eval('df1 + df2 + df3 + df4')
10 loops, best of 3: 42.4 ms per loop

```

The `eval()` version of this expression is about 50% faster (and uses much less memory), while giving the same result:

```

np.allclose(df1 + df2 + df3 + df4,
            pd.eval('df1 + df2 + df3 + df4'))
True

```

Operations Supported by `pd.eval()`

As of Pandas v0.16, `pd.eval()` supports a wide range of operations. To demonstrate these, we'll use the following integer dataframes:

```

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; e.g.

```

result1 = -df1 * df2 / (df3 + df4) - df5
result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
np.allclose(result1, result2)
True

```

Comparison Operators. `pd.eval()` supports all comparison operators, including chained expressions; e.g.

```

result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
result2 = pd.eval('df1 < df2 <= df3 != df4')
np.allclose(result1, result2)
True

```

Bitwise Operators. `pd.eval()` supports the & and | bitwise operators:

```
result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
np.allclose(result1, result2)

True
```

In addition, it supports the use of the literal `and` and `or` in boolean expressions:

```
result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
np.allclose(result1, result3)

True
```

Object Attributes and indices. `pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

```
result1 = df2.T[0] + df3.iloc[1]
result2 = pd.eval('df2.T[0] + df3.iloc[1]')
np.allclose(result1, result2)

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, you can use the `numexpr` library itself, discussed in Section X.X.

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 columns can be referred to **by name**. We'll use this labeled array as an example:

```
df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
df.head()
```

	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

Using `pd.eval()` as above, we can compute expressions with the three columns like this:

```

result1 = (df['A'] + df['B']) / (df['C'] - 1)
result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
np.allclose(result1, result2)

True

```

The DataFrame `eval()` method allows much more succinct evaluation of expressions with the columns:

```

result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)

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 discussed above, `DataFrame.eval()` also allows assignment to any column. Let's use the dataframe from above, which has columns 'A', 'B', and 'C':

```
df.head()
```

	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 computed from the other columns:

```

df.eval('D = (A + B) / C')
df.head()

```

	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:

```
df.eval('D = (A - B) / C')
df.head()
```

	A	B	C	D
0	0.375506	0.406939	0.069938	-0.449425
1	0.069087	0.235615	0.154374	-1.078728
2	0.677945	0.433839	0.652324	0.374209
3	0.264038	0.808055	0.347197	-1.566886
4	0.589161	0.252418	0.557789	0.603708

Local Variables in DataFrame.eval()

The DataFrame.eval() method supports an additional syntax which lets it work with local Python variables. Consider the following:

```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)

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:

```
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)

True
```

As with the example in DataFrame.eval() above, this is an expression involving columns of the dataframe. It cannot, however, be expressed using the DataFrame.eval() syntax! Instead, for this type of filtering operation, you can use the query() method:

```
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)

True
```

In addition to being a more efficient computation, compared to the masking expression this is much easier to read and understand. Note that the `query()` method also accepts the `@` flag to mark local variables:

```
Cmean = df['C'].mean()  
result1 = df[(df.A < Cmean) & (df.B < Cmean)]  
result2 = df.query('A < @Cmean and B < @Cmean')  
np.allclose(result1, result2)  
  
True
```

Performance: When to Use these functions

This is all well and good, but when should you use this functionality? There are two considerations here: *computation time* and *memory use*.

Memory use is the most predictable aspect. As mentioned above, every compound expression involving NumPy arrays or Pandas DataFrames will result in implicit creation of temporary arrays: For example, this:

```
x = df[(df.A < 0.5) & (df.B < 0.5)]
```

Is roughly equivalent to this:

```
tmp1 = df.A < 0.5  
tmp2 = df.B < 0.5  
tmp3 = tmp1 & tmp2  
x = df[tmp3]
```

If the size of the temporary DataFrames is significant compared to your available system memory (typically a few gigabytes in 2015) then it's a good idea to use an `eval()` or `query()` expression. You can check the approximate size of your array in bytes using, e.g.

```
df.values.nbytes  
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 2015); if they are much bigger then `eval()` can avoid some potentially slow movement of values between the different memory caches.

For a simple benchmark of these two, we can use `%timeit` to see the performance breakdown. The following cells will take a few minutes to run, as they automatically repeat the calculations in order to remove system timing variations.

```
sizes = (10 ** np.linspace(3, 7, 7)).astype(int)  
times_eval_p = np.zeros_like(sizes, dtype=float)  
times_eval = np.zeros_like(sizes, dtype=float)  
x = rng.rand(4, max(sizes), 2)
```

```

for i, size in enumerate(sizes):
    rng = np.random.RandomState(0)
    df1, df2, df3, df4 = (pd.DataFrame(x[i, :size])
                           for i in range(4))
    t = %timeit -oq df1 + df2 + df3 + df4
    times_eval_p[i] = t.best
    t = %timeit -oq pd.eval('df1 + df2 + df3 + df4')
    times_eval[i] = t.best

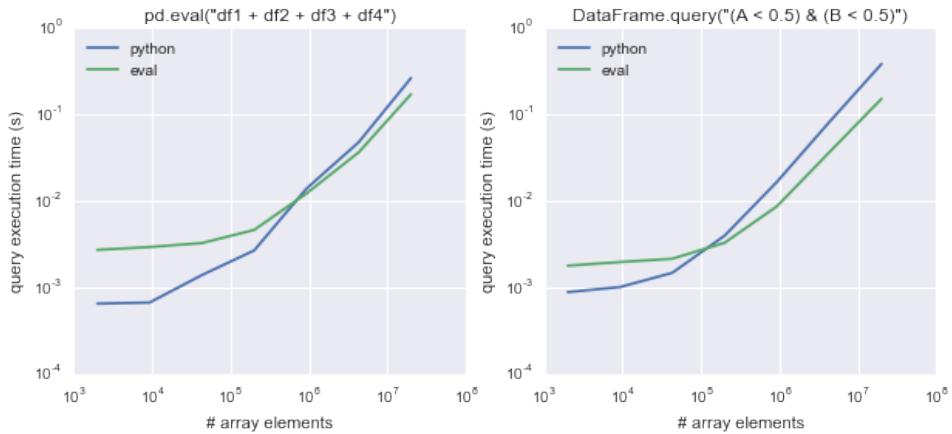
times_query_p = np.zeros_like(sizes, dtype=float)
times_query = np.zeros_like(sizes, dtype=float)
x = rng.rand(max(sizes), 2)

for i, size in enumerate(sizes):
    rng = np.random.RandomState(0)
    df1 = pd.DataFrame(x[:size], columns=['A', 'B'])
    t = %timeit -oq df1[(df1.A < 0.5) & (df1.B < 0.5)]
    times_query_p[i] = t.best
    t = %timeit -oq df1.query('(A < 0.5) & (B < 0.5)')
    times_query[i] = t.best

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].loglog(2 * sizes, times_eval_p, label='python')
ax[0].loglog(2 * sizes, times_eval, label='eval')
ax[0].legend(loc='upper left')
ax[0].set(xlabel='# array elements',
           ylabel='query execution time (s)',
           title='pd.eval("df1 + df2 + df3 + df4")')

ax[1].loglog(2 * sizes, times_query_p, label='python')
ax[1].loglog(2 * sizes, times_query, label='eval')
ax[1].legend(loc='upper left')
ax[1].set(xlabel='# array elements',
           ylabel='query execution time (s)',
           title='DataFrame.query("(A < 0.5) & (B < 0.5)")');

```



We see that on my machine, the simple Python expression is faster for arrays with fewer than roughly 10^5 or 10^6 elements (though it will use more memory), and the `eval()`/`query()` approach is faster for arrays much larger than this. Keep this in mind as you decide how to optimize your own code!

Learning More

We've covered most of the details of `eval()` and `query()` here; for more information on 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 discussion within the [Enhancing Performance](#) section of the documentation. Regarding more general use of this `numexpr` string-to-compiled-code interface, refer to Section X.X, where we discuss the NumExpr package in more detail.

Further Resources

In this chapter we've covered many of the basics of using Pandas effectively for data analysis. Still, much has been left out. To learn more about Pandas, I recommend some of the following resources:

- **Pandas Online Documentation**: this is the go-to source for complete documentation of the package. While the examples in the documentation tend to be small generated datasets, the description of the options is complete and generally very useful for understanding the use of various functions.
- **Python for Data Analysis** by Wes McKinney (O'Reilly, 2012). Wes is the creator of Pandas, and his book contains much more detail on the Pandas package than we had room for in this chapter. In particular, he takes a deep dive into tools for TimeSeries, which were his bread and butter as a financial consultant. The book

also has many entertaining examples of applying Pandas to gain insight from real-world datasets. Keep in mind, though, that the book is now several years old, and the Pandas package has quite a few new features that this book does not cover.

- **StackOverflow Pandas:** Pandas has so many users that any question you have has likely been asked and answered on StackOverflow. Using Pandas is a case where some Google-Fu is your best friend. Simply type-in to your favorite search engine the question, problem, or error you're coming across, and more than likely you'll find your answer on a StackOverflow page.
- **Pandas on PyVideo:** From PyCon to SciPy to PyData, many conferences have featured Pandas tutorials from Pandas developers and power-users. The PyCon tutorials in particular tend to be given by very well-vetted presenters.

Using the above resources, combined with the walk-through given in this section, my hope is that you'll be poised to use Pandas to tackle any data analysis problem you come across!

Introduction to Matplotlib

This chapter introduces the Matplotlib library for visualization in Python. Matplotlib is a multi-platform data visualization tool built upon the Numpy and Scipy framework. It was conceived by John Hunter in 2002, originally as a patch to IPython to enable interactive MatLab-style plotting via 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, and the 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 Telescope 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 user-base, which in turn has led to an active developer base and matplotlib's powerful tools and ubiquity within the scientific Python world.

In recent years, however, the decades-old interface and style of matplotlib have begun to show their age. Newer tools like ggplot and ggviz in the R language, along with web visualization toolkits based on D3js and HTML5 canvas, often make matplotlib feel clunky and old-fashioned. Still, I'm of the opinion that we cannot ignore matplotlib's strength as a well-tested, cross-platform graphics engine. Recent matplotlib versions make it relatively easy to set new global plotting styles, and people have been developing new packages which build on matplotlib's powerful internals to drive matplotlib via cleaner, more modern APIs. For this reason, I believe that Matplotlib itself will

remain a vital piece of the data visualization stack, even if new tools mean the community gradually moves away from using the Matplotlib API directly.

General matplotlib tips

Before we dive into the details of creating visualizations with Matplotlib, there are a few useful things you should know about using the package.

Importing Matplotlib

Just as we use the `np` shorthand for NumPy and the `pd` shorthand for Pandas, we will use some standard shorthands for matplotlib imports:

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

The `plt` interface is what we will use most often, as we shall see through this chapter.

show() or no show()? How to Display your Plots

It goes without saying that a visualization must be seen to be useful. There are several patterns for viewing your visualizations, and which one to use 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 which display your figure or figures.

So, for example, you may have a file called `myplot.py` which contains the following:

```
# ----- file: myplot.py -----
import matplotlib.pyplot as plt
import numpy 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:

```
$ python myplot.py
```

and this will result in a window opening with your figure. 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 section X.X). IPython is built to work well with Matplotlib if you specify matplotlib mode. To enable this mode, you can use the `%matplotlib` magic command after starting `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 properties 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

The IPython notebook is a browser-based interactive data analysis tool which can combine narrative, code, graphics, HTML elements, and much more into a single executable document (see section X.X).

Plotting interactively within an IPython notebook can be done with the `%matplotlib` command, and works similarly to the IPython shell, above. In the IPython notebook, you also have the option of embedding graphics directly in the notebook. This can be done by adding `inline` to the magic command:

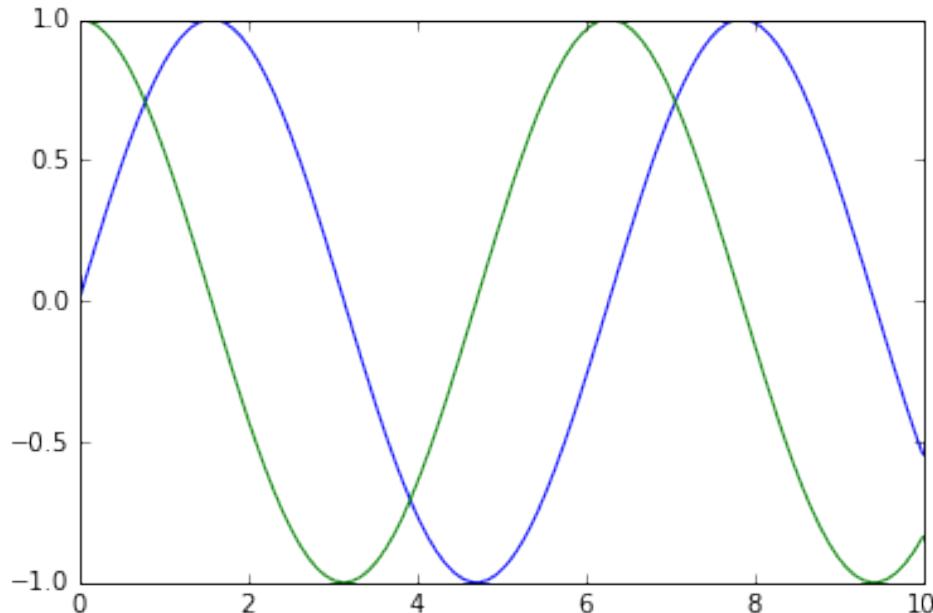
```
%matplotlib inline
```

After running this command (it needs to be done only once per kernel/session), any cell within the notebook which creates a plot will embed a png image of the resulting graphic:

```
import numpy as np
x = np.linspace(0, 10, 100)

fig = plt.figure()
```

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```



Saving Figures to File

One nice feature of matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the `savefig()` command. For example, to save the previous figure as a png file, you can run

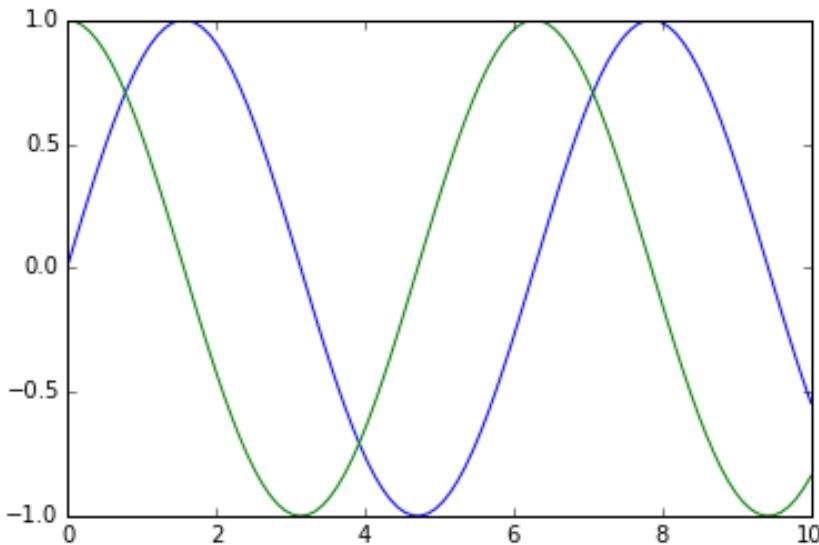
```
fig.savefig('my_figure.png')
```

We now have a file called `my_figure.png` in the current working directory:

```
!ls -lh my_figure.png
-rw-r--r-- 1 jakevdp staff 19K Jun 22 16:53 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:

```
from IPython.display import Image
Image('my_figure.png')
```



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. These can be found for your system by using the following method of the figure canvas object:

```
fig.canvas.get_supported_filetypes()  
  
{'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, you need not use `plt.show()` or related commands discussed above.

Learning More about `matplotlib`

A single chapter in a book can never hope to cover all the available features and plot types available in `matplotlib`. In addition to the help routines described in section

X.X, an extremely helpful reference is matplotlib's online documentation at <http://matplotlib.org/>. In particular see the Gallery linked on that page: this shows thumbnails of hundreds of different plot types, each one linked to a page with the Python code snippet used to generate it. In this way, you can visually inspect and learn about a wide range of different visualization techniques.

Sidebar: 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.

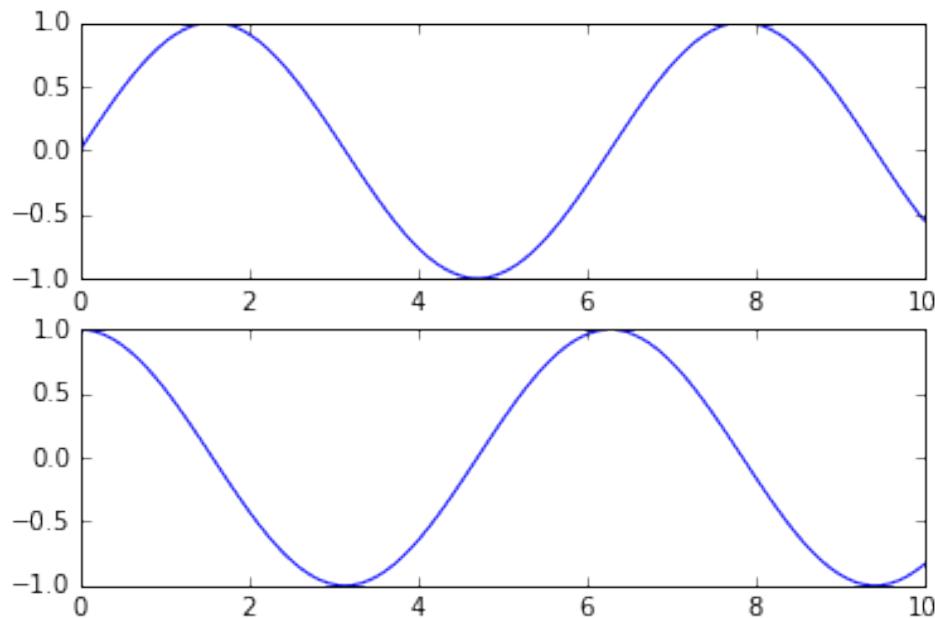
MatLab-style Interface

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. The pyplot interface is a state-based interface: that is, Matplotlib keeps track of a current axes object and runs commands on that. When a new axes object is created, that new object becomes current. The following code will probably look quite familiar to MatLab users:

```
plt.figure()

# create the first of 2 panels & set current axis
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x, np.sin(x))

# create the second panel & set current axis
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x));
```



This interface is *stateful*: it keeps track of the “current” axes and figure, which is where all `plt` commands are applied. You can get a reference to these using the `plt.gca()` (get current axes) and `plt.gcf()` (get current figure) routines.

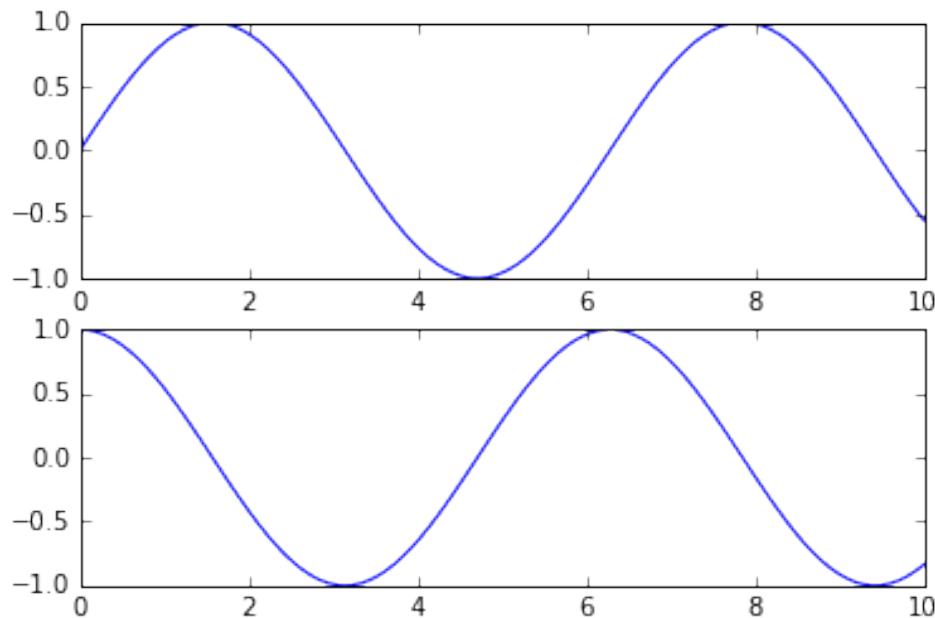
While this stateful interface is fast and convenient for simple plots, it is easy to run into problems. For example, once the second panel is created, how can we go back and add something to the first? It’s possible within the matlab-style interface, but there is a better way.

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 functions are *methods* of `Figure` and `Axes` objects. To recreate the above plot using this style of plotting, you might do the following:

```
# 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));
```



Which style to use is largely a matter of taste for more simple plots, 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.

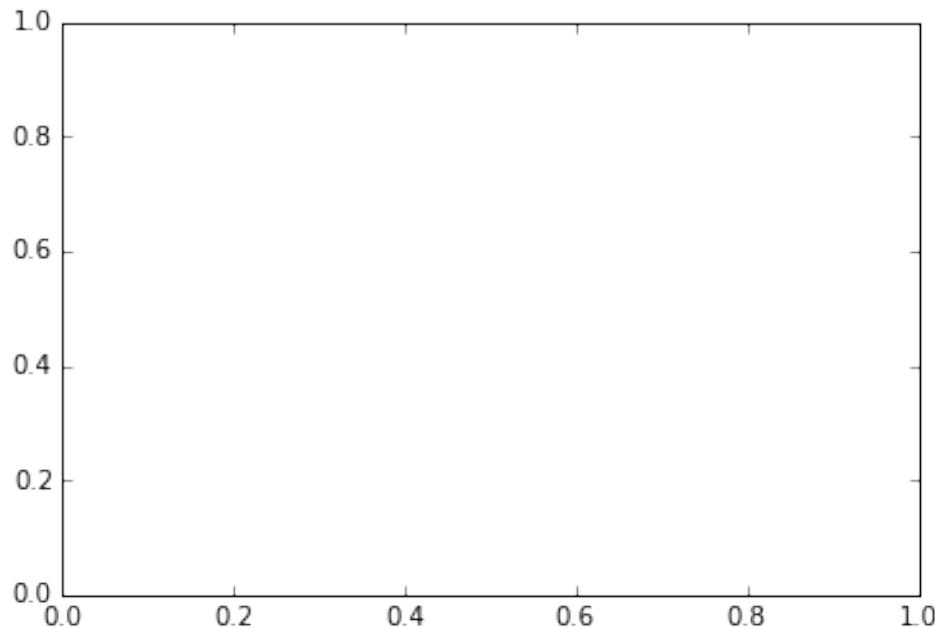
Simple Line Plots

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. For all matplotlib plots, we start by creating a figure and an axes. As with all the following sections, we'll start with setting up the notebook for plotting, and with importing the functions we will use:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

In their simplest form, a figure and axes can be created as follows:

```
fig = plt.figure()
ax = plt.axes()
```

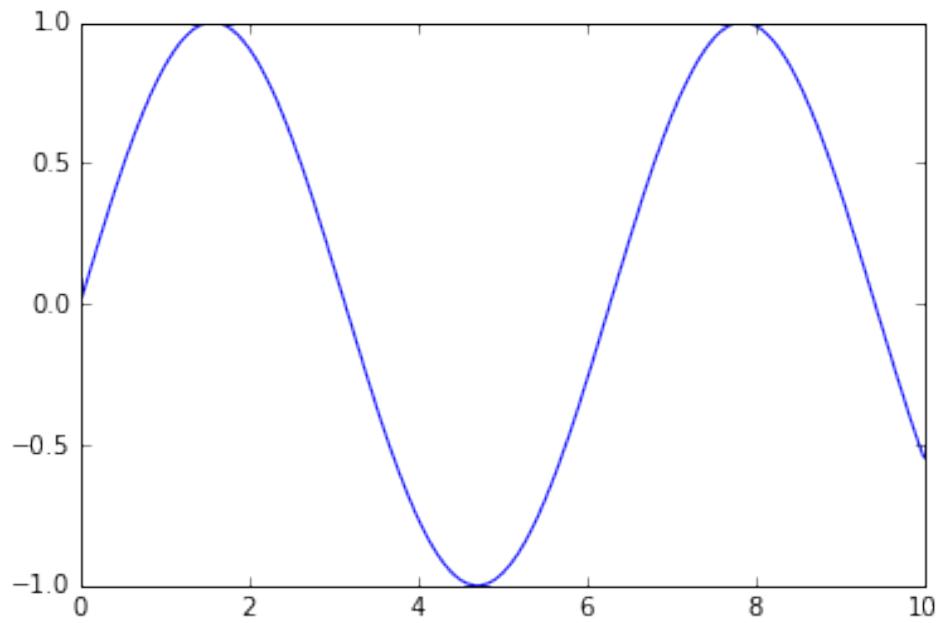


In matplotlib, the *figure* (an instance of the class `plt.Figure`) can be thought of as a single container which contains all the objects representing axes, graphics, text, labels, etc. The *axes* (an instance of the class `plt.Axes`) is what we see above: a bounding box with ticks and labels, which will eventually contain other plot elements. Through 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 set 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:

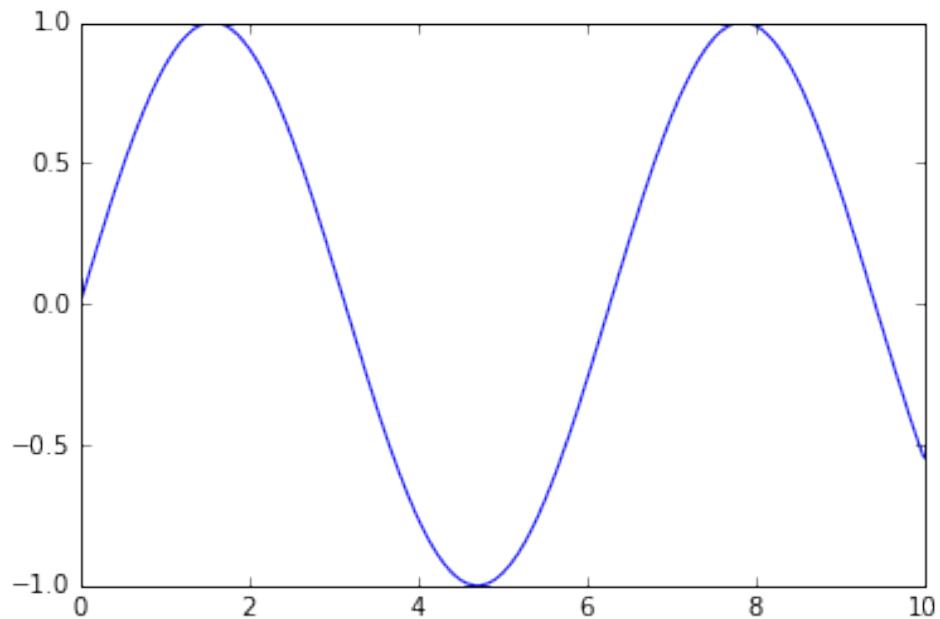
```
fig = plt.figure()
ax = plt.axes()

x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```



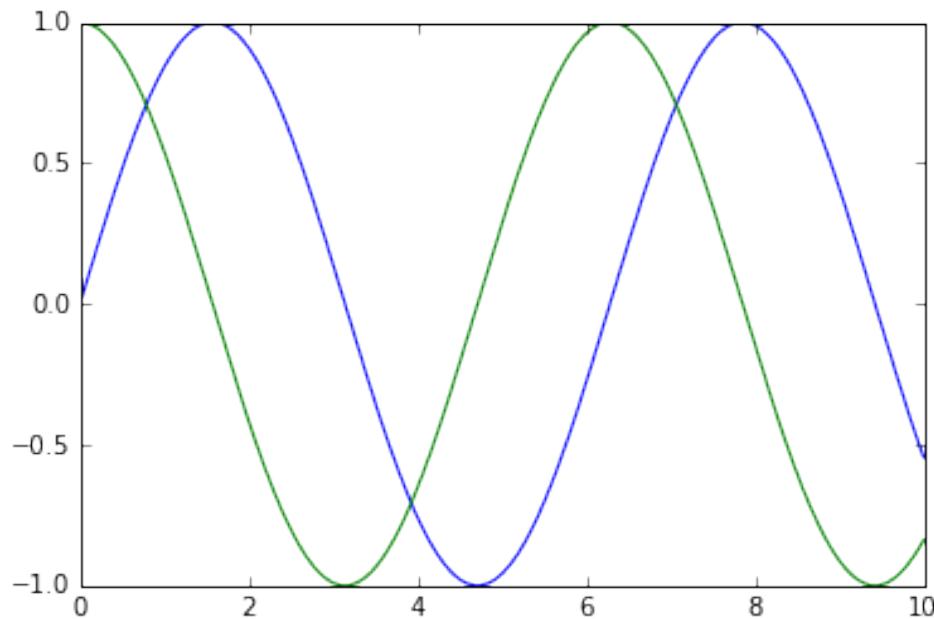
Alternatively, we can use the pylab interface and let the figure and axes be created for us in the background. (See the preface to this chapter for a discussion of these two interfaces):

```
plt.plot(x, np.sin(x));
```



If we want to create a single figure with multiple lines, we can simply call the `plot` function multiple times:

```
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

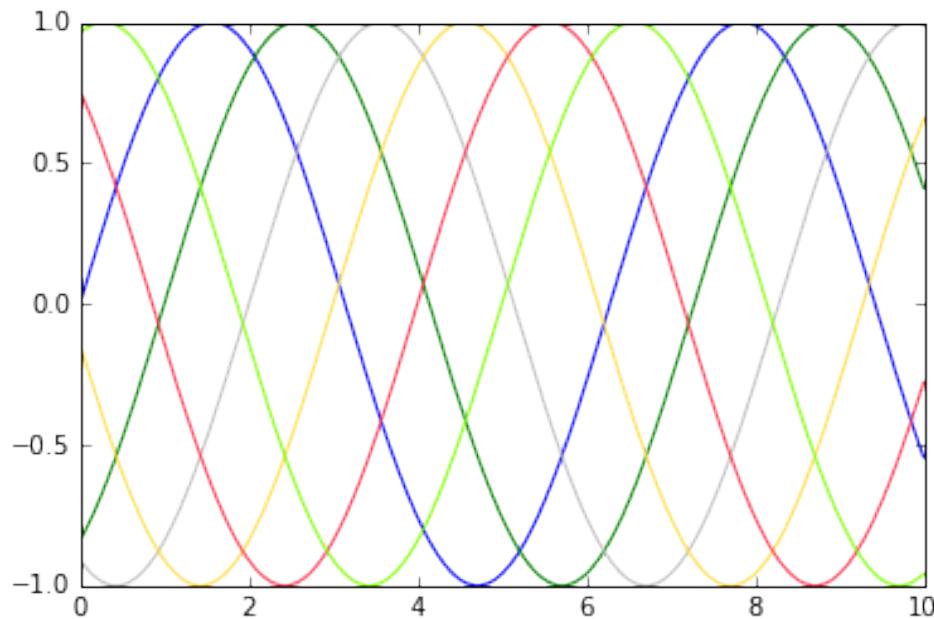


That's all there is to plotting simple functions in matplotlib! Below we'll dive into some more details about how to control the appearance of the axes and lines.

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 which can be used to specify 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, which we'll show below:

```
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 (works for rgb & cmyk)
plt.plot(x, np.sin(x - 2), color=0.75)         # Greyscale between 0 and 1
plt.plot(x, np.sin(x - 3), color="#FFDD44")     # Hex color code (RRGGBB from 00 to FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # RGB tuple, values between 0 and 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # all html color names are supported;
```

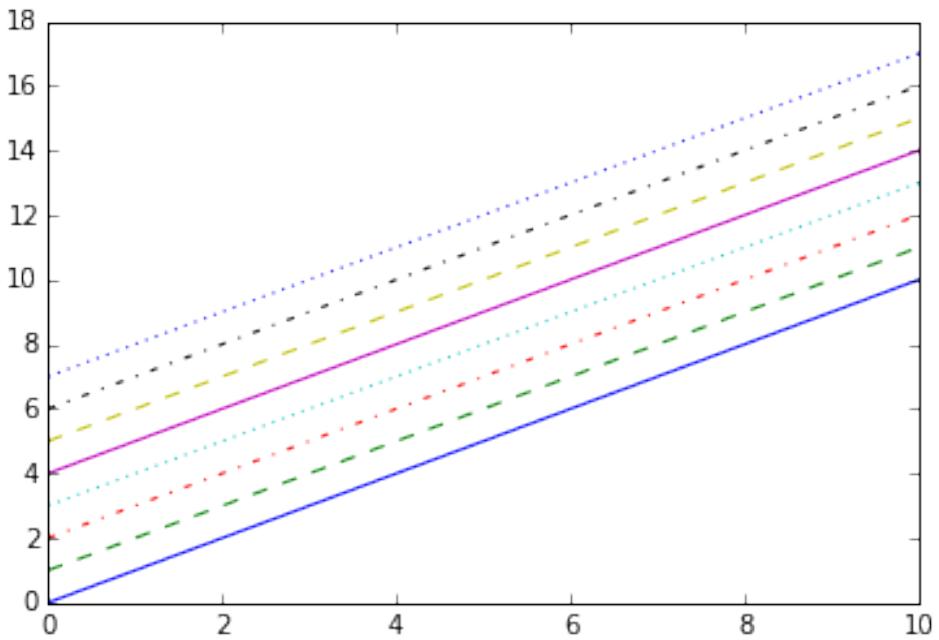


Note that if no color is specified, matplotlib will automatically cycle through a set of default colors for multiple lines.

Similarly, the line style can be adjusted using the `linestyle` keyword:

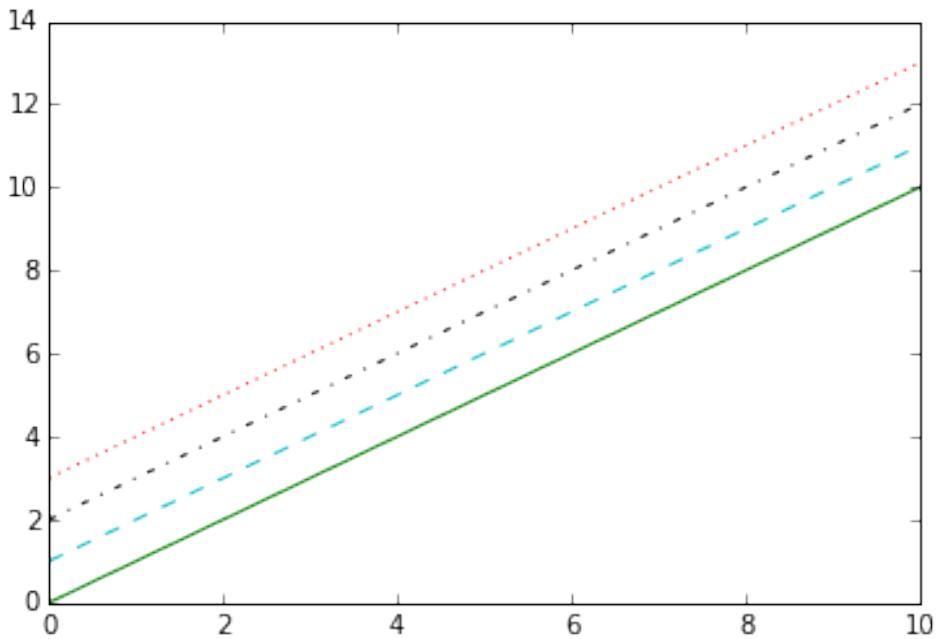
```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted');

# 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;
```



If you would like to be extremely terse, these linestyle codes and color codes can be combined into a single non-keyword argument to the `plt.plot()` function:

```
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;
```



These single-character color codes reflect the standard abbreviations in the RGB (Red/Green/Blue) and CMYK (Cyan/Magenta/Yellow/black) color systems, commonly used for digital color graphics.

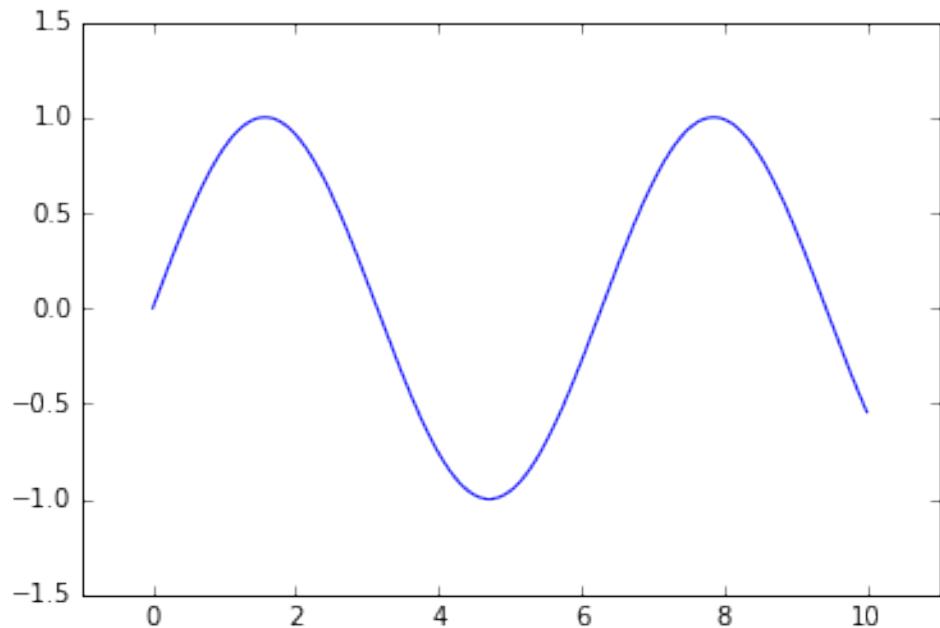
There are many other keyword arguments that can be used to fine-tune the appearance of the plot: for more details, refer to matplotlib's online documentation, or the docstring of the `plt.plot()` function.

Adjusting the Plot: Axes limits

Matplotlib does a fairly good job of choosing default axes limits for your plot, but sometimes it's nice to have finer control. Here we'll briefly see how to change the limits of the x and y axes. The most basic way to do this is to use the `plt.xlim()` and `plt.ylim()` methods to set the numerical limits of the x and y axes:

```
plt.plot(x, np.sin(x))

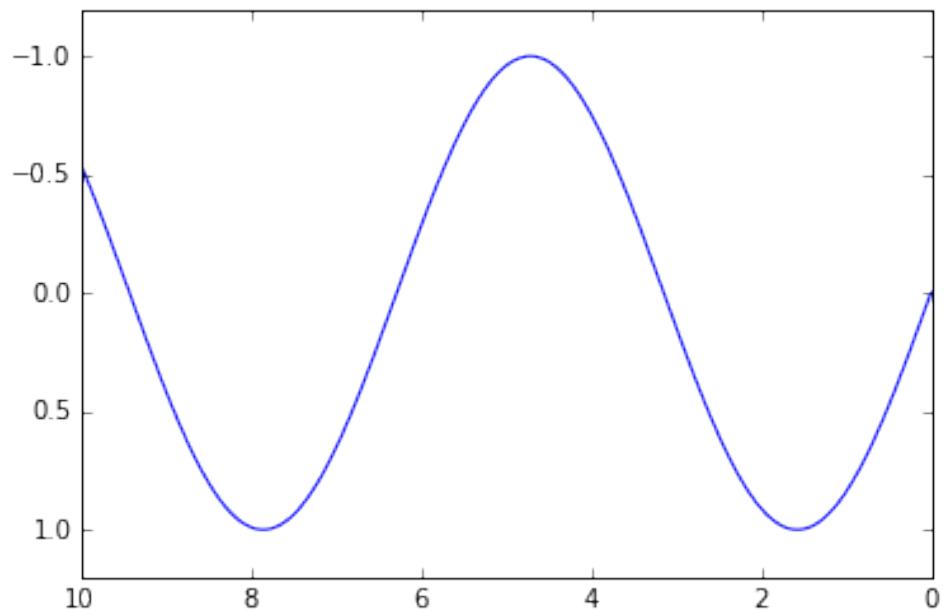
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```



If for some reason you'd like either axis to be displayed in reverse, you can simply reverse the order of the arguments:

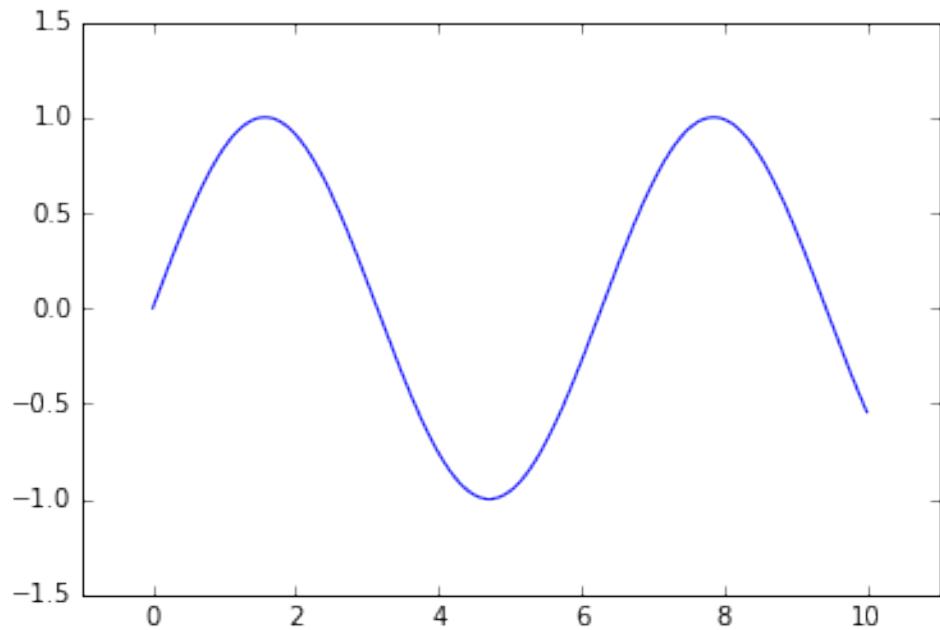
```
plt.plot(x, np.sin(x))

plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```



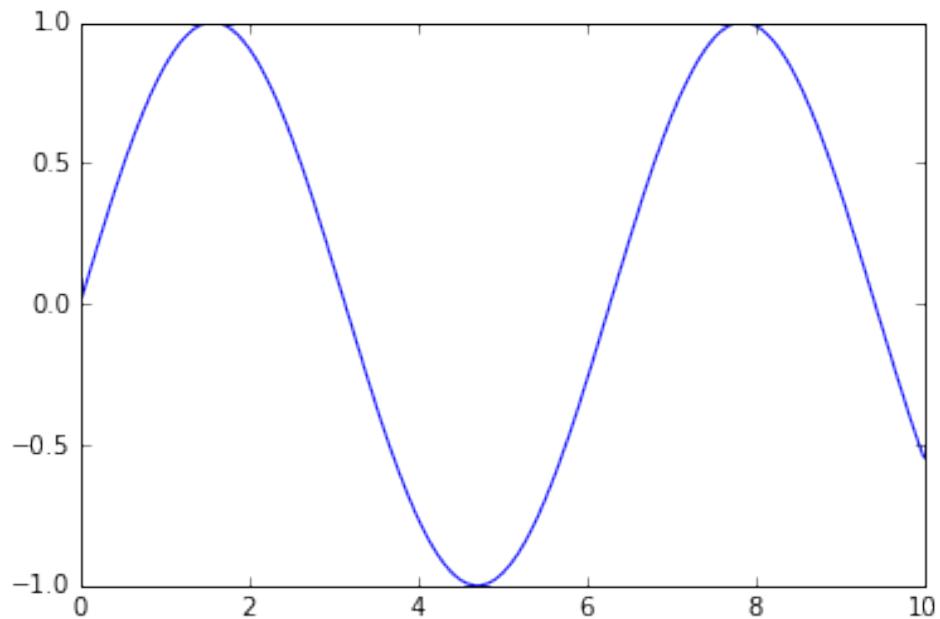
A useful related method is `plt.axis()` (note here the potential confusion between *axes* with an *e*, and *axis* with an *i*). The `plt.axis()` method allows you to set the x and y limits with a single call, by passing a list which specifies [`xmin`, `xmax`, `ymin`, `ymax`]:

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```



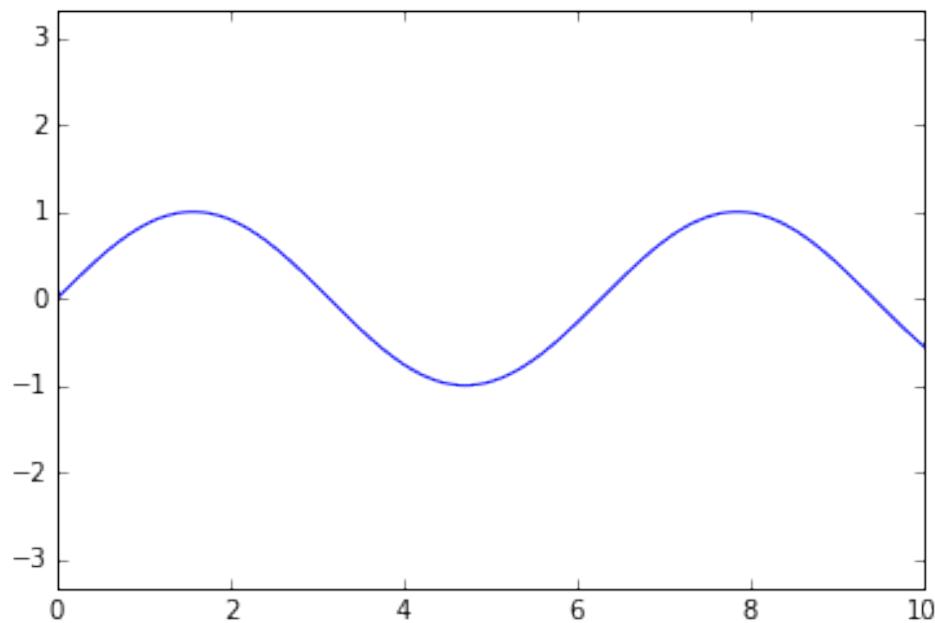
The `plt.axis()` method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot:

```
plt.plot(x, np.sin(x))
plt.axis('tight');
```



It allows even higher-level specifications, such as ensuring an equal aspect ratio so that on your screen, one unit in x is equal to one unit in y:

```
plt.plot(x, np.sin(x))
plt.axis('equal');
```



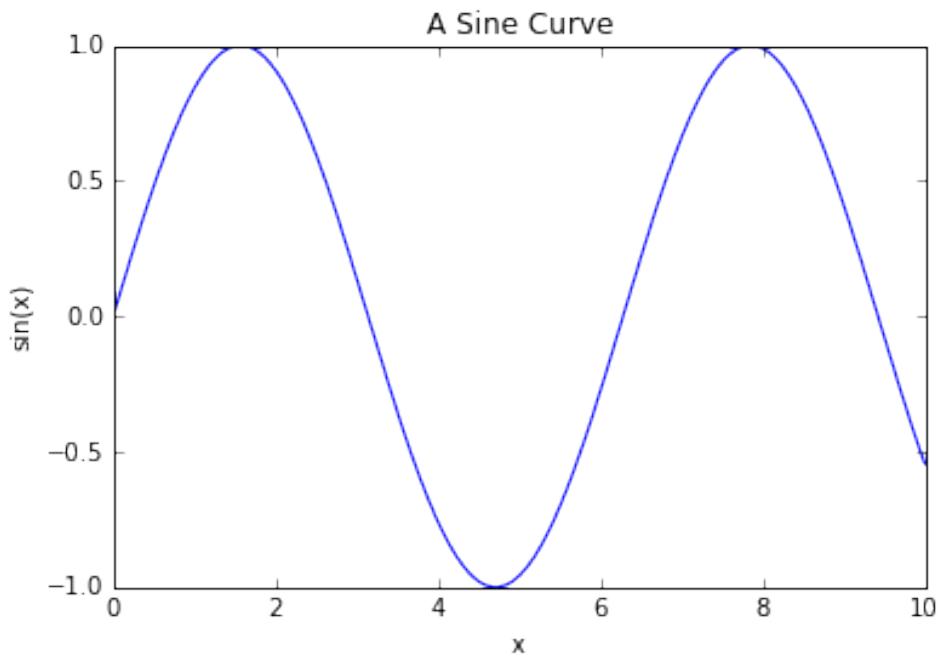
For more information on axis limits and the other capabilities of the `plt.axis` method, refer to the `plt.axis` docstring.

Labeling Plots

As the last piece of this recipe, we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.

Titles and axis labels are the simplest of these: there are methods which can be used to quickly set these:

```
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```

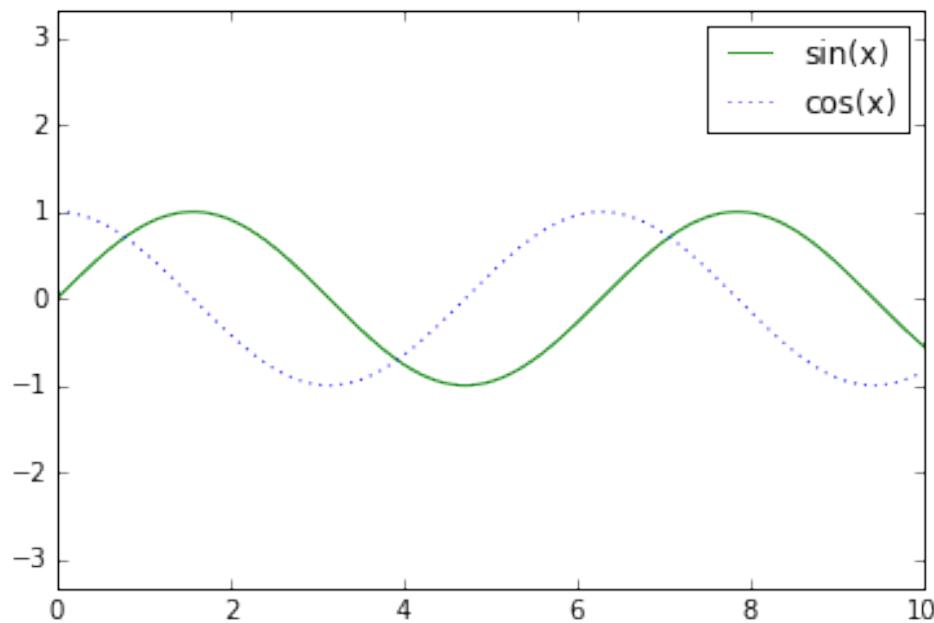


The position, size, and style of these labels can be adjusted using optional arguments to the function. For more information, see the matplotlib documentation and the docstrings of each of these functions.

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 `label` keyword of the plot function:

```
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();
```



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` doc string; additionally, we will cover some more advanced legend options in section X.X.

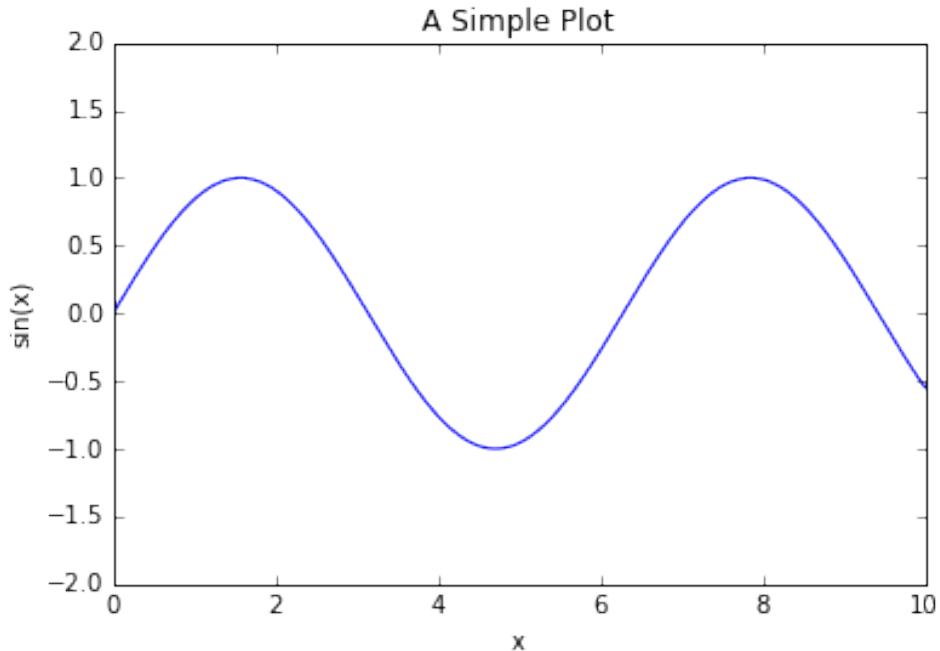
Sidebar: Gotchas

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 more convenient to use the `ax.set()` method to set all these properties at once:

```
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');
```



If you work much with matplotlib, these types of simple axes notations will become very familiar!

Simple Scatter Plots

In the previous recipe we saw how to create simple two-dimensional line plots with matplotlib. 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.

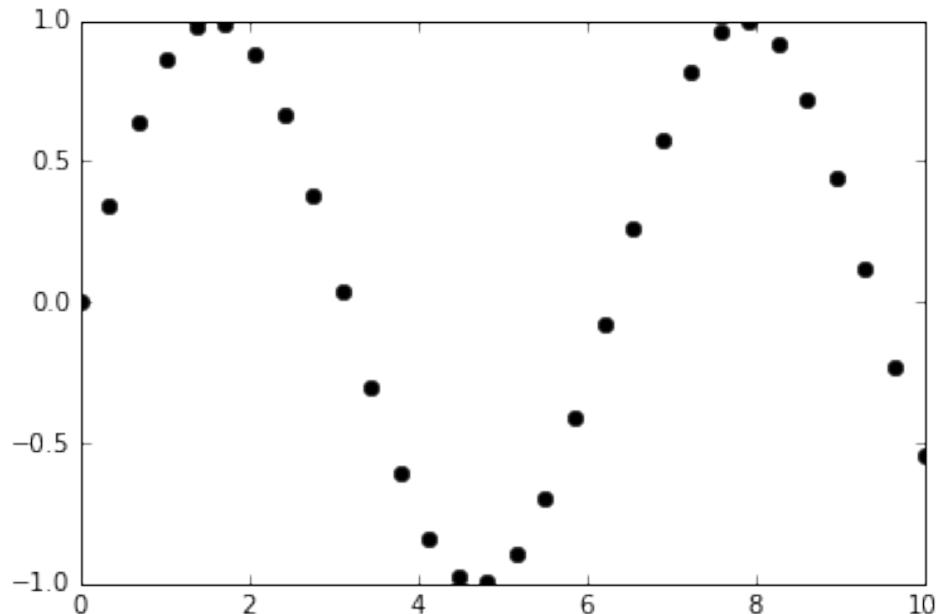
Scatter Plots with `plt.plot`

In the previous recipe we looked at `plt.plot/ax.plot` to produce line plots. It turns out that this function serves more than one purpose: it can produce scatter plots as well:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

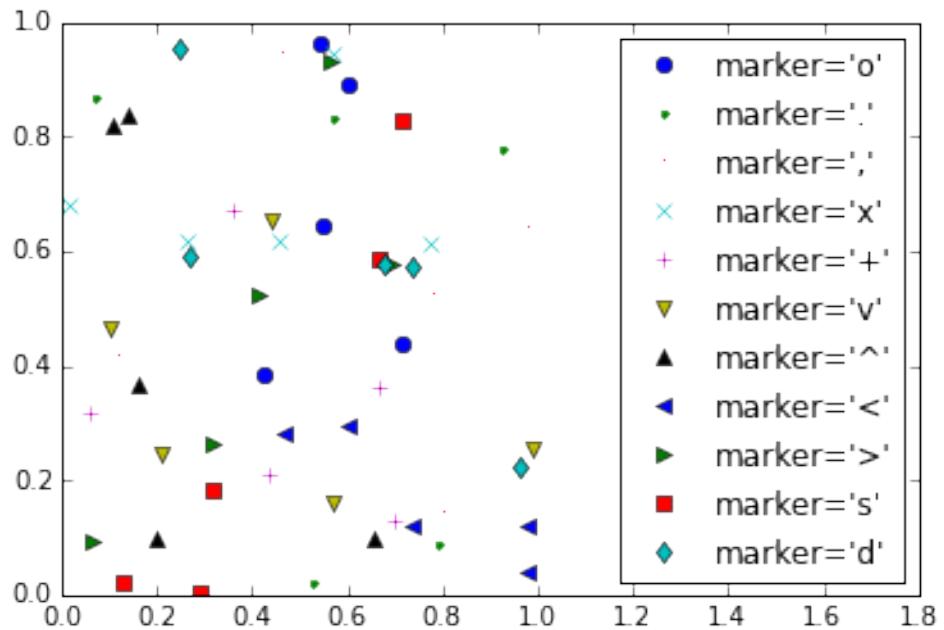
x = np.linspace(0, 10, 30)
y = np.sin(x)

plt.plot(x, y, 'o', color='black');
```



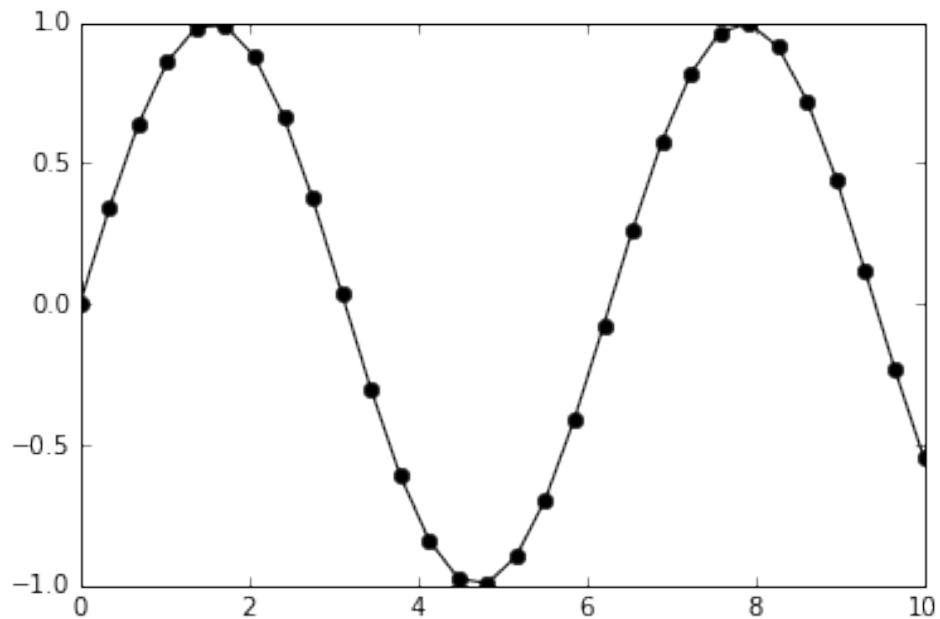
The third argument in the function call is a character which represents the type of symbol used for the plotting. Just as you can specify `'-'`, `'--'`, etc. to control the line style, the marker style has its own set of short string codes. The full list of available symbols can be seen in the documentation 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:

```
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(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```



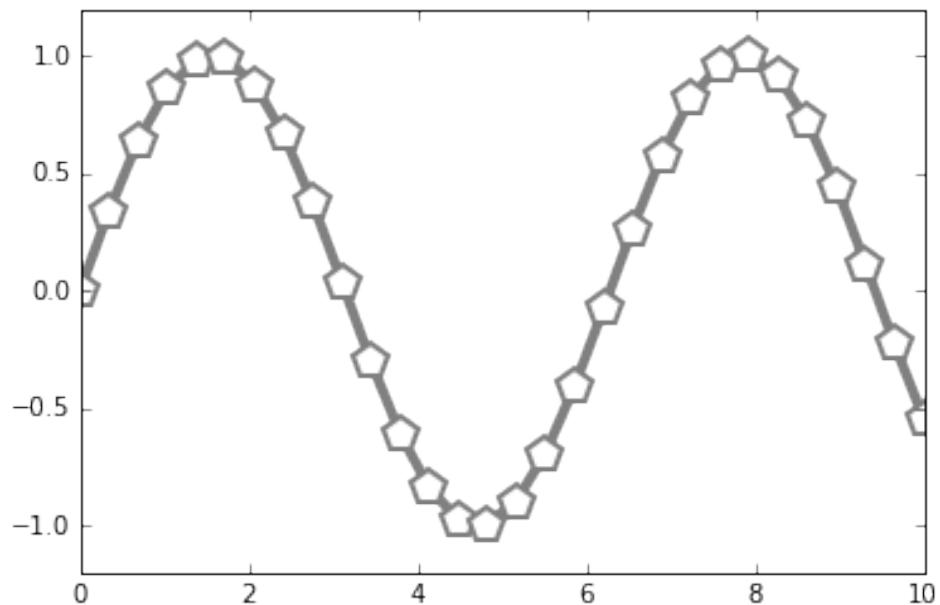
These character codes can be used together with line character codes to plot points along with a line connecting them:

```
plt.plot(x, y, '-ok');
```



keyword arguments to the `plt.plot` function can be used to specify a wide range of properties of the lines and markers:

```
plt.plot(x, y, '-p', color='gray',
         markersize=15,
         markerfacecolor='white',
         markeredgecolor='gray',
         markeredgewidth=2,
         linewidth=4)
plt.ylim(-1.2, 1.2);
```

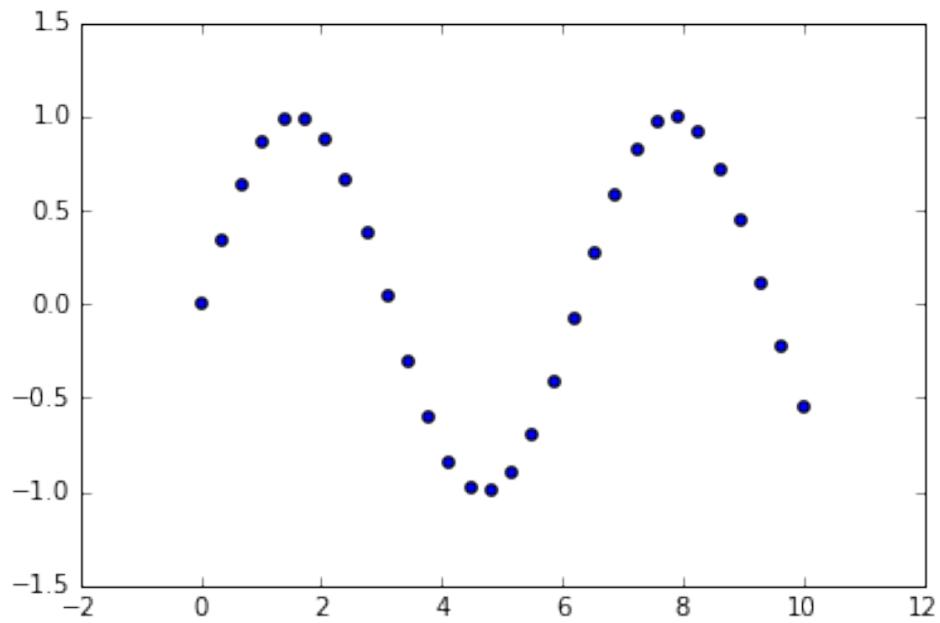


This type of flexibility in the `plt.plot` function allows for a wide variety of possible visualization options. For a full description of the options available, refer to the `plt.plot` documentation.

Scatter Plots with `plt.scatter`

A second, more powerful method of creating scatterplots is the `plt.scatter` function, which can be used very similarly to the `plt.plot` function:

```
plt.scatter(x, y, marker='o');
```

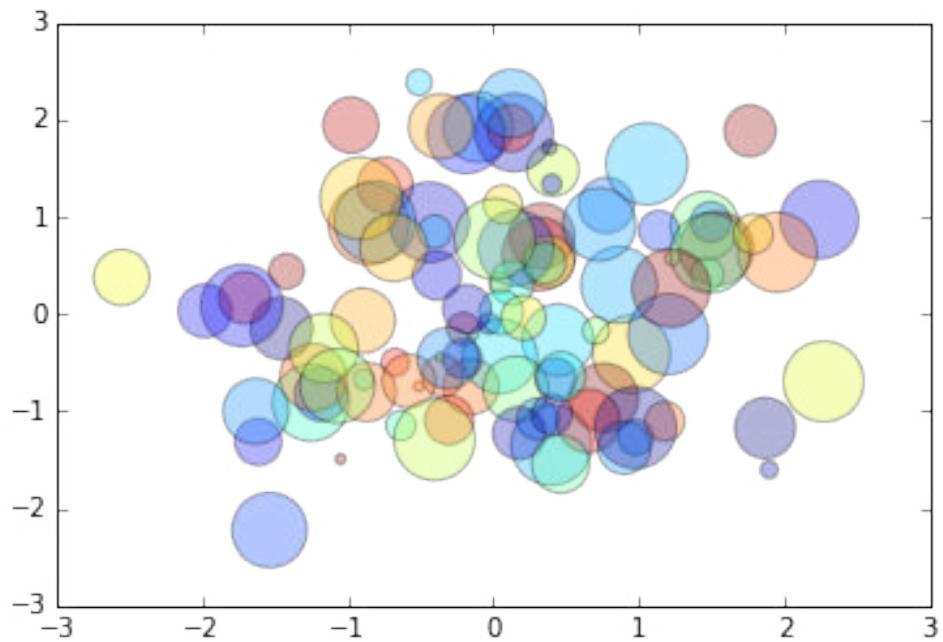


The primary difference of `plt.scatter` from `plt.plot` is that it can be used to create scatter plots where each individual point has a different size, face color, edge color, or other property.

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:

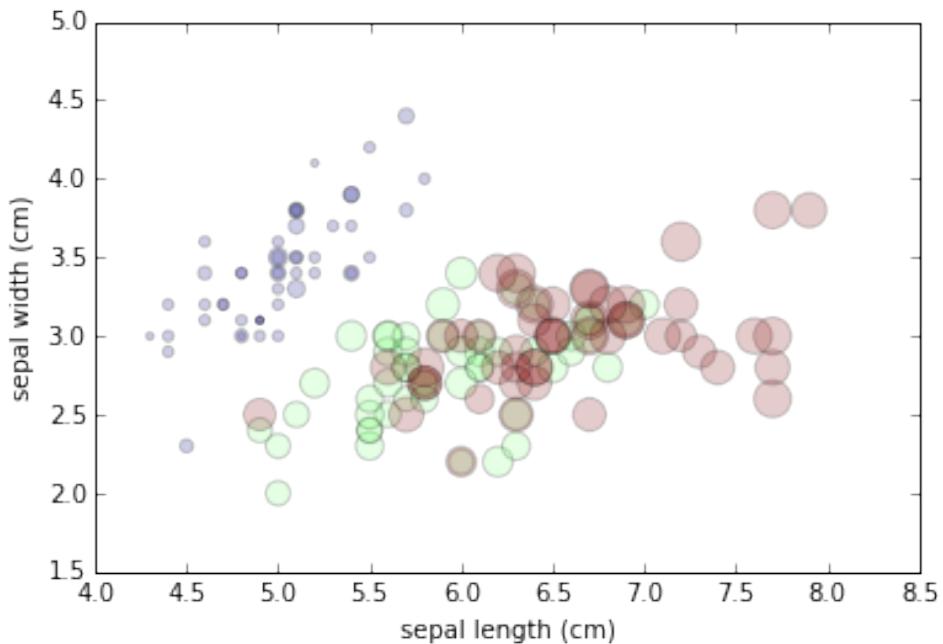
```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3);
```



Adjusting the size and color of different points can offer a very useful means of visualizing multi-dimensional data. For example, we might use the iris data from scikit-learn, where each sample is one of three types of flowers which has had the size of its petals and sepals carefully measured:

```
from sklearn.datasets import load_iris
data = load_iris()
features = data.data.T
plt.scatter(features[0], features[1], alpha=0.2,
           s=100*features[3], c=data.target)
plt.xlabel(data.feature_names[0])
plt.ylabel(data.feature_names[1]);
```



We can see that this scatter plot has given us the ability to simultaneously explore four different dimensions of the data set! 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. Multi-color and multi-feature scatter-plots like this can be extremely useful for both exploration and presentation of data.

A Note on Efficiency

Aside from the different features available in `plt.plot` and `plt.scatter`, why might you 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 simple: `plt.scatter` has the capability to render a different size and/or color for each point. For this reason, the renderer must do the extra work of constructing each point individually, even when it does not have to. In `plt.plot`, on the other hand, the points are always essentially clones of each other, so the work of determining 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. For this reason, when working with larger datasets, `plt.plot` should be preferred over `plt.scatter` whenever possible.

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 Constant, 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 of 74 (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 errorbars we'll discuss the meaning of these errorbars further in section X.X: The current literature 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? Most scientists would say yes.

In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

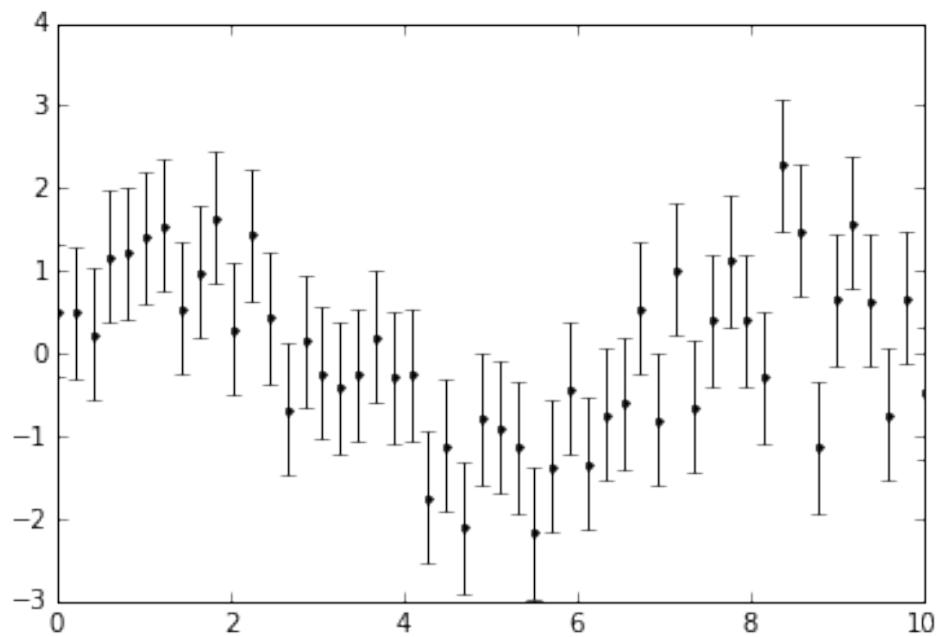
Basic Errorbars

A basic errorbar can be created with a single matplotlib function call. We'll see this here:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)

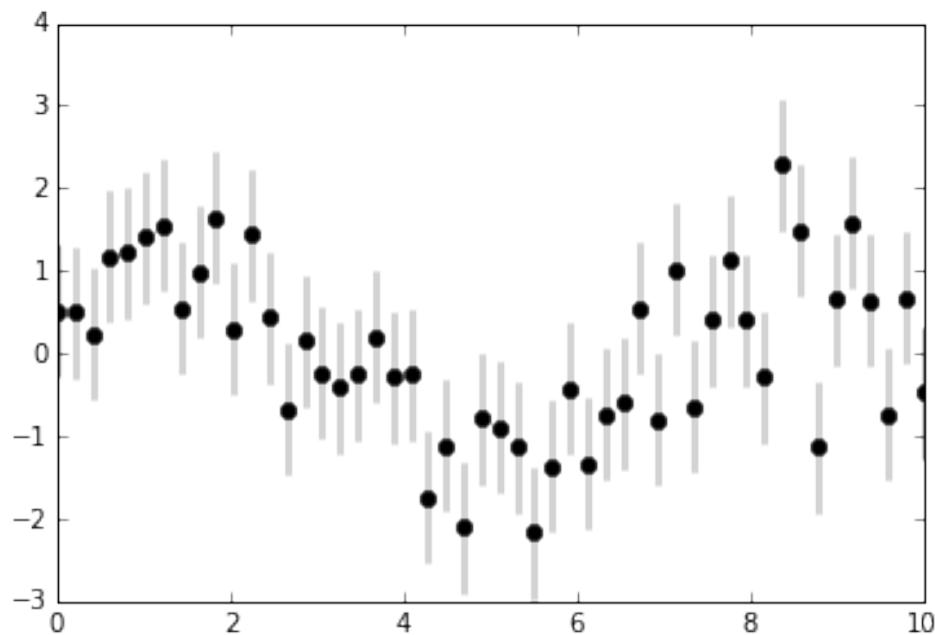
plt.errorbar(x, y, yerr=dy, fmt='.k');
```



Here the `fmt` is a format code controlling the appearance of lines and points, and has the same syntax as the shorthand used in `plt.plot`, outlined in Section X.X.

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 aesthetics of your errorbar plot. I often find it helpful, especially in crowded plots, to make the errorbars lighter than the points themselves:

```
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
             ecolor='lightgray', elinewidth=3, capsize=0);
```



In addition to these options, you can also specify horizontal error bars (`xerr`), one-sided errorbars, and many other variants. For more information on the options available, refer to the doc string 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 (GP) Regression*, explored more fully in section X.X. This is a method of fitting a very flexible non-parametric function to data with a continuous measure of the uncertainty. For the meantime, we won't say anything more about Gaussian Process Regression, but will focus instead on how one might visualize such a continuous error measurement:

```
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[:, None], ydata)

xfit = np.linspace(0, 10, 1000)
yfit, MSE = gp.predict(xfit[:, None], eval_MSE=True)
dyfit = 2 * np.sqrt(MSE) # 2*sigma ~ 95% confidence region

```

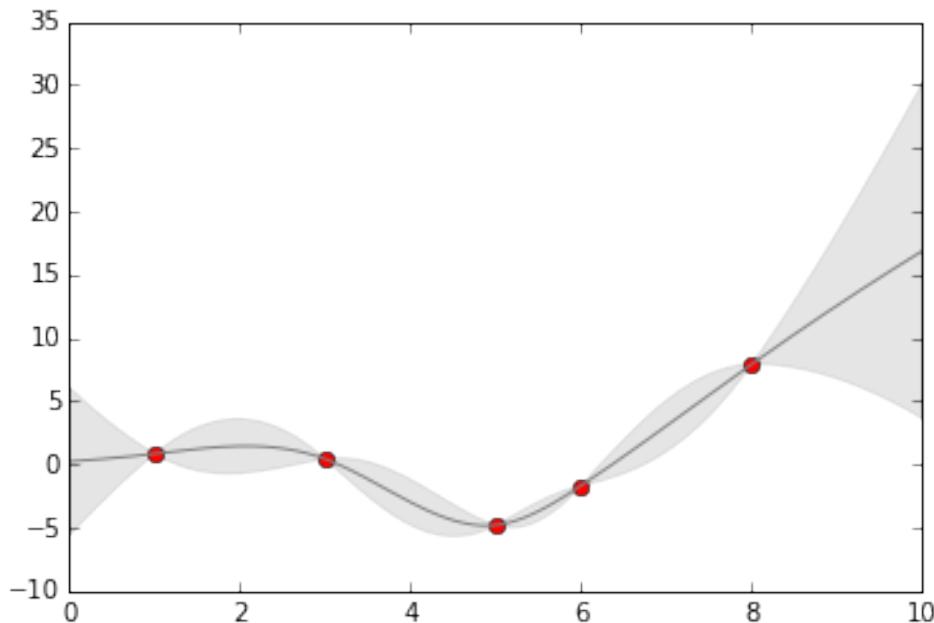
We now have `xfit`, `yfit`, and `dyfit`, which sample the continuous fit to our data. We could pass these to the `plt.errorbar` function as above, but we don't really want to plot 1000 points with 1000 errorbars. Instead, we can use the `plt.fill_between` function with a light color to visualize this continuous error:

```

# 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);

```



Note what we've done here with the `fill_between` function: we pass an x value, then the 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 Regression 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, keep in mind for this sort of visualization that the Seaborn package can be helpful; in section X.X we discuss Seaborn's more streamlined 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 which can be helpful for this task: `plt.contour` for contour plots, `plt.contourf` for filled contour plots, and `plt.imshow` for showing images. We'll see several examples of using these below.

Visualizing a 3D function

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

We'll start by demonstrating a contour plot using a function $z = f(x, y)$, using the following particular choice for f (we've seen this before in section X.X, when we used it as a motivating example for array broadcasting):

```
def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

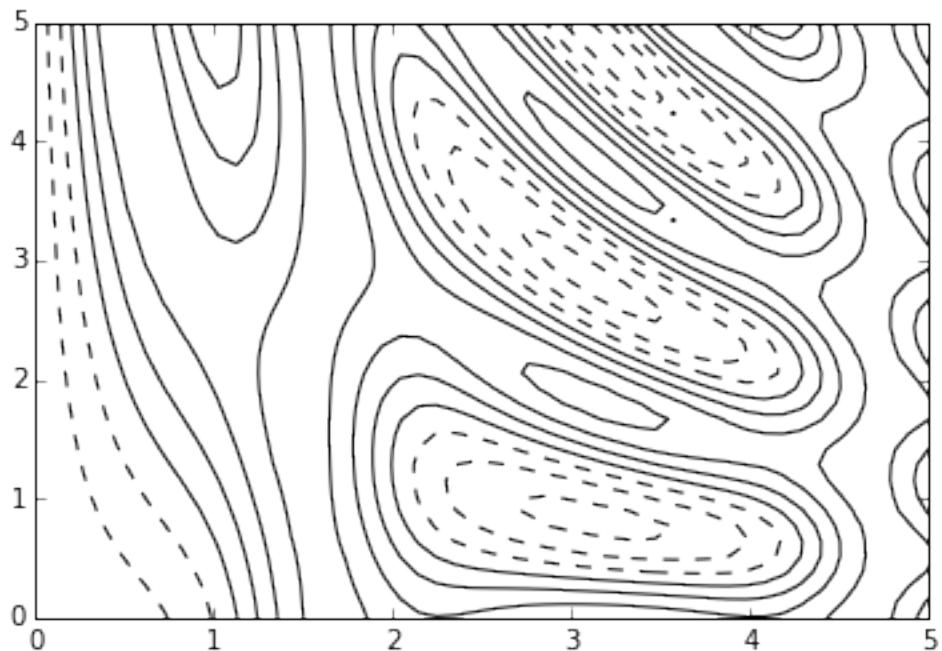
A contour plot can be created with the `plt.contour` function. It takes three arguments: a grid of x values, a grid of y values, and a grid of z values. The x and y values represent positions on the plot, and the z values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the `plt.meshgrid` function, which we explored in section X.X:

```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

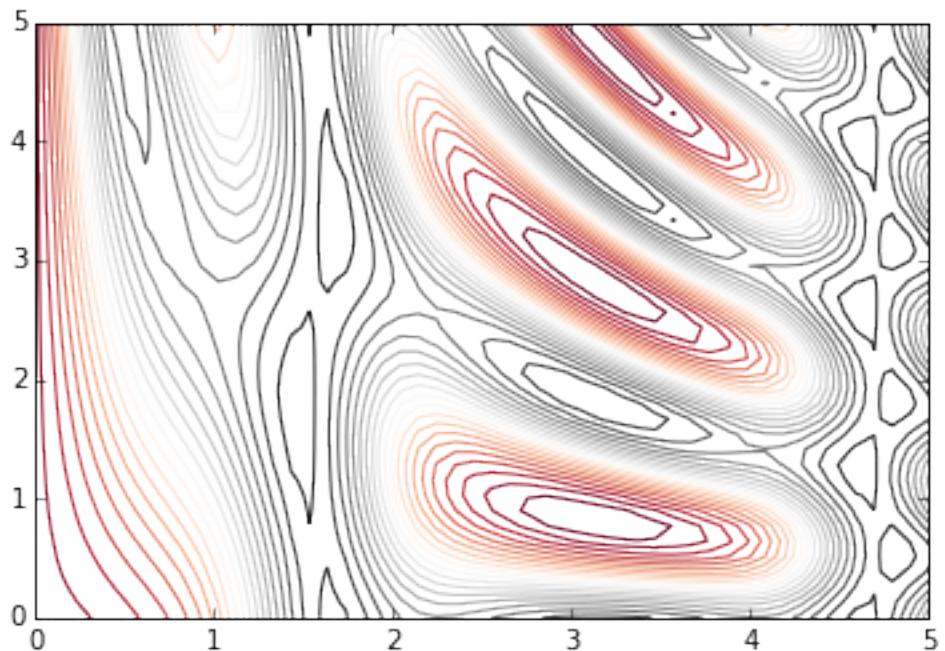
Now let's look at this with a standard line-only contour plot:

```
plt.contour(X, Y, Z, colors='black');
```



Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines. Alternatively, the lines can be color-coded by specifying a colormap with the `cmap` argument. Here, we'll also specify that we want more lines to be drawn: 20 equally-spaced intervals within the data range:

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```



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 typing

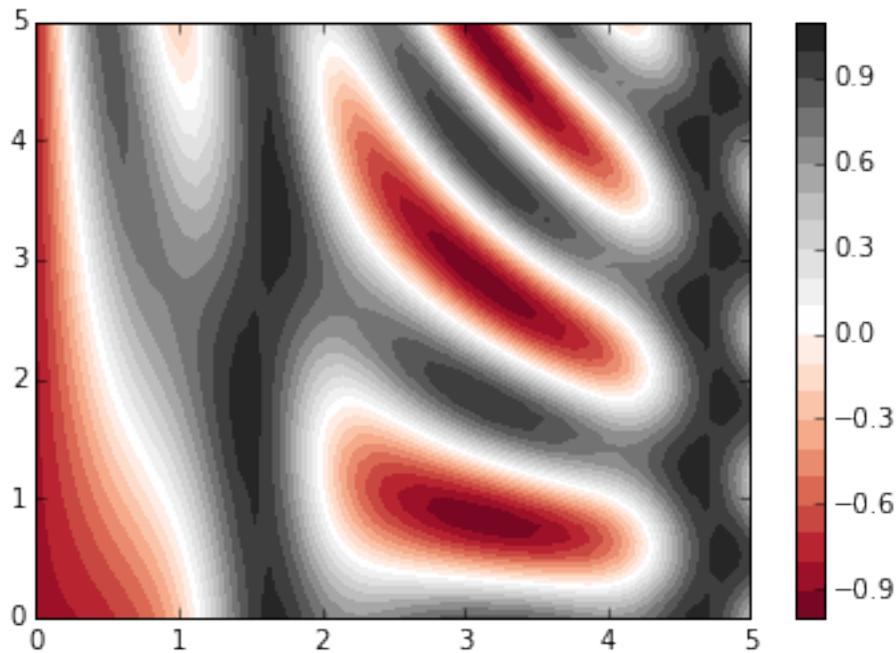
```
plt.cm.<TAB>
```

All available colormaps are within this namespace.

The above 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.contour()`.

Additionally, we'll add a `plt.colorbar()` command, which automatically creates an additional axis with labeled color information for the plot:

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```



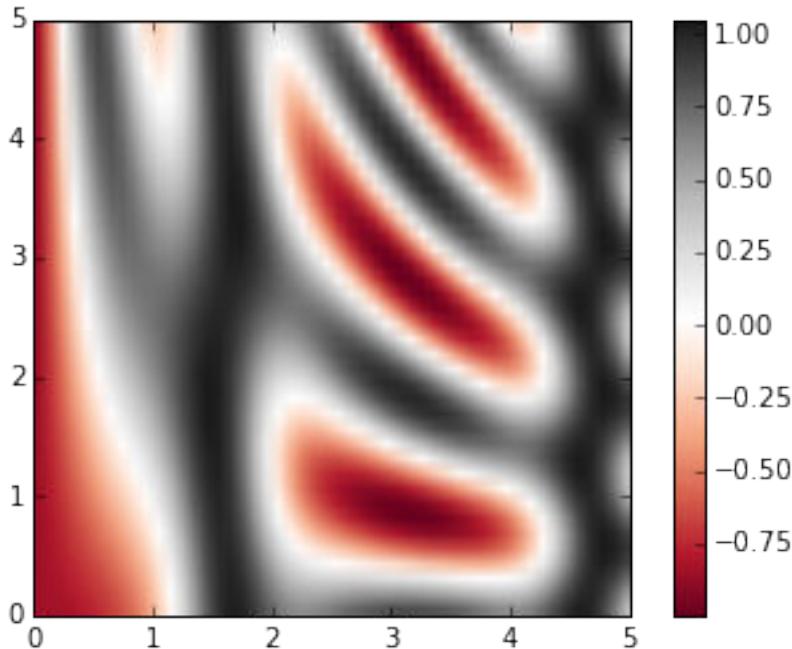
The colorbar makes it clear that the dark regions are “peaks”, while the red regions are “valleys”.

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. This could be remedied 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 interprets a two-dimensional grid of data as an image. There are a few potential gotchas in this, 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. This must be changed when showing gridded data.
- `plt.imshow()` will automatically adjust the axis aspect ratio to match the input data; this can be changed by setting, e.g. `plt.axis(aspect='image')` to make x and y units match.

Here is what the result looks like:

```
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
           cmap='RdGy')
plt.colorbar()
plt.axis(aspect='image');
```

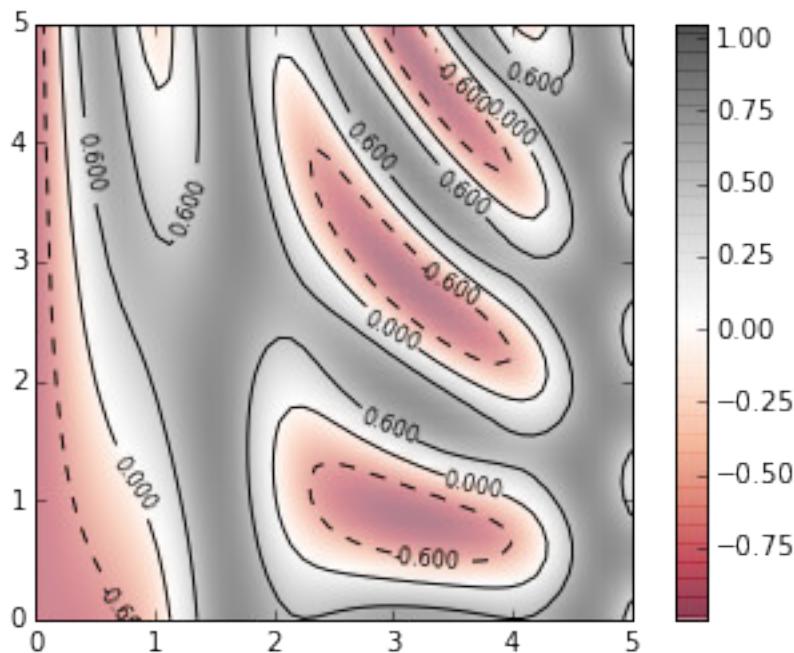


Finally, it can sometimes be useful to combine contour plots and image plots. Here is an example of this where we'll use a partially transparent background image (with transparency set via the `alpha` parameter) and overplot contours with labels on the contours themselves (using the `plt.clabel()` function).

