

ADDISON
WESLEY
DATA &
ANALYTICS
SERIES



Rough Cuts

Pandas for Everyone

Python Data Analysis

DANIEL Y. CHEN

Contents

Part I: Introduction

Chapter 1. Pandas DataFrame Basics

1.1 Introduction

1.2 Loading Your First Dataset

1.3 Looking at Columns, Rows, and Cells 1.4

Grouped and Aggregated Calculations 1.5

Basic Plot

1.6 Conclusion

Chapter 2. Pandas Data Structures

2.1 Introduction

2.2 Creating Your Own Data

2.3 The Series

2.4 The DataFrame

2.5 Making Changes to Series and DataFrames

2.6 Exporting and Importing Data

2.7 Conclusion

Chapter 3. Introduction to Plotting

3.1 Introduction

3.2 matplotlib

3.3 Statistical Graphics using matplotlib 3.4

seaborn

3.5 pandas

3.6 Seaborn Themes and Styles

3.7 Conclusion

Part II: Data Manipulation

Chapter 4. Data Assembly

4.1 Introduction

4.2 Tidy Data

4.3 Concatenation

4.4 Merging Multiple Datasets

4.5 Conclusion

Chapter 5. Missing Data

5.1 Introduction

5.2 What is a `NaN` value

5.3 Where do missing values come from?

5.4 Working with missing data

5.5 Conclusion

Chapter 6. Tidy Data

6.1 Introduction

6.2 Columns Contain Values, Not Variables

6.3 Columns Contain Multiple Variables

6.4 Variables in Both Rows and Columns

6.5 Multiple Observational Units in a Table (Normalization)

6.6 Observational Units Across Multiple Tables

6.7 Conclusion

Part III: Data Munging

Chapter 7. Data Types

7.1 Introduction

7.2 Data Types

7.3 Converting Types

7.4 Categorical Data

7.5 Conclusion

Chapter 8. Strings and Text Data

8.1 Introduction

8.2 Strings

8.3 String Methods

8.4 More String Methods

8.5 String Formatting

8.6 Regular Expressions (RegEx)

8.7 The `regex` Library

8.8 Conclusion

Chapter 9. Apply

9.1 Introduction

9.2 Functions

9.3 Apply (Basics)

9.4 Apply

9.5 Vectorized Functions

9.6 Lambda Functions

9.7 Conclusion

Chapter 10. Groupby operations: split-apply-combine

10.1 Introduction

10.2 Aggregate

10.3 Transform

10.4 Filter

10.5 The `pandas.core.groupby.DataFrameGroupBy` object

10.6 Working with a MultiIndex

10.7 Conclusion

Chapter 11. The `dateti` Data Type

11.1 Introduction

[11.2 Python's `datetime`](#)

[11.3 Converting to `datetime`](#)

[11.4 Loading data with dates](#)

[11.5 Extracting date components](#) [11.6](#)

[Date Calculations and `TimedeltaS`](#) [11.7](#)

[`datetime` methods](#)

[11.8 Getting stock data](#)

[11.9 Subsetting data based on dates](#)

[11.10 Date Ranges](#)

[11.11 Shifting Values](#)

[11.12 Resampleing](#)

[11.13 Timezones](#)

[11.14 Conclusion](#)

[Part IV: Data Modeling](#)

[Chapter 12. Linear Models](#)

[12.1 Introduction](#)

[12.2 Simple Linear Regression](#)

[12.3 Multiple Regression](#)

[12.4 Keeping index labels from `sklearn`](#)

[12.5 Conclusion](#)

[Chapter 13. Generalized Linear Models](#)

[13.1 Introduction](#)

[13.2 Logistic Regression](#)

[13.3 Poisson Regression](#)

[13.4 More Generalized Linear Models](#)

[13.5 Survival Analysis](#)

[13.6 Conclusion](#)

Chapter 14. Model Diagnostics

14.1 Introduction

14.2 Residuals

14.3 Comparing Multiple Models

14.4 K-fold Cross Validation

14.5 Conclusion

Chapter 15. Regularization

15.1 Introduction

15.2 Why Regularize?

15.3 LASSO

15.4 Ridge

15.5 Elastic Net

15.6 Cross Validation

15.7 Conclusion

Chapter 16. Clustering

16.1 Introduction

16.2 K-means

16.3 Hierarchical Clustering

16.4 Conclusion

Part V: Conclusion

Chapter 17. Life Outside of Pandas

17.1 The (Scientific) Computing Stack 17.2

Performance

17.3 Going Bigger and Faster

Chapter 18. Towards a Self-Directed Learner

18.1 It's Dangerous to Go Alone!

18.2 Local Meetups

[18.3 Conferences](#)

[18.4 The Internet](#)