```
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()

plt.axis(aspect='image');
```



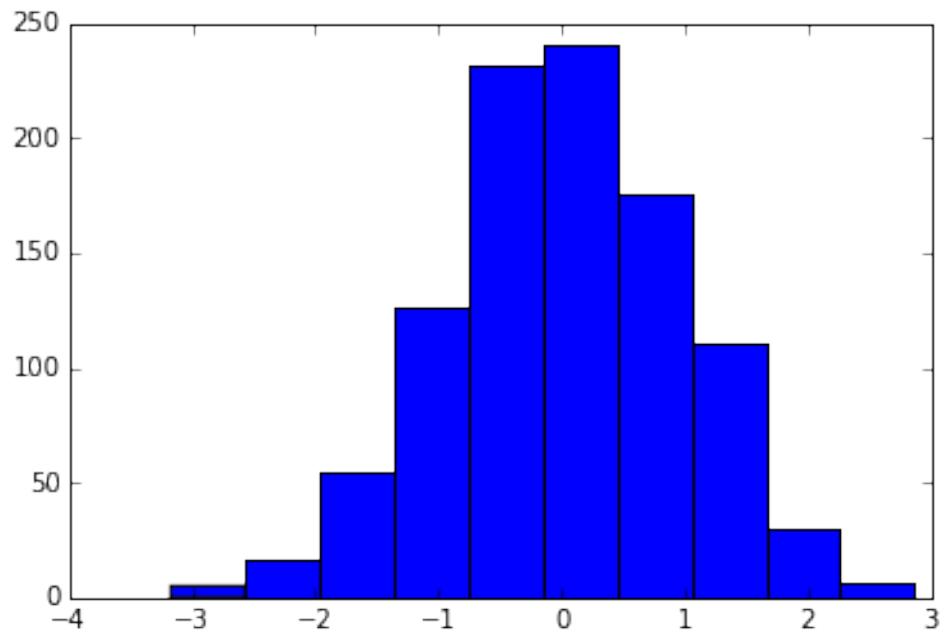
The combination of these three functions, `plt.contour`, `plt.contourf`, and `plt.imshow` gives nearly limitless possibilities for displaying this sort of three-dimensional data within a two-dimensional plot. For more information on the options available in these functions, refer to their docstrings. If you are interested in three-dimensional visualizations of this type of data, see Section X.X.

Histograms and Binnings

When exploring various datasets, a simple histogram is often one of the most useful tools. We saw in the previous chapter a preview of matplotlib's histogram function, which creates a basic histogram in one line:

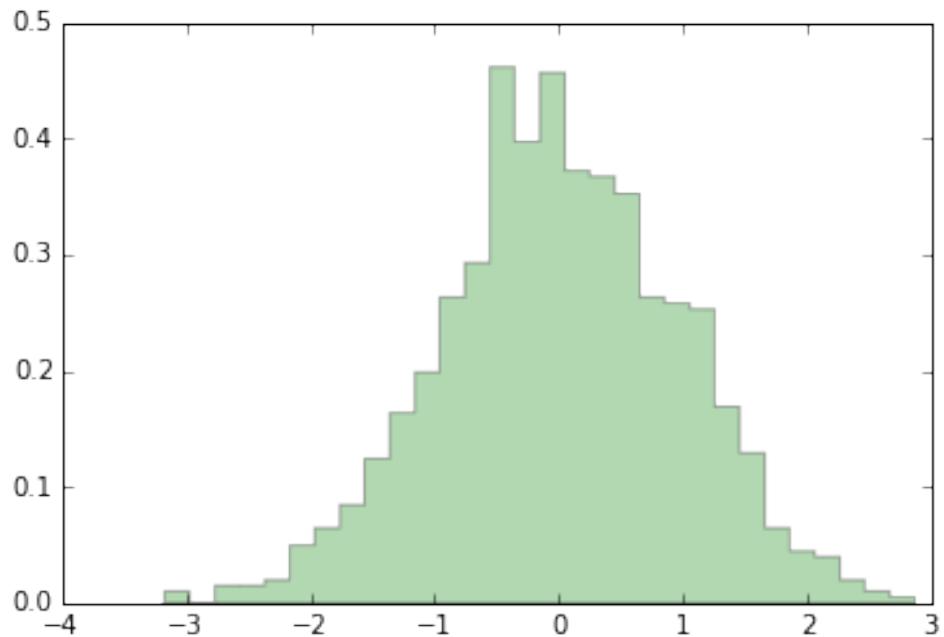
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

x = np.random.randn(1000)
plt.hist(x);
```



The `hist()` function has many options to tune both the calculation and the display, which are well explained in the documentation. Here's an example of a more customized histogram:

```
plt.hist(x, bins=30, normed=True, alpha=0.3,  
         histtype='stepfilled', color='green');
```

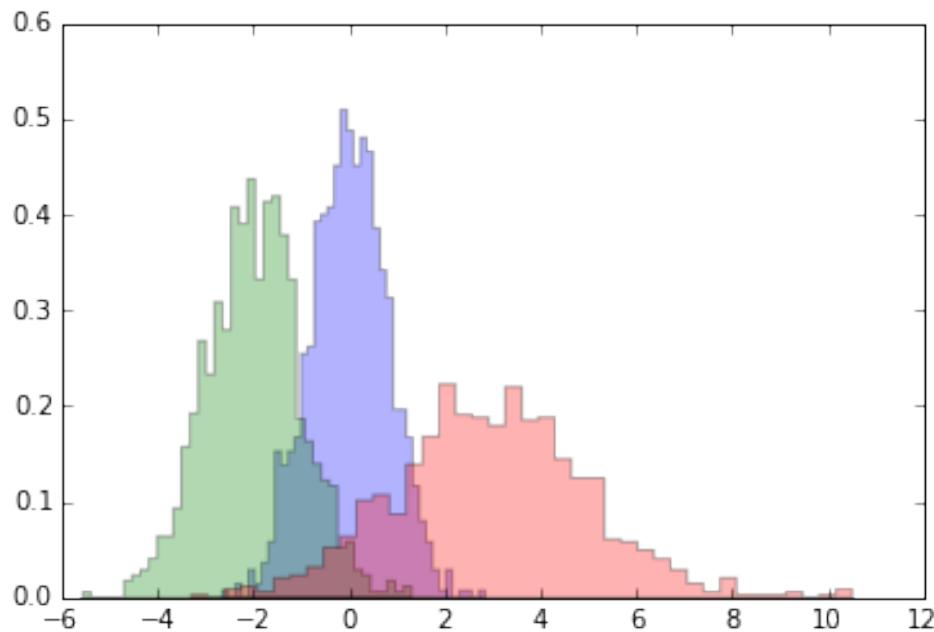


I find the combination of `histtype='stepfilled'` along with some transparency `alpha` to be very useful when comparing histograms of several distributions:

```
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);
```



If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the `np.histogram()` function is available

```
counts, bin_edges = np.histogram(x, bins=5)
print(counts)

[ 22 181 473 287 37]
```

Two-dimensional Histograms and Binnings

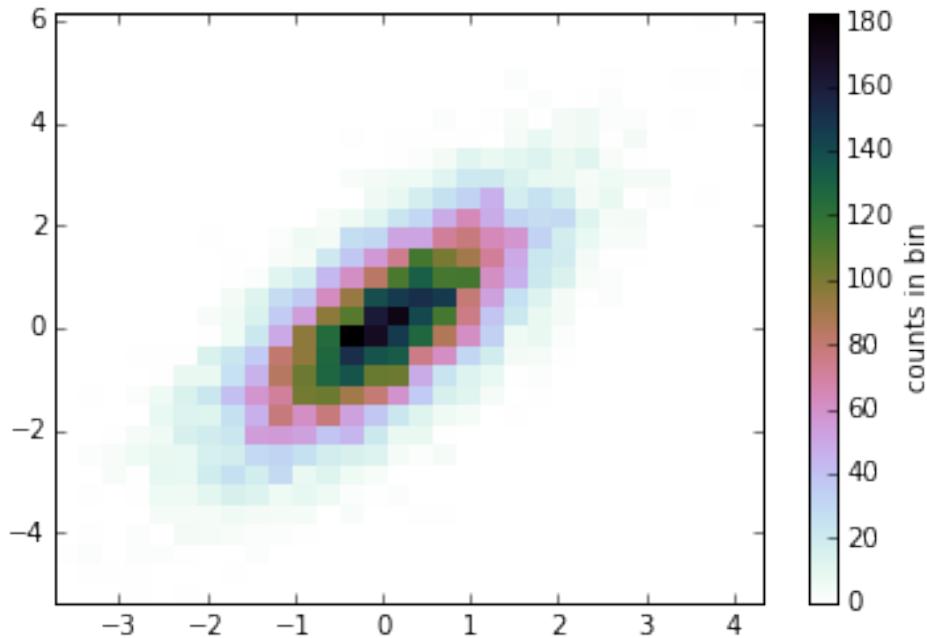
Just as we create histograms in one dimension by dividing the number-line into bins, we can also create histograms in two-dimensions by dividing points among two-dimensional bins. We'll take a brief look at several ways to do this here. We'll start by defining some data: an `x` and `y` array drawn from a multivariate Gaussian distribution:

```
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 2D histogram is to use matplotlib's `plt.hist2d` function:

```
plt.hist2d(x, y, bins=30, cmap='cubehelix_r')
cb = plt.colorbar()
cb.set_label('counts in bin')
```



The `hist2d` function has a number of extra options to fine-tune the plot and the binning, outlined in the function docstring. Further, just as `plt.hist` a counterpart in `np.histogram`, `plt.hist2d` has a counterpart in `np.histogram2d` which can be used as follows:

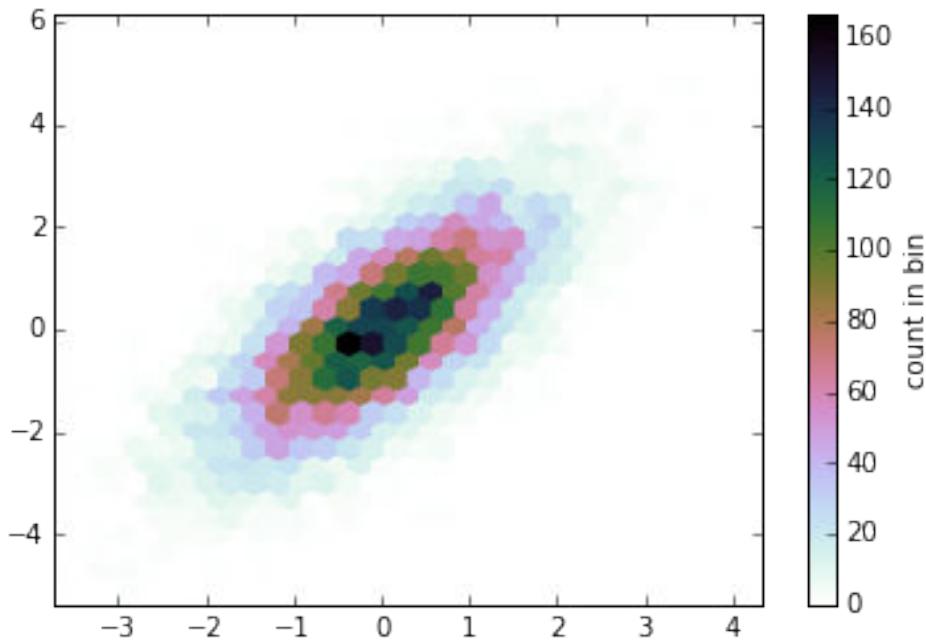
```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

For the generalization of this histogram binning in dimensions higher than 2, see the `np.histogramdd` function.

`plt.hexbin`: Hexagonal Binnings

One natural shape to use for a tessellation across a two-dimensional space is the regular hexagon. For this purpose, matplotlib provides the `plt.hexbin` routine, which automatically represents a two-dimensional dataset binned within a grid of hexagons:

```
plt.hexbin(x, y, gridsize=30, cmap='cubehelix_r')
cb = plt.colorbar(label='count in bin')
```



`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). This will be discussed more fully in section X.X, but for now we'll simply mention that KDE can be thought 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 we'll give a quick example of using the KDE on this data:

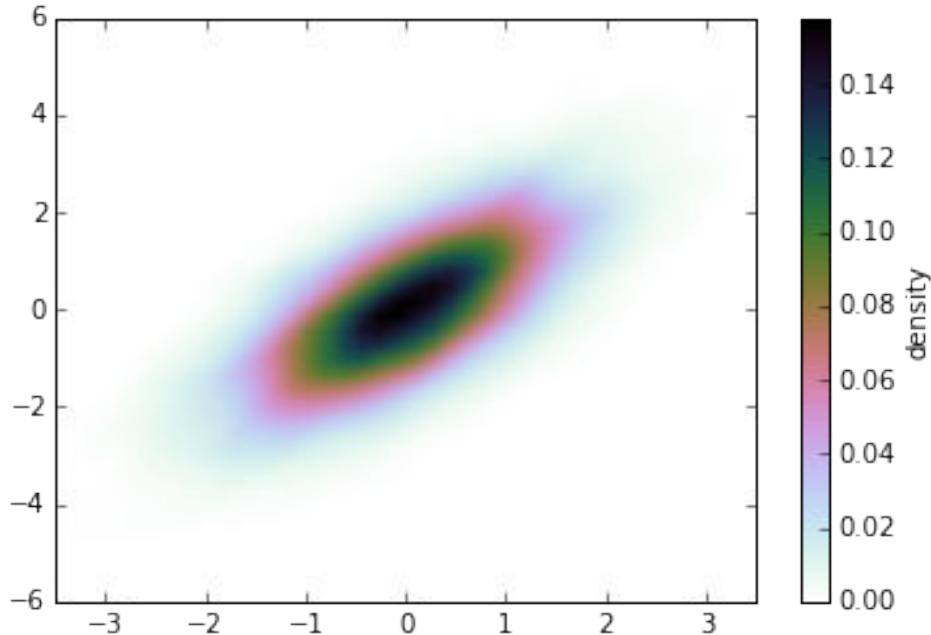
```
from scipy.stats import gaussian_kde

# fit an array of size [Ndim, Nsamples]
kde = gaussian_kde(np.vstack([x, y]))

# 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='cubehelix_r')
cb = plt.colorbar()
cb.set_label("density")
```



KDE has a smoothing length which is effectively slides the knob between detail and smoothness (one example of the ubiquitous bias/variance tradeoff). The literature on 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 which have various strengths and weaknesses; see for example `sklearn.neighbors.KernelDensity` and `statsmodels.nonparametric.kernel_density.KDEMultivariate`.

For visualizations based on KDE, using `matplotlib` as above tends to be overly verbose. The Seaborn library, discussed in Section X.X, provides a much more terse API for creating KDE-based visualizations.

Binned Statistics

TODO: show `scipy.stats.binned_statistic` or `plt.hexbin` with custom C argument? What data to use?

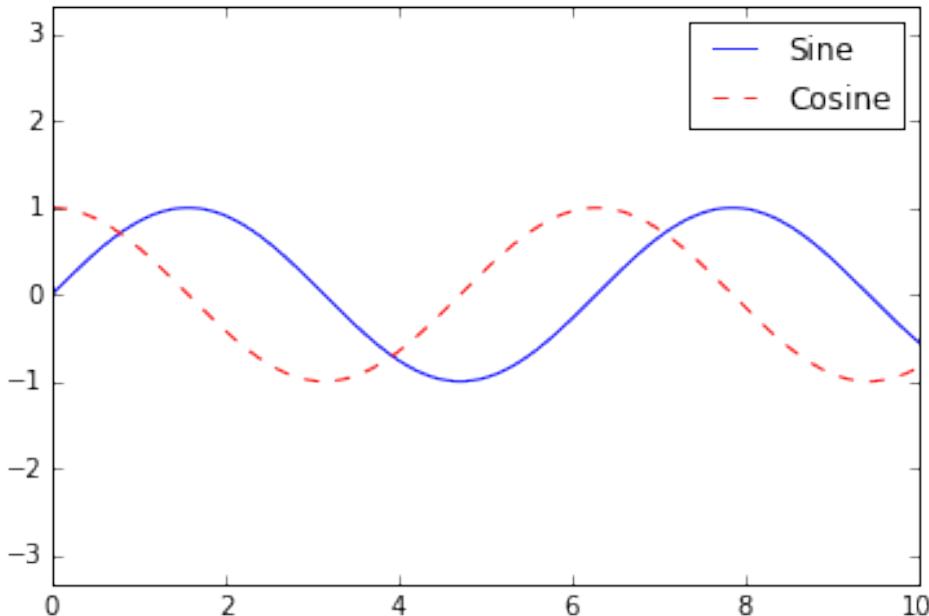
Customizing Legends

We saw previously how to create a simple legend on an axes. Here we'll take a look at customizing the placement and aesthetics of the legend in matplotlib.

From previous sections, you should already have seen the `plt.legend()` command, which automatically creates a legend for any labeled plot elements. For example:

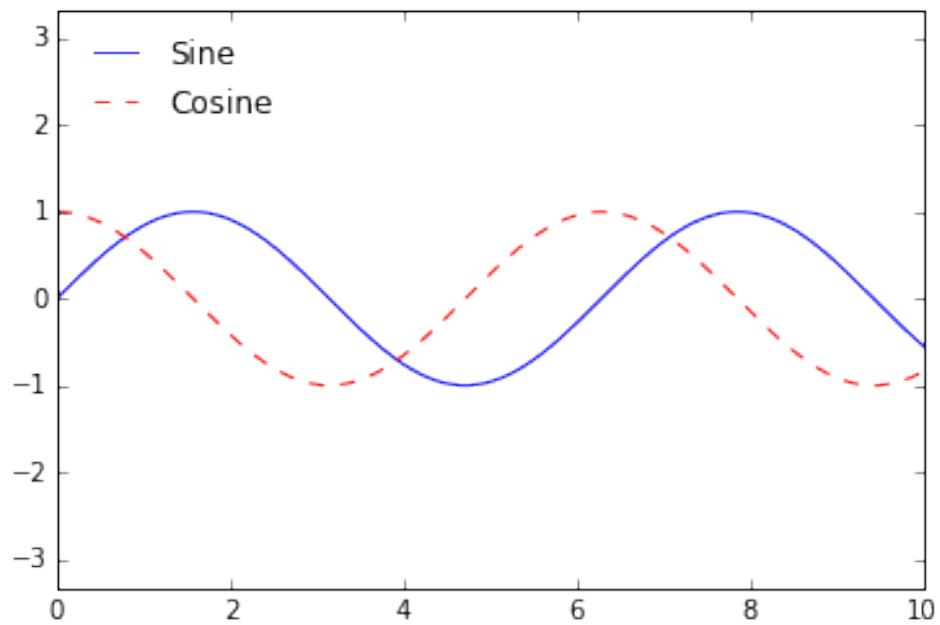
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

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();
```



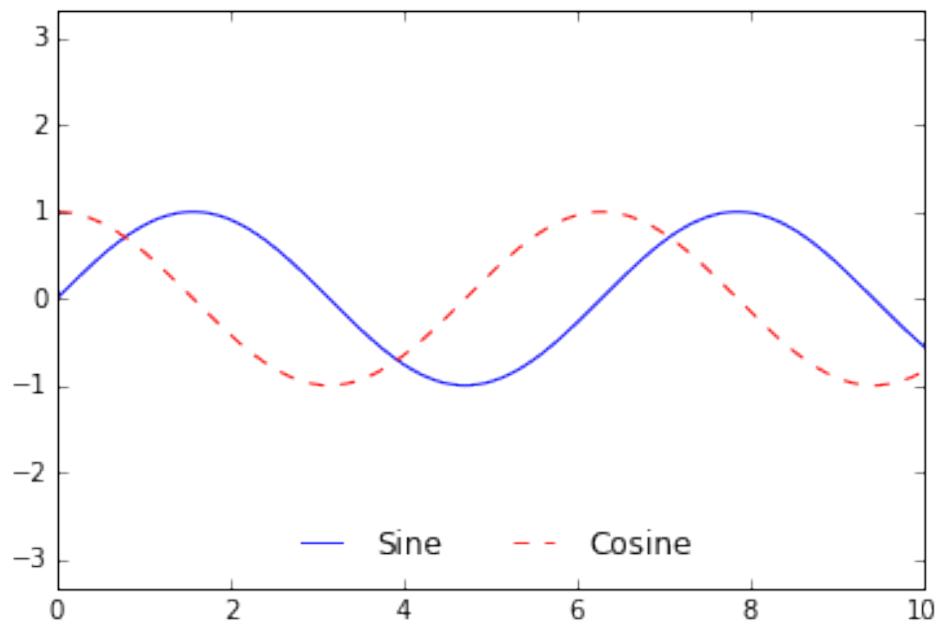
But there are many ways we might want to customize such a legend. For example, we can specify the location and turn off the frame:

```
ax.legend(loc='upper left', frameon=False)
fig
```



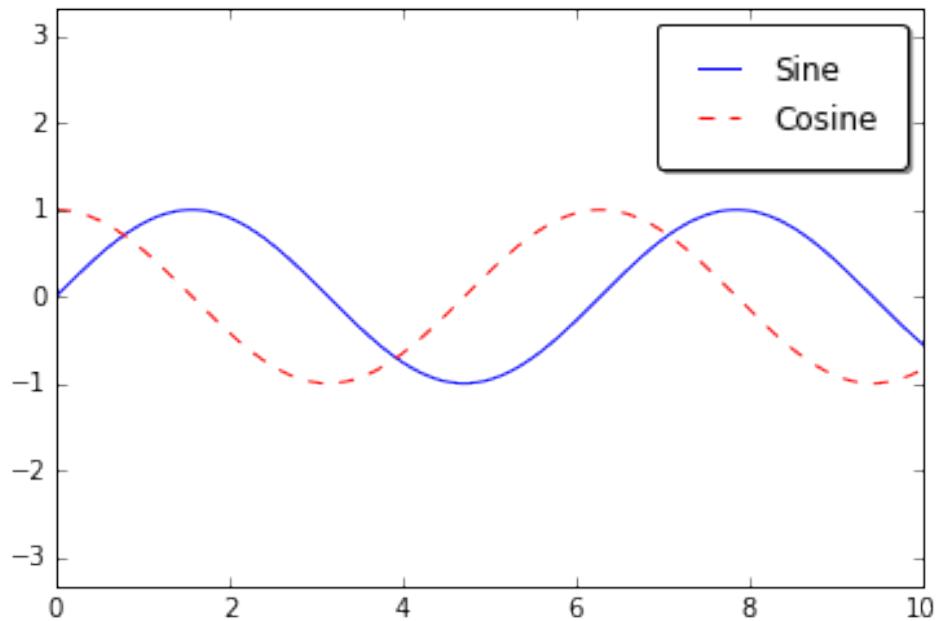
We can use the `ncol` command to specify the number of columns in the legend:

```
ax.legend(frameon=False, loc='lower center', ncol=2)  
fig
```



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:

```
ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)  
fig
```



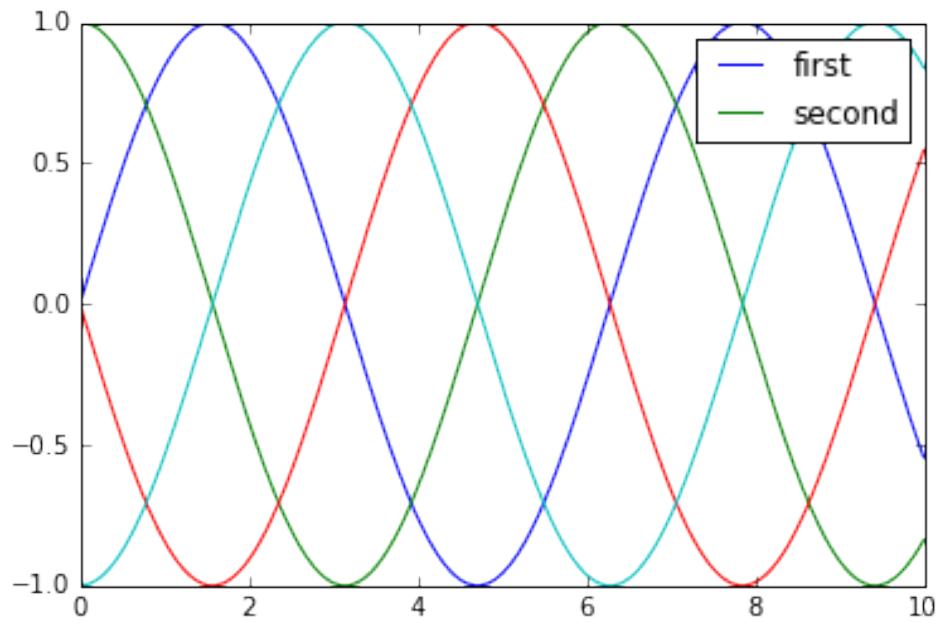
For more information on available legend options, see the `plt.legend` docstring.

Choosing Elements for the Legend

Above we saw that by default, the legend includes all labeled elements. 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 of these to `plt.legend()` will tell it which to identify, along with the labels we'd like to specify:

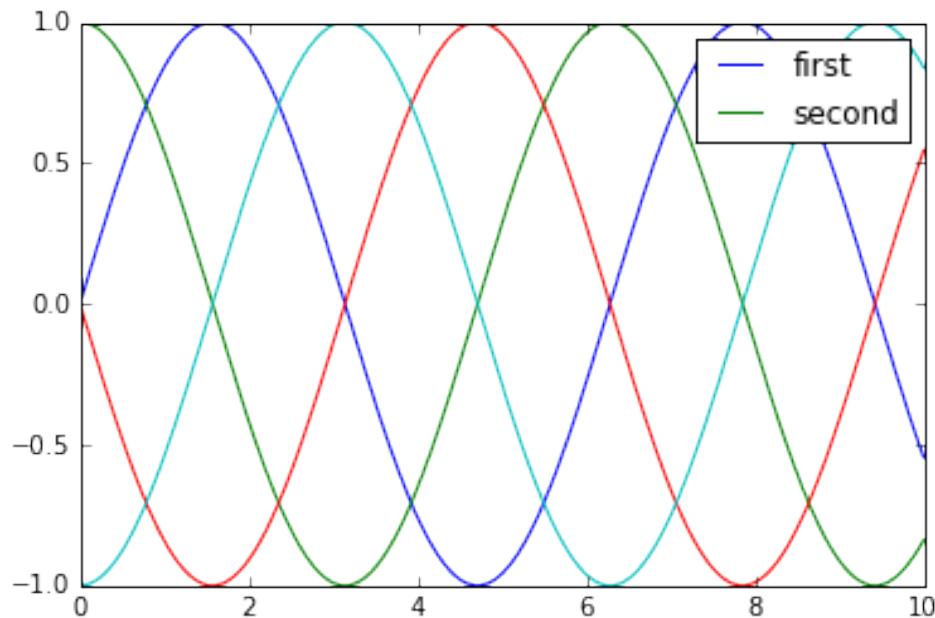
```
y = np.sin(x[:, np.newaxis] + np.pi * np.arange(0, 2, 0.5))
lines = plt.plot(x, y)

# lines is a list of plt.Line2D instances
plt.legend(lines[:2], ['first', 'second'], framealpha=1);
```



I generally find in practice that it is more clear to use the first method, applying labels to the plot elements you'd like to show on the legend:

```
plt.plot(x, y[:, 0], label='first')
plt.plot(x, y[:, 1], label='second')
plt.plot(x, y[:, 2:])
plt.legend(framealpha=1);
```



Notice that by default, the legend ignores all elements without a `label` attribute set.

Faking the Legend

Sometimes the legend defaults are not sufficient for the given visualization. For example, you may be 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 of points to indicate populations of California cities. We'd like a legend which specifies the scale of the sizes of the points, and we'll accomplish this by plotting some labeled data with no entries:

```

import pandas as pd
cities = pd.read_csv('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 & color
plt.scatter(lon, lat,
            c=np.log10(population), cmap='Greens_r',
            s=area, linewidth=0, alpha=0.5)
plt.axis(aspect='equal')
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar(label='log$_{10}$(population)')

```

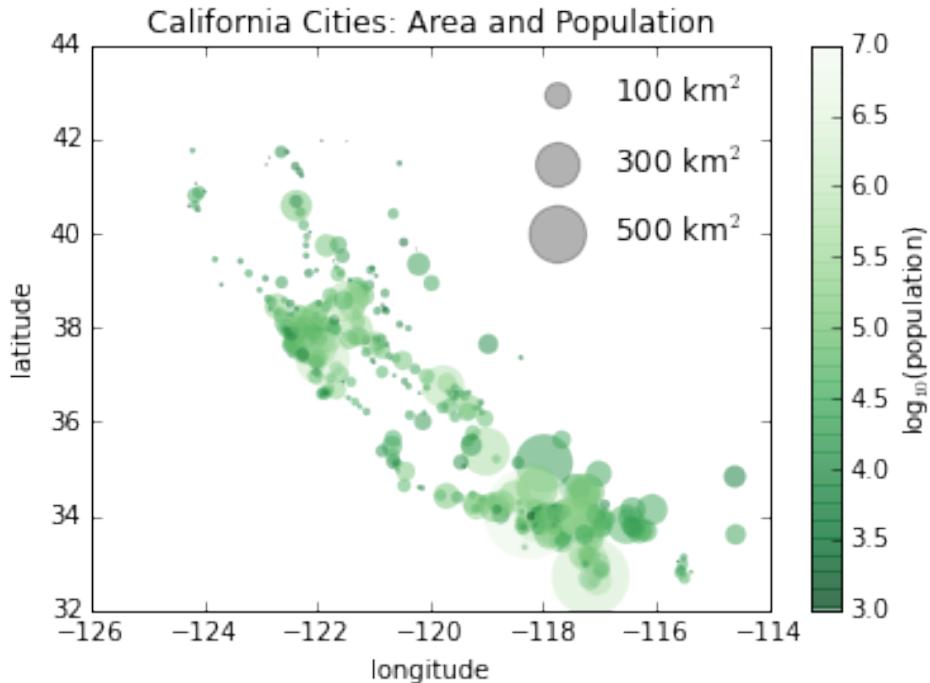
```

plt.clim(3, 7)

# Here we create a legend:
# we'll plot empty lists with the desired size & 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)

plt.title('California Cities: Area and Population');

```



The legend will always reference some object which is on the plot, so if we'd like to display a particular shape we need to plot it. In this case, the objects we want (gray circles) are not on the plot, so we fake them by plotting empty lists. Notice too that the legend only lists plot elements which have a label specified

By plotting empty lists, we create labeled plot objects which 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 more clear if we could show state boundaries or other map-specific elements. For this, an excellent choice of tool is matplotlib's Basemap addon toolkit, which we'll explore in section X.X.

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 first one. 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:

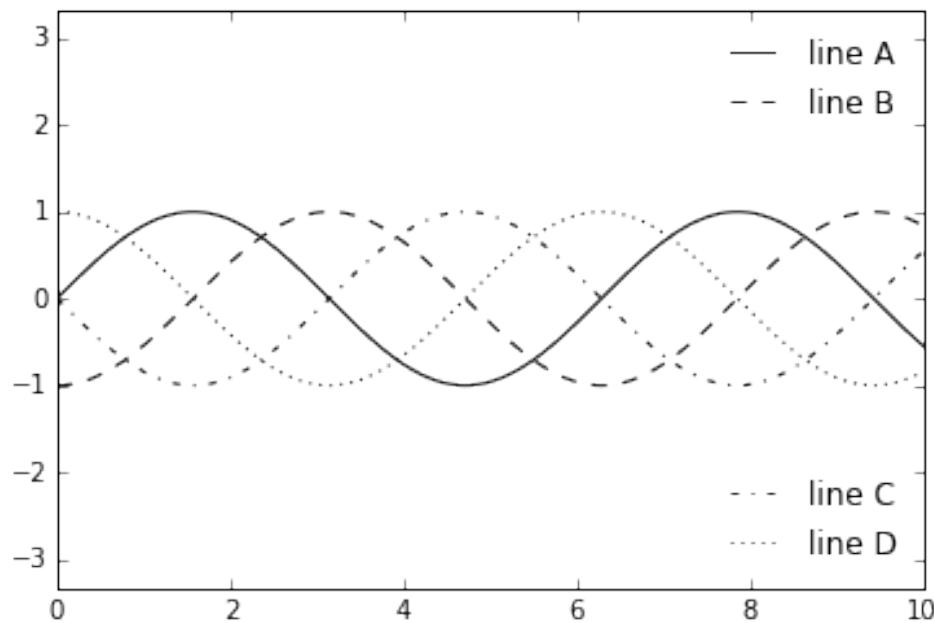
```
fig, ax = plt.subplots()

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);
```



This is a peek into the low-level artist objects which comprise any matplotlib plot. Indeed, if you examine the source code of `ax.legend()` (recall that you can do this with `ax.legend??` within the IPython notebook) 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 at draw time.

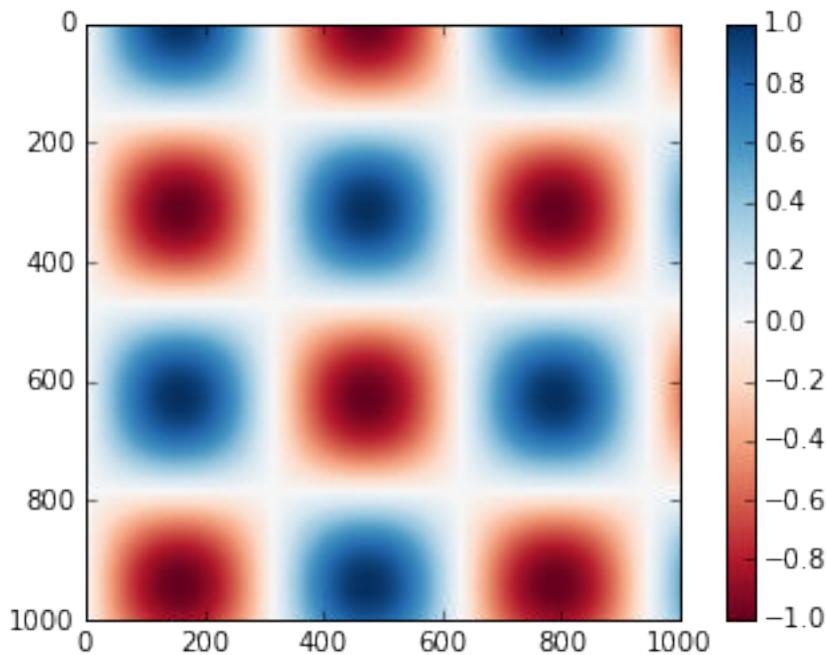
Customizing Colorbars

Plot legends identify discrete labels of discrete points. For continuous labels based on the color of points, lines, or regions, a labeled colorbar can be a great tool. In matplotlib, a colorbar is a separate axes which can provide a key for the meaning of colors in a plot.

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

The simplest colorbar can be created with the `plt.colorbar` function:

```
x = np.linspace(0, 10, 1000)
I = np.sin(x) * np.cos(x[:, np.newaxis])
plt.imshow(I, cmap='RdBu')
plt.colorbar();
```



Below we'll discuss a few ideas for customizing these colorbars and using them effectively in various situations.

Customizing Colorbars

Choosing the Colormap

One of the most important considerations when using colorbars is to choose an appropriate colormap, set using the `cmap` argument of a variety of plotting functions. 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:

1. *Sequential* colormaps: these are made up of one continuous sequence of colors (e.g. `binary` or `cubehelix`).
2. *Divergent* colormaps: these usually contain two distinct colors, which show positive and negative deviations from a mean (e.g. `RdBu` or `PuOr`).
3. *Qualitative* colormaps: these mix colors with no particular sequence (e.g. `accent` or `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 not a useful choice for representing quantitative data. Among the problems is the fact that qualitative maps usually do not display any uniform progression in brightness as the scale increases.

We can see this by converting the jet colorbar into black and white:

```
def grayscale_cmap(cmap):
    """Return a greyscale version of the given colormap"""
    cmap = plt.cm.get_cmap(cmap)
    colors = cmap(np.arange(cmap.N))

    # convert RGBA to perceived greyscale 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 cmap.from_list(cmap.name + "_gray", colors, cmap.N)

def view_colormap(cmap):
    """Plot a colormap with its greyscale 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])

view_colormap('jet')
```



Notice the bright stripes in the grayscale image. Even in full-color, this uneven brightness means that the eye will be drawn to certain portions of the color range, which will potentially emphasize unimportant parts of the dataset. Better is to use a color-

map such as `cubehelix`, which is specifically constructed 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 greyscale printing:

```
view_colormap('cubehelix')
```



For other situations such as showing positive and negative deviations from some mean, dual-color colorbars such as `RdBu` (Red-Blue) can be useful. Note, however, that the positive-negative information will be lost upon translation to greyscale!

```
view_colormap('RdBu')
```



We'll see examples of using some of these color maps below.

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 within the Seaborn library (Section X.X).

Color Limits and Extensions

There are many available options for customization of a colorbar in matplotlib. 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 displaying an image which is subject to noise:

```

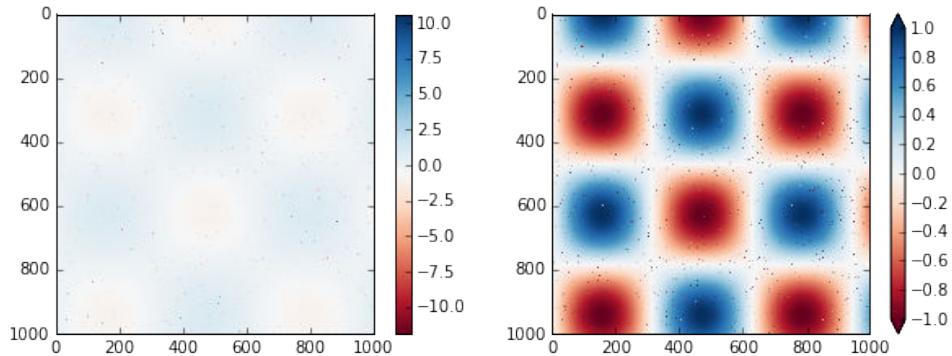
# make noise in 1% of the image pixels
speckles = (np.random.random(I.shape) < 0.01)
I[speckles] = np.random.normal(0, 3, np.count_nonzero(speckles))

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='both')
plt.clim(-1, 1);

```



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 which are above or below those limits.

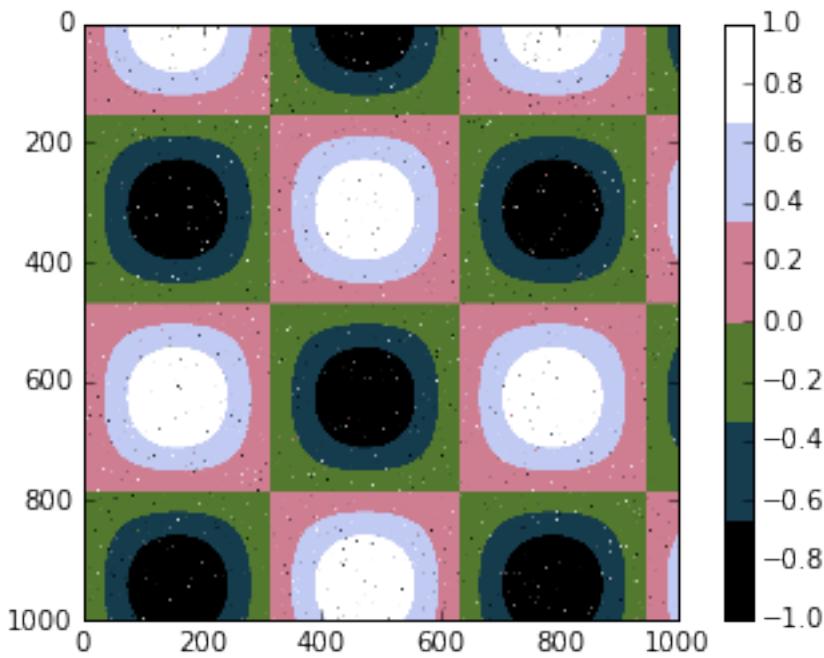
Discrete Color Bars

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 pass the name of the colormap along with the number of desired bins:

```

plt.imshow(I, cmap=plt.cm.get_cmap('cubehelix', 6))
plt.colorbar()
plt.clim(-1, 1);

```



This discrete colormap can now be used just like any other colormap.

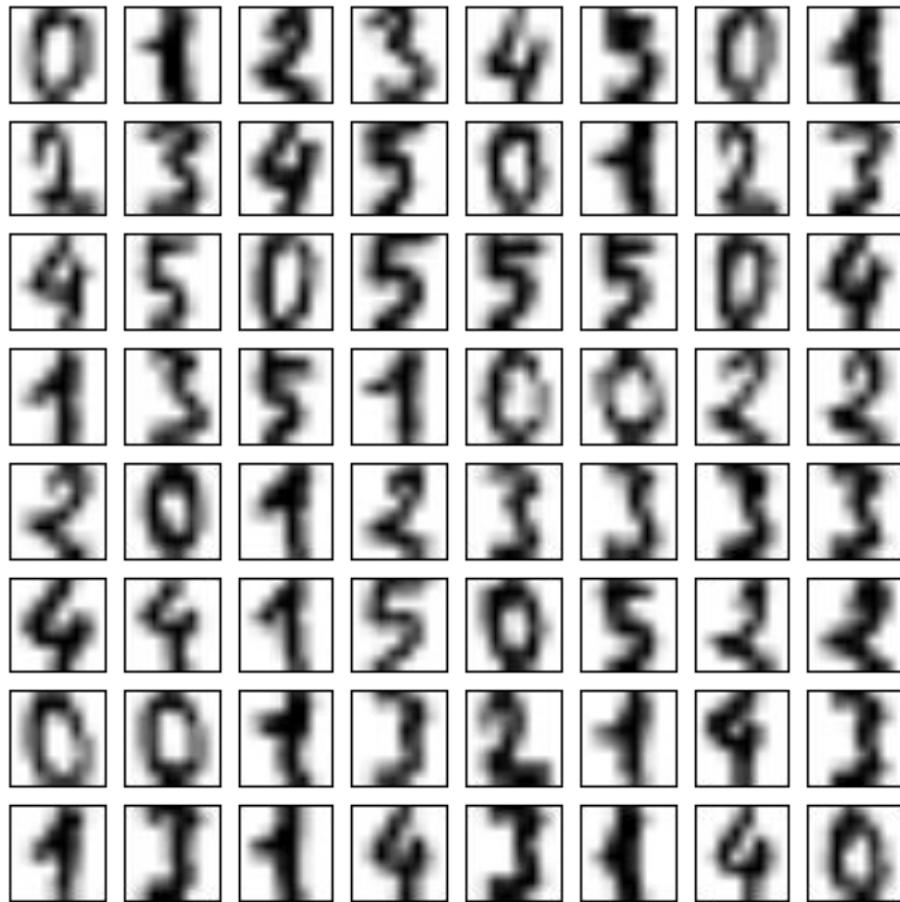
Example: Hand-written Digits

For an example of where this might be useful, let's look at an interesting visualization of some hand-written digits data. This data is included in scikit-learn, and consists of nearly 2000 8x8 thumbnails showing various hand-written digits.

For now, let's start by downloading the digits data and visualizing several of the example images with `plt.imshow()`:

```
# load images of the digits 0 through 5 and visualize several of them
from sklearn.datasets import load_digits
digits = load_digits(n_class=6)

fig, ax = plt.subplots(8, 8, figsize=(6, 6),
                      subplot_kw=dict(xticks=[], yticks=[]))
for i, axi in enumerate(ax.flat):
    axi.imshow(digits.images[i], cmap='binary')
```



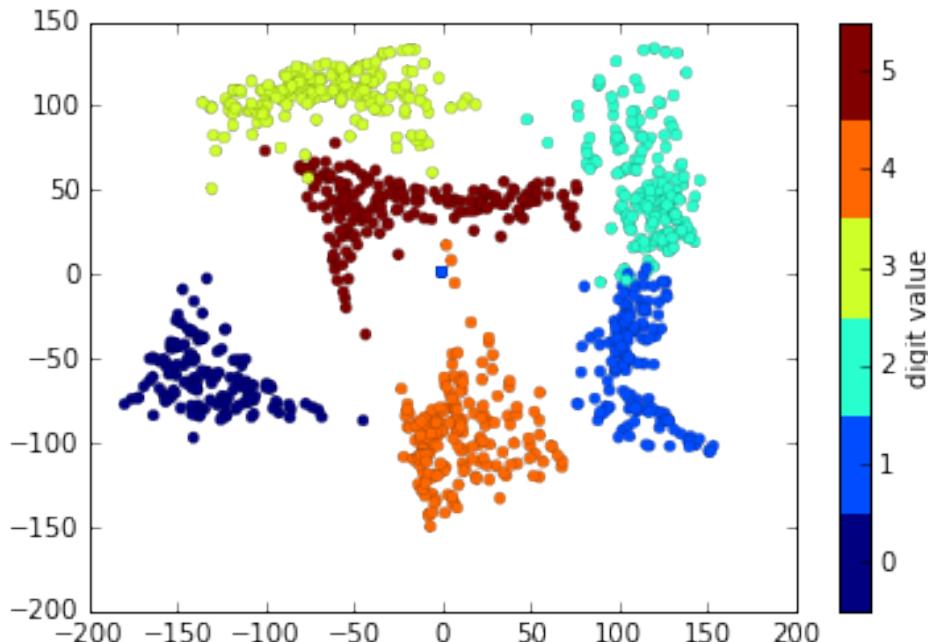
Because each digit is defined by the hue of its 64 pixels, we can consider each digit to be a point lying in 64-dimensional space: each dimension represents the brightness of one pixel. But visualizing relationships in such high-dimensional spaces can be extremely difficult. One way to approach this 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, and we will discuss it in more detail in Section X.X.

Deferring the discussion of these details, let's take a look at a two-dimensional manifold learning projection of these digits data:

```
# project the digits into 2 dimensions using Isomap
from sklearn.manifold import Isomap
iso = 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:

```
# plot the results
plt.scatter(projection[:, 0], projection[:, 1], lw=0.1,
            c=digits.target, cmap=plt.cm.get_cmap('jet', 6))
plt.colorbar(ticks=range(6), label='digit value')
plt.ylim(-0.5, 5.5)
```



This plot gives us some interesting insights: the nonlinear mapping shows that samples of handwritten digits from 0 to 5 can be quite well separated based on their pixel values. The projection also gives us some insight: for example, the ranges of 5 and 3 nearly overlap in this projection, indicating that some hand-written 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 confused. This observation agrees with our intuition, because 5 and 3 look much more similar than do 0 and 1.

We'll return to manifold learning and to digit classification in a later recipe.

Other Colorbar Options

We saw in the last example that we can set the `ticks` and `label` attributes of the colorbar when it is constructed. There are many more options available for colorbars:

their size, shape, location, and orientation can be adjusted, as can the style of their ticks, labels, and color differentiations. For much more information on these, see the docstring of `plt.colorbar` or refer to the online matplotlib documentation.

Multiple Subplots

It is often desirable to show multiple subplots within a single figure. These subplots might be insets, grids of plots, or other more complicated layouts. Here we'll explore four routines for creating subplots in matplotlib.

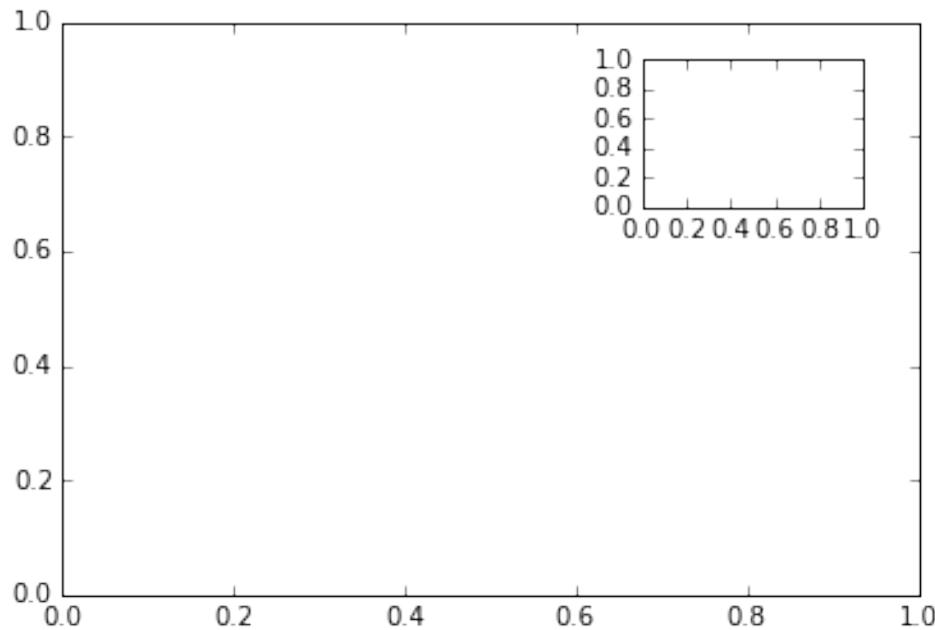
`plt.axes`: subplots by-hand

The 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 which fills the entire figure. `plt.axes` also takes an optional argument which 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 the figure 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):

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

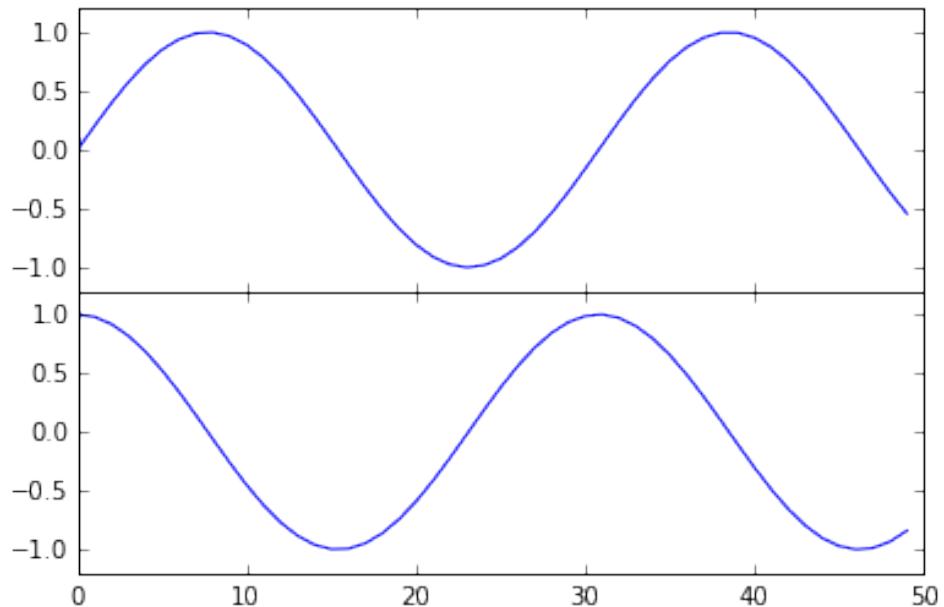
ax1 = plt.axes() # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```



The equivalent of this command within the object-oriented interface is `fig.add_axes()`. Let's use this to create two side-by-side axes:

```
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));
```

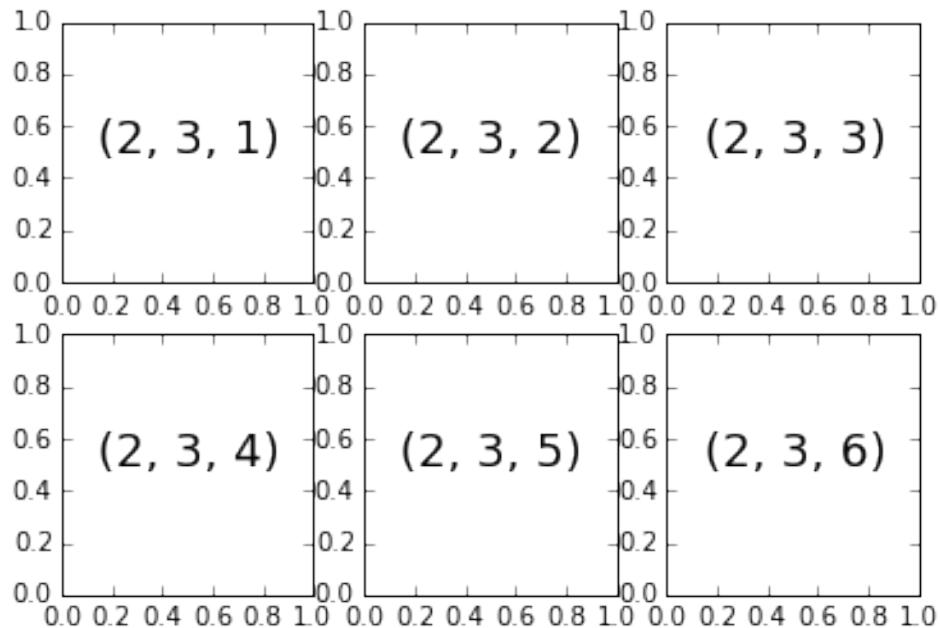


We see that we now have two axes (the top with no tick labels) which are just touching: The bottom of the upper panel (at position 0.5) matches the top of the lower panel (at position 0.1 + 0.4).

plt.subplot: simple grids of subplots

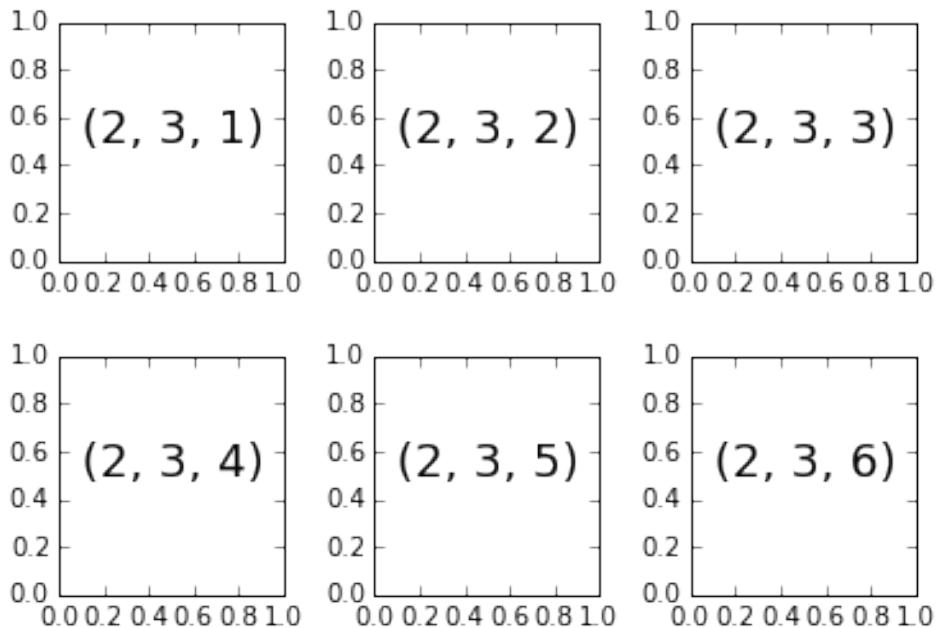
Aligned columns or rows of subplots are a common-enough need that matplotlib has several convenience routines which make them easy to create. The lowest-level of these is `plt.subplot()`, which creates a single subplot within a grid. The `plt.subplot()` command takes three integer arguments: the number of rows, the number of columns, and the index of the plot to be created in this scheme, which runs from the upper left to the bottom right. For example:

```
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')
```



The spacing between these plots can be adjusted by using the command `plt.subplots_adjust`. Here we'll use the equivalent object-oriented command, `fig.add_subplot()`:

```
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')
```



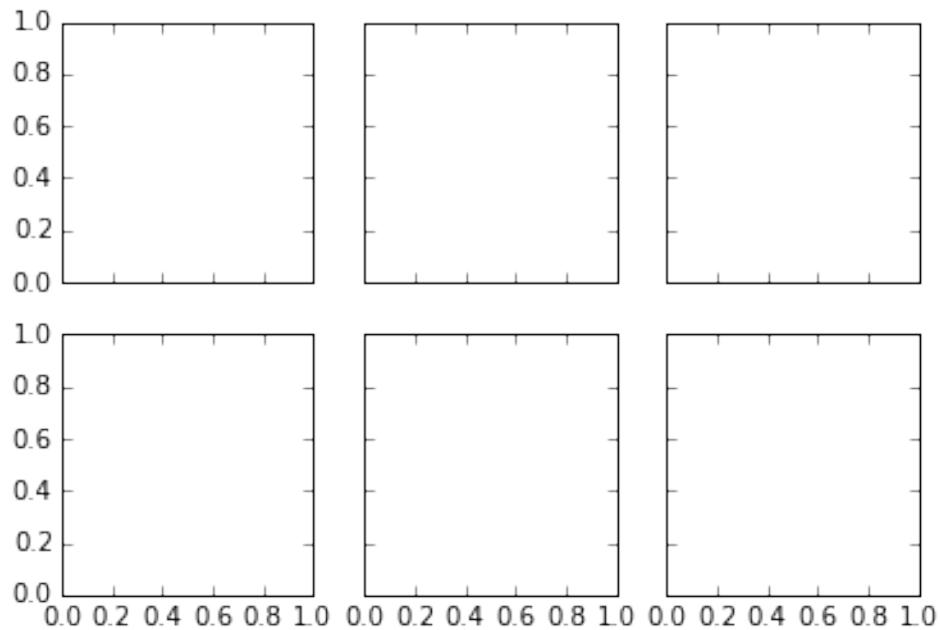
We've used the `hspace` and `wspace` arguments of `plt.subplots_adjust`, which specify the spacing along the height and width of the figure, in units of the subplot size (in this case, the space is 40% of the subplot width and height).

`plt.subplots`: the whole grid in one go

Even the above can become quite tedious when 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 `s` at the end of `subplots`). Rather than creating a single subplot, this function creates a full grid of subplots 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:

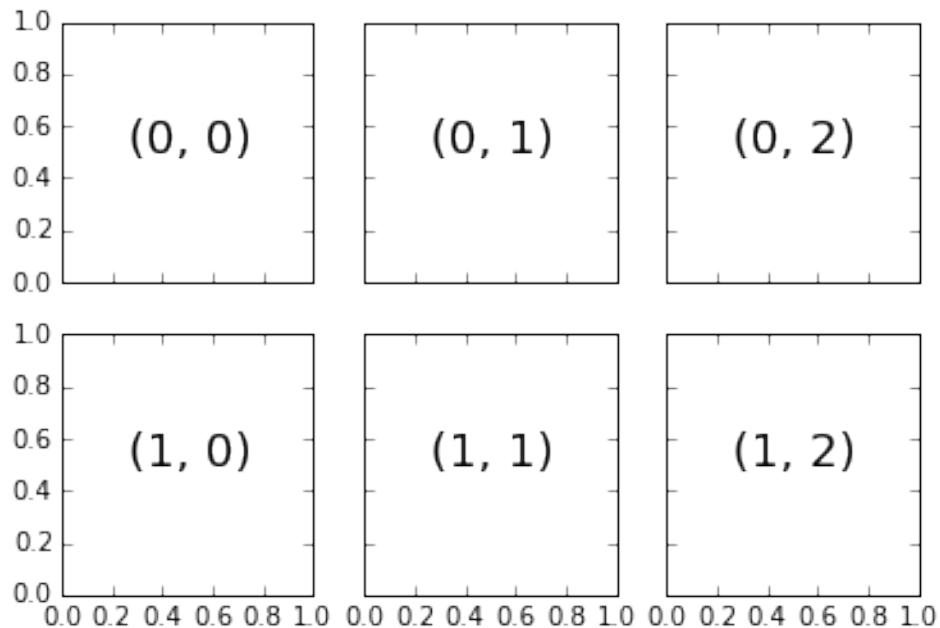
Here we'll create a 2×3 grid of subplots, where all axes in the same row share their y-axis scale, and all axes in the same column share their x-axis scale.

```
fig, ax = plt.subplots(2, 3, sharex='col', sharey='row')
```



Note that by specifying `sharex` and `sharey`, we've automatically removed inner labels on 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 using standard array indexing notation:

```
# axes are in a 2D 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
```



In comparison to `plt.subplot()`, `plt.subplots()` is more consistent with Python's conventional 0-based indexing.

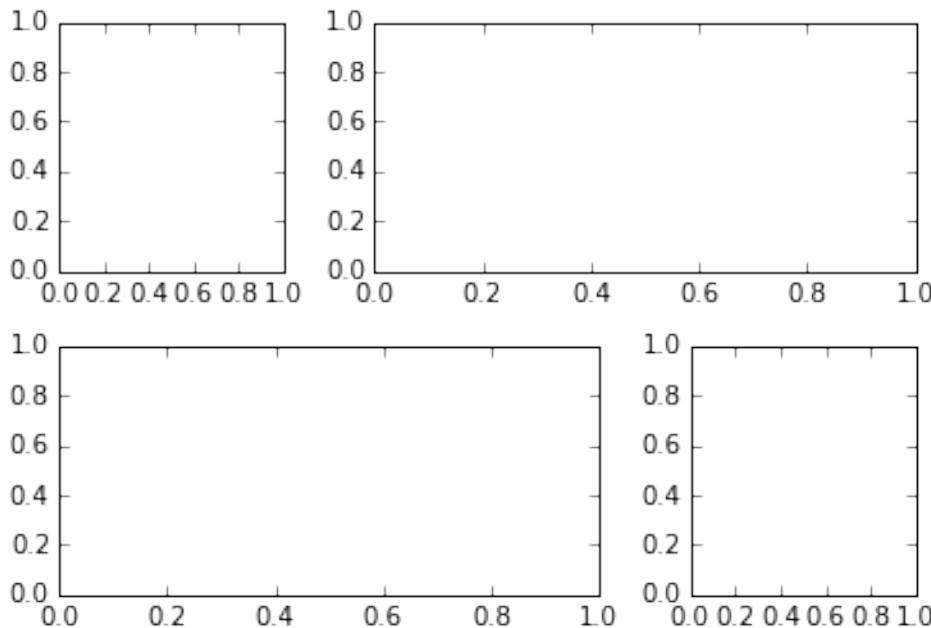
`plt.GridSpec`: more complicated arrangements

Sometimes a regular grid is not what you desire. If you'd like some subplots to span multiple rows and columns, `plt.GridSpec()` is what you're after. The `plt.GridSpec()` object does not create a plot by itself; it is simply a convenient interface which is recognized by the `plt.subplot()` command. For example, a gridspec for a grid of 2 rows and 3 columns with some specified width and height space looks like this:

```
grid = plt.GridSpec(2, 3, wspace=0.4, hspace=0.3)
```

We can now use slicing notation on the grid to specify plots and their extent:

```
plt.subplot(grid[0, 0])
plt.subplot(grid[0, 1:3])
plt.subplot(grid[1, 0:2])
plt.subplot(grid[1, 2]);
```



This type of flexible grid alignment has a wide range of uses. I most often use it when creating multi-axes histogram plots like the following:

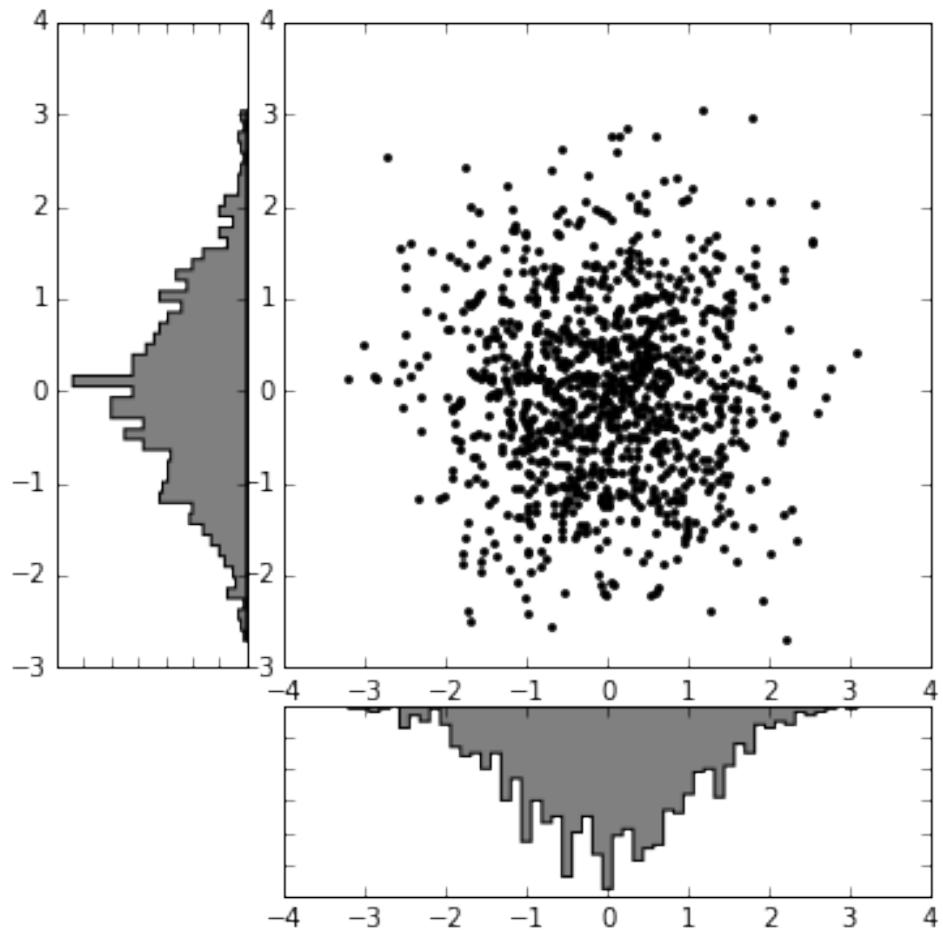
```
# Create some normally distributed data
x, y = np.random.randn(2, 1000)

# 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', ms=3)

# histogram on the attached axes
x_hist.hist(x, 50, histtype='stepfilled',
            orientation='vertical', color='gray')
x_hist.invert_yaxis()

y_hist.hist(y, 50, histtype='stepfilled',
            orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```



This type of distribution plotted alongside its margins is common enough that it has its own plot type in the Seaborn package; see Section X.X for more details.

Text and Annotation

Creating a good visualization is about guiding the reader so that the figure tells a story. Sometimes, this story can be told visually, without the need for added text. Other times, small textual cues and labels are a necessity.

Perhaps the 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 and annotate it to help convey interesting information.

The Cost of Storage over Time

For this example, we'll take a look at the cost of hard drive storage with time. This data was downloaded from <http://www.mkomodo.com/cost-per-gigabyte-update>. We'll start by using pandas to read-in the tab-separated data:

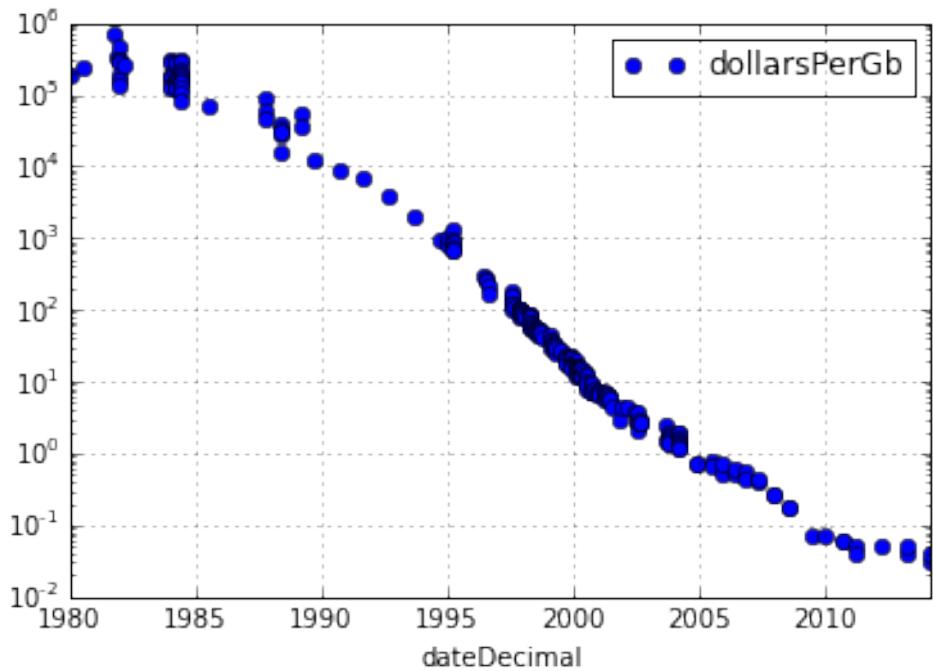
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

data = pd.read_csv('memory-price.tsv', sep='\t')
data.head()
```

	dateDecimal	date	driveInfo	sizeInMb	sizeInGb	cost	dollarsPerGb	predictedDollarsPerGb	logCostPerGb	price
0	1980.00	1980 January	Morrow Designs	26.0	0.026	5000	193000	1974730.74	5.29	6.3
1	1980.50	1980 July	North Star	18.0	0.018	4199	233000	1480897.96	5.37	6.1
2	1981.67	1981 September	Apple	5.0	0.005	3500	700000	756656.41	5.85	5.8
3	1981.83	1981 November	Seagate	5.0	0.005	1700	340000	687444.75	5.53	5.8
4	1981.92	1981 December	VR Data Corp.	6.3	0.006	2895	460000	655250.47	5.66	5.8

We can use Pandas' simple plotting function (see Section X.X) to visualize the change in cost per GB with time:

```
data.plot('dateDecimal', 'dollarsPerGb', logy=True,
          linestyle='none', marker='o');
```

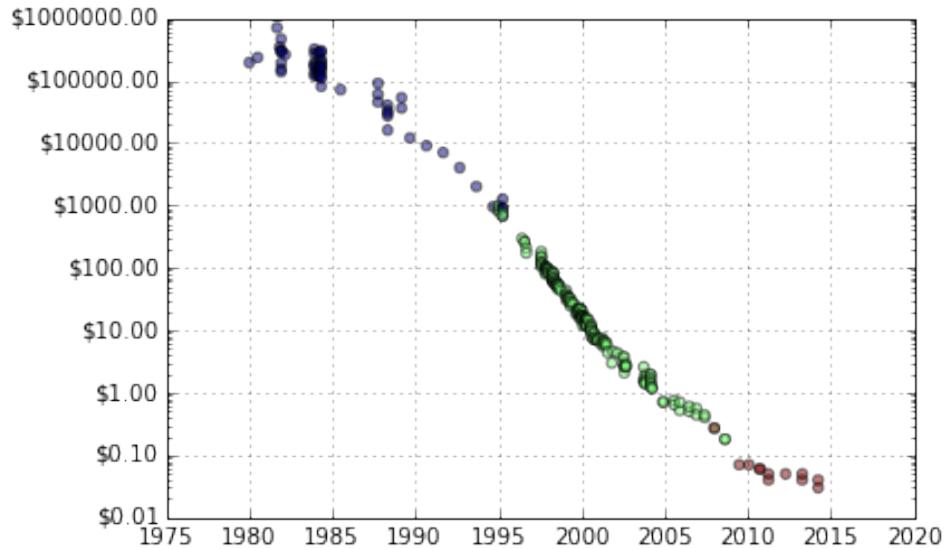


But let's think about how we can better represent and communicate the content of this data, using textual annotations. Let's start by coloring the points according to the size of the storage device. We'll also change the y-axis formatter to show that the value is dollars:

```
fig, ax = plt.subplots()
ax.set_yscale('log')
ax.grid(True)

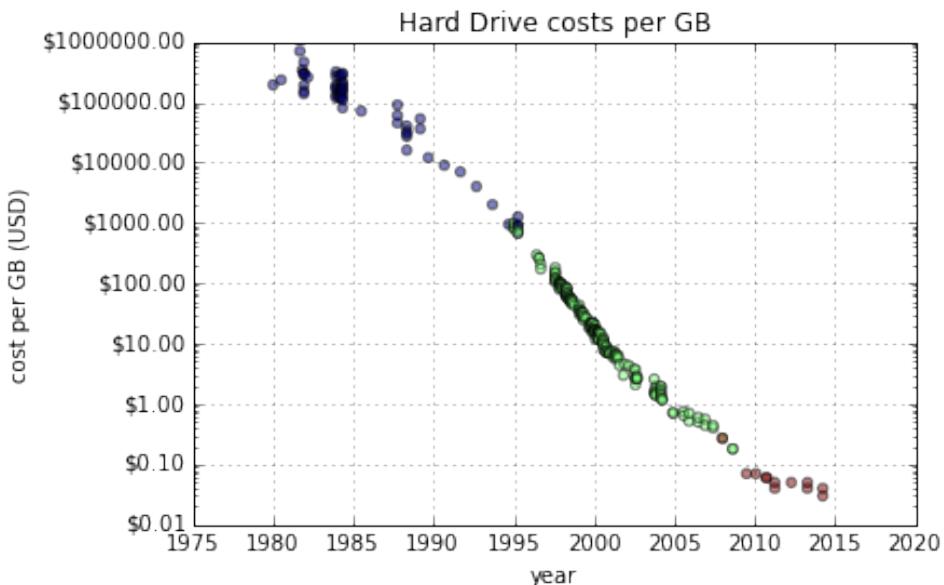
color = np.log10(data['sizeInGb']) // 3
color[np.isnan(color)] = -1 # missing data

ax.scatter(data['dateDecimal'], data['dollarsPerGb'],
           c=color, alpha=0.5, cmap='jet')
ax.yaxis.set_major_formatter(plt.FormatStrFormatter('$%.2f'));
```



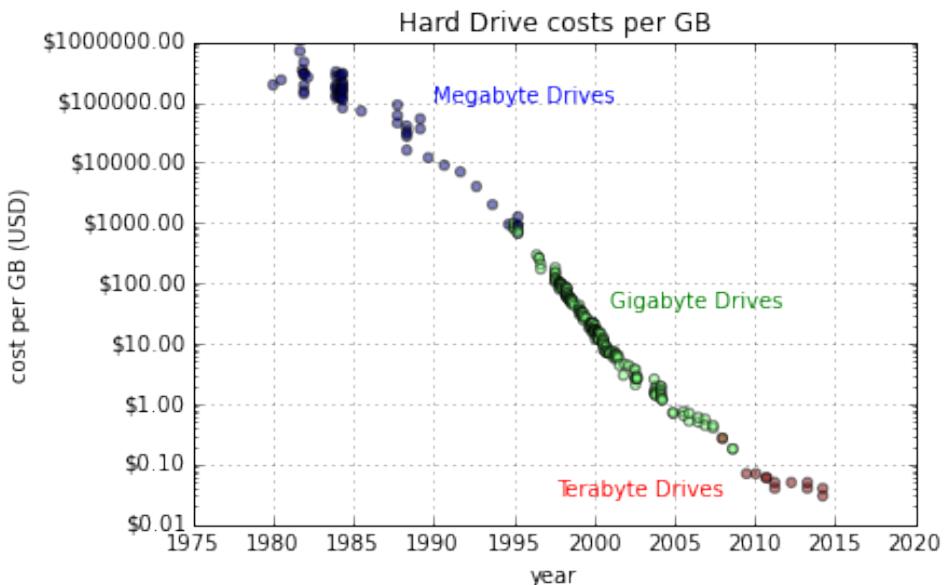
Now we can add an x label and a title to the plot. For axes methods, these are `set_xlabel()`, `set_ylabel()`, and `set_title()`. The equivalent plt functions are `plt.xlabel()`, `plt.ylabel()`, and `plt.title()`.

```
ax.set_xlabel('year')
ax.set_ylabel('cost per GB (USD)')
ax.set_title('Hard Drive costs per GB')
fig
```



Finally, we'd like to give some indication of what the colors mean. We could do this using a colorbar (see Section X.X), but instead we might like to manually insert some text which shows the meaning. We can do this using the `ax.text()` method as follows:

```
ax.text(1990, 1E5, 'Megabyte Drives', color='blue')
ax.text(2001, 40, 'Gigabyte Drives', color='green')
ax.text(2008, 0.03, 'Terabyte Drives', color='red', ha='right')
fig
```



The `ax.text` method takes an x position, a y position, a string, and then optional keywords specifying the color, size, style, and alignment of the text. Above we used `ha='right'`, where `ha` is short for *horizontal alignment*. See the docstring of `plt.text()` for more information on available options.

Transforms and Text Position

Above we've anchored our text annotations to data locations. Sometimes it's preferable to anchor the text to a position on the axes or figure, independent of the data. In matplotlib, this is done by modifying the *transform*.

Any graphics display framework needs some scheme for translating between coordinate systems. For example, a data point at $(x, y) = (1, 1)$ needs to somehow be represented at a certain location on the figure, which in turn needs to be represented in pixels on the screen. Such coordinate transformations are mathematically relatively straightforward, and matplotlib has a well-developed set of tools that it uses internally to perform these transformations, which can be explored in the `matplotlib.transforms` submodule.

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 pre-defined transforms which can be useful in this situation:

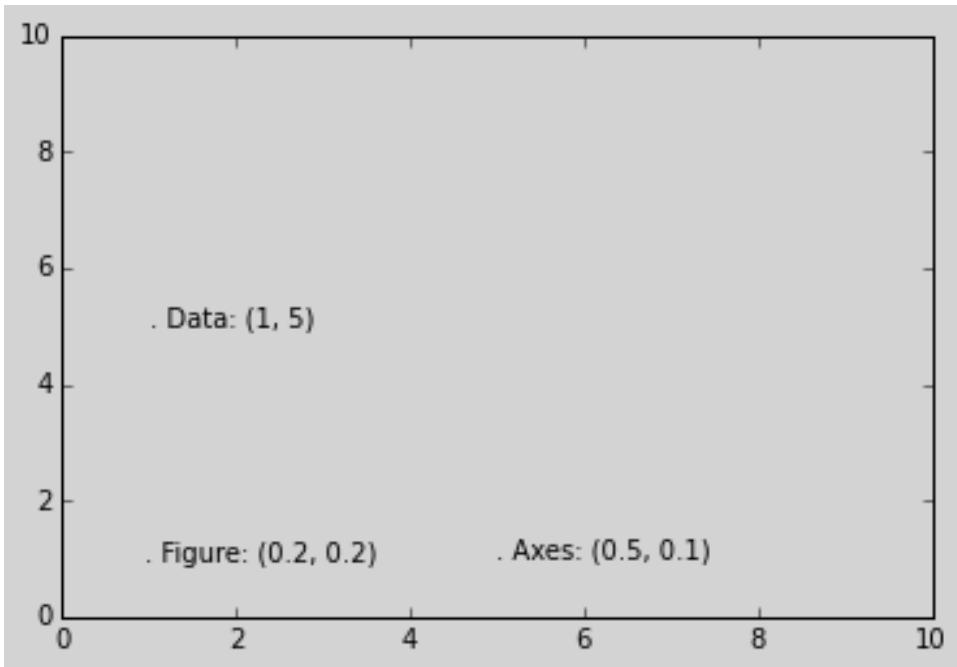
- `ax.transData`: transform associated with data coordinates
- `ax.transAxes`: transform associated with the axes (in units of axes dimensions)

- `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 transforms:

```
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);
```

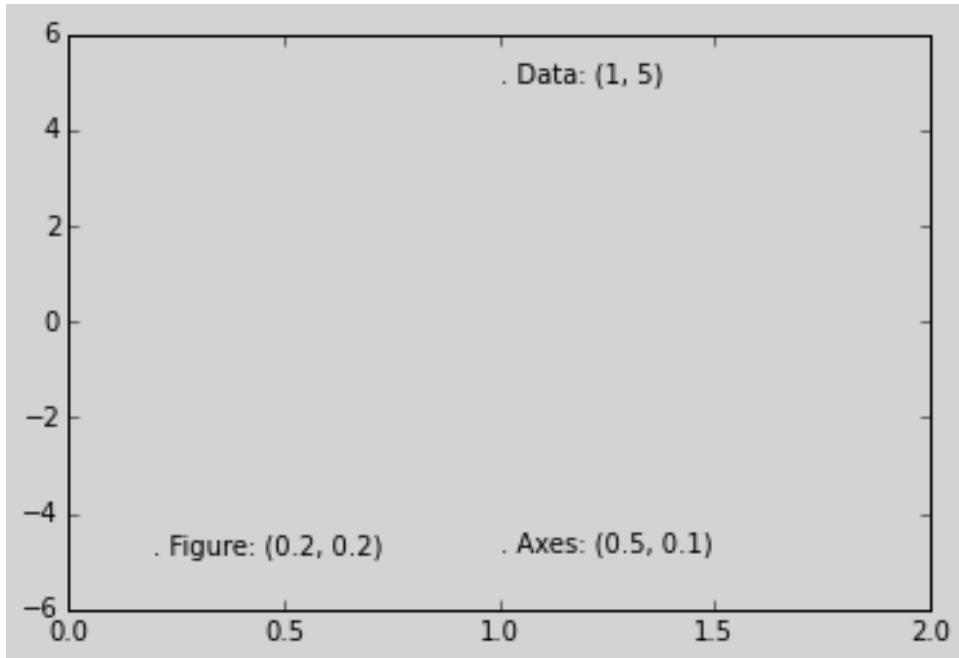


Note that by default, the text is aligned above and to the left of the specified coordinates: here the “.” at the beginning of each string will approximately mark the given coordinate location.

The `transData` coordinates give the usual data coordinates associated with the x and y axis labels. The `transAxes` coordinates give the location from the bottom-left corner 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 `transData` coordinates that will be affected, while the others remain stationary:

```
ax.set_xlim(0, 2)
ax.set_ylim(-6, 6)
fig
```



This behavior can be seen more clearly by opening an interactive figure window and changing the axes limits interactively (in IPython notebook, this will require changing to an interactive backend: running the `%matplotlib` magic function will switch from the inline backend to the default interactive backend for your system, assuming one is available).

Alternatively, you can explore this within the notebook itself using the `mpld3` package, which creates an interactive browser-based D3js view of a matplotlib figure:

```
import mpld3
mpld3.display(fig)
```

IMAGE TO COME

In the browser, you can click the arrow button in the bottom-left corner to enable panning and zooming.

Arrows and Annotation

Another useful annotation mark is the simple arrow.

Drawing arrows in matplotlib is often much harder than one would wish. While there is a `plt.arrow()` function available, I'd not suggest using it: the arrows it creates are SVG objects which will be subject to the varying aspect ratio of your plots, and the result are rarely what the user wishes. Instead, I'd suggest using the `plt.annotate()` function. This function creates some text and an arrow, and the arrows can be very flexibly specified.

Below is an example of using `annotate` with several options

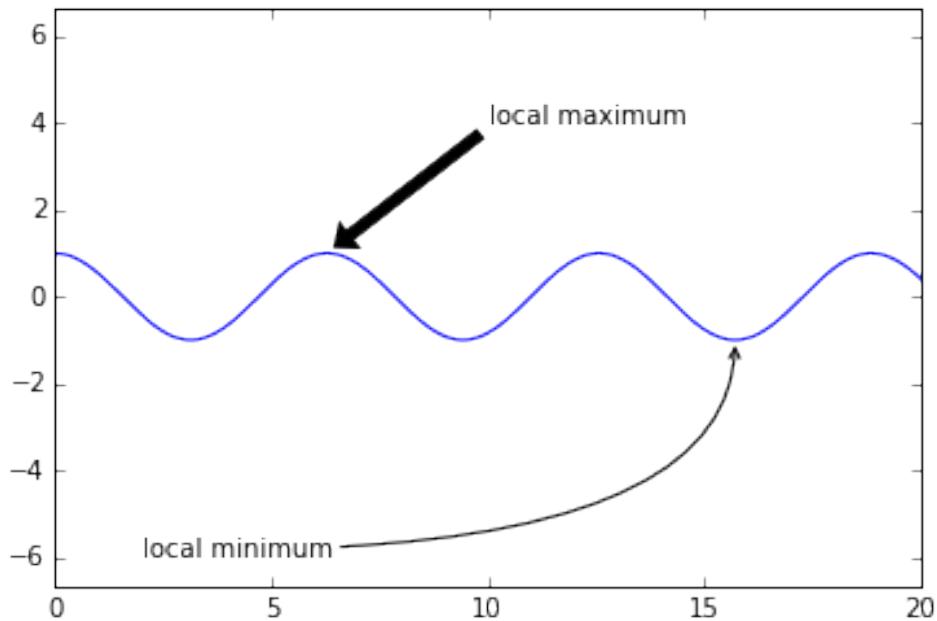
```
import warnings
warnings.simplefilter('ignore')

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"));
```



There are numerous options available in the `arrowprops` dictionary. Rather than enumerating all of the options, it's probably more useful to quickly show some of the possibilities. Following is a nice demonstration of many available options, adapted from the official matplotlib documentation. Notice that within `arrowprops`, the coordinates can be expressed in terms of data transforms or axes transforms, as we mentioned above:

```
# Adapted from http://matplotlib.org/examples/pylab_examples/annotation_demo2.html
from matplotlib.patches import Ellipse

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 4))

# plot a line on the first axes
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = ax1.plot(t, s, lw=3, color='purple')
ax1.axis([-1, 5, -4, 3])

# add an ellipse to the second axes
el = Ellipse((2, -1), 0.5, 0.5)
ax2.add_patch(el)
ax2.axis([-1, 5, -5, 3])

# Now for some annotations
ax1.annotate('arrowstyle', xy=(0, 1), xycoords='data',
            xytext=(-50, 30), textcoords='offset points',
            arrowprops=dict(arrowstyle='triangle-left',
                           connectionstyle='arc3,rad=15',
                           fill='black',
                           lw=1.5))
ax1.annotate('angle', xy=(10, 4), xycoords='data',
            xytext=(15, 0), textcoords='data',
            arrowprops=dict(angle=15,
                           arrowstyle='triangle-right',
                           connectionstyle='arc3,rad=15',
                           fill='black',
                           lw=1.5))
ax1.annotate('fancy', xy=(15, -1), xycoords='data',
            xytext=(15, -4), textcoords='data',
            arrowprops=dict(arrowstyle='triangle-right,fill=white,stroke-width=2',
                           connectionstyle='arc3,rad=15',
                           facecolor='black',
                           lw=2))
ax1.annotate('dash', xy=(15, -2), xycoords='data',
            xytext=(15, -5), textcoords='data',
            arrowprops=dict(arrowstyle='triangle-right,dash=[5, 5]',
                           connectionstyle='arc3,rad=15',
                           dasharray=[5, 5],
                           fill='black',
                           lw=1.5))
ax1.annotate('head', xy=(15, -3), xycoords='data',
            xytext=(15, -6), textcoords='data',
            arrowprops=dict(arrowstyle='triangle-right,head_width=10,head_length=10',
                           connectionstyle='arc3,rad=15',
                           fill='black',
                           lw=1.5))
ax1.annotate('bar', xy=(15, -4), xycoords='data',
            xytext=(15, -7), textcoords='data',
            arrowprops=dict(arrowstyle='triangle-right,bar=True,bar_angle=45',
                           connectionstyle='arc3,rad=15',
                           fill='black',
                           lw=1.5))
ax1.annotate('angle2', xy=(15, -5), xycoords='data',
            xytext=(15, -8), textcoords='data',
            arrowprops=dict(arrowstyle='triangle-right,angle2=15',
                           connectionstyle='arc3,rad=15',
                           fill='black',
                           lw=1.5))
ax1.annotate('angle3', xy=(15, -6), xycoords='data',
            xytext=(15, -9), textcoords='data',
            arrowprops=dict(arrowstyle='triangle-right,angle3=15',
                           connectionstyle='arc3,rad=15',
                           fill='black',
                           lw=1.5))
```

```

        arrowprops=dict(arrowstyle="->"))

ax1.annotate('arc3', xy=(0.5, -1), xycoords='data',
             xytext=(-30, -30), textcoords='offset points',
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3,rad=.2"))

ax1.annotate('arc #1', xy=(1., 1), xycoords='data',
             xytext=(-40, 30), textcoords='offset points',
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc,angleA=0,armA=30,rad=10"))

ax1.annotate('arc #2', xy=(1.5, -1), xycoords='data',
             xytext=(-40, -30), textcoords='offset points',
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc,angleA=0,armA=20,angleB=-90,armB=15,rad=7"))

ax1.annotate('angle3', xy=(2.5, -1), xycoords='data',
             xytext=(-50, -30), textcoords='offset points',
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle3,angleA=0,angleB=-90"))

ax1.annotate('angle #1', xy=(2., 1), xycoords='data',
             xytext=(-50, 30), textcoords='offset points',
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle,angleA=0,angleB=90,rad=10"))

ax1.annotate('angle #2', xy=(3., 1), xycoords='data',
             xytext=(-50, 30), textcoords='offset points',
             bbox=dict(boxstyle="round", fc="0.8"),
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle,angleA=0,angleB=90,rad=10"))

ax1.annotate('angle #3', xy=(4., 1), xycoords='data',
             xytext=(-50, 30), textcoords='offset points',
             bbox=dict(boxstyle="round", fc="0.8"),
             arrowprops=dict(arrowstyle="->",
                            shrinkA=0, shrinkB=10,
                            connectionstyle="angle,angleA=0,angleB=90,rad=10"))

ax1.annotate('angle #4', xy=(3.5, -1), xycoords='data',
             xytext=(-70, -60), textcoords='offset points',
             size=20,
             bbox=dict(boxstyle="round4,pad=.5", fc="0.8"),
             arrowprops=dict(arrowstyle="->",
                            connectionstyle="angle,angleA=0,angleB=-90,rad=10"))

ax1.annotate('', xy=(4., 1.), xycoords='data',
             xytext=(4.5, -1), textcoords='data',
             arrowprops=dict(arrowstyle+"<->",
                            connectionstyle="bar",
                            ec="k",

```

```

        shrinkA=5, shrinkB=5))

ax2.annotate('$->$', xy=(2., -1), xycoords='data',
            xytext=(-150, -90), textcoords='offset points',
            bbox=dict(boxstyle="round", fc="0.8"),
            arrowprops=dict(arrowstyle="->",
                           patchB=el,
                           connectionstyle="angle,angleA=90,angleB=0,rad=10"))

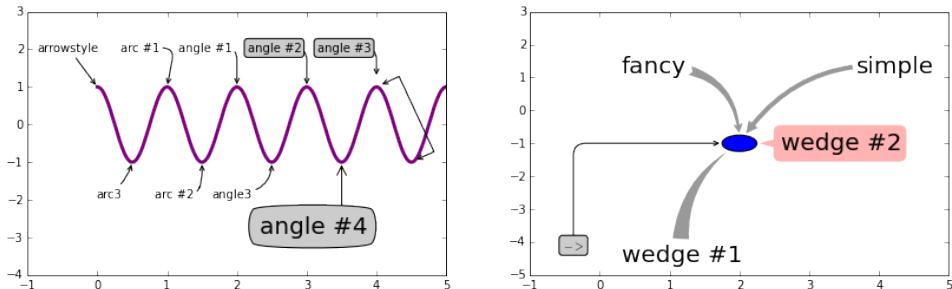
ax2.annotate('fancy', xy=(2., -1), xycoords='data',
             xytext=(-100, 60), textcoords='offset points',
             size=20,
             arrowprops=dict(arrowstyle="fancy",
                            fc="0.6", ec="none",
                            patchB=el,
                            connectionstyle="angle3,angleA=0,angleB=-90"))

ax2.annotate('simple', xy=(2., -1), xycoords='data',
             xytext=(100, 60), textcoords='offset points',
             size=20,
             arrowprops=dict(arrowstyle="simple",
                            fc="0.6", ec="none",
                            patchB=el,
                            connectionstyle="arc3,rad=0.3"))

ax2.annotate('wedge #1', xy=(2., -1), xycoords='data',
             xytext=(-100, -100), textcoords='offset points',
             size=20,
             arrowprops=dict(arrowstyle="wedge,tail_width=0.7",
                            fc="0.6", ec="none",
                            patchB=el,
                            connectionstyle="arc3,rad=-0.3"))

ax2.annotate('wedge #2', xy=(2., -1), xycoords='data',
             xytext=(35, 0), textcoords='offset points',
             size=20, va="center",
             bbox=dict(boxstyle="round", fc=(1.0, 0.7, 0.7), ec="none"),
             arrowprops=dict(arrowstyle="wedge,tail_width=1.",
                            fc=(1.0, 0.7, 0.7), ec="none",
                            patchA=None,
                            patchB=el,
                            relpos=(0.2, 0.5)));

```



Using the above example as a reference, you should be able to construct any sort of arrow or annotation that you wish.

Customizing Ticks

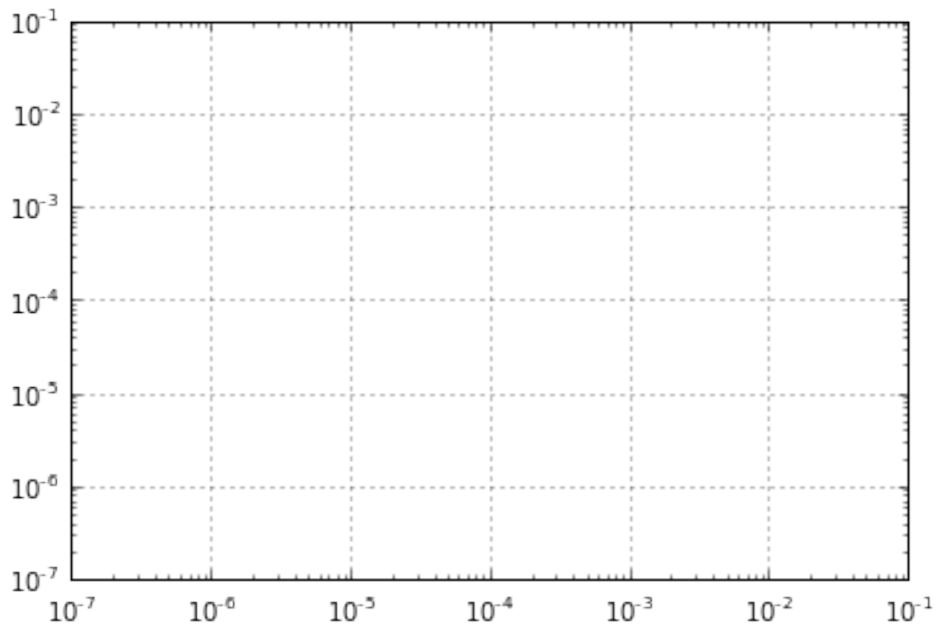
Matplotlib's default tick locators and formatters are designed to be generally sufficient in many common situations, but are in no way optimal for every plot. This Section will give several examples of adjusting the tick locations and formatting for the particular plot type you're interested in.

Before we go into examples, it will be best for us to understand the object hierarchy of matplotlib plots. Each `axes` object has attributes `xaxis` and `yaxis`, which contain all the properties of the lines, ticks, and labels that make up the axes.

Within each axis, there is the concept of a *major* tick mark, and a *minor* tick mark. As the 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:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

ax = plt.axes(xscale='log',yscale='log')
ax.grid() # add a grid on major ticks
```



We see here that each major tick shows by a large tickmark and a label, while each minor tick shows a smaller tickmark with no label.

These tick properties – locations and labels – can be customized by setting the `formatter` and `locator` objects of each axis. Let's examine these for the x axis of the above plot:

```
print(ax.xaxis.get_major_locator())
print(ax.xaxis.get_minor_locator())

<matplotlib.ticker.LogLocator object at 0x106e069d0>
<matplotlib.ticker.LogLocator object at 0x106cd1410>

print(ax.xaxis.get_major_formatter())
print(ax.xaxis.get_minor_formatter())

<matplotlib.ticker.LogFormatterMathtext object at 0x106cbd510>
<matplotlib.ticker.NullFormatter object at 0x106e02650>
```

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.

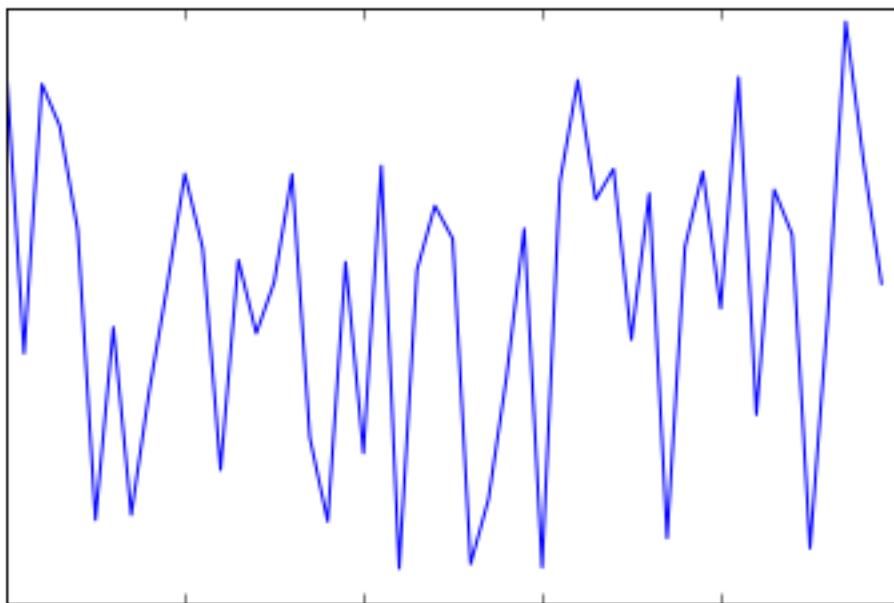
Below we'll show a few examples of setting these locators and formatters for various plots.

Removing Ticks or Labels

Sometimes for a cleaner plot, you'd like to hide ticks or labels. This can be done using `plt.NullLocator()` and `plt.NullFormatter()` respectively. Let's take a quick look at this:

```
ax = plt.axes()
ax.plot(np.random.rand(50))

ax.yaxis.set_major_locator(plt.NullLocator())
ax.xaxis.set_major_formatter(plt.NullFormatter())
```



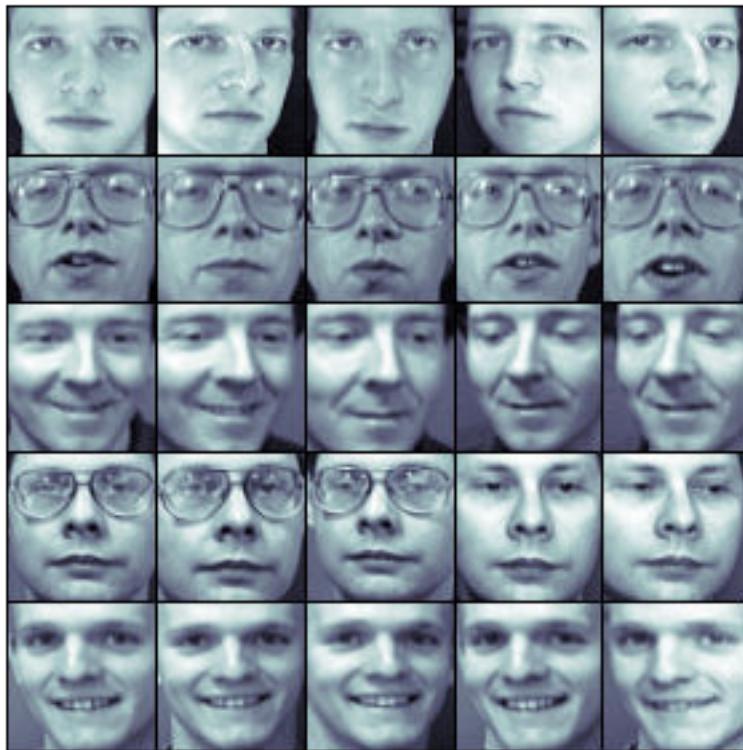
Notice that we've removed the labels (but left the ticks) from the x axis, and removed the ticks (and thus the labels as well) from the y axis. Having no ticks at all can be useful in many situations, for example when you want to show a grid of images. For an example of where removing ticks can be useful, take a look at these images of faces, often used in supervised Machine Learning problems (see Section X.X):

```
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")
```

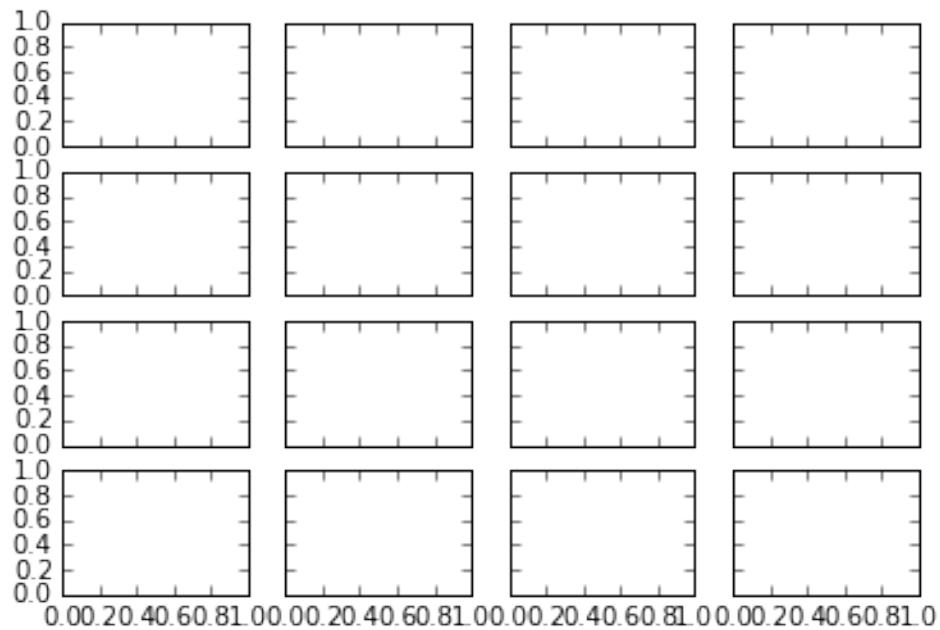


Notice that each image has its own axes, and we've set the locators to Null because the tick values (pixel number in this case) do not convey relevant information for this particular visualization.

Reducing or Increasing the Number of Ticks

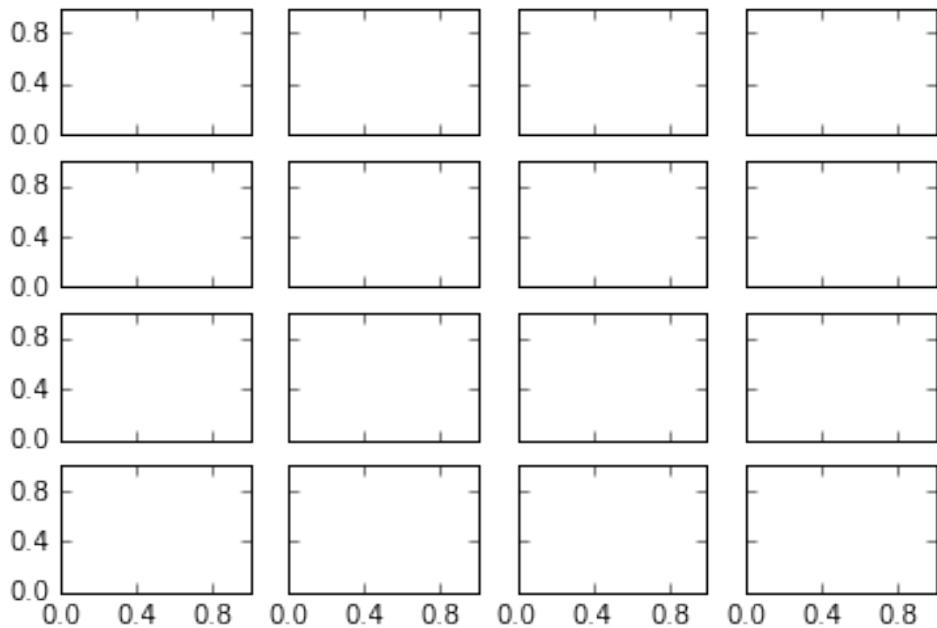
One common problem with the default settings is that smaller subplots can end up with crowded labels. We can see this in the following plot grid:

```
fig, ax = plt.subplots(4, 4, sharex=True, sharey=True)
```



Particularly for the x ticks, the numbers nearly overlap and make them nearly impossible to figure out. We can fix this with the `plt.MaxNLocator()`, which allows us to specify the maximum number of ticks which will be displayed. Given this maximum number, matplotlib will use internal logic to choose the particular tick locations:

```
# For every axis, set the x & y major locator
for axx in ax.flat:
    axx.xaxis.set_major_locator(plt.MaxNLocator(3))
    axx.yaxis.set_major_locator(plt.MaxNLocator(3))
fig
```



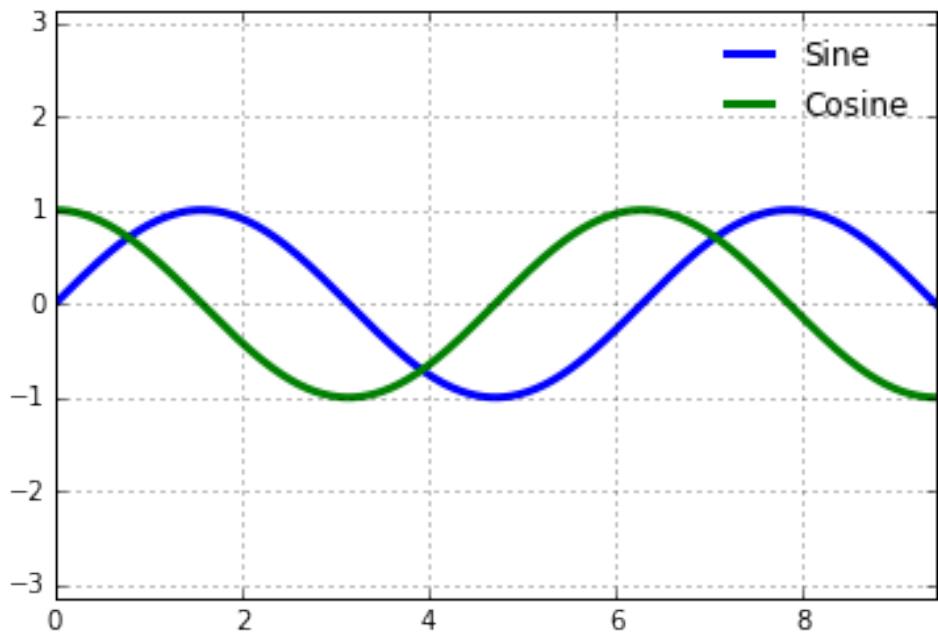
This makes things much cleaner. If you want even more control over the locations of regularly-spaced ticks, you might also use `plt.MultipleLocator`, as we'll see below.

Fancy Tick Formats

Matplotlib's default tick formatting can leave a lot to be desired: it works broadly well as a default, but sometimes you'd like to do something more. Consider the following plot, a sine and a cosine:

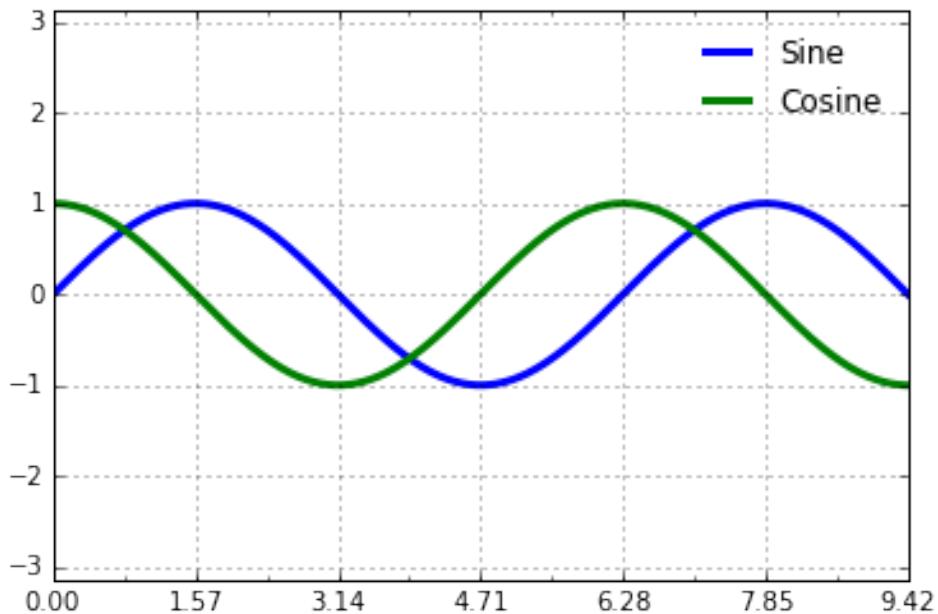
```
# Plot a sine and cosine curve
fig, ax = plt.subplots()
x = np.linspace(0, 3 * np.pi, 1000)
ax.plot(x, np.sin(x), lw=3, label='Sine')
ax.plot(x, np.cos(x), lw=3, label='Cosine')

# Set up grid, legend, and limits
ax.grid(True)
ax.legend(frameon=False)
ax.axis('equal')
ax.set_xlim(0, 3 * np.pi);
```



There are a couple changes we might like to make. First, it's more natural for this data to space the ticks and grid lines in multiples of π . We can do this by setting a `MultipleLocator`, 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 $\pi/4$:

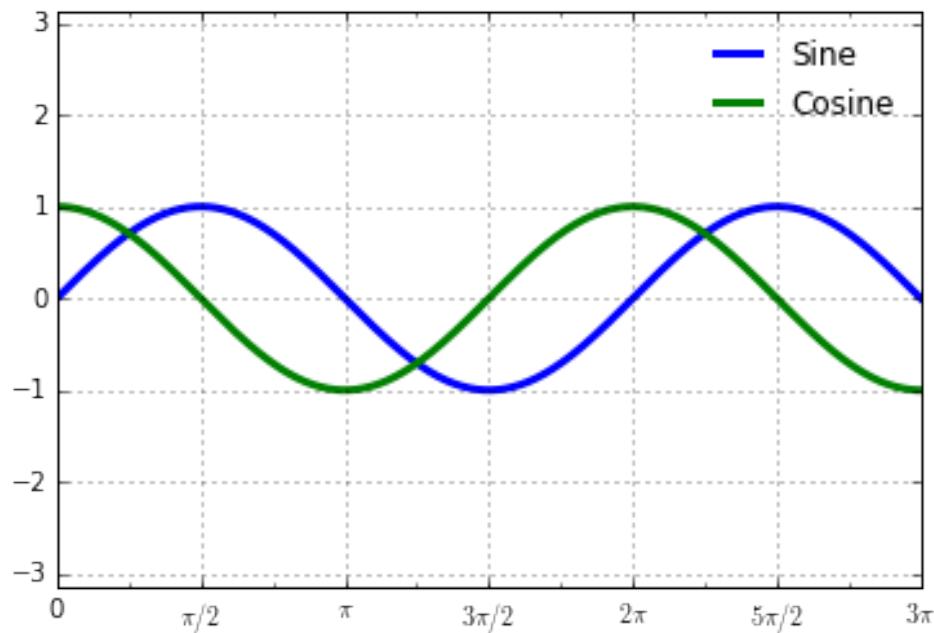
```
ax.xaxis.set_major_locator(plt.MultipleLocator(np.pi / 2))
ax.xaxis.set_minor_locator(plt.MultipleLocator(np.pi / 4))
fig
```



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:

```
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
```



This is much better! Notice that we've made use of the matplotlib's LaTeX support, specified 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()` gives you extremely fine-grained control over the appearance of your plot ticks, and comes in very handy when preparing plots for presentation or publication.

Summary of Formatters and Locators

We've mentioned a couple of the available formatters and locators. We'll finish 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 documentation. Each of the following is available in the `plt` namespace:

Locator class	Description
<code>NullLocator</code>	No ticks
<code>FixedLocator</code>	Tick locations are fixed
<code>IndexLocator</code>	locator for index plots (e.g., where $x = \text{range}(\text{len}(y))$)
<code>LinearLocator</code>	evenly spaced ticks from min to max
<code>LogLocator</code>	logarithmically ticks from min to max

Locator class	Description
MultipleLocator	ticks and range are a multiple of base
MaxNLocator	finds up to a max number of ticks at nice locations
AutoLocator	(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 labels
FuncFormatter	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

Customizing Matplotlib: Configurations and Style Sheets

Matplotlib's default plot settings are often the subject of complaint among its users. To some degree, all plotting packages have defaults that are not optimal in some situations. However, as some in the visualization community are fond of pointing out, matplotlib makes some objectively poor choices in some situations (particularly default color choices). Casting these defaults aside, however, it is also true that matplotlib can be coaxed into producing some extremely beautiful and compelling plots.

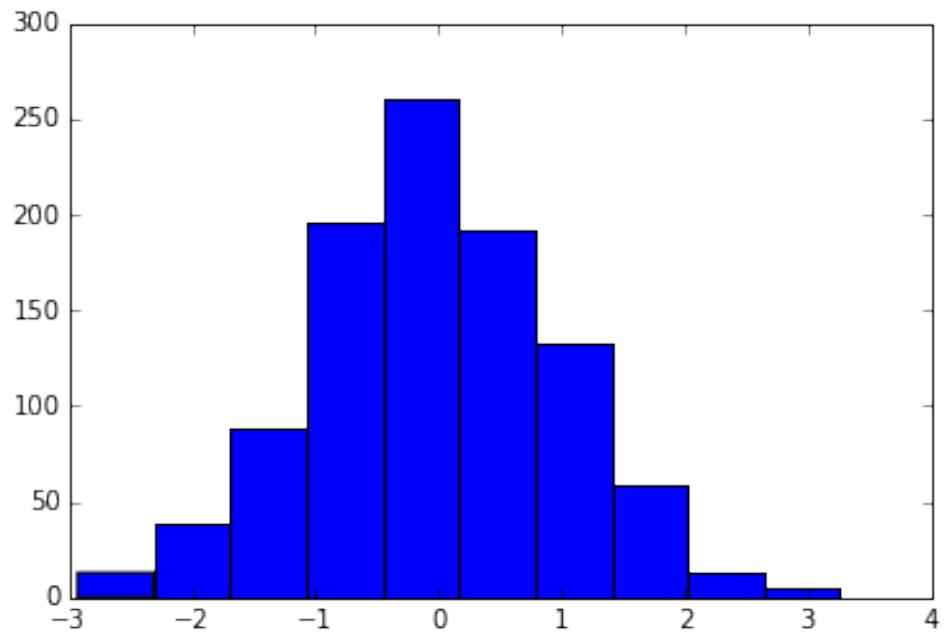
Here we'll walk through some of matplotlib's runtime configuration (`rc`) options, and take a look at the relatively new `stylesheets` feature, which contains some nice sets of default configurations.

Plot Customization By Hand

Through the recipes in this chapter, we've seen how it is possible to adjust the default plot settings to end up with something that looks a little bit nicer than the default. It's possible to do these customizations for each individual plot. For example, here is a fairly drab default histogram:

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(1000)
plt.hist(x);
```



We can adjust this by-hand to make it a much more visually pleasing plot:

```
# use a gray background
ax = plt.axes(axisbg="#E6E6E6")
ax.set_axisbelow(True)

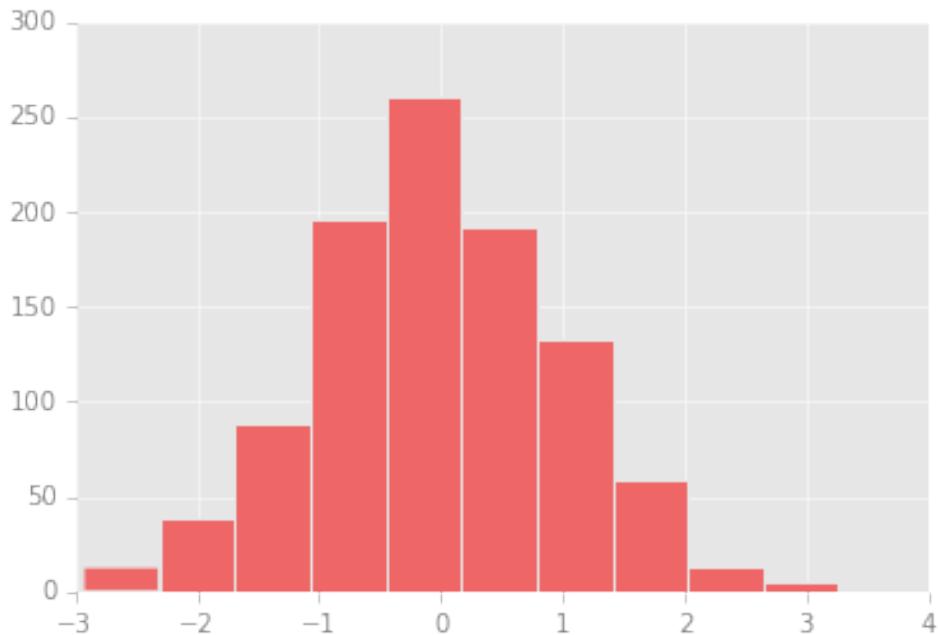
# draw solid white grid lines
plt.grid(color='w', linestyle='solid')

# 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");
```



This looks better, and you may recognize the look as inspired by the look of R's ggplot visualization package. But this took a whole lot of effort! We definitely do not want to have to do all that tweaking each time we create a plot. Fortunately, 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`) which contains the default styles for every plot element you create. This configuration can be adjusted at any time using the `plt.rc` convenience routine. We won't list the full set of `rc` parameter options here, but these are fully enumerated in the matplotlib documentation. Let's see what it looks like to modify the `rc` parameters so that our default plot will look similar to what we did above.

We'll start by saving a copy of the current `rcParams` dictionary, so we can easily reset these changes in the current session:

```
IPython_default = plt.rcParams.copy()
```

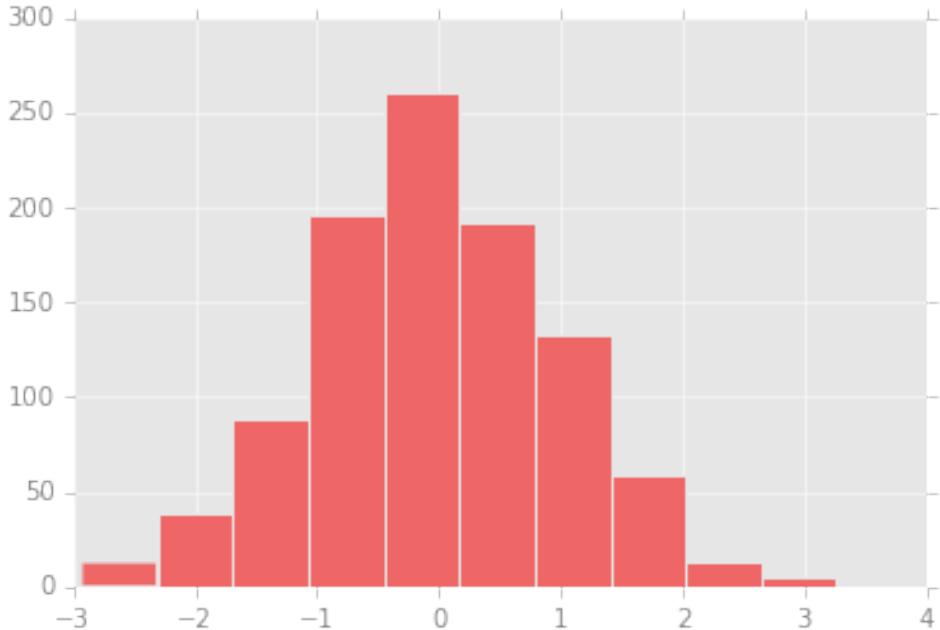
Now we can use the `plt.rc` function to change some of these settings:

```
color_cycle = ['#EE6666', '#3388BB', '#9988DD',
               '#EECC55', '#88BB44', '#FFBBBB']
plt.rc('axes', facecolor='#E6E6E6', edgecolor='none',
       axisbelow=True, grid=True, color_cycle=color_cycle)
```

```
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)
```

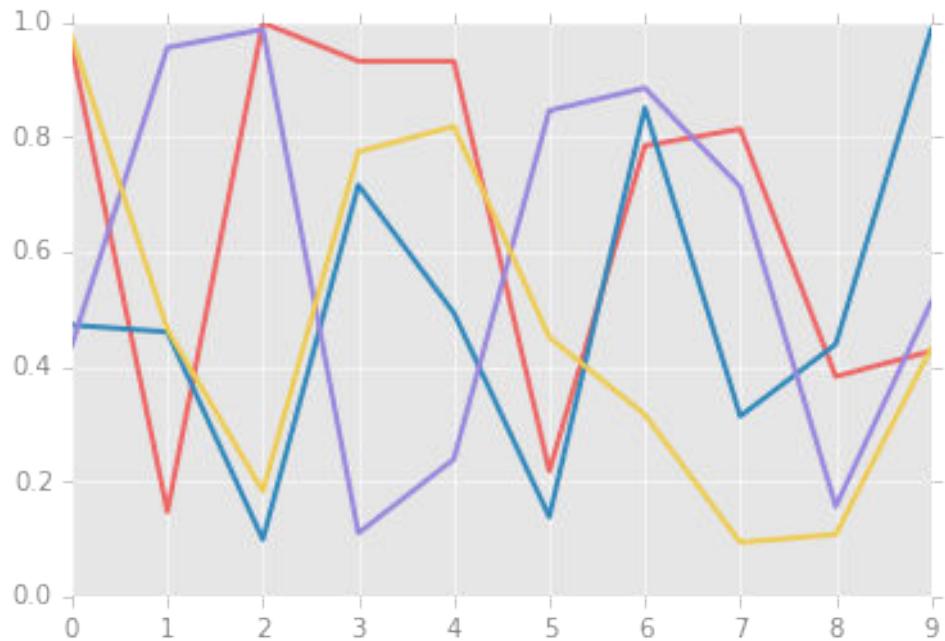
With these settings defined, we can now create a plot and see our settings in action:

```
plt.hist(x);
```



Let's see what simple line plots look like with these rc parameters:

```
for i in range(4):
    plt.plot(np.random.rand(10))
```



I think this looks *much* better than the default plot styling. If you disagree with my aesthetic sense, the good news is that you can adjust the rc parameters to suit your own tastes!

Saving Your rc Settings

If you find a particular set of rc settings that you would like to use by default every time, you can create a `matplotlibrc` file. Every time matplotlib is loaded, it looks for a file called `matplotlibrc` in the following locations:

1. The current working directory
2. A platform-specific location, (`~/.config/matplotlib/matplotlibrc` on Linux or `~/.matplotlib/matplotlibrc` on OSX)
3. `$INSTALL/matplotlib/mpl-data/matplotlibrc`, where `$INSTALL` is the path of your Python installation.

If a `matplotlibrc` file is found in any of these locations, it will be used to override the default values.

A `matplotlibrc` file is a simple text file which enumerates the settings you'd like to specify. For the above settings, it would look something like this:

```
# matplotlibrc file
axes.facecolor : E6E6E6
```

```
axes.edgecolor : none
axes.axisbelow : True
axes.grid : True
axes.color_cycle : EE6666, 3388BB, 9988DD, EECC55, 88BB44, FFBBBB

grid.color : w
grid.linestyle : solid

xtick.direction : out
xtick.color : gray
ytick.direction : out
ytick.color : gray

patch.edgecolor : E6E6E6
lines.linewidth : 2
```

With this saved to file in the appropriate location, the settings we used above would become the defaults whenever matplotlib is loaded.

While the `matplotlibrc` file may be convenient for local use, I personally avoid them. The reason is this: when you share your code, other users will probably not have the same default settings as you do. For this reason, it's probably better to either include the rc settings within your code, or to use the stylesheet feature described below.

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 style sheets are formatted similarly to the `matplotlibrc` files mentioned above, 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. You can list the available styles as follows:

```
plt.style.available
['ggplot', 'bmh', 'grayscale', 'fivethirtyeight', 'dark_background']
```

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! Alternatively, you can use the style context manager, which sets it temporarily:

```
with plt.style.context('stylename'):
    make_a_plot()
```

Let's create a function which will make two basic types of plot:

```
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(10))
ax[1].legend(['a', 'b', 'c'], loc='lower left')

```

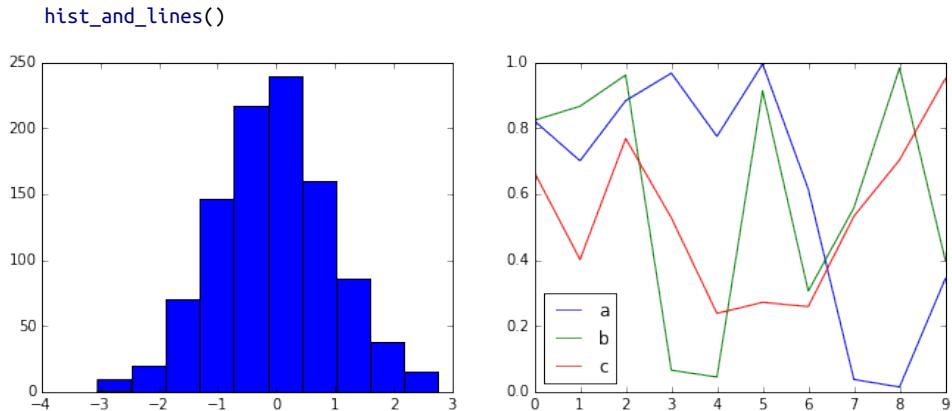
We'll use this to explore how these plots look using the various built-in styles.

Default Style

The default style is what we've been seeing so far through the book; we'll start with that. First, let's reset our runtime configuration to the notebook default:

```
# reset rcParams
plt.rcParams.update(IPython_default);
```

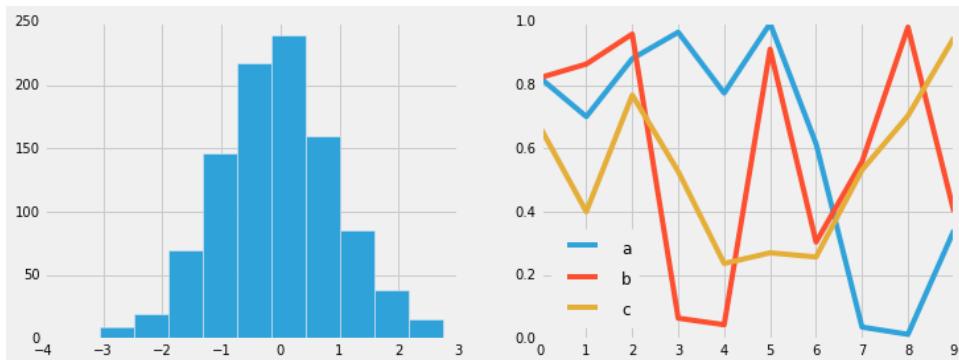
Now let's see how it looks:



Five Thirty Eight style

The "fivethirtyeight" style mimics the graphics found on the popular <http://fivethirtyeight.com> website. It is typified by bold colors, thick lines, and transparent axes.

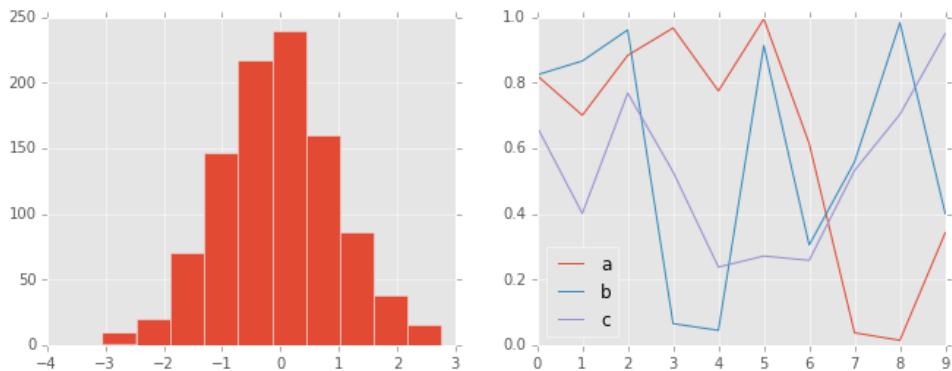
```
with plt.style.context('fivethirtyeight'):
    hist_and_lines()
```



GGPlot

The `ggplot` package in the R language is a very popular visualization tool. Matplotlib's "ggplot" style mimics the default styles from that package:

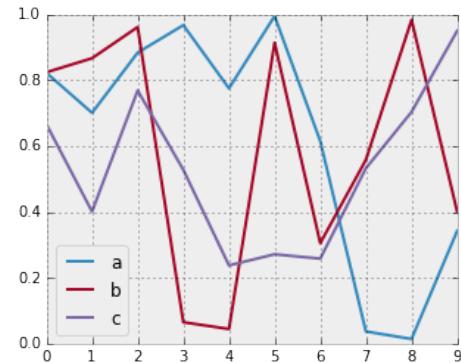
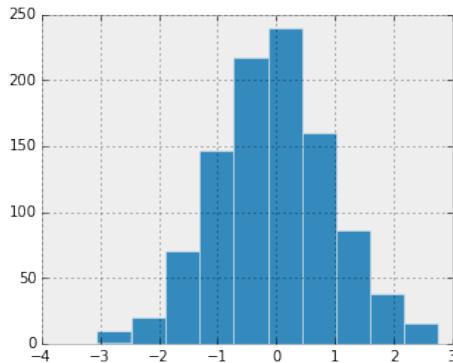
```
with plt.style.context('ggplot'):
    hist_and_lines()
```



Bayesian Methods for Hackers style

There is a very nice online textbook called *Probabilistic Programming and Bayesian Methods for Hackers*; it features figures created with matplotlib, and uses a nice set of rc parameters to create a consistent and visually-appealing style throughout the book. This style is reproduced in the "bmh" style-sheet:

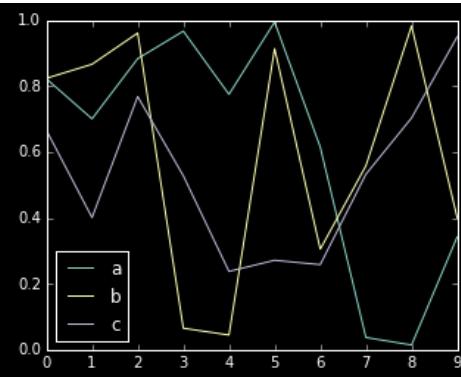
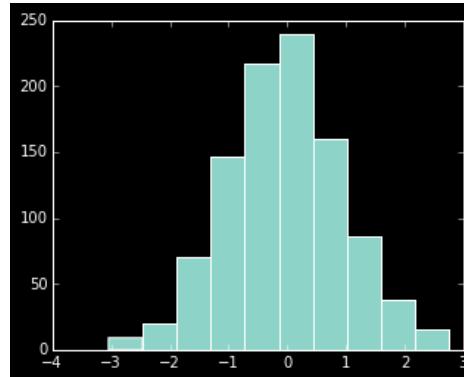
```
with plt.style.context('bmh'):
    hist_and_lines()
```



Dark Background

For figures used within presentations, it is often useful to have a dark rather than light background. The "dark_background" style provides this:

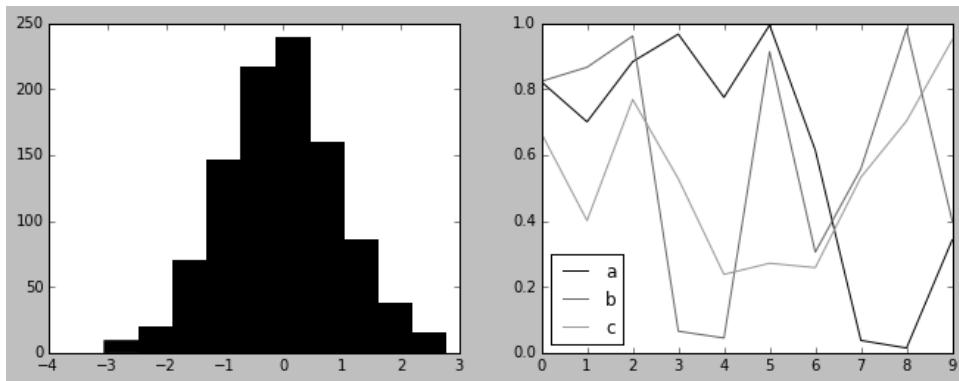
```
with plt.style.context('dark_background'):
    hist_and_lines()
```



Grayscale

Sometimes you might find yourself preparing figures for a print publication which does not accept color figures. For this, the "grayscale" style can be very useful:

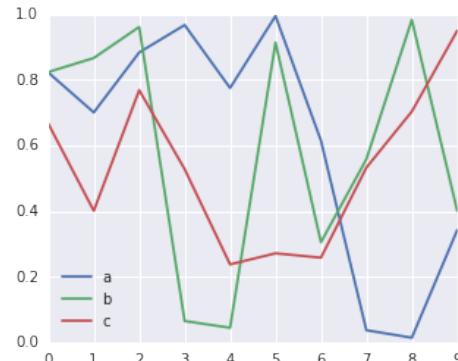
```
with plt.style.context('grayscale'):
    hist_and_lines()
```



Seaborn Style

While this is not technically a stylesheet, the Seaborn library (discussed more fully in Section X.X) comes with its own set of matplotlib defaults. These defaults can be set by calling the `set()` function within the `seaborn` namespace:

```
import seaborn
seaborn.set()
hist_and_lines()
```



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. Through the rest of the book, we'll generally use one or more of these style conventions when creating plots.

Three-dimensional Plotting in Matplotlib

Matplotlib was designed to be a two-dimensional plotting library. Around the time of the 1.0 release, some 3D plotting utilities were built on top of matplotlib's 2D display,

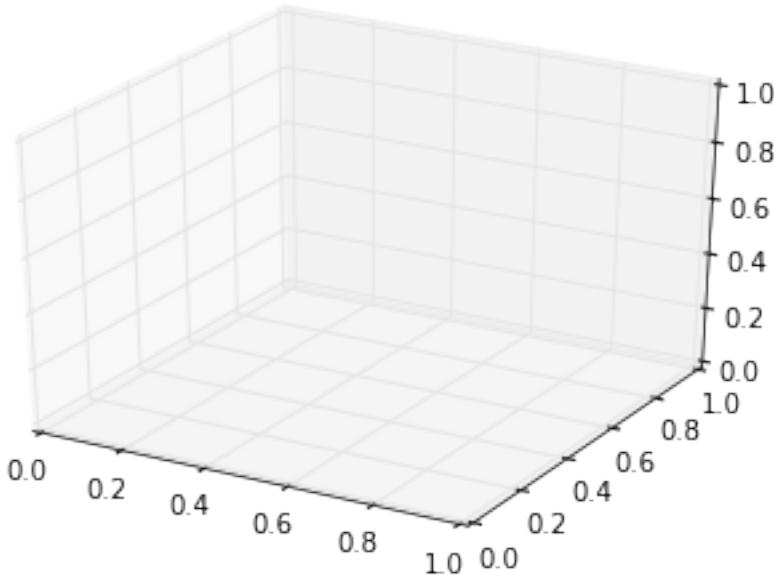
and the result is a convenient (if somewhat limited) set of tools for three-dimensional data visualization. 3D plots are enabled by importing the `mplot3d` submodule:

```
from mpl_toolkits import mplot3d
```

Once this submodule is imported, a three-dimensional axes can be created by passing the keyword `projection='3d'` to any of the normal axes creation routines:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure()
ax = plt.axes(projection='3d')
```



With this 3D axes enabled, we can now plot a variety of three-dimensional plot types, as we'll see below. Three-dimensional plotting is one of the functionalities which benefits immensely from viewing figures interactively rather than statically in the notebook; recall that to use interactive figures, you can either run a stand-alone Python script with the `plt.show()` command, or in the IPython notebook switch to the non-inline backend using the magic command `%matplotlib` rather than the usual `%matplotlib inline`.

3D Points and Lines

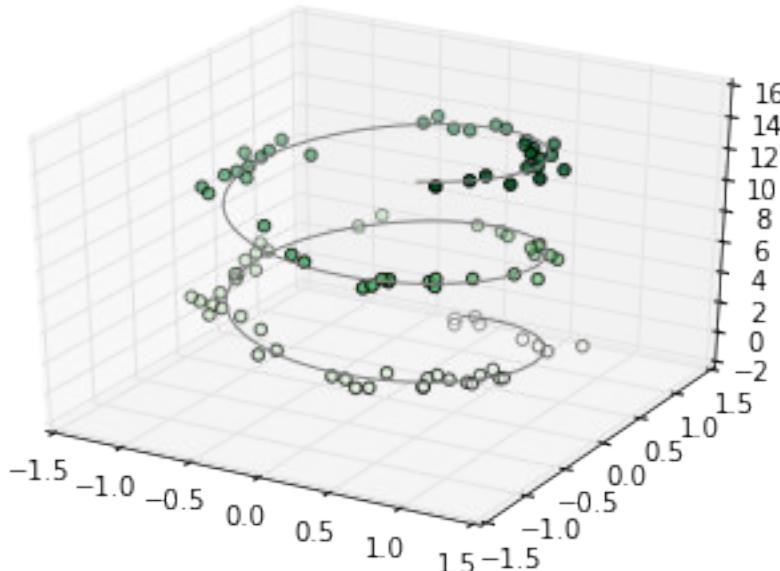
The most basic 3D plot is a line or collection of scatter plot created from sets of (x , y , z) triples. In analogy with the more common two-dimensional discussed earlier, these

can be created 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 Section X.X for more information on controlling the output. Here we'll plot a trigonometric spiral, along with some data drawn about the line:

```
ax = plt.axes(projection='3d')

# Data for a 3D line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')

# Data for 3D 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');
```



Notice that by default, the scatter points have their transparency adjusted to give a sense of depth on the page. While the 3D 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.

3D Contour Plots

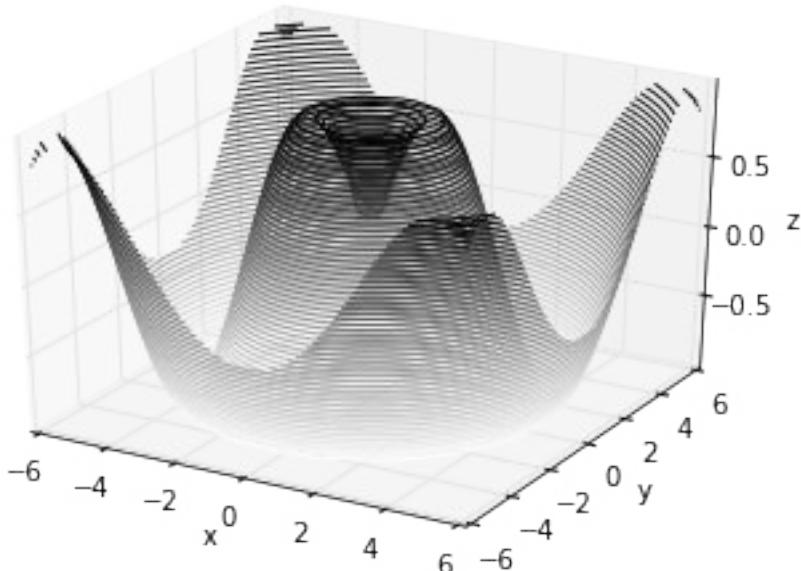
Analogous to the contour plots we explored briefly in Section X.X, `mplot3d` contains tools to create three-dimensional relief plots using the same inputs. Like two-dimensional `ax.contour` plots, `ax.contour3D` requires all the input data to be in the form of two-dimensional regular grids, with the Z data evaluated at each point. Here we'll show a 3D contour diagram of a three-dimensional sinusoidal function:

```
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)

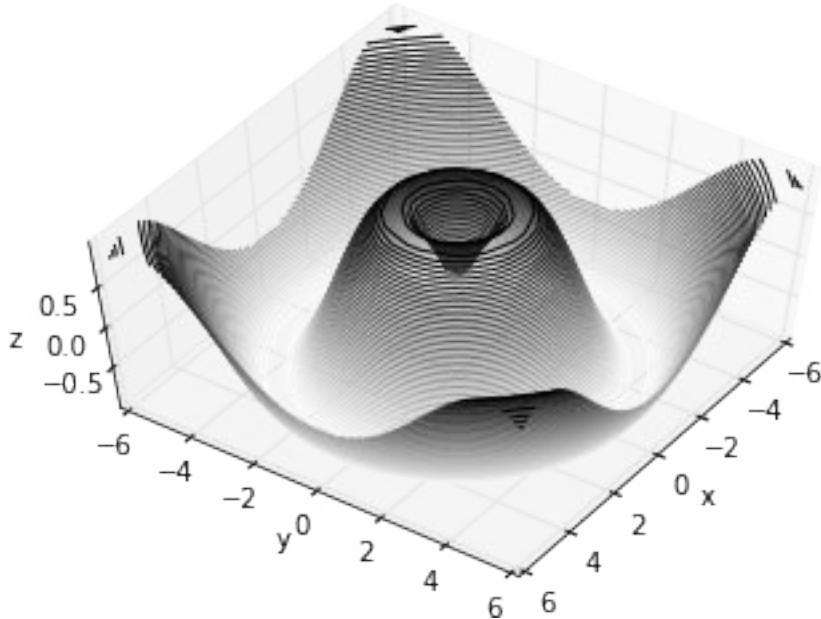
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



Sometimes the default viewing angle is not optimal; in this case we can use the `view_init` method to set the elevation and azimuthal angles. Here we'll use an eleva-

tion 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):

```
ax.view_init(60, 35)
fig
```

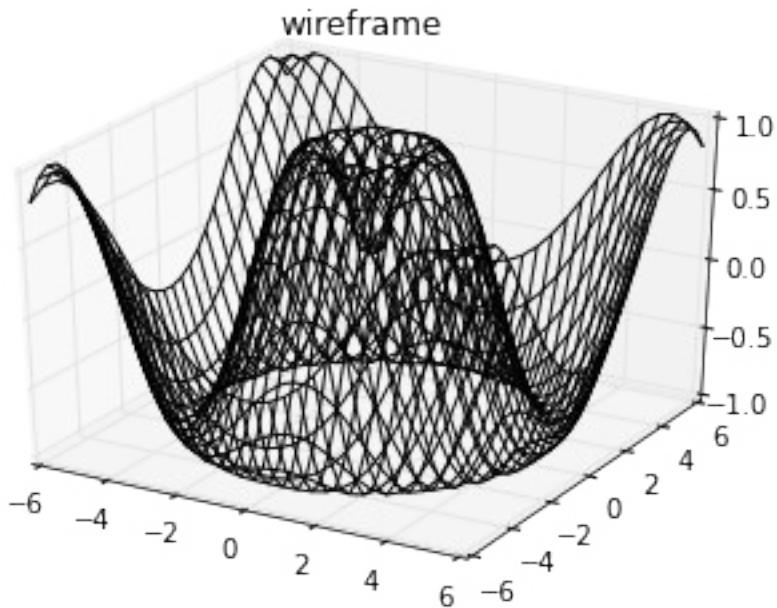


Again, note that this type of rotation can be accomplished interactively by clicking and dragging when using one of matplotlib's interactive backends.

Wireframes and Surface Plots

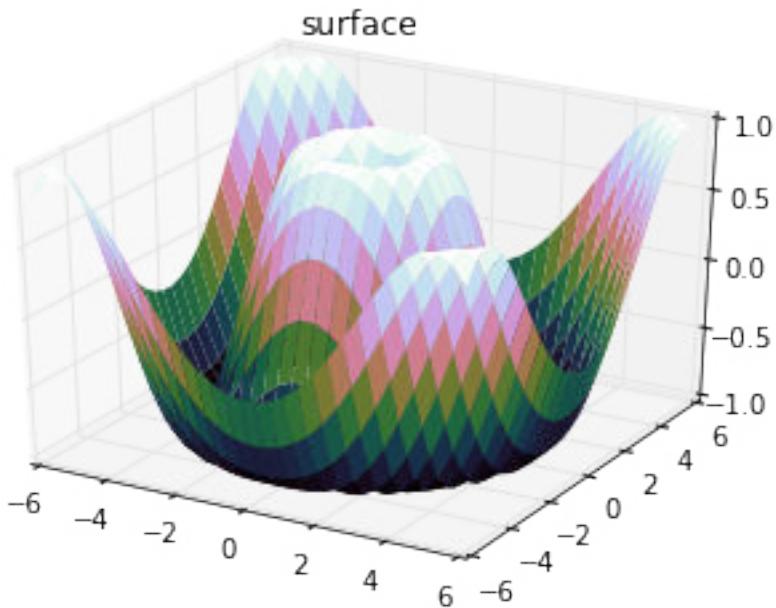
Two other types of 3D plots which work on gridded data are wireframes and surface plots. These take a grid of values and project it onto the specified three-dimensional surface, and can make the resulting three-dimensional forms quite easy to visualize:

```
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
```



A surface plot is like a wireframe plot, but each face of the wireframe is a filled polygon. Adding a colormap to the filled polygons can aid perception of the topology of the surface being visualized:

```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='cubehelix', edgecolor='none')
ax.set_title('surface');
```

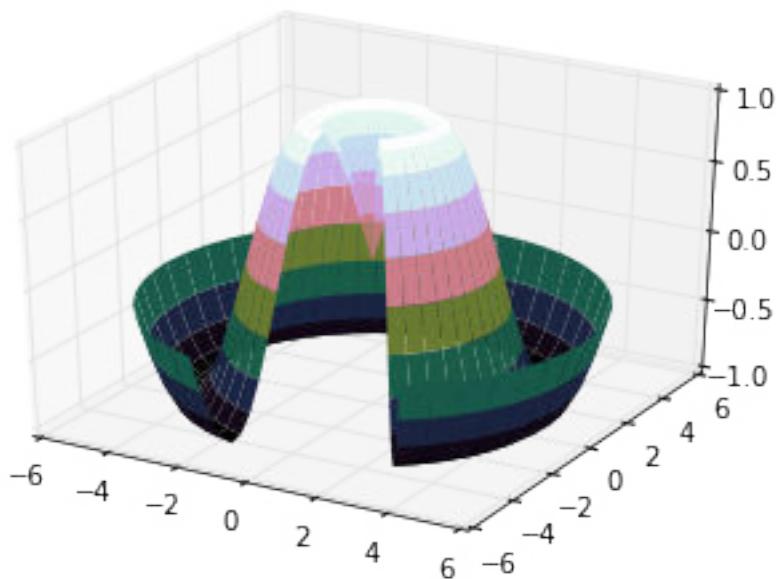


Note that though the grid of values for a surface plot needs to be two-dimensional, it need not be rectilinear. Here is an example of creating a partial polar grid, which when used with the `surface3D` plot can give us a slice into the function we're visualizing:

```
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='cubehelix', edgecolor='none');
```



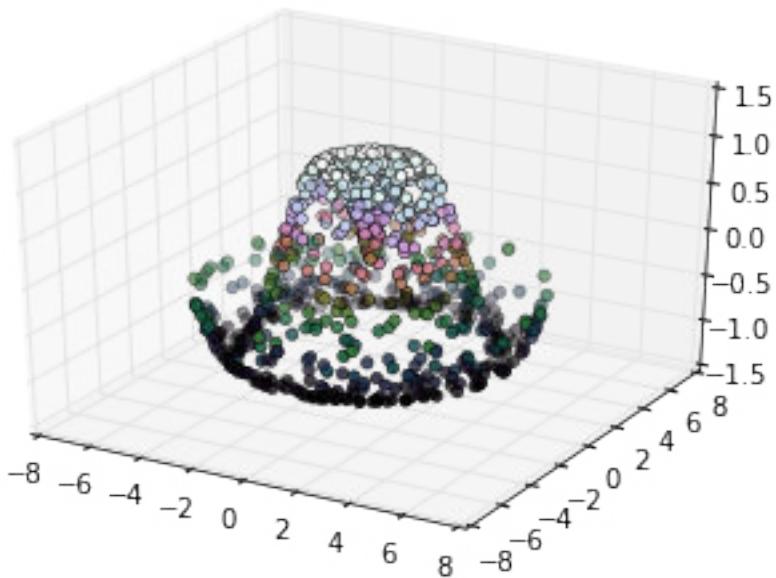
Surface Triangulations

For some applications, the rectilinear grid required by the above routines is 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 polar grid, we instead have a set of random draws?

```
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)
```

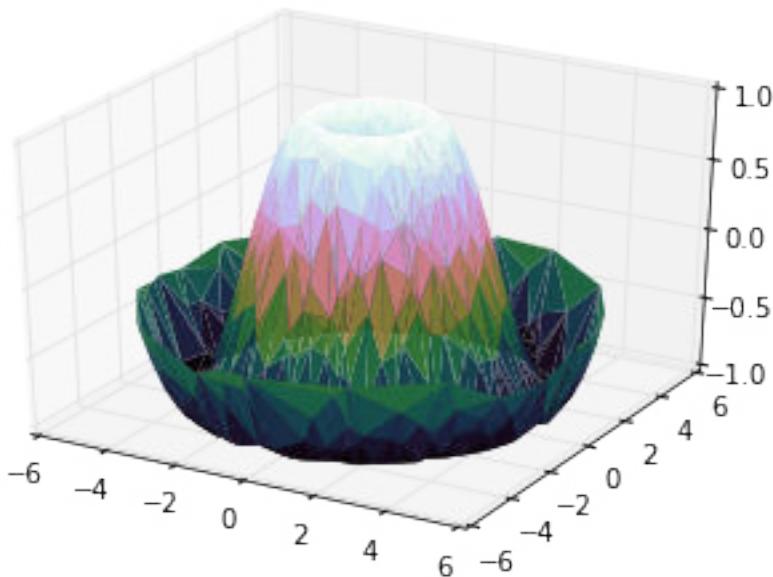
We could create a scatterplot of the points to get an idea of the surface we're sampling from:

```
ax = plt.axes(projection='3d')
ax.scatter(x, y, z, c=z, cmap='cubehelix', linewidth=0.5);
```



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. Remember that `x`, `y`, and `z` here are one-dimensional arrays.

```
ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z,
                 cmap='cubehelix', edgecolor='none');
```



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 3D plots. For example, it is actually possible to plot a 3D Möbius strip using this, as we'll see next.

Triangulation Example: Creating A Möbius Strip

The key to creating the mobius strip is to think about its parametrization: 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.

Let's create this parametrization:

```
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 parametrization, we must determine the (x, y, z) positions of the embedded strip.

The key to creating the mobius strip is to recognize 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 twisting of the strip about its axis (we'll call this ϕ). For a Möbius strip, we must have the strip makes half a twist during a full loop, or $\Delta\pi = \Delta\theta/2$.

```
phi = 0.5 * theta
```

Now we use simple geometry to derive the three-dimensional embedding. We'll define r , the distance of each point from the center, and use this to find the embedded (x, y, z) coordinates:

```
# radius in x-y plane
r = 1 + w * np.cos(phi)

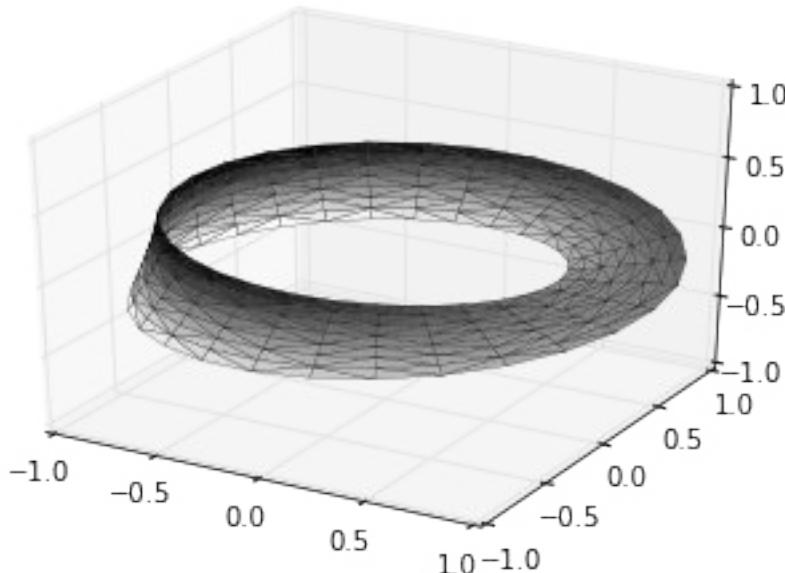
x = np.ravel(r * np.cos(theta))
y = np.ravel(r * np.sin(theta))
z = np.ravel(w * np.sin(phi))
```

Finally, to plot the object, we must make sure the triangulation is correct. The best way to do this is to define the triangularization *within the underlying parametrization*, and then let matplotlib project this triangulation into the 3-dimensional space of the Möbius strip. This can be accomplished as follows:

```
# triangulate in the underlying parametrization
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='binary', linewidths=0.2);

ax.set_xlim(-1, 1); ax.set_ylim(-1, 1); ax.set_zlim(-1, 1);
```



Combining the above techniques, it is possible to create and display a wide variety of 3D objects and patterns in matplotlib.

Geographic Data with Basemap

One common type of visualization in Data Science contexts is that of geographic data. Matplotlib's main tool for this type of visualization is the `basemap` toolkit, which is one of several matplotlib toolkits which lives under the `mpl_toolkits` namespace. In this section, we'll show several examples of the type of map visualization which is possible with this toolkit; there is much more that can be done than is listed here. Hopefully this can be a jumping-off point for you to learn more using Basemap's documentation and other online resources.

Installation of Basemap is straightforward; if you're using conda you can type

```
$ conda install basemap
```

Otherwise, you can use

```
$ pip install basemap
```

and the package will be downloaded.

```
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap
import numpy as np
```

Once you have the BaseMap toolkit installed and imported, you can plot geographic data in just a few lines:

```
plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
            lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```



The meaning of the arguments to `Basemap` will be discussed below.

The useful thing is that this isn't just an image; it is a `matplotlib` axes which understands spherical coordinates and which allows us to easily over-plot data on the map! For example, we can use a different map projection, zoom-in to North America and plot the location of Seattle. We'll use an `etopo` image (which shows topographical features both on land and under the ocean) as the map background:

```
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);
```



This gives you just a taste of the sort of geographic visualizations that are possible with just a few lines of Python. Below we'll discuss the features of basemap in more depth, and provide several examples of visualizing map data. Using these brief examples as building blocks, you should be able to create nearly any map visualization that you desire.

Map Projections

The first thing to think about when using maps is what projection to use. You're probably familiar with the fact that it is impossible to project a spherical map, such as

that 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 that are useful to maintain through this projection, whether its direction, area, distance, shape, or other considerations.

The Basemap package implements a large number of projections, all referenced by a short format code. These are listed in the following table:

Format Code	Description	Format Code	Description
gnom	Gnomonic	aeqd	Azimuthal Equidistant
nsper	Near-Sided Perspective	splaea	South-Polar Lambert Azimuthal
moll	Mollweide	cea	Cylindrical Equal Area
mbtfpq	McBryde-Thomas Flat-Polar Quartic	eqdc	Equidistant Conic
kav7	Kavrayskiy VII	poly	Polyconic
aea	Albers Equal Area	rotpole	Rotated Pole
cyl	Cylindrical Equidistant	hammer	Hammer
eck4	Eckert IV	omerc	Oblique Mercator
tmerc	Transverse Mercator	laea	Lambert Azimuthal Equal Area
gall	Gall Stereographic Cylindrical	lcc	Lambert Conformal
nplaea	North-Polar Lambert Azimuthal	npaeqd	North-Polar Azimuthal Equidistant
sinu	Sinusoidal	vandg	van der Grinten
mill	Miller Cylindrical	merc	Mercator
npstere	North-Polar Stereographic	spstere	South-Polar Stereographic
geos	Geostationary	cass	Cassini-Soldner
stere	Stereographic	robin	Robinson
spaeqd	South-Polar Azimuthal Equidistant	ortho	Orthographic

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

```
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 distortions near the poles. The spacing of latitude lines varies between different cylindrical projections, leading to different conservation properties, and different distortion near the poles. Below we show an example of the *Equidistant Cylindrical Projection*, which chooses a latitude scaling which preserves distances along meridians. Other cylindrical projections are the Mercator (`projection='merc'`) and the Cylindrical Equal Area (`projection='cea'`) projections.

```

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)

```

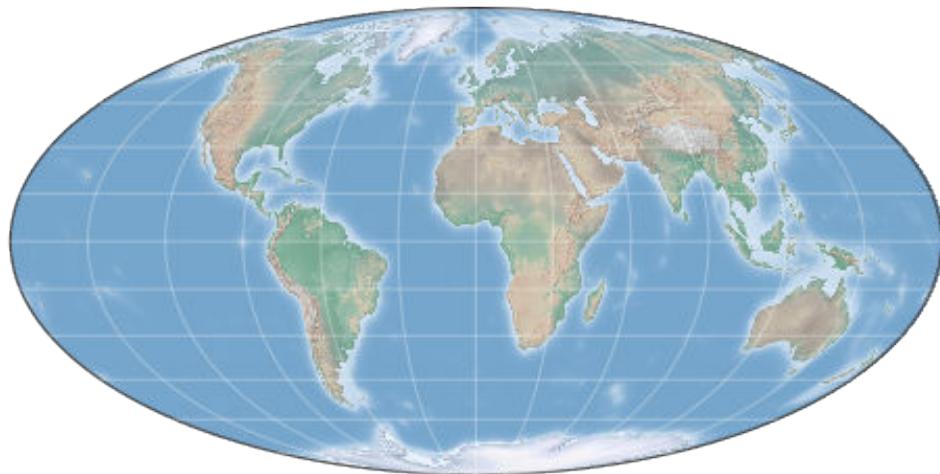


The additional arguments to Basemap for this view specify the latitude (`lat`) and longitude (`lon`) of the lower-left corner (`llcrnrl`) and upper-right corner (`urcrnrl`) for the desired map, in units of degrees.

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 projection. The Mollweide projection (`projection='moll'`) is one common example of this, in which all meridians are elliptical arcs. It is constructed so as to preserve 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.

```
fig = plt.figure(figsize=(8, 6), edgecolor='w')
m = Basemap(projection='moll', resolution=None,
            lat_0=0, lon_0=0)
draw_map(m)
```



The extra arguments to BaseMap here refer to the central latitude (`lat_0`) and longitude (`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 example is the Orthographic projection (`projection='ortho'`) which shows one side of the globe as seen from a viewer at a very long distance. As such, it can show only half the globe at a time. Other perspective-based projections include the Gnomonic (`projection='gnom'`) and Stereographic (`projection='stere'`) projections. These are often the most useful for showing small portions of the map.

Here is an example of the Orthographic Projection:

```
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None,
            lat_0=50, lon_0=0)
draw_map(m);
```



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 the cone may become very distorted. One example of this is the Lambert Conformal Conic projection (`projection='lcc'`), which we saw above 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 projections are the Equidistant Conic (`projection='eqdc'`) and the Albers Equal-Area (`projection='aea'`) projection. Conic projections, like perspective projections, tend to be good choices for representing small to medium patches of the globe.

```
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)
```



Other Projections

If you're going to do much with map-based visualizations, I encourage you to read-up on other available projections, along with their properties, advantages, and disadvantages. If you dig deep enough, you'll find an incredible subculture of geo-viz geeks who will be ready to argue fervently in support of their favorite projection for any given application!

Drawing a Map Background

Above we saw examples of the `bluemarble()` and `shadedrelief()` methods for projecting global images on the map, as well as the `drawparallels()` and `drawmeridians()` methods for drawing lines of constant latitude and longitude. The basemap package contains a range 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 & counties. The following are some of the available drawing functions that you may wish to explore using IPython's help features:

Physical boundaries; Bodies of Water

- `drawcoastlines()`: draw continental coast lines
- `drawlsmask()`: draw a mask between the land and sea, for use with projecting images on one 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

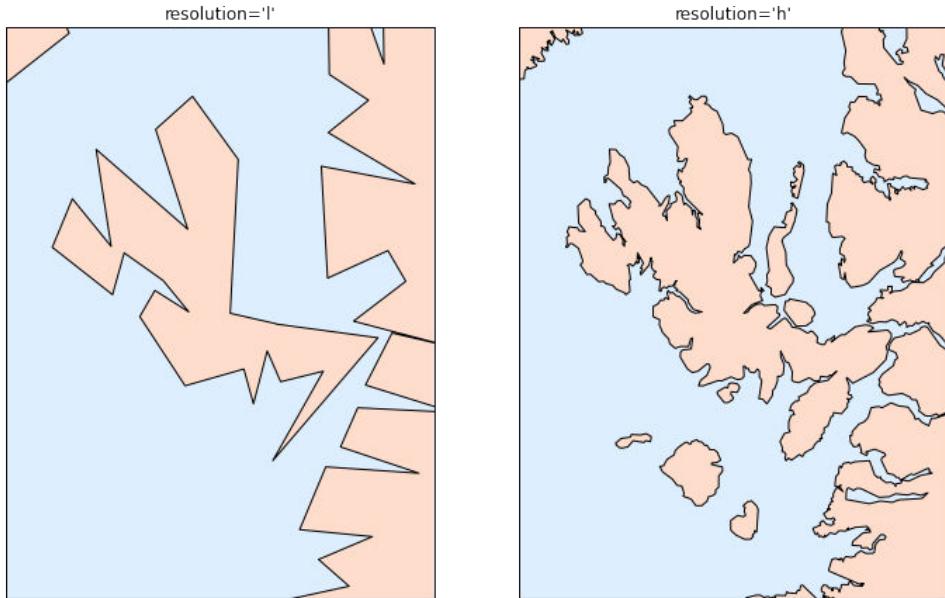
- `bluemarble()`: project NASA's blue marble image onto the map
- `shadedrelief()`: project a shaded relief image onto the map
- `etopo()`: draw an etopo relief image onto the map
- `warpimage()`: project a user-provided image onto the map

For 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 x 120,000 kilometers shows it well:

```
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));
```



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's important to set the resolution parameter correctly for your given map view.

Plotting Data On Maps

Perhaps the most useful piece of the Basemap toolkit is the ability to over-plot a variety 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 longitude

coordinates to (x, y) coordinates for plotting with plt, as we saw in the Seattle example above.

In addition to this, there are many map-specific functionality available as methods of the Basemap instance. These work very similarly to their standard matplotlib counterparts, 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 some examples of a few of these below. For more information on these functions, including several example plots, see the online Basemap documentation at <http://matplotlib.org/basemap/>.

Example: California Cities

Recall that in Section X.X, we demonstrated the use of size and color in a scatterplot to convey information about the location, size, and population of California cities. Here, we'll create this plot again, but using Basemap to put the data in context.

We start with loading the data, as we did in Section X.X:

```
import pandas as pd
cities = pd.read_csv('california_cities.csv')

# Extract the data we're interested in
lat = cities['latd'].values
lon = cities['longd'].values
population = cities['population_total'].values
area = cities['area_total_km2'].values
```

Next we set up the map projection, scatter the data, and then create a colorbar and legend as we did in section X.X:

```
# 1. Draw the map background
fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='lcc', resolution='h',
            lat_0=37.5, lon_0=-119,
```

```

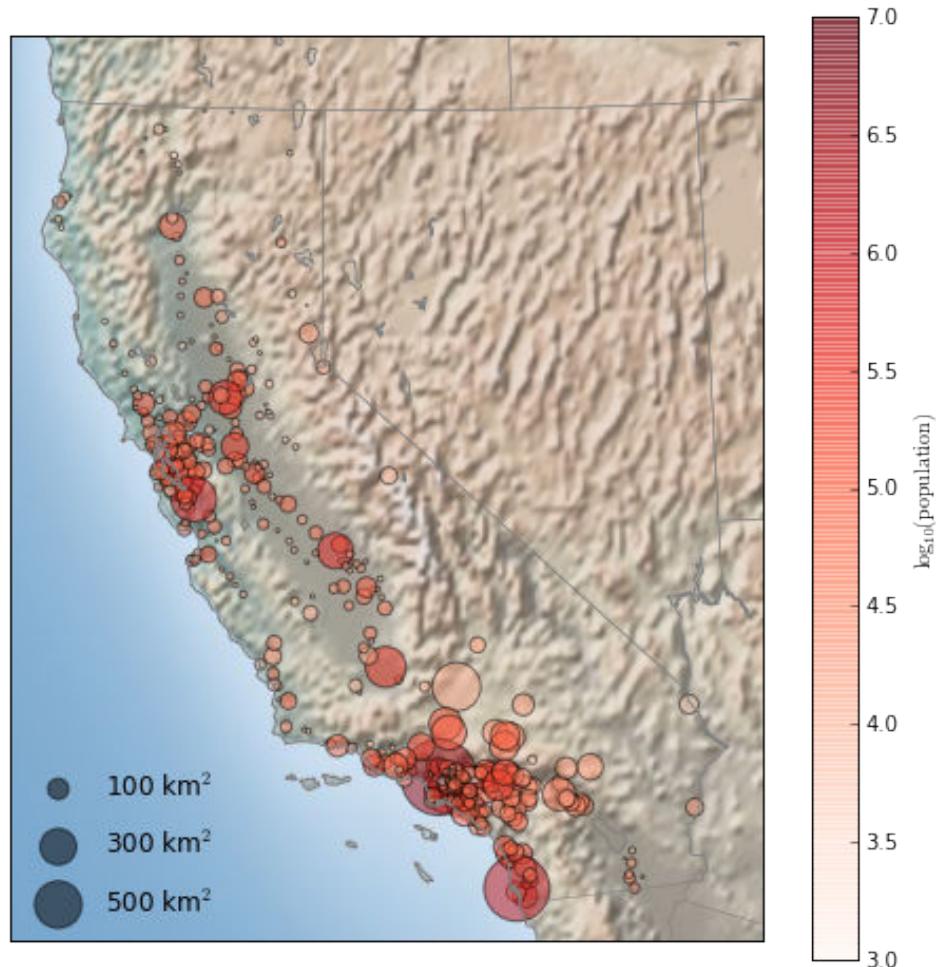
    width=1E6, height=1.2E6)
m.shadedrelief()
m.drawcoastlines(color='gray')
m.drawcountries(color='gray')
m.drawstates(color='gray')

# 2. scatter city data,
# with color reflecting population
# and with size reflecting area
m.scatter(lon, lat, latlon=True,
          c=np.log10(population), s=area,
          cmap='Reds', alpha=0.5)

# 3. create legend for size, colorbar for color
plt.colorbar(label=r'$\log_{10}(\{\text{population}\})$')
plt.clim(3, 7)

# plot some empty lists for the legend (see section X.X)
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');

```



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, strung like pearls along the highways in the flat central valley, and avoiding almost completely the mountainous regions along the borders of the state.

Example: Surface Temperature Data

As an example of visualizing some more continuous geographic data, let's take a look at the "Polar Vortex" which hit the eastern half of the United States in January of 2014. A great source for any sort of climatic data is NASA's Goddard Institute for Space Studies, at <http://data.giss.nasa.gov/>. Here we'll use the GIS 250 temperature data, which we can download using shell commands (these commands may have to

be modified on Windows machines). The data used here was downloaded on 3/12/2015, and the file size is approximately 9MB:

```
# !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 using

```
$ conda install netcdf4
```

or if you're not using conda,

```
$ pip install netcdf4
```

We read the data as follows:

```
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:

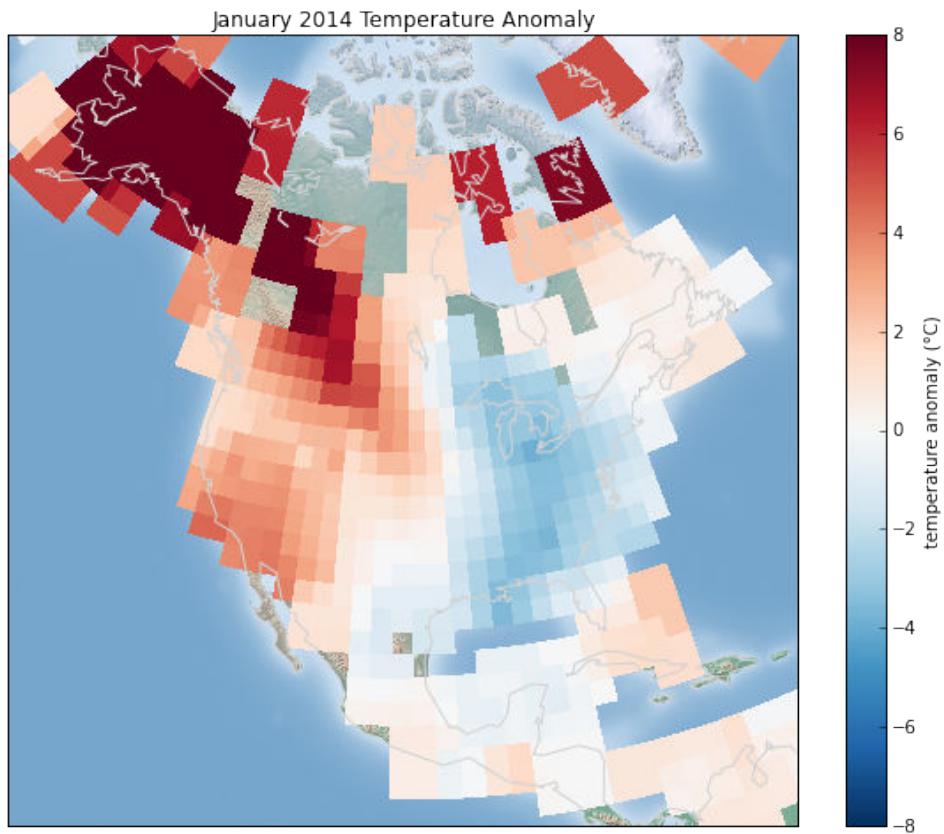
```
from netCDF4 import date2index  
from datetime import datetime  
timeindex = date2index(datetime(2014, 1, 15),  
                      data.variables['time'])
```

Now we can load the latitude and longitude data, as well as the temperature anomaly for this index:

```
lat = data.variables['lat'][:]  
lon = data.variables['lon'][:]  
lon, lat = np.meshgrid(lon, lat)  
temp_anomaly = data.variables['temp_anomaly'][timeindex]
```

Finally, we'll use the pcolormesh() method to draw a color mesh of the data. We'll look at North America, and use a shaded relief map in the background. Note that for this data we specifically choose a divergent colormap, which has a neutral color at zero and two contrasting colors at negative and positive values. We'll also lightly draw the coastlines over the colors for reference.

```
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.pcolorshaded(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 which happened during that month. The eastern half of the US was much colder than normal, while the western half and Alaska were much warmer. Regions with no recorded temperature show the map background.

Visualization With Seaborn

Matplotlib is a useful tool, but it 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 is relatively low-level. Doing sophisticated statistical visualization is possible, but often requires a *lot* of boilerplate code.
- Matplotlib predicated 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 these series' together into the right format. It would be nicer to have a plotting library that can intelligently use the DataFrame labels in a plot.

The answer to these problems is [Seaborn](#). Seaborn provides an API on top of matplotlib that offers sane choices for plot style & color defaults, defines simple high-level functions for common statistical plot types, and that integrates with the functionality provided by Pandas dataframes.

To be fair, the Matplotlib team is addressing this: it has recently added the `plt.style` tools covered in section X.X, and in the 2.0 release of the library will include a new default style sheet that will improve on the current status quo. But for all the reasons covered above, Seaborn remains an extremely useful addon.

Seaborn vs. Matplotlib

Here is an example of a simple random-walk plot in matplotlib, using its default plot formatting and colors. We start with the typical imports:

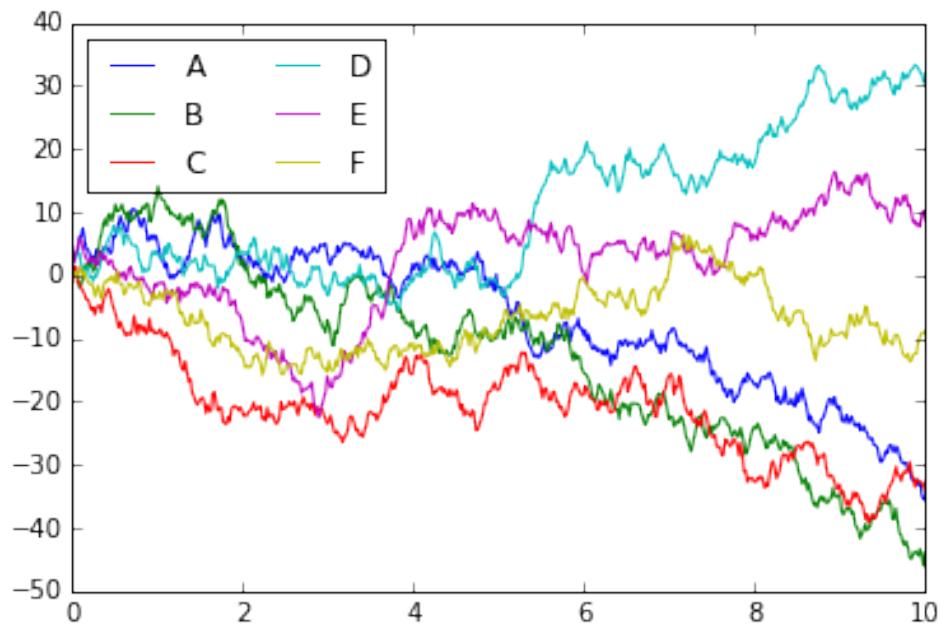
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Now we create some random walk data:

```
# 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:

```
# Plot the data with matplotlib defaults
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```



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 the context of 21st-century data visualization.

Now let's see how it works with Seaborn. Though seaborn has many of its own high-level plotting routines that we'll explore below, one component of it is that it can overwrites matplotlib's default parameters and cause even simple matplotlib scripts to produce vastly superior output. We can set the style by calling Seaborn's `set()` method. By convention, `seaborn` is imported as `sns`:

```
import seaborn as sns  
sns.set()
```

Now let's rerun the same two lines as above:

```
# same plotting code as above!  
plt.plot(x, y)  
plt.legend('ABCDEF', ncol=2, loc='upper left');
```



Ah, much better!

Exploring Seaborn Plots

The main idea of Seaborn is that it can create complicated plot types from Pandas data with relatively simple commands.

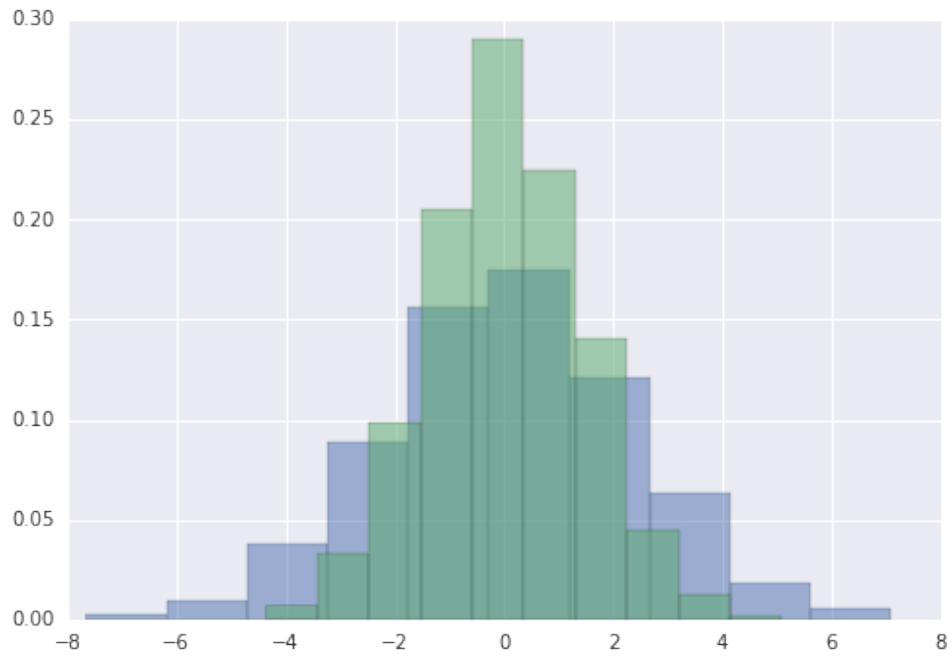
Let's take a look at a few of the datasets and plot types available in Seaborn. Note that all of the following *could* be done using raw matplotlib commands (this is, in fact, what Seaborn does under the hood) but the seaborn API is much more convenient.

Histograms, KDE, and Densities

Often in statistical data visualization, all you want is to plot histograms and joint distributions of variables. Seaborn provides simple tools to make this happen:

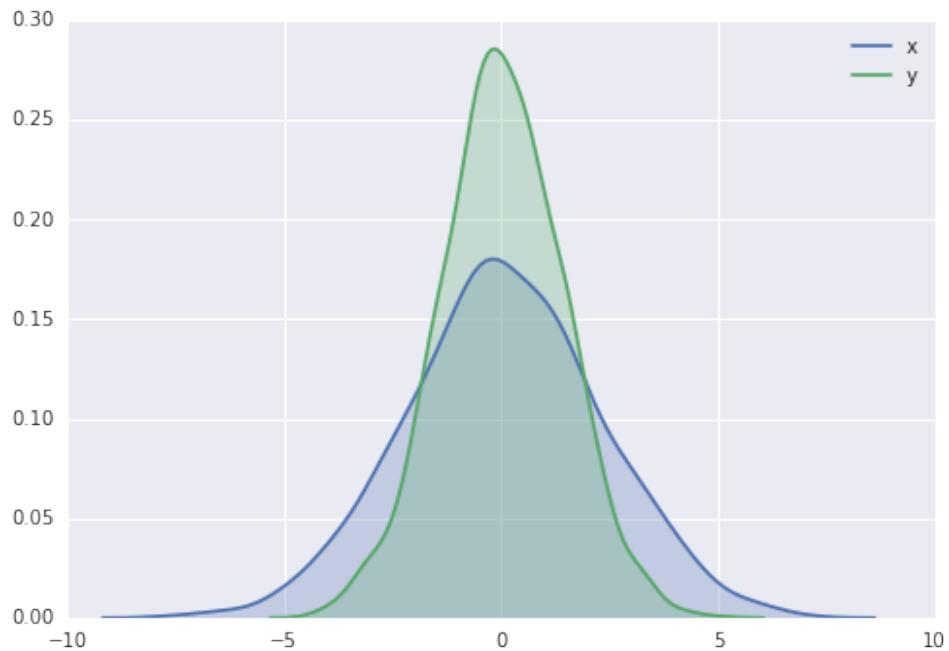
```
data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])

for col in 'xy':
    plt.hist(data[col], normed=True, alpha=0.5)
```



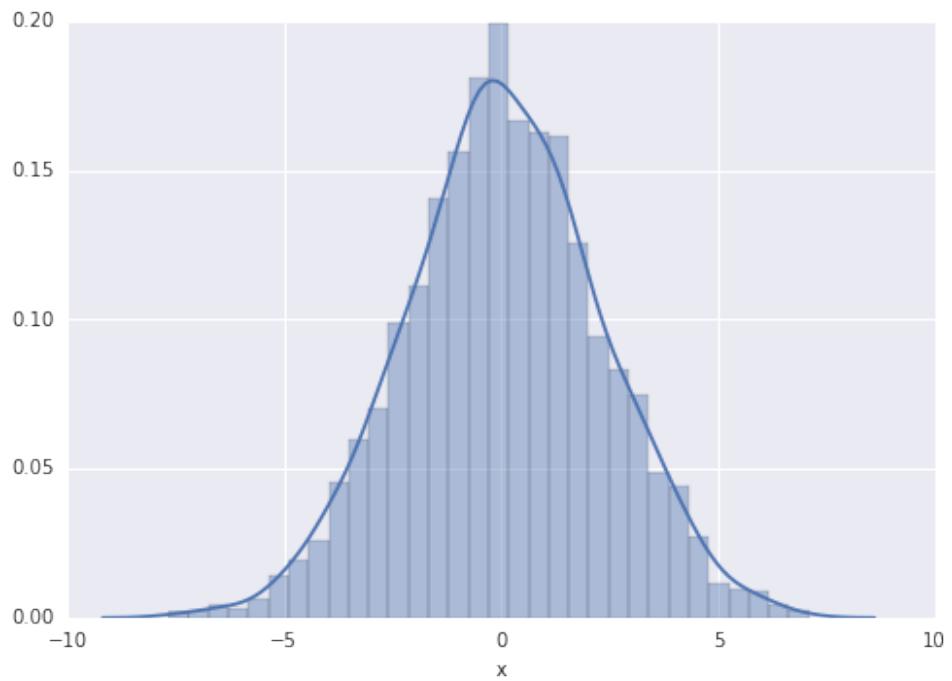
Rather than a histogram, we can get a smooth estimate of the distribution using a kernel density estimation:

```
for col in 'xy':  
    sns.kdeplot(data[col], shade=True)
```



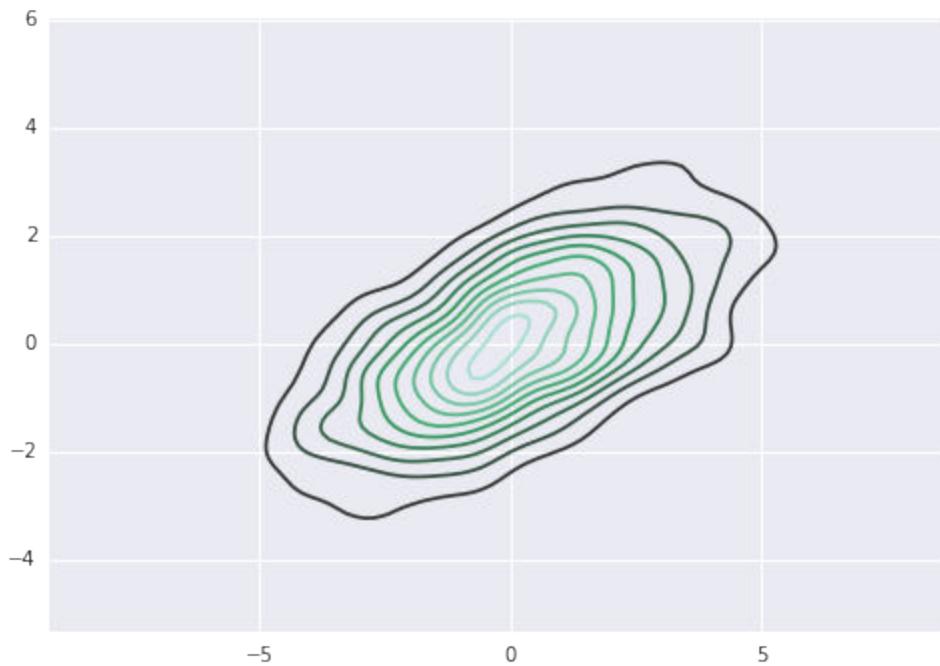
Histograms and KDE can be combined using `distplot`:

```
sns.distplot(data['x']);
```



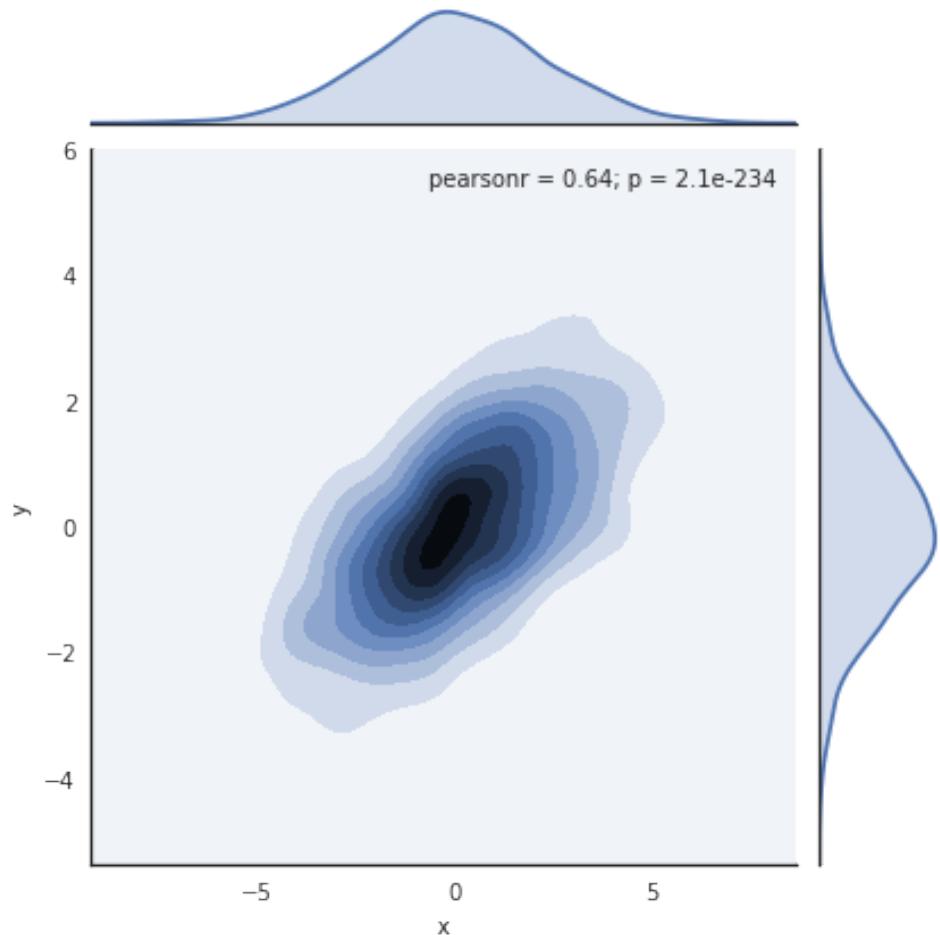
If we pass the full two-dimensional dataset to `kdeplot`, we will get a two-dimensional visualization of the data:

```
sns.kdeplot(data);
```



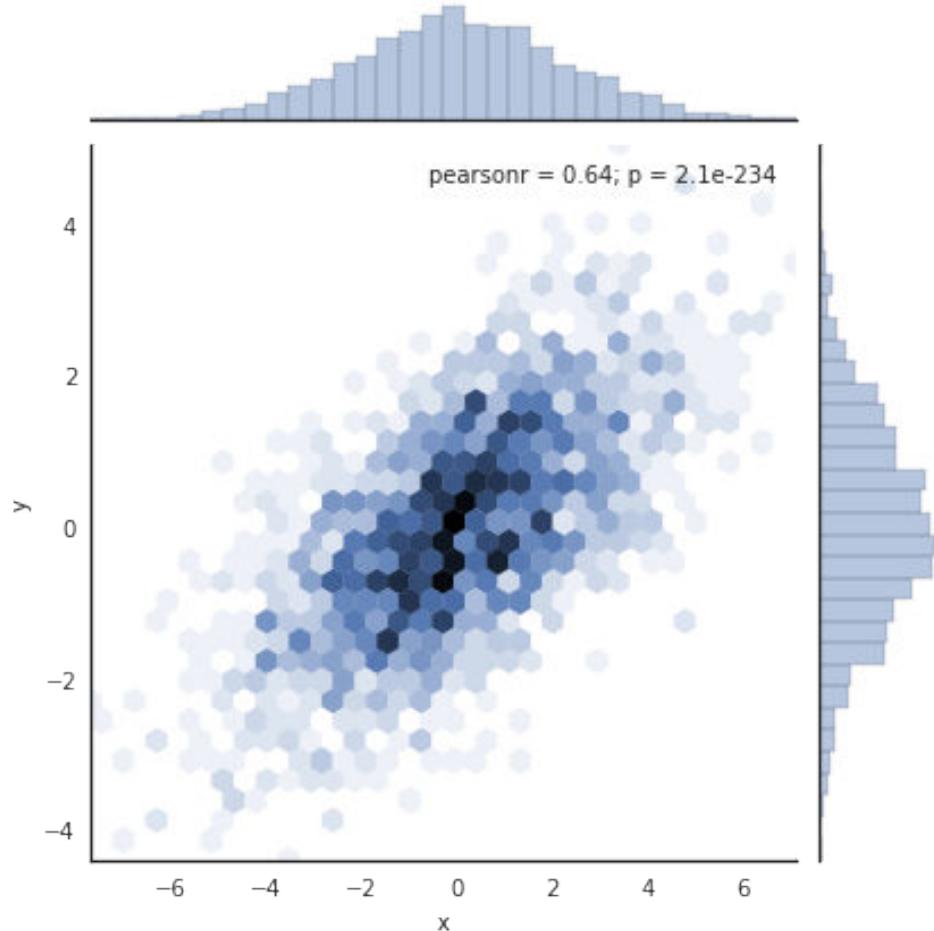
We can see the joint distribution and the marginal distributions together using `sns.jointplot`. For this plot, we'll set the style to a white background:

```
with sns.axes_style('white'):
    sns.jointplot("x", "y", data, kind='kde');
```



There are other parameters which can be passed to `jointplot`: for example, we can use a hexagonally-based histogram instead:

```
with sns.axes_style('white'):
    sns.jointplot("x", "y", data, kind='hex')
```



Pairplots

When you generalize joint plots to data sets of larger dimensions, you end up with *pair plots*. This is very useful for exploring correlations between multi-dimensional data, when you'd like to plot all pairs of values against each other.