[18.5 Podcasts](#)

[18.6 Conclusion](#)

[Part VI: Appendix](#)

[Appendix A. Installation](#)

[A.1 Installing Anaconda](#)

[A.2 Uninstall Anaconda](#)

[Appendix B. Command line](#)

[B.1 Installation](#)

[B.2 Basics](#)

[Appendix C. Project Templates](#)

[Appendix D. Using Python](#)

[D.1 Command line and text editor](#)

[D.2 Python and IPython](#)

[D.3 Jupyter](#)

[D.4 Integrated Development Environments \(IDEs\)](#)

[Appendix E. Working Directories](#)

[Appendix F. Environments](#)

[Appendix G. Install packages](#)

[G.1 Updating Packages](#)

[Appendix H. Importing Libraries](#)

[Appendix I. Lists](#)

[Appendix J. Tuples](#)

[Appendix K. Dictionaries](#)

[Appendix L. Slicing Values](#)

[Appendix M. Loops](#)

[Appendix N. Comprehensions](#)

[Appendix O. Functions](#)

[O.1 Default Parameters](#)

[O.2 Arbitrary Parameters](#)

[Appendix P. Ranges and Generators](#)

[Appendix Q. Multiple Assignment](#)

[Appendix R. numpy ndarray](#)

[Appendix S. Classes](#)

[Appendix T. Odo: The Shapeshifter](#)

Part I: Introduction

[Chapter 1](#), “[Pandas DataFrame Basics](#),” Loading and looking at some observations.

[Chapter 2](#), “[Pandas Data Structures](#),” Looking at what a Series and DataFrame can do.

[Chapter 3](#), “[Introduction to Plotting](#),” Plotting in matplotlib, seaborn, and pandas.

Chapter 1. Pandas DataFrame Basics

1.1 Introduction

Pandas is an open source Python library for data analysis. It gives Python the ability to work with spreadsheet-like data for fast data loading, manipulating, aligning, merging, etc. To give Python these enhanced features, Pandas introduces two new data types to Python: Series and DataFrame. The DataFrame will represent your entire spreadsheet or rectangular data, whereas the Series is a single column of the DataFrame. A Pandas DataFrame can also be thought of as a dictionary or collection of Series.

Why should you use a programming language like Python and a tool like Pandas to work with data? It boils down to automation and reproducibility. If there is a particular set of analysis that needs to be performed on multiple datasets, a programming language has the ability to automate the analysis on the datasets. Although many spreadsheet programs have their own macro programming language, many users do not use them. Furthermore, not all spreadsheet programs are available on all operating systems. Performing data tasks using a programming language forces the user to have a running record of all steps performed on the data. I, like many people, have accidentally hit a key while viewing data in a spreadsheet program, only to find out that my results do not make any sense anymore due to bad data. This is not to say spreadsheet programs are bad or do not have their place in the data workflow, they do, but there are better and more reliable tools out there.

Concept map

1. Prior knowledge needed (appendix)
 - (a) relative directories
 - (b) calling functions
 - (c) dot notation
 - (d) primitive python containers
 - (e) variable assignment
 - (f) the print statement in various Python environments
2. This chapter
 - (a) loading data
 - (b) subset data
 - (c) slicing
 - (d) filtering

- (e) basic pandas data structures (`Series`, `DataFrame`)
- (f) resemble other python containers (`list`, `numpy.ndarray`)
- (g) basic indexing

Objectives

This chapter will cover:

1. loading a simple delimited data file
2. count how many rows and columns were loaded
3. what is the type of data that was loaded
4. look at different parts of the data by subsetting rows and columns
5. saving a subset of data

1.2 Loading Your First Dataset

When given a data set, we first load it and begin looking at its structure and contents. The simplest way of looking at a data set is to look and subset specific rows and columns. We can see what type of information is stored in each column, and can start looking for patterns by aggregating descriptive statistics.

Since Pandas is not part of the Python standard library, we have to first tell Python to load (`import`) the library.

```
import pandas
```

With the library loaded we can use the `read_csv` function to load a CSV data file. In order to access the `read_csv` function from pandas, we use something called ‘dot notation’. More on dot notations can be found in [Appendix O](#), [S](#), and [H](#).

About the Gapminder dataset

The Gapminder dataset originally comes from: <https://www.gapminder.org/>. This particular version the book is using Gapminder data prepared by Jennifer Bryan from the University of British Columbia. The repository can be found at: [www.github.com/jennybc/gapminder](https://github.com/jennybc/gapminder).

```
# by default the read_csv function will read a comma separated file,
# our gapminder data set is separated by a tab
# we can use the sep parameter and indicate a tab with '\t'
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')
# we use the head function so Python only shows us the first 5 rows
print(df.head())
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710

```

3      Afghanistan      Asia      1967    34.020   11537966   836.197138
4      Afghanistan      Asia      1972    36.088   13079460   739.981106

```

Since we will be using Pandas functions many times throughout the book as well as your own programming. It is common to give `pandas` the alias `pd`. The above code will be the same as below:

```

import pandas as pd
df = pd.read_csv('../data/gapminder.tsv', sep='\t')

```

We can check to see if we are working with a Pandas Dataframe by using the built-in `type` function (i.e., it comes directly from Python, not any package such as Pandas).

```

print(type(df))
<class 'pandas.core.frame.DataFrame'>

```

The `type` function is handy when you begin working with many different types of Python objects and need to know what object you are currently working on.

The data set we loaded is currently saved as a Pandas DataFrame object and is relatively small. Every DataFrame object has a `shape` attribute that will give us the number of rows and columns of the DataFrame.

```

# get the number of rows and columns
print(df.shape)

(1704, 6)

```

The `shape` attribute returns a tuple ([Appendix J](#)) where the first value is the number of rows and the second number is the number of columns. From the results above, we see our gapminder data set has 1704 rows and 6 columns.

Since `shape` is an attribute of the dataframe, and not a function or method of the DataFrame, it does not have parenthesis after the period. If you made the mistake of putting parenthesis after the `shape` attribute, it would return an error.

```

# shape is an attribute, not a method
# this will cause an error
print(df.shape())

Traceback (most recent call last):
  File "<ipython-input-1-e05f133c2628>", line 2, in <module>
    print(df.shape())
TypeError: 'tuple' object is not callable

```

Typically, when first looking at a dataset, we want to know how many rows and columns there are (we just did that), and to get a gist of what information it contains, we look at the columns. The column names, like `shape`, is given using the `columns` attribute of the dataframe object.

```

# get column names
print(df.columns)

Index(['country', 'continent', 'year', 'lifeExp', 'pop',
       'gdpPercap'],
      dtype='object')

```

Question

What is the `type` of the column names?

(3)

The Pandas DataFrame object is similar to other languages that have a DataFrame-like object (e.g., Julia and R) Each column (Series) has to be the same type, whereas, each row can contain mixed types. In our current example, we can expect the `country` column to be all strings and the `year` to be integers. However, it's best to make sure that is the case by using the `dtypes` attribute or the `info` method. [Table 1-1](#) on page 7 shows what the type in Pandas is relative to native Python.

```
# get the dtype of each column
print(df.dtypes)

country      object
continent    object
year         int64
lifeExp     float64
pop          int64
gdpPercap   float64
dtype: object

# get more information about our data
print(df.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1704 entries, 0 to 1703
Data columns (total 6 columns):
country      1704 non-null object
continent    1704 non-null object
year         1704 non-null int64
lifeExp     1704 non-null float64
pop          1704 non-null int64
gdpPercap   1704 non-null float64
dtypes: float64(2), int64(2), object(2)
memory usage: 80.0+ KB
None
```

Table 1–1: Table of Pandas dtypes and Python types

Pandas Type	Python Type	Description
object	string	most common data type
int64	int	whole numbers
float64	float	numbers with decimals
datetime64	datetime	datetime is found in the Python standard library (i.e., it is not loaded by default and needs to be imported)

1.3 Looking at Columns, Rows, and Cells

Now that we're able to load up a simple data file, we want to be able to inspect its contents. We could `print` out the contents of the dataframe, but with todays data, there are too many cells to make sense of all the printed information. Instead, the best way to look at our data is to inspect it in parts by looking at various subsets of the data. We already saw above that we can use the `head` method of a datafram to look at the first 5 rows of our data. This is useful to see if our data loaded properly, get a sense of the columns, its name and its contents. However, there are going to be times when we only want particular rows, columns, or values from our data.

Before continuing, make sure you are familiar with Python containers ([Appendix I, K](#)).

1.3.1 Subsetting Columns

If we wanted multiple columns we can specify them a few ways: by names, positions, or ranges.

1.3.1.1 Subsetting Columns by Name

If we wanted only a specific column from our data we can access the data using square brackets.

```
# just get the country column and save it to its own variable
country_df = df['country']

# show the first 5 observations
print(country_df.head())

0    Afghanistan
1    Afghanistan
2    Afghanistan
3    Afghanistan
4    Afghanistan
Name: country, dtype: object

# show the last 5 observations
print(country_df.tail())

1699    Zimbabwe
1700    Zimbabwe
1701    Zimbabwe
1702    Zimbabwe
1703    Zimbabwe
Name: country, dtype: object
```

In order to specify multiple columns by the column name, we need to pass in a python `list` between the square brackets. This may look a bit strange since there will be 2 sets of square brackets.