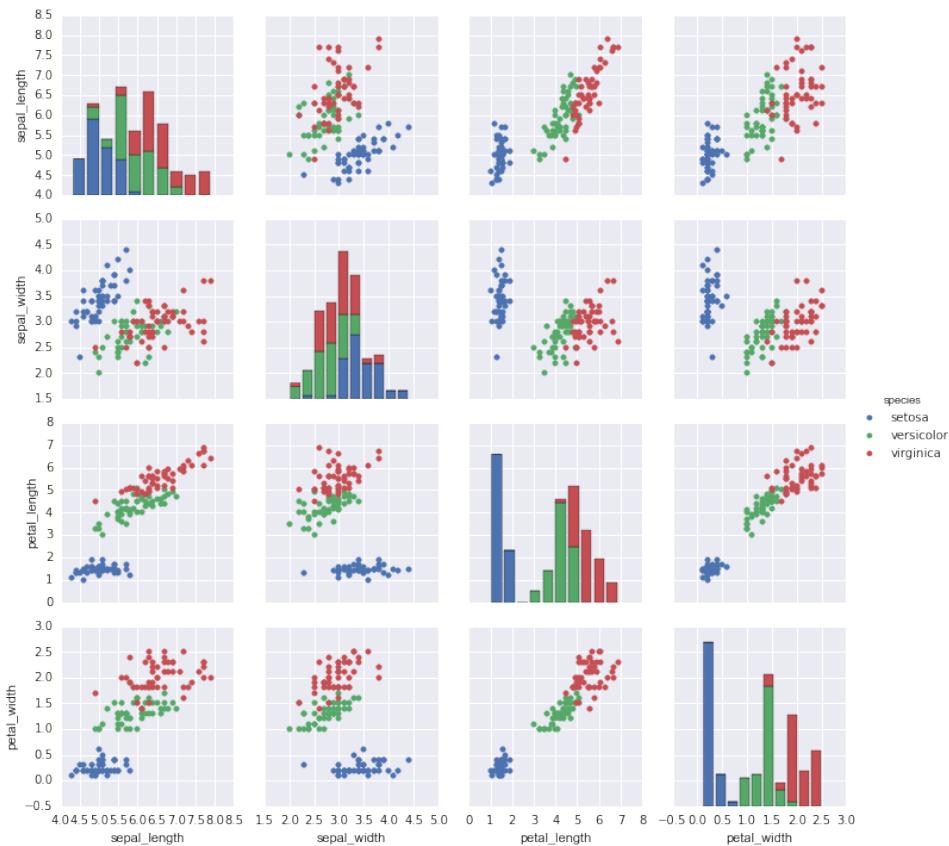
We'll demo this with the well-known *iris* dataset, which lists measurements of petals and sepals of three iris species:

```
iris = sns.load_dataset("iris")
iris.head()
```

	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 multi-dimensional relationships among the samples is as easy as calling `sns.pairplot`:

```
sns.pairplot(iris, hue='species', size=2.5);
```



Faceted Histograms

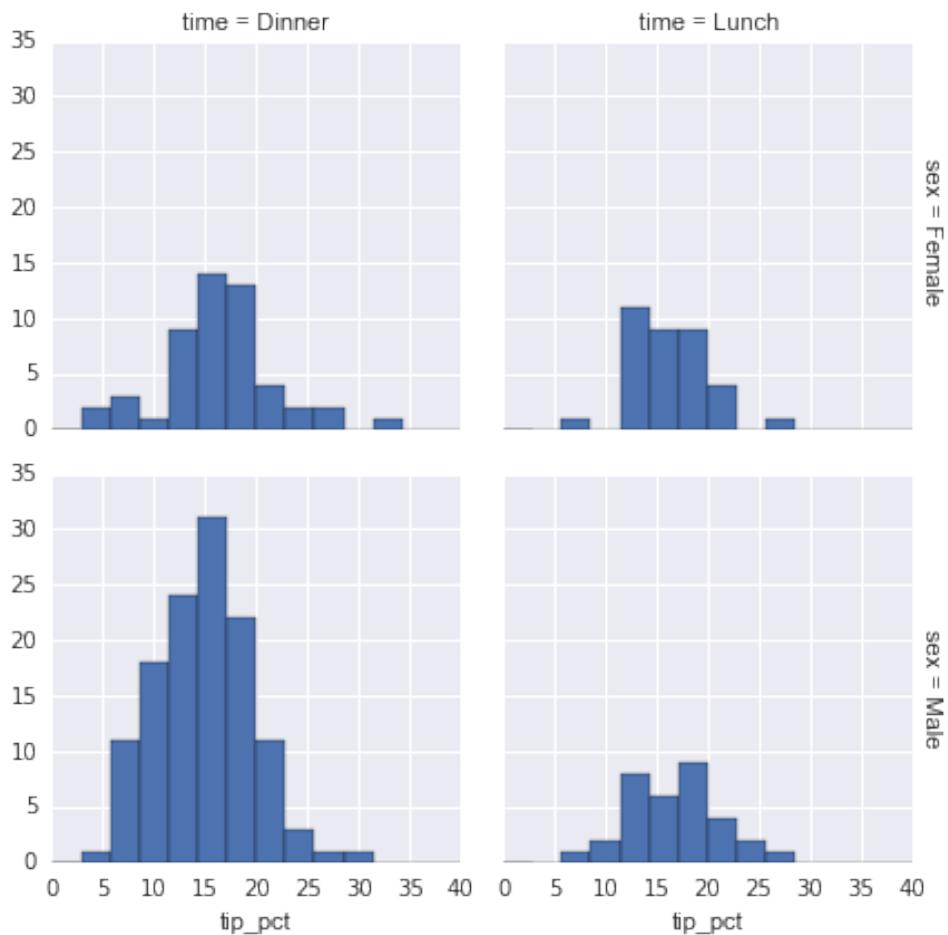
Sometimes the best way to view data is via histograms of subsets. Seaborn's FacetGrid makes this extremely simple. We'll take a look at some data which shows the amount that restaurant staff receive in tips based on various indicator data:

```
tips = sns.load_dataset('tips')
tips.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

```
tips['tip_pct'] = 100 * tips['tip'] / tips['total_bill']

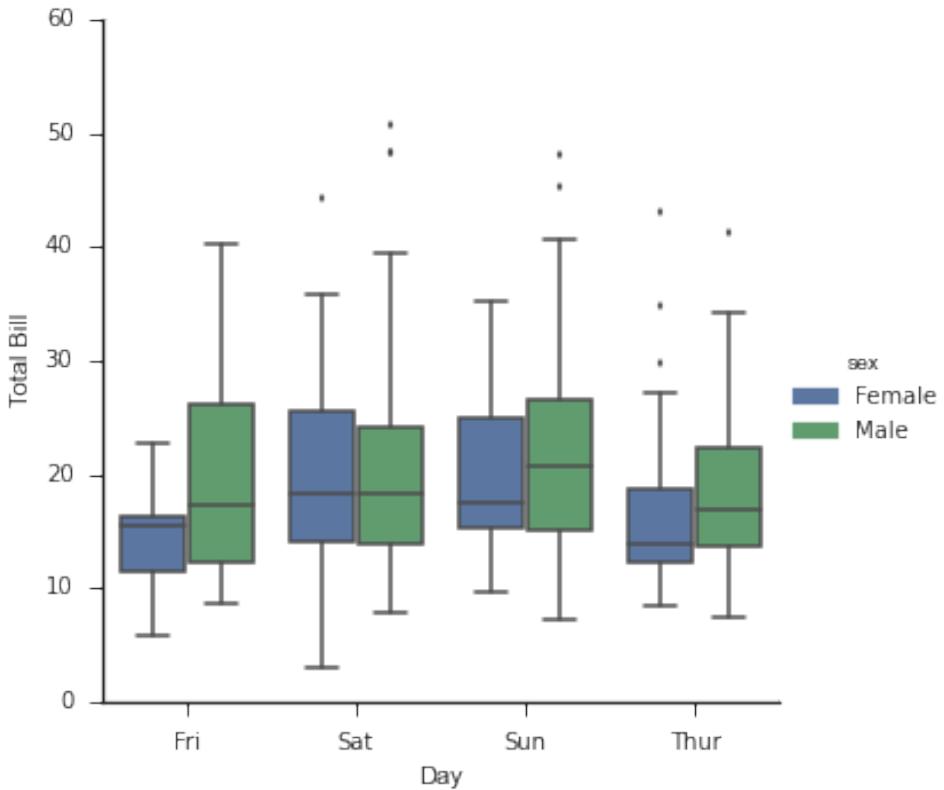
grid = sns.FacetGrid(tips, row="sex", col="time", margin_titles=True)
grid.map(plt.hist, "tip_pct", bins=np.linspace(0, 40, 15));
```



Factor Plots

Factor plots can be used to visualize this data as well. This allows you to view the distribution of a parameter within bins defined by any other parameter:

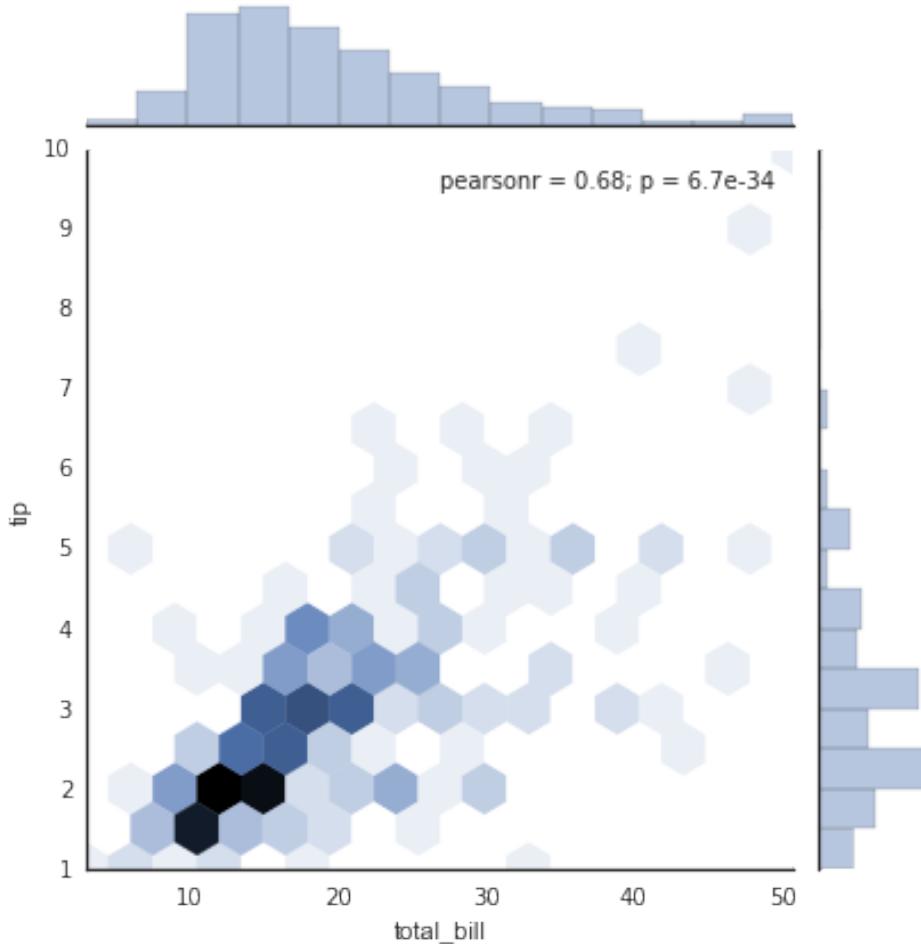
```
with sns.axes_style(style='ticks'):
    g = sns.factorplot("day", "total_bill", "sex", data=tips, kind="box")
    g.set_axis_labels("Day", "Total Bill");
```



Joint Distributions

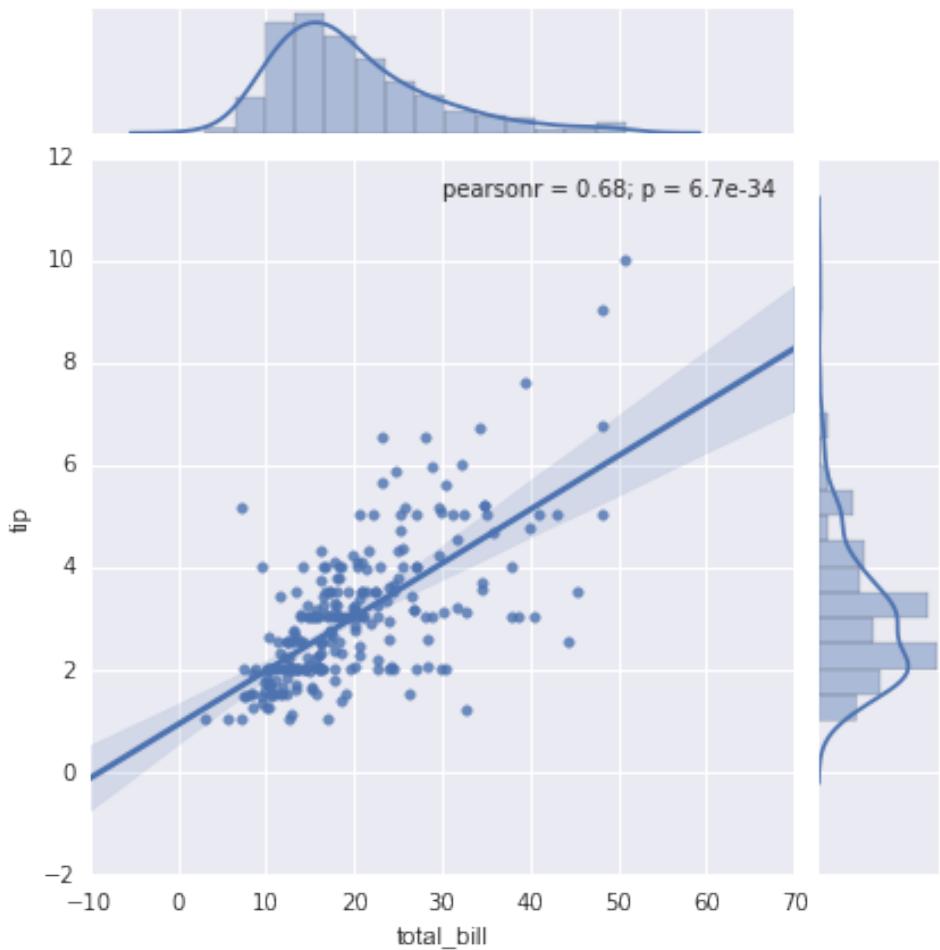
Similar to the pairplot we saw above, we can use `sns.jointplot` to show the joint distribution between different datasets, along with the associated marginal distributions:

```
with sns.axes_style('white'):
    sns.jointplot("total_bill", "tip", data=tips, kind='hex')
```



The joint plot can even do some automatic kernel density estimation and regression:

```
sns.jointplot("total_bill", "tip", data=tips, kind='reg');
```



Bar Plots

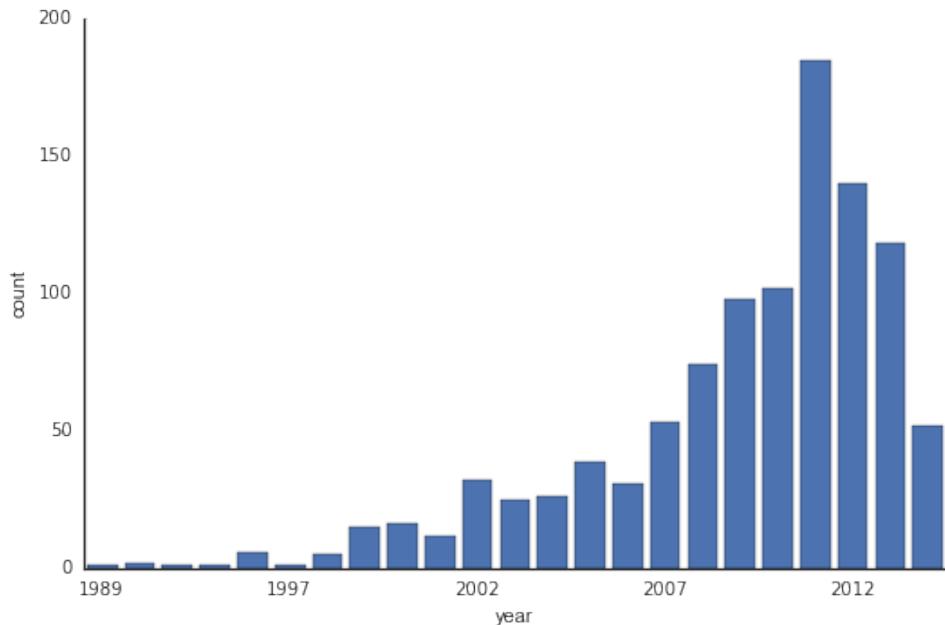
Time series can be plotted using `sns.factorplot`:

```
planets = sns.load_dataset('planets')
planets.head()
```

	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

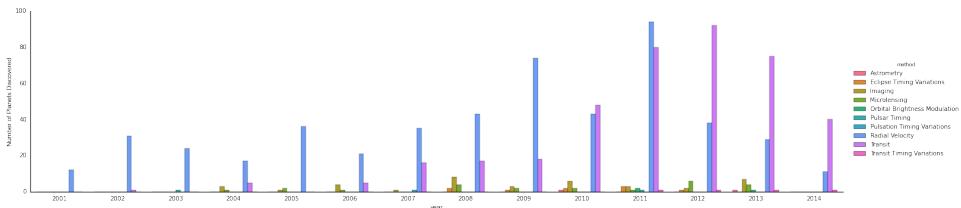
	method	number	orbital_period	mass	distance	year
4	Radial Velocity	1	516.220	10.50	119.47	2009

```
with sns.axes_style('white'):
    g = sns.factorplot("year", data=planets, aspect=1.5)
    g.set_xticklabels(step=5)
```



We can learn more by looking at the **method** of discovery of each of these planets:

```
with sns.axes_style('white'):
    g = sns.factorplot("year", data=planets, aspect=4.0,
                       hue='method', x_order=range(2001, 2015))
    g.set_ylabels('Number of Planets Discovered')
```



For more information on plotting with Seaborn, see the [seaborn documentation](#), the [seaborn gallery](#), and the official [seaborn tutorial](#).

Example: Exploring New York City Marathon Data

Here we'll look at using Seaborn to help visualize and understand the results of the 2008 New York City marathon. I've scraped the data from the web, removed any identifying information, and put it on github where it can be downloaded:

```
# !curl -O https://raw.githubusercontent.com/jakevdp/data-NYCmarathon/master/NYCM2008.csv  
  
nyc_data = pd.read_csv('NYCM2008.csv')  
nyc_data.head()
```

	age	gender	split	final
0	31	M	01:06:07	02:08:43
1	32	M	01:06:06	02:09:07
2	30	M	01:06:06	02:11:22
3	39	M	01:06:06	02:13:10
4	30	M	01:06:06	02:13:33

By default, Pandas loaded the time columns as Python strings (type `object`); we can see this by looking at the `dtypes` attribute of the DataFrame:

```
nyc_data.dtypes  
  
age        int64  
gender     object  
split      object  
final      object  
dtype: object
```

Let's fix this by providing a converter for the times:

```
def convert_time(s):  
    h, m, s = map(int, s.split(':'))  
    return pd.datetools.timedelta(hours=h, minutes=m, seconds=s)  
  
nyc_data = pd.read_csv('NYCM2008.csv',  
                      converters={'split':convert_time, 'final':convert_time})  
nyc_data.head()
```

	age	gender	split	final
0	31	M	01:06:07	02:08:43
1	32	M	01:06:06	02:09:07
2	30	M	01:06:06	02:11:22
3	39	M	01:06:06	02:13:10

	age	gender	split	final
4	30	M	01:06:06	02:13:33

```
nyc_data.dtypes  
  
age           int64  
gender        object  
split    timedelta64[ns]  
final    timedelta64[ns]  
dtype: object
```

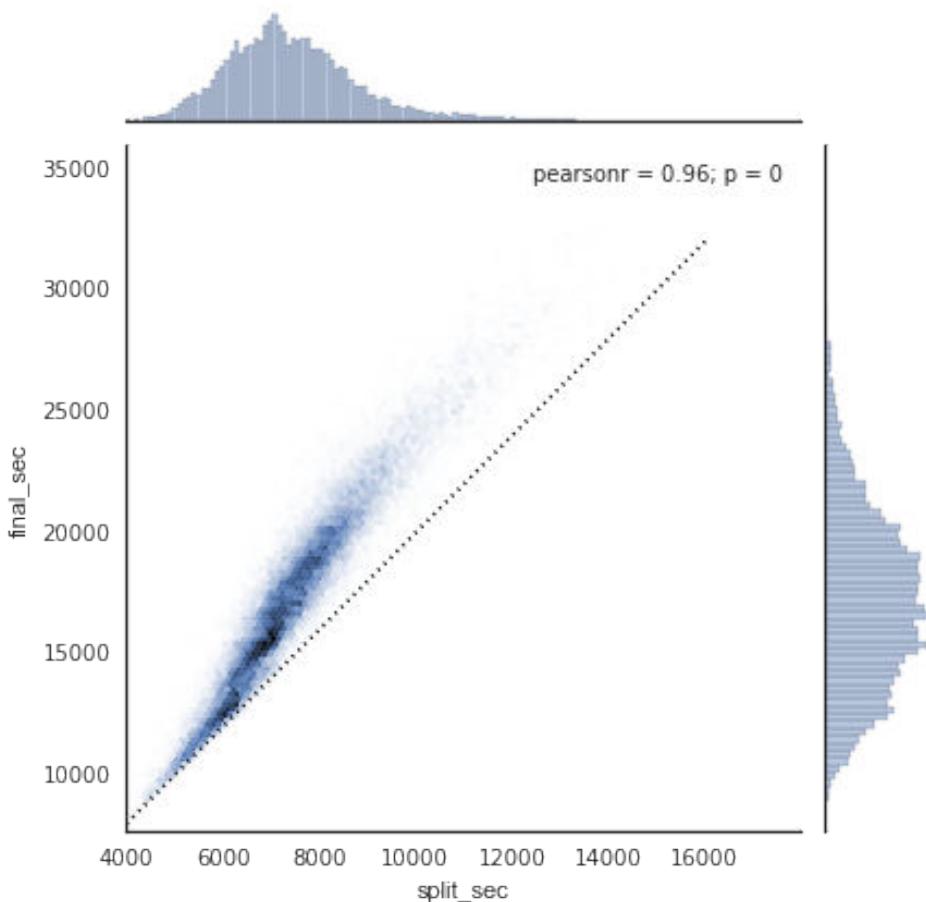
That looks much better. For the purpose of our seaborn plotting utilities, let's next add columns which give the times in seconds:

```
nyc_data['split_sec'] = nyc_data['split'].astype(int) / 1E9  
nyc_data['final_sec'] = nyc_data['final'].astype(int) / 1E9  
nyc_data.head()
```

	age	gender	split	final	split_sec	final_sec
0	31	M	01:06:07	02:08:43	3967	7723
1	32	M	01:06:06	02:09:07	3966	7747
2	30	M	01:06:06	02:11:22	3966	7882
3	39	M	01:06:06	02:13:10	3966	7990
4	30	M	01:06:06	02:13:33	3966	8013

To get an idea of what the data looks like, we can plot a `jointplot` over the data:

```
with sns.axes_style('white'):  
    g = sns.jointplot("split_sec", "final_sec", nyc_data, kind='hex')  
    g.ax_joint.plot(np.linspace(4000, 16000),  
                    np.linspace(8000, 32000), ':k')
```



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.

Let's create another column in the data, the split fraction, which tells whether someone did a negative split or positive split:

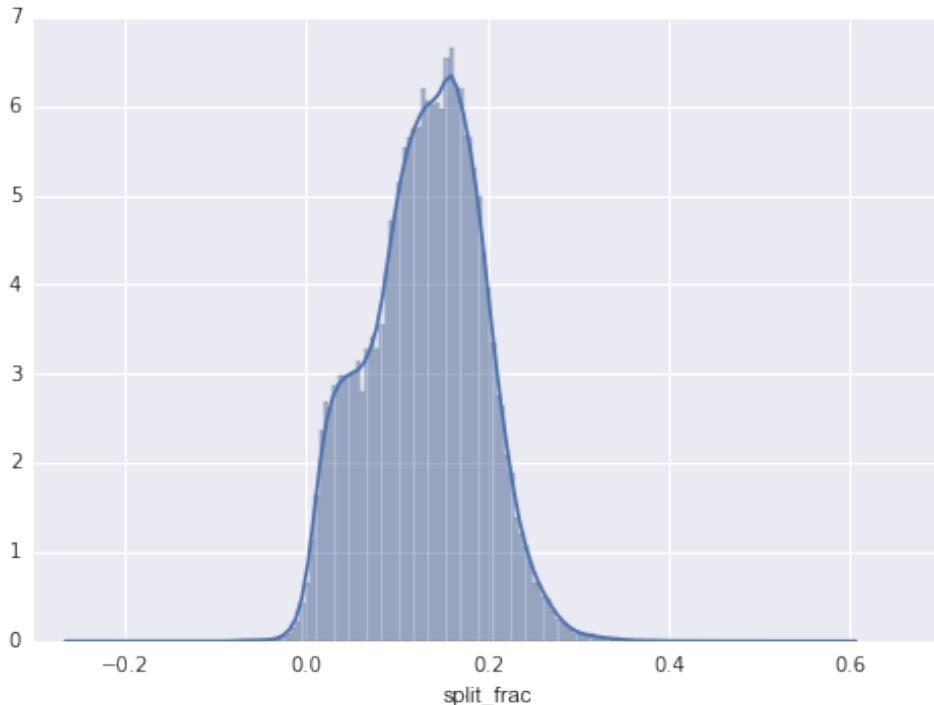
```
nyc_data['split_frac'] = 1 - 2 * nyc_data['split_sec'] / nyc_data['final_sec']
nyc_data.head()
```

	age	gender	split	final	split_sec	final_sec	split_frac
0	31	M	01:06:07	02:08:43	3967	7723	-0.027321
1	32	M	01:06:06	02:09:07	3966	7747	-0.023880
2	30	M	01:06:06	02:11:22	3966	7882	-0.006344

	age	gender	split	final	split_sec	final_sec	split_frac
3	39	M	01:06:06	02:13:10	3966	7990	0.007259
4	30	M	01:06:06	02:13:33	3966	8013	0.010109

Where this split difference is less than zero, the person negative-split the race by that fraction. Let's do a distribution plot of this split fraction:

```
sns.distplot(nyc_data['split_frac']);
```



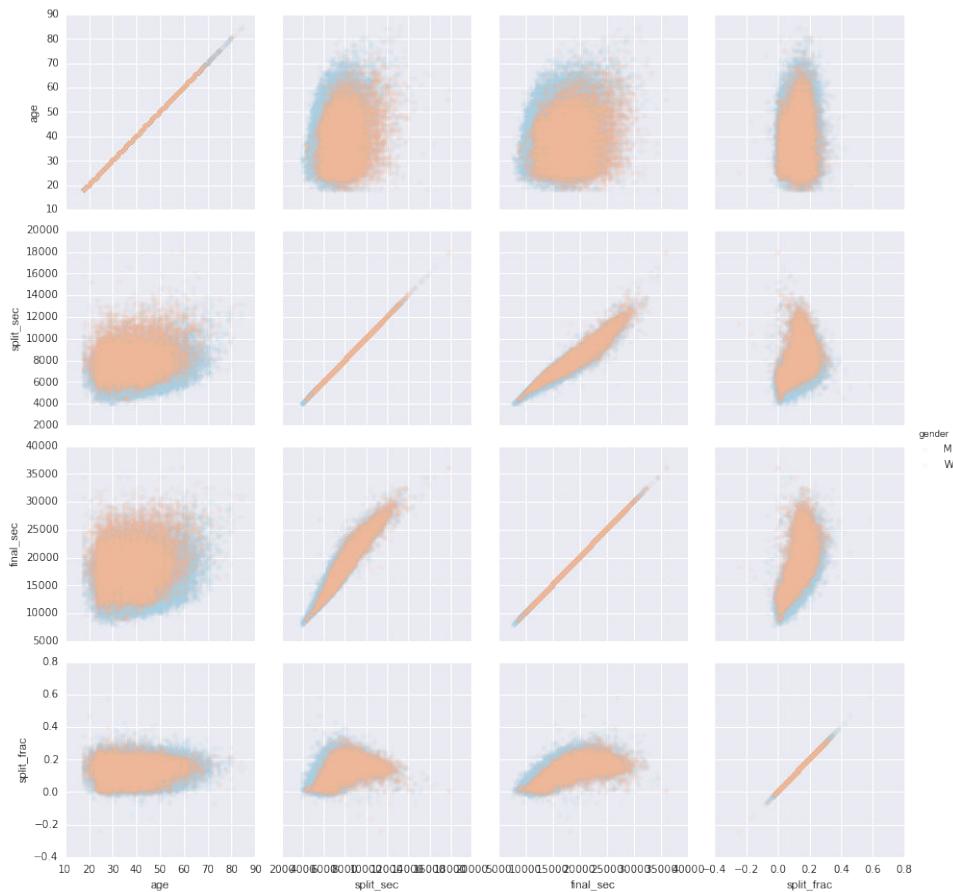
```
sum(nyc_data.split_frac < 0)
```

```
240
```

There were 240 people who negative-split their race.

Let's see whether there is any correlation between this split fraction and other variables. We'll do this using a `pairgrid`, which draws plots of all these correlations:

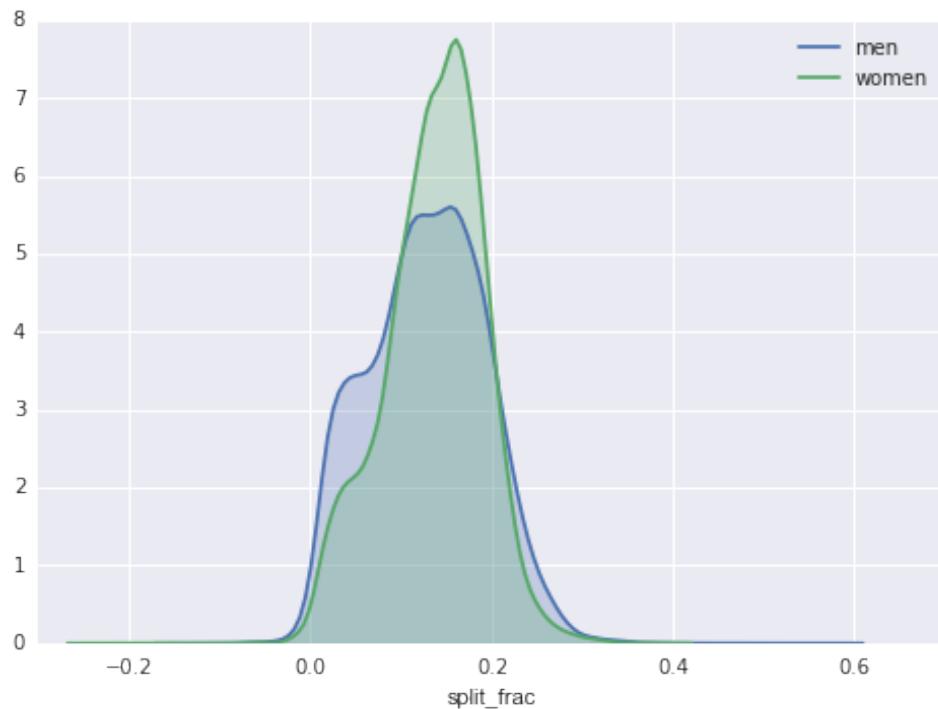
```
g = sns.PairGrid(nyc_data, vars=['age', 'split_sec', 'final_sec', 'split_frac'],
                  hue='gender', palette='RdBu_r')
g.map(plt.scatter, alpha=0.1)
g.add_legend();
```



It looks like the split fraction does not correlate particularly with age, but does correlate with the final time: faster runners tend to have closer to even splits on their marathon time.

The difference between men and women here is interesting. Let's look at the histogram of split fractions for these two groups:

```
sns.kdeplot(nyc_data.split_frac[nyc_data.gender=='M'], label='men', shade=True)
sns.kdeplot(nyc_data.split_frac[nyc_data.gender=='W'], label='women', shade=True)
plt.xlabel('split_frac');
```

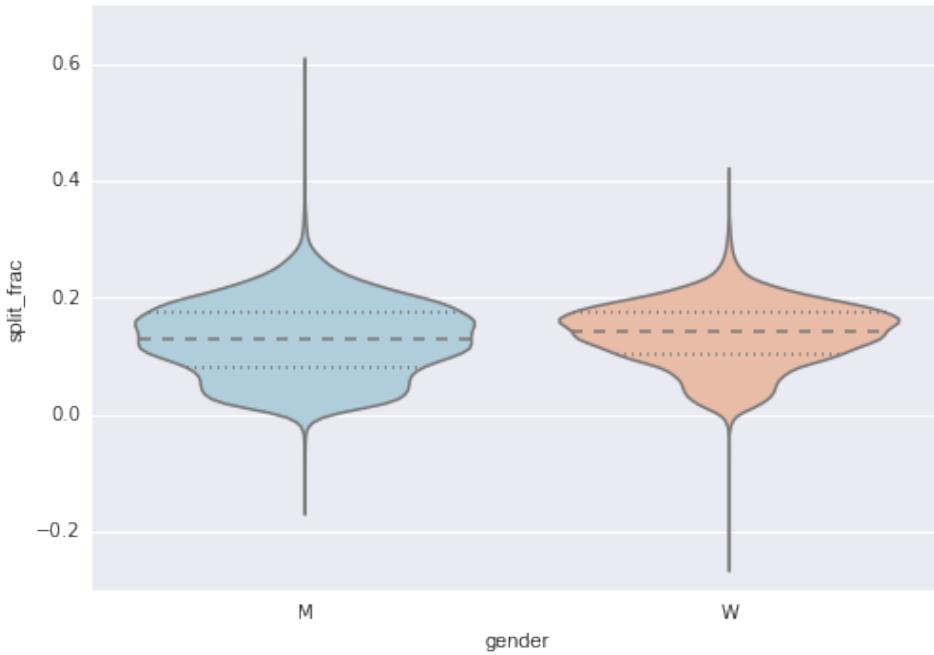


The interesting thing here is that there are many more men than women who are running close to an even split! This almost looks like some kind of bimodal distribution among the men and women. Let's see if we can suss-out what's going on by looking at the distributions as a function of age.

A nice way to compare distributions is to use a *Violin Plot*

```
def age_range(age_min, age_max):
    return (nyc_data.age >= age_min) & (nyc_data.age < age_max)

sns.violinplot(nyc_data.split_frac, nyc_data.gender, color='RdBu_r');
```



This is yet another way to view the distributions among men and women.

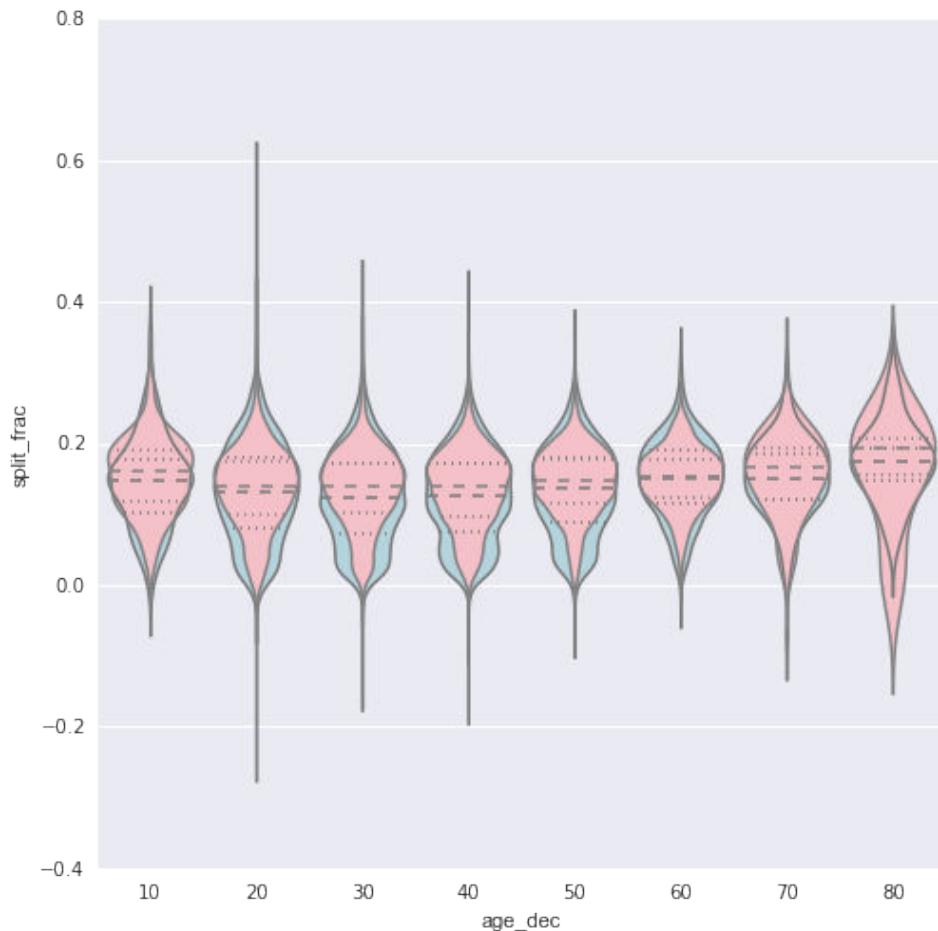
Let's look a little deeper, and compare these violin plots as a function of age. We'll start by creating a new column in the array which specifies the decade of age that each person is in:

```
nyc_data['age_dec'] = nyc_data.age.map(lambda age: 10 * (age // 10))
nyc_data.head()
```

	age	gender	split	final	split_sec	final_sec	split_frac	age_dec
0	31	M	01:06:07	02:08:43	3967	7723	-0.027321	30
1	32	M	01:06:06	02:09:07	3966	7747	-0.023880	30
2	30	M	01:06:06	02:11:22	3966	7882	-0.006344	30
3	39	M	01:06:06	02:13:10	3966	7990	0.007259	30
4	30	M	01:06:06	02:13:33	3966	8013	0.010109	30

```
men = (nyc_data.gender == 'M')
women = (nyc_data.gender == 'W')

with sns.axes_style(style=None):
    fig, ax = plt.subplots(figsize=(8, 8))
    g = sns.violinplot(nyc_data[men].split_frac, nyc_data[men].age_dec, color='lightblue')
    g = sns.violinplot(nyc_data[women].split_frac, nyc_data[women].age_dec, color='lightpink')
```

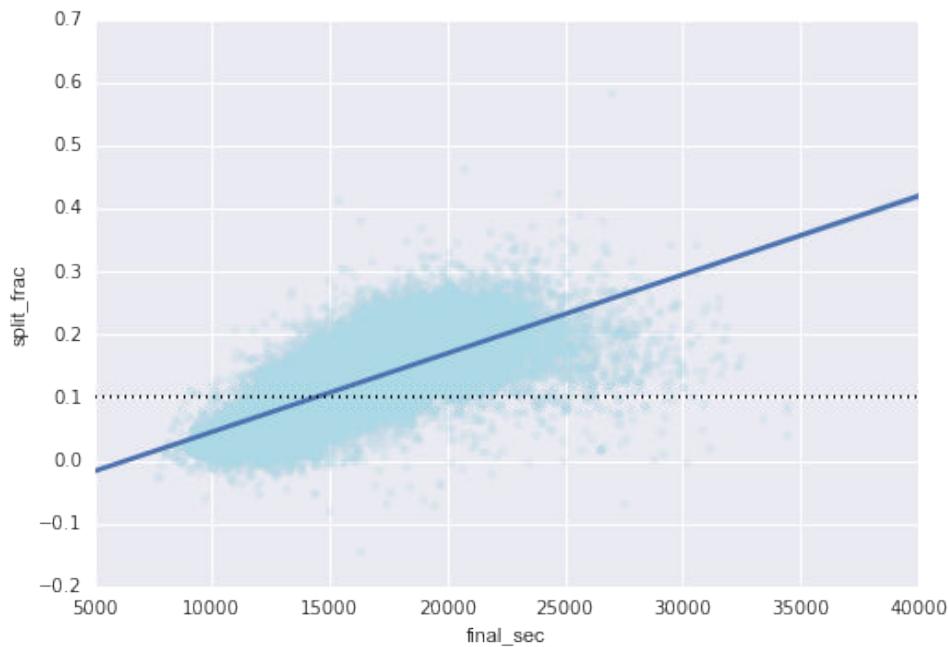


Looking at this, we can see where the distributions of men and women differ: the split distributions of men in their 20s-50s show a pronounced over-density toward lower splits when compared to women of the same age (or of any age, for that matter).

Also surprisingly, the 80-year-old women seem to out-perform *everyone* in terms of their split time. I'm not sure how to explain that.

Back to the men with fast second-halves: 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:

```
sns.regplot('final_sec', 'split_frac', data=nyc_data[men],
            scatter_kws=dict(alpha=0.2, color='lightblue'))
plt.plot([5000, 40000], [0.1, 0.1], ':k');
```



Apparently the people with fast splits are the elite runners who are finishing within ~15000 sec, or about 4 hours. People slower than that are much less likely to have a fast second split.

I would hypothesize that you could describe the distribution of runners with a two-component Gaussian distribution: there are the *elite* runners who are in shape and have fast splits, and there are the *casual* runners who are less in shape and tend to tire out more. When we get to *Unsupervised Machine Learning*, we'll have a chance to test this theory out.

CHAPTER 6

Machine Learning

In many ways, machine learning is the primary bridge between the central concept of the discipline of data science, and in practice is how data science manifests itself to the broader world. In previous chapters we have discussed how to effectively work with data, and discussed how we should think about statistical inference on this data. Machine Learning is where these computational and algorithmic skills meet the statistical thinking, and the result is a collection of approaches to inference and data exploration that are not about effective theory so much as about effective computation.

The term “machine learning” is sometimes thrown around as if it is some kind of magic pill: *apply machine learning to your data, and all your problems will be solved!* As you might expect, the reality is rarely this simple. While these methods can be incredibly powerful, to be effective they must be approached with a firm grasp of the strengths and weaknesses of each method, as well as a grasp of general concepts such as bias, variance, overfitting, underfitting, etc.

This chapter will dive into practical aspects of machine learning, primarily using Python’s `scikit-learn` package. This is not meant to be a comprehensive introduction to the field of machine learning; that is a large subject and necessitates a more technical approach than we take here. Nor is it meant to be a comprehensive manual for the use of the `scikit-learn` package. Rather, the goals of this chapter are:

- to introduce the fundamental vocabulary and concepts of Machine Learning.
- to introduce the `scikit-learn` API and show some examples of its use.
- to take a deeper-dive into the details of several of the most important machine learning approaches, and develop an intuition into how they work and when and where they are applicable.

Much of this material is drawn from the scikit-learn tutorials and workshops I have given on several occasions at PyCon, SciPy, PyData, and other conferences. Any clarity in the following pages is likely due to the many workshop participants who have given me valuable feedback on this material over the years!

Finally, if you are seeking a more comprehensive or technical treatment of any of these subjects, I've listed several resources and references at the end of this chapter.

What Is Machine Learning?

Before we take a look at the details of various machine learning methods, let's start by looking at what machine learning is, and what it isn't. Machine learning is often categorized as a sub-field of the study 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 methods, it's more helpful to think of it in relation to the statistical models described in the previous chapter.

That is, fundamentally, machine learning is about building mathematical models to help understand data. "Learning" enters the fray when we give these models **tunable parameters** which can be adapted to observed data; in this way the program can be considered to be "learning" from the data. Once these models have been fit to previously seen data, they can be used to predict aspects of newly-observed data. I'll leave to the reader the more philosophical digression regarding the extent to which this type mathematical model-based "learning" is similar to the "learning" exhibited by the human brain.

Machine Learning vs. Statistical Modeling

At the very basic level, machine learning is simply about building models and fitting them to data. Given that, you would be right to wonder what the difference is between machine learning and the statistical modeling we discussed in the previous chapter! Indeed, this seems to be a matter of much debate, and many answers to this question tend to be tongue-in-cheek quips, along the lines of "Machine learning is statistical modeling done in the Computer Science building". An interesting early writeup of the divide can be found in [Statistical Modeling: the Two Cultures \(2001\)](#) by Leo Breiman.

My own observation is that the fundamental divide between the two approaches is this:

- **Statistical Modeling** tends to focus on the model as an end in itself, i.e. what the model *means*. The fundamental explanation of the world comes through understanding the model and its particular parameter values.

- **Machine Learning** tends to focus on the application of models to new data, i.e. what the model *predicts*. The fundamental explanation of the world comes through observing what the model predicts about new data.

Certainly, this is not a neat divide – statistical modeling at times does deal with prediction and machine learning at times does interpret model parameters – but this is a helpful broad distinction.

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 determined, it can be used to apply labels to new, unknown data. This is further subdivided into **Classification** tasks and **Regression** tasks: in classification, the labels are discrete categories, while in regression, the labels are continuous quantities. Below we will see examples of supervised learning, both classification and regression.

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. Below we will see examples of unsupervised learning, both clustering and dimensionality reduction.

In addition, there are so-called **Semi-supervised learning** methods which combine some aspects of both the above methods. These are often useful when labels are available for only a part of the dataset.

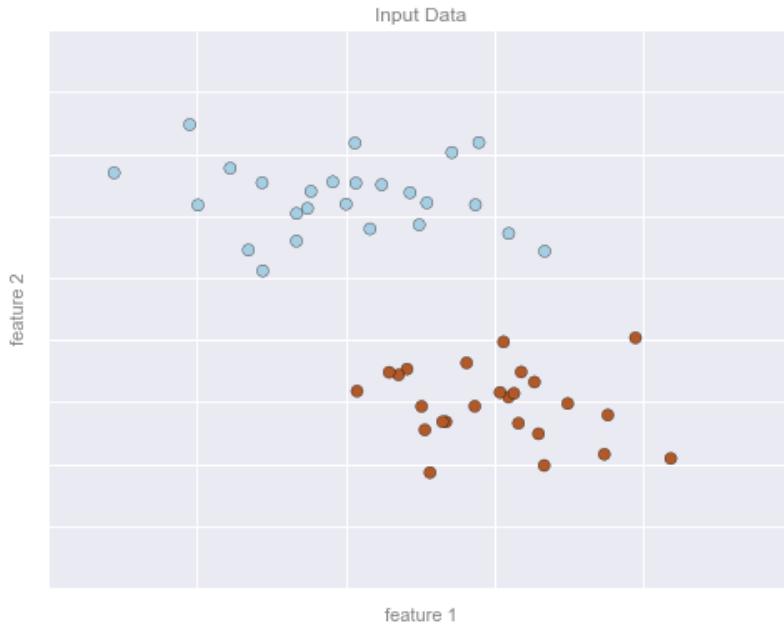
Qualitative Examples of Machine Learning Applications

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 can find the Python source which generates the following figures at the end of the section.

Classification: Predicting Discrete Labels

We will first take a look at a simple *classification* task, in which you are given a set of labeled points and want to use these to classify some unlabeled points.

Imagine that we have the following data:



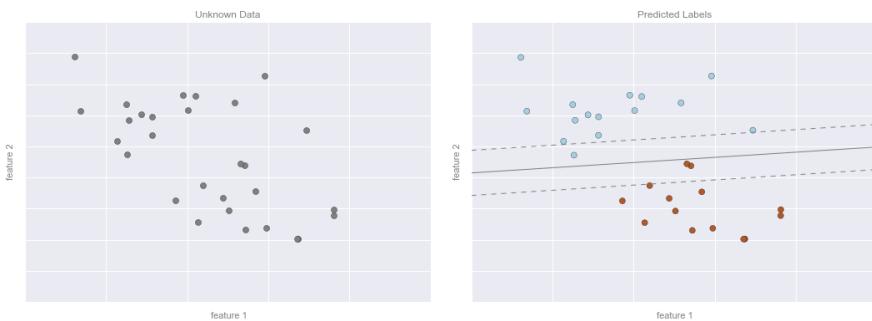
Here we have two-dimensional data: that is, there we have two *features* for each point, represented by the (x,y) positions of the points on the plane. In addition, we have one of two *class labels* for each point, here represented by the colors of the points. From these features and labels, we would like to create a model which 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 will use an extremely simple one. We will make the assumption that the two groups can be 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 quantitative version of the statement “a straight line separates the classes”, while the *model parameters* are the particular numbers describing the location and orientation of that line for our data. The optimal values for these model parameters are learned from the data (this is the “learning” in machine learning), which is often called *training the model*.

Here is a visual representation of what the trained model looks like for this data:



Now that this model has been trained, it can be generalized to new, unlabeled data. In other words, we can take a new set of data, draw this model line through it, and assign labels to the new points based on this model. This stage is usually called *prediction*.



This is the basic idea of a classification task in machine learning, where “classification” indicates that the data have 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 more dimensions.

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”

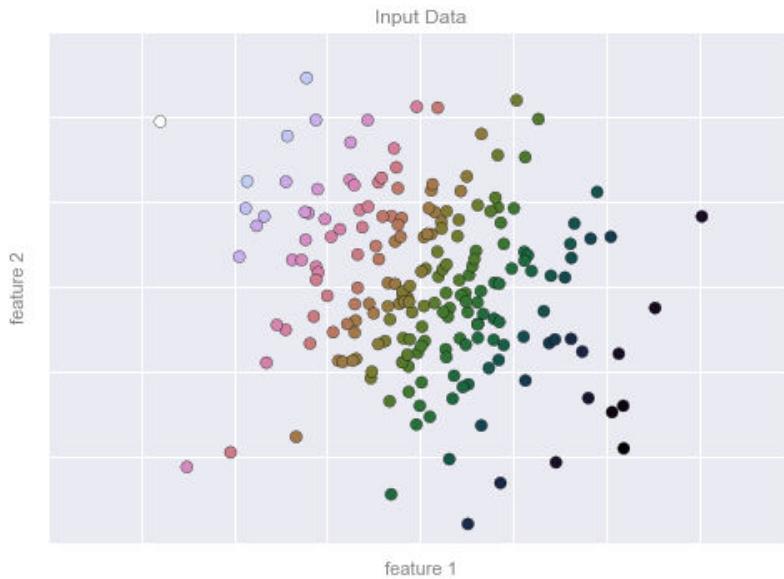
For the training set, these labels might be determined by individual inspection of a small representative sample of emails; for the rest of the emails, the label would be determined using the model. For a suitably trained classification algorithm with enough well-constructed features (typically thousands or millions of words or phrases), this type of approach can be very effective. We will see an example of such text-based classification in Section X.X.

Some important classification algorithms that we will discuss in more detail are Gaussian Naive Bayes (Section X.X), Random Forest Classification (Section X.X), and Support Vector Machines (Section X.X).

Regression: Predicting Continuous Labels

In contrast with the discrete labels of a classification algorithm, we will next look at a simple *regression* task in which the labels are continuous quantities.

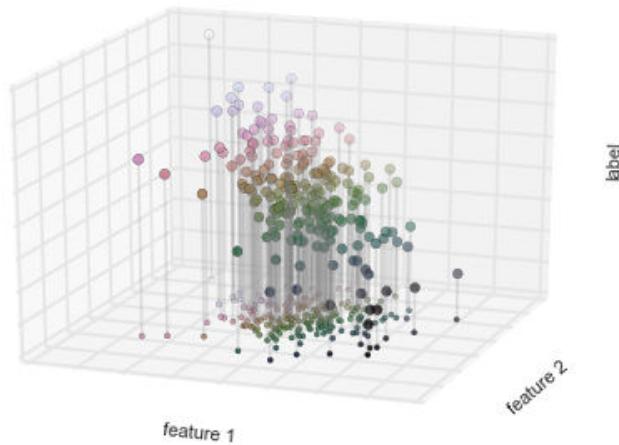
Consider the following data, which consists of a set of points each with a continuous label:



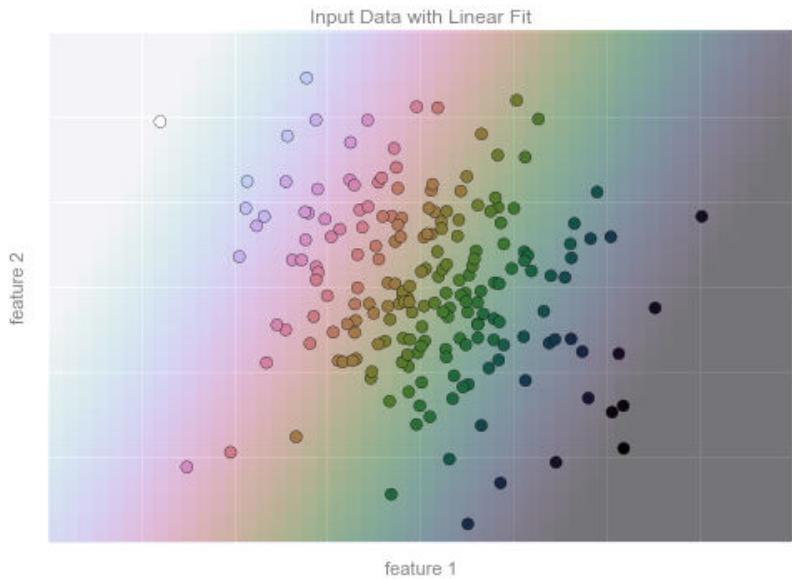
As with the classification example, we have two-dimensional data: that is, there are two features describing each data point. The color of each point represents the continuous label for that point.

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 linear regression 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.

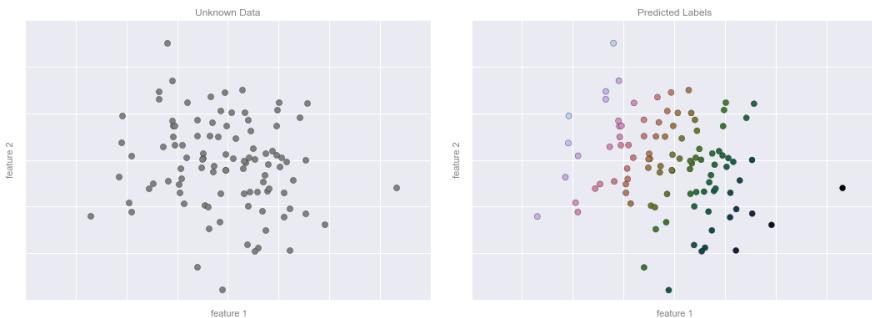
We can visualize this setup as follows:



Notice that the *feature 1*-*feature 2* plane here is the same as in the two-dimensional plot above; what we have done is to represent the labels by both color and 3D axis position. From this view, it seems reasonable that fitting a plane through this three-dimensional data would allow us to predict the expected label for any set of input parameters. Returning to the 2D projection, when we fit such a plane we get the following result:



This plane of fit gives us what we need to predict labels for new points. Visually, we find the following:



As with the classification example above, 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 observed through 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 wavelengths or colors

- *label* → distance or redshift of the galaxy

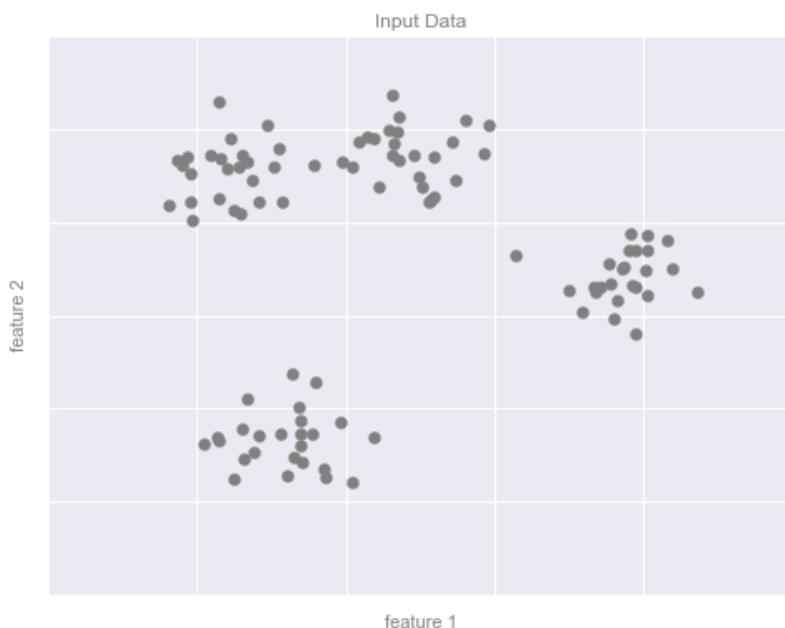
The distances for a small number of these galaxies might be determined through an independent set of (typically more expensive) observations. Distances to remaining galaxies could then be estimated using a suitable regression model, without the need to employ the more expensive observation across the entire set. In astronomy circles, this is known as the “photometric redshift” problem.

Some important regression algorithms that we will discuss are Linear Regression (section X.X) Random Forest Regression (Section X.X) and Support Vector Machines (Section X.X).

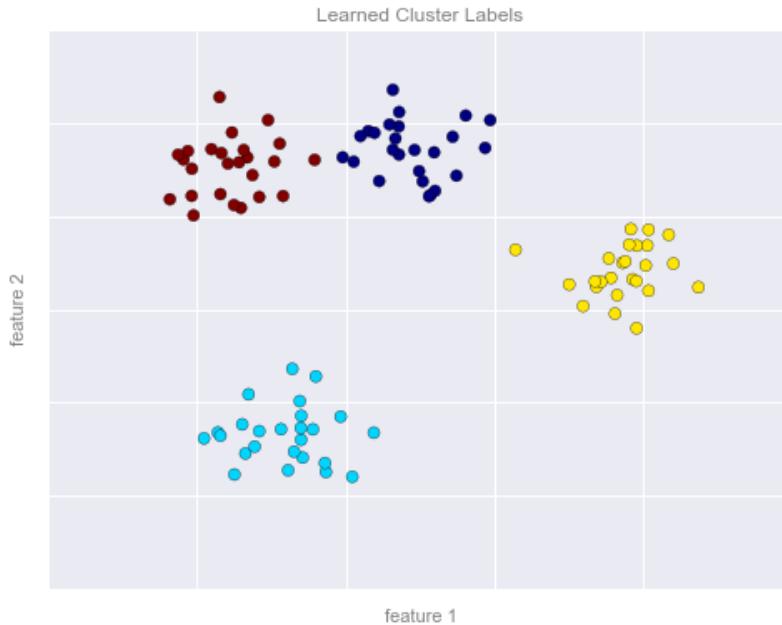
Clustering: Inferring Labels on Unlabeled Data

The classification and regression illustrations above are examples of supervised learning algorithms, in which we are trying to build a model which will predict labels for new data. Unsupervised learning involves models which describe data without reference to any known labels.

One common case of unsupervised learning is “clustering”, in which data is automatically assigned to some number of discrete groups. For example, we might have some two-dimensional data which looks like the following:



By eye, it is clear that these data are members of distinct groups. 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 (Section X.X), we find the following clusters within this data:



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 useful information from the dataset.

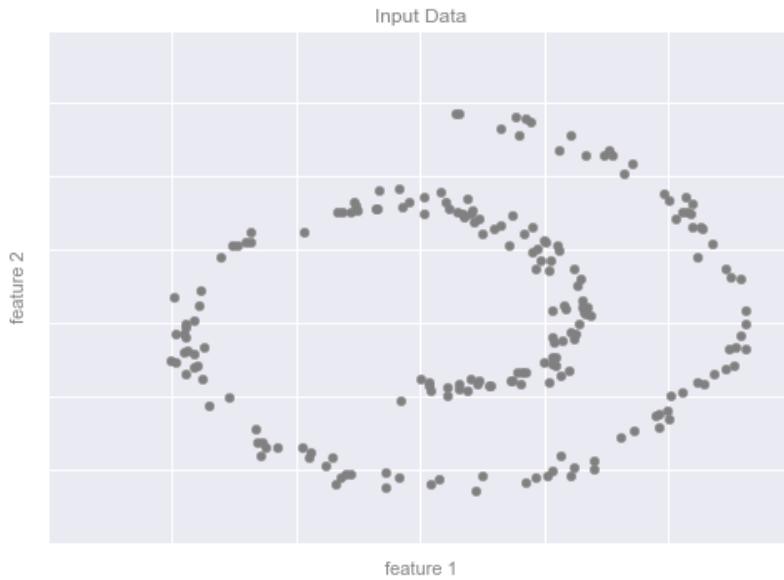
We will discuss the K Means algorithm in more depth in Section X.X. Other important clustering algorithms include Gaussian Mixture Models (Section X.X) and Spectral clustering (Section X.X).

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 other examples above, but generally it seeks to pull-out some low-dimensional representation of data which in some way preserves relevant qualities of the full dataset. Different dimensionality

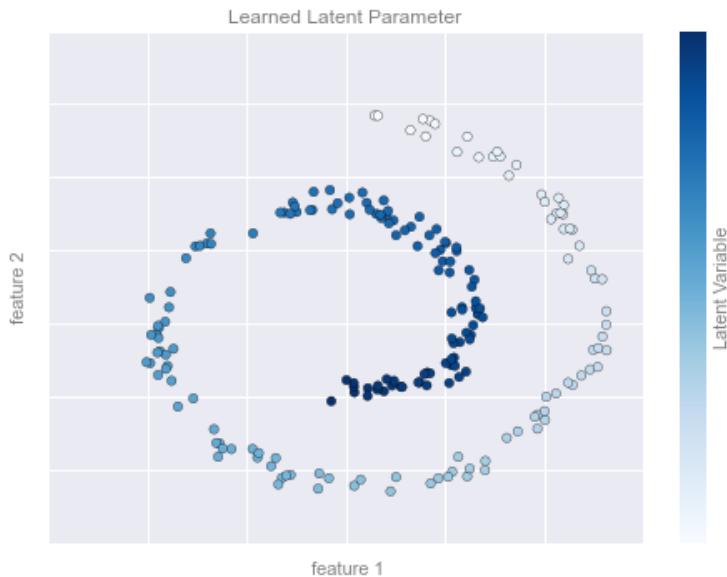
reduction routines measure these relevant qualities in different ways, as we will see in Section X.X.

As an example of this, consider the following data:



Visually, it is clear that there is some structure in this data: it is drawn from a one-dimensional line which 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 this one-dimensional data is embedded in higher-dimensional space. A suitable dimensionality reduction model in this case would be sensitive to this nonlinear embedded structure, and be able to pull-out this lower-dimensionality representation.

The following is a visualization of the results of the Isomap algorithm, a manifold learning algorithm that does exactly this:



Notice that the colors (which represent the extracted one-dimensional latent variable) change uniformly along the spiral, which indicates that the algorithm did in fact detect the structure we saw by eye. As with the above examples, the power of dimensionality reduction algorithms becomes more clear in higher-dimensional cases. For example, we might wish to visualize important relationships within a dataset that has 100 or 1000 features. Visualizing 1000-dimensional data is a challenge, and one way we can make this more manageable is to use a dimensionality reduction technique to reduce the data to two or three dimensions.

Some important dimensionality reduction algorithms that we will discuss are Principal Component Analysis (Section X.X) and various manifold learning algorithms, including Isomap and Locally Linear Embedding (Section X.X).

Summary

Here we have seen a few simple examples of some of the basic types of machine learning approaches. Needless to say, there are a number of important practical details that we 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 which can predict labels based on labeled training data
 - **Classification:** models which predict labels as two or more discrete categories
 - **Regression:** models which predict continuous labels
- **Unsupervised Learning:** models which identify structure in unlabeled data
 - **Clustering:** models which detect and identify distinct groups in the data
 - **Dimensionality Reduction:** models which 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.

Figure Code

All of the figures in the above discussion are generated based on actual machine learning computations; following is the code which implements it. If you have not yet read through later sections of this chapter, you can feel free to skip over this code for the moment.

General Imports

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
import numpy as np

import os
if not os.path.exists('fig'):
    os.makedirs('fig')

# common plot formatting for below
def format_plot(ax, title):
    ax.xaxis.set_major_formatter(plt.NullFormatter())
    ax.yaxis.set_major_formatter(plt.NullFormatter())
    ax.set_xlabel('feature 1', color='gray')
    ax.set_ylabel('feature 2', color='gray')
    ax.set_title(title, color='gray')
```

Classification Example Figures

The following code generates the figures from the Classification section above.

```
from sklearn.datasets.samples_generator import make_blobs
from sklearn.svm import SVC

# create 50 separable points
X, y = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)
```

```

# fit the support vector classifier model
clf = SVC(kernel='linear')
clf.fit(X, y)

# create some new points to predict
X2, _ = make_blobs(n_samples=80, centers=2,
                    random_state=0, cluster_std=0.80)
X2 = X2[50:]

# predict the labels
y2 = clf.predict(X2)

```

Classification Example Figure 1.

```

# plot the data
fig, ax = plt.subplots(figsize=(8, 6))
point_style = dict(cmap='Paired', s=50)
ax.scatter(X[:, 0], X[:, 1], c=y, **point_style)

# format plot
format_plot(ax, 'Input Data')
ax.axis([-1, 4, -2, 7])

fig.savefig('fig/07.01-classification-1.png')
plt.close(fig)

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
 if self._edgecolors == str('face'):

Classification Example Figure 2.

```

# Get contours describing the model
xx = np.linspace(-1, 4, 10)
yy = np.linspace(-2, 7, 10)
xy1, xy2 = np.meshgrid(xx, yy)
Z = np.array([clf.decision_function(t)
              for t in zip(xy1.flat, xy2.flat)]).reshape(xy1.shape)

# plot points and model
fig, ax = plt.subplots(figsize=(8, 6))
line_style = dict(levels = [-1.0, 0.0, 1.0],
                  linestyles = ['dashed', 'solid', 'dashed'],
                  colors = 'gray', linewidths=1)
ax.scatter(X[:, 0], X[:, 1], c=y, **point_style)
ax.contour(xy1, xy2, Z, **line_style)

# format plot
format_plot(ax, 'Model Learned from Input Data')
ax.axis([-1, 4, -2, 7])

fig.savefig('fig/07.01-classification-2.png')
plt.close(fig)

```

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

Classification Example Figure 3.
# plot the results
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

ax[0].scatter(X2[:, 0], X2[:, 1], c='gray', **point_style)
ax[0].axis([-1, 4, -2, 7])

ax[1].scatter(X2[:, 0], X2[:, 1], c=y2, **point_style)
ax[1].contour(xy1, xy2, Z, **line_style)
ax[1].axis([-1, 4, -2, 7])

format_plot(ax[0], 'Unknown Data')
format_plot(ax[1], 'Predicted Labels')

fig.savefig('fig/07.01-classification-3.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

Regression Example Figures

The following code generates the figures from the regression section above.

```

from sklearn.linear_model import LinearRegression

# Create some data for the regression
rng = np.random.RandomState(1)

X = rng.randn(200, 2)
y = np.dot(X, [-2, 1]) + 0.1 * rng.randn(X.shape[0])

# fit the regression model
model = LinearRegression()
model.fit(X, y)

# create some new points to predict
X2 = rng.randn(100, 2)

# predict the labels
y2 = model.predict(X2)
```

Regression Example Figure 1.

```

# plot data points
fig, ax = plt.subplots()
points = ax.scatter(X[:, 0], X[:, 1], c=y, s=50,
                     cmap='cubehelix')

# format plot
format_plot(ax, 'Input Data')
ax.axis([-4, 4, -3, 3])

fig.savefig('fig/07.01-regression-1.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```

Regression Example Figure 2.

```

from mpl_toolkits.mplot3d.art3d import Line3DCollection

points = np.hstack([X, y[:, None]]).reshape(-1, 1, 3)
segments = np.hstack([points, points])
segments[:, 0, 2] = -8

# plot points in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X[:, 0], X[:, 1], y, c=y, s=35,
           cmap='cubehelix')
ax.add_collection3d(Line3DCollection(segments, colors='gray', alpha=0.2))
ax.scatter(X[:, 0], X[:, 1], -8 + np.zeros(X.shape[0]), c=y, s=10,
           cmap='cubehelix')

# format plot
ax.patch.set_facecolor('white')
ax.view_init(elev=20, azim=-70)
ax.set_zlim3d(-8, 8)
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.yaxis.set_major_formatter(plt.NullFormatter())
ax.zaxis.set_major_formatter(plt.NullFormatter())
ax.set(xlabel='feature 1', ylabel='feature 2', zlabel='label')

# Hide axes (is there a better way?)
ax.w_xaxis.line.set_visible(False)
ax.w_yaxis.line.set_visible(False)
ax.w_zaxis.line.set_visible(False)
for tick in ax.w_xaxis.get_ticklines():
    tick.set_visible(False)
for tick in ax.w_yaxis.get_ticklines():
    tick.set_visible(False)
for tick in ax.w_zaxis.get_ticklines():
    tick.set_visible(False)

```

```

fig.savefig('fig/07.01-regression-2.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):

Regression Example Figure 3.

from matplotlib.collections import LineCollection

# plot data points
fig, ax = plt.subplots()
pts = ax.scatter(X[:, 0], X[:, 1], c=y, s=50,
                  cmap='cubehelix', zorder=2)

# compute and plot model color mesh
xx, yy = np.meshgrid(np.linspace(-4, 4),
                      np.linspace(-3, 3))
Xfit = np.vstack([xx.ravel(), yy.ravel()]).T
yfit = model.predict(Xfit)
zz = yfit.reshape(xx.shape)
ax.pcolorfast([-4, 4], [-3, 3], zz, alpha=0.5,
              cmap='cubehelix', norm=pts.norm, zorder=1)

# format plot
format_plot(ax, 'Input Data with Linear Fit')
ax.axis([-4, 4, -3, 3])

fig.savefig('fig/07.01-regression-3.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```

Regression Example Figure 4.

```

# plot the model fit
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

ax[0].scatter(X2[:, 0], X2[:, 1], c='gray', s=50)
ax[0].axis([-4, 4, -3, 3])

ax[1].scatter(X2[:, 0], X2[:, 1], c=y2, s=50,
              cmap='cubehelix', norm=pts.norm)
ax[1].axis([-4, 4, -3, 3])

# format plots
format_plot(ax[0], 'Unknown Data')
format_plot(ax[1], 'Predicted Labels')


```

```

fig.savefig('fig/07.01-regression-4.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```

Clustering Example Figures

The following code generates the figures from the clustering section above.

```

from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import KMeans

# create 50 separable points
X, y = make_blobs(n_samples=100, centers=4,
                   random_state=42, cluster_std=1.5)

# Fit the K Means model
model = KMeans(4, random_state=0)
y = model.fit_predict(X)

```

Clustering Example Figure 1.

```

# plot the input data
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(X[:, 0], X[:, 1], s=50, color='gray')

# format the plot
format_plot(ax, 'Input Data')

fig.savefig('fig/07.01-clustering-1.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```

Clustering Example Figure 2.

```

# plot the data with cluster labels
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(X[:, 0], X[:, 1], s=50, c=y, cmap='jet')

# format the plot
format_plot(ax, 'Learned Cluster Labels')

fig.savefig('fig/07.01-clustering-2.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```

Dimensionality Reduction Example Figures

The following code generates the figures from the dimensionality reduction section above.

Dimensionality Reduction Example Figure 1.

```
from sklearn.datasets import make_swiss_roll

# make data
X, y = make_swiss_roll(200, noise=0.5, random_state=42)
X = X[:, [0, 2]]

# visualize data
fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], color='gray', s=30)

# format the plot
format_plot(ax, 'Input Data')

fig.savefig('fig/07.01-dimesionality-1.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

Dimensionality Reduction Example Figure 2.

```
from sklearn.manifold import Isomap

model = Isomap(n_neighbors=8, n_components=1)
y_fit = model.fit_transform(X).ravel()

# visualize data
fig, ax = plt.subplots()
pts = ax.scatter(X[:, 0], X[:, 1], c=y_fit, cmap='Blues', s=30)
cb = fig.colorbar(pts, ax=ax)

# format the plot
format_plot(ax, 'Learned Latent Parameter')
cb.set_ticks([])
cb.set_label('Latent Variable', color='gray')

fig.savefig('fig/07.01-dimesionality-2.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

Introducing Scikit-Learn

There are several Python libraries which provide solid implementations of a range of machine learning algorithms. One of the best known is **scikit-learn**, a package which 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 and complete 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 API of scikit-learn; 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 tools for exploring a set of images of hand-written 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 **data tables**.

Data As Table

A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements. For example, consider the [iris dataset](#), famously analyzed by Fisher in 1936. We can download this dataset in the form of a Pandas dataframe using the [seaborn](#) library:

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

	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 information that describes each sample. In general, we will refer to the columns of the matrix as *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 convention, this features matrix is often stored in a variable named `X`. The features matrix is assumed to be two-dimensional, with shape `[n_samples, n_features]`, and is most often contained in a NumPy array or Pandas DataFrame, though some scikit-learn models also accept SciPy sparse matrices.

The samples 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 always refer to the distinct observations which describe each sample in a quantitative manner. Features are generally real-valued, but may be boolean or discrete-valued in some cases.

Target Array

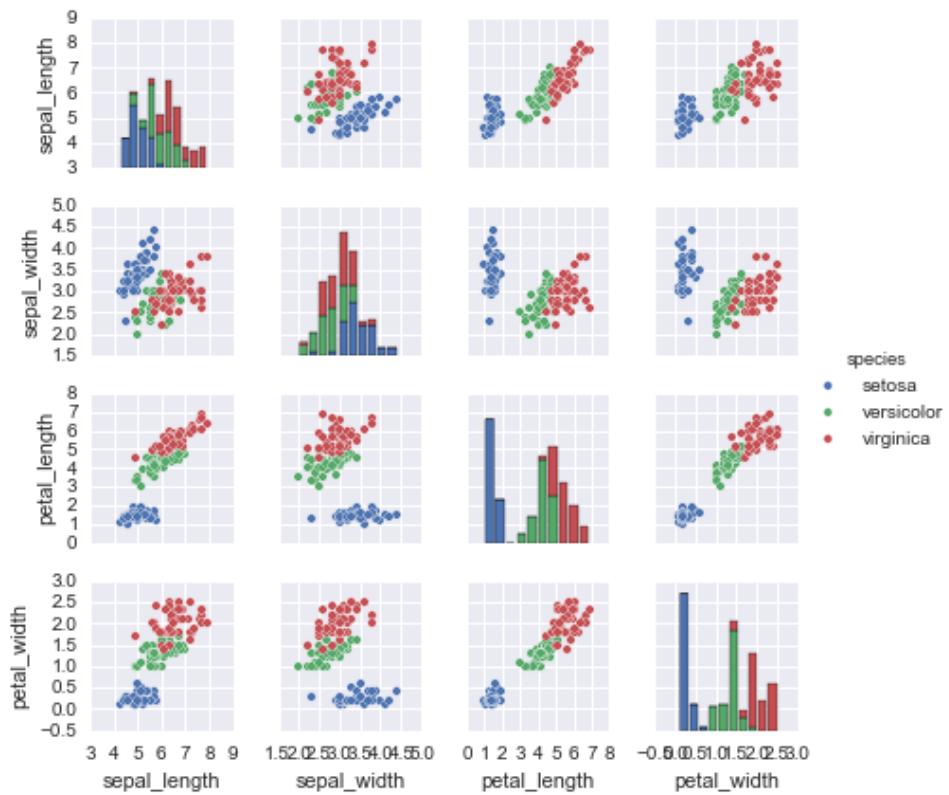
In addition to the feature matrix `X`, we also generally work with a *label* or *target* array, which by convention we will usually call `y`. The target array is usually one-dimensional, with length `n_samples`, and is generally contained in a NumPy array or Pandas Series. The target array may have continuous numerical values, or discrete classes/labels. While some scikit-learn estimators do handle multiple target values in the form of a two-dimensional, `[n_samples, n_targets]` target array, we will primarily 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 quantity we want to *predict from the data*: in statistical terms, it is the dependent variable. For example, in the above 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` column would be considered the feature.

With this target array in mind, we can use Seaborn (see Section X.X) to conveniently visualize the data:

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', size=1.5);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

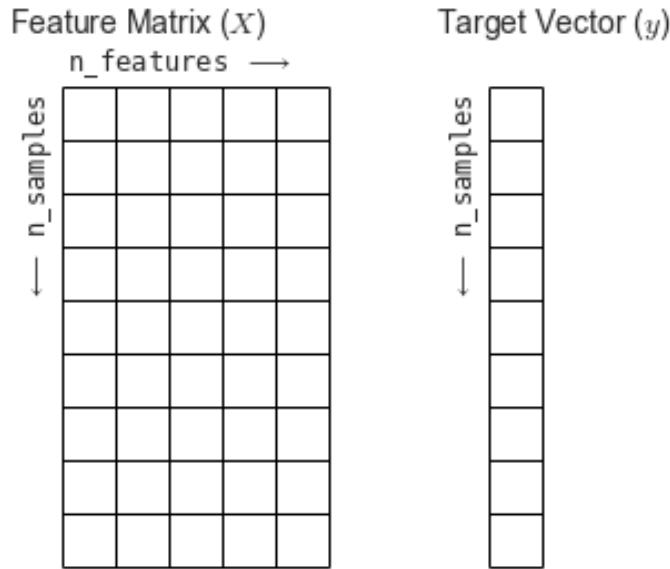


For use in scikit-learn, we will extract the features matrix and target array from the dataframe, which we can do using the Pandas dataframe operations discussed in Section X.X:

```
X_iris = iris.drop('species', axis=1)
X_iris.shape
(150, 4)

y_iris = iris['species']
y_iris.shape
(150,)
```

To summarize, the expected layout of features and target values is visualized in the following diagram:



With this data properly formatted, we can move on to consider the *estimator* API of scikit-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, with consistent 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 DataFrames, SciPy sparse matrices) and parameter names use standard Python strings.
- **Composition:** Many machine learning tasks can be expressed as sequences of more fundamental algorithms, and scikit-learn makes use of this wherever possible.
- **Sensible defaults:** When models require user-specified parameters, the library defines an appropriate default value.

In practice, these principles make scikit-learn very easy to use, once the basic principles are understood. Every machine learning algorithm in Scikit-learn is implemented 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 below:

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 discussion above
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 data** using the `transform()` or `predict()` method.

Below we will step through several simple examples of applying supervised and unsupervised learning methods.

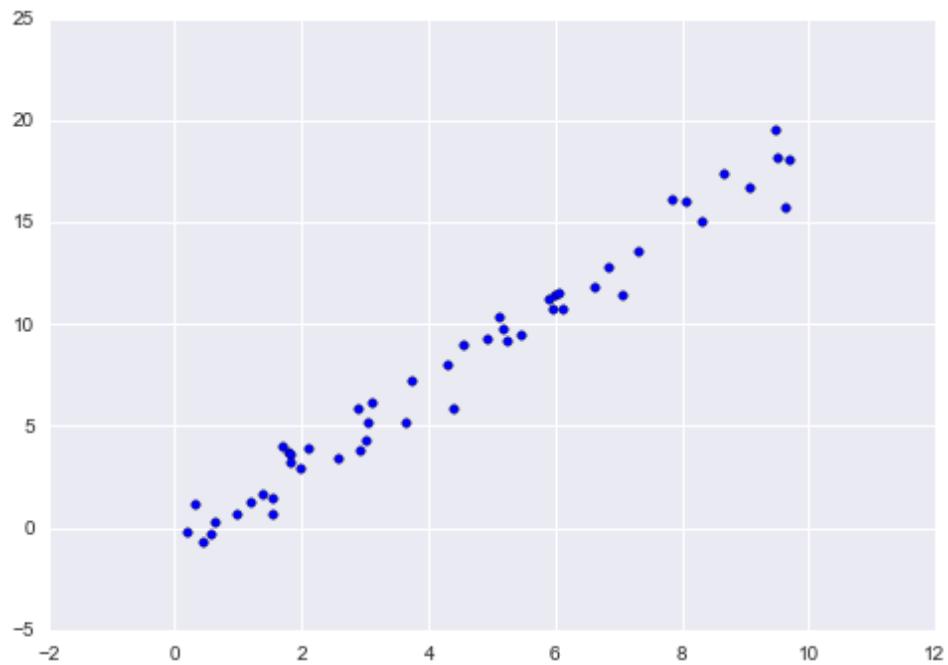
Supervised Learning Example: Simple Linear Regression

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:

```
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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



With this data in place, we can follow the above recipe.

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:

```
from sklearn.linear_model import LinearRegression
```

Note that other more general linear regression models exist as well; you can read more 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 answer one 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 model class 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, hyperparameters are chosen by passing values at model instantiation. We will explore in Section X.X how you can quantitatively motivate the choice of hyperparameters.

For our linear regression example, we can instantiate the `LinearRegression` class and specify that we would like to fit the intercept using the `fit_intercept` hyperparameter:

```
model = LinearRegression(fit_intercept=True)
model
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 storing of these hyperparameter values. In particular, we have not yet applied the model to any data: the scikit-learn API makes very clear the distinction between *choice of model* and *application of model to data*.

3. Arrange data into a features matrix and target vector. Above 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 we need 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:

```
X = x[:, np.newaxis]
X.shape
(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:

```
model.fit(X, y)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

This `fit()` command causes a number of model-dependent internal computations 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 which were learned during the `fit()` process have trailing underscores; for example in this linear model, we have the following:

```
model.coef_
array([ 1.9776566])
model.intercept_
```

```
-0.90331072553111635
```

These two parameters represent the slope and intercept of the simple linear fit to the data. Comparing to the data definition above, 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: this is much more a *statistical modeling* question than a *machine learning* question. Recall from Section X.X that we differentiated between statistical modeling, which generally asks what the models and their parameters *mean*, and machine learning, which generally asks what the models and their parameters *predict*. If you want to understand what the model parameters mean, machine learning is not really the appropriate set of tools: instead, I would recommend a Bayesian modeling approach to your problem (see Section X.X).

5. 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 which was not part of the training set. In scikit-learn, this can be done 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:

```
xfit = np.linspace(-1, 11)
```

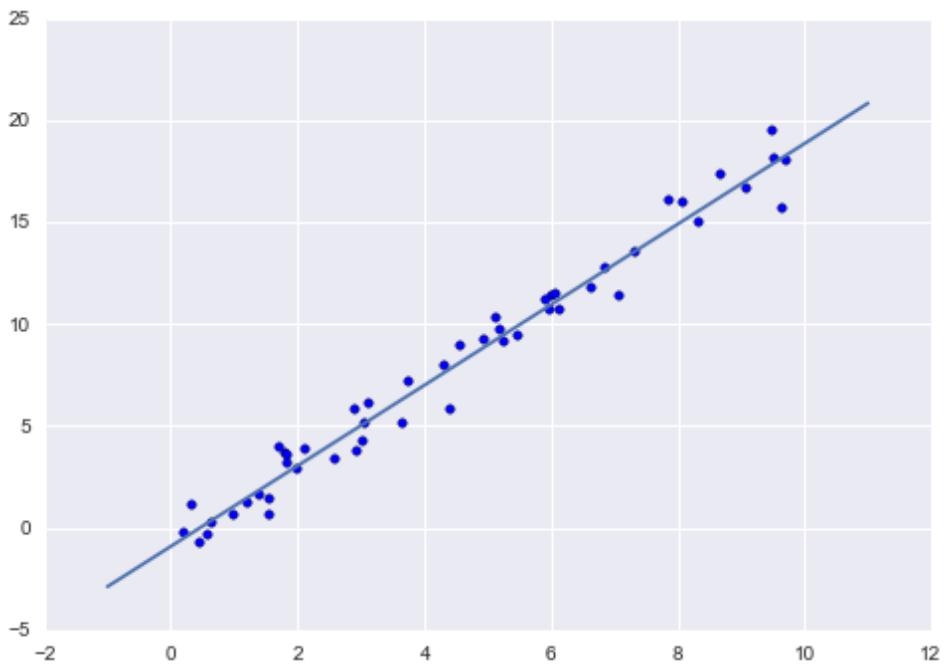
As above, we need to coerce these x values into a `[n_samples, n_features]` features matrix, after which we can feed it to the model:

```
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:

```
plt.scatter(x, y)
plt.plot(xfit, yfit);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
  if self._edgecolors == str('face'):
```



Typically the efficiency of the model is evaluated by comparing its results to some known baseline, as we will see in the next example

Supervised Learning Example: Iris Classification

Let's take a look at another example of this process, using the iris dataset shown above. 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 Section X.X 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 exploring 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 is more convenient to use the `train_test_split` utility function:

```
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 the recipe above to predict our labels:

```
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 labels which match their true value:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)

0.97368421052631582
```

Evidently, even this very naive classification algorithm 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 data is four-dimensional: there are four features mentioned for each sample.

The task of dimensionality reduction is to ask: can we find 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 easier to plot data in two dimensions than in four dimensions or higher!

Here we will use Principal Component Analysis (PCA; see Section X.X), which is a fast linear dimensionality reduction technique. We will ask the model to return two components – that is, a two-dimensional representation of the data.

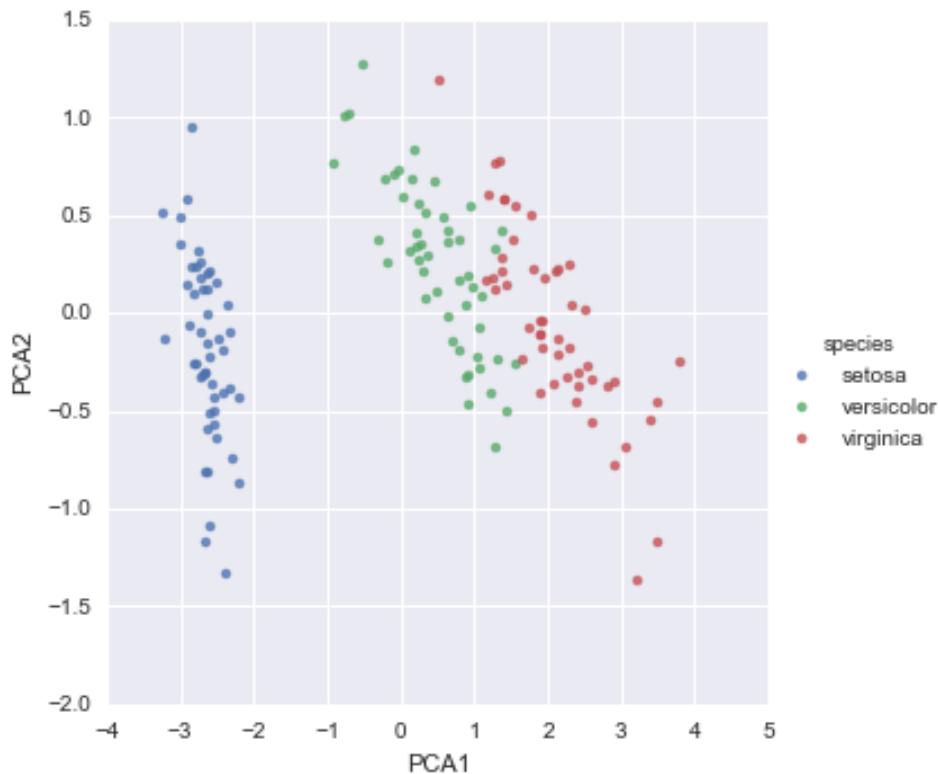
Following the sequence of steps above, we have:

```
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 original iris dataframe, and use seaborn's `lmplot` to show the results:

```
iris['PCA1'], iris['PCA2'] = X_2D.T
sns.lmplot("PCA1", "PCA2", data=iris, hue='species', fit_reg=False);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



We see that in the two-dimensional representation, the species are fairly well-separated, even though the PCA algorithm had no knowledge of the species labels!

Unsupervised Learning: Iris Clustering

Let's next look at applying clustering to the iris data. A clustering algorithm attempts to 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 Section X.X. A GMM attempts to model the data as a collection of Gaussian blobs.

We can fit the Gaussian mixture model as follows:

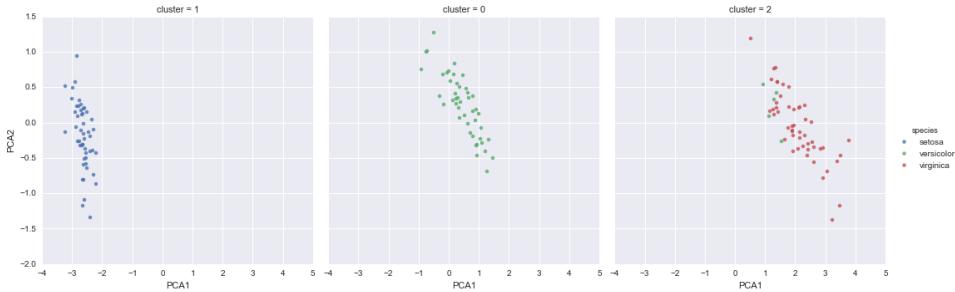
```
from sklearn.mixture import GMM      # 1. Choose the model class
model = GMM(n_components=3,          # 2. Instantiate the model with hyperparameters
            covariance_type='full')
model.fit(X_iris)                  # 3. Fit to data. Notice y is not specified!
y_gmm = model.predict(X_iris)       # 4. Determine cluster labels
```

As above, we will add the cluster label to the iris DataFrame and use seaborn to plot the results:

```

iris['cluster'] = y_gmm
sns.lmplot("PCA1", "PCA2", data=iris, hue='species', col='cluster', fit_reg=False);
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



By splitting the data by cluster number, we see exactly how well the GMM algorithm has recovered the underlying label: the *setosa* species is separated perfectly, while there remains a small amount of mixing between *versicolor* and *virginica*. What this tells us is that even without an expert to tell us the species labels of the individual 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 field clues as to the relationship between the samples they are observing.

Application: Exploring Hand-written Digits

To demonstrate the above principles on a more interesting problem, let's consider one piece of the Optical Character Recognition problem: the identification of hand-written 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 pre-formatted digits, which is built-in to the library.

Loading and visualizing the digits data

We'll use scikit-learn's data access interface and take a look at this data:

```

from sklearn.datasets import load_digits
digits = load_digits()
digits.images.shape
(1797, 8, 8)

```

The images data is a three-dimensional array: 1797 samples each consisting of an 8x8 grid of pixels. Let's visualize the first hundred of these:

```

import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(8, 8),

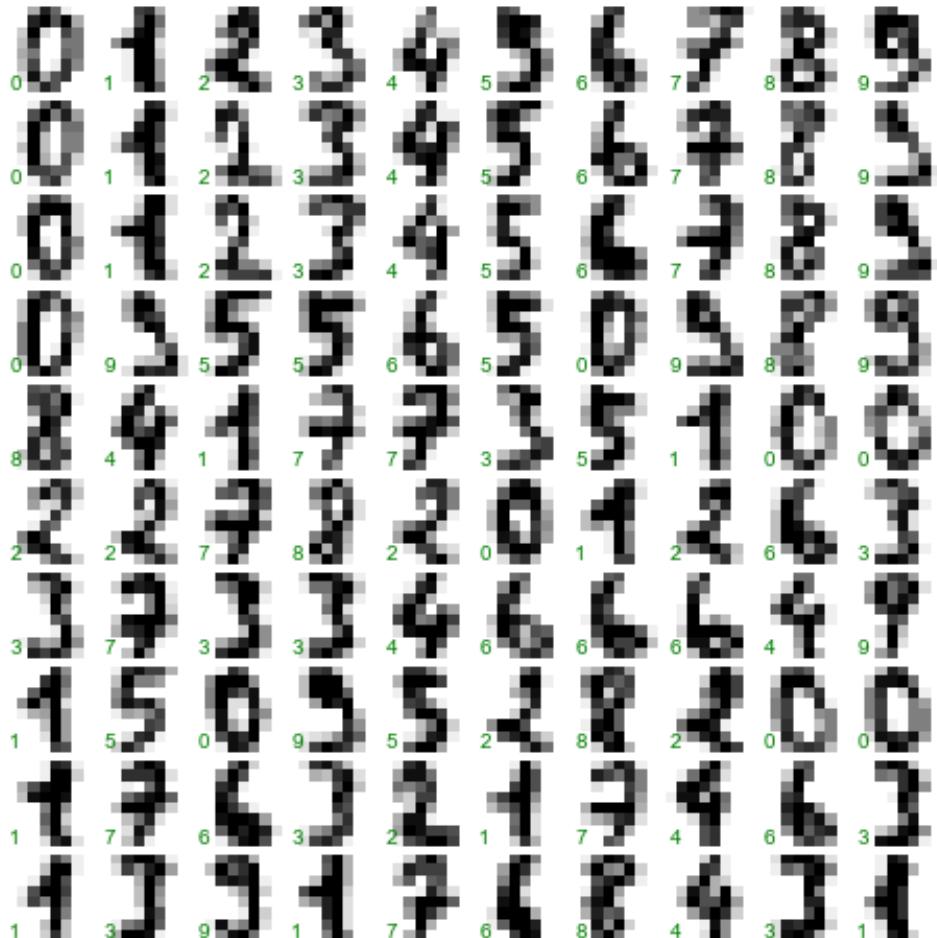
```

```

    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')

```



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 two quantities are built-in to the digits dataset under the `data` and `target` attributes, respectively:

```
X = digits.data
X.shape
(1797, 64)

y = digits.target
y.shape
(1797,)
```

We see here that there are 1797 samples and 64 features.

Unsupervised Learning: Dimensionality Reduction

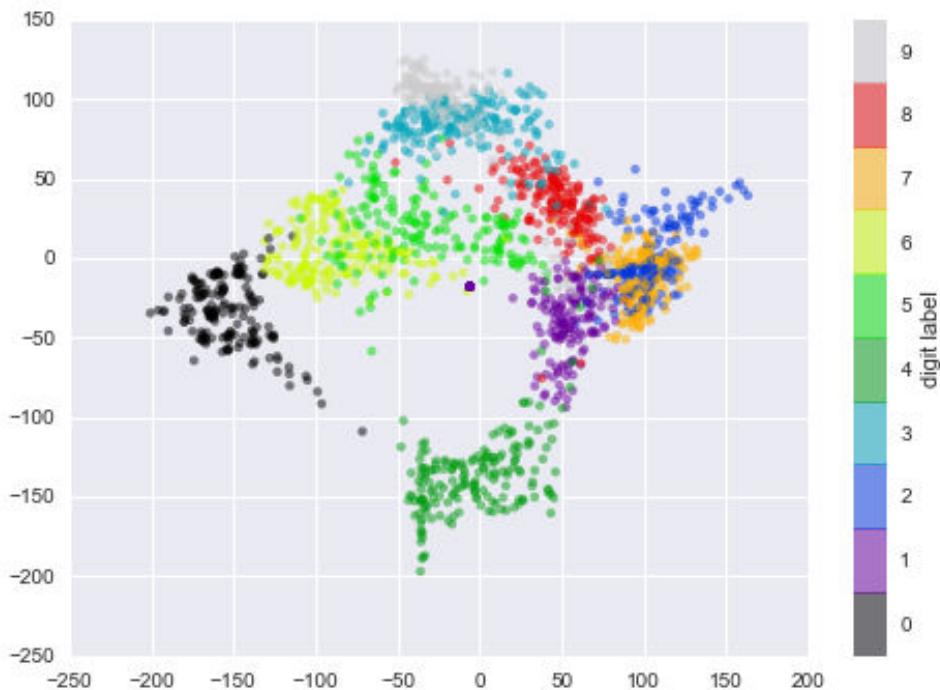
We'd like to visualize our points within the 64-dimensional parameter space, but it's difficult to effectively visualize points in such a high-dimensional space. Instead we'll reduce the dimensions to 2, using an unsupervised method. Here, we'll make use of a manifold learning algorithm called *Isomap* (see Section X.X), and transform the data to two dimensions.

```
from sklearn.manifold import Isomap
iso = Isomap(n_components=2)
iso.fit(digits.data)
data_projected = iso.transform(digits.data)
data_projected.shape
(1797, 2)
```

We see that the projected data is now two-dimensional. Let's plot this data to see if we can learn anything from its structure:

```
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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



This plot gives us some good intuition into how well various numbers are separated in 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 parameter space: this tells us that even a very straightforward supervised classification algorithm should perform suitably on this data. Let’s give it a try.

Classification on Digits

Let’s apply a classification algorithm to the digits. As with the iris data above, we will split the data into a train and test set, and fit the model with Gaussian Naive Bayes:

```
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=0)
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:

```
from sklearn.metrics import accuracy_score
accuracy_score(ytest, y_model)

0.8088888888888888
```

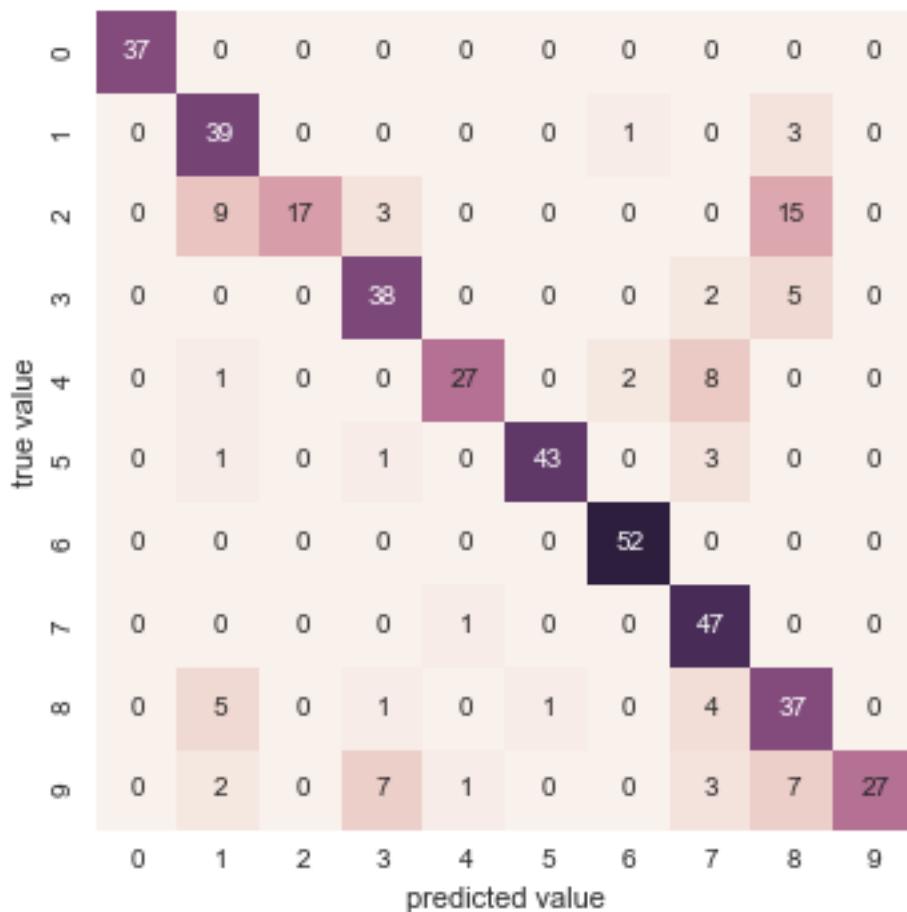
With even this extremely simplistic model, we find about 80% accuracy for classification of the digits! However, this single number doesn't tell us **where** we've gone wrong: one nice way to do this is to use the *confusion matrix*, which we can compute with scikit-learn and plot with Seaborn:

```
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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



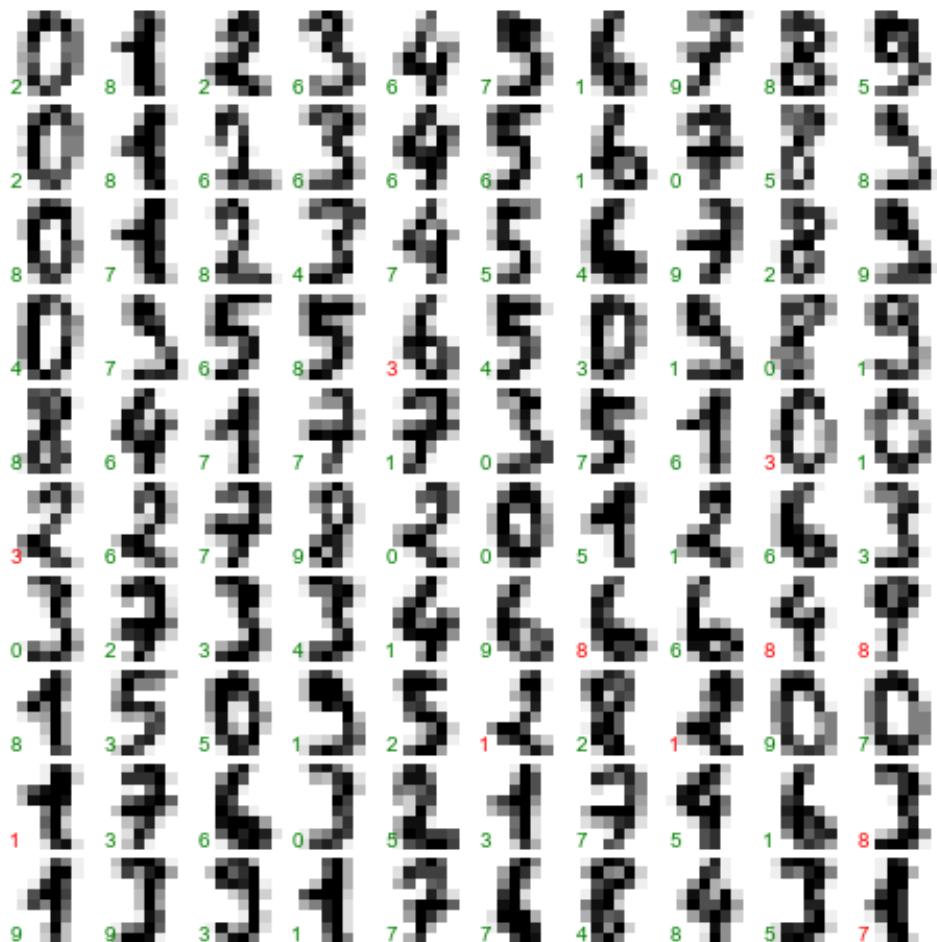
This shows us where the mis-labeled points tend to be: for example, a large number of twos here are mis-classified as either ones or eights. Another way to gain intuition into the characteristics of the model is to plot the inputs again, with their predicted labels. We'll use green for correct labels, and red for incorrect labels:

```

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')

```



Examining this subset of the data, we can gain insight regarding where the algorithm might be mis-performing. To go beyond our 80% classification rate, we might move to a more sophisticated algorithm such as random forests (Section X.X) or support vector machines (Section X.X), or perhaps a less-naive version of Bayesian classification (Section X.X).

In this section we have covered the essential features of the scikit-learn data representation, and the estimator API. Regardless of the type of estimator, the same import/instantiate/fit/predict pattern holds. Armed with this information about the estimator API, you can explore the scikit-learn documentation and begin to try-out various models on your data.

Armed with this information about scikit-learn's estimator API and assumed data representation, the possibilities are nearly limitless. In the next section we will explore

perhaps the most important topic in machine learning: how to select and validate your model.

Figure Code

The following code produces one of the figures used in the above text

```
import os
if not os.path.exists('fig'):
    os.makedirs('fig')
```

Features and Labels Grid

The following is the code generating the diagram showing the features matrix and target array.

```
fig = plt.figure(figsize=(6, 4))
ax = fig.add_axes([0, 0, 1, 1])
ax.axis('off')
ax.axis('equal')

# Draw features matrix
ax.vlines(range(6), ymin=0, ymax=9, lw=1)
ax.hlines(range(10), xmin=0, xmax=5, lw=1)
font_prop = dict(size=12, family='monospace')
ax.text(-1, -1, "Feature Matrix ($X$)", size=14)
ax.text(0.1, -0.3, r'n_features $\longrightarrow$ n_samples', **font_prop)
ax.text(-0.1, 0.1, r'$\longleftarrow$ n_samples', rotation=90,
        va='top', ha='right', **font_prop)

# Draw labels vector
ax.vlines(range(8, 10), ymin=0, ymax=9, lw=1)
ax.hlines(range(10), xmin=8, xmax=9, lw=1)
ax.text(7, -1, "Target Vector ($y$)", size=14)
ax.text(7.9, 0.1, r'$\longleftarrow$ n_samples', rotation=90,
        va='top', ha='right', **font_prop)

ax.set_ylim(10, -2)

fig.savefig('fig/07.02-samples-features.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

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 – are perhaps the most important part of using these tools and techniques effectively. In order to make an informed choice, we need a way of **validating** 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 hyperparameters, we can estimate how effective it is by applying it to some of the training data and comparing the prediction to the known value. A naive approach might look like this:

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:

```
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 classifier with `n_neighbors=1`. This is a very simple and intuitive model which says “the label of an unknown point is the same as the label of its closest training point”.

```
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:

```
model.fit(X, y)
y_model = model.predict(X)
```

Finally, we compute the fraction of correctly-labeled points:

```
from sklearn.metrics import accuracy_score
accuracy_score(y, y_model)
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 accurate? 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 fundamental flaw: **it trains and evaluates the model on the same data**. Furthermore, the nearest neighbor model is an *instance-based* estimator which simply stores the training data, and predicts labels by comparing new data to these stored points: except in contrived cases, it will get 100% accuracy *every time!*

Model Validation the Right Way: Holdout Sets

So what can be done? A better sense of a models performance can be found using what's known as a *holdout set*: that is, we hold-back some subset of the data from the training of the model, and then use this holdout set to check the model performance. This splitting can be done using the `train_test_split` utility in scikit-learn:

```
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)

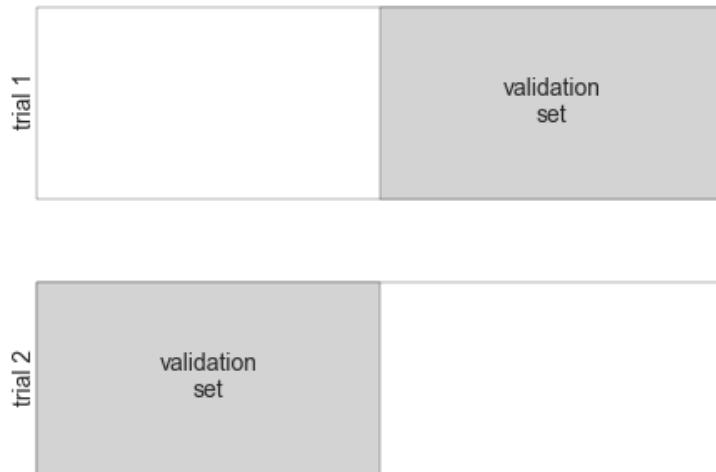
0.9066666666666662
```

We see here a more reasonable result: the nearest-neighbor classifier is about 90% accurate on this hold-out set.

Model Validation via Cross Validation

One disadvantage of the use of a holdout set for model validation is that we have lost a portion of our data to the model training. In the above case, half the dataset does not contribute to the training of the model! This is not optimal, and can cause problems 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 this:



Here we do two validation trials, alternately using each half of the data as a holdout set. Using the split data from above, we could implement it like this:

```
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)
(0.9599999999999996, 0.9066666666666662)
```

What comes out are two accuracy scores, which we could combine (by, say, taking the mean) to get a better measure of the global model performance. This particular form of cross-validation is a *2-fold cross-validation*, i.e. one in which we have split the data into 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: for example, here is a visual depiction of 5-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:

```
from sklearn.cross_validation import cross_val_score
cross_val_score(model, X, y, cv=5)

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 useful cross-validation schemes that are useful in particular situations; these are implemented via iterators in the `cross_validation` module. 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:

```
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., 1., 1., 1., 1., 1., 1.]
```

Because we have 150 samples, the leave one out cross-validation yields scores for 150 trials, 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:

```
scores.mean()  
0.9599999999999996
```

Other cross-validation schemes can be used similarly. For a description of what scikit-learn makes available, you can use IPython to explore the `sklearn.cross_validation` submodule, or take a look at scikit-learn's online [cross validation documentation](#).

Selecting the Best Model

Now that we've gone over 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 learning tutorials.

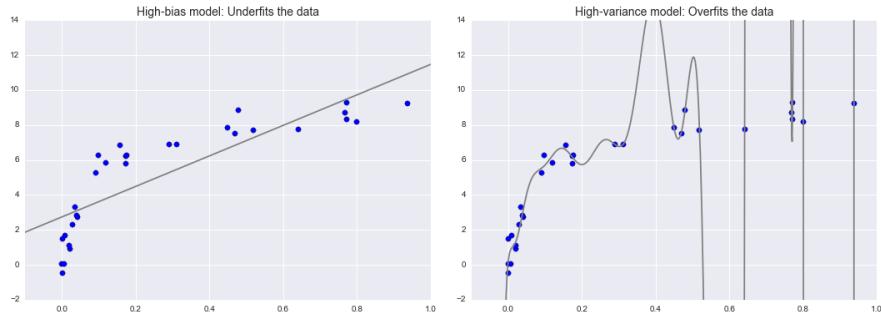
Of core importance is the following question: **if our estimator is underperforming, how should we move forward?** There are several possible answers:

- 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 counter-intuitive. In particular, sometimes using a more complicated model will give worse results, and adding more training samples may not improve your results! will 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 Tradeoff

Fundamentally, the question of “the best model” is about finding a sweet spot in the tradeoff between *bias* and *variance*. Consider the following two regression fits to the same dataset:

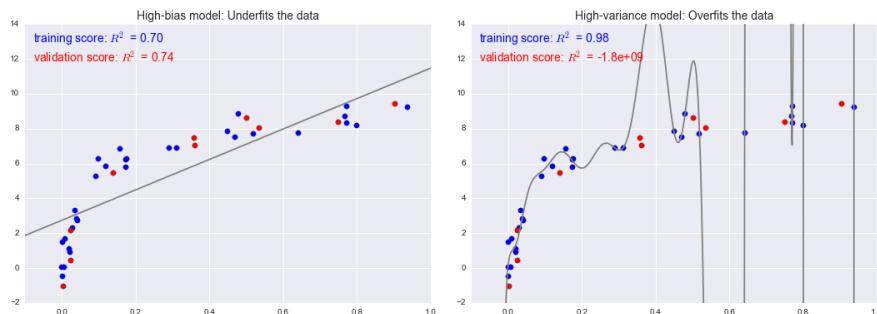


It is clear that neither of these models is a particularly good fit to the data, but they fail in different ways.

The model on the left attempts to find a straight-line fit through the data. Because the data are intrinsically more complicated than a straight line, such a 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 features 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 distribution; another way of saying this is that the model has high *variance*.

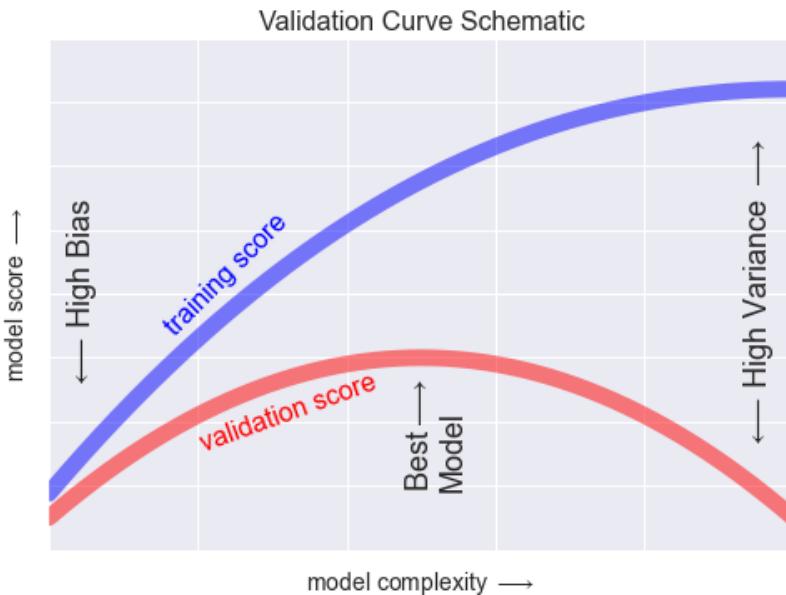
To look at this in another light, consider what happens if we use these two models to predict the y-value for some new data. In the following figures, the red points indicate data which is left-out of the training set:



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 associated 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 somewhat like this:



This diagram 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 under-fit, 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 over-fit, 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 tradeoff between bias and variance.

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 for a class of models. Here we will use a **Polynomial Regression** model: this is a general-

ized 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:

```
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:

```
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)
```

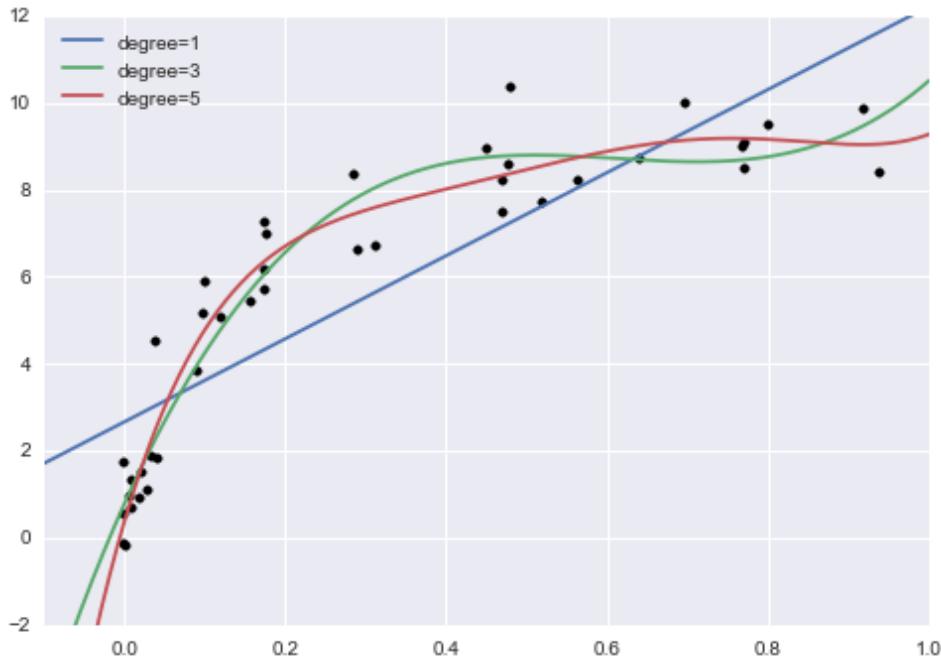
We can now visualize our data, along with polynomial fits of several degrees:

```
%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={}'.format(degree))
plt.xlim(-0.1, 1.0)
plt.ylim(-2, 12)
plt.legend(loc='best');
```

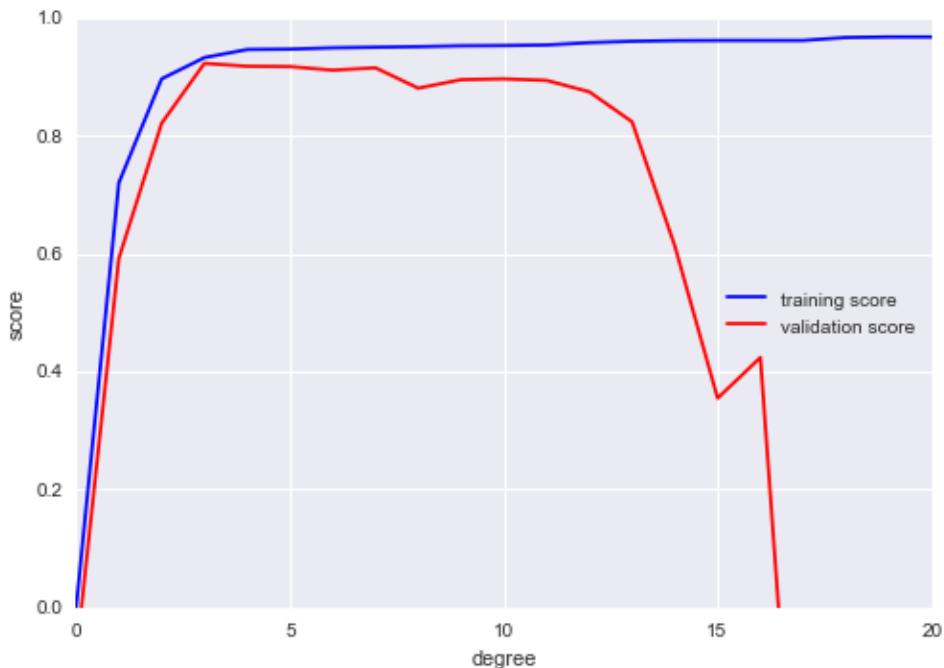
```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



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 tradeoff between bias (under-fitting) and variance (over-fitting)?

We can make progress in this by visualizing the validation curve for this particular data and model; this can be done straightforwardly using the `validation_curve` convenience routine provided by scikit-learn. Given a model, data, parameter name, and a range to explore, this function will automatically compute both the training score and validation score across the range:

```
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');
```

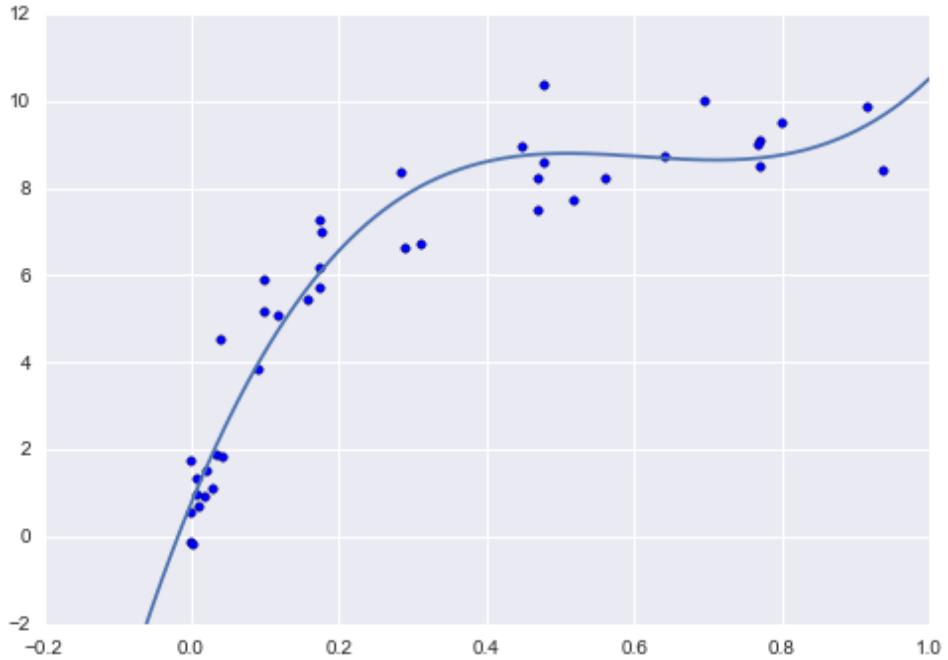


This shows precisely the qualitative behavior we expect: the training score is everywhere higher than the validation score; the training score is monotonically improving with increased model complexity; and the validation score reaches a maximum before dropping off as the model becomes over-fit.

From the validation curve, we can read-off that the optimal tradeoff between bias and variance is found for a 3rd-order polynomial; we can compute and display this fit over the original data as follows:

```
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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



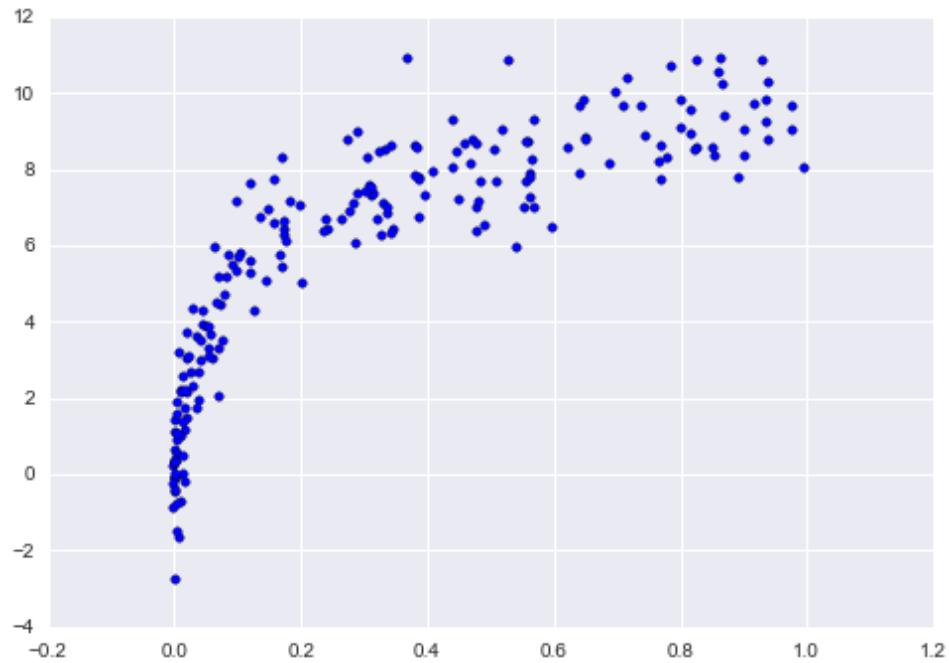
Notice that finding this optimal model did not actually require us to compute the training score, but examining the relationship between the training score and validation score can give us useful insight into the performance of the model.

Learning Curves

One important piece of the above discussion on 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 5 more points:

```
X2, y2 = make_data(200)
plt.scatter(X2.ravel(), y2);

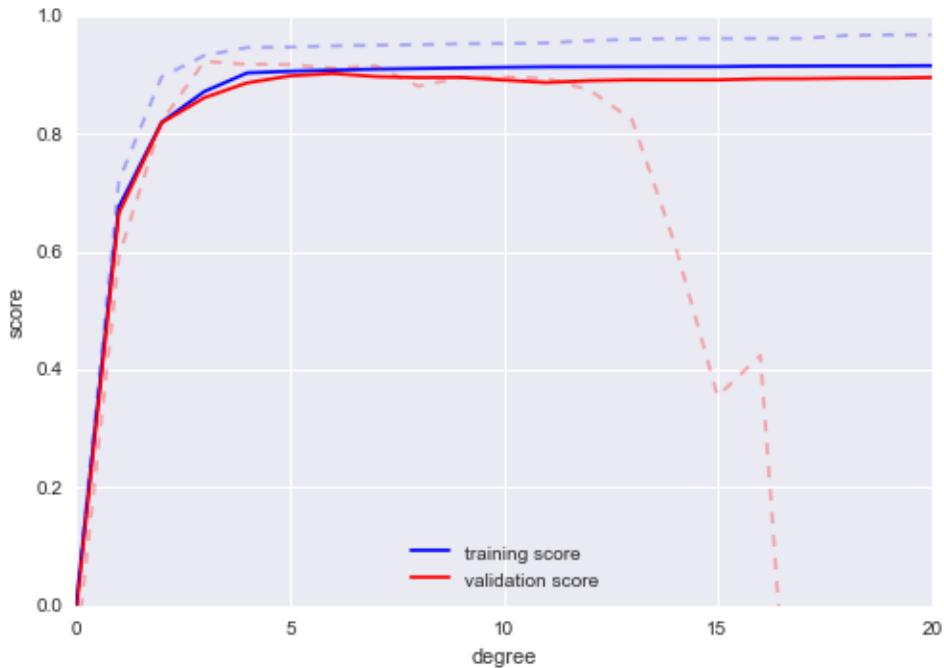
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



We will duplicate the code above to plot the validation curve for this larger dataset; for reference let's over-plot the previous results as well:

```
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');
```



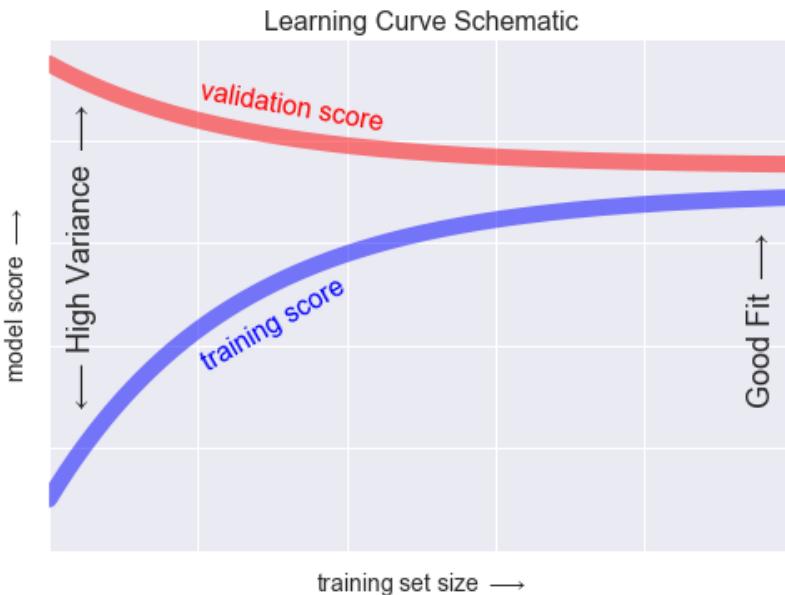
The solid lines show the new results, while the fainter dashed lines show the results of the previous smaller dataset. It is clear from the validation curve that the larger dataset can support a much more complicated model: the peak here is probably around a degree of 6, but even a degree-20 model is not seriously over-fitting the data – the validation 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 qualitatively like this:



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 points that 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 (often more complex) model.

Learning Curves in Scikit-Learn

Scikit-learn offers a convenient utility for computing such learning curves from your models; here we will compute a learning curve for our original dataset with a second-order polynomial model and a [TODO]:

```
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')
```

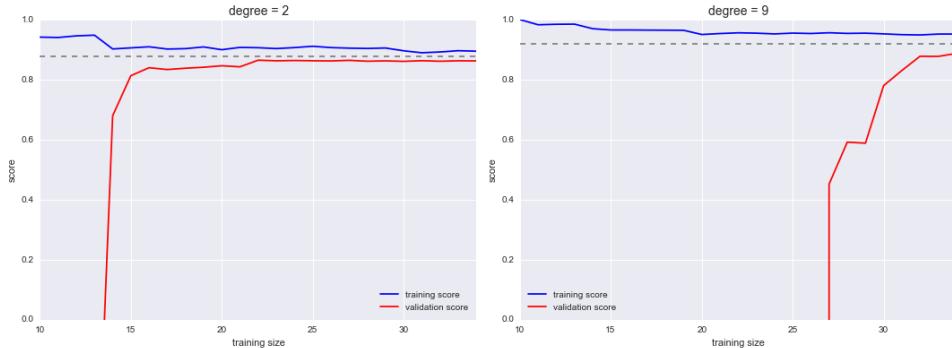
```

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 = {}'.format(degree), size=14)
ax[i].legend(loc='best')

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```



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 – when the training and validation curves are already close to each other – *adding more training data will not significantly improve the fit!* This situation 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 complicated) model. We see this in the right panel: by moving to a much more complicated model, we increase the score of convergence (indicated by the dashed line), but at 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 & dataset can help you to make this type of decision about how to move forward in improving your analysis.

Validation in Practice: Grid Search

The above discussion is meant to give you some intuition into the tradeoff 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 multi-dimensional surfaces. In these

cases, such visualizations are difficult and we would rather simply find the particular model which maximizes the validation score.

Scikit-learn provides automated tools to do this in the grid search module. Here is an 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 normalize the problem. This can be set up using scikit-learn's `GridSearchCV` meta-estimator:

```
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:

```
grid.fit(X, y);
```

Now that this is fit, we can ask for the best parameters as follows:

```
grid.best_params_

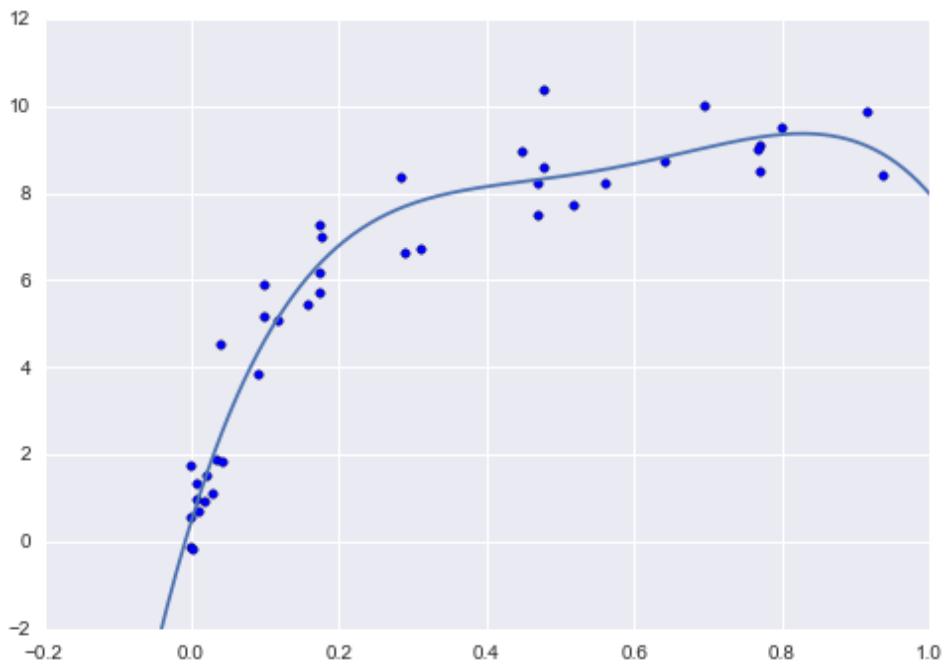
{'linearregression_fit_intercept': False,
 'linearregression_normalize': True,
 'polynomialfeatures_degree': 4}
```

Finally, if we wish, we can use the best model and show the fit to our data using code from above:

```
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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



The grid search provides many more options, including the ability to specify a custom scoring function, to parallelize the computations, to do randomized searches, and more. For information, see the examples in Section X.X, or refer to scikit-learn's [grid search documentation](#).

Summary

In this section we have begun to explore the concept of model validation and hyper-parameter optimization, focusing on intuitive aspects of the bias-variance tradeoff and how it comes into play when fitting models to data. In particular, we found that the use of a validation set or cross-validation approach is *vital* when tuning parameters in order to avoid over-fitting for more complex/flexible models.

In later sections, we will discuss the details of particularly useful models, and throughout will talk about what tuning is available for these models and how these free parameters affect model complexity. Keep the lessons of this section in mind as you read on and learn about these machine learning approaches!

Figure Code

Below is the code that produces some of the explanatory figures used above. The results are saved to file to be included in the flow of the above text at the appropriate place.

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

import os
if not os.path.exists('fig'):
    os.makedirs('fig')
```

Cross-Validation Figures

```
def draw_rects(N, ax, textprop={}):
    for i in range(N):
        ax.add_patch(plt.Rectangle((0, i), 5, 0.7, fc='white'))
        ax.add_patch(plt.Rectangle((5. * i / N, i), 5. / N, 0.7, fc='lightgray'))
        ax.text(5. * (i + 0.5) / N, i + 0.35,
                "validation\nset", ha='center', va='center', **textprop)
        ax.text(0, i + 0.35, "trial {0}".format(N - i),
                ha='right', va='center', rotation=90, **textprop)
    ax.set_xlim(-1, 6)
    ax.set_ylim(-0.2, N + 0.2)
```

2-Fold Cross-Validation.

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.axis('off')
draw_rects(2, ax, textprop=dict(size=14))

fig.savefig('fig/07.03-2-fold-CV.png')
plt.close(fig)
```

5-Fold Cross-Validation.

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
ax.axis('off')
draw_rects(5, ax, textprop=dict(size=10))

fig.savefig('fig/07.03-5-fold-CV.png')
plt.close(fig)
```

Overfitting and Underfitting

```
import numpy as np

def make_data(N=30, err=0.8, 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
```

```

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))

```

Bias-Variance Tradeoff.

```

X, y = make_data()
xfit = np.linspace(-0.1, 1.0, 1000)[:, None]
model1 = PolynomialRegression(1).fit(X, y)
model20 = PolynomialRegression(20).fit(X, y)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

ax[0].scatter(X.ravel(), y, s=40)
ax[0].plot(xfit.ravel(), model1.predict(xfit), color='gray')
ax[0].axis([-0.1, 1.0, -2, 14])
ax[0].set_title('High-bias model: Underfits the data', size=14)

ax[1].scatter(X.ravel(), y, s=40)
ax[1].plot(xfit.ravel(), model20.predict(xfit), color='gray')
ax[1].axis([-0.1, 1.0, -2, 14])
ax[1].set_title('High-variance model: Overfits the data', size=14)

fig.savefig('fig/07.03-bias-variance.png')
plt.close(fig)

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):

Bias-Variance Tradeoff Metrics.

```

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

X2, y2 = make_data(10, rseed=42)

ax[0].scatter(X.ravel(), y, s=40, c='blue')
ax[0].plot(xfit.ravel(), model1.predict(xfit), color='gray')
ax[0].axis([-0.1, 1.0, -2, 14])
ax[0].set_title('High-bias model: Underfits the data', size=14)
ax[0].scatter(X2.ravel(), y2, s=40, c='red')
ax[0].text(0.02, 0.98, "training score: $R^2$ = {:.2f}".format(model1.score(X, y)),
           ha='left', va='top', transform=ax[0].transAxes, size=14, color='blue')
ax[0].text(0.02, 0.91, "validation score: $R^2$ = {:.2f}".format(model1.score(X2, y2)),
           ha='left', va='top', transform=ax[0].transAxes, size=14, color='red')

ax[1].scatter(X.ravel(), y, s=40, c='blue')
ax[1].plot(xfit.ravel(), model20.predict(xfit), color='gray')
ax[1].axis([-0.1, 1.0, -2, 14])

```

```

ax[1].set_title('High-variance model: Overfits the data', size=14)
ax[1].scatter(X2.ravel(), y2, s=40, c='red')
ax[1].text(0.02, 0.98, "training score: $R^2\$ = {0:.2g}".format(model20.score(X, y)),
           ha='left', va='top', transform=ax[1].transAxes, size=14, color='blue')
ax[1].text(0.02, 0.91, "validation score: $R^2\$ = {0:.2g}".format(model20.score(X2, y2)),
           ha='left', va='top', transform=ax[1].transAxes, size=14, color='red')

fig.savefig('fig/07.03-bias-variance-2.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```

Validation Curve.

```

x = np.linspace(0, 1, 1000)
y1 = -(x - 0.5) ** 2
y2 = y1 - 0.33 + np.exp(x - 1)

fig, ax = plt.subplots()
ax.plot(x, y2, lw=10, alpha=0.5, color='blue')
ax.plot(x, y1, lw=10, alpha=0.5, color='red')

ax.text(0.15, 0.2, "training score", rotation=45, size=16, color='blue')
ax.text(0.2, -0.05, "validation score", rotation=20, size=16, color='red')

ax.text(0.02, 0.1, r'$\longleftarrow$ High Bias', size=18, rotation=90, va='center')
ax.text(0.98, 0.1, r'$\longleftarrow$ High Variance $\rightarrow$', size=18, rotation=90, ha='left')
ax.text(0.48, -0.12, 'Best$\rightarrow$\nModel', size=18, rotation=90, va='center')

ax.set_xlim(0, 1)
ax.set_ylim(-0.3, 0.5)

ax.set_xlabel(r'model complexity $\rightarrow$', size=14)
ax.set_ylabel(r'model score $\rightarrow$', size=14)

ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.yaxis.set_major_formatter(plt.NullFormatter())

ax.set_title("Validation Curve Schematic", size=16)

fig.savefig('fig/07.03-validation-curve.png')
plt.close(fig)

```

Learning Curve.

```

N = np.linspace(0, 1, 1000)
y1 = 0.75 + 0.2 * np.exp(-4 * N)
y2 = 0.7 - 0.6 * np.exp(-4 * N)

fig, ax = plt.subplots()
ax.plot(x, y2, lw=10, alpha=0.5, color='blue')
ax.plot(x, y1, lw=10, alpha=0.5, color='red')

```

```

ax.text(0.2, 0.5, "training score", rotation=30, size=16, color='blue')
ax.text(0.2, 0.88, "validation score", rotation=-10, size=16, color='red')

ax.text(0.98, 0.45, r'Good Fit $\rightarrow$', size=18, rotation=90, ha='right', va='center')
ax.text(0.02, 0.57, r'$\leftarrow$ High Variance $\rightarrow$', size=18, rotation=90, va=)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

ax.set_xlabel(r'training set size $\rightarrow$', size=14)
ax.set_ylabel(r'model score $\rightarrow$', size=14)

ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.yaxis.set_major_formatter(plt.NullFormatter())

ax.set_title("Learning Curve Schematic", size=16)

fig.savefig('fig/07.03-learning-curve.png')
plt.close(fig)

```

In Depth: Naive Bayes Classification

The previous four sections have given a general overview of the concepts of machine learning. In this section and the ones that follow, we will be taking a closer look at several specific algorithms for supervised and unsupervised learning, starting here with Naive Bayes Classification.

Naive Bayes models are a group of extremely fast and simple classification algorithms which are often suitable for very high-dimensional datasets. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem. This section will focus on an intuitive explanation of how Naive Bayes classifiers work, followed by a couple examples of them in action on some datasets.

Bayesian Classification

Naive Bayes classifiers are built on Bayesian classification methods. These rely on Bayes' theorem, which we discussed in Section X.X. The posterior probability of a label L given some observed features can be written

$$P(L \mid \text{features}) = \frac{P(\text{features} \mid L)P(L)}{P(\text{features})}$$

This expression is the *posterior probability* for the given label. If we are trying to decide between two labels – let's call them L_1 and L_2 – then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$\frac{P(L_1 \mid \text{features})}{P(L_2 \mid \text{features})} = \frac{P(\text{features} \mid L_1)P(L_1)}{P(\text{features} \mid L_2)P(L_2)}$$

All we need now is some model by which we can compute $P(\text{features} \mid L_i)$ for each label. Such a model is called a *generative model* because it specifies the hypothetical random process which generates the data. Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier. The general version of such a training step is a very difficult task, but we can make it simpler through the use of some simplifying assumptions about the form of this model.

This is where the “naive” in “naive Bayes” comes in: if we make very naive assumptions about the generating model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification. Different types of Naive Bayes classifiers rest on different naive assumptions about the data, and we will examine a few of these in the following sections.

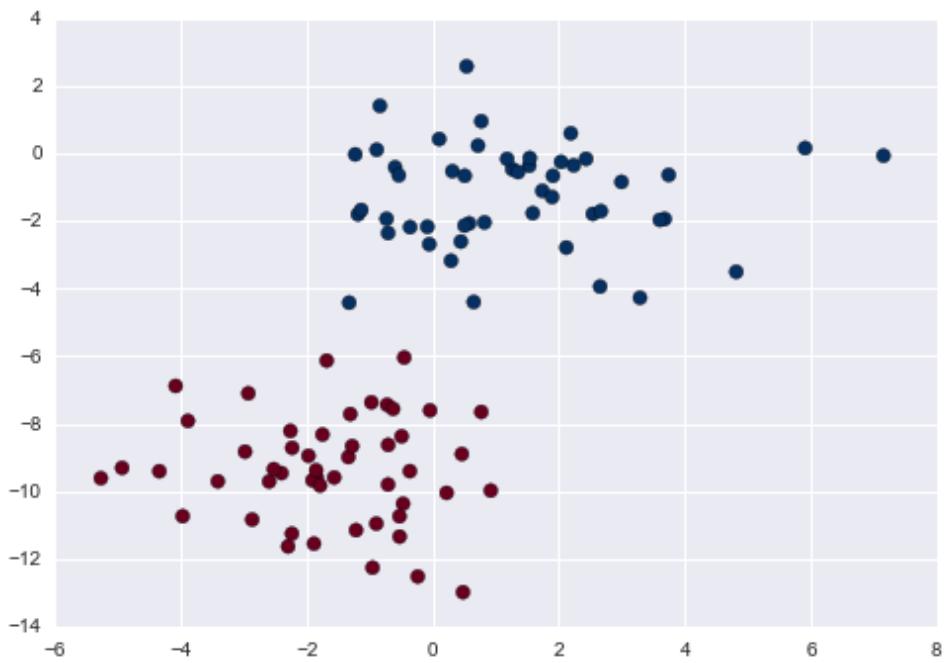
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Gaussian Naive Bayes

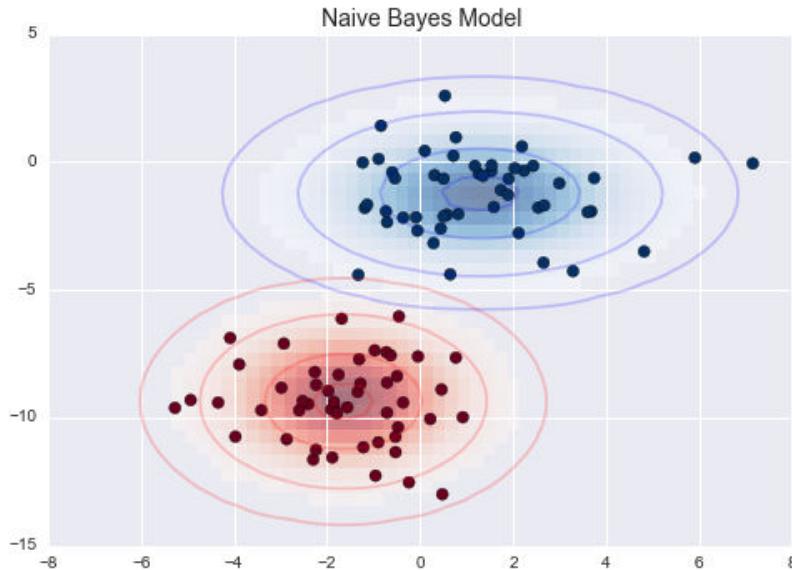
Perhaps the easiest Naive Bayes classifier to understand is Gaussian Naive Bayes. In this classifier, the assumption is that **data from each label is drawn from a simple Gaussian distribution**. Pictorially, imagine that you have the following data:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



One extremely fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by simply finding the mean and standard deviation of the points within each label, which is all you need to define such a distribution. The result of this naive Gaussian assumption is shown in the following panel:



The colored ellipses here represent the Gaussian generative model for each label, with larger probability toward the center of the ellipses. With this generative model in place for each class, we have a simple recipe to compute the likelihood $P(\text{features} \mid L_1)$ for any data point, and thus we can quickly compute the posterior ratio and determine which label is the most probable for a given point.

Using scikit-learn, this procedure is implemented in the `sklearn.naive_bayes.GaussianNB` estimator:

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y)

GaussianNB()
```

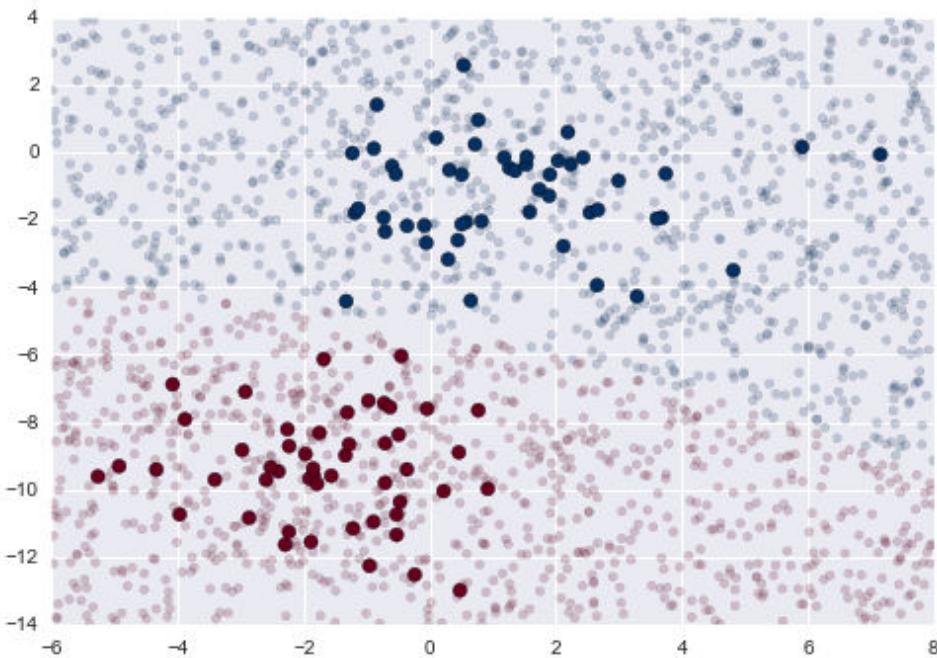
Now let's generate some new data and predict the label:

```
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
ynew = model.predict(Xnew)
```

Now we can plot this new data to get an idea of where the decision boundary is:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.2)
plt.axis(lim);
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



We see a slightly curved boundary in the classifications – in general the boundary in Gaussian Naive Bayes is quadratic.

A nice piece of this Bayesian formalism is that it naturally allows for probabilistic classification, which we can compute using the `predict_proba` method:

```
yprob = model.predict_proba(Xnew)  
yprob[-8: ].round(2)  
  
array([[ 0.89,  0.11],  
       [ 1.  ,  0.  ],  
       [ 1.  ,  0.  ],  
       [ 1.  ,  0.  ],  
       [ 1.  ,  0.  ],  
       [ 1.  ,  0.  ],  
       [ 0.  ,  1.  ],  
       [ 0.15,  0.85]])
```

The columns give the posterior probabilities of the first and second label, respectively. If you are looking for uncertainties in your classification, Bayesian approaches like this one are a natural fit.

Of course, the final classification will only be as good as the model assumptions that lead to it, which is why Gaussian naive Bayes often does not produce very good

results. Still, in many cases – especially as the number of features grows – this assumption is not detrimental enough to prevent Gaussian Naive Bayes from being a useful method.

Multinomial Naive Bayes

The Gaussian assumption above is by no means the only simple assumption that could be used to specify the generating distribution for each label. Another useful example is Multinomial Naive Bayes, where the features are assumed to be generated from a simple multinomial distribution. The multinomial distribution describes the probability of observing counts among a number of categories, and thus Multinomial Naive Bayes is most appropriate for features which represent counts or count rates.

The idea is precisely the same as above, except that instead of modeling the data distribution with the best-fit Gaussian, we model the data distribution with a best-fit Multinomial distribution.

Example: Classifying Text

One place where multinomial Naive Bayes is often used is in text classification, where the features are related to word counts or frequencies within the documents to be classified. We discussed the extraction of such features from text in Section X.X; here we will use the sparse word count features from the 20 newsgroups corpus to show how we might classify these short documents into categories.

Let's download the data and take a look at the target names:

```
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups()
data.target_names

['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
```

```
'talk.politics.misc',
'talk.religion.misc']
```

For simplicity here, we will select just a few of these categories, and download the training and testing set:

```
categories = ['talk.religion.misc', 'soc.religion.christian', 'sci.space', 'comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)
```

Here is a representative entry from the data:

```
print(train.data[5])

From: dmcmc@uluhe.soest.hawaii.edu (Don McGee)
Subject: Federal Hearing
Originator: dmcmc@uluhe
Organization: School of Ocean and Earth Science and Technology
Distribution: usa
Lines: 10
```

Fact or rumor....? Madalyn Murray O'Hare an atheist who eliminated the use of the bible reading and prayer in public schools 15 years ago is now going to appear before the FCC with a petition to stop the reading of the Gospel on the airways of America. And she is also campaigning to remove Christmas programs, songs, etc from the public schools. If it is true then mail to Federal Communications Commission 1919 H Street Washington DC 20054 expressing your opposition to her request. Reference Petition number

2493.

In order to use this data for machine learning, we need to be able to convert the content of each string into a vector of numbers. For this we will use the TFIDF vectorizer (discussed in Section X.X), and create a pipeline which attaches it to a multinomial naive Bayes classifier:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

With this pipeline, we can apply the model to the training data, and predict labels for the test data:

```
model.fit(train.data, train.target)
labels = model.predict(test.data)
```

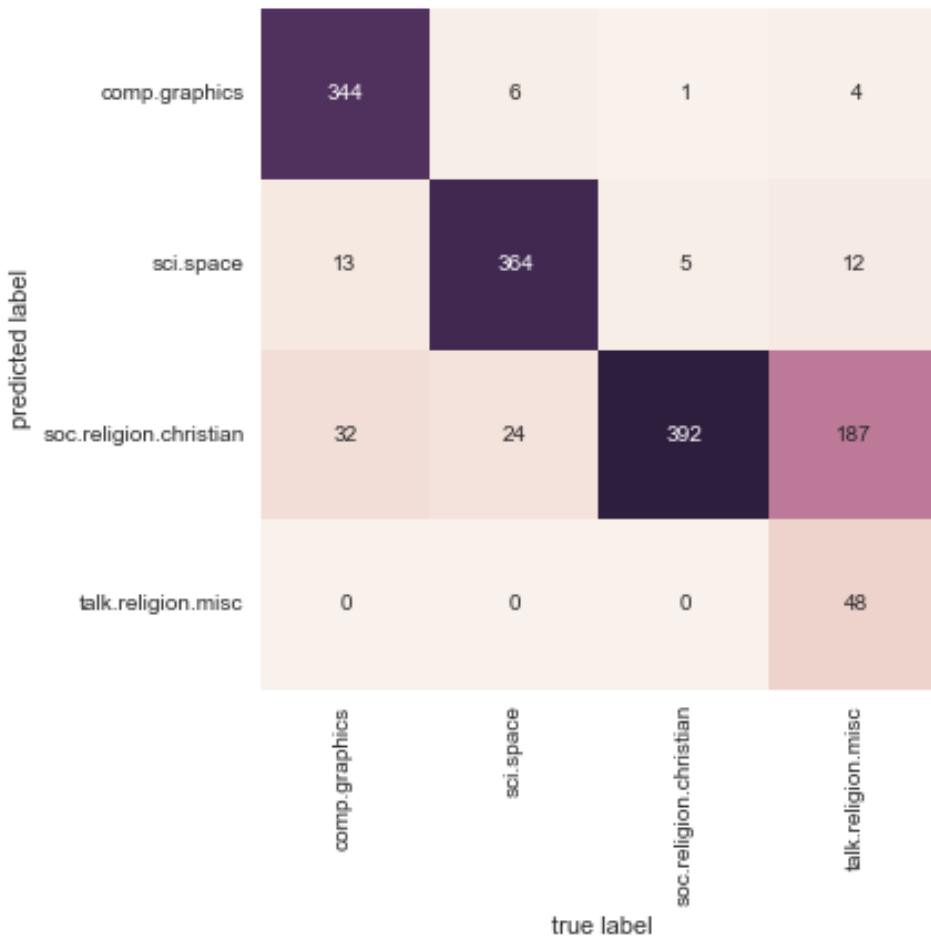
Now that we have predicted the labels for the test data, we can evaluate them to learn about the performance of the estimator. For example, here is the confusion matrix between the true and predicted labels for the test data:

```

from sklearn.metrics import confusion_matrix
mat = confusion_matrix(test.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=train.target_names, yticklabels=train.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



Evidently, even this very simple classifier can successfully separate space talk from computer talk, but it gets confused between talk about religion and talk about Christianity. This is perhaps an expected area of confusion!

The very cool thing here is that we now have the tools to determine the category for *any* string, using the `predict()` method of this pipeline. Here's a quick utility function that will return the prediction for a single string:

```
def predict_category(s, train=train, model=model):
    return train.target_names[model.predict([s])]
```

Let's try it out:

```
predict_category('sending a payload to the ISS')
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/IPython/kernel/_main_.py:2: D
  from ipykernel import kernelapp as app
```

```
'sci.space'
```

```
predict_category('discussing islam vs atheism')
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/IPython/kernel/_main_.py:2: D
  from ipykernel import kernelapp as app
```

```
'soc.religion.christian'
```

```
predict_category('determining the screen resolution')
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/IPython/kernel/_main_.py:2: D
  from ipykernel import kernelapp as app
```

```
'comp.graphics'
```

Remember that this is nothing more sophisticated than a simple probability model for the frequency of each word in the string; nevertheless, the result is striking. Even a very naive algorithm, when used carefully and trained on a large set of high-dimensional data, can be surprisingly effective.

When to Use Naive Bayes

Because naive Bayesian classifiers make such stringent assumptions about data, they will generally not perform as well as a more complicated model. That said, they have several advantages:

- they are extremely fast for both training and prediction
- they provide straightforward probabilistic prediction
- they are often very easily interpretable
- they have have very few (if any) tunable parameters

These advantages mean a naive Bayesian classifier is often a good choice as an initial baseline classification. If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem. If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

Naive Bayes classifiers tend to perform especially well in one of the following situations:

- when the naive assumptions acutally match the data (very rare in practice)
- for very well-separated categories, when model precision is less important
- for very high-dimensional data, when model precision is less important

The last two points seem distinct, but they actually are related: as the dimension of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in *every single dimension* to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information. For this reason, simplistic classifiers like naive Bayes tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

Figures

The following code generates figures used in the above text.

```
import os
if not os.path.exists('fig'):
    os.makedirs('fig')
```

Gaussian Naive Bayes

```
from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)

fig, ax = plt.subplots()

ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
ax.set_title('Naive Bayes Model', size=14)

xlim = (-8, 8)
ylim = (-15, 5)
```

```

xg = np.linspace(xlim[0], xlim[1], 60)
yg = np.linspace(ylim[0], ylim[1], 40)
xx, yy = np.meshgrid(xg, yg)
Xgrid = np.vstack([xx.ravel(), yy.ravel()]).T

for label, color in enumerate(['red', 'blue']):
    mask = (y == label)
    mu, std = X[mask].mean(0), X[mask].std(0)
    P = np.exp(-0.5 * (Xgrid - mu) ** 2 / std ** 2).prod(1)
    Pm = np.ma.masked_array(P, P < 0.03)
    ax.pcolorfast(xg, yg, Pm.reshape(xx.shape), alpha=0.5,
                  cmap=color.title() + 's')
    ax.contour(xx, yy, P.reshape(xx.shape),
               levels=[0.01, 0.1, 0.5, 0.9],
               colors=color, alpha=0.2)

ax.set(xlim=xlim, ylim=ylim)

fig.savefig('fig/07.05-gaussian-NB.png')
plt.close(fig)

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):

In Depth: Linear Regression

Just as Naive Bayes (Section X.X) is a good starting point for classification tasks, linear regression models are a 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 simplest form of a linear regression model – fitting a straight line to data – but the 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 seeing how before moving on to see how linear models can be generalized to account for more complicated patterns in data.

```
%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

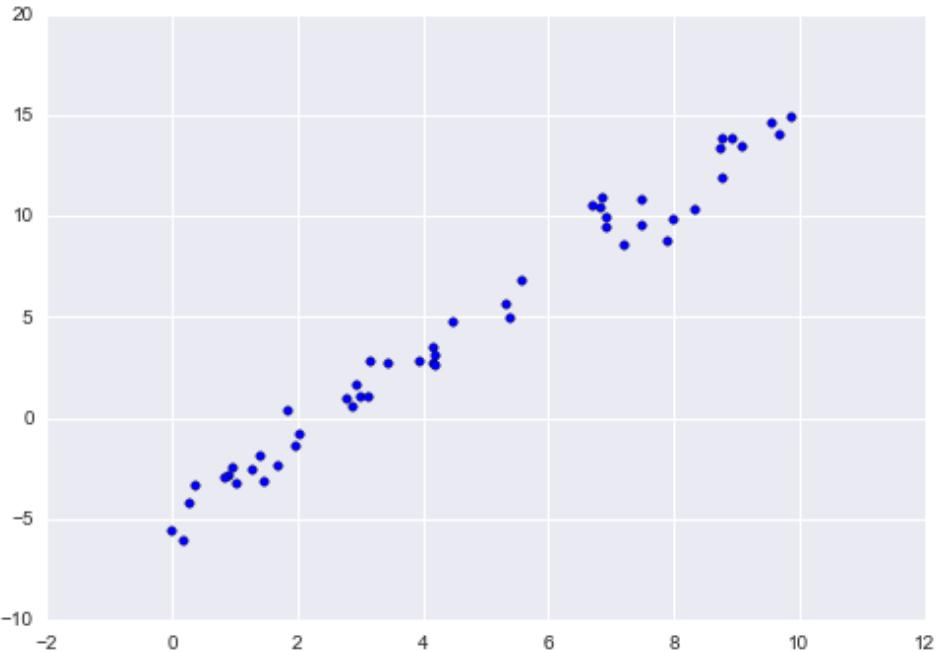
$$y = ax + b$$

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:

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = 2 * x - 5 + rng.randn(50)
plt.scatter(x, y);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



We can use scikit-learn's `LinearRegression` estimator to fit this data and construct the best-fit line:

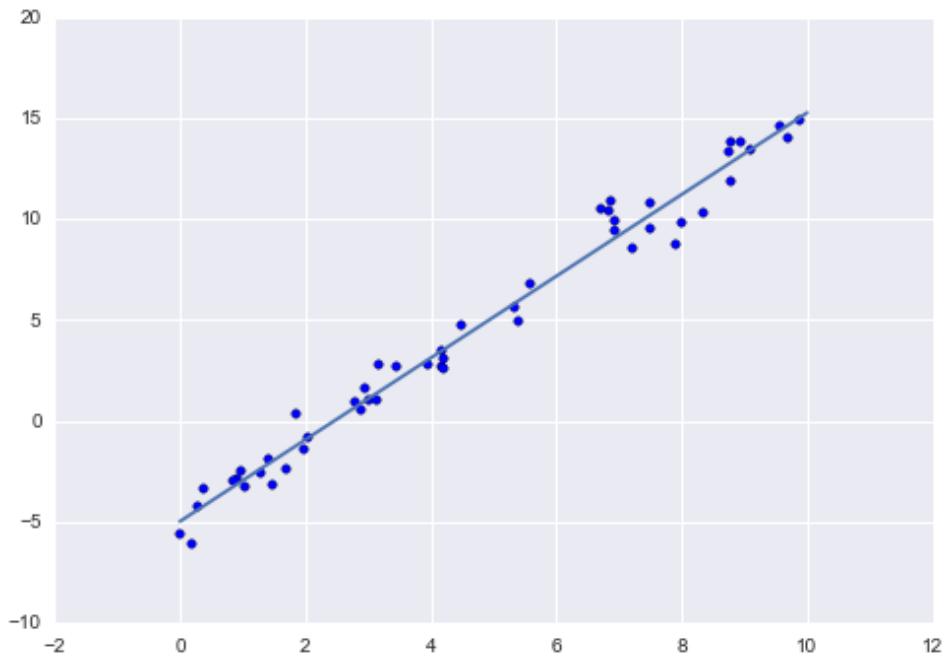
```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)

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);
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



The slope and intercept of the data are contained in the model's fit parameters, which in scikit-learn are always marked by a trailing underscore. Here the relevant parameters are `coef_` and `intercept_`:

```
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 addition to simple straight-line fits, it can also handle multi-dimensional linear models of the form

$$y = a_0 + a_1 x_1 + a_2 x_2 + \dots$$

where there are multiple x values. Geometrically, this is like fitting a plane to points in three dimensions, or fitting a hyper-plane to points in higher dimensions.

The multi-dimensional nature of such regressions make them more difficult to visualize, but we can see one of these fits in action by building some data

```

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 non-linear relationships between variables is to transform the data according to *basis functions*. We have seen one version of this before, in the `PolynomialRegression` pipeline used in Section X.X. The idea is to take our multi-dimensional linear model:

$$y = a_0 + a_1x_1 + a_2x_2 + a_3x_3 + \dots$$

and build the x_1, x_2, x_3 , etc. from our single-dimensional input x . That is, we let $x_n = f_n(x)$, where $f_n()$ is some function which transforms our data.

For example, if $f_n(x) = x^n$, our model becomes a polynomial regression:

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots$$

Notice that this is *still a linear model* – the linearity refers to the fact that the coefficients 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-in to scikit-learn, using the `PolynomialFeatures` transformer:

```

from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])

```

```
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. The cleanest way to accomplish this is to use a pipeline. Let's make a 7th-degree polynomial model in this way:

```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

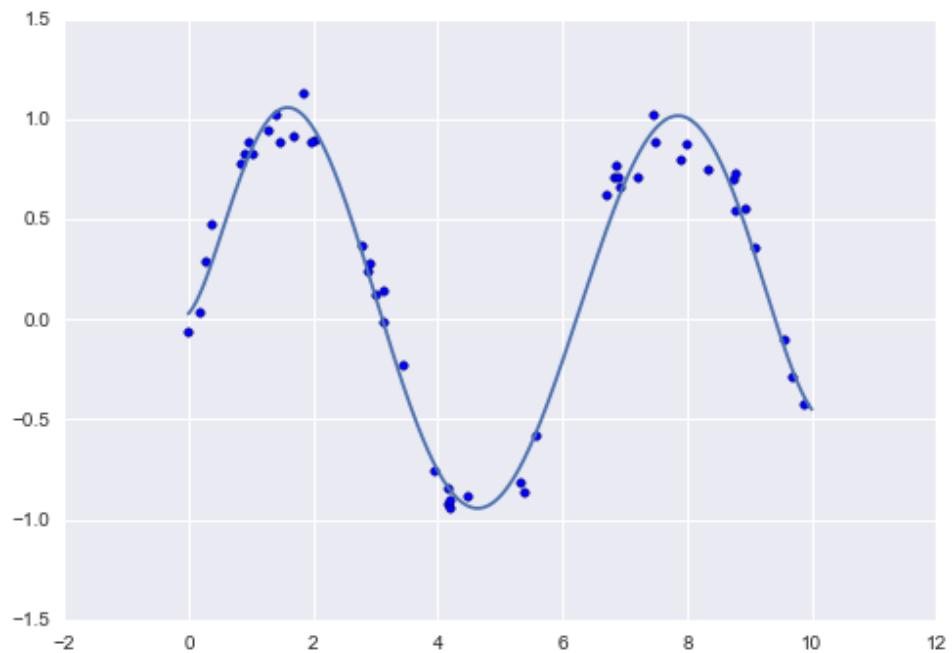
With this transform in place, we can use the linear model to fit much more complicated relationships between x and y . For example, here is a sine wave with noise:

```
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);

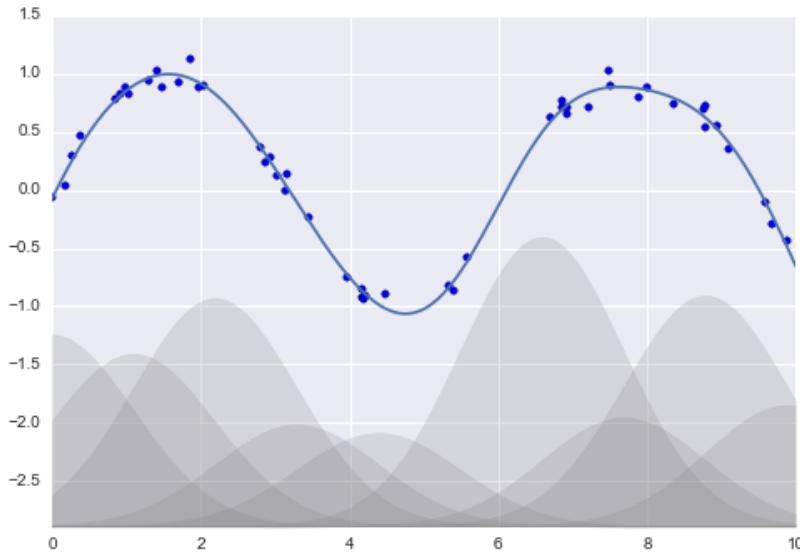
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
  if self._edgecolors == str('face'):
```



Our linear model, through the use of 7th-order polynomial basis functions, can provide an excellent fit to this non-linear data!

Gaussian Basis Functions

Of course, other basis functions are possible. For example, one useful pattern is to fit a model which is not a sum of polynomial bases, but a sum of Gaussian bases. The result might look something like this:



The shaded regions in the above plot are the scaled basis functions, and when added together they reproduce the smooth curve through the data. These Gaussian basis functions are not built-in to scikit-learn, but we can write a custom transformer which will create them:

```
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly-spaced Gaussian Features for 1D 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_,
```

```

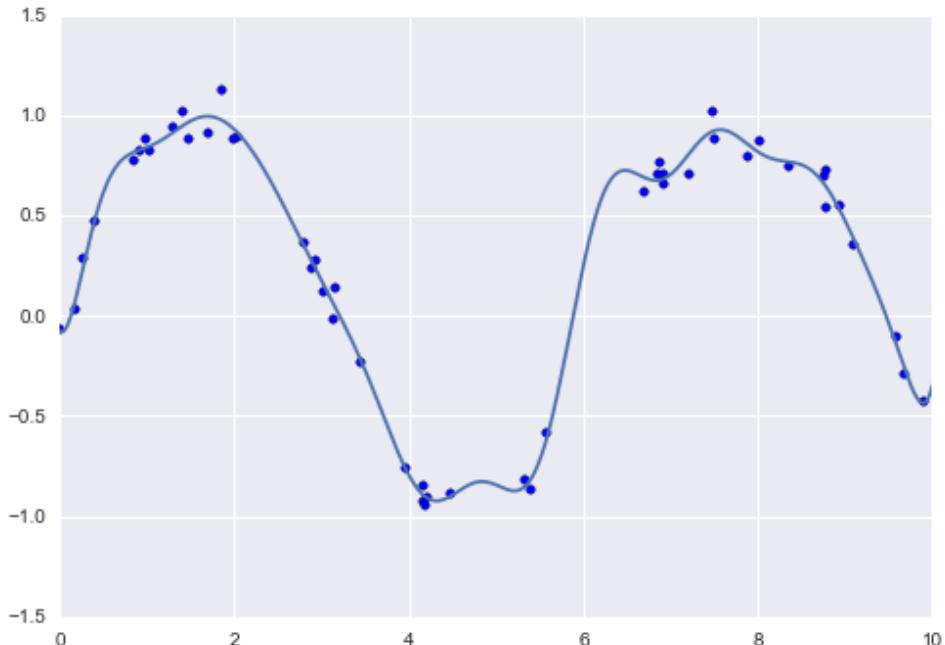
        self.width_, axis=1)

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);

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):



We put this example here just to make clear that there is nothing magic about polynomial 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 over-fitting (see Section X.X). For example, if we choose too many Gaussian basis functions, we end up with results that don't look so good:

```

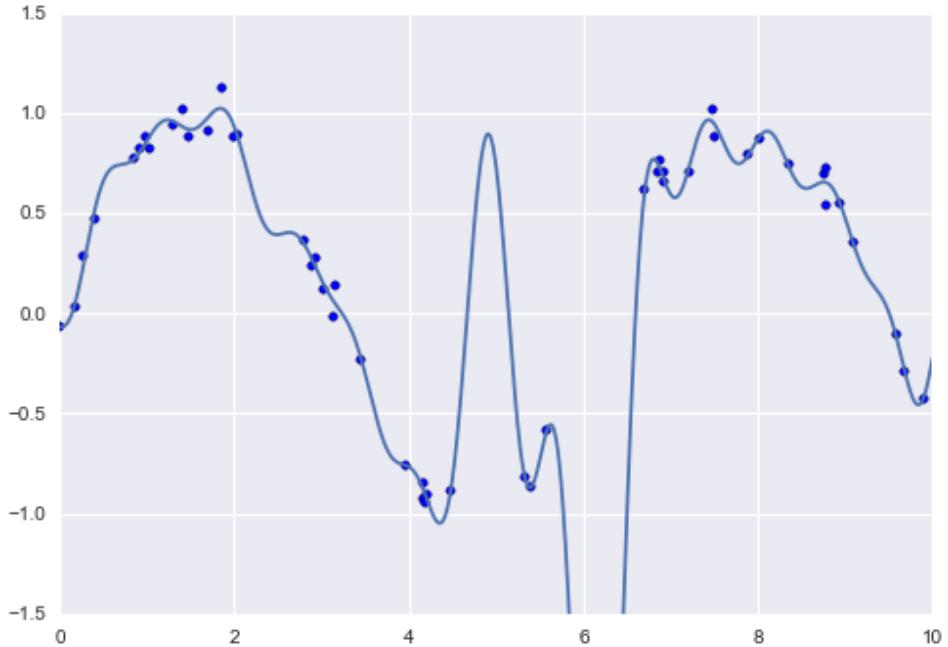
model = make_pipeline(GaussianFeatures(30),
                      LinearRegression())
model.fit(x[:, np.newaxis], y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
 if self._edgecolors == str('face'):



With the data projected to the 30-dimensional basis, the model has far too much flexibility 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:

```

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)

```

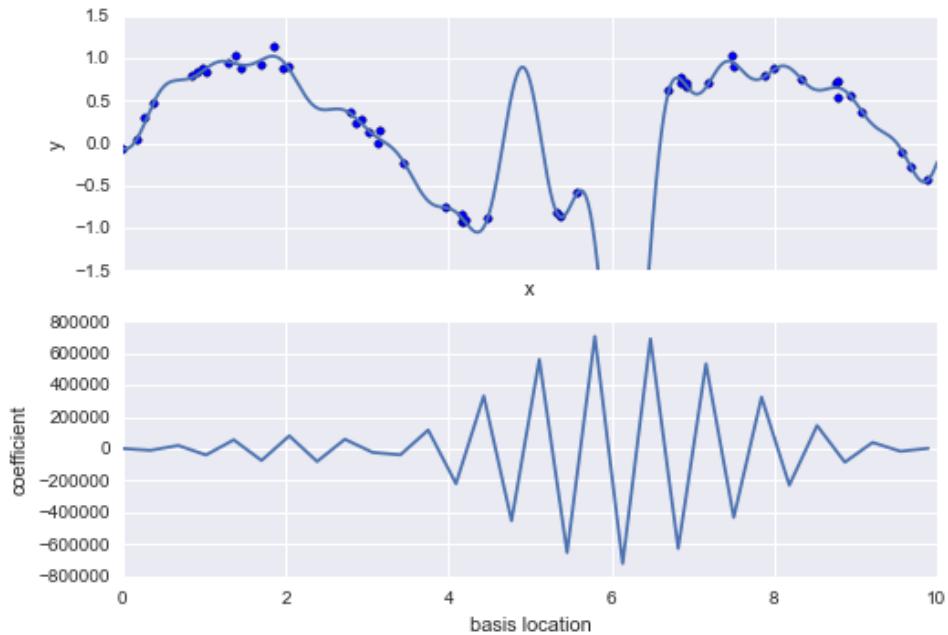
```

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)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```



This is typical over-fitting behavior when basis functions overlap: the coefficients 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 (L_2) Regularization

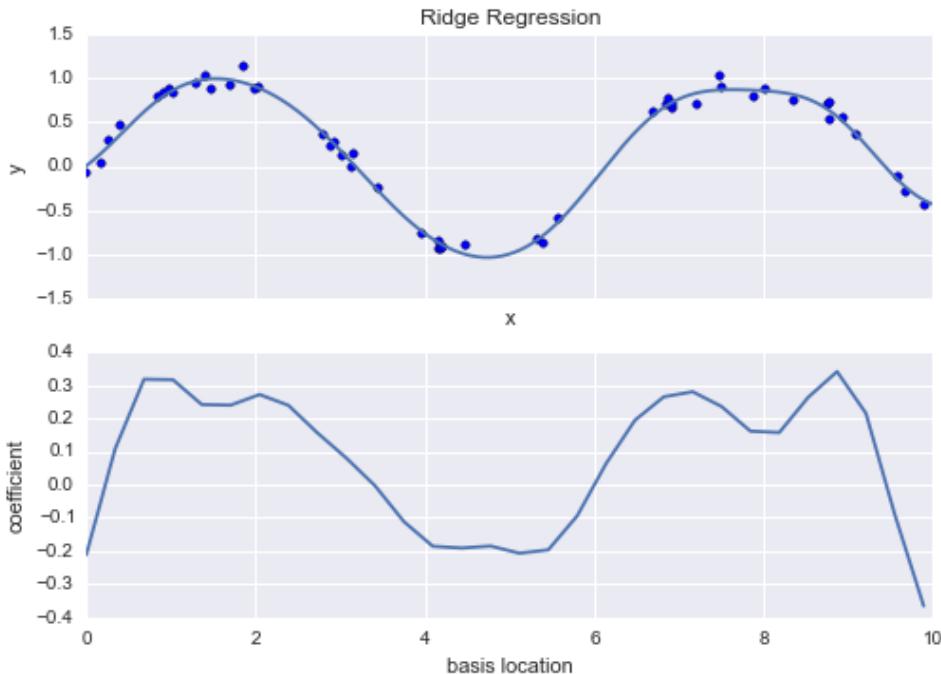
Perhaps the most common form of regularization is known as *Ridge regression* or L_2 regularization, sometimes also called *Tikhonov regularization*. This proceeds by penalizing the sum of squares (2-norms) of the model coefficients; in this case, the penalty on the model fit would be

$$P = \alpha \sum_{n=1}^N \theta_n^2$$

where α is a free parameter that controls the strength of the penalty. This type of penalized model is built-in to scikit-learn with the Ridge estimator:

```
from sklearn.linear_model import Ridge
model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
basis_plot(model, title='Ridge Regression')

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):
```



The α parameter is essentially a knob controlling the complexity of the resulting model. In the limit $\alpha \rightarrow 0$, we recover the standard linear regression result; in the limit $\alpha \rightarrow \infty$, all model response will be suppressed. One advantage of ridge regression in particular is that it can be computed very efficiently – at hardly more computational cost than the original linear regression model.

Lasso (L_1) Regularization

Another very common type of regularization is known as Lasso, and involves penalizing the sum of absolute values (1-norms) of regression coefficients:

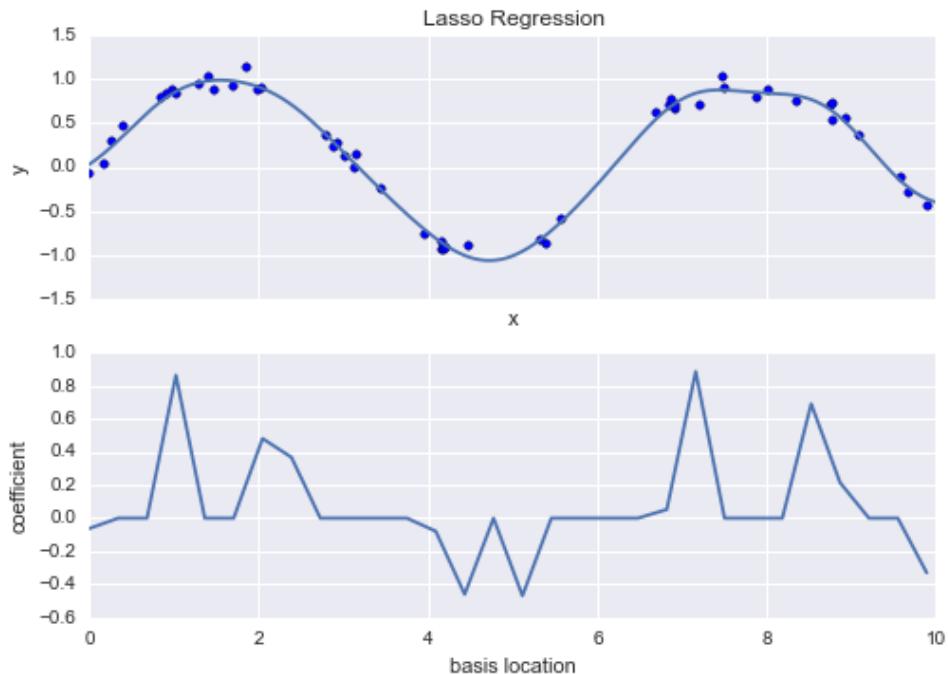
$$P = \alpha \sum_{n=1}^N |\theta_n|$$

Though this is conceptually very similar to Ridge regression, the results can differ surprisingly: for example, due to geometric reasons lasso regression tends to favor *sparse models* where possible: that is, it preferentially sets model coefficients to exactly zero.

We can see this behavior in duplicating the above plot, but using L1-normalized coefficients:

```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
basis_plot(model, title='Lasso Regression')

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/sklearn/linear_model/coordinate_descent.py:217: ConvergenceWarning:
  /Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



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 the ridge regularization above, the α parameter tunes the strength of the penalty, and should be determined via, e.g. cross-validation (Section X.X).

Example: Predicting Bicycle Traffic

As an example, let's take a look at whether we can predict the number of bicycle trips across Seattle's Fremont bridge based on weather, season, and other effects. We have seen this data already in Section X.X.

In this section, we will join the bike data with another dataset, and try to determine the extent to which weather and seasonal effects – 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 statistical 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.

[TODO: provide direct data download?]

Let's start by loading the two datasets, indexing by date:

```
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:

```
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 which indicate the day of the week:

```
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
for i in range(7):
    daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

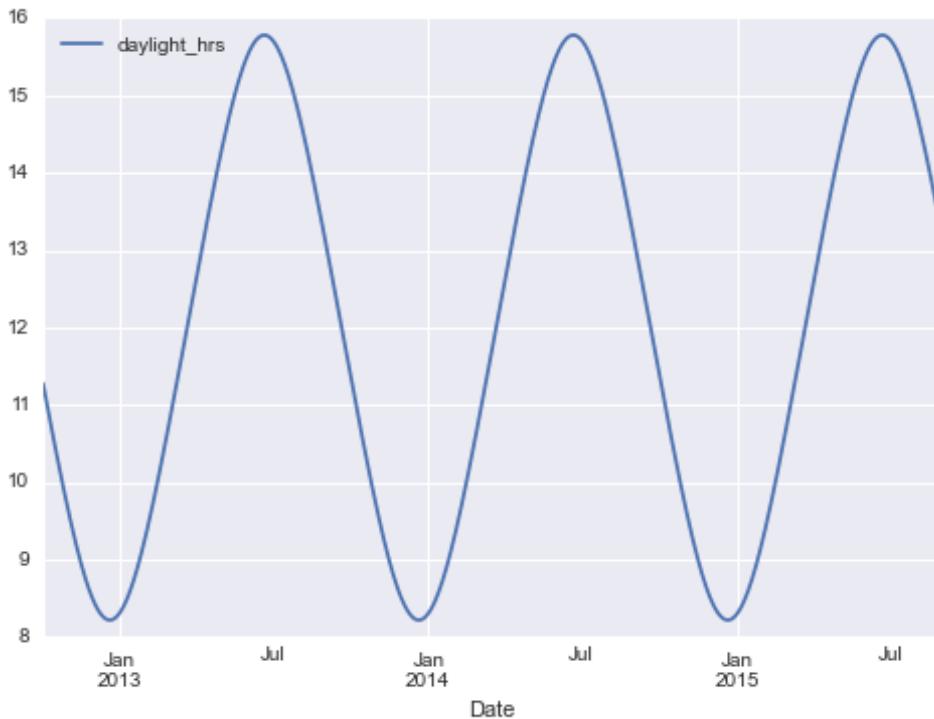
Similarly, we might expect riders to behave differently on holidays; let's add an indicator of this as well:

```
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:

```
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();
```



We can also add the average temperature and total precipitation to the data. In addition to the inches of precipitation, let's add a flag which indicates whether a day is dry (has zero precipitation).

```
# 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 which 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:

```

daily['annual'] = (daily.index - daily.index[0]).days / 365.

```

Now our data is in order, and we can take a look at it:

```

daily.head()

```

	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun	holiday	daylight_hrs	PRCP	Temp (C)	dry day	annual
Date														
2012-10-03	3521	0	0	1	0	0	0	0	0	11.277359	0	13.35	1	0.000000
2012-10-04	3475	0	0	0	1	0	0	0	0	11.219142	0	13.60	1	0.002740
2012-10-05	3148	0	0	0	0	1	0	0	0	11.161038	0	15.30	1	0.005479
2012-10-06	2006	0	0	0	0	0	1	0	0	11.103056	0	15.85	1	0.008219
2012-10-07	2142	0	0	0	0	0	0	1	0	11.045208	0	15.85	1	0.010959

With this in place, we can choose the columns to use, and fit a linear regression model to our data. We will set `fit_intercept = False`, because the daily flags essentially operate as their own day-specific intercepts:

```

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)

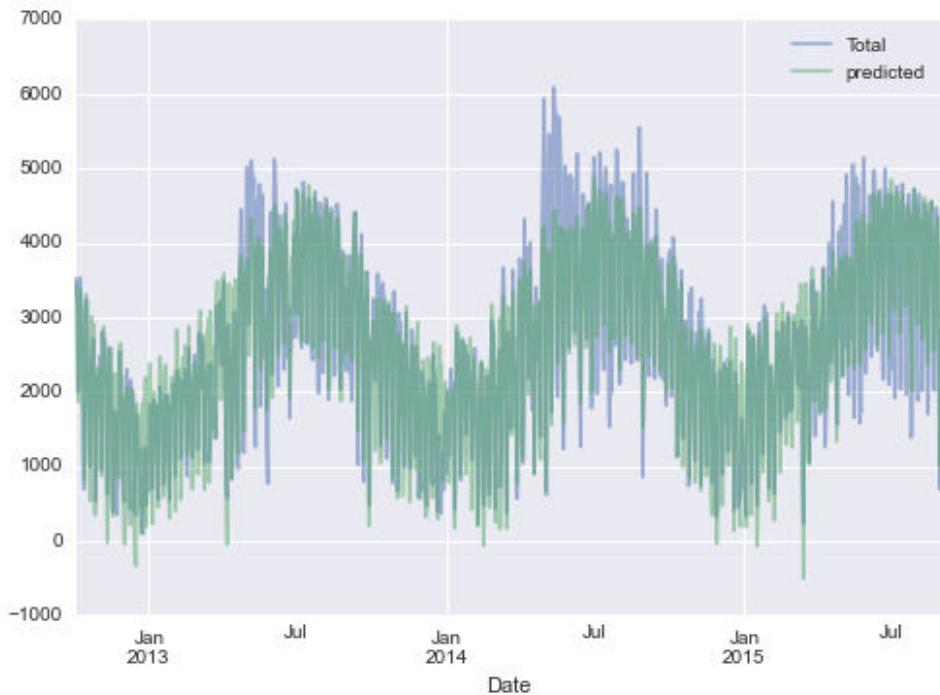
```

Finally, we can compare the total and predicted bicycle traffic visually:

```

daily[['Total', 'predicted']].plot(alpha=0.5);

```



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 work based 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 temperatures). Nevertheless, our rough approximation has is enough to give us some insights, and we can take a look at the coefficients of the linear model to get a rough estimate of how much each feature contributes to the daily bicycle count:

```
params = pd.Series(model.coef_, index=X.columns)
params
```

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:

```
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:

```
print(pd.DataFrame({'effect': params.round(0),
                     'error': err.round(0)}))
```

	effect	error
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_hrs	129	9
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 effect of an unexpected sunny day after a streak of rainy days). These are all potentially interesting effects, and we now have the tools to begin exploring them!

Figures

The following code creates some of the figures used above:

```
import os
if not os.path.exists('fig'):
    os.makedirs('fig')

Gaussian Basis

rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

gauss_model = make_pipeline(GaussianFeatures(10, 1.0),
                             LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

gf = gauss_model.named_steps['gaussianfeatures']
lm = gauss_model.named_steps['linearregression']

fig, ax = plt.subplots()

for i in range(10):
    selector = np.zeros(10)
    selector[i] = 1
    Xfit = gf.transform(xfit[:, None]) * selector
    yfit = lm.predict(Xfit)
    ax.fill_between(xfit, yfit.min(), yfit, color='gray', alpha=0.2)

ax.scatter(x, y)
ax.plot(xfit, gauss_model.predict(xfit[:, np.newaxis]))
ax.set_xlim(0, 10)
ax.set_ylim(yfit.min(), 1.5)

fig.savefig('fig/07.06-gaussian-basis.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

In-Depth: 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.

```
%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

In Section X.X we discussed Bayesian classification, in which we learn a simple model describing the distribution of each underlying class, and use these generative models to probabilistically determine labels for new points. That was an example of *generative classification*; here we will consider instead *discriminative classification*: rather than modeling each class, we simply find a line or curve (in two dimensions) or manifold (in multiple dimensions) which 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:

```

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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



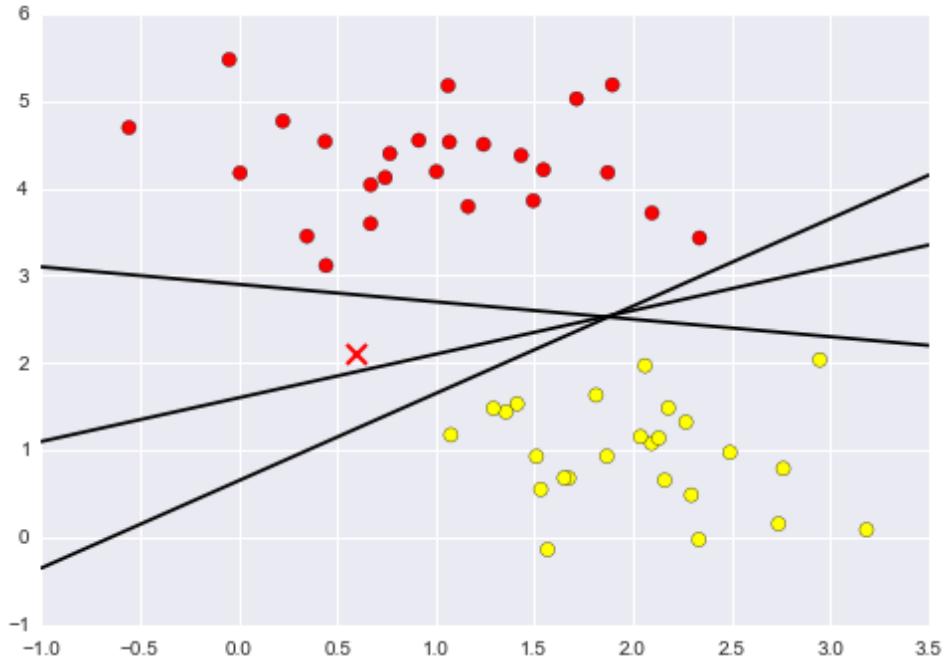
A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification. For two dimensional data like that shown above, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes here!

We can draw them as follows:

```
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);
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



These are three *very* different separators which, nevertheless, perfectly discriminate between these samples. Depending on which you choose, a new data point (for example, the one marked by the red “X” above) will be assigned a different label! Evidently our simple intuition of “drawing a line between classes” is not enough, and we need to think a bit deeper.

Support Vector Machines: Maximizing the *Margin*

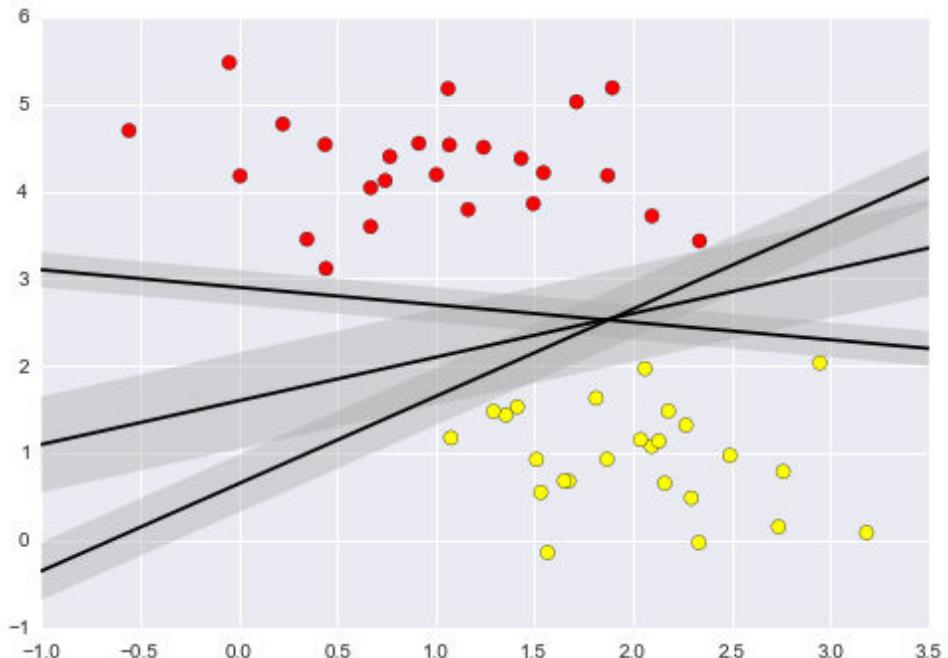
Support Vector Machines offer one way to improve on this. The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a *margin* of some width, up to the nearest point. Here is an example of how this might look:

```
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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



In support vector machines, the line which maximizes this margin is the one we will choose as the optimal model. Support Vector Machines are an example of such a *maximum 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 these data. For the time being, we will use a linear kernel and set the C parameter to a very large number: more on the meaning of these below.

```
from sklearn.svm import SVC # "Support Vector Classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)

SVC(C=10000000000.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
     gamma=0.0, 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:

```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D 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)

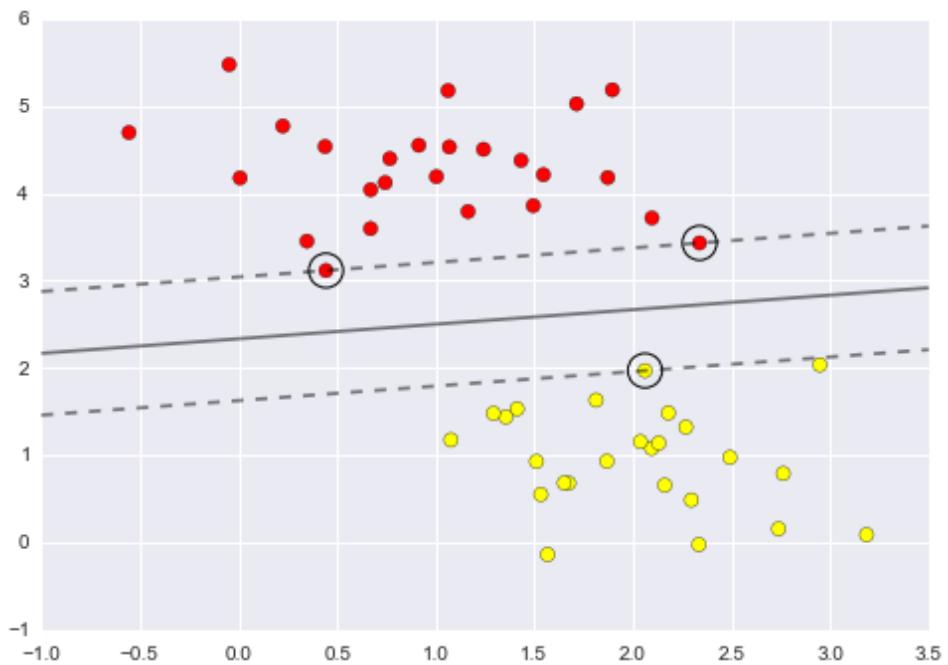
    # plot decision boundary & margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # 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)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(model);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



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 by the black circles in the above plot. 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, the identity of these points are stored in the `support_vectors_` attribute of the classifier:

```
model.support_vectors_
array([[ 0.44359863,  3.11530945],
       [ 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 vectors matter; any points further from the margin which 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 and first 120 points of this dataset:

```
def plot_svm(N=10, ax=None):
    X, y = make_blobs(n_samples=200, centers=2,
                      random_state=0, cluster_std=0.60)
```

```

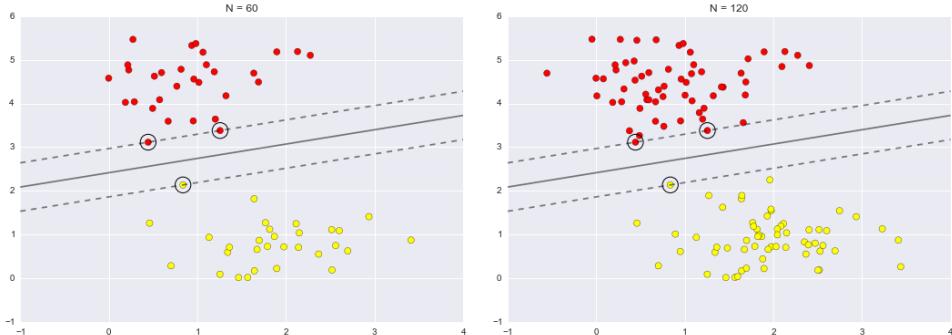
X = X[:N]
y = y[:N]
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)

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 = {}'.format(N))

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



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 has not 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 of the strengths of the SVM model.

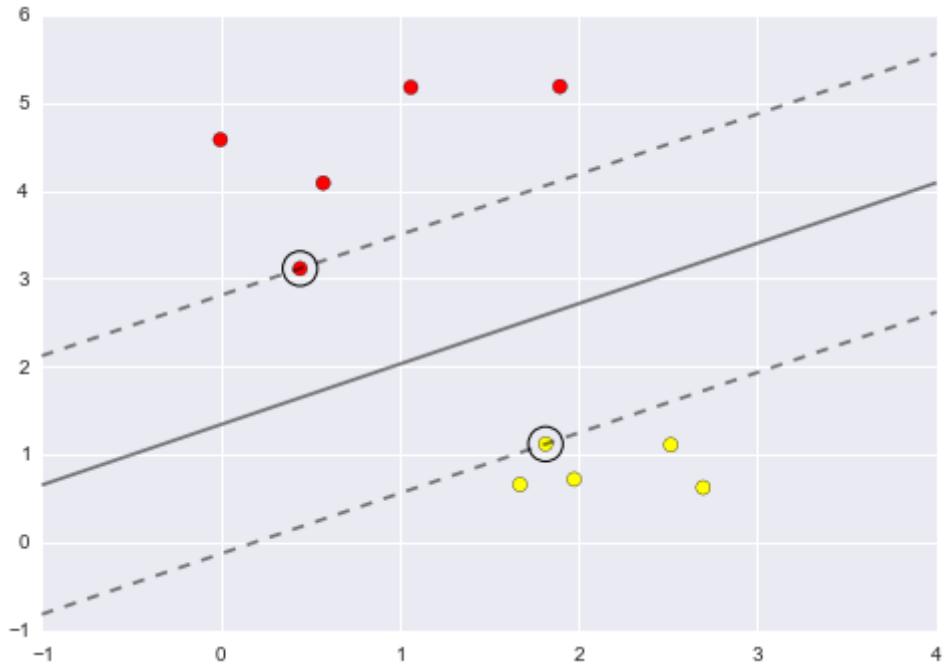
If you are running this notebook live, you can use IPython's interactive widgets to view this feature of the SVM model interactively:

```

from ipywidgets import interact, fixed
interact(plot_svm, N=[10, 200], ax=fixed(None));

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



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 linear regression examples of Section X.X. There we projected our data into higher-dimensional space defined by polynomials and gaussian basis functions, and thereby were able to fit for nonlinear relationships with a linear classifier.

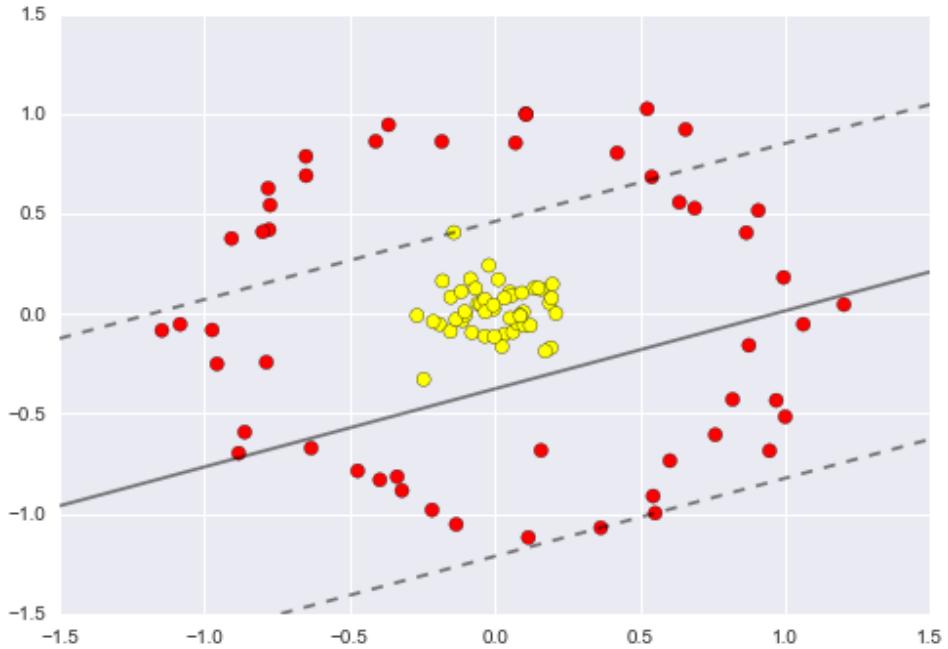
In SVM models, we can use a version of the same idea. To motivate the need for kernels, let's look at some data which is not linearly separable:

```
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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



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 linear regression we explored in Section X.X, 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:

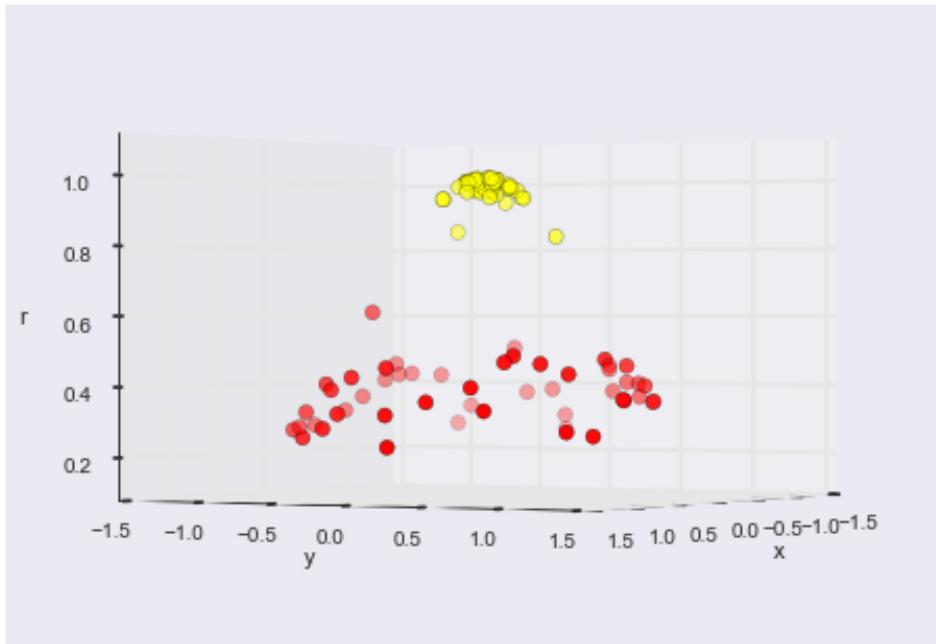
```
r = np.exp(-(X ** 2).sum(1))
```

We can visualize this extra data dimension using a three-dimensional plot – if you are running this notebook live, you will be able to use the sliders to rotate the plot:

```
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], azim=(-180, 180),
         X=fixed(X), y=fixed(y));
```



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, linearly separable 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 function transformation is known as a *kernel transformation*, as it is based on a similarity relationship (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 to an RBF (radial basis function) kernel, using the `kernel` model hyperparameter:

```

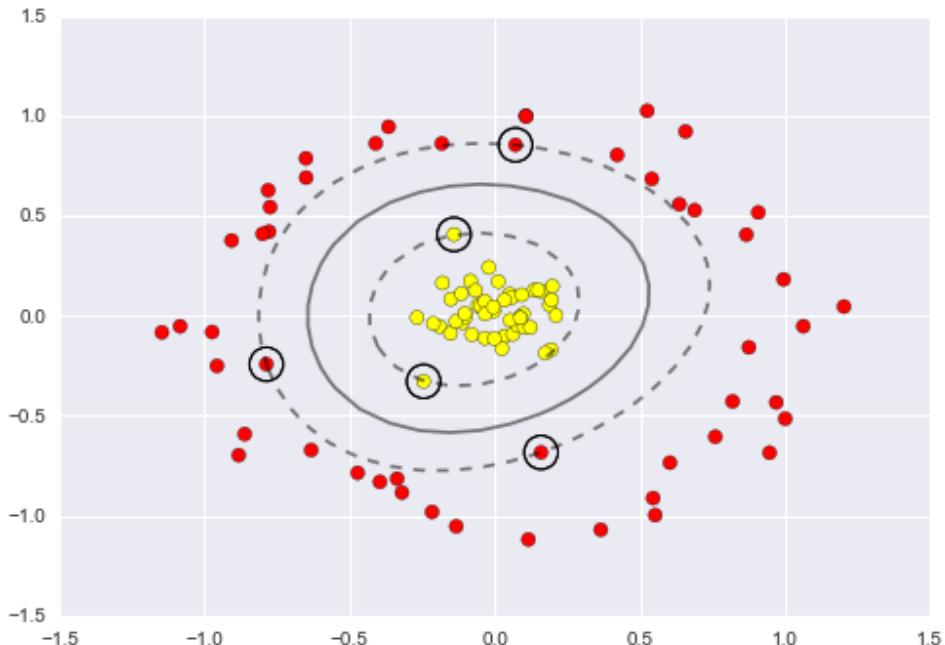
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)

SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
     gamma=0.0, kernel='rbf', max_iter=-1, probability=False,
     random_state=None, shrinking=True, tol=0.001, verbose=False)

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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```



Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary. This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods – especially for models in which the kernel trick can be used.

Tuning the SVM: Softening Margins

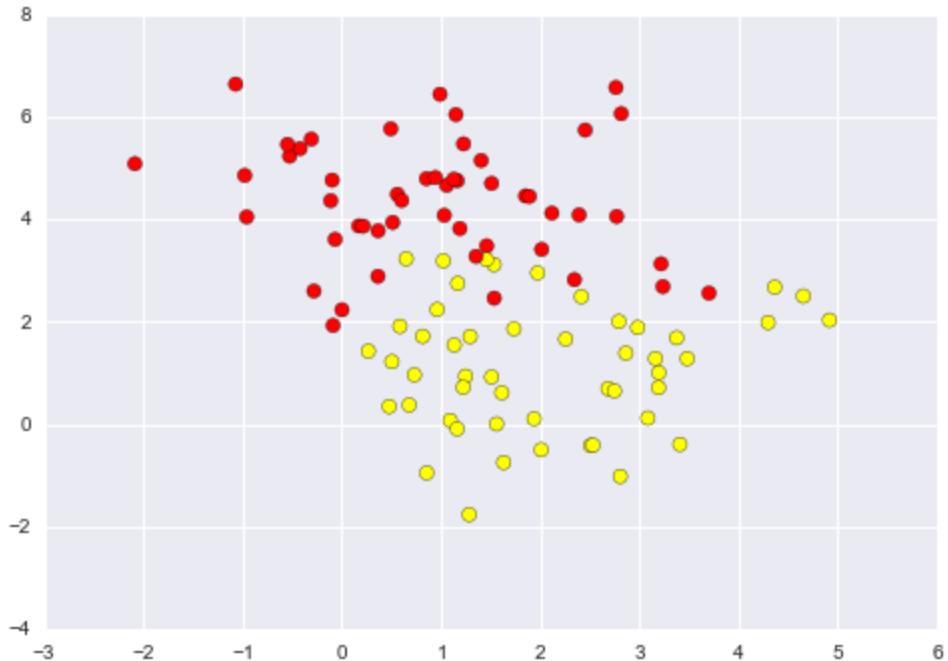
All of the above discussion centered around very clean datasets, in which a perfect decision boundary exists. But what if your data have some amount of overlap? For example, you may have data like this:

```

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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



To handle this case, the SVM implementation has a bit of a fudge-factor which “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, most often known as C . For very large C , the margin is hard, and points cannot lie in it. For smaller C , the margin is softer, and can grow to encompass some points.

The following plot gives a visual picture of how a changing C parameter affects the final fit, via the softening of the margin:

```

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)

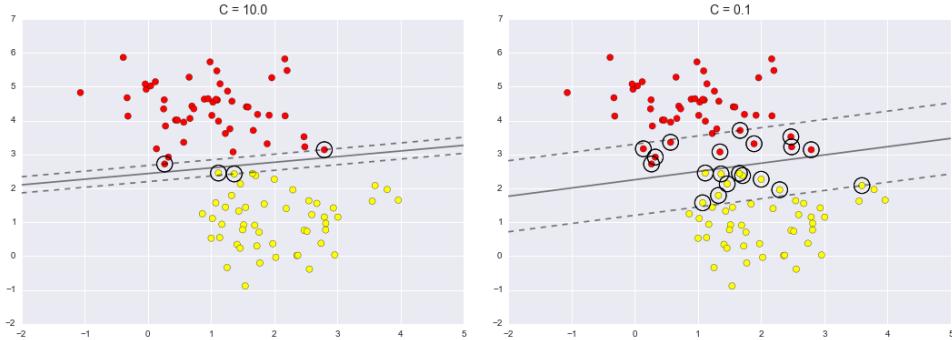
```

```

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)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```



The optimal value of the C parameter will depend on your dataset, and should be tuned using cross-validation or a similar procedure (see Section X.X).

Example: Face Recognition

As an example of support vector machines in action, let's take a look at the facial recognition 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 dataset is built into scikit-learn:

```

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:

```

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]])

```



Each image contains [62x47] or nearly 3000 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 Component Analysis (see Section X.X) to extract 150 fundamental components to feed into our support vector machine classifier. We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

```
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='auto')
model = make_pipeline(pca, svc)
```

For the sake of testing our classifier output, we will split the data into a training and testing set:

```
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 parameters. 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:

```

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 35 s, sys: 3.22 s, total: 38.2 s
Wall time: 28.2 s
{'svc__gamma': 0.0005, 'svc__C': 5}

```

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, which the model has not yet seen:

```

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:

```

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);

```

Predicted Names; Incorrect Labels in Red



Out of this small sample, our optimal estimator mis-labeled only a few faces (marked in red). We can get a better sense of our estimator's performance using the classification report (see Section X.X), which lists recovery statistics label-by-label:

```
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit,
                            target_names=faces.target_names))

          precision    recall  f1-score   support

      Ariel Sharon      0.71      0.80      0.75       15
      Colin Powell      0.84      0.87      0.86       68
      Donald Rumsfeld     0.74      0.81      0.77       31
      George W Bush     0.96      0.79      0.87      126
      Gerhard Schroeder    0.68      0.83      0.75       23
      Hugo Chavez        0.89      0.80      0.84       20
      Junichiro Koizumi     0.80      1.00      0.89       12
      Tony Blair         0.80      0.98      0.88       42

 avg / total         0.86      0.84      0.84      337
```

We might also display the confusion matrix between these classes:

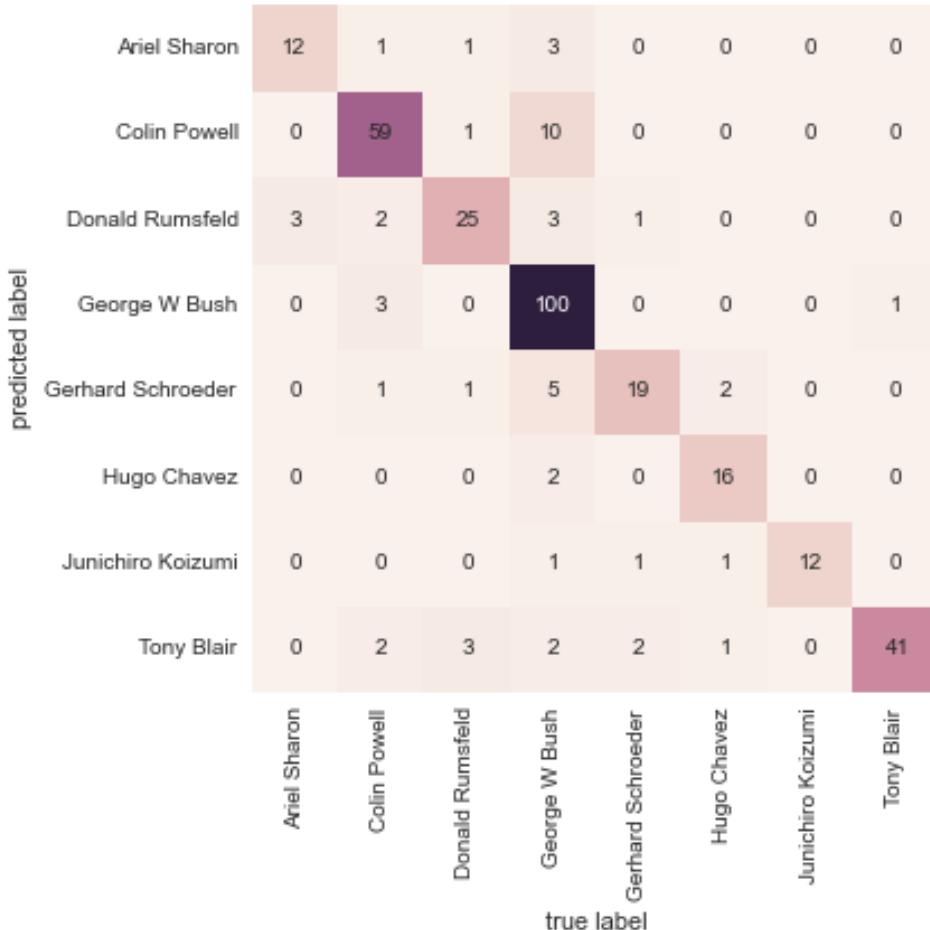
```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(ytest, yfit)
```

```

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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



This helps us get a sense of which labels are likely to be confused by the estimator.

For a real life facial recognition task, in which the photos do not come pre-cropped 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, and extract features which 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 general and faces in particular.

Support Vector Machine Summary

We have seen here a brief intuitive introduction to the principals behind Support Vector Machines. These methods are a powerful classification method for a number of reasons:

- their dependence on relatively few support vectors means that they are very compact models, and take up very little memory.
- Once the model is trained, the prediction phase is very fast
- because they are affected only by points near the margin, they work well with high-dimensional data: even data with more dimensions than samples, which is a challenging regime for other algorithms
- their integration with kernel methods makes them very versatile, able to adapt to many types of data

However, SVMs have several disadvantages as well:

- The scaling with the number of samples N is $\mathcal{O}[N^3]$ at worst, or $\mathcal{O}[N^2]$ for efficient implementations. For large numbers of training samples, this computational cost can be prohibitive.
- The results are strongly dependent on a suitable choice for the softening parameter C . This must be carefully chosen via cross-validation, which can be expensive as datasets grow.
- The results do not have a direct probabilistic interpretation. This can be estimated via an internal cross-validation (see the `probability` parameter of SVC), but this extra estimation is costly.

With those traits in mind, I generally only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for my needs. Nevertheless, if you have the CPU cycles to commit to training and cross-validating an SVM on your data, the method can lead to excellent results.

In-Depth: Decision Trees and Random Forests

Previously we have looked in-depth at a simple generative classifier (Naive Bayes; Section X.X) and a powerful discriminative classifier (Support Vector Machines; Section X.X) 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, meaning that it relies on aggregating the results of an ensemble of simpler estimators. The somewhat surprising result with such ensemble methods is that the sum can be greater than the parts: that is, a majority vote among a number of

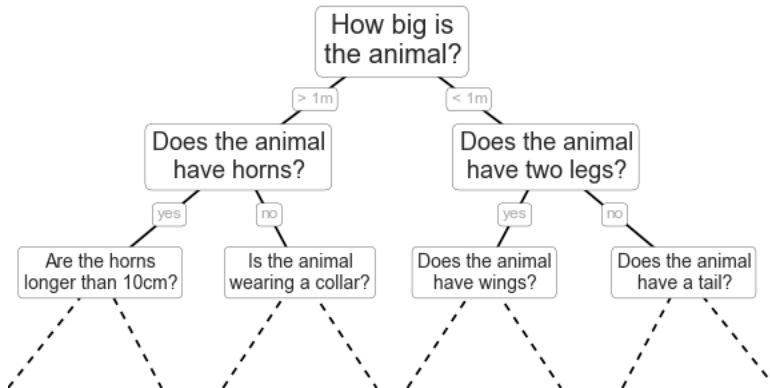
estimators can end up being better than any of the individual estimators doing the voting! We will see examples of this below.

```
%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 this reason we'll start by discussing decision trees themselves.

Decision trees are extremely intuitive ways to classify or label objects: you simply ask a 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 following:



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 narrowing 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 of decision trees, the “questions” above 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 value within one of the features. We will see an example of this below.

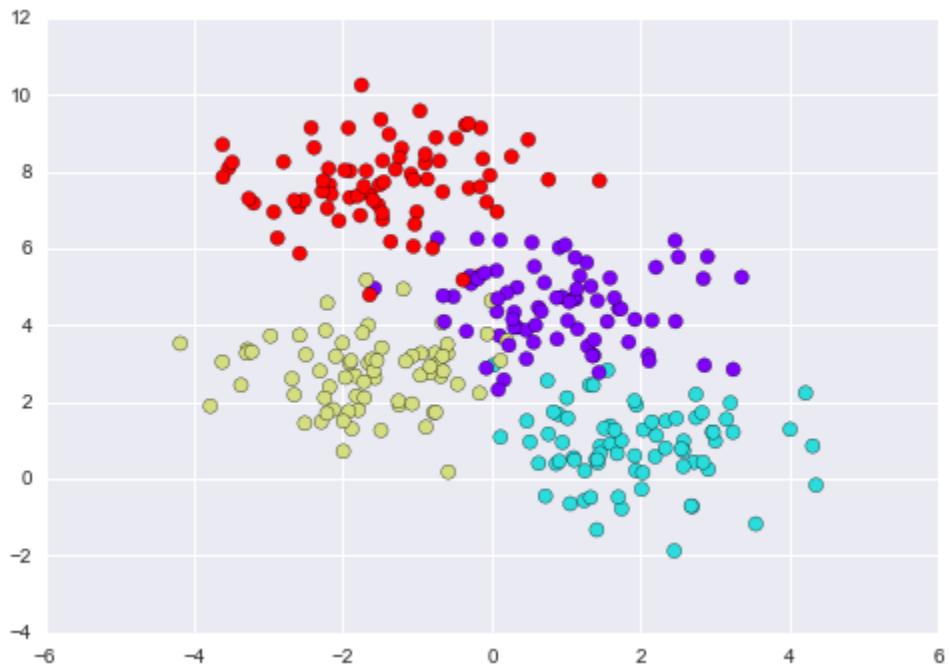
Creating a Decision Tree

Consider the following two-dimensional data, which has one of four class labels:

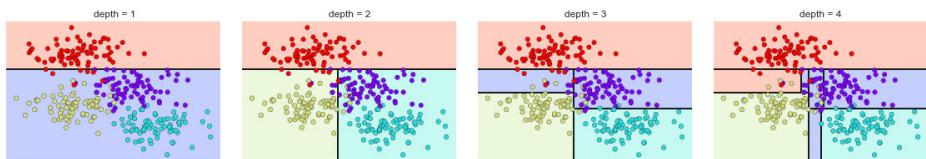
```
from sklearn.datasets import make_blobs

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');
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



A simple decision tree built on this data will iteratively split the data along one or the other 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. Here is a visualization of the first four levels of a decision tree classifier for this data:



Notice that after the first split, every point in the upper branch is red, so there is no need to further subdivide this branch. Except for nodes which 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:

```
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

```
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)

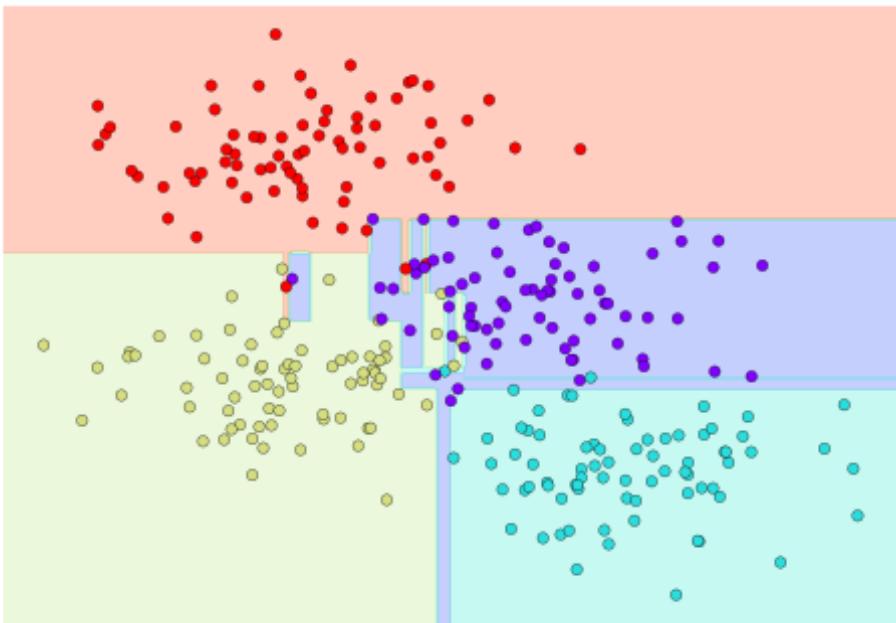
    # Create a color plot with the results
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap=cmap, clim=(y.min(), y.max()),
                           zorder=1)

    ax.set(xlim=xlim, ylim=ylim)
```

Now we can examine what the decision tree classification looks like:

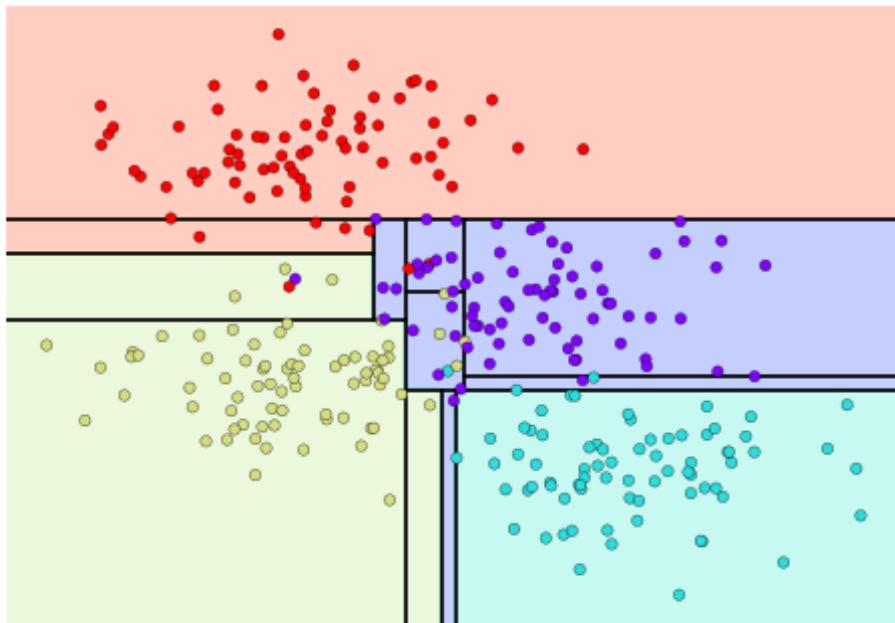
```
visualize_classifier(DecisionTreeClassifier(), X, y)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



If you're running this notebook live, you can use the helpers script included at the end of this notebook to bring up an interactive visualization of the decision tree building process:

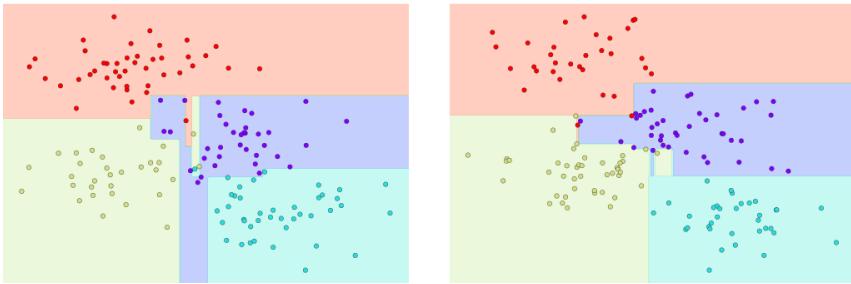
```
# if this raises an import error, first execute the cells in the Figures section
# at the end of this notebook.
import helpers_07_08
helpers_07_08.plot_tree_interactive(X, y);
```



Notice that as the depth increases, we tend to get very strangely-shaped classification regions; for example, at a depth of five (above) there is a tall and skinny purple region between the yellow and blue regions above. 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 properties of the data. That is, this decision tree, even at only 5 levels deep, is clearly over-fitting our data.

Decision Trees and over-fitting

Such over-fitting 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 over-fitting is to look at models trained on different subsets of the data – for example, in the following we train two different trees, each on half of the original data:

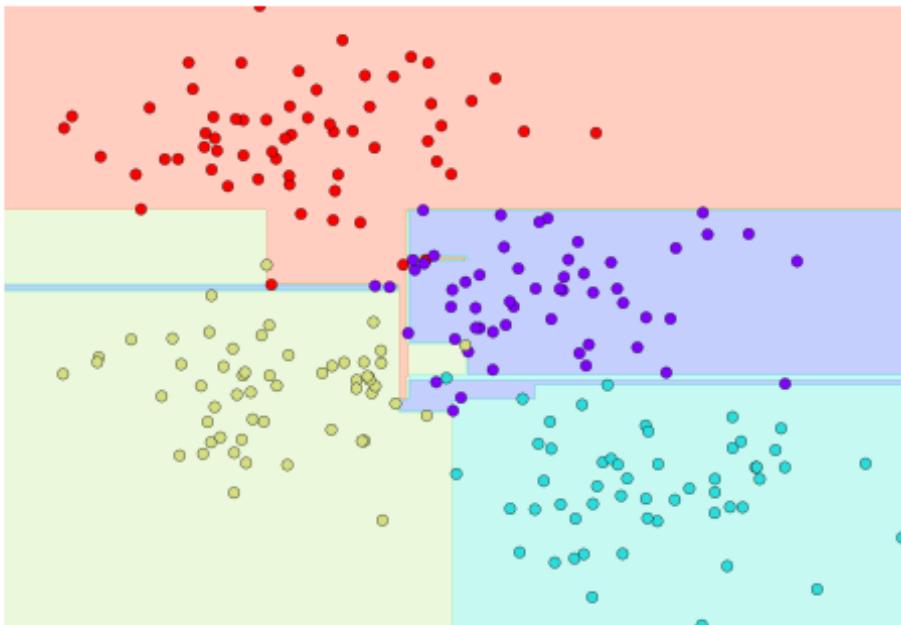


It is clear that in some places, the two trees produce consistent results (e.g. in the four corners), while in other places, the two trees give very different classifications (e.g. in the regions between any two clusters). The key observation is that the inconsistencies tend to happen where the classification 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 interactively display the fits of trees trained on a random subset of the data:

```
# if this raises an ImportError, first execute the cells in the Figures section
# at the end of this notebook.
import helpers_07_08
helpers_07_08.randomized_tree_interactive(X, y)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



Just as using information from two trees improves our results, we might expect that using 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 the effect of this overfitting – is what underlies an ensemble method called *bagging*. Bagging makes use of a set (a grab-bag, perhaps) of parallel randomized estimators, each of which over-fits the data, and averages the results to find a better classification. An ensemble of randomized decision trees is known as a *random forest*.

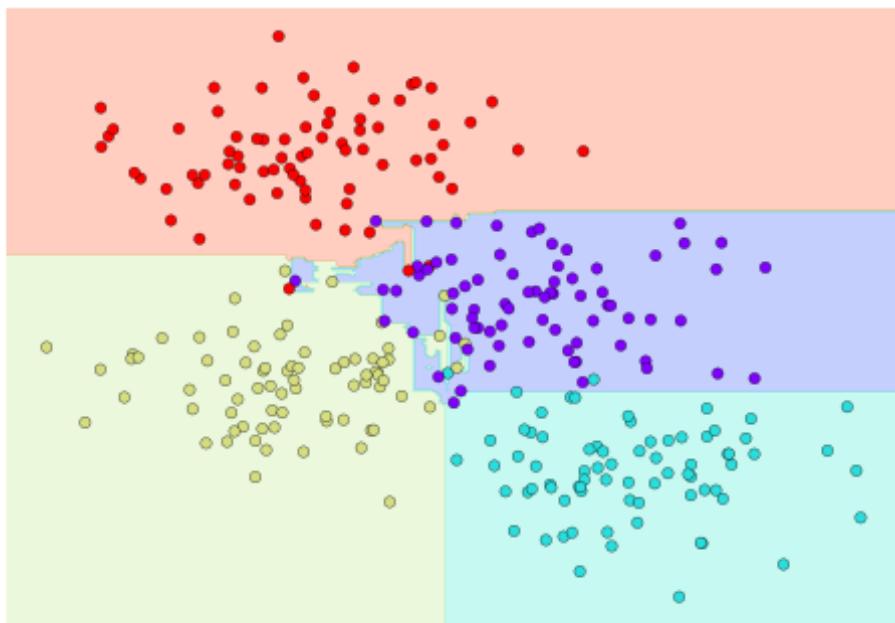
This type of Bagging classification can be done manually using scikit-learn's `BaggingClassifier` meta-estimator, for example like this:

```
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)
```

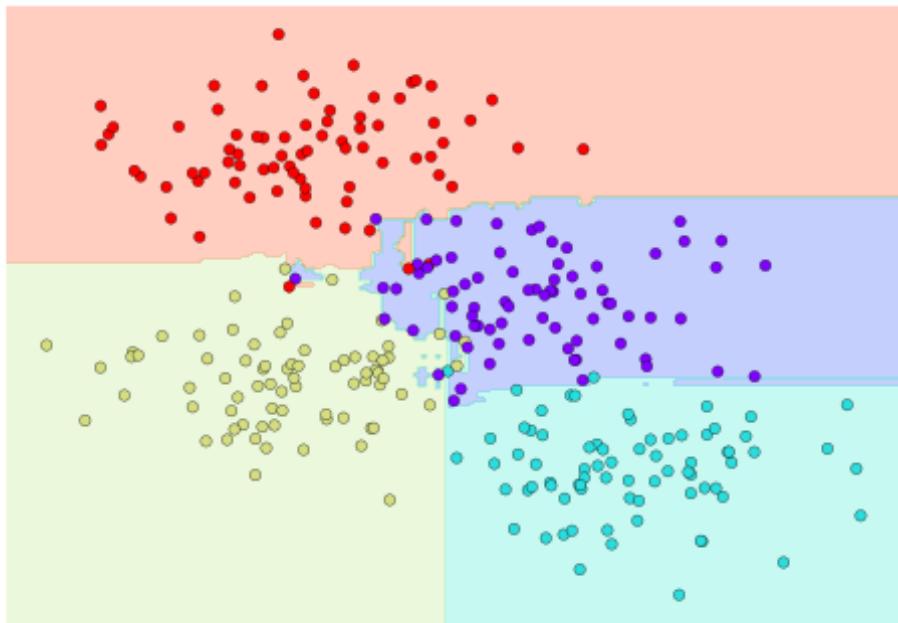
```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



In this example, we have randomized the data by fitting each estimator with a random subset of 80% of the training points. In practice, decision trees are more effectively randomized by injecting some stochasticity in how the splits are chosen: this way all the data contributes to the fit each time, but the results of the fit still have the desired randomness. For example, when determining which feature to split on, the randomized tree might select randomly from among the top several features. You can read more technical details about these randomization strategies in the [scikit-learn documentation](#) and references within.

In scikit-learn, such an optimized ensemble of randomized decision trees is implemented in the `RandomForestClassifier` estimator, which takes care of all the randomization 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:

```
from sklearn.ensemble import RandomForestClassifier  
  
model = RandomForestClassifier(n_estimators=100, random_state=0)  
visualize_classifier(model, X, y);  
  
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



We see that by averaging over 100 randomly perturbed models, we end up with an overall model which is much closer to our intuition about how the parameter space should be split.

Random Forest Regression

Above we were considering random forests within the context of classification. Random forests can also be made to work in the case of regression (that is, continuous rather than categorical variables). The estimator to use for this is the `RandomForestRegressor`, and the syntax is very similar to what we saw above.

Consider the following data, drawn from the combination of a fast and slow oscillation:

```
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))

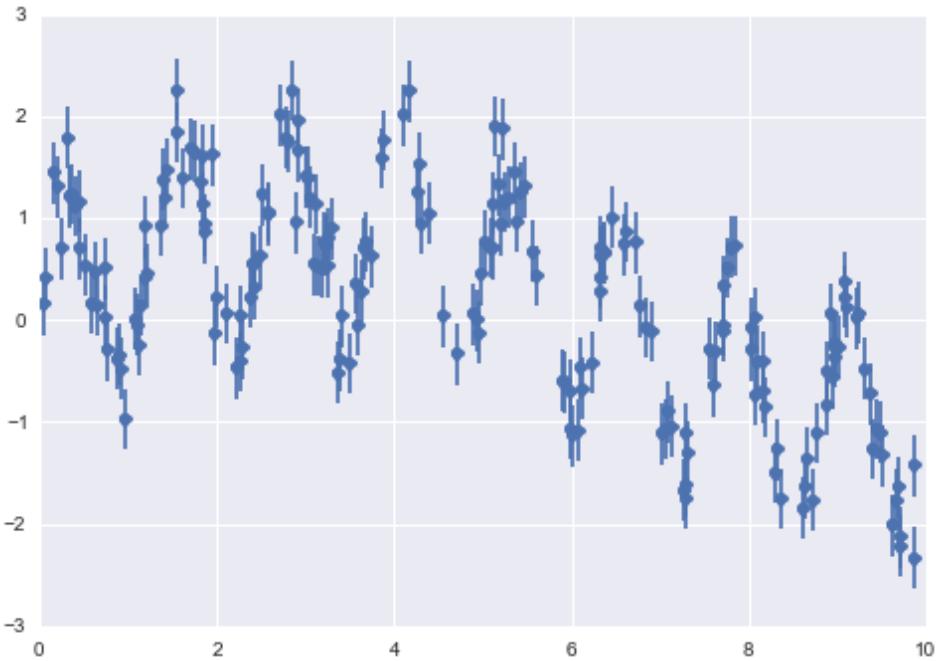
    return slow_oscillation + fast_oscillation + noise
```

```

y = model(x)
plt.errorbar(x, y, 0.3, fmt='o');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```



Using the random forest regressor, we can find the best fit curve as follows:

```

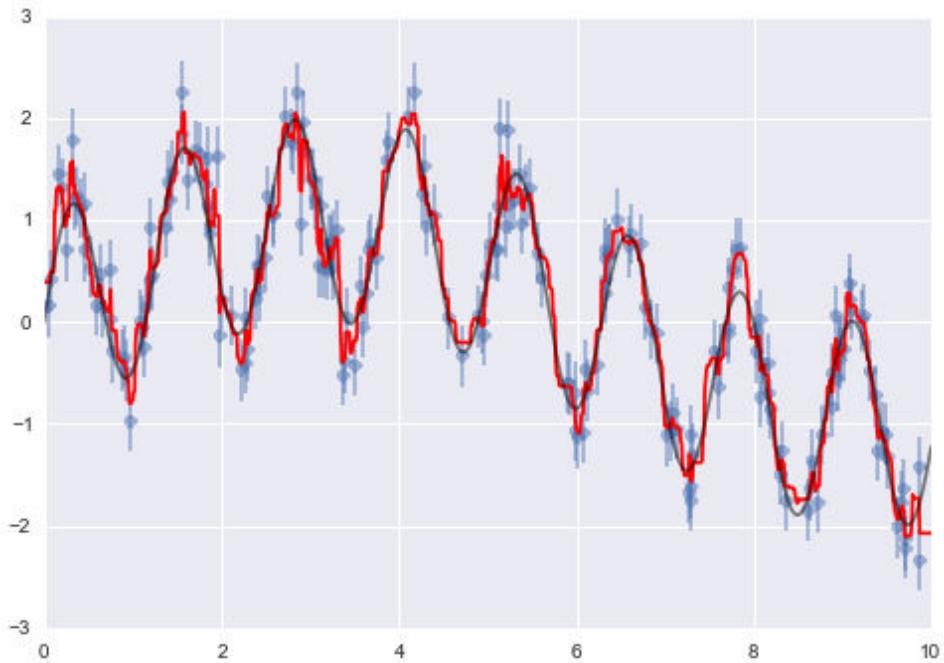
from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(200)
forest.fit(x[:, None], y)

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);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```



Here the true model is shown in the smooth gray curve, while the random forest model is shown in red. As you can see, the non-parametric random forest model is flexible enough to fit the multi-period data, without us needing to specifying a multi-period model!

Example: Random Forest for Classifying Digits

In Section X.X we took a quick look at the hand-written digits data. Let's use that again here to see how the random forest classifier can be used in this context.

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.keys()

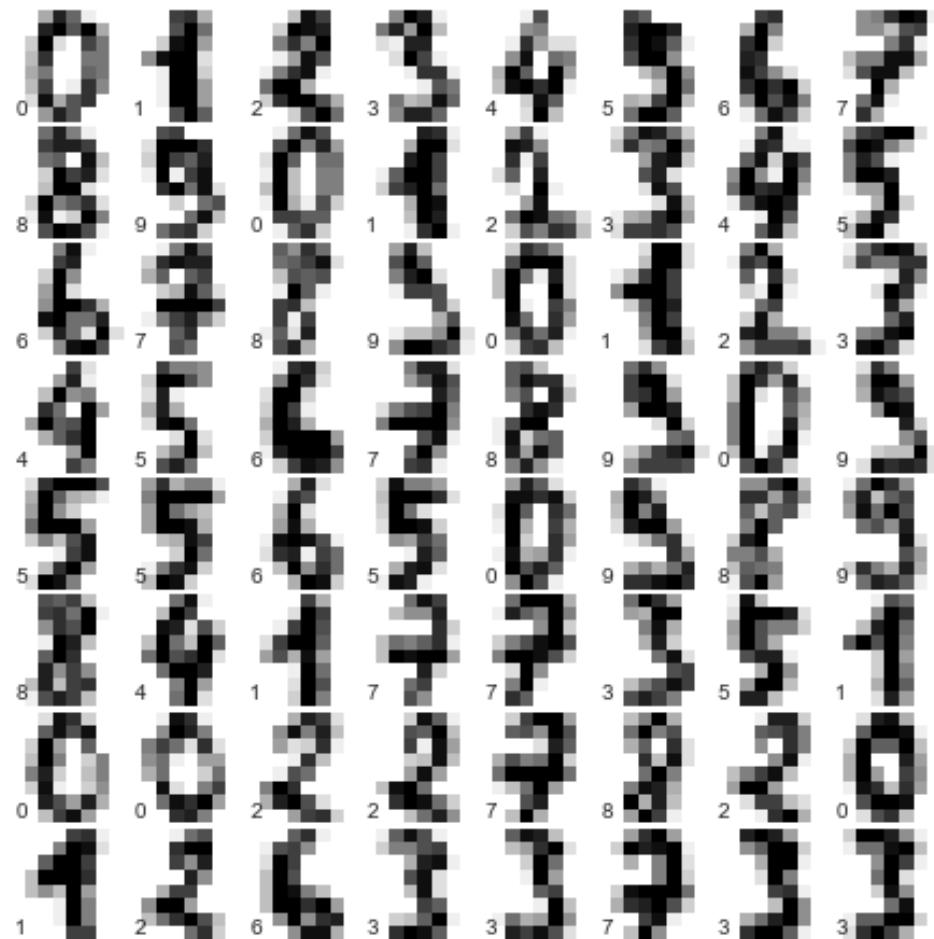
dict_keys(['target', 'DESCR', 'target_names', 'images', 'data'])
```

To remind us what we're looking at, we'll visualize the first few data points:

```
# 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]))
```



We can quickly classify the digits using a random forest as follows:

```
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:

```
from sklearn import metrics  
print(metrics.classification_report(ypred, ytest))
```

	precision	recall	f1-score	support
0	1.00	0.97	0.99	38
1	0.98	0.98	0.98	43
2	0.95	1.00	0.98	42
3	0.98	0.94	0.96	47
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 / total	0.98	0.98	0.98	450

and for good measure, plot the confusion matrix:

```
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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

	0	1	2	3	4	5	6	7	8	9
predicted label	37	0	1	0	0	0	0	0	0	0
0	0	42	0	0	0	0	0	0	1	0
1	0	0	42	0	0	0	0	0	0	0
2	0	0	0	44	0	0	0	0	0	0
3	0	0	1	0	37	0	0	0	1	1
4	0	0	0	0	0	47	0	0	0	0
5	0	1	0	0	0	0	52	0	0	0
6	0	0	0	0	0	0	0	48	1	0
7	0	0	0	0	1	0	0	0	45	0
8	0	0	0	1	0	0	0	0	0	45
9	0	0	0	0	0	1	0	0	0	0
true label	0	1	2	3	4	5	6	7	8	9

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*, and in particular the Random Forest – an ensemble of randomized decision trees. Random Forests are a powerful method with several advantages:

- Both training and prediction is 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 under-fit by other estimators.

A primary disadvantage of random forests is that the results are not easily interpretable: that is, if you would like to draw conclusions about the *meaning* of the classification model, random forests are not the best choice.

Figure Code

```
import os
if not os.path.exists('fig'):
    os.makedirs('fig')
```

Helper Code

The following will create a module `helpers_07_08.py` which contains some tools used in the above scripts.

```
%%file helpers_07_08.py

import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from IPython.html.widgets import interact


def visualize_tree(estimator, X, y, boundaries=True,
                   xlim=None, ylim=None, ax=None):
    ax = ax or plt.gca()

    # Plot the training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap='rainbow',
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    if xlim is None:
        xlim = ax.get_xlim()
    if ylim is None:
        ylim = ax.get_ylim()

    # fit the estimator
    estimator.fit(X, y)
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = estimator.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    n_classes = len(np.unique(y))
```

```

Z = Z.reshape(xx.shape)
contours = ax.contourf(xx, yy, Z, alpha=0.3,
                      levels=np.arange(n_classes + 1) - 0.5,
                      cmap='rainbow', clim=(y.min(), y.max()),
                      zorder=1)

ax.set(xlim=xlim, ylim=ylim)

# Plot the decision boundaries
def plot_boundaries(i, xlim, ylim):
    if i >= 0:
        tree = estimator.tree_

        if tree.feature[i] == 0:
            ax.plot([tree.threshold[i], tree.threshold[i]], ylim, '-k', zorder=2)
            plot_boundaries(tree.children_left[i],
                            [xlim[0], tree.threshold[i]], ylim)
            plot_boundaries(tree.children_right[i],
                            [tree.threshold[i], xlim[1]], ylim)

        elif tree.feature[i] == 1:
            ax.plot(xlim, [tree.threshold[i], tree.threshold[i]], '-k', zorder=2)
            plot_boundaries(tree.children_left[i], xlim,
                            [ylim[0], tree.threshold[i]])
            plot_boundaries(tree.children_right[i], xlim,
                            [tree.threshold[i], ylim[1]]))

    if boundaries:
        plot_boundaries(0, xlim, ylim)

def plot_tree_interactive(X, y):
    def interactive_tree(depth=5):
        clf = DecisionTreeClassifier(max_depth=depth, random_state=0)
        visualize_tree(clf, X, y)

    return interact(interactive_tree, depth=[1, 5])

def randomized_tree_interactive(X, y):
    N = int(0.75 * X.shape[0])

    xlim = (X[:, 0].min(), X[:, 0].max())
    ylim = (X[:, 1].min(), X[:, 1].max())

    def fit_randomized_tree(random_state=0):
        clf = DecisionTreeClassifier(max_depth=15)
        i = np.arange(len(y))
        rng = np.random.RandomState(random_state)
        rng.shuffle(i)
        visualize_tree(clf, X[i[:N]], y[i[:N]], boundaries=False,
                      xlim=xlim, ylim=ylim)

```

```
    interact(fit_randomized_tree, random_state=[0, 100]);  
Overwriting helpers_07_08.py
```

Decision Tree Example

```
fig = plt.figure(figsize=(10, 4))  
ax = fig.add_axes([0, 0, 0.8, 1], frameon=False, xticks=[], yticks=[])  
ax.set_title('Example Decision Tree: Animal Classification', size=24)  
  
def text(ax, x, y, t, size=20, **kwargs):  
    ax.text(x, y, t,  
            ha='center', va='center', size=size,  
            bbox=dict(boxstyle='round', ec='k', fc='w'), **kwargs)  
  
text(ax, 0.5, 0.9, "How big is\nthe animal?", 20)  
text(ax, 0.3, 0.6, "Does the animal\nhave horns?", 18)  
text(ax, 0.7, 0.6, "Does the animal\nhave two legs?", 18)  
text(ax, 0.12, 0.3, "Are the horns\nlonger than 10cm?", 14)  
text(ax, 0.38, 0.3, "Is the animal\nwearing a collar?", 14)  
text(ax, 0.62, 0.3, "Does the animal\nhave wings?", 14)  
text(ax, 0.88, 0.3, "Does the animal\nhave a tail?", 14)  
  
text(ax, 0.4, 0.75, "> 1m", 12, alpha=0.4)  
text(ax, 0.6, 0.75, "< 1m", 12, alpha=0.4)  
  
text(ax, 0.21, 0.45, "yes", 12, alpha=0.4)  
text(ax, 0.34, 0.45, "no", 12, alpha=0.4)  
  
text(ax, 0.66, 0.45, "yes", 12, alpha=0.4)  
text(ax, 0.79, 0.45, "no", 12, alpha=0.4)  
  
ax.plot([0.3, 0.5, 0.7], [0.6, 0.9, 0.6], '-k')  
ax.plot([0.12, 0.3, 0.38], [0.3, 0.6, 0.3], '-k')  
ax.plot([0.62, 0.7, 0.88], [0.3, 0.6, 0.3], '-k')  
ax.plot([0.0, 0.12, 0.20], [0.0, 0.3, 0.0], '--k')  
ax.plot([0.28, 0.38, 0.48], [0.0, 0.3, 0.0], '--k')  
ax.plot([0.52, 0.62, 0.72], [0.0, 0.3, 0.0], '--k')  
ax.plot([0.8, 0.88, 1.0], [0.0, 0.3, 0.0], '--k')  
ax.axis([0, 1, 0, 1])  
  
fig.savefig('fig/07.08-decision-tree.png')  
plt.close(fig)
```

Decision Tree Levels

```
from helpers_07_08 import visualize_tree  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.datasets import make_blobs  
  
fig, ax = plt.subplots(1, 4, figsize=(16, 3))
```

```

fig.subplots_adjust(left=0.02, right=0.98, wspace=0.1)

X, y = make_blobs(n_samples=300, centers=4,
                   random_state=0, cluster_std=1.0)

for axi, depth in zip(ax, range(1, 5)):
    model = DecisionTreeClassifier(max_depth=depth)
    visualize_tree(model, X, y, ax=axi)
    axi.set_title('depth = {}'.format(depth))

fig.savefig('fig/07.08-decision-tree-levels.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```

Decision Tree Overfitting

```

model = DecisionTreeClassifier()

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
visualize_tree(model, X[::2], y[::2], boundaries=False, ax=ax[0])
visualize_tree(model, X[1::2], y[1::2], boundaries=False, ax=ax[1])

fig.savefig('fig/07.08-decision-tree-overfitting.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):


```

In Depth: Principal Component Analysis

Up until now, we have been looking in depth at supervised learning estimators: those estimators which predict labels based on labeled training data. Here we begin looking at several 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 unsupervised algorithms, Principal Component Analysis (PCA). PCA is fundamentally a dimensionality reduction algorithm, but it can also be useful as a tool for visualization, 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 of these further applications.

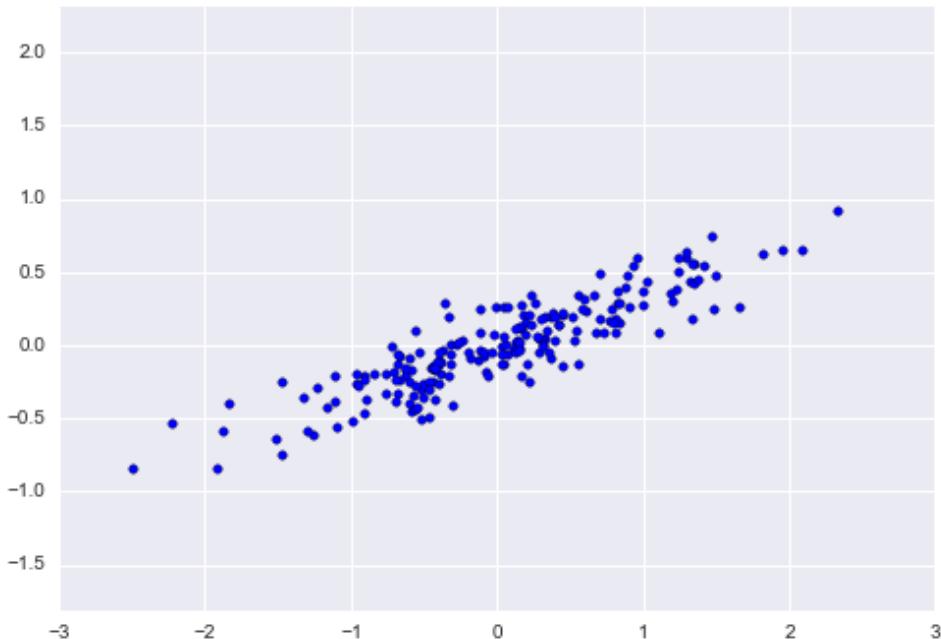
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
```

Introducing Principal Component Analysis

Principal Component Analysis is a fast and flexible unsupervised method for dimensionality reduction in data. Its behavior is easiest to visualize by looking at a two-dimensional dataset. Consider the following 200 points:

```
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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):
```



By eye, it is clear that there is nearly linear relationship between the x and y variables. This is reminiscent of the linear regression data we explored in Section X.X, 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.

In Principal Component Analysis, this relationship is quantified 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:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)

PCA(copy=True, n_components=2, whiten=False)
```

The fit learns some quantities from the data, most importantly the “components” and “explained variance”:

```
print(pca.components_)
[[ 0.94446029  0.32862557]
 [-0.32862557  0.94446029]]

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 variance” to define the squared-length of the vector:

```
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)

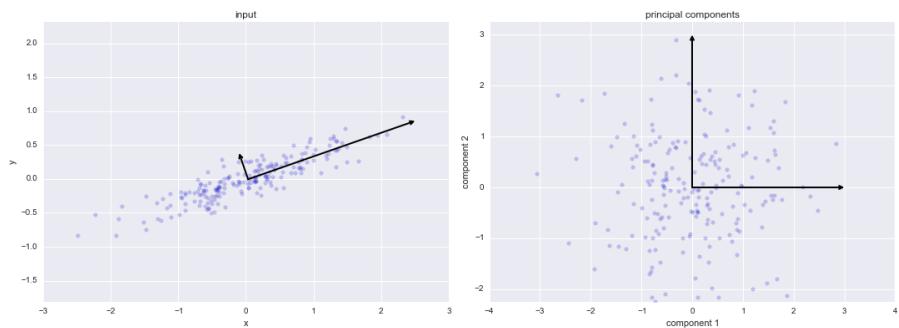
# plot data
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length)
    draw_vector(pca.mean_, pca.mean_ + v)
plt.axis('equal');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):
```



These vectors represent the *principal axes* of the data, and the length shown above is an indication of how “important” that axis is in describing the distribution of the data – more precisely, it is a measure of the variance of the data when projected 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 following:



This transformation from data axes to principal axes is known 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 mathematical curiosity, 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, which results in a lower-dimensional projection of the data which preserves the maximal data variance.

Here is an example of using PCA as a dimensionality reduction transform:

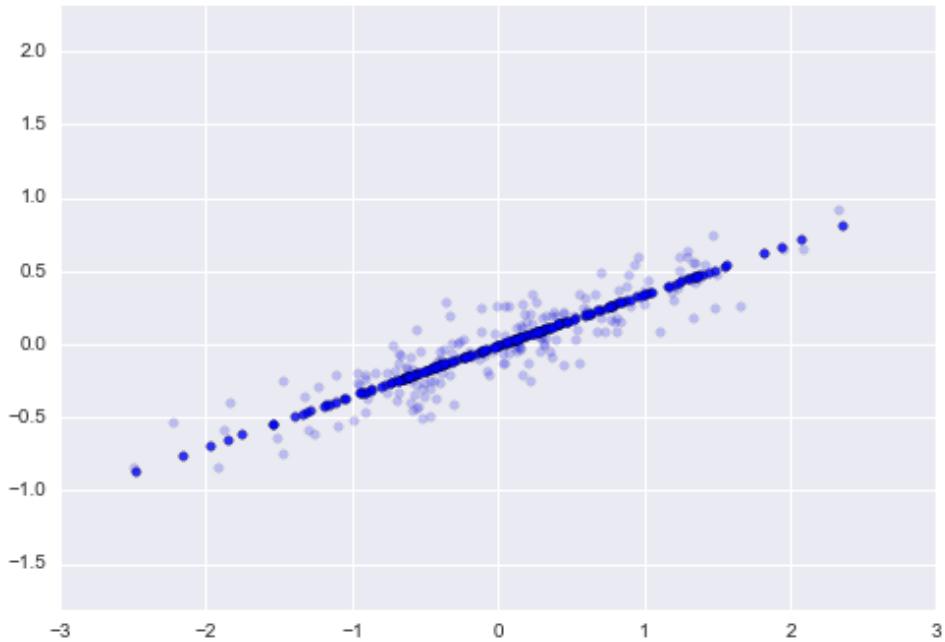
```
pca = PCA(n_components=1)
pca.fit(X)
X_pca = pca.transform(X)
print("original shape: ", X.shape)
print("transformed shape:", X_pca.shape)

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:

```
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');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



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 which is cut out (proportional to the spread of points about the line formed above) is roughly a measure 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 the data by 50%, the overall relationship between the data points are mostly preserved.

PCA for Visualization: Hand-written Digits

The usefulness of the dimensionality reduction may not be entirely apparent in only two dimensions, but becomes much more clear when looking at high-dimensional data. To see this, let’s take a quick look at the application of PCA to the digits data we saw in Section X.X.

We start by loading the data:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape

(1797, 64)
```

Recall that the data consist of 8x8 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:

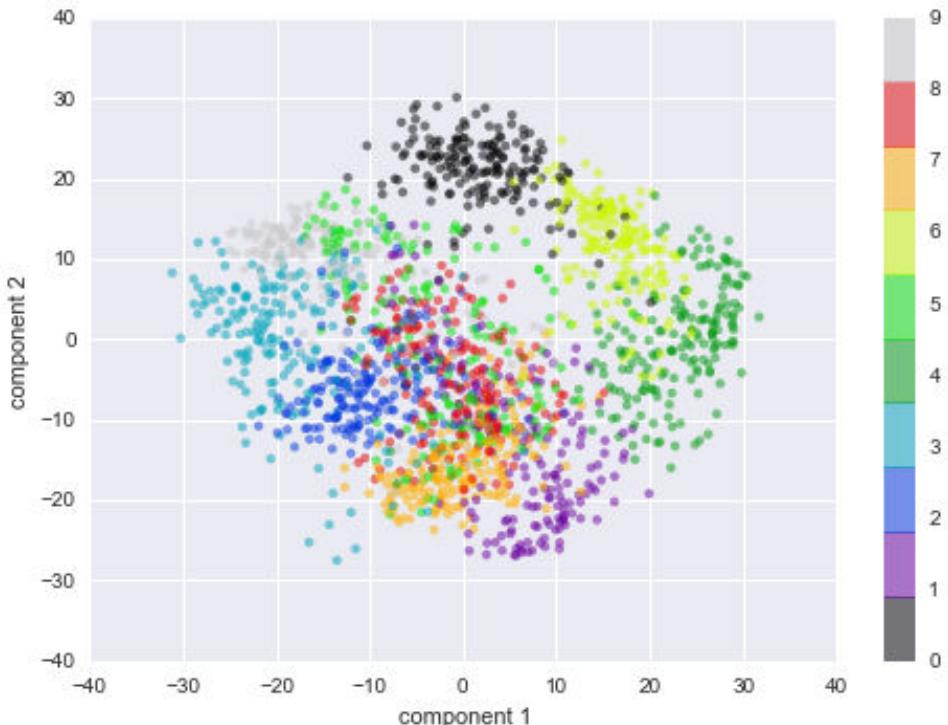
```
pca = PCA(2) # project from 64 to 2 dimensions
projected = pca.fit_transform(digits.data)
print(digits.data.shape)
print(projected.shape)

(1797, 64)
(1797, 2)
```

We can now plot the first two principal components of each point to learn about the data:

```
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();

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):
```



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, and have done this in an unsupervised manner – that is, without reference to the labels.

What do the Components Mean?

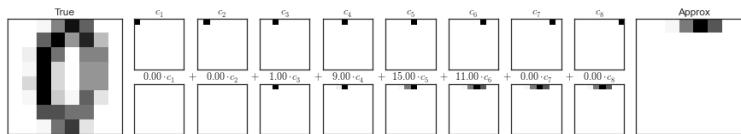
We can go a bit further here, and begin to ask what the reduced dimensions *mean*. This meaning can be understood in terms of combinations of basis vectors. For example, each image in the training set is defined by a collection of 64 pixel values, which we will call the vector x :

$$x = [x_1, x_2, x_3 \dots x_{64}]$$

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:

$$\text{image}(x) = x_1 \cdot (\text{pixel 1}) + x_2 \cdot (\text{pixel 2}) + x_3 \cdot (\text{pixel 3}) \dots x_{64} \cdot (\text{pixel 64})$$

One way we might imagine reducing the dimension of this data is to zero-out all but a few of these basis vectors. For example, if we use only the first eight pixels, we get an eight-dimensional projection of the data, but it is not very reflective of the whole image: we've thrown-out over 12% of the pixels!



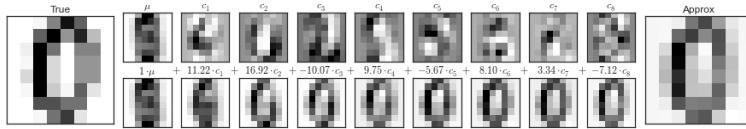
The upper row of panels shows the individual pixels, and the lower row shows the cumulative contribution of these pixels to the construction of the image. Using only eight of the pixel-basis components, we can only construct a small portion of the 64-pixel image. Were we to continue this sequence and 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 other basis functions, which each contain some pre-defined contribution from each pixel, and write something like

$$\text{image}(x) = \text{mean} + x_1 \cdot (\text{basis 1}) + x_2 \cdot (\text{basis 2}) + x_3 \cdot (\text{basis 3}) \dots$$

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 of the 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. Here is a similar depiction of reconstructing this digit using the mean plus the first eight PCA basis functions:

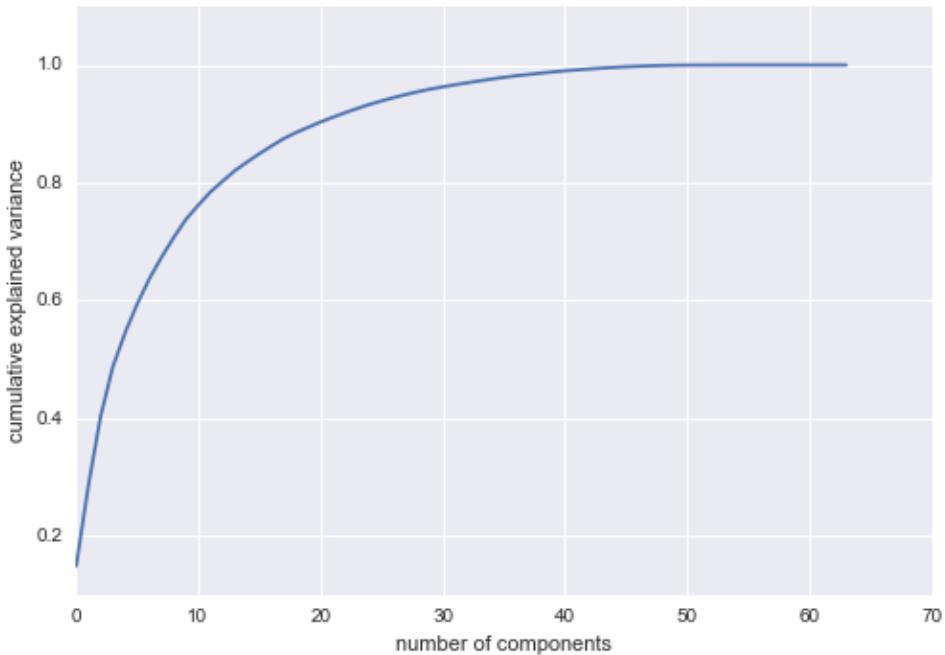


Unlike the pixel basis, the PCA basis allows us to recover the salient features of the input image with just a mean plus eight components! The amount of each pixel in each component is the corollary of the orientation of the vector in our 2D example. This is the sense in which PCA provides a low-dimensional representation of the data: it discovers a set of basis functions which 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. This can be determined by looking at the cumulative *explained variance ratio* as a function of the number of components:

```
pca = PCA().fit(digits.data)
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');
```



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 measured 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 you understand 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 components 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 input noise-free data:

```
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)

```



Now lets add some random noise to create a noisy dataset, and re-plot it:

```

np.random.seed(42)
noisy = np.random.normal(digits.data, 4)
plot_digits(noisy)

```



It's clear by eye that the images are noisy, and contain spurious pixels. Let's train a PCA on the noisy data, requesting that the projection preserve 50% of the variance:

```

pca = PCA(0.50).fit(noisy)
pca.n_components_

```

12

50% of the variance here amounts to twelve principal components. Now we compute these components, and then use the inverse of the transform to reconstruct the filtered digits:

```
components = pca.transform(noisy)
filtered = pca.inverse_transform(components)
plot_digits(filtered)
```



This signal preserving/noise filtering property makes PCA a very useful feature selection 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

In Section X.X, we showed an example of using a PCA projection as a feature selector for facial recognition with a support vector machine. Here we will take a look back and explore a bit more of what went into that. Recall that we were using the *Labeled Faces in the Wild* dataset made available through scikit-learn:

```
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 approximate the first N principal components much more quickly than the standard PCA estimator, and thus is very useful for high-dimensional data (here, a dimensionality of nearly 3000). We will take a look at the first 150 components:

```

from sklearn.decomposition import RandomizedPCA
pca = RandomizedPCA(150)
pca.fit(faces.data)

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 several principal components. These components are technically known as “eigenvectors”, so these types of images are often called “eigenfaces”, and they are as creepy as they sound:

```

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')

```

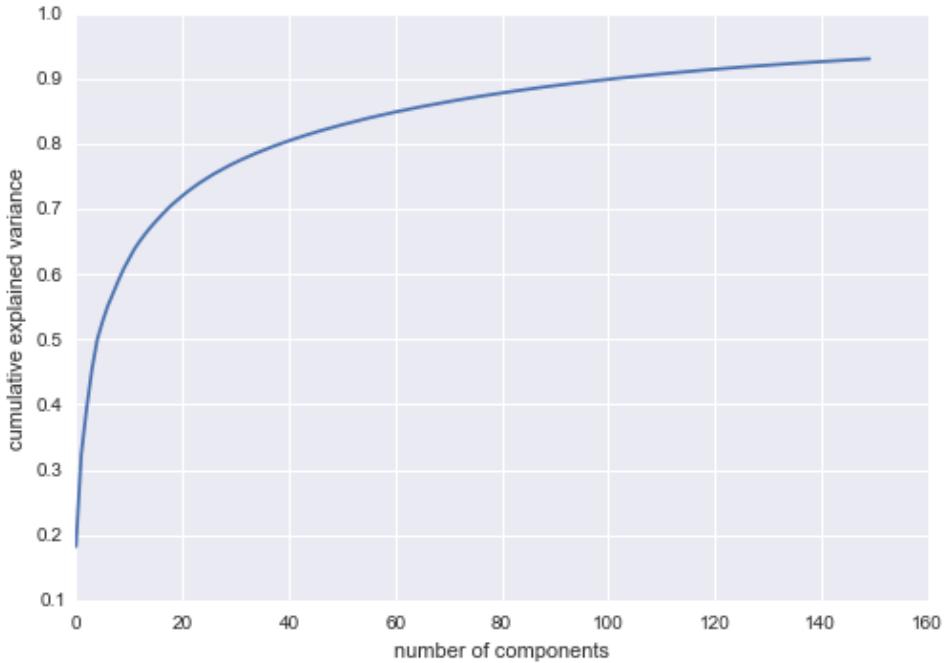


The results are very interesting, and give us insight into how the images vary: for example, the first few eigenfaces (from the top left) seem to be associated with the angle of lighting on the face, and later principal vectors seem to be picking-out certain features, like eyes, noses, and lips. Let's take a look at the cumulative variance of these components to see how much of the data information the projection is preserving:

```

plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance');

```



We see that the 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:

```
# Compute the components & projected faces
pca = RandomizedPCA(150).fit(faces.data)
components = pca.transform(faces.data)
projected = pca.inverse_transform(components)

# 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');
```



The top row here shows the input images, while the bottom row shows the reconstruction of the images from just 150 of the ~3000 initial features. This visualization makes clear why the PCA feature selection used in Section X.X 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 that our classification algorithm needs to be trained on 150-dimensional data rather than 3000 dimensional data which, depending on the particular algorithm we choose, can lead to a much more efficient classification.

Principal Component Analysis Summary

In this section we have discussed the use of Principal Component Analysis for dimensionality reduction, for visualization of high-dimensional data, for noise filtering, and for feature selection within high-dimensional images. Because of the versatility and interpretability of PCA, it has been shown to be effective in a wide variety of contexts and disciplines. Given any high-dimensional dataset, I tend to start with PCA in order to visualize the relationship between points (as we did with the digits above), to understand the main variance in the data (as we did with the eigen-faces above), and to understand the intrinsic dimensionality (by plotting the explained variance ratio). Certainly PCA is not useful for every high-dimensional dataset, but it offers a nice path to gaining insight into high-dimensional data.

A main weakness of PCA is that it tends to be highly affected by outliers in the data. For this reason, many robust variants of PCA have been developed, many of which act to iteratively discard datapoints which are poorly described by the initial components. Scikit-learn contains a couple interesting variants on PCA, including `RandomizedPCA` and `SparsePCA`, both also in the `sklearn.decomposition` submodule. `RandomizedPCA`, which we saw above, uses a non-deterministic method to quickly approximate the first few principal components in very high-dimensional data, while `SparsePCA` introduces a regularization term (see Section X.X) that serves to enforce sparsity of the components.

In the coming sections, we will look at other unsupervised learning methods which build on some of the ideas of PCA.

Figures

The following code is used to generate some of the inline figures above.

```
import os
if not os.path.exists('fig'):
    os.makedirs('fig')



### Principal Components Rotation


rng = np.random.RandomState(1)
X = np.dot(rng.rand(2, 2), rng.randn(2, 200)).T
pca = PCA(n_components=2, whiten=True)
pca.fit(X)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

# plot data
ax[0].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, ax=ax[0])
ax[0].axis('equal');
ax[0].set(xlabel='x', ylabel='y', title='input')

# plot principal components
X_pca = pca.transform(X)
ax[1].scatter(X_pca[:, 0], X_pca[:, 1], alpha=0.2)
draw_vector([0, 0], [0, 3], ax=ax[1])
draw_vector([0, 0], [3, 0], ax=ax[1])
ax[1].axis('equal')
ax[1].set(xlabel='component 1', ylabel='component 2',
           title='principal components')

fig.savefig('fig/07.09-PCA-rotation.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

Digits Pixel Components

```
def plot_pca_components(x, coefficients=None, mean=0, components=None,
                        imshape=(8, 8), n_components=8, fontsize=12,
                        show_mean=True):
    if coefficients is None:
        coefficients = x

    if components is None:
        components = np.eye(len(coefficients), len(x))

    mean = np.zeros_like(x) + mean
```

```

fig = plt.figure(figsize=(1.2 * (5 + n_components), 1.2 * 2))
g = plt.GridSpec(2, 4 + bool(show_mean) + n_components, hspace=0.3)

def show(i, j, x, title=None):
    ax = fig.add_subplot(g[i, j], xticks=[], yticks[])
    ax.imshow(x.reshape(imshape), interpolation='nearest')
    if title:
        ax.set_title(title, fontsize=fontsize)

show(slice(2), slice(2), x, "True")

approx = mean.copy()

counter = 2
if show_mean:
    show(0, 2, np.zeros_like(x) + mean, r'$\mu$')
    show(1, 2, approx, r'$1 \cdot \mu$')
    counter += 1

for i in range(n_components):
    approx = approx + coefficients[i] * components[i]
    show(0, i + counter, components[i], r'$c_{\theta}$'.format(i + 1))
    show(1, i + counter, approx,
         r"${:0.2f} \cdot c_{1}$".format(coefficients[i], i + 1))
    if show_mean or i > 0:
        plt.gca().text(0, 1.05, '$+$', ha='right', va='bottom',
                       transform=plt.gca().transAxes, fontsize=fontsize)

show(slice(2), slice(-2, None), approx, "Approx")
return fig

digits = load_digits()
sns.set_style('white')

fig = plot_pca_components(digits.data[10],
                           show_mean=False)

fig.savefig('fig/07.09-digits-pixel-components.png')
plt.close(fig)

```

Digits PCA Components

```

pca = PCA(n_components=8)
Xproj = pca.fit_transform(digits.data)
sns.set_style('white')
fig = plot_pca_components(digits.data[10], Xproj[10],
                           pca.mean_, pca.components_)

fig.savefig('fig/07.09-digits-pca-components.png')
plt.close(fig)

```

In-Depth: Manifold Learning

We have seen how Principal Component Analysis (PCA) can be used in the dimensionality reduction task – reducing the number of features of a dataset while maintaining the essential relationships between the points. While PCA is flexible, fast, and easily interpretable, it does not perform so well when there are **nonlinear** relationships within the data – we will see some examples of these below.

To address this deficiency, we can turn to a class of methods known as *manifold learning* – a class of unsupervised estimators that seek to describe datasets as low-dimensional manifolds embedded in high-dimensional spaces. When you think of a manifold, I'd suggest imagining a sheet of paper: this is a two-dimensional object which lives in our familiar three-dimensional world, and can be bent or rolled in that two dimensions. In the parlance of manifold learning, we can think of this sheet is a two-dimensional manifold embedded in three-dimensional space.

Rotating, re-orienting, or stretching the piece of paper in three-dimensional space doesn't change the flat geometry of the paper: such operations are akin to linear embeddings. If you bend, curl, or crumple the paper, it is still a two-dimensional manifold, but the embedding into the three-dimensional space is no longer linear. Manifold learning algorithms would seek to learn about the fundamental two-dimensional nature of the paper, even as it is contorted to fill the three-dimensional space.

Here we will demonstrate a number of manifold methods, going most deeply into a couple techniques: Multidimensional Scaling (MDS), Locally Linear Embedding (LLE), and Isometric Mapping (IsoMap).

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Manifold Learning: “HELLO”

To make these concepts more clear, let's start by generating some two-dimensional data that we can use to define a manifold. Here is a function that will create data in the shape of the word, “HELLO”:

```
def make_hello(N=1000, rseed=42):
    # Make a plot with "HELLO" text; save as png
    fig, ax = plt.subplots(figsize=(4, 1))
    fig.subplots_adjust(left=0, right=1, bottom=0, top=1)
    ax.axis('off')
    ax.text(0.5, 0.4, 'HELLO', va='center', ha='center', weight='bold', size=85)
    fig.savefig('hello.png')
    plt.close(fig)
```

```

# Open this PNG and draw random points from it
from matplotlib.image import imread
data = imread('hello.png')[::-1, :, 0].T
rng = np.random.RandomState(rseed)
X = rng.rand(4 * N, 2)
i, j = (X * data.shape).astype(int).T
mask = (data[i, j] < 1)
X = X[mask]
X[:, 0] *= (data.shape[0] / data.shape[1])
X = X[:N]
return X[np.argsort(X[:, 0])]

```

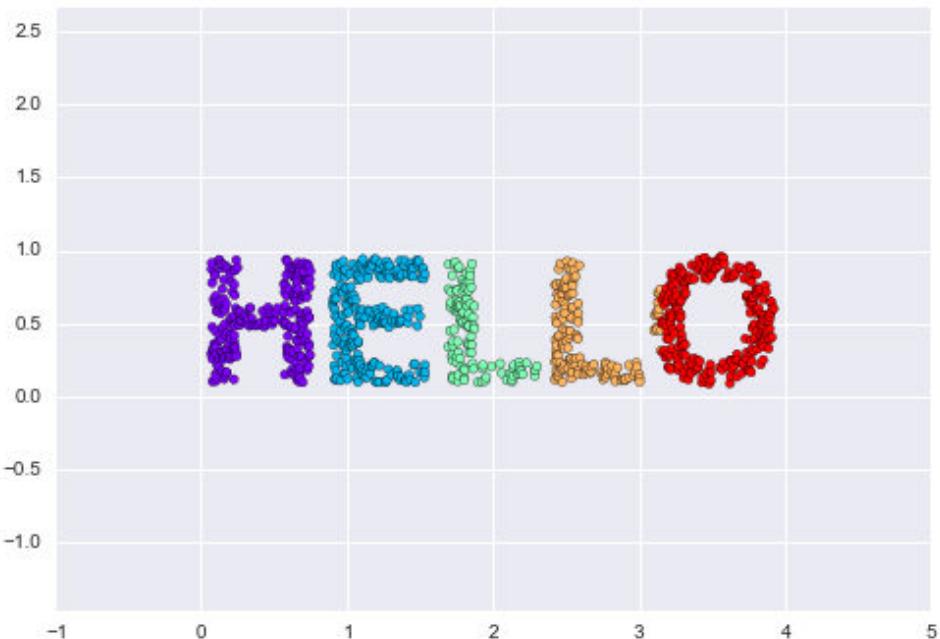
Let's create some data and see what it looks like:

```

X = make_hello(1000)
colorize = dict(c=X[:, 0], cmap=plt.cm.get_cmap('rainbow', 5))
plt.scatter(X[:, 0], X[:, 1], **colorize)
plt.axis('equal');

/Users/jakevdः/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



The output is two-dimensional, and consists of points drawn in the shape of the word, “HELLO”. This data form will help us to see visually what these algorithms are doing.

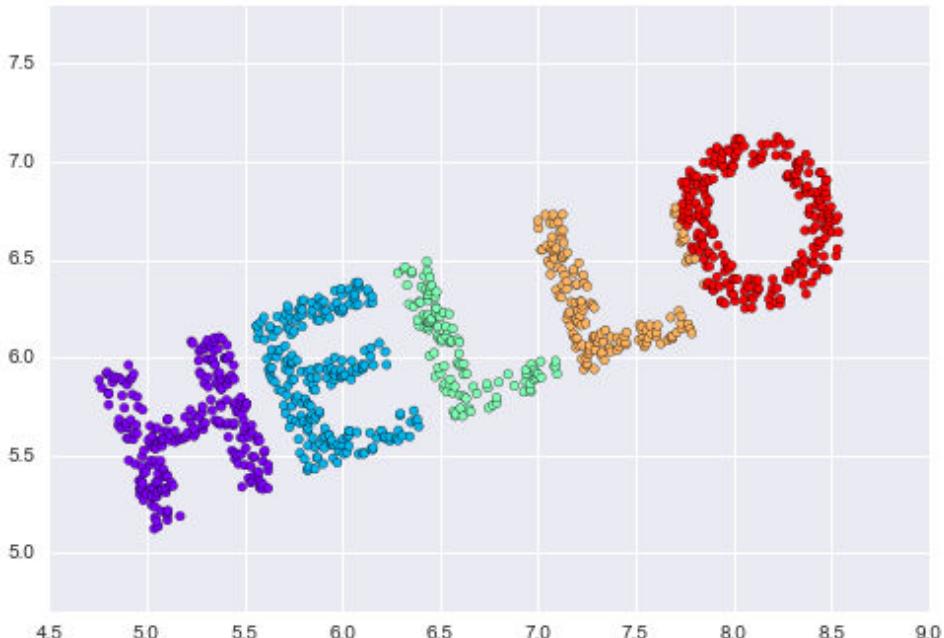
Multidimensional Scaling (MDS)

Looking at data like this, we can see that the particular choice of x and y values of the dataset are not the most fundamental description of the data: we can scale, shrink, or rotate the data, and the “HELLO” will still be apparent. For example, if we use a rotation matrix to rotate the data, the x and y values change, but the data is still fundamentally the same:

```
def rotate(X, angle):
    theta = np.deg2rad(angle)
    R = [[np.cos(theta), np.sin(theta)],
         [-np.sin(theta), np.cos(theta)]]
    return np.dot(X, R)

X2 = rotate(X, 20) + 5
plt.scatter(X2[:, 0], X2[:, 1], **colorize)
plt.axis('equal');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



This tells us that the x and y values are not necessarily fundamental to the relationships in the data. What *is* fundamental, in this case, is the *distance* between each point and the other points in the dataset. A common way to represent this is to use a distance matrix: for N points, we construct an $N \times N$ array such that entry (i, j) contains

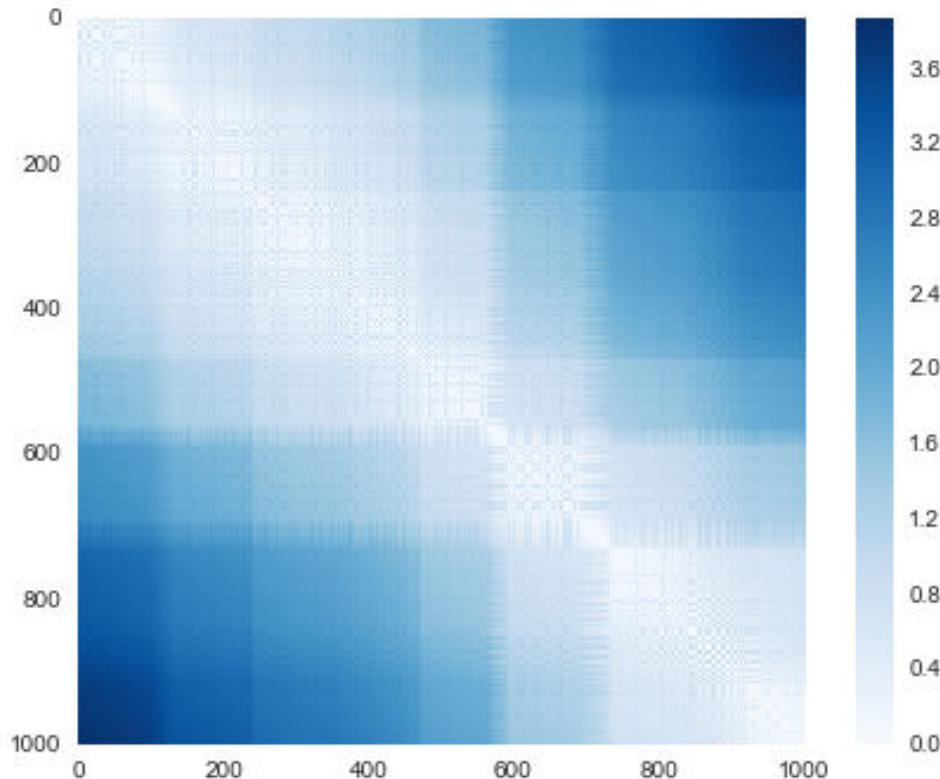
the distance between point i and point j . Let's use scikit-learn's efficient `pairwise_distances` function to do this for our original data:

```
from sklearn.metrics import pairwise_distances
D = pairwise_distances(X)
D.shape
(1000, 1000)
```

As promised, for our $N=1000$ points, we obtain a 1000×1000 matrix, which can be visualized like this:

```
plt.imshow(D, zorder=2, cmap='Blues', interpolation='nearest')
plt.colorbar();
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
  if self._edgecolors == str('face'):
```



If we similarly construct a distance matrix for our rotated and translated data, we see that it is the same:

```
D2 = pairwise_distances(X2)
np.allclose(D, D2)
```

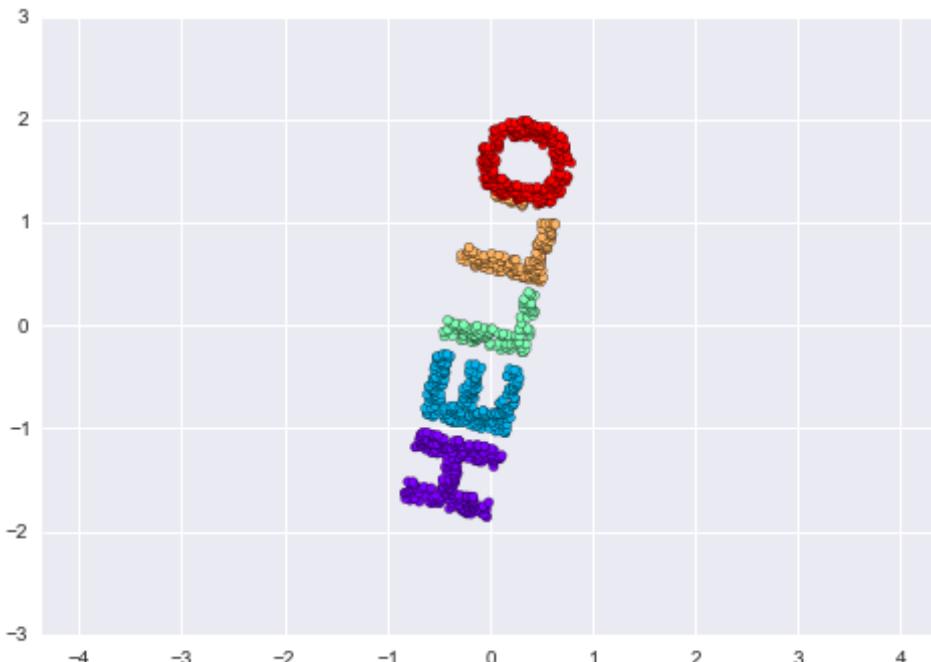
True

This distance matrix gives us a representation of our data that is invariant to rotations and translations, but the visualization of the matrix above is not entirely intuitive. In this representation, we have lost any visible sign of the interesting structure in the data: the “HELLO” that we saw above.

Further, while computing this distance matrix from the (x, y) coordinates is straightforward, transforming the distances back into x and y coordinates is rather difficult. This is exactly what the Multidimensional Scaling algorithm aims to do: given a distance matrix between points, it recovers a D -dimensional coordinate representation of the data. Let’s see how it works for our distance matrix, using the precomputed dissimilarity to specify that we are passing a distance matrix:

```
from sklearn.manifold import MDS
model = MDS(n_components=2, dissimilarity='precomputed', random_state=1)
out = model.fit_transform(D)
plt.scatter(out[:, 0], out[:, 1], **colorize)
plt.axis('equal');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



The MDS algorithm recovers one of the possible 2-dimensional coordinate representations of our data, using *only* the $N \times N$ distance matrix describing the relationship between the data points.

MDS as Manifold Learning

The usefulness of this becomes more apparent when we consider the fact that distance matrices can be computed from data in *any* dimension. So, for example, instead of simply rotating the data in the two-dimensional plane, we can project it into three dimensions using the following function (essentially a 3D generalization of the rotation matrix used above):

```
def random_projection(X, dimension=3, rseed=42):
    assert dimension >= X.shape[1]
    rng = np.random.RandomState(rseed)
    C = rng.randn(dimension, dimension)
    e, V = np.linalg.eigh(np.dot(C, C.T))
    return np.dot(X, V[:X.shape[1]])
```

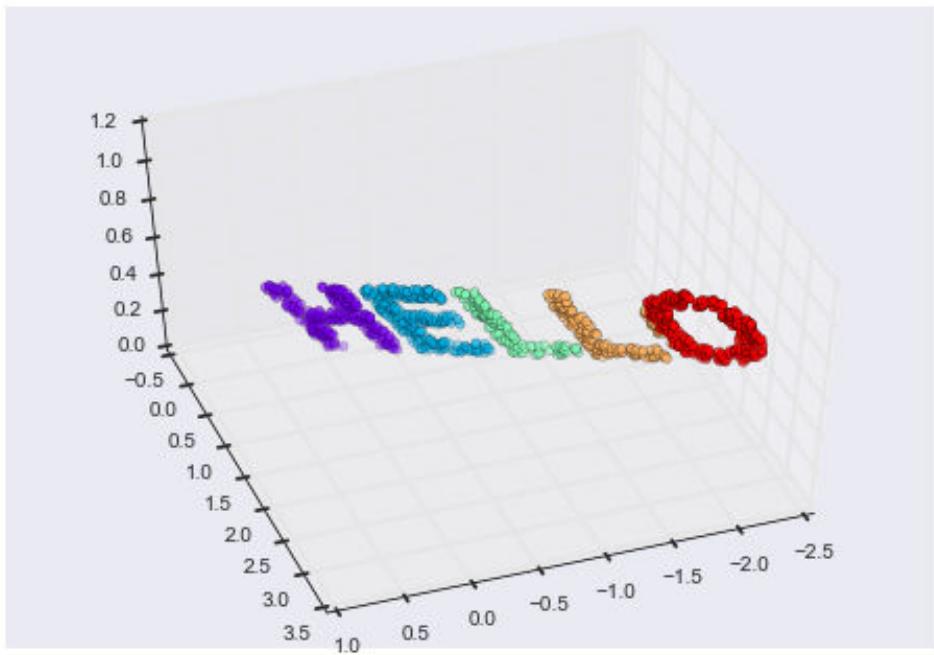


```
X3 = random_projection(X, 3)
X3.shape
(1000, 3)
```

Let's visualize these points to see what we're working with:

```
from mpl_toolkits import mplot3d
ax = plt.axes(projection='3d')
ax.scatter3D(X3[:, 0], X3[:, 1], X3[:, 2],
             **colorize)
ax.view_init(azim=70, elev=50)

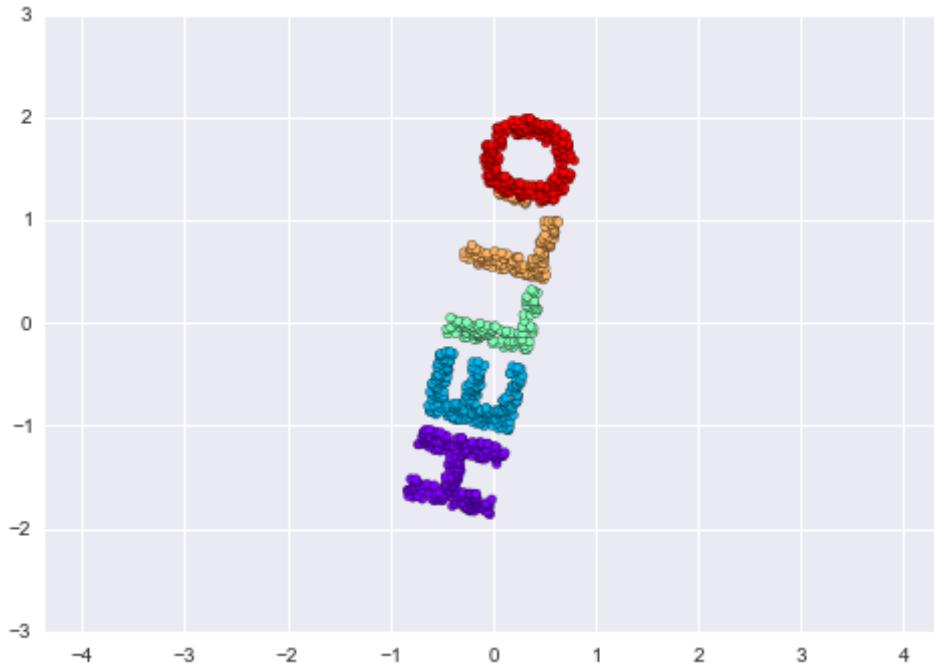
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



We can now ask the MDS estimator to input this three-dimensional data, compute the distance matrix, and then determine the optimal two-dimensional embedding for this distance matrix. The result recovers a representation of the original data:

```
model = MDS(n_components=2, random_state=1)
out3 = model.fit_transform(X3)
plt.scatter(out3[:, 0], out3[:, 1], **colorize)
plt.axis('equal');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
  if self._edgecolors == str('face'):
```



This is essentially the goal of a manifold learning estimator: given high-dimensional embedded data, it seeks a low-dimensional representation of the data that preserves certain relationships within the data. In the case of MDS, the quantity preserved is the distance between every pair of points.

Nonlinear Embeddings: Where MDS Fails

Above we have been considering *linear* embeddings, which essentially consist of rotations, translations, and scalings of data into higher-dimensional spaces. Where MDS breaks down is when the embedding is non-linear: i.e. when it goes beyond this simple set of operations. Consider the following embedding, which takes the input and contorts it into an “S” shape in three dimensions:

```
def make_hello_s_curve(X):
    t = (X[:, 0] - 2) * 0.75 * np.pi
    x = np.sin(t)
    y = X[:, 1]
    z = np.sign(t) * (np.cos(t) - 1)
    return np.vstack((x, y, z)).T

XS = make_hello_s_curve(X)
```

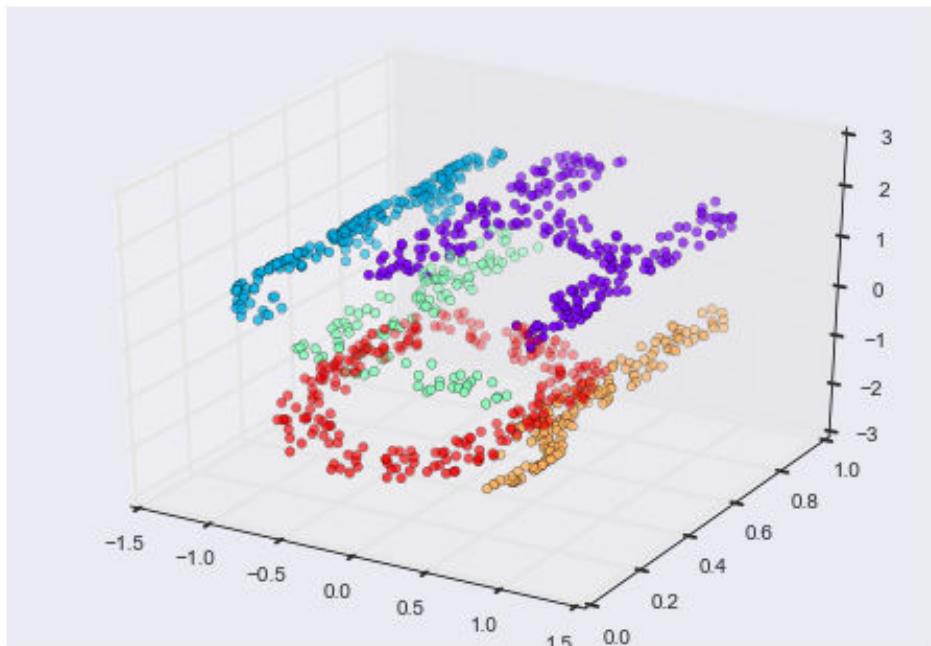
This is again 3D data, but we can see that the embedding is much more complicated:

```

from mpl_toolkits import mplot3d
ax = plt.axes(projection='3d')
ax.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2],
             **colorize);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



The fundamental relationships between the data points are still there, but this time the data has been transformed in a nonlinear way: it has been wrapped-up into the shape of an “S”.

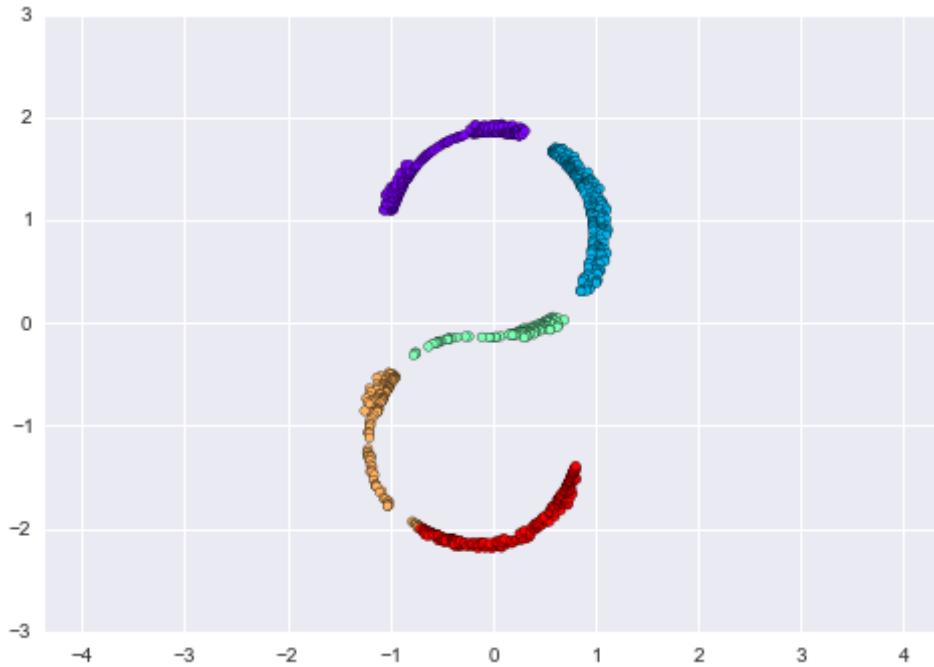
If we try a simple MDS algorithm on this data, it is not able to “unwrap” this nonlinear embedding, and we lose track of the fundamental relationships in the embedded manifold:

```

from sklearn.manifold import MDS
model = MDS(n_components=2, random_state=2)
outS = model.fit_transform(XS)
plt.scatter(outS[:, 0], outS[:, 1], **colorize)
plt.axis('equal');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



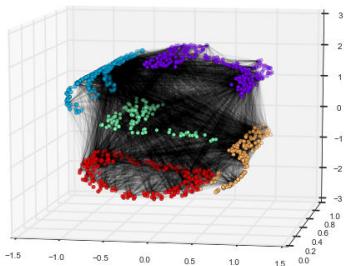
The best two-dimensional **linear** embedding does not unwrap the S-curve, but instead throws-out the original y-axis.

Nonlinear Manifolds: Locally Linear Embedding

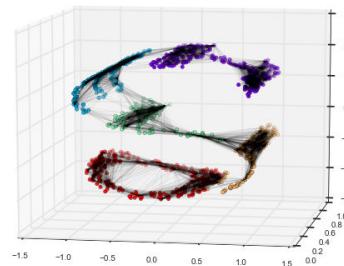
How can we move forward here? Stepping back, we can see that the source of the problem is that MDS tries to preserve distances between faraway points when constructing the embedding. But what if we instead modified the algorithm such that it only preserves distances between nearby points? The resulting embedding would be closer to what we want.

Visually, we can think of it this way:

MDS Linkages



LLE Linkages (100 NN)



Here each faint line represents a distance that should be preserved in the embedding. On the left is a representation of the model used by MDS: it tries to preserve the distances between each pair of points in the dataset. On the right is a representation of the model used by a manifold learning algorithm called Locally Linear Embedding (LLE): rather than preserving *all* distances, it instead tries to preserve only the distances between *neighboring points*: in this case, the nearest 100 neighbors of each point.

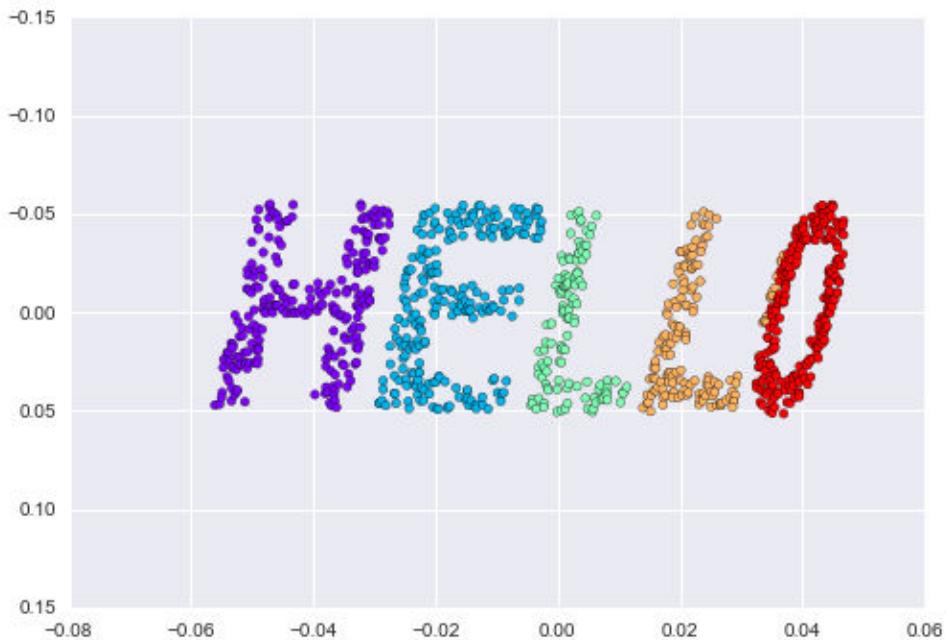
Thinking about the left panel, we can see why MDS fails: there is no way to flatten this data while adequately preserving the length of every line drawn between the two points. For the right panel, on the other hand, things look a bit more optimistic. We could imagine unrolling the data in a way that keeps the lengths of the lines approximately the same. This is precisely what LLE does, through a global optimization of a cost function reflecting this logic.

LLE comes in a number of flavors; here we will use the *Modified LLE* algorithm to recover the embedded 2-dimensional manifold. In general, modified LLE does better than other flavors of the algorithm at recovering well-defined manifolds with very little distortion:

```
from sklearn.manifold import LocallyLinearEmbedding
model = LocallyLinearEmbedding(n_neighbors=100, n_components=2, method='modified',
                               eigen_solver='dense')
out = model.fit_transform(XS)

fig, ax = plt.subplots()
ax.scatter(out[:, 0], out[:, 1], **colorize)
ax.set_xlim(0.15, -0.15);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



The result remains somewhat distorted compared to our original manifold, but captures the essential relationships in the data!

Some Thoughts on Manifold Methods

Though the above story and motivation is compelling, in practice manifold learning techniques tend to be finicky enough that they are rarely used for anything more than simple qualitative visualization of high-dimensional data.

Some of the particular challenges of manifold learning, which all contrast poorly with PCA:

- In manifold learning, there is no good framework for handling missing data. In contrast, there are straightforward iterative approaches for missing data in PCA.
- In manifold learning, the presence of noise in the data can “short-circuit” the manifold and drastically change the embedding. In contrast, PCA naturally filters noise from the most important components.
- The manifold embedding result is generally highly dependent on the number of neighbors chosen, and there is generally no solid quantitative way to choose an optimal number of neighbors. In contrast, PCA does not involve such a choice.
- In manifold learning, the globally-optimal number of output dimensions is difficult to determine. In contrast, PCA lets you determine the output dimension based on the explained variance.

- In manifold learning, the meaning of the embedded dimensions is not always clear. In PCA, the principal components have a very clear meaning.
- In manifold learning the computational expense of manifold methods scales as $O[N^2]$ or $O[N^3]$. For PCA, there exist randomized approaches that are generally much faster.

With all that on the table, the only clear advantage of manifold learning methods over PCA is their ability to preserve nonlinear relationships in the data; for that reason I tend to explore data with manifold methods only after first exploring them with PCA.

Scikit-learn implements several common variants of manifold learning beyond Isomap and LLE: the scikit-learn documentation has a [nice discussion and comparison of them](#). Based on my own experience, I would give the following recommendations:

- For toy problems such as the S-curve we saw above, Locally Linear Embedding (LLE) and its variants (especially *modified LLE*), perform very well. This is implemented in `sklearn.manifold.LocallyLinearEmbedding`.
- For high dimensional data from real-world sources, LLE often produces poor results, and Isometric Mapping (IsoMap) seems to generally lead to more meaningful embeddings. This is implemented in `sklearn.manifold.Isomap`
- For data which is highly clustered, *t-distributed Stochastic Neighbor Embedding* (t-SNE) seems to work very well, though can be very slow compared to other methods. This is implemented in `sklearn.manifold.TSNE`.

If you're interested in getting a feel for how these work, I'd suggest running each of the methods on the data in this section.

Example: Isomap on Faces

One place manifold learning is often used is in understanding the relationship between high-dimensional data points. A common case of high-dimensional data is images: for example, a set of images with 1000 pixels each can be thought of as collection of points in 1000 dimensions – the brightness of each pixel in each image defines the coordinate in that dimension.

Here let's apply **Isomap** on some faces data. We will use the *labeled faces in the wild* dataset, which we previously saw in Section X.X. Running this command will download the data and cache it in your home directory for later use:

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=30)
faces.data.shape
(2370, 2914)
```

We have 2370 images, each with 2914 pixels. In other words, the images can be thought of as data points in a 2914-dimensional space!

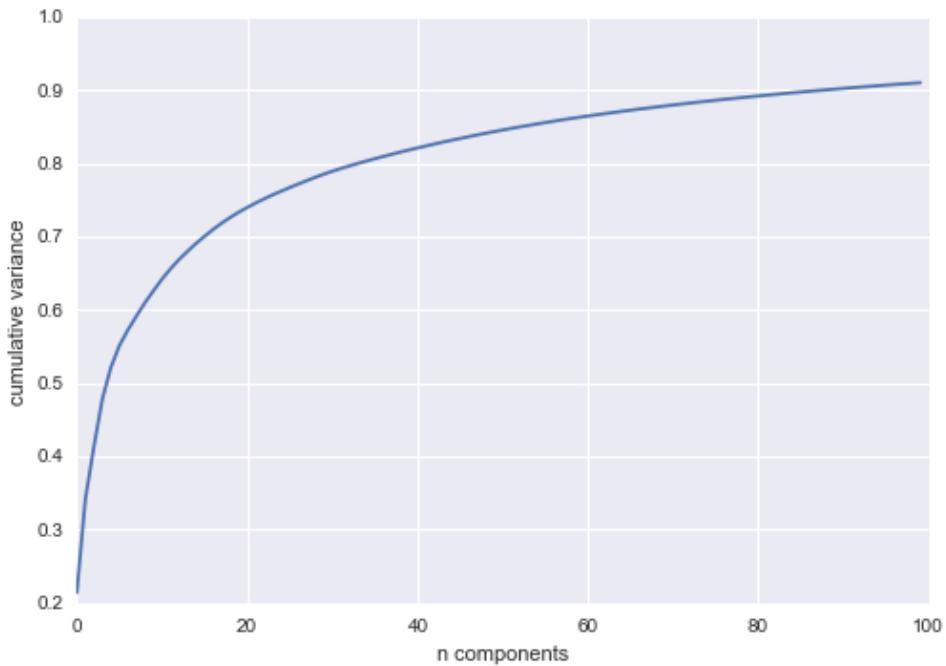
Let's quickly visualize several of these images to see what we're working with:

```
fig, ax = plt.subplots(4, 8, subplot_kw=dict(xticks=[], yticks=[]))
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='gray')
```



We would like to plot a low-dimensional embedding of the 2914-dimensional data to learn the fundamental relationships between the images. One useful way to start is to compute a PCA, and examine the explained variance ratio, which will give us an idea of how many linear features are required to describe the data:

```
from sklearn.decomposition import RandomizedPCA
model = RandomizedPCA(100).fit(faces.data)
plt.plot(np.cumsum(model.explained_variance_ratio_))
plt.xlabel('n components')
plt.ylabel('cumulative variance');
```



We see that for this data, nearly 100 components are required to preserve 90% of the variance: this tells us that the data is intrinsically very high dimensional – it can't be described linearly with just a few components.

When this is the case, nonlinear manifold embeddings like LLE and Isomap can be helpful. We can compute an Isomap embedding on these faces using the same pattern shown above:

```
from sklearn.manifold import Isomap
model = Isomap(n_components=2)
proj = model.fit_transform(faces.data)
proj.shape
```

(2370, 2)

The output is a two-dimensional projection of all the input images. To get a better idea of what the projection tells us, let's define a function which will output image thumbnails at the locations of the projections:

```
from matplotlib import offsetbox

def plot_components(data, model, images=None, ax=None,
                    thumb_frac=0.05, cmap='gray'):
    ax = ax or plt.gca()

    proj = model.fit_transform(data)
```

```

ax.plot(proj[:, 0], proj[:, 1], '.k')

if images is not None:
    min_dist_2 = (thumb_frac * max(proj.max(0) - proj.min(0))) ** 2
    shown_images = np.array([2 * proj.max(0)])
    for i in range(data.shape[0]):
        dist = np.sum((proj[i] - shown_images) ** 2, 1)
        if np.min(dist) < min_dist_2:
            # don't show points that are too close
            continue
        shown_images = np.vstack([shown_images, proj[i]])
    imagebox = offsetbox.AnnotationBbox(
        offsetbox.OffsetImage(images[i], cmap=cmap),
        proj[i])
    ax.add_artist(imagebox)

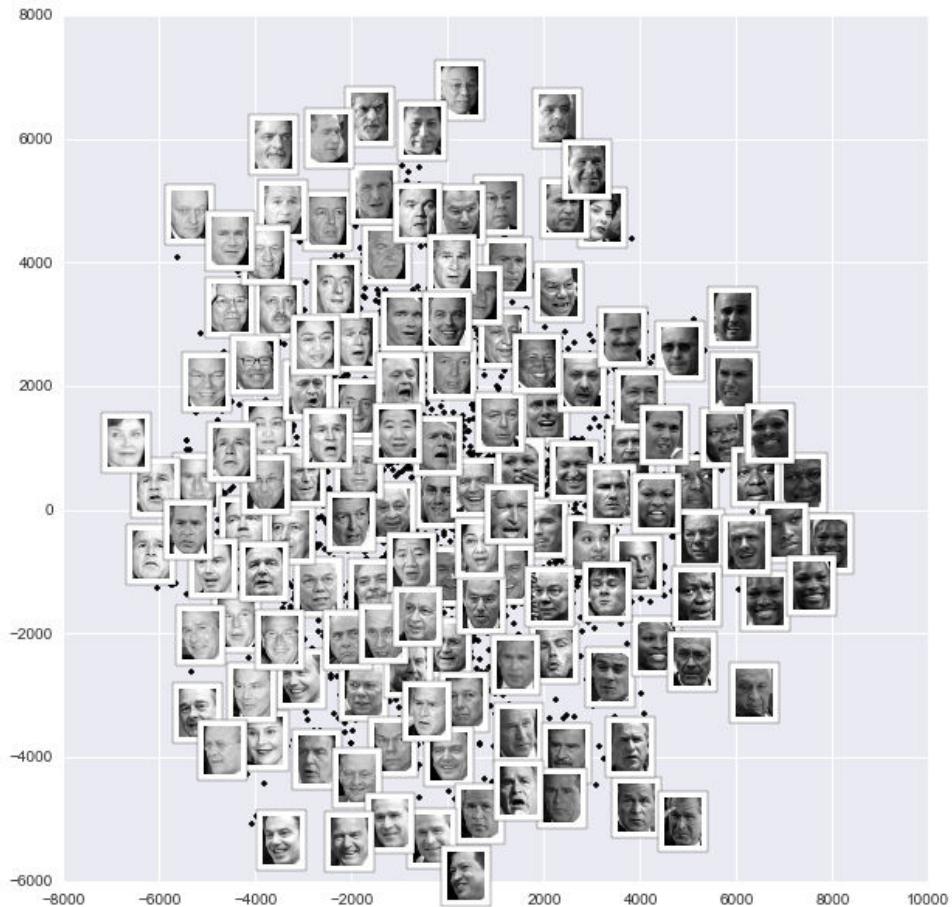
```

Calling this function now, we see the result:

```

fig, ax = plt.subplots(figsize=(10, 10))
plot_components(faces.data,
                 model=Isomap(n_components=2),
                 images=faces.images[:, ::2, ::2])

```



The result is interesting: the first two Isomap dimensions seem to describe global image features: the overall darkness or lightness of the image from left to right, and the general orientation of the face from bottom to top. This gives us a nice visual indication of some of the fundamental features in our data.

We could then go on to classify this data – perhaps using manifold features as inputs to the classification algorithm – as we did in Section X.X.

Example: Visualizing Structure in Digits

As another example of using manifold learning for visualization, let's take a look at the MNIST handwritten digits set. These data are similar to the digits we saw in Section X.X, but with many more pixels per image. They can be downloaded from <http://mldata.org/> with the scikit-learn utility:

```

from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
mnist.data.shape
(70000, 784)

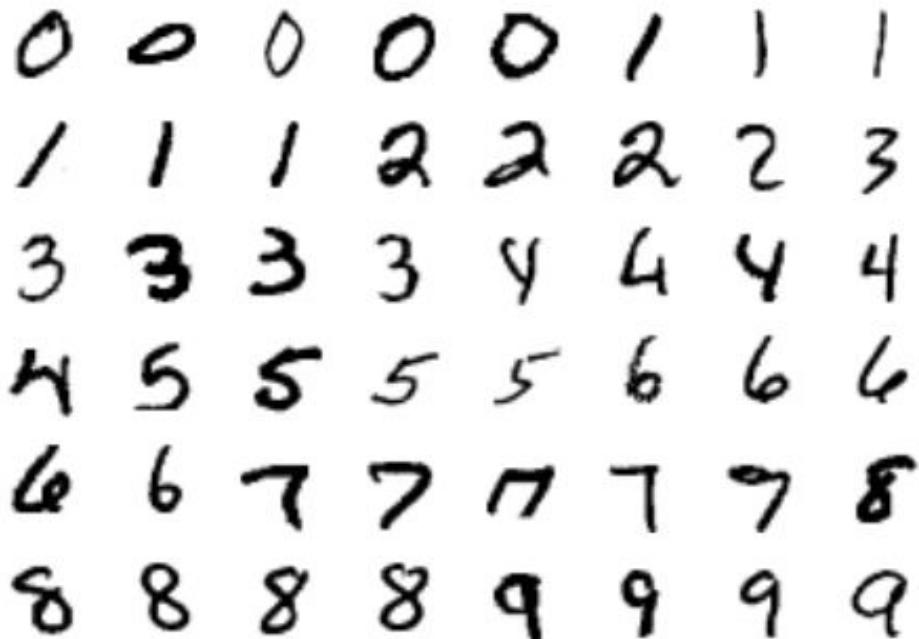
```

This consists of 70,000 images, each with 784 pixels (i.e. the images are 28 x 28). As above, we can take a look at the first few images:

```

fig, ax = plt.subplots(6, 8, subplot_kw=dict(xticks=[], yticks=[]))
for i, axi in enumerate(ax.flat):
    axi.imshow(mnist.data[1250 * i].reshape(28, 28), cmap='gray_r')

```



This gives us an idea of the variety of handwriting styles in the dataset.

Let's compute a manifold learning projection across the data. For speed here, we'll only use 1/30 of the data (that is, about ~2000 points). Because of the relatively poor scaling of manifold learning, I find that a few thousand samples is a good number to start with for relatively quick exploration before moving to a full calculation.

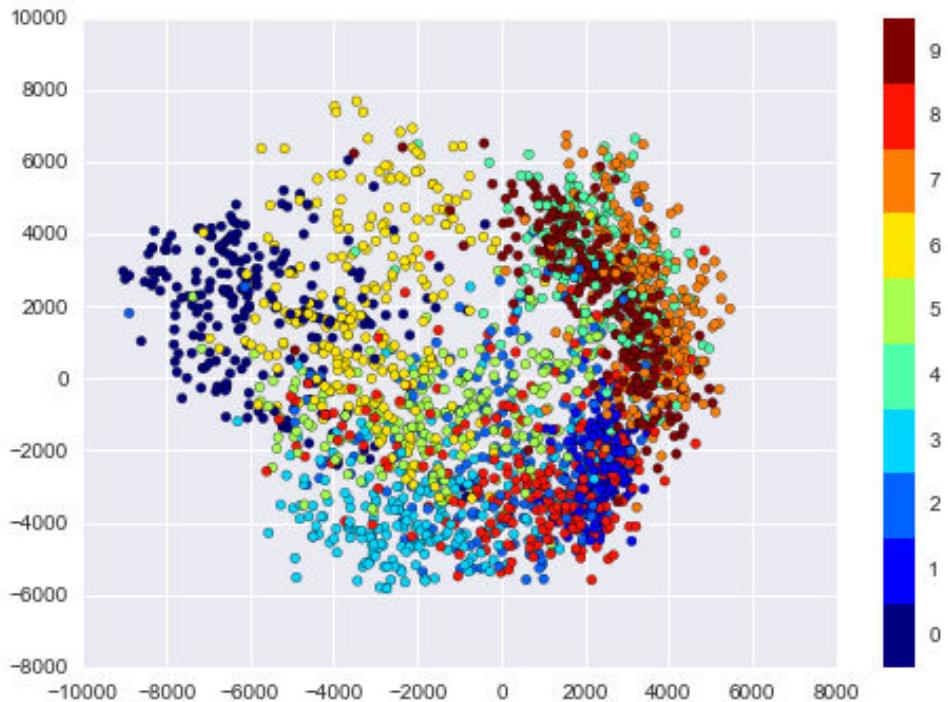
```

# use only 1/30 of the data: full dataset takes a long time!
data = mnist.data[::30]
target = mnist.target[::30]

model = Isomap(n_components=2)
proj = model.fit_transform(data)
plt.scatter(proj[:, 0], proj[:, 1], c=target, cmap=plt.cm.get_cmap('jet', 10))

```

```
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5);
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

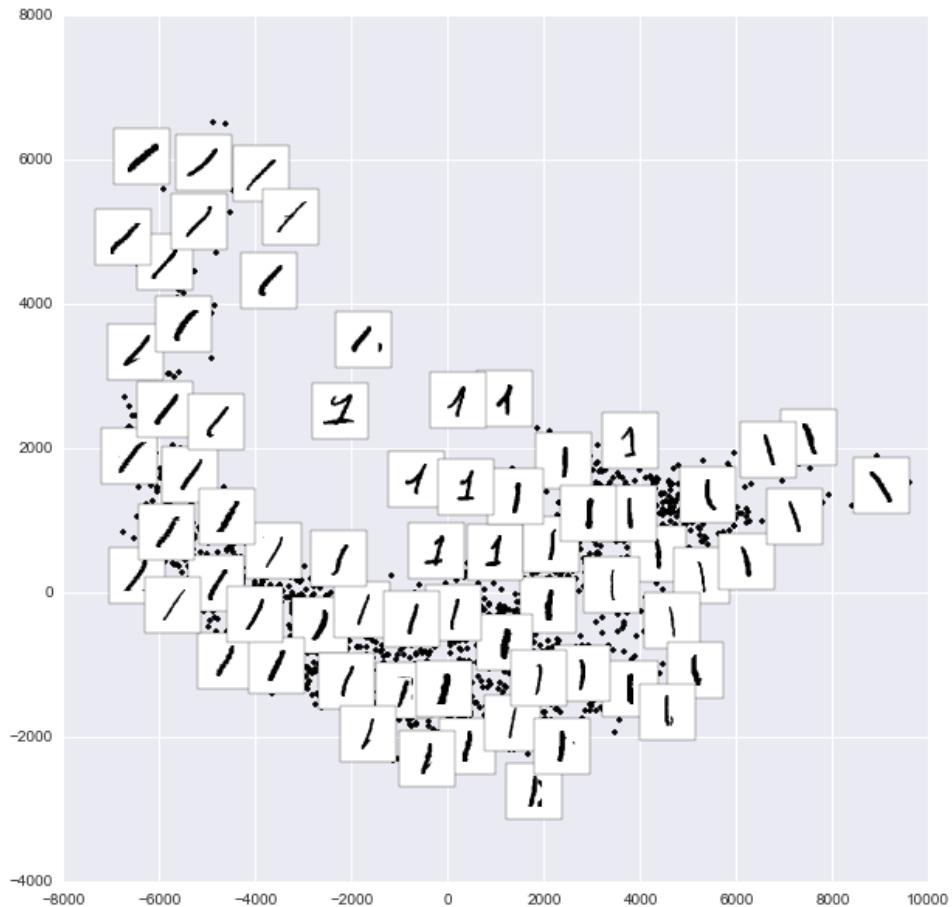


The resulting scatter-plot shows some of the relationships between the data points, but is a bit crowded. We can gain more insight by looking at just a single number at a time:

```
from sklearn.manifold import Isomap

# Choose 1/4 of the "1" digits to project
data = mnist.data[mnist.target == 1][:4]

fig, ax = plt.subplots(figsize=(10, 10))
model = Isomap(n_neighbors=5, n_components=2, eigen_solver='dense')
plot_components(data, model, images=data.reshape((-1, 28, 28)),
                 ax=ax, thumb_frac=0.05, cmap='gray_r')
```



The result gives you an idea of the variety of forms that the number “1” can take within the dataset. The data lie along a broad curve in the projected space, which appears to trace the orientation of the digit. As you move up the plot, you find ones which have hats and/or bases, though these are very sparse within the dataset. The projection lets us pick-out outliers which have data issues: e.g. pieces of the neighboring digits which snuck into the extracted images.

Now, this in itself may not be useful for the task of classifying digits, but it does help us get an understanding of the data, and may give us ideas about how to move forward, such as how we might want to clean the data before building a classification pipeline.

Figures

The following code produces some of the figures seen above:

```

import os
if not os.path.exists('fig'):
    os.makedirs('fig')

LLE vs MDS

from mpl_toolkits.mplot3d.art3d import Line3DCollection
from sklearn.neighbors import NearestNeighbors

# construct lines for MDS
rng = np.random.RandomState(42)
ind = rng.permutation(len(X))
lines_MDS = [(XS[i], XS[j]) for i in ind[:100] for j in ind[100:200]]

# construct lines for LLE
nbrs = NearestNeighbors(n_neighbors=100).fit(XS).kneighbors(XS[ind[:100]])[1]
lines_LLE = [(XS[ind[i]], XS[j]) for i in range(100) for j in nbrs[i]]
titles = ['MDS Linkages', 'LLE Linkages (100 NN)']

# plot the results
fig, ax = plt.subplots(1, 2, figsize=(16, 6),
                      subplot_kw=dict(projection='3d', axisbg='none'))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0, wspace=0)

for axi, title, lines in zip(ax, titles, [lines_MDS, lines_LLE]):
    axi.scatter3D(XS[:, 0], XS[:, 1], XS[:, 2], **colorize);
    axi.add_collection(Line3DCollection(lines, lw=1, color='black',
                                         alpha=0.05))
    axi.view_init(elev=10, azim=-80)
    axi.set_title(title, size=18)

fig.savefig('fig/07.10-LLE-vs-MDS.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:650:
    if self._edgecolors_original != str('face'):

```

In Depth: K-Means Clustering

In the previous few sections, we have explored one category of unsupervised machine learning models: dimensionality reduction. Here we will move on to another class of unsupervised machine learning models: clustering algorithms. Clustering algorithms seek to learn, from the properties of the data, an optimal division or discrete labeling of groups of points.

Many clustering algorithms are available in Scikit-learn and elsewhere, but perhaps the simplest to understand is an algorithm known as *K-Means Clustering*, which is implemented in `sklearn.cluster.KMeans`.

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set() # for plot styling
import numpy as np
```

Introducing K-Means

The K Means algorithm searches for a pre-determined number of clusters within an unlabeled multidimensional dataset. It accomplishes this using a simple conception of what the optimal clustering looks like, using the following assertions:

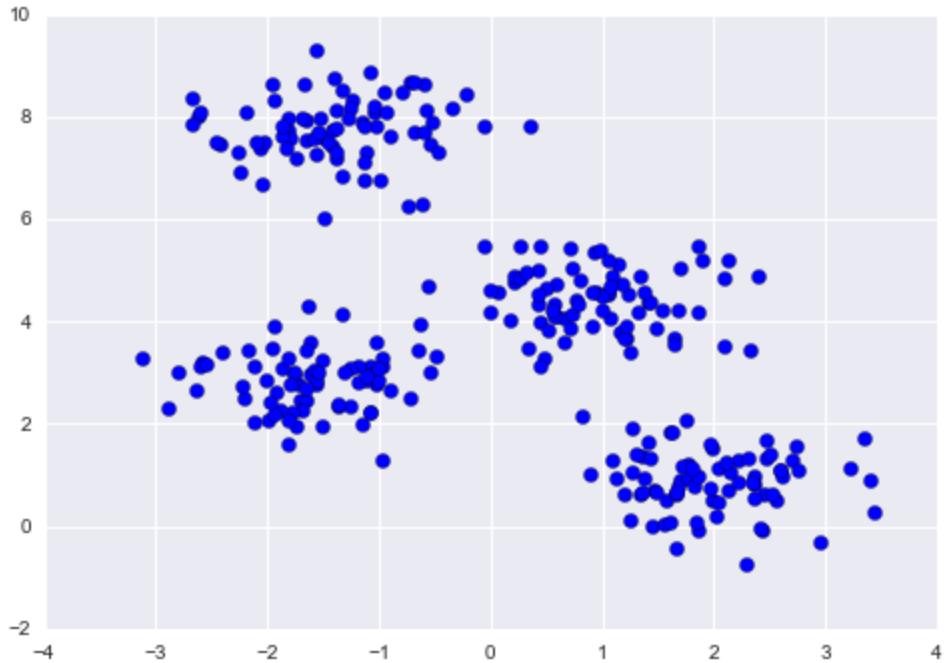
1. The “cluster center” is the arithmetic mean of all the points belonging to the cluster.
2. Each point is closer to its own cluster center than to other cluster centers.

Those two assertions form the K-Means model. Below we will dive-in to exactly *how* the algorithm reaches this solution, but for now let’s take a look at a simple dataset and see the K-Means result.

First, let’s generate a two-dimensional dataset containing four distinct blobs. To emphasize that this is an unsupervised algorithm, we will leave the labels out of the visualization

```
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=300, centers=4,
                      cluster_std=0.60, random_state=0)
plt.scatter(X[:, 0], X[:, 1], s=50);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



By eye, it is relatively easy to pick out the four clusters. The K-Means algorithm does this automatically. With scikit-learn, we can accomplish this using the typical estimator API:

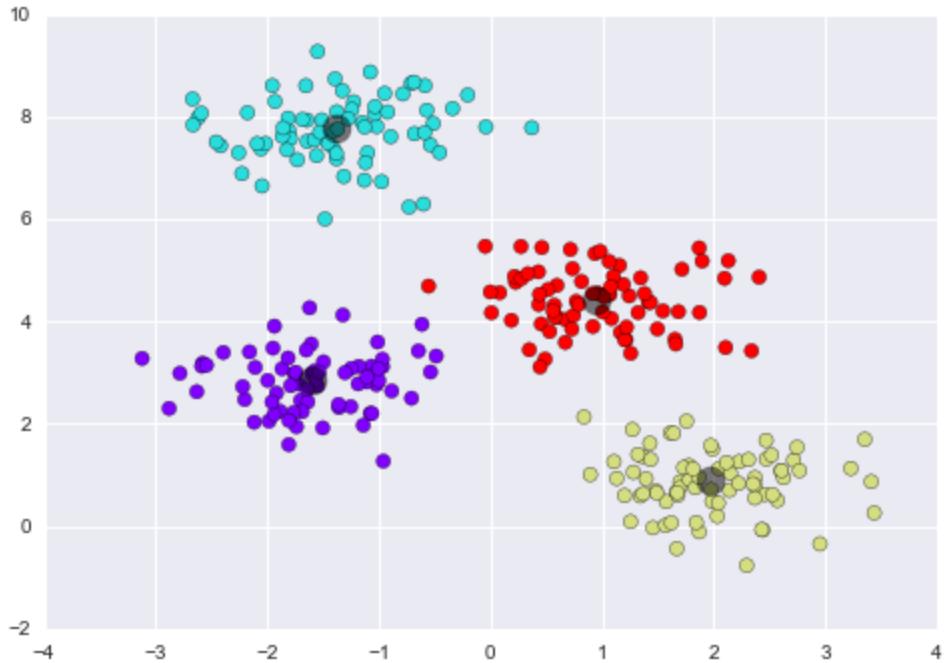
```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
```

Let's visualize the results by plotting the data colored by these labels. We will also plot the cluster centers as determined by the K-Means estimator:

```
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='rainbow')

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black', s=200, alpha=0.5);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



The good news is that the K-means algorithm (at least in this simple case) assigns the points to clusters very similarly to how we might assign them by-eye. But you might wonder how this algorithm finds these clusters so quickly! After all, the number of possible combinations of cluster assignments is exponential in the number of data points – an exhaustive search would be very, very costly. Fortunately for us, such an exhaustive search is not necessary: instead the typical approach to K Means involves an intuitive iterative approach known as *Expectation Maximization*.

K-Means Algorithm: Expectation Maximization

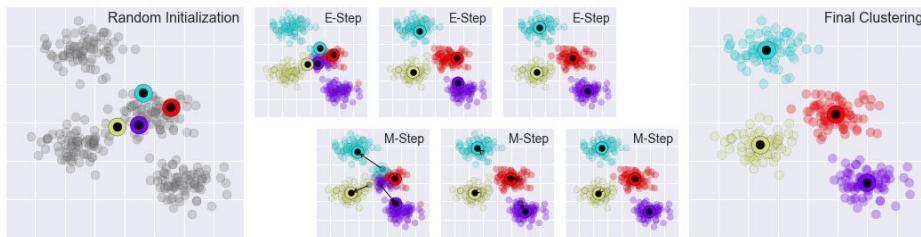
Expectation Maximization (E-M) is a powerful algorithm that comes up in a variety of contexts within data science. K Means is a particularly simple and easy-to-understand application of the algorithm, and we will walk through it briefly here. In short, the expectation-maximization approach here consists of the following procedure:

1. Guess some cluster centers
2. Repeat until converged
 - *E-Step*: assign points to the nearest cluster center
 - *M-Step*: Set the cluster centers to the mean

Here the “E-step” or “Expectation step” is so-named because it involves updating our expectation of which cluster each point belongs to. The “M-step” or “Maximization step” is so-named because it involves maximizing some fitness function which defines the location of the cluster centers – in this case, that maximization is accomplished by taking a simple mean of the data in each cluster.

The literature about this algorithm is vast, but can be summarized as follows: under typical circumstances, each repetition of the E-step and M-step will always result in a better estimate of the cluster characteristics.

We can visualize the algorithm as follows:



For the particular initialization shown in the above figure, the clusters converge in just three iterations.

The K Means algorithm is simple enough that we can write it in a few lines of code. The following is a very basic implementation:

```
from sklearn.metrics import pairwise_distances_argmin

def find_clusters(X, n_clusters, rseed=2):
    # 1. Randomly choose clusters
    rng = np.random.RandomState(rseed)
    i = rng.permutation(X.shape[0])[:n_clusters]
    centers = X[i]

    while True:
        # 2A. Assign labels based on closest center
        labels = pairwise_distances_argmin(X, centers)

        # 2B. Find new centers from means of points
        new_centers = np.array([X[labels == i].mean(0)
                               for i in range(n_clusters)])

        # 2C. Check for convergence
        if np.all(centers == new_centers):
            break
        centers = new_centers

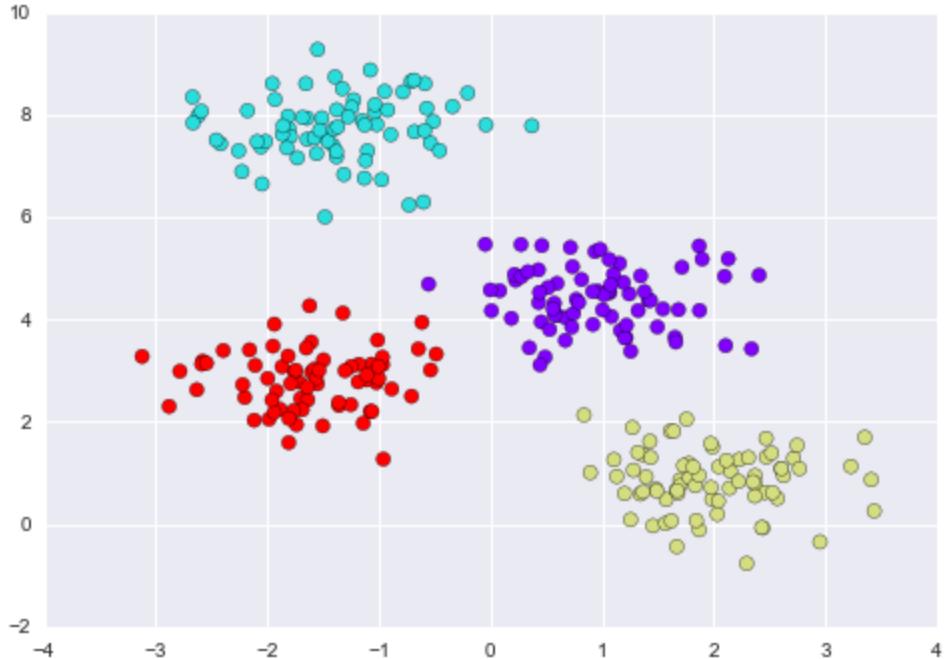
    return centers, labels
```

```

centers, labels = find_clusters(X, 4)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='rainbow');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



Most well-tested implementations will do a bit more than this under the hood, but the above function gives the gist of the expectation-maximization approach.

Caveats of Expectation-Maximization

There are a few issues to be aware of when using the above algorithm.

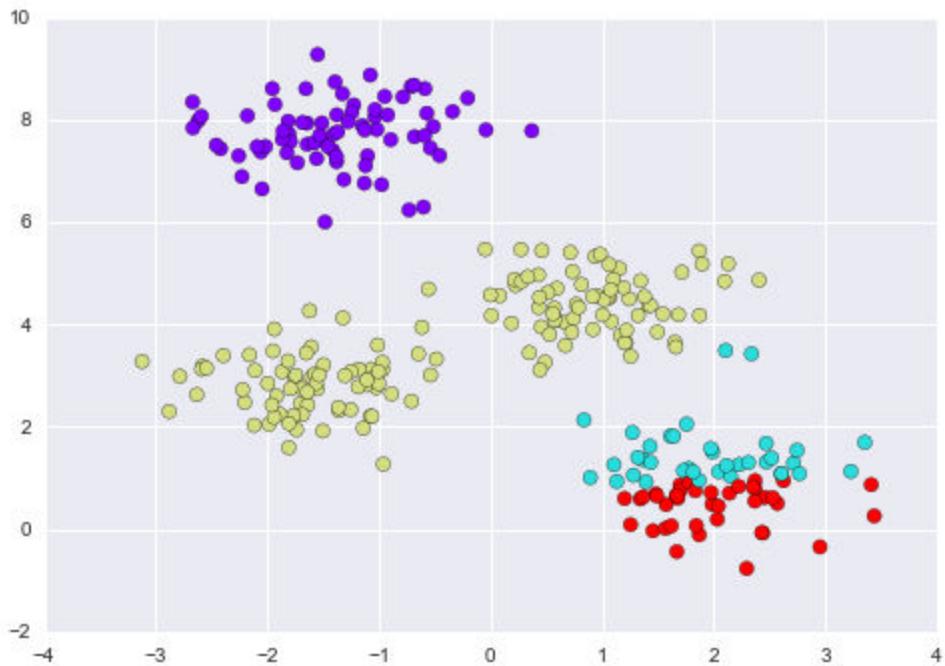
1. The globally optimal result may not be achieved. First, although the E-M procedure is guaranteed to improve the result in each step, there is no guarantee that it will lead to the *global* best solution. For example, if we use a different random seed in our simple procedure above, the particular starting guesses lead to poor results:

```

centers, labels = find_clusters(X, 4, rseed=0)
plt.scatter(X[:, 0], X[:, 1], c=labels,
            s=50, cmap='rainbow');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```

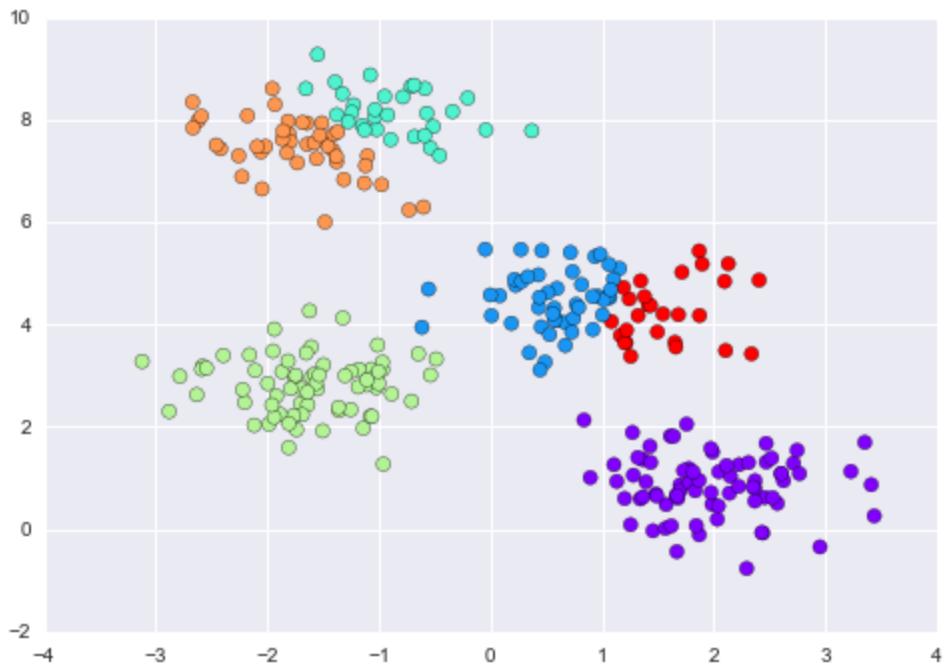


Here the E-M approach has converged, but has not converged to a globally optimal configuration. For this reason, it is common for the algorithm to be run for multiple starting guesses, as indeed scikit-learn does by default (set by the `n_init` parameter, which defaults to 10).

2. The number of clusters must be selected beforehand. Another common challenge with K Means is that you must tell it how many clusters you expect: it cannot learn the number of clusters from the data. For example, if we ask the algorithm to identify six clusters, it will happily proceed and find the best six clusters:

```
labels = KMeans(6, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
           s=50, cmap='rainbow');
```

```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



Whether the result is meaningful is a question that is difficult to answer definitively; one approach that is rather intuitive, but that we won't discuss further here, is called [silhouette analysis](#).

Alternatively, you might use a more complicated clustering algorithm which has a better quantitative measure of the fitness per number of clusters (e.g. Gaussian Mixture Models; see Section X.X) or which *can* choose a suitable number of clusters (e.g. DBSCAN, Mean-shift, or Affinity Propagation, all available in the `sklearn.cluster` submodule)

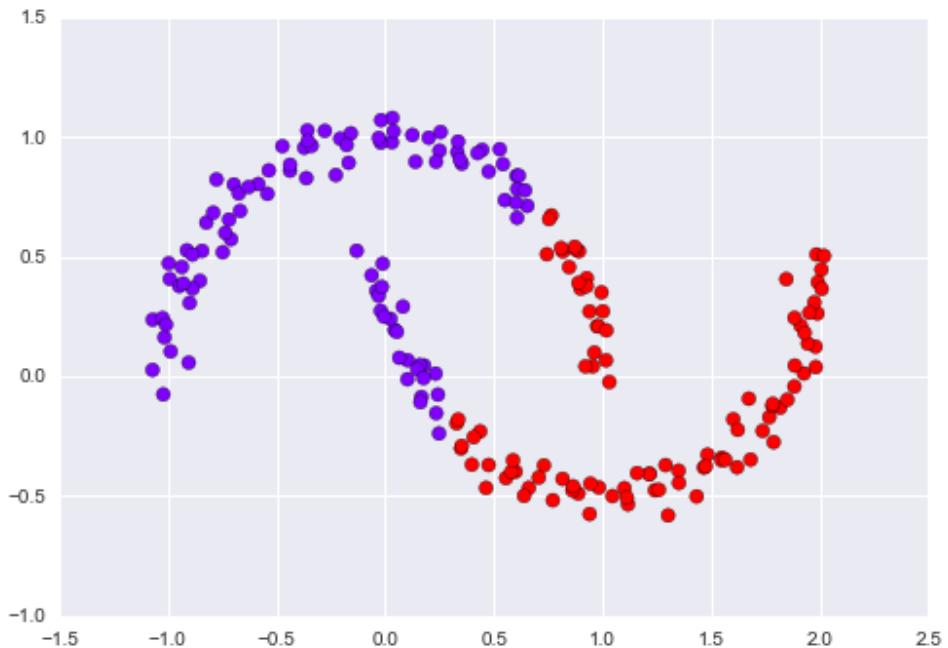
3. K Means is limited to linear cluster boundaries. The fundamental model assumptions of K Means (points will be closer to their own cluster center than to others) means that the algorithm will often be ineffective if the clusters are more complicated.

In particular, the boundaries between K-means clusters will always be linear, which means that it will fail for more complicated boundaries. Consider the following data, along with the cluster labels found by the typical K Means approach:

```
from sklearn.datasets import make_moons
X, y = make_moons(200, noise=.05, random_state=0)

labels = KMeans(2, random_state=0).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels,
           s=50, cmap='rainbow');
```

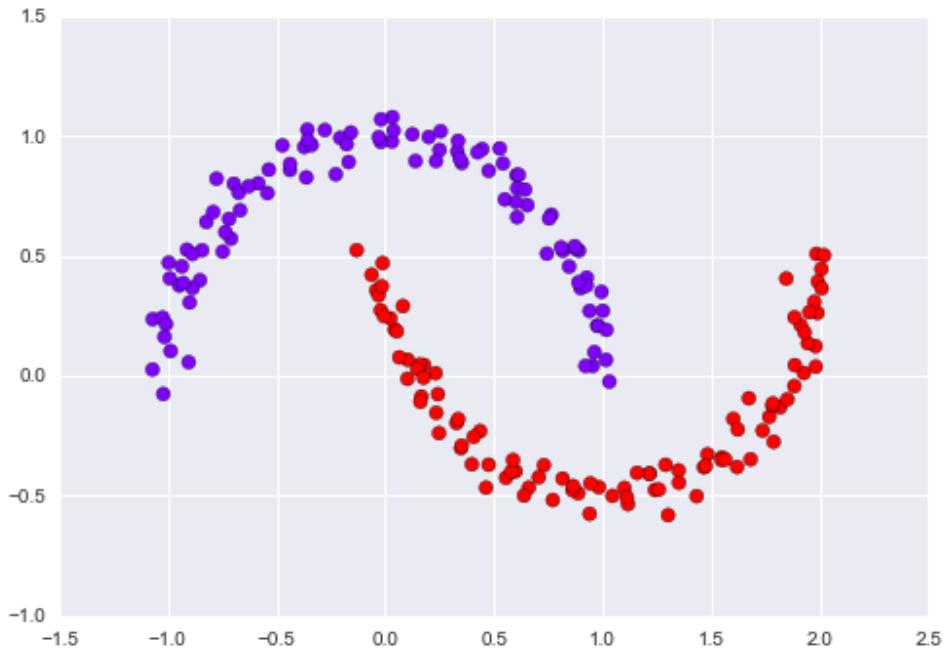
```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face'):
```



This situation is reminiscent of the discussion in Section X.X, where we used a kernel transformation to project the data into a higher dimension where a linear separation is possible. We might imagine using the same trick to allow K-means to discover non-linear boundaries.

One version of this Kernelized K-Means is implemented in scikit-learn within the Spectral Clustering estimator. It uses the graph of nearest neighbors to compute a higher-dimensional representation of the data, and then assigns labels using a K-Means algorithm:

```
from sklearn.cluster import SpectralClustering  
model = SpectralClustering(n_clusters=2, affinity='nearest_neighbors',  
                           assign_labels='kmeans')  
labels = model.fit_predict(X)  
plt.scatter(X[:, 0], X[:, 1], c=labels,  
            s=50, cmap='rainbow');  
  
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/sklearn/manifold/spectral_embedding  
    warnings.warn("Graph is not fully connected, spectral embedding"  
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:  
    if self._edgecolors == str('face')):
```



We see that with this kernel transform approach, K means is able to find the more complicated nonlinear boundaries between clusters.

4. K Means can be slow for large numbers of samples. Because each iteration of K-Means must access every point in the dataset, the algorithm can be relatively slow as the number of samples grows. You might wonder if this requirement to use all data at each iteration can be relaxed; for example, you might just use a subset of the data to update the cluster centers at each step. This is the idea behind batch-based K-Means algorithms, one form of which is implemented in `sklearn.cluster.MiniBatchKMeans`. The interface for this is the same as for standard `KMeans`; we will see an example of its use below.

Being careful about these limitations of the algorithm, we can use K Means to our advantage in a wide variety of situations. We will see a couple examples of this below.

Example 1: K-Means on Digits

To start, let's take a look at applying K Means on the same simple digits data that we saw in Section X.X. Here we will attempt to use K Means to try to identify similar digits without using the original label information; this might be similar to a first-step in extracting meaning from a new dataset about which you don't have any a priori information.

We will start by loading the digits and then finding the KMeans clusters. Recall that the digits consist of 1797 samples with 64 features, where each of the 64 features is the brightness of one pixel in an 8x8 image:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape

(1797, 64)
```

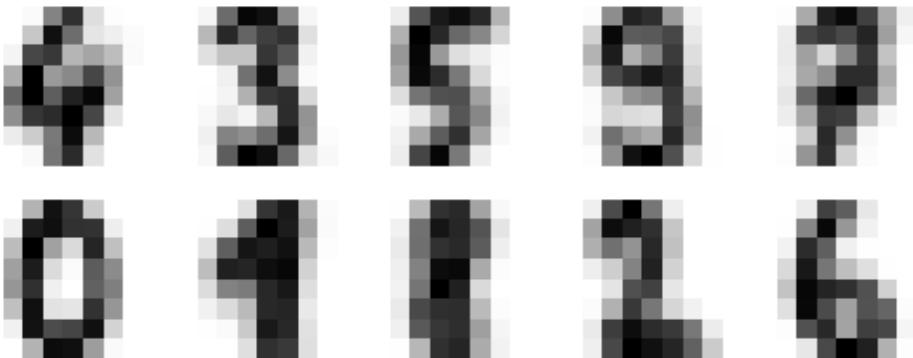
The clustering can be performed as we did above:

```
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape

(10, 64)
```

The result is ten clusters in 64 dimensions. Notice that the cluster centers themselves are 64-dimensional points, and can themselves be interpreted as the “typical” digit within the cluster. Let’s see what these cluster centers look like

```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks[])
    axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```



We see that *even without the labels*, KMeans is able to find clusters whose centers are recognizable digits, with perhaps the exception of 1 and 8.

Since K-Means knows nothing about the identity of the cluster, the 0-9 labels may be permuted. We can fix this by matching each learned cluster label with the true labels found in them:

```
from scipy.stats import mode

labels = np.zeros_like(clusters)
for i in range(10):
```

```
mask = (clusters == i)
labels[mask] = mode(digits.target[mask])[0]
```

Now we can check how accurate our unsupervised clustering was in finding similar digits within the data:

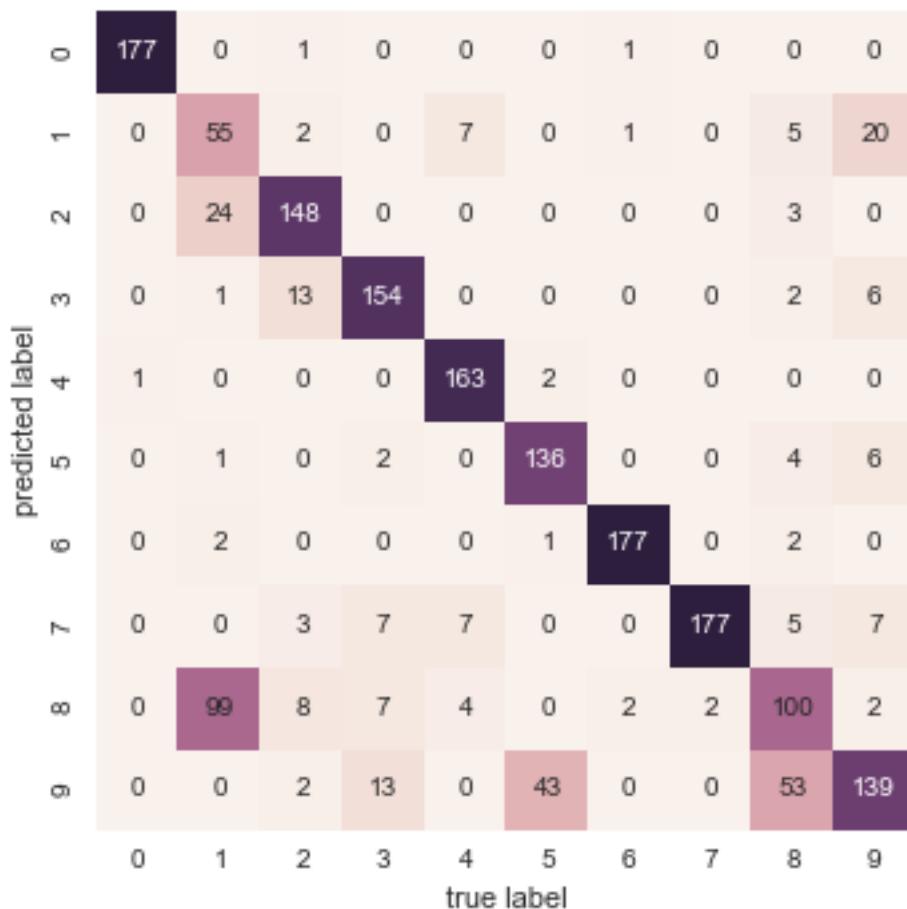
```
from sklearn.metrics import accuracy_score
accuracy_score(digits.target, labels)

0.79354479688369506
```

With just a simple K Means algorithm, we discovered the correct grouping for 80% of the input digits! Let's check the confusion matrix for this:

```
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(digits.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=digits.target_names,
            yticklabels=digits.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



As we might expect from the cluster centers we visualized above, the main point of confusion is between the eights and ones. But this still shows that using K-Means, we can essentially build a digit classifier *without reference to any labels!*

Just for fun, let's try to push this even farther. We can use the t-distributed stochastic neighbor embedding (t-SNE) algorithm discussed in Section X.X to pre-process the data before performing K-Means. T-SNE is a nonlinear embedding algorithm which is particularly adept at preserving points within clusters. Let's see how it does:

```
from sklearn.manifold import TSNE

# Project the data: this step will take several seconds
tsne = TSNE(n_components=2, init='pca', random_state=0)
digits_proj = tsne.fit_transform(digits.data)

# Compute the clusters
```

```

kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits_proj)

# Permute the labels
labels = np.zeros_like(clusters)
for i in range(10):
    mask = (clusters == i)
    labels[mask] = mode(digits.target[mask])[0]

# Compute the accuracy
accuracy_score(digits.target, labels)
0.93878686700055647

```

That's nearly 94% classification accuracy *without using the labels*. This is the power of unsupervised learning when used carefully: it can extract information from the dataset that it might be difficult to do by hand or by eye.

Example 2: KMeans for Color Compression

One interesting application of clustering is in color compression within images. For example, imagine you have an image with millions of colors. In most images, a large number of the colors will be unused, and many of the pixels in the image will have similar or even identical colors.

For example, consider this image from the scikit-learn datasets module (for this to work, you'll have to have the `pillow` Python package installed).

```

# Note: this requires the ``pillow`` package to be installed
from sklearn.datasets import load_sample_image
china = load_sample_image("china.jpg")
ax = plt.axes(xticks=[], yticks[])
ax.imshow(china);

```



The image itself is stored in a 3-dimensional array of size (`height`, `width`, `RGB`), containing red/blue/green contributions as integers from 0 to 255:

```
china.shape  
(427, 640, 3)
```

One way we can view this set of pixels is as a cloud of points in a three-dimensional color space. We will reshape the data to `[n_samples x n_features]`, and rescale the colors so that they lie between 0 and 1:

```
data = china / 255.0 # use 0...1 scale  
data = data.reshape(427 * 640, 3)  
data.shape  
(273280, 3)
```

We can visualize these pixels in this color space, using a subset of 10,000 pixels for efficiency:

```
def plot_pixels(data, title, colors=None, N=10000):  
    if colors is None:  
        colors = data  
  
    # choose a random subset  
    rng = np.random.RandomState(0)  
    i = rng.permutation(data.shape[0])[0:N]  
    colors = colors[i]  
    R, G, B = data[i].T
```

```

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
ax[0].scatter(R, G, color=colors, marker='.')
ax[0].set(xlabel='Red', ylabel='Green', xlim=(0, 1), ylim=(0, 1))

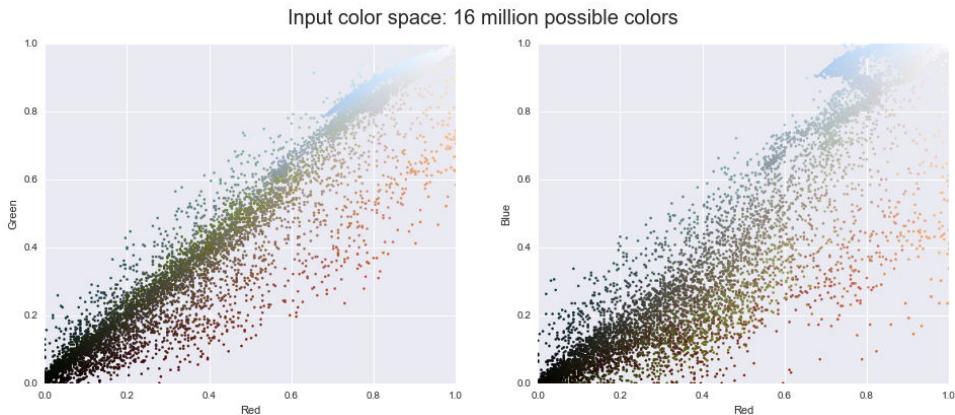
ax[1].scatter(R, B, color=colors, marker='.')
ax[1].set(xlabel='Red', ylabel='Blue', xlim=(0, 1), ylim=(0, 1))

fig.suptitle(title, size=20);

plot_pixels(data, title='Input color space: 16 million possible colors')

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



Now let's reduce these 16 million colors to just 16 colors, using a K-Means clustering across the pixel space. Because we are dealing with a very large dataset, we will use the mini batch K Means, which computes the result much more quickly than the vanilla K Means algorithm:

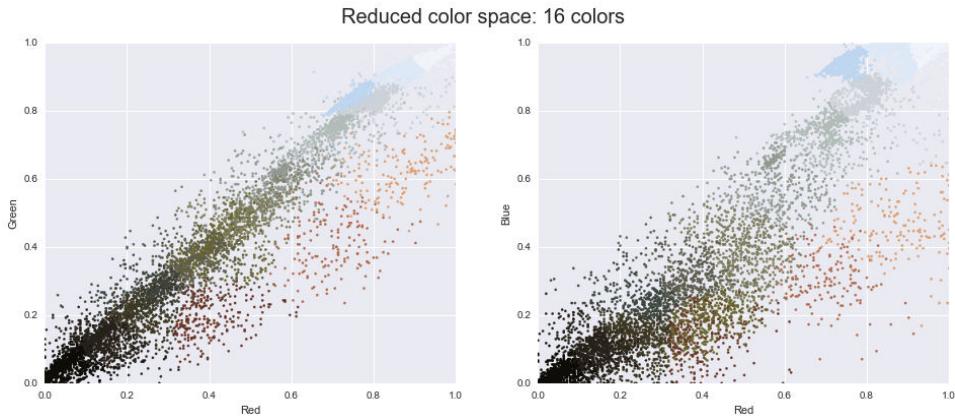
```

from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(16)
kmeans.fit(data)
new_colors = kmeans.cluster_centers_[kmeans.predict(data)]

plot_pixels(data, colors=new_colors,
            title="Reduced color space: 16 colors")

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



The result is a re-coloring of the original pixels, where each pixel is assigned the color of its closest cluster center. Plotting these new colors in the image space rather than the pixel space shows us the effect of this:

```
china_recolored = new_colors.reshape(china.shape)

fig, ax = plt.subplots(1, 2, figsize=(16, 6), subplot_kw=dict(xticks=[], yticks=[]))
fig.subplots_adjust(wspace=0.05)
ax[0].imshow(china)
ax[0].set_title('Original Image', size=16)
ax[1].imshow(china_recolored)
ax[1].set_title('16-color Image', size=16);
```



Some detail is certainly lost in the left image, but the overall image is still easily recognizable. This image on the right achieves a compression factor of around 1 million! While this is an interesting application of K Means, there are certainly better way to compress information in images. But the example shows the power of thinking out-of-the-box with unsupervised methods like K-Means.

Figures

```
import os
if not os.path.exists('fig'):
    os.makedirs('fig')
```

Expectation-Maximization

The following figure shows a visual depiction of the Expectation-Maximization approach to K Means:

```
from sklearn.datasets.samples_generator import make_blobs
from sklearn.metrics import pairwise_distances_argmin

X, y_true = make_blobs(n_samples=300, centers=4,
                       cluster_std=0.60, random_state=0)

rng = np.random.RandomState(42)
centers = [0, 4] + rng.randn(4, 2)

def draw_points(ax, c, factor=1):
    ax.scatter(X[:, 0], X[:, 1], c=c, cmap='rainbow',
               s=50 * factor, alpha=0.3)

def draw_centers(ax, centers, factor=1, alpha=1.0):
    ax.scatter(centers[:, 0], centers[:, 1],
               c=np.arange(4), cmap='rainbow', s=200 * factor,
               alpha=alpha)
    ax.scatter(centers[:, 0], centers[:, 1],
               c='black', s=50 * factor, alpha=alpha)

def make_ax(fig, gs):
    ax = fig.add_subplot(gs)
    ax.xaxis.set_major_formatter(plt.NullFormatter())
    ax.yaxis.set_major_formatter(plt.NullFormatter())
    return ax

fig = plt.figure(figsize=(15, 4))
gs = plt.GridSpec(4, 15, left=0.02, right=0.98, bottom=0.05, top=0.95, wspace=0.2, hspace=0.2)
ax0 = make_ax(fig, gs[:4, :4])
ax0.text(0.98, 0.98, "Random Initialization", transform=ax0.transAxes,
         ha='right', va='top', size=16)
draw_points(ax0, 'gray', factor=2)
draw_centers(ax0, centers, factor=2)

for i in range(3):
    ax1 = make_ax(fig, gs[4:, 4 + 2 * i:6 + 2 * i])
    ax2 = make_ax(fig, gs[2:, 5 + 2 * i:7 + 2 * i])

    # E-step
    y_pred = pairwise_distances_argmin(X, centers)
    draw_points(ax1, y_pred)
```

```

draw_centers(ax1, centers)

# M-step
new_centers = np.array([X[y_pred == i].mean(0) for i in range(4)])
draw_points(ax2, y_pred)
draw_centers(ax2, centers, alpha=0.3)
draw_centers(ax2, new_centers)
for i in range(4):
    ax2.annotate('', new_centers[i], centers[i],
                 arrowprops=dict(arrowstyle='->', linewidth=1))

# Finish iteration
centers = new_centers
ax1.text(0.95, 0.95, "E-Step", transform=ax1.transAxes, ha='right', va='top', size=14)
ax2.text(0.95, 0.95, "M-Step", transform=ax2.transAxes, ha='right', va='top', size=14)

# Final E-step
y_pred = pairwise_distances_argmin(X, centers)
axf = make_ax(fig, gs[:4, -4:])
draw_points(axf, y_pred, factor=2)
draw_centers(axf, centers, factor=2)
axf.text(0.98, 0.98, "Final Clustering", transform=axf.transAxes,
         ha='right', va='top', size=16)

fig.savefig('fig/07.11-expectation-maximization.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```

Interactive K-Means

The following script uses IPython's interactive widgets to demonstrate the K-means algorithm. Run this within the IPython notebook to explore the expectation maximization algorithm for computing K Means.

```

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling
import numpy as np

from ipywidgets import interact
from sklearn.metrics import pairwise_distances_argmin
from sklearn.datasets.samples_generator import make_blobs

def plot_kmeans_interactive(min_clusters=1, max_clusters=6):
    X, y = make_blobs(n_samples=300, centers=4,
                      random_state=0, cluster_std=0.60)

```

```

def plot_points(X, labels, n_clusters):
    plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='rainbow',
                vmin=0, vmax=n_clusters - 1);

def plot_centers(centers):
    plt.scatter(centers[:, 0], centers[:, 1], marker='o',
                c=np.arange(centers.shape[0]),
                s=200, cmap='rainbow')
    plt.scatter(centers[:, 0], centers[:, 1], marker='o',
                c='black', s=50)

def _kmeans_step(frame=0, n_clusters=4):
    rng = np.random.RandomState(2)
    labels = np.zeros(X.shape[0])
    centers = rng.randn(n_clusters, 2)

    nsteps = frame // 3

    for i in range(nsteps + 1):
        old_centers = centers
        if i < nsteps or frame % 3 > 0:
            labels = pairwise_distances_argmin(X, centers)

        if i < nsteps or frame % 3 > 1:
            centers = np.array([X[labels == j].mean(0)
                               for j in range(n_clusters)])
            nans = np.isnan(centers)
            centers[nans] = old_centers[nans]

    # plot the data and cluster centers
    plot_points(X, labels, n_clusters)
    plot_centers(old_centers)

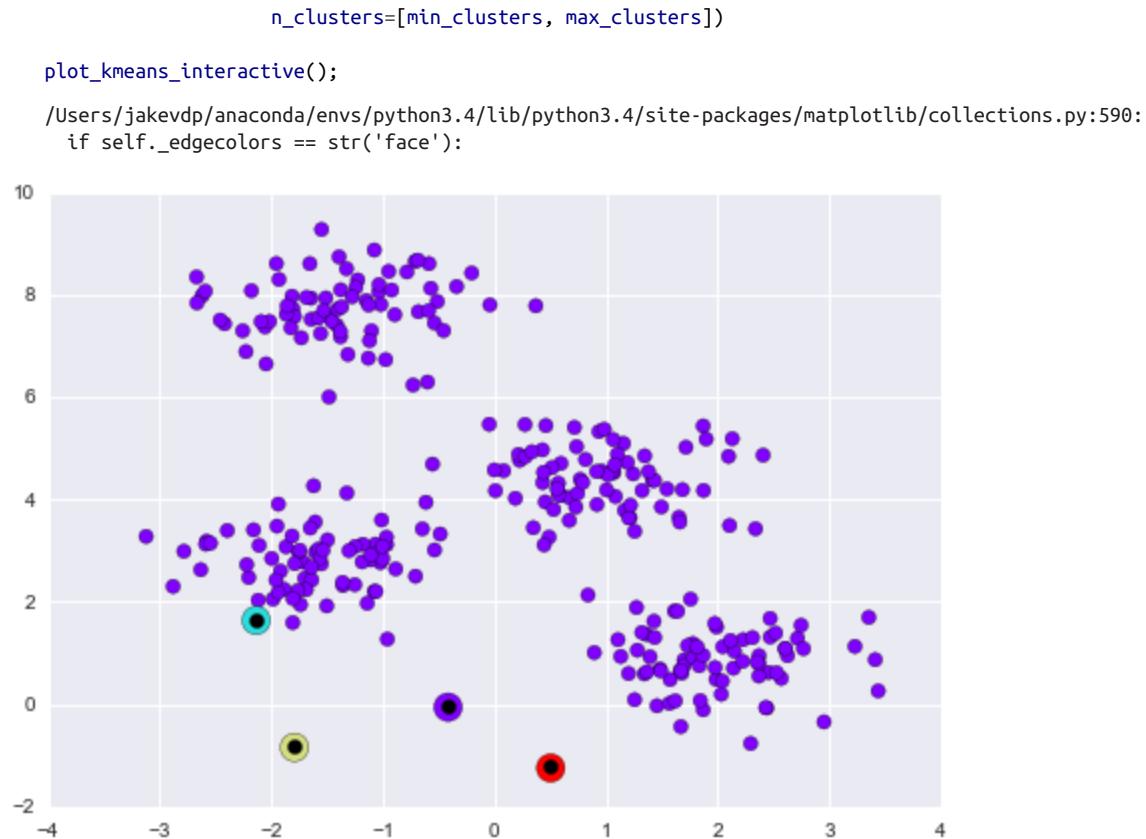
    # plot new centers if third frame
    if frame % 3 == 2:
        for i in range(n_clusters):
            plt.annotate('', centers[i], old_centers[i],
                        arrowprops=dict(arrowstyle='->', linewidth=1))
    plot_centers(centers)

    plt.xlim(-4, 4)
    plt.ylim(-2, 10)

    if frame % 3 == 1:
        plt.text(3.8, 9.5, "1. Reassign points to nearest centroid",
                 ha='right', va='top', size=14)
    elif frame % 3 == 2:
        plt.text(3.8, 9.5, "2. Update centroids to cluster means",
                 ha='right', va='top', size=14)

return interact(_kmeans_step, frame=[0, 50],

```



In Depth: Gaussian Mixture Models

The K-Means clustering model explored in the previous section is simple and relatively easy to understand, but its simplicity leads to practical challenges in its application. In particular, the non-probabilistic nature of KMeans and its use of simple distance-from-cluster-center to assign cluster membership leads to poor performance for many real-world situations. In this section we will take a look at Gaussian Mixture Models, which can be viewed as an extension of the ideas behind KMeans, but can also be a powerful tool for estimation beyond simple clustering.

```

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np

```

Motivating GMM: Weaknesses of K Means

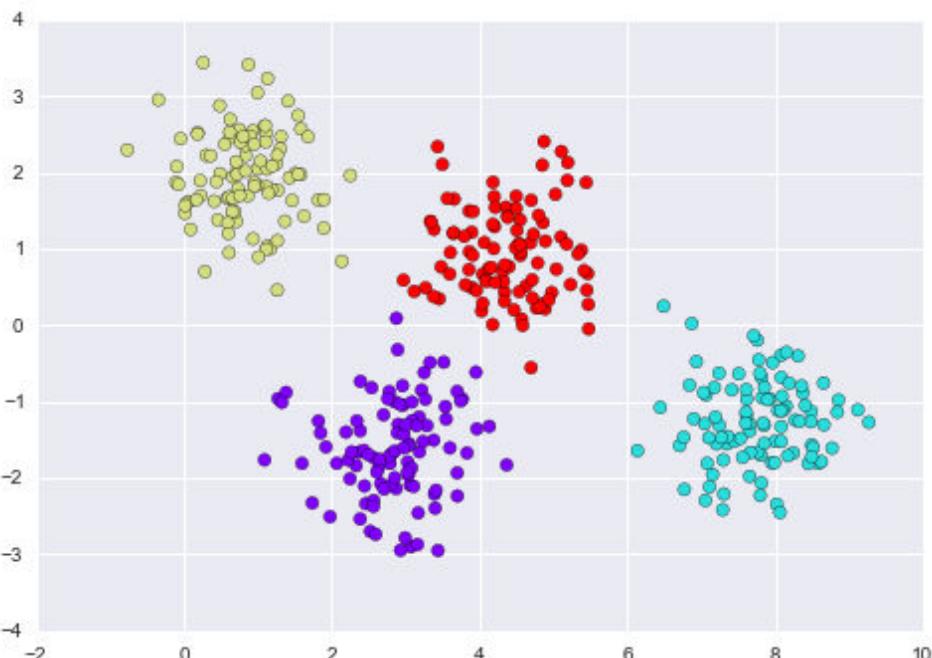
Let's take a look at some of the weaknesses of K Means and think about how we might improve the cluster model. As we saw in the previous section, given simple, well-separated data, K Means finds suitable clustering results.

For example, if we have simple blobs of data, the K Means algorithm can quickly label those clusters in a way that closely matches what we might do by eye:

```
# Generate some data
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=400, centers=4,
                      cluster_std=0.60, random_state=0)
X = X[:, ::-1] # flip axes for better plotting

# Plot the data with K Means Labels
from sklearn.cluster import KMeans
kmeans = KMeans(4, random_state=0)
labels = kmeans.fit(X).predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='rainbow');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):
```



From an intuitive standpoint, we might expect that the clustering assignment for some points is more certain than others: for example, there appears to be a very slight overlap between the two middle clusters, such that we might not have complete confi-

dence in the cluster assignment of points between them. Unfortunately, the KMeans model has no intrinsic measure of probability or uncertainty of cluster assignments (although it may be possible to use a bootstrap approach to estimate this uncertainty). For this, we must think about generalizing the model.

One way to think about the K Means model is that it places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster. This radius acts as a hard cutoff for cluster assignment within the training set: any point outside this circle is not considered a member of the cluster. We can visualize this cluster model with the following function:

```
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist

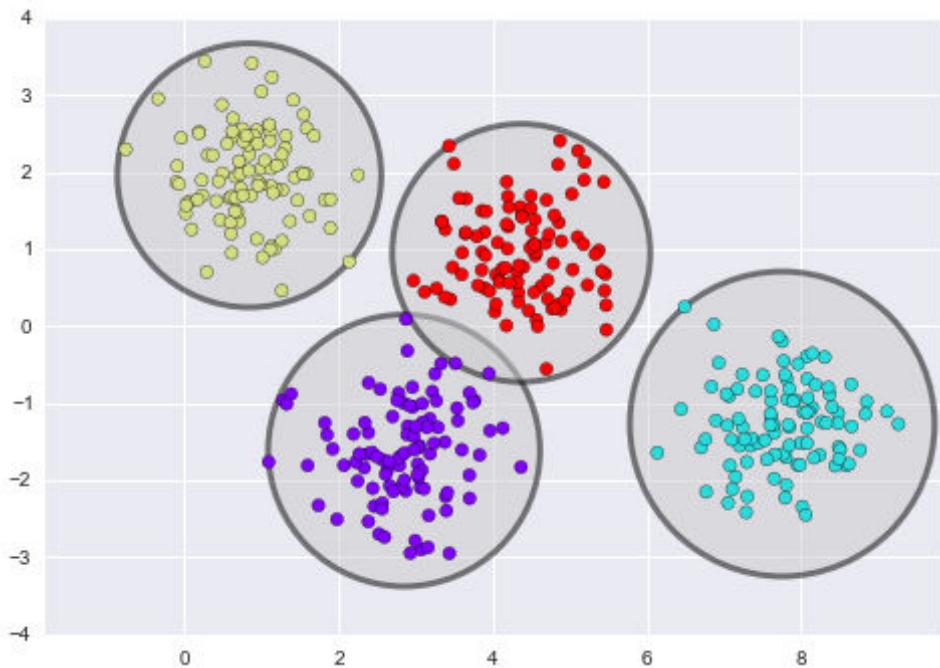
def plot_kmeans(kmeans, X, n_clusters=4, rseed=0, ax=None):
    labels = kmeans.fit_predict(X)

    # plot the input data
    ax = ax or plt.gca()
    ax.axis('equal')
    ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='rainbow', zorder=2)

    # plot the representation of the KMeans model
    centers = kmeans.cluster_centers_
    radii = [cdist(X[labels == i], [center]).max()
             for i, center in enumerate(centers)]
    for c, r in zip(centers, radii):
        ax.add_patch(plt.Circle(c, r, fc='#CCCCCC', lw=3, alpha=0.5, zorder=1))

kmeans = KMeans(n_clusters=4, random_state=0)
plot_kmeans(kmeans, X)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



An important observation for K means is that these cluster models *must be circular*: K Means has no built-in way of accounting for oblong or elliptical clusters. So, for example, if we take the same data and transform it, the cluster assignments end up becoming muddled:

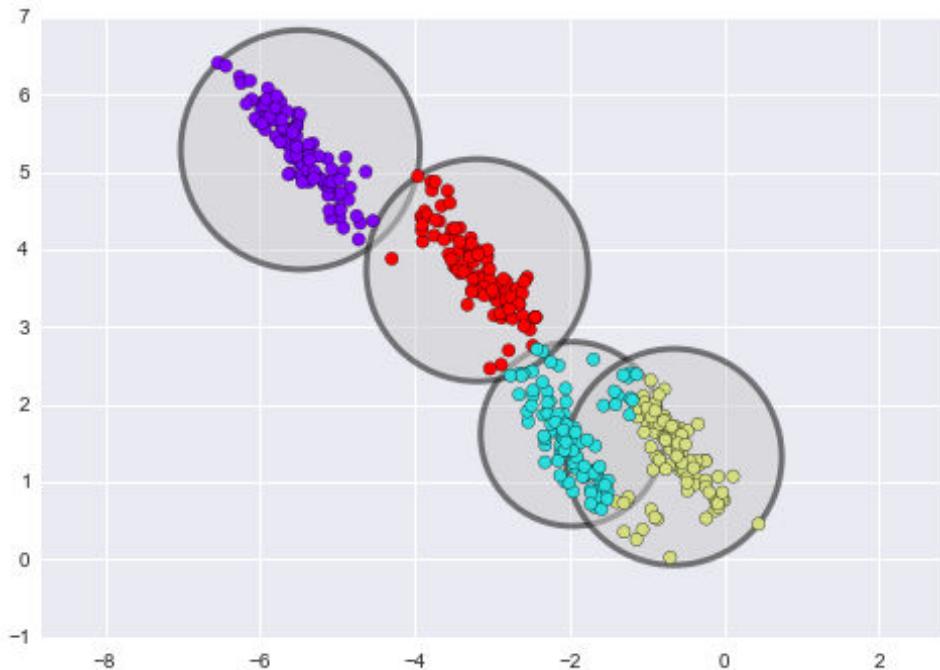
```

rng = np.random.RandomState(13)
X_stretched = np.dot(X, rng.randn(2, 2))

kmeans = KMeans(n_clusters=4, random_state=0)
plot_kmeans(kmeans, X_stretched)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):

```



By eye, we recognize that these transformed clusters are non-circular, and thus circular clusters would be a poor fit. Nevertheless, KMeans is not flexible enough to account for this, and tries to force-fit the data into four circular clusters. This results in a mixing of cluster assignments where the resulting circles overlap: see especially the bottom-right of this plot. One might imagine addressing this particular situation by preprocessing the data with PCA (see Section X.X), but in practice there is no guarantee that such a global operation will circularize the individual data.

These two disadvantages of K Means – its lack of flexibility in cluster shape and lack of probabilistic cluster assignment – means that for many data sets (especially low-dimensional datasets) it may not perform as well as you might hope.

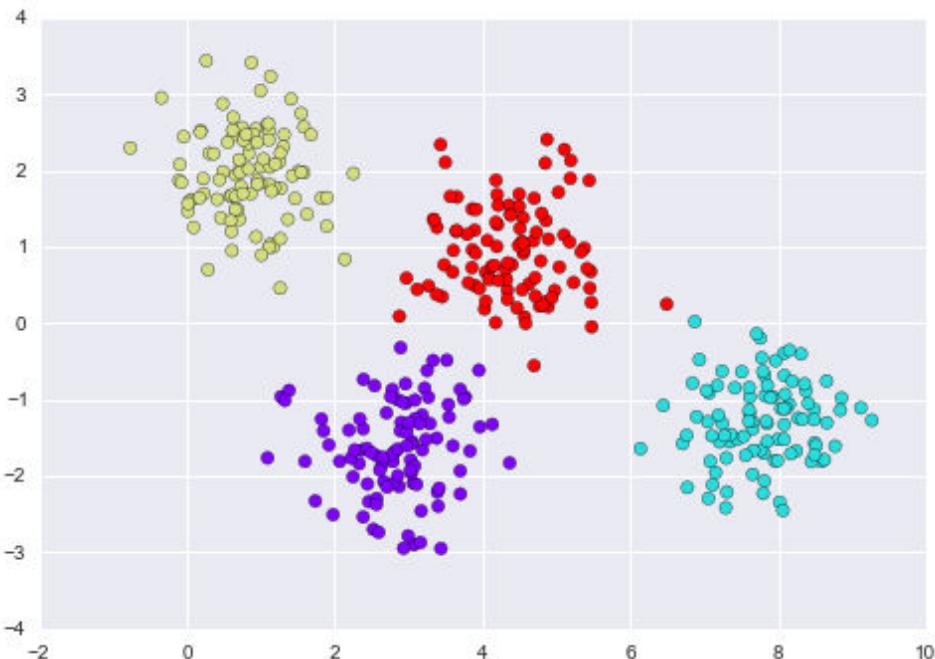
You might imagine addressing these weaknesses by generalizing the K Means model: for example, you could measure uncertainty in cluster assignment by comparing the distances of each point to *all* cluster centers, rather than focusing on just the closest. You might also imagine allowing the cluster boundaries to be ellipses rather than circles, so as to account for non-circular clusters. It turns out these are two essential components of a different type of clustering model, Gaussian Mixture Models.

Generalizing E-M: Gaussian Mixture Models

A Gaussian Mixture Model (GMM) attempts to find a mixture of multi-dimensional Gaussian probability distributions which best model any input dataset. In the simplest case, GMMs can be used for finding clusters in the same manner as K Means:

```
from sklearn.mixture import GMM
gmm = GMM(n_components=4).fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='rainbow');

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



But because GMM contains a probabilistic model under the hood, it is also possible to find probabilistic cluster assignments – in scikit-learn this is done using the `predict_proba` method. This returns a matrix of size `[n_samples, n_clusters]` which measures the probability that any point belongs to the given cluster:

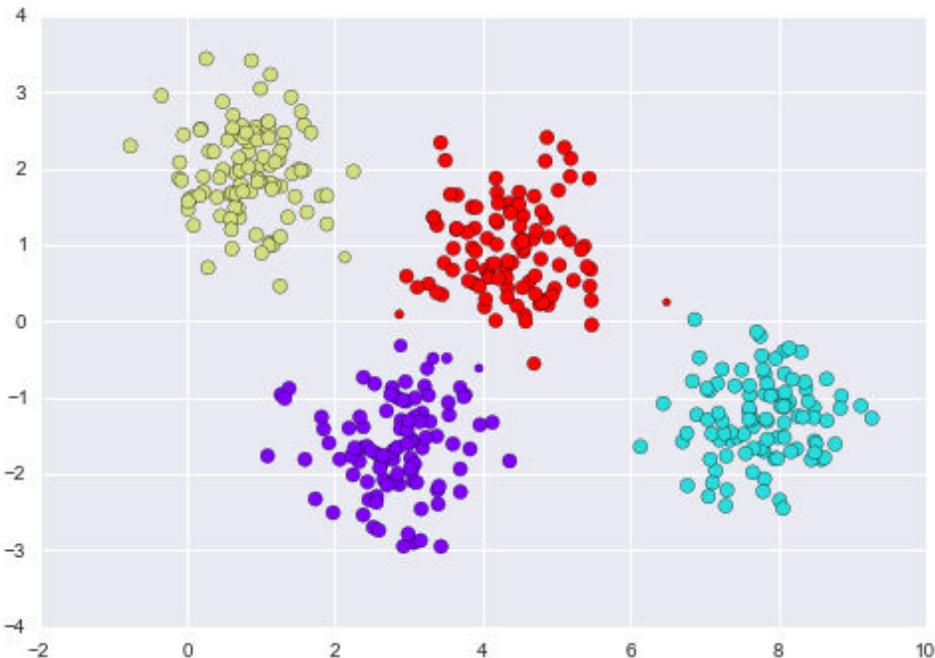
```
probs = gmm.predict_proba(X)
print(probs[:5].round(3))

[[ 0.        0.475   0.        0.525]
 [ 1.        0.        0.        0.      ]
 [ 1.        0.        0.        0.      ]
 [ 0.        0.        0.        1.      ]
 [ 1.        0.        0.        0.      ]]
```

We can visualize this uncertainty by making the size of each point proportional to the certainty of its prediction; we can see that it is precisely the points at the boundaries between clusters that reflect this uncertainty of cluster assignment:

```
size = 50 * probs.max(1) ** 2 # square emphasizes differences
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow', s=size);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



Under the hood, a Gaussian Mixture Model is very similar to K-Means: it uses an Expectation-Maximization approach which qualitatively does the following:

1. Choose starting guesses for the location and shape
2. Repeat until converged:
 - *E-step*: for each point, find weights encoding the probability of membership in each cluster
 - *M-step*: for each cluster, update its location, normalization, and shape based on *all* data points, making use of the weights

The result of this is that each cluster is associated not with a hard-edged sphere, but with a smooth Gaussian model. Just as in the K Means expectation-maximization

approach, this algorithm can sometimes miss the globally-optimal solution, and thus in practice multiple random initializations are used.

Let's create a function that will help us visualize the locations and shapes of the GMM clusters by drawing ellipses based on the gmm output:

```
from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()

    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))

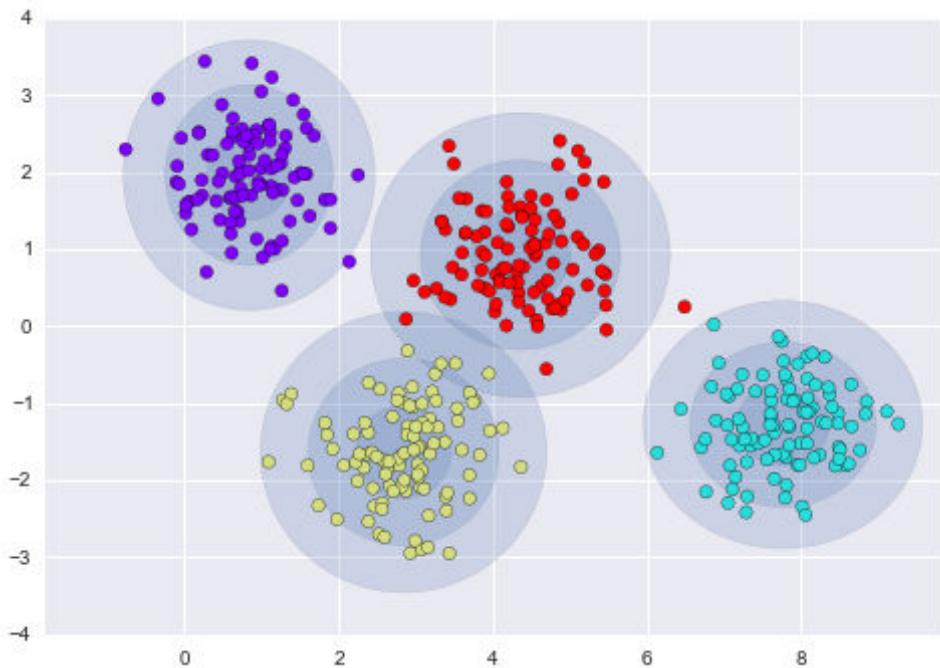
def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='rainbow', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covars_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

With this in place, we can take a look at what the GMM model gives us for our initial data above:

```
gmm = GMM(n_components=4, random_state=42)
plot_gmm(gmm, X)
```

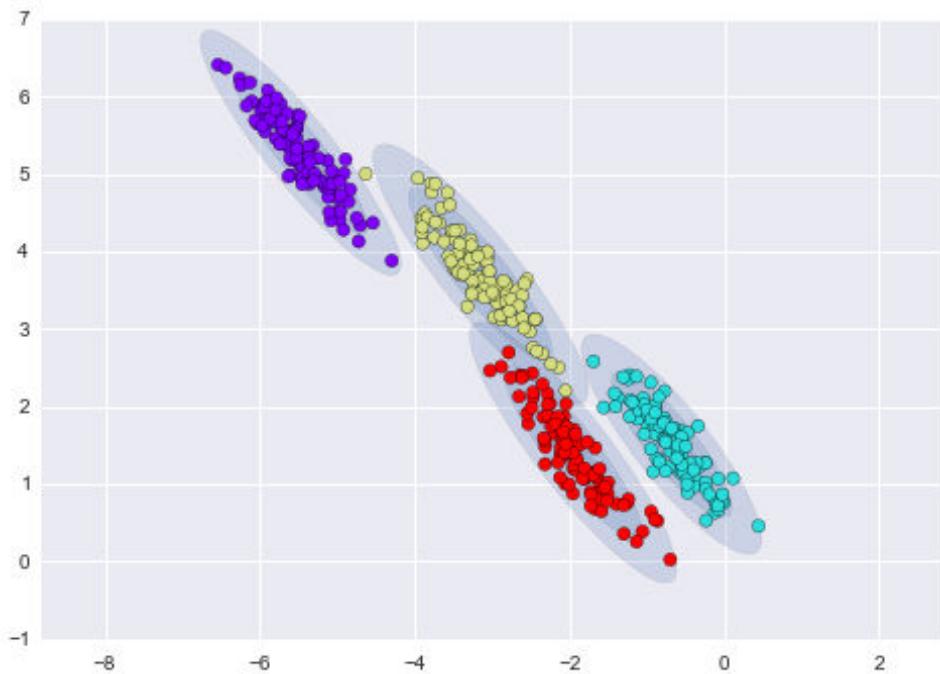
```
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



Similarly, we can use the GMM approach to fit our stretched dataset; allowing for a full covariance the model will fit even very oblong, stretched-out clusters:

```
gmm = GMM(n_components=4, covariance_type='full', random_state=42)
plot_gmm(gmm, X_stretched)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

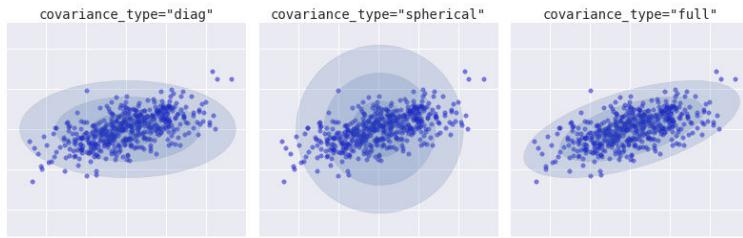


This makes clear that GMM addresses the two main practical issues with KMeans encountered above.

Choosing the Covariance Type

If you look at the details of the above fits, you will see that the `covariance_type` option was set differently within each. This hyperparameter controls the degrees of freedom in the shape of each cluster; it is essential to set this carefully for any given problem. The default is `covariance_type="diag"`, which means that the size of the cluster along each dimension can be set independently, with the resulting ellipse constrained to align with the axes. A slightly simpler and faster model is `covariance_type="spherical"`, which constrains the shape of the cluster such that all dimensions are equal. The resulting clustering will have similar characteristics to that of K Means, though it is not entirely equivalent. A more complicated and computationally expensive model (especially as the number of dimensions grows) is to use `covariance_type="full"`, which allows each cluster to be modeled as an ellipse with arbitrary orientation.

We can see a visual representation of these three choices for a single cluster within the following figure:



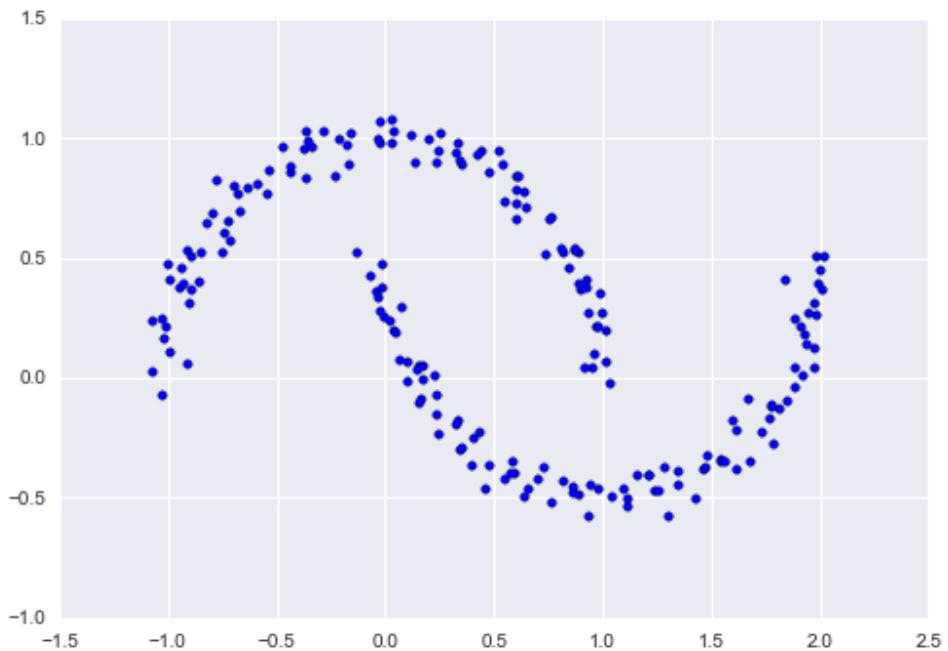
GMM as *Density Estimation*

Though GMM is often categorized as a clustering algorithm, fundamentally it is an algorithm for *density estimation*. That is to say, the result of a GMM fit to some data is technically not a clustering model, but a generative probabilistic model describing the distribution of the data.

As an example, consider some data generated from scikit-learn's `make_moons` function, which we saw in the previous section.

```
from sklearn.datasets import make_moons
Xmoon, ymoon = make_moons(200, noise=.05, random_state=0)
plt.scatter(Xmoon[:, 0], Xmoon[:, 1]);

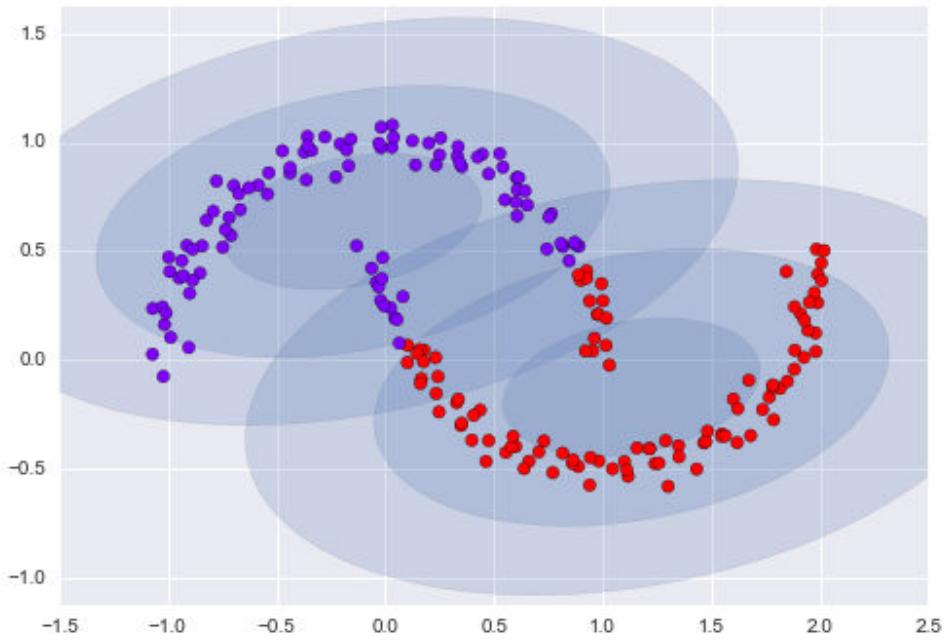
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



If we try to fit to this a two-component GMM viewed as a clustering model, the results are not particularly useful:

```
gmm2 = GMM(n_components=2, covariance_type='full', random_state=0)
plot_gmm(gmm2, Xmoon)

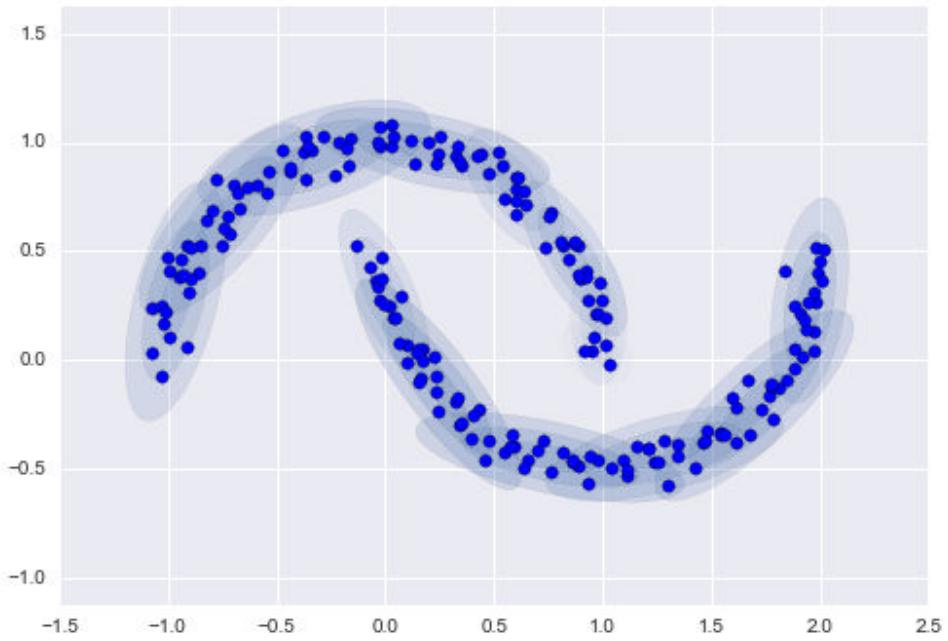
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



But if we instead use many more components and ignore the cluster labels, we find a fit that is much closer to the input data:

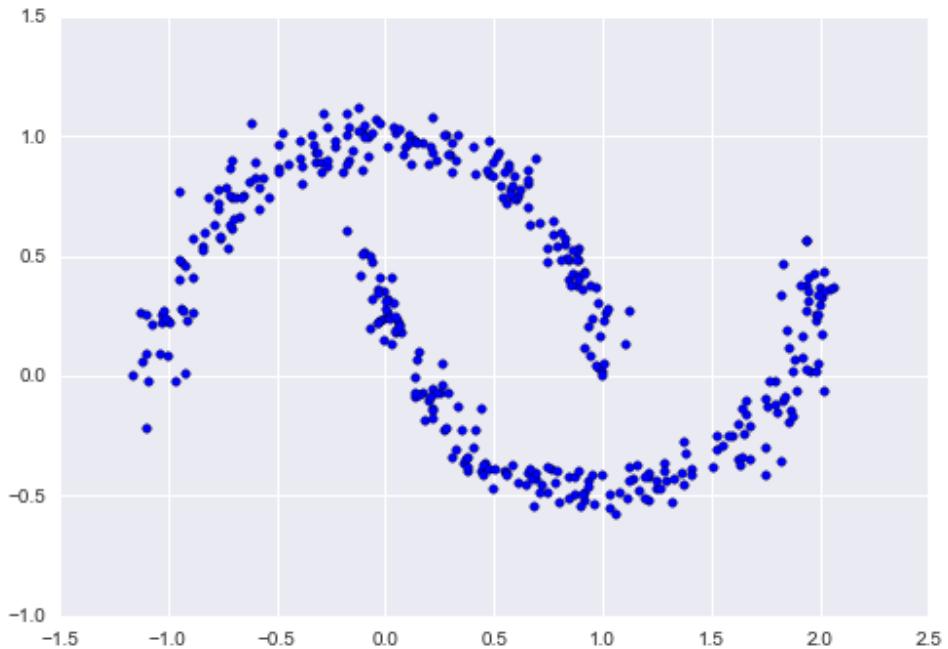
```
gmm16 = GMM(n_components=16, covariance_type='full', random_state=0)
plot_gmm(gmm16, Xmoon, label=False)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



Here the mixture of sixteen Gaussians serves not to find separated clusters of data, but rather to model the overall *distribution* of the input data. This is a generative model of the distribution, meaning that the GMM model gives us the recipe to generate new random data distributed similarly to our input. For example, here are 400 new points drawn from this 16-component GMM model to our original data:

```
Xnew = gmm16.sample(400, random_state=42)
plt.scatter(Xnew[:, 0], Xnew[:, 1]);
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
  if self._edgecolors == str('face'):
```



GMM is convenient as a flexible means of modeling an arbitrary multi-dimensional distribution of data.

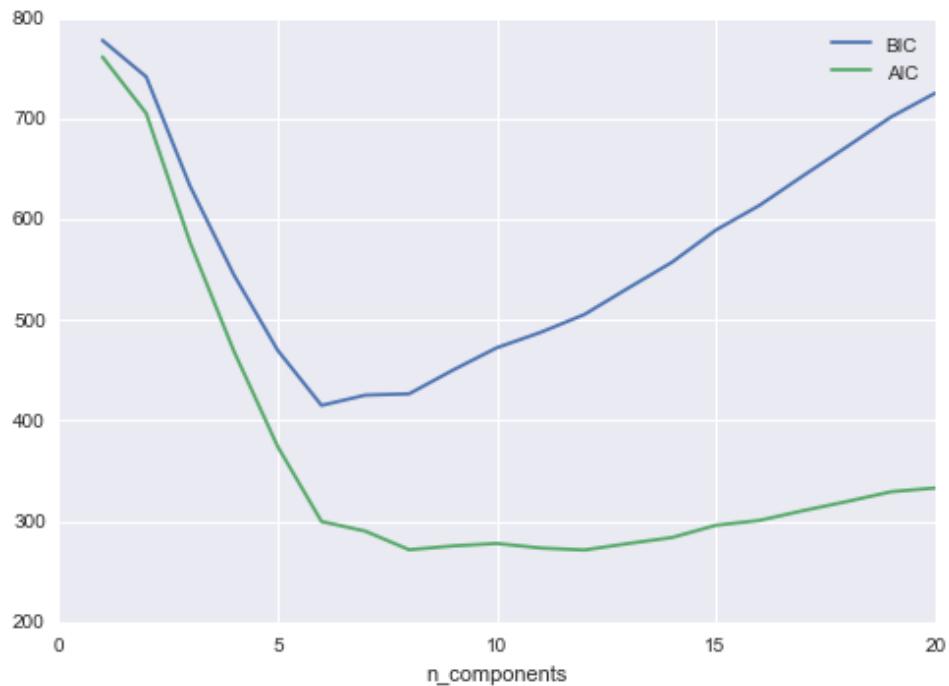
How Many Components?

The fact that GMM is a generative model gives us a natural means of determining the optimal number of components for a given dataset. A generative model is inherently a probability distribution for the dataset, and so we can simply evaluate the *likelihood* of the data under the model, using Cross-Validation to avoid over-fitting. Another means of correcting for over-fitting is to adjust the model likelihoods using an Information criterion such as the AIC or BIC (see Section X.X). Scikit-learn's GMM model actually includes built-in methods which compute the AIC and BIC, and so it is very easy to operate on this approach.

Let's look at the AIC and BIC as a function as the number of GMM components for our moon dataset:

```
n_components = np.arange(1, 21)
models = [GMM(n, covariance_type='full', random_state=0).fit(Xmoon)
          for n in n_components]

plt.plot(n_components, [m.bic(Xmoon) for m in models], label='BIC')
plt.plot(n_components, [m.aic(Xmoon) for m in models], label='AIC')
plt.legend(loc='best')
plt.xlabel('n_components');
```



The optimal number of clusters is the value that minimizes the AIC or BIC, depending on which approximation we wish to use. The AIC tells us that our choice of 16 components above was probably too many: around 8-12 components would have been a better choice. As is typical with this sort of problem, the BIC recommends a simpler model; see Section X.X for more discussion of the AIC and BIC.

Notice the important point: this choice of number of components measures how well GMM works *as a density estimator*, not how well it works *as a clustering algorithm*. I'd encourage you to think of GMM primarily as a density estimator, and use it for clustering only when warranted within simple datasets.

Example: GMM for Generating New Data

We saw above a simple example of using GMM as a generative model of data in order to create new samples from the distribution defined by the input data. Here we will run with this idea and generate *new handwritten digits* from the standard digits corpus that we have used before.

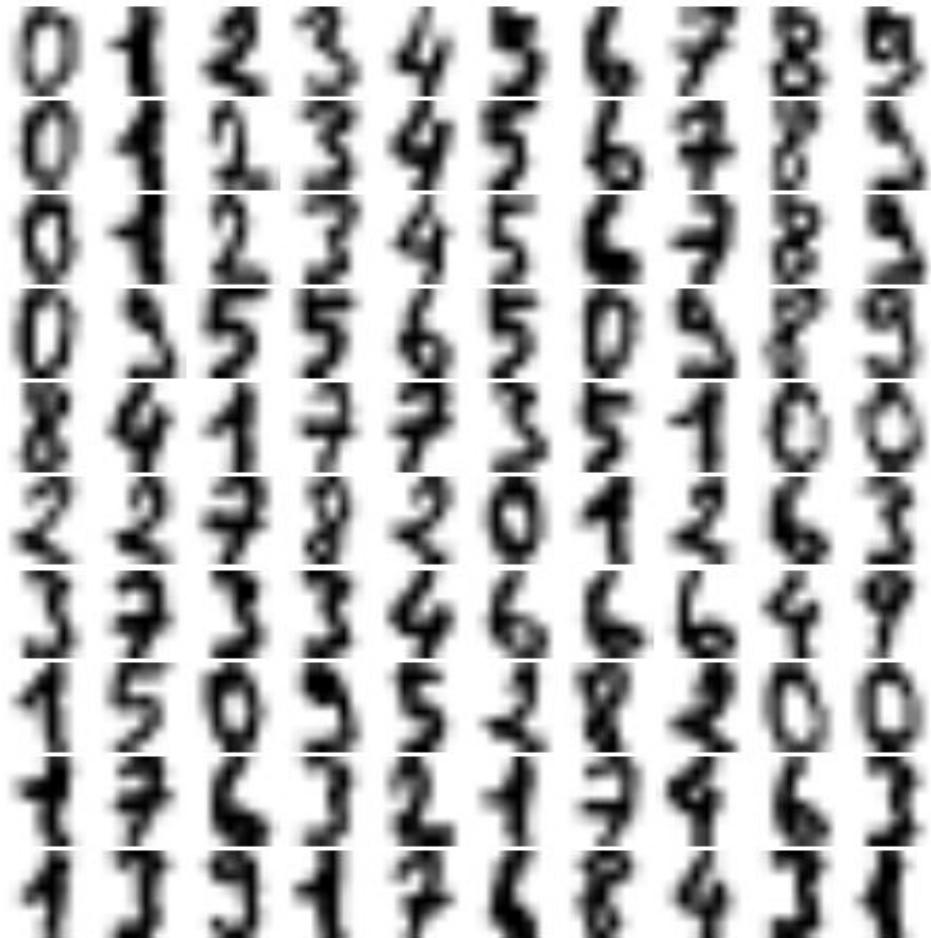
To start with, let's load the digits data using scikit-learn's data tools:

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

(1797, 64)

Next let's plot the first 100 of these to recall exactly what we're looking at:

```
def plot_digits(data):
    fig, ax = plt.subplots(10, 10, figsize=(8, 8),
                          subplot_kw=dict(xticks=[], yticks=[]))
    fig.subplots_adjust(hspace=0.05, wspace=0.05)
    for i, axi in enumerate(ax.flat):
        im = axi.imshow(data[i].reshape(8, 8), cmap='binary')
        im.set_clim(0, 16)
plot_digits(digits.data)
```



We have nearly 1800 digits in 64 dimensions – let's create a generative model for the digits. GMM can have difficulty converging in such a high dimensional space, so we will start with an invertible dimensionality reduction algorithm on the data. Here we

will use a straightforward PCA, asking it to preserve 99% of the variance in the projected data:

```
from sklearn.decomposition import PCA
pca = PCA(0.99, whiten=True)
data = pca.fit_transform(digits.data)
data.shape

(1797, 41)
```

The result is 41 dimensions, a reduction of nearly 1/3 with almost no information loss. Given this projected data, let's use the AIC to get a gauge for the number of GMM components we should use:

```
n_components = np.arange(50, 210, 10)
models = [GMM(n, covariance_type='full', random_state=0)
          for n in n_components]
aics = [model.fit(data).aic(data) for model in models]
plt.plot(n_components, aics);
```



It appears that around 110 components minimizes the AIC; we will use this model. Let's quickly fit this to the data and confirm that it has converged:

```
gmm = GMM(110, covariance_type='full', random_state=0)
gmm.fit(data)
print(gmm.converged_)

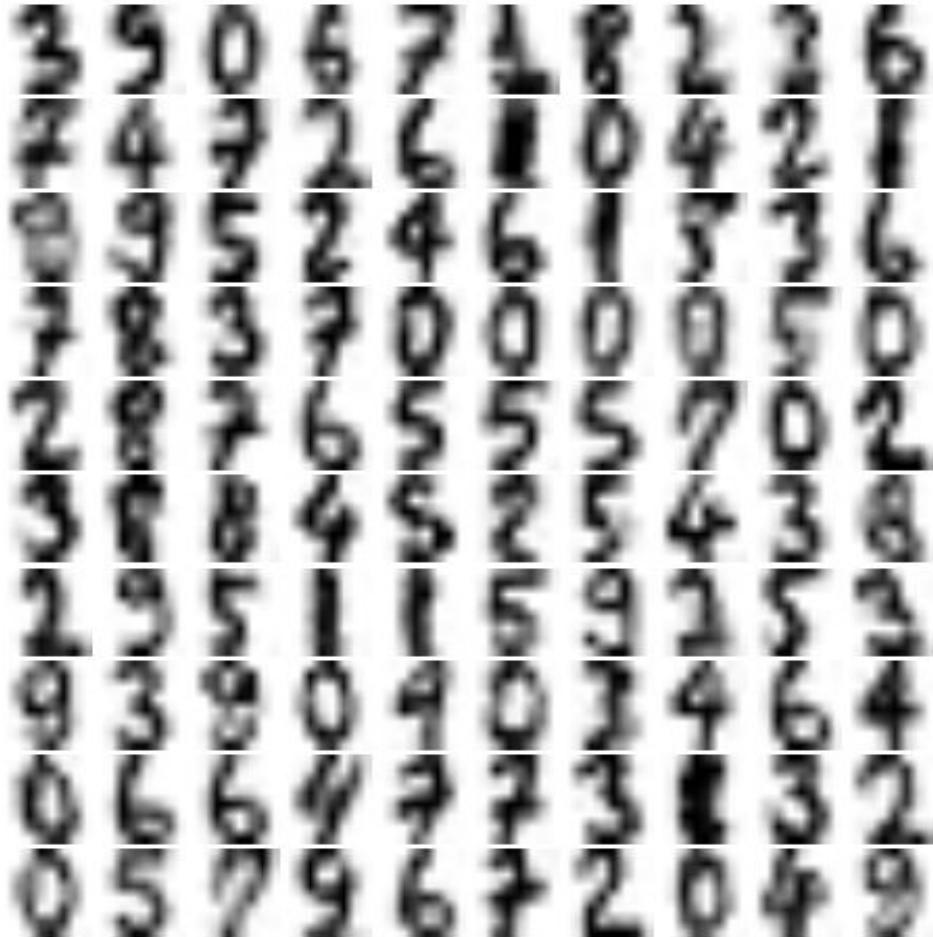
True
```

Now we can draw samples of 100 new points within this 41-dimensional projected space, using the GMM as a generative model:

```
data_new = gmm.sample(100, random_state=0)
data_new.shape
(100, 41)
```

Finally, we can use the inverse transform of the PCA object to construct the new digits:

```
digits_new = pca.inverse_transform(data_new)
plot_digits(digits_new)
```



The results for the most part look like plausible digits from the dataset! Consider what we've done here: given a sampling of hand-written digits, we have modeled the

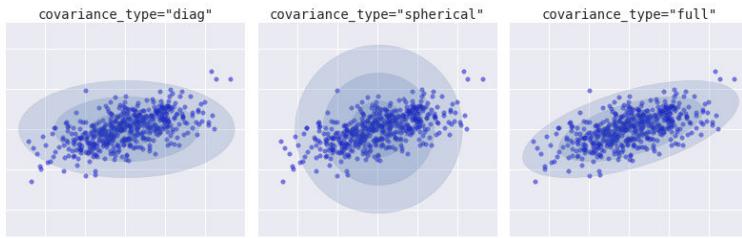
distribution of that data in such a way that we can generate brand new samples of digits from the data: these are “hand-written digits” which do not individually appear in the original dataset, but instead in some senses capture the general features of the input data. Such a generative model of digits can prove very useful as a component of a Bayesian generative classifier, as we shall see in the example of the next section.

Figures

Below is the code to create figures used in the above text.

```
import os; os.makedirs('fig', exist_ok=True)
```

Covariance Type



```
fig, ax = plt.subplots(1, 3, figsize=(14, 4), sharex=True, sharey=True)
fig.subplots_adjust(wspace=0.05)

rng = np.random.RandomState(5)
X = np.dot(rng.randn(500, 2), rng.randn(2, 2))

for i, cov_type in enumerate(['diag', 'spherical', 'full']):
    model = GMM(1, covariance_type=cov_type).fit(X)
    ax[i].axis('equal')
    ax[i].scatter(X[:, 0], X[:, 1], alpha=0.5)
    ax[i].set_xlim(-3, 3)
    ax[i].set_title('covariance_type="{}"'.format(cov_type),
                    size=14, family='monospace')
    draw_ellipse(model.means_[0], model.covars_[0], ax[i], alpha=0.2)
    ax[i].xaxis.set_major_formatter(plt.NullFormatter())
    ax[i].yaxis.set_major_formatter(plt.NullFormatter())

fig.savefig('fig/07.12-covariance-type.png')
plt.close(fig)

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

In-Depth: Kernel Density Estimation

In the previous section we covered Gaussian Mixture Models (GMM), which are a kind of hybrid between a clustering estimator and a density estimator. Recall that a density estimator is an algorithm which takes a D -dimensional dataset and produces an estimate of the D -dimensional probability distribution which that data is drawn from. The GMM algorithm accomplishes this by representing the density as a weighted sum of Gaussian distributions. *Kernel density estimation* (KDe) is in some senses an algorithm which takes the mixture-of-Gaussians idea to its logical extreme: it uses a mixture consisting of one component *per point*, resulting in an essentially non-parametric estimator of density. In this section, we will explore the motivation and uses of KDE.

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Motivating KDE: Histograms

As we mentioned above, a density estimator is an algorithm which seeks to model the probability distribution which generated a dataset. For one dimensional data, you are probably already familiar with one simple density estimator: the histogram. A histogram divides the data into discrete bins, counts the number of points which fall in each bin, and then visualizes the results in an intuitive manner.

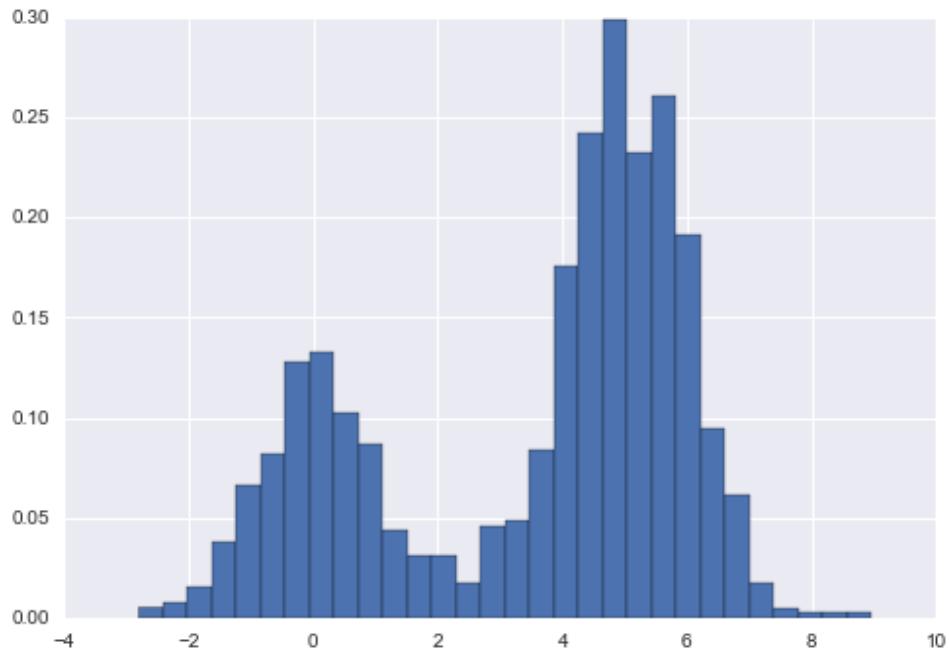
For example, let's create some data that is drawn from two normal distributions:

```
def make_data(N, f=0.3, rseed=1):
    rng = np.random.RandomState(rseed)
    x = rng.randn(N)
    x[int(f * N):] += 5
    return x

x = make_data(1000)
```

We have previously seen that the standard count-based histogram can be created with the `plt.hist()` function. By specifying the `normed` parameter of the histogram, we end up with a normalized histogram where the height of the bins does not reflect counts, but instead reflects probability density:

```
hist = plt.hist(x, bins=30, normed=True)
```



Notice that for equal binning, this normalization simply changes the scale on the y-axis, leaving the proportions between the heights essentially the same as in a histogram built from counts. This normalization is chosen so that the total area under the histogram is equal to 1, as we can confirm by looking at the output of the histogram function:

```
density, bins, patches = hist
widths = bins[1:] - bins[:-1]
(density * widths).sum()
1.0
```

One of the issues with using a histogram as a density estimator is that the choice of bin size and location can lead to representations which have qualitatively different features. For example, if we look at a version of the above data with only 20 points, the choice of how to draw the bins can lead to an entirely different interpretation of the data! Consider this example,

```
x = make_data(20)
bins = np.linspace(-5, 10, 10)

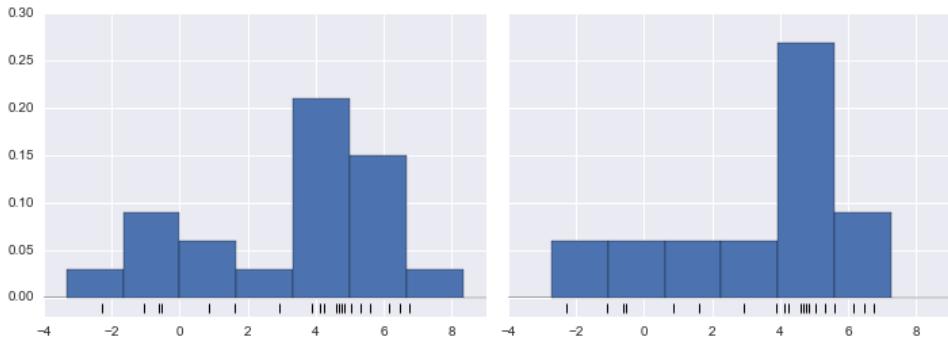
fig, ax = plt.subplots(1, 2, figsize=(12, 4),
                      sharex=True, sharey=True,
                      subplot_kw={'xlim':(-4, 9),
                                 'ylim':(-0.02, 0.3)})

fig.subplots_adjust(wspace=0.05)
for i, offset in enumerate([0.0, 0.6]):
```

```

ax[i].hist(x, bins=bins + offset, normed=True)
ax[i].plot(x, np.full_like(x, -0.01), '|k',
            markeredgewidth=1)

```



On the left, the histogram makes clear that this is a bimodal distribution. On the right, we see a unimodal distribution with a long tail. Without being told, you would probably not guess that these two histograms were built from the same data: with that in mind, how can you trust the intuition that histograms confer? And how might we improve on this?

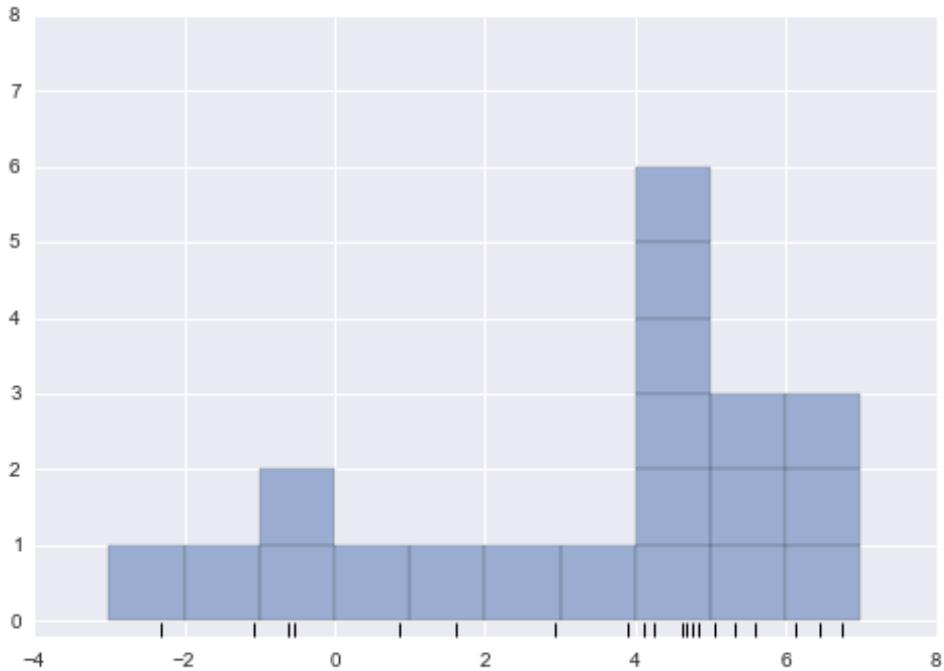
Stepping back, we can think of a histogram as a stack of blocks, where we put one block within each bin on top of each point in the dataset. Let's view this directly:

```

fig, ax = plt.subplots()
bins = np.arange(-3, 8)
ax.plot(x, np.full_like(x, -0.1), '|k',
         markeredgewidth=1)
for count, edge in zip(*np.histogram(x, bins)):
    for i in range(count):
        ax.add_patch(plt.Rectangle((edge, i), 1, 1,
                                  alpha=0.5))
ax.set_xlim(-4, 8)
ax.set_ylim(-0.2, 8)

```

(-0.2, 8)

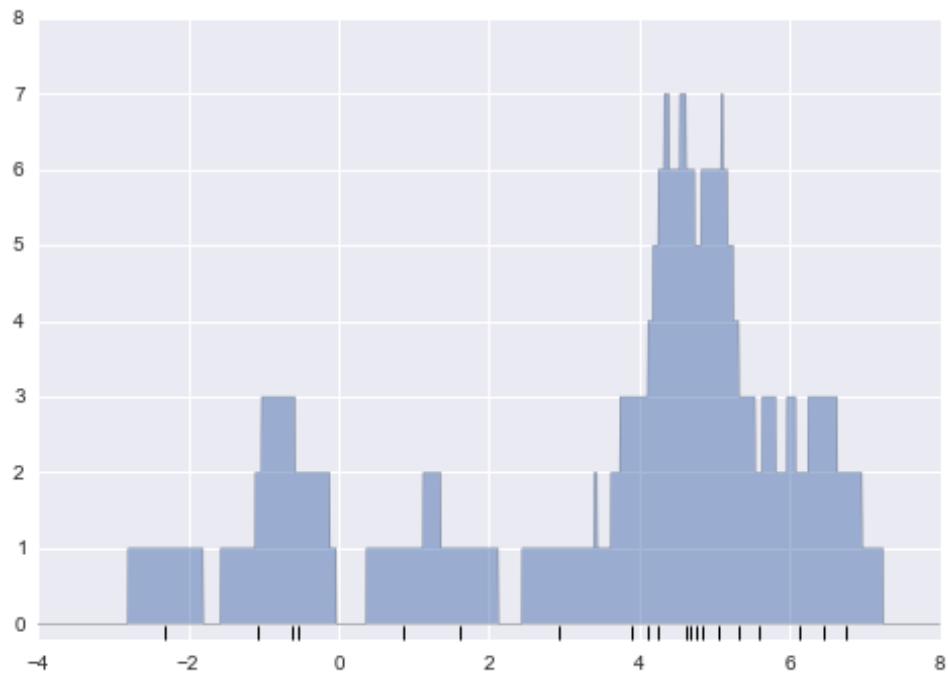


The problem with our two binnings above stem from the fact that the height of the block stack often reflects not on the actual density of points nearby, but on coincidences of how the bins align with the data points. This mis-alignment between points and their blocks is a potential cause of the poor histogram results seen above. But what if, instead of stacking the blocks aligned with the *bins*, we were to stack the blocks aligned with the *points they represent*? If we do this, the blocks won't be aligned, but we can add their contributions at each location along the x-axis to find the result. Let's try this:

```
x_d = np.linspace(-4, 8, 2000)
density = sum((abs(xi - x_d) < 0.5) for xi in x)

plt.fill_between(x_d, density, alpha=0.5)
plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

plt.axis([-4, 8, -0.2, 8]);
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```

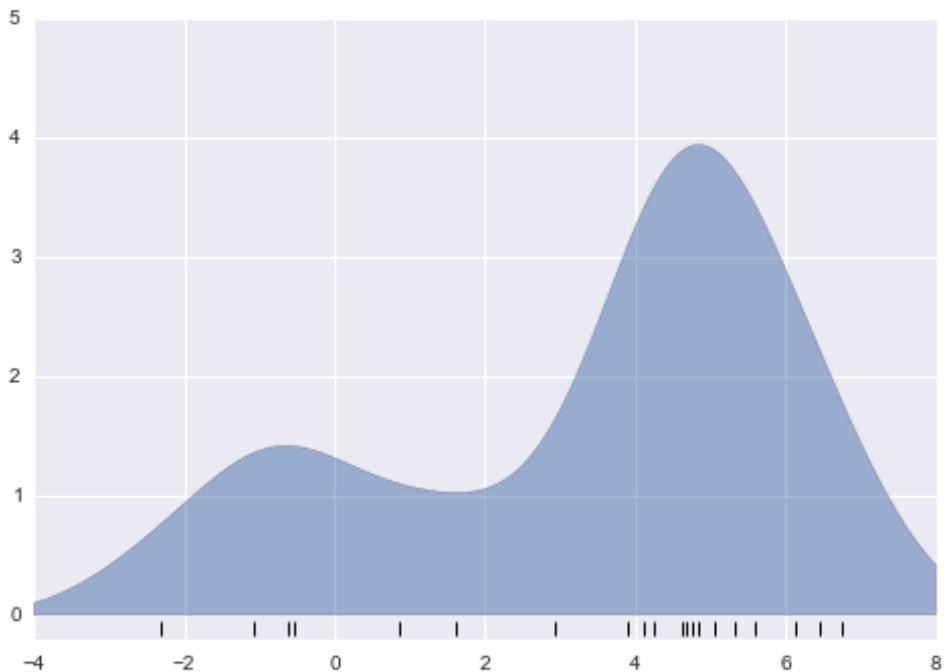


The result looks a bit messy, but is a much more robust reflection of the actual data characteristics than is the standard histogram. Still, the rough edges are not aesthetically pleasing, nor are they reflective of any true properties of the data. In order to smooth them out, we might decide to replace the blocks at each location with a smooth function, like a Gaussian. Let's use a standard normal curve at each point instead of a block:

```
from scipy.stats import norm
x_d = np.linspace(-4, 8, 1000)
density = sum(norm(xi).pdf(x_d) for xi in x)

plt.fill_between(x_d, density, alpha=0.5)
plt.plot(x, np.full_like(x, -0.1), '|k', markeredgewidth=1)

plt.axis([-4, 8, -0.2, 5]);
/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



This smoothed-out plot, with a Gaussian distribution contributed at the location of each input point, gives a much more accurate idea of the shape of the data distribution, and one which has much less variance (i.e. changes much less in response to differences in sampling).

These last two plots are examples of Kernel Density Estimation in one dimension: the first uses a so-called “tophat” kernel and the second uses a Gaussian kernel. Let’s look in more detail at Kernel Density Estimation below.

Kernel Density Estimation in Practice

The free parameters of Kernel Density Estimation are the *kernel*, and the *kernel bandwidth*, which controls the size of the kernel at each point. In practice, there are many kernels you might use for a kernel density estimation: in particular, the scikit-learn KDE implementation supports one of six kernels, which you can read about in scikit-learn’s [Density Estimation](#) documentation.

While there are several versions of kernel density estimation implemented in Python (notably in the SciPy and StatsModels packages), I prefer to use scikit-learn’s version because of its power and flexibility. It is implemented in the `sklearn.neighbors.KernelDensity` estimator, which handles KDE in multiple dimensions with one of six kernels and one of a couple dozen distance metrics. Because KDE can be fairly com-

putationally intensive, the scikit-learn estimator uses a tree-based algorithm under the hood and can trade-off computation time for accuracy using the `atol` and `rtol` parameters. The kernel bandwidth, which is a free parameter, can be determined using scikit-learn's standard cross validation tools as we will see below.

Let's first show a simple example of replicating the above plot using the scikit-learn `KernelDensity` estimator:

```
from sklearn.neighbors import KernelDensity

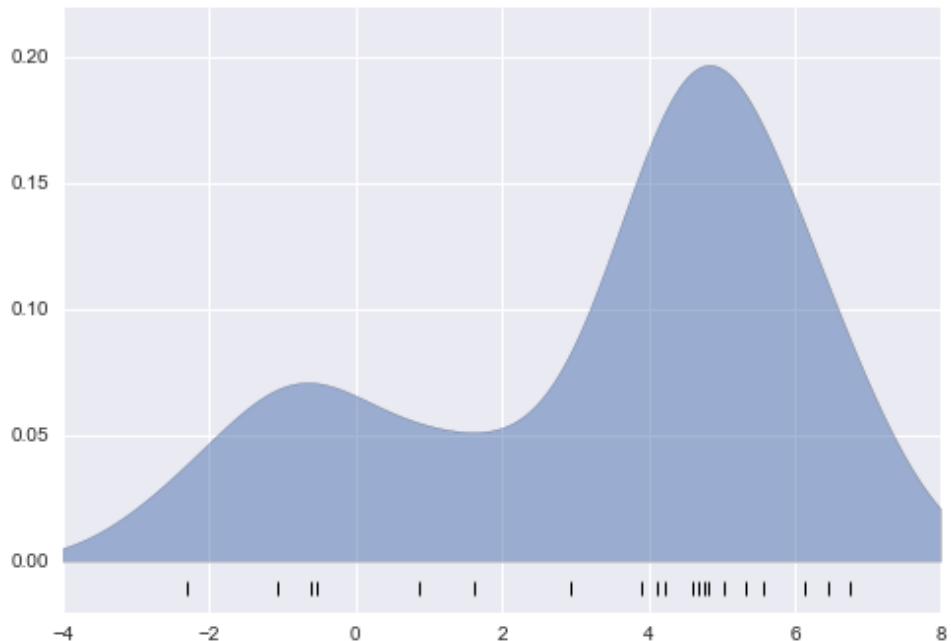
# instantiate and fit the KDE model
kde = KernelDensity(bandwidth=1.0, kernel='gaussian')
kde.fit(x[:, None])

# score_samples returns the log of the probability density
logprob = kde.score_samples(x_d[:, None])

plt.fill_between(x_d, np.exp(logprob), alpha=0.5)
plt.plot(x, np.full_like(x, -0.01), '|k', markeredgewidth=1)
plt.ylim(-0.02, 0.22)

(-0.02, 0.22)
```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):



The result here is normalized such that the area under the curve is equal to 1.

Selecting the Bandwidth via Cross-Validation

The choice of bandwidth within KDE is extremely important to finding a suitable density estimate, and is the knob that controls the bias-variance tradeoff in the estimate of density: too narrow a bandwidth leads to a high-variance estimate (i.e. over-fitting), where the presence or absence of a single point makes a large difference. Too wide a bandwidth leads to a high-bias estimate (i.e. under-fitting) where the structure in the data is washed-out by the wide kernel.

There is a long history in statistics of methods to quickly estimate the best bandwidth based on rather stringent assumptions about the data: if you look up the KDE implementations in the SciPy and StatsModels packages, for example, you will see implementations based on some of these rules.

In machine learning contexts, we've seen that such hyperparameter tuning often is done empirically via a cross-validation approach. With this in mind, the `KernelDensity` estimator in scikit-learn is designed such that it can be used directly within the scikit-learn's standard grid search tools. Below we will use `GridSearchCV` to optimize the bandwidth for the above dataset. Because we are looking at such a small dataset, we will use leave-one-out cross validation, which minimizes the reduction in training set size for each cross-validation trial:

```
from sklearn.grid_search import GridSearchCV
from sklearn.cross_validation import LeaveOneOut

bandwidths = 10 ** np.linspace(-1, 1, 100)
grid = GridSearchCV(KernelDensity(kernel='gaussian'),
                     {'bandwidth': bandwidths},
                     cv=LeaveOneOut(len(x)))
grid.fit(x[:, None]);
```

Now we can find the choice of bandwidth which maximizes the score (which in this case defaults to the the log-likelihood):

```
grid.best_params_
{'bandwidth': 1.1233240329780276}
```

The optimal bandwidth is actually very close to what we used in the example plot above, where the bandwidth was 1.0 (i.e. the default width of `scipy.stats.norm`).

Example: KDE on a Sphere

Perhaps the most common use of KDE is in graphically representing distributions of points. For example, in the Seaborn visualization library, KDE is built-in and automatically used to help visualize points in one and two dimensions (see Section X.X).

Here we will look at a slightly more sophisticated use of KDE for visualization of distributions. We will make use of some geographic data that can be loaded with scikit-learn: the geographic distributions of recorded observations of two South American mammals, “*Bradypus variegatus*” (known as the Brown-throated Sloth) and “*Microtomyces minutus*” (known as the Forest Small Rice Rat).

With scikit-learn, we can fetch this data as follows:

```
from sklearn.datasets import fetch_species_distributions

data = fetch_species_distributions()

# Get matrices/arrays of species IDs and locations
latlon = np.vstack([data.train['dd lat'],
                   data.train['dd long']]).T
species = np.array([d.decode('ascii').startswith('micro')
                    for d in data.train['species']], dtype='int')
```

With this data loaded, we can use the basemap toolkit (see Section X.X) to plot the observed locations of these two species on the map of South America.

```
from mpl_toolkits.basemap import Basemap
from sklearn.datasets.species_distributions import construct_grids

xgrid, ygrid = construct_grids(data)

# plot coastlines with basemap
m = Basemap(projection='cyl', resolution='c',
            llcrnrlat=ygrid.min(), urcrnrlat=ygrid.max(),
            llcrnrlon=xgrid.min(), urcrnrlon=xgrid.max())
m.drawmapboundary(fill_color='#DDDEFF')
m.fillcontinents(color="#FFEEDD")
m.drawcoastlines(color='gray', zorder=2)
m.drawcountries(color='gray', zorder=2)

# plot locations
m.scatter(latlon[:, 1], latlon[:, 0], zorder=3,
          c=species, cmap='rainbow', latlon=True);

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
    if self._edgecolors == str('face'):
```



Unfortunately, this doesn't give a very good idea of the density of the species, because points in the species range may overlap one another. You may not realize it by looking at this plot, but there are over 1600 points shown here!

Let's use kernel density estimation to show this distribution in a more interpretable way: as a smooth indication of density on the map. Because the coordinate system here lies on a spherical surface rather than a flat plane, we will use the haversine distance metric, which will correctly represent distances on a curved surface.

There is a bit of boilerplate code here (one of the disadvantages of the basemap toolkit) but the meaning of each code block should be clear:

```
# Set up the data grid for the contour plot
X, Y = np.meshgrid(xgrid[::5], ygrid[::5][::-1])
land_reference = data.coverages[6][::5, ::5]
land_mask = (land_reference > -9999).ravel()
xy = np.vstack([Y.ravel(), X.ravel()]).T
xy = np.radians(xy[land_mask])
```

```

# Create two side-by-side plots
fig, ax = plt.subplots(1, 2)
fig.subplots_adjust(left=0.05, right=0.95, wspace=0.05)
species_names = ['Bradypus Variegatus', 'Microtromys Minutus']
cmaps = ['Purples', 'Reds']

for i, axi in enumerate(ax):
    axi.set_title(species_names[i])

# plot coastlines with basemap
m = Basemap(projection='cyl', llcrnrlat=Y.min(),
            urcrnrlat=Y.max(), llcrnrlon=X.min(),
            urcrnrlon=X.max(), resolution='c', ax=axi)
m.drawmapboundary(fill_color='#DDEEFF')
m.drawcoastlines()
m.drawcountries()

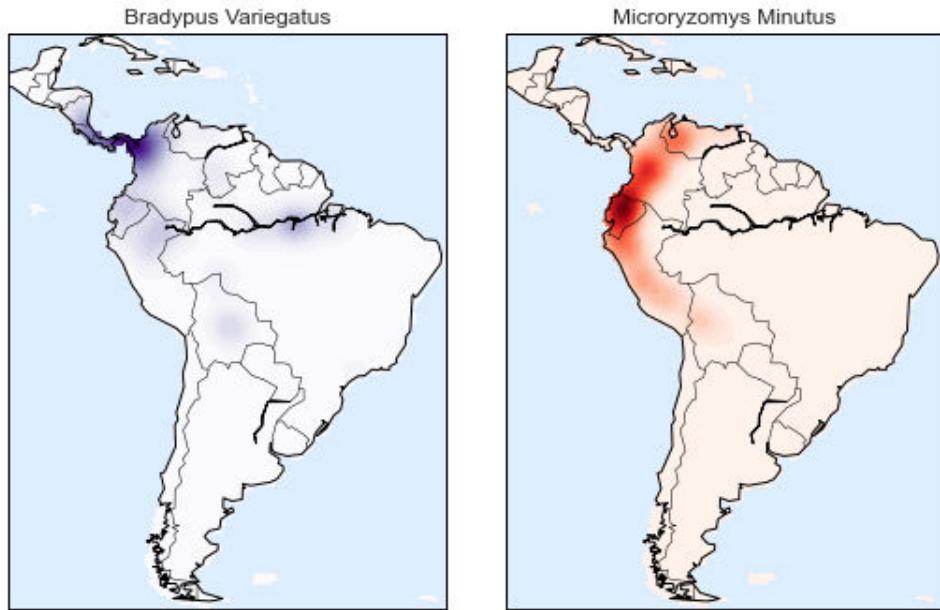
# construct a spherical kernel density estimate of the distribution
kde = KernelDensity(bandwidth=0.03, metric='haversine')
kde.fit(np.radians(latlon[species == i]))

# evaluate only on the land: -9999 indicates ocean
Z = np.full(land_mask.shape[0], -9999.0)
Z[land_mask] = np.exp(kde.score_samples(xy))
Z = Z.reshape(X.shape)

# plot contours of the density
levels = np.linspace(0, Z.max(), 25)
axi.contourf(X, Y, Z, levels=levels, cmap=cmaps[i])

```

/Users/jakevdp/anaconda/envs/python3.4/lib/python3.4/site-packages/matplotlib/collections.py:590:
if self._edgecolors == str('face'):



Compared to the simple scatter plot we initially used, this visualization gives us a much more clear view of the geographical distribution of observations of these two species.

Example: Not-So-Naive Bayes

This example looks at Bayesian generative classification with KDE, and demonstrates how to use the scikit-learn architecture to create a custom estimator.

In Section X.X, we took a look at Naive Bayesian classification, in which we created a simple generative model for each class, and used these models to build a fast classifier. For Naive Bayes, the generative model is a simple axis-aligned Gaussian. With a density estimation algorithm like KDE, we can remove the “naive” element and perform the same classification with a more sophisticated generative model for each class. It’s still Bayesian classification, but it’s no longer naive.

The general approach for generative classification is this:

1. Split the training data by label
2. For each set, fit a KDE to obtain a generative model of the data. This allows you for any observation x and label y to compute a likelihood $P(x \mid y)$.
3. From the number of examples of each class in the training set, compute the *class prior*, $P(y)$.

- For an unknown point x , the posterior probability for each class is $P(y | x) \propto P(x | y)P(y)$. The class which maximizes this posterior is the label assigned to the point

The algorithm is straightforward and intuitive to understand; the more difficult piece is couching it within the scikit-learn framework in order to make use of the grid search and cross-validation architecture.

Below is the code that implements this algorithm within the scikit-learn framework; we will step through it following the code block:

```
from sklearn.base import BaseEstimator, ClassifierMixin

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Bayesian Generative Classification based on KDE

    Parameters
    -----
    bandwidth : float
        the kernel bandwidth within each class
    kernel : str
        the kernel name, passed to KernelDensity
    """

    def __init__(self, bandwidth=1.0, kernel='gaussian'):
        self.bandwidth = bandwidth
        self.kernel = kernel

    def fit(self, X, y):
        self.classes_ = np.sort(np.unique(y))
        training_sets = [X[y == yi] for yi in self.classes_]
        self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                      kernel=self.kernel).fit(Xi)
                       for Xi in training_sets]
        self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                           for Xi in training_sets]
        return self

    def predict_proba(self, X):
        logprobs = np.array([model.score_samples(X)
                            for model in self.models_]).T
        result = np.exp(logprobs + self.logpriors_)
        return result / result.sum(1, keepdims=True)

    def predict(self, X):
        return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

The Anatomy of a Custom Estimator

Let's step through this code and discuss the essential features:

```
from sklearn.base import BaseEstimator, ClassifierMixin
```

```

class KDEClassifier(BaseEstimator, ClassifierMixin):
    """Bayesian Generative Classification based on KDE

    Parameters
    -----
    bandwidth : float
        the kernel bandwidth within each class
    kernel : str
        the kernel name, passed to KernelDensity
    """

```

Each estimator in scikit-learn is a class, and it is most convenient for this class to inherit from the `BaseEstimator` class as well as the appropriate mixin, which provides standard functionality. For example, among other things, here the `BaseEstimator` contains the logic necessary to clone/copy an estimator for use in a cross validation procedure, and `ClassifierMixin` defines a default `score()` method used by such routines. We also provide a doc string, which will be captured by IPython's help functionality (see Section X.X).

Next comes the class initialization method:

```

def __init__(self, bandwidth=1.0, kernel='gaussian'):
    self.bandwidth = bandwidth
    self.kernel = kernel

```

This is the actual code that is executed when the object is instantiated with `KDEClassifier()`. In scikit-learn, it is important that **initialization contains no operations** other than assigning the passed values by name to `self`. This is due to the logic contained in `BaseEstimator` required for cloning and modifying estimators for cross validation, grid search, and other functions. Similarly, all arguments to `__init__` should be explicit: i.e. `*args` or `**kwargs` should be avoided, as they will not be correctly handled within cross validation routines.

Next comes the `fit()` method, where we handle training data:

```

def fit(self, X, y):
    self.classes_ = np.sort(np.unique(y))
    training_sets = [X[y == yi] for yi in self.classes_]
    self.models_ = [KernelDensity(bandwidth=self.bandwidth,
                                  kernel=self.kernel).fit(Xi)
                   for Xi in training_sets]
    self.logpriors_ = [np.log(Xi.shape[0] / X.shape[0])
                      for Xi in training_sets]
    return self

```

Here we find the unique classes in the training data, train a `KernelDensity` model for each class, and compute the class priors based on the number of input samples. Finally, `fit()` should always return `self` so that we can chain commands, e.g.

```
label = model.fit(X, y).predict(X)
```

Notice that each persistent result of the fit is stored with a trailing underscore (e.g. `self.logpriors_`). This is a convention used in scikit-learn so that you can quickly scan the members of an estimator (using, e.g. IPython's tab-completion) and see exactly which members are fit to training data.

Finally, we have the logic for predicting labels on new data:

```
def predict_proba(self, X):
    logprobs = np.vstack([model.score_samples(X)
                          for model in self.models_]).T
    result = np.exp(logprobs + self.logpriors_)
    return result / result.sum(1, keepdims=True)

def predict(self, X):
    return self.classes_[np.argmax(self.predict_proba(X), 1)]
```

Because this is a probabilistic classifier, we first implement `predict_proba()` which returns an array of class probabilities of shape [`n_samples`, `n_classes`]. Entry [`i`, `j`] of this array is the posterior probability that sample `i` is a member of class `j`, computed by multiplying the likelihood by the class prior and normalizing.

Finally, the `predict()` method uses these probabilities and simply returns the class with the largest probability.

Demonstrating the Custom Estimator

Let's try this custom estimator on a problem we have seen before: the classification of hand-written digits. Here we will load the digits, and compute the cross-validation score for a range of candidate bandwidths using the `GridSearchCV` meta-estimator (see Section X.X):

```
from sklearn.datasets import load_digits
from sklearn.grid_search import GridSearchCV

digits = load_digits()

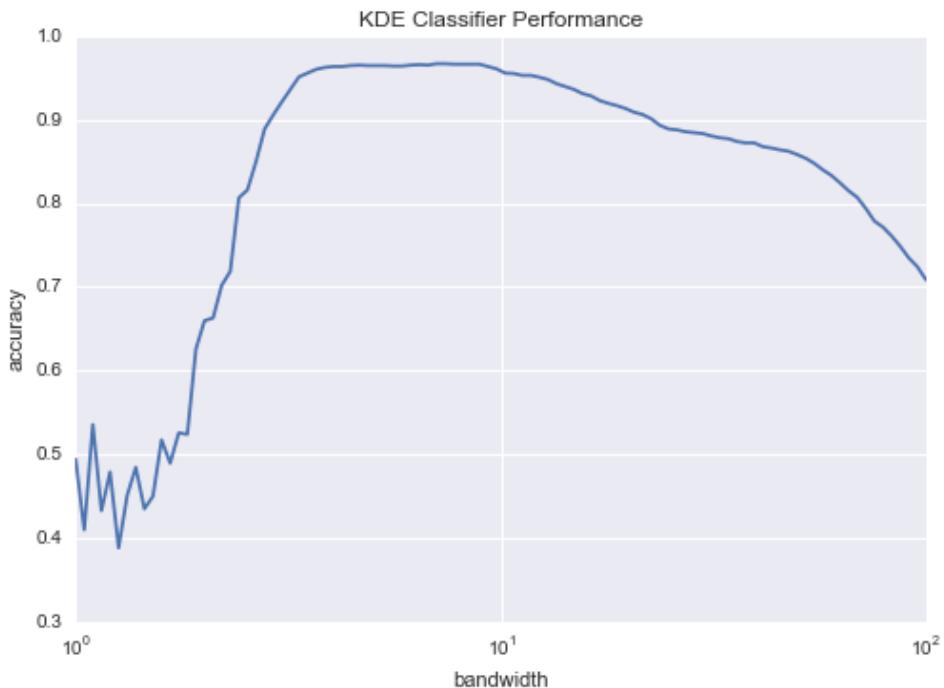
bandwidths = 10 ** np.linspace(0, 2, 100)
grid = GridSearchCV(KDEClassifier(), {'bandwidth': bandwidths})
grid.fit(digits.data, digits.target)

scores = [val.mean_validation_score for val in grid.grid_scores_]
```

Next we can plot the cross-validation score as a function of bandwidth:

```
plt.semilogx(bandwidths, scores)
plt.xlabel('bandwidth')
plt.ylabel('accuracy')
plt.title('KDE Classifier Performance')
print(grid.best_params_)
print('accuracy =', grid.best_score_)
```

```
{'bandwidth': 7.0548023107186433}  
accuracy = 0.966611018364
```



We see that this not-so-naive Bayesian classifier reaches a cross-validation accuracy of just over 96%; this is compared to less than 80% for the naive Bayesian classification from Section X.X:

```
from sklearn.naive_bayes import GaussianNB  
from sklearn.cross_validation import cross_val_score  
cross_val_score(GaussianNB(), digits.data, digits.target).mean()  
0.79134615910759132
```

One benefit of such a generative classifier is interpretability of results: for each unknown sample, we not only get a probabilistic classification, but a *full model* of the distribution of points we are comparing it to! If desired, this offers an intuitive window into the reasons for a particular classification that an algorithm like SVM or Random Forest tends to obscure.

If you would like to take this further, there are some improvements that could be made to our KDE classifier model:

- we could allow the bandwidth in each class to vary independently

- we could optimize these bandwidths not based on their prediction score, but on the likelihood of the training data under the generative model within each class (i.e. use the scores from `KernelDensity` itself rather than the global prediction accuracy)

Finally, if you want some practice building your own estimator, you might tackle building a similar Bayesian classifier using Gaussian Mixture Models instead of KDE.

Feature Engineering: Working with Images

This chapter has explored a number of the central concepts and algorithms of machine learning. But moving from these concepts to real-world application can be a challenge. Real-world datasets are noisy & heterogeneous, may have missing features, and data may be in a form that is difficult to map to a `[n_samples, n_features]` matrix. Before applying any of the methods discussed here, you must first extract these features from your data: there is no formula for how to do this that applies across all domains, and thus this is where you as a data scientist must exercise your own intuition and expertise.

One interesting and compelling application of machine learning is to images, and we have already seen a few examples of this where pixel-level features are used for classification. In the real world, data is rarely so uniform and simple pixels will not be suitable: this has led to a large literature on *feature extraction* methods for image data.

In this section, we will take a look at one such feature extraction technique, the **Histogram of Oriented Gradients** (HOG), which transforms image pixels into a vector representation that is sensitive to broadly-informative image features regardless of confounding factors like illumination. Below, we will use these features to develop a simple sliding-window face detection pipeline, using Machine Learning algorithms and concepts we've seen through this chapter.

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

HOG features

The Histogram of Gradients is a straightforward feature extraction procedure which was developed in the context of identifying pedestrians within images. HOG involves the following steps:

1. optionally pre-normalize images. This leads to features which resist dependence on variations in illumination.
2. convolve the image with two filters which are sensitive to horizontal and vertical brightness gradients. These capture edge, contour, and texture information.

3. Subdivide the image into cells of a predetermined size, and compute a histogram of the gradient orientations within each cell.
4. Normalize the histograms in each cell by comparing to the block of neighboring cells. This further suppresses the effect of illumination across the image.
5. Construct a 1D feature vector from the information in each cell.

A fast HOG extractor is built-in to the scikit-image project, and we can try it out relatively quickly and visualize the oriented gradients within each cell:

```
from skimage import data, color, feature
import skimage.data

image = color.rgb2gray(data.chelsea())
hog_vec, hog_vis = feature.hog(image, visualise=True)

fig, ax = plt.subplots(1, 2, figsize=(12, 6),
                      subplot_kw=dict(xticks=[], yticks=[]))
ax[0].imshow(image, cmap='gray')
ax[0].set_title('input image')

ax[1].imshow(hog_vis)
ax[1].set_title('visualization of HOG features');
```



HOG in Action: A Simple Face Detector

Using these HOG features, we can build up a simple face detection algorithm with any classifier; here we will use a linear support vector machine. The steps are as follows:

1. Obtain a set of image thumbnails of faces to constitute “positive” training samples.
2. Obtain a set of image thumbnails of non-faces to constitute “negative” training samples.
3. Extract HOG features from these training samples.
4. Train a linear SVM classifier on these samples.

5. For an “unknown” image, pass a sliding window across the image, using the model to evaluate whether that window contains a face or not.
6. If detections overlap, combine them into a single window.

Let's go through these steps and try it out:

1. Obtain a set of positive training samples

Let's start by finding some positive training samples that show a variety of faces. We have one easy set of data to work with: the Labeled Faces in the Wild dataset which can be downloaded by scikit-learn:

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people()
positive_patches = faces.images
positive_patches.shape

(13233, 62, 47)
```

This gives us a sample of 13000 face images to use for training.

2. Obtain a set of negative training samples

Next we need a set of similarly-sized thumbnails which *do not* have a face in them. One way to do this is to take any corpus of input images, and extract thumbnails from them at a variety of scales. Here we can use some of the images shipped with scikit-image, along with scikit-learn's patch extractor:

```
from skimage import data, transform

imgs_to_use = ['camera', 'text', 'coins', 'moon',
               'page', 'clock', 'immunohistochemistry',
               'chelsea', 'coffee', 'hubble_deep_field']
images = [color.rgb2gray(getattr(data, name)())
          for name in imgs_to_use]

from sklearn.feature_extraction.image import PatchExtractor

def extract_patches(img, N, scale=1.0, patch_size=positive_patches[0].shape):
    extracted_patch_size = tuple((scale * np.array(patch_size)).astype(int))
    extractor = PatchExtractor(patch_size=extracted_patch_size,
                               max_patches=N, random_state=0)
    patches = extractor.transform(img[np.newaxis])
    if scale != 1:
        patches = np.array([transform.resize(patch, patch_size)
                           for patch in patches])
    return patches

negative_patches = np.vstack([extract_patches(im, 1000, scale)
                             for im in images for scale in [0.5, 1.0, 2.0]])
negative_patches.shape

(30000, 62, 47)
```

We now have 30000 suitable image patches which do not contain faces. Let's take a look at a few of them to get an idea of what they look like:

```
fig, ax = plt.subplots(6, 10)
for i, axi in enumerate(ax.flat):
    axi.imshow(negative_patches[500 * i], cmap='gray')
    axi.axis('off')
```



Our hope is that these would sufficiently cover the space of “non-faces” that our algorithm is likely to see.

3. Combine sets and extract HOG features

Now that we have these positive samples and negative samples, we can combine them and compute HOG features. This step takes a little while, because the HOG features are a nontrivial computation for each image:

```
from itertools import chain
from skimage import feature
X_train = np.array([feature.hog(im)
                    for im in chain(positive_patches,
                                    negative_patches)])
y_train = np.zeros(X_train.shape[0])
y_train[:positive_patches.shape[0]] = 1
X_train.shape
```

```
(43233, 1215)
```

We are left with 43,000 training samples in 1215 dimensions, and we now have our data in a form that we can feed into scikit-learn!

4. Training a Support Vector Machine

Next we use the tools we have been exploring in this chapter to create a classifier of thumbnail patches. For such a high-dimensional binary classification task, a Linear support vector machine is a good choice. We will use scikit-learn's `LinearSVC`, because in comparison to `SVC` it has better scaling for large number of samples.

First, though, let's use a simple Gaussian Naive Bayes to get a quick baseline:

```
from sklearn.naive_bayes import GaussianNB
from sklearn.cross_validation import cross_val_score

cross_val_score(GaussianNB(), X_train, y_train)
array([ 0.94587468,  0.87856499,  0.94184998])
```

We see that on our training data, even a simple naive Bayes algorithm gets us to above 90% accuracy. Let's try the support vector machine, with a grid search over a few choices of the C parameter

```
from sklearn.svm import LinearSVC
from sklearn.grid_search import GridSearchCV
grid = GridSearchCV(LinearSVC(), {'C': [1.0, 2.0, 4.0, 8.0]})
grid.fit(X_train, y_train)
grid.best_score_

0.9862373649758287
grid.best_params_
{'C': 4.0}
```

Let's take the best estimator and re-train it on the full dataset:

```
model = grid.best_estimator_
model.fit(X_train, y_train)

LinearSVC(C=4.0, class_weight=None, dual=True, fit_intercept=True,
          intercept_scaling=1, loss='squared_hinge', max_iter=1000,
          multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
          verbose=0)
```

5. Finding faces in a new image

Now that we have this model in place, let's grab a new image and see how the model does. We will use one portion of the astronaut image for simplicity (see discussion of this in caveats, below), and run a sliding window over it and evaluate each patch:

```
test_image = skimage.data.astronaut()
test_image = skimage.color.rgb2gray(test_image)
test_image = skimage.transform.rescale(test_image, 0.5)
test_image = test_image[:160, 40:180]

plt.imshow(test_image, cmap='gray')
plt.axis('off');
```



Next let's create a window which iterates over patches of this image, and compute HOG features for each patch:

```
def sliding_window(img, patch_size=positive_patches[0].shape,
                  istep=2, jstep=2, scale=1.0):
    Ni, Nj = (int(scale * s) for s in patch_size)
    for i in range(0, img.shape[0] - Ni, istep):
        for j in range(0, img.shape[1] - Ni, jstep):
            patch = img[i:i + Ni, j:j + Nj]
            if scale != 1:
```

```

        patch = transform.resize(patch, patch_size)
        yield (i, j), patch

indices, patches = zip(*sliding_window(test_image))
patches_hog = np.array([feature.hog(patch) for patch in patches])
patches_hog.shape

(1911, 1215)

```

Finally, we can take these HOG-featured patches and use our model to evaluate whether each patch contains a face:

```

labels = model.predict(patches_hog)
labels.sum()

30.0

```

We see that out of nearly 2000 patches, we have found 30 detections. Let's use the information we have about these patches to show where they lie on our test image, drawing them as rectangles:

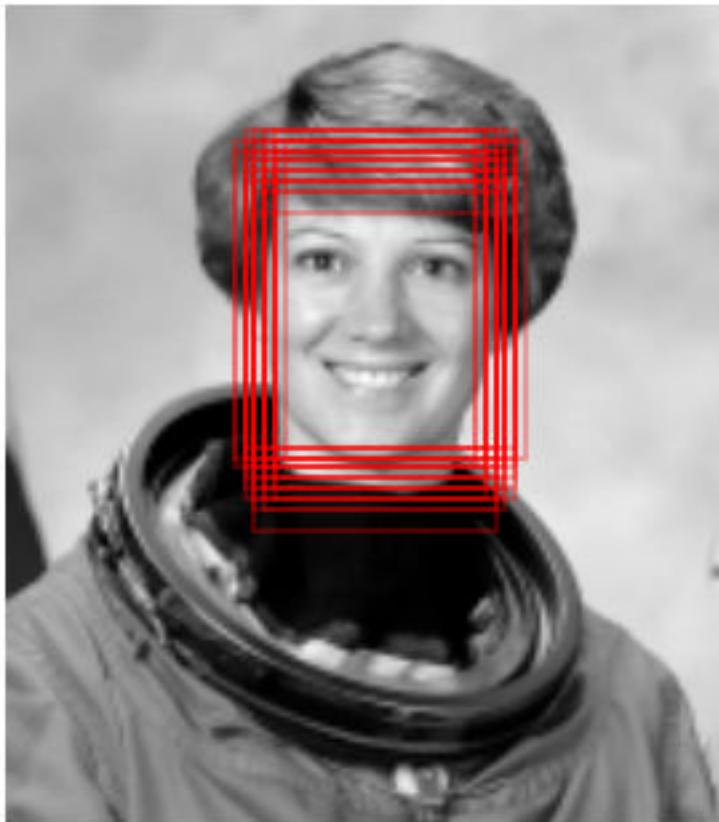
```

fig, ax = plt.subplots()
ax.imshow(test_image, cmap='gray')
ax.axis('off')

Ni, Nj = positive_patches[0].shape
indices = np.array(indices)

for i, j in indices[labels == 1]:
    ax.add_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',
                               alpha=0.3, lw=2, facecolor='none'))

```



All of the detected patches overlap and bound the face in the image! Not bad for a few lines of Python.

Caveats and Improvements

If you dig a bit deeper into the above code and examples, you'll see that we still have a bit of work before we can claim a production-ready face detector. There are several issues with what we've done, and several improvements that could be made. In particular:

1. Our training set, especially for negative features, is not very complete

The central issue is that there are many face-like textures which are not in the training set, and so our current model is very prone to false positives. You can see this if you try out the above algorithm on the *full* astronaut image: the current model leads to many false detections in other regions of the image.

We might imagine addressing this by adding a wider variety of images to the negative set. This would be a very good approach. Another way to address this is to use a more directed approach, such as *hard negative mining*. In hard negative mining, we take a new set of images that our classifier has not seen, find all the patches representing false positives, and explicitly add them as negative instances in the training set before re-training the classifier.

2. Our current pipeline searches only at one scale

As currently written, our algorithm will miss faces that are not approximately 62 x 47 pixels. This can be straightforwardly addressed by using sliding windows of a variety of sizes, and re-sizing each patch using, e.g. `skimage.transform.resize` before feeding it into the model. In fact, the `sliding_window()` utility used above is already built with this in mind.

3. We should combine overlapped detection patches

For a production-ready pipeline, we would prefer not to have 30 detections of the same face, but to somehow reduce overlapping groups of detections down to a single detection. This could be done via an unsupervised clustering approach (MeanShift Clustering is one good candidate for this), or via a procedural approach such as *non-maximum suppression*, an algorithm common in machine vision.

4. The pipeline should be streamlined

Once we address the above issues, it would also be nice to create a more streamlined pipeline for ingesting training images and predicting sliding-window outputs. This is where Python as a data science tool really shines: with a bit of work, we could take our prototype code above and package it with a well-designed object-oriented API that give the user the ability to use this easily. I will leave this as a proverbial “exercise for the reader”.

Finally, I should add that HOG and other procedural feature extraction methods for images are no longer the state-of-the-art. Instead, many modern object detection pipelines use variants of deep neural networks, which in a sense learn what features to extract based on a large training set of input images. An intro to these deep neural net methods is conceptually (and computationally!) beyond the scope of this section, but it is an interesting subject to read about.

Learning More

This chapter has been a quick tour of machine learning in Python, primarily using the tools within the scikit-learn library. As long as the chapter is, it is still too short to cover many interesting and important algorithms, approaches, and discussions. Here

I want to suggest some resources to learn more about machine learning for those who are interested.

Machine Learning in Python

To learn more about machine learning in Python, I'd suggest some of the following resources:

- <http://scikit-learn.org>: the scikit-learn website has an impressive breadth of documentation and examples covering some of the models discussed here, and much, much more. If you want a brief survey of the most important and often-used ML algorithms, this website is a good place to start.
- Free Tutorial Videos: Scikit-learn and other machine learning topics are perennial favorites in the tutorial tracks of many Python-focused conference series, in particular the PyCon, SciPy, and PyData conferences. You can find the most recent ones via a simple web search, or browse most of what's available at <http://pyvideo.org>.
- *Introductory Machine Learning* by Andy Mueller (O'Reilly, 2016) [TODO: update reference] is a book which includes a fuller treatment of the topics in this chapter, written by one of the most prolific developers on the scikit-learn team. If you're interested in reviewing the fundamentals of Machine Learning and pushing the scikit-learn toolkit to its limits, this is a great resource
- *Python Machine Learning* by Sebastian Raschka (Packt, 2015) is a book which focuses less on Scikit-learn itself, and more on the breadth of Machine Learning tools available in Python. In particular, there is some very useful discussion on how to scale Python-based machine learning approaches to large and complex datasets.

General Machine Learning

Of course, Machine learning is much broader than just the Python world. There are many good resources to take your knowledge further, and I will highlight a few here:

- *Machine Learning*, taught by Andrew Ng (Coursera), is a very clearly-taught free online course which covers the basics of Machine Learning from an algorithmic perspective. It assumes undergraduate-level understanding of mathematics and programming, and steps through detailed considerations of some of the most important ML algorithms. Homework assignments, which are algorithmically graded, have you actually implement some of these models yourself.

- *Pattern Recognition and Machine Learning* by Christopher Bishop (Springer, 2006) is a classic upper-level text covering the topics of this chapter in detail. If you plan to go further in this subject, you should have this book on your shelf.
- *Machine Learning: a Probabilistic Perspective* by Kevin Murphy (MIT Press, 2012) is an excellent graduate-level text that explores nearly all important ML algorithms from a ground-up, unified probabilistic perspective.