```
# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]

print(subset.head())

   country  continent  year
0  Afghanistan      Asia  1952
1  Afghanistan      Asia  1957
2  Afghanistan      Asia  1962
3  Afghanistan      Asia  1967
4  Afghanistan      Asia  1972

print(subset.tail())

   country  continent  year
1699    Zimbabwe    Africa  1987
1700    Zimbabwe    Africa  1992
1701    Zimbabwe    Africa  1997
1702    Zimbabwe    Africa  2002
1703    Zimbabwe    Africa  2007
```

Again, you can opt to `print` the entire `subset` dataframe. I am not doing this for the book as it would take up an unnecessary amount of space.

1.3.1.2 Subsetting Columns by Index Position Break In pandas v0.20

At times, you may only want to get a particular column by its position, rather than its name. For example, you want to get the first ('country') column and third column ('year'), or just the last column ('gdpPercap').

As of `pandas v0.20`, you are no longer able to pass in a list of integers in the square brackets to subset columns. For example, `df [[1]]`, `df [[0, -1]]`, and `df [list (range(5))]` no longer work.

There's other ways of subsetting columns ([Section 1.3.3](#)), but those build on the technique used

to subset rows.

1.3.2 Subsetting Rows

Rows can be subset in multiple ways, by row name or row index. [Table 1-2](#) gives a quick overview of the various methods.

Table 1–2: Different methods of indexing rows (and or columns)

Subset method	Description
loc	subset based on index label (a.k.a. row name)
iloc	subset based on row index (a.k.a. row number)
ix (no longer works in pandas v0.20)	subset based on index label or row index

1.3.2.1 Subset Rows by index Label - loc

If we take a look at our gapminder data

```
print(df.head())
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

We can see on the left side of the printed dataframe, what appears to be row numbers. This column-less row of values is the index label of the dataframe. Think of it like column names, but instead for rows. By default, Pandas will fill in the index labels with the row numbers (note it starts counting from 0). A common example where the row index labels are not the row number is when we work with time series data. In that case, the index label will be a timestamps of sorts, but for now we will keep the default row number values.

We can use the `.loc` method on the dataframe to subset rows based on the index label.

```
# get the first row
# python counts from 0
print(df.loc[0])

country      Afghanistan
continent        Asia
year            1952
lifeExp         28.801
pop             8425333
gdpPercap       779.445
Name: 0, dtype: object

# get the 100th row
# python counts from 0
print(df.loc[99])

country      Bangladesh
continent        Asia
year            1967
lifeExp         43.453
pop             62821884
gdpPercap       721.186
Name: 99, dtype: object

# get the last row
# this will cause an error
```

```

print(df.loc[-1])

Traceback (most recent call last):
  File "/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/pandas/core/indexing.py", line 1434, in _has_valid_type
    error()
KeyError: 'the label [-1] is not in the [index]'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<ipython-input-1-5c89f7ac3971>", line 2, in <module>
    print(df.loc[-1])
KeyError: 'the label [-1] is not in the [index]'
```

Note that passing `-1` as the `loc` will cause an error, because it is actually looking for the row index label (row number) `-1`, which does not exist in our example. Instead we can use a bit of Python to calculate the number of rows and pass that value into `loc`.

```

# get the last row (correctly)
# use the first value given from shape to get the number of rows
number_of_rows = df.shape[0]

# subtract 1 from the value since we want the last index value
last_row_index = number_of_rows - 1

# finally do the subset using the index of the last row
print(df.loc[last_row_index])

country      Zimbabwe
continent     Africa
year          2007
lifeExp       43.487
pop           12311143
gdpPercap     469.709
Name: 1703, dtype: object
```

Or use the `tail` method to return the last `1` row, instead of the default `5`.

```

# there are many ways of doing what you want
print(df.tail(n=1))

   country  continent  year  lifeExp  pop  gdpPercap
1703    Zimbabwe     Africa  2007     43.487  12311143  469.709298
```

Notice that using `tail ()` and `loc` printed out the results differently. Let's look at what type is returned when we use these methods.

```

subset_loc = df.loc[0]
subset_head = df.head(n=1)

# type using loc of 1 row
print(type(subset_loc))

<class 'pandas.core.series.Series'>

# type of using head of 1 row
print(type(subset_head))

<class 'pandas.core.frame.DataFrame'>
```

The beginning of the chapter mentioned how Pandas introduces two new data types into Python. Depending on what method we use and how many rows we return, pandas will return a different object. The way an object gets printed to the screen can be an indicator of the type, but it's always best to use the `type` function to be sure. We go into more details about these objects in [Chapter 2](#).

Subsetting Multiple Rows Just like with columns we can select multiple rows.

```

# select the first, 100th, and 1000th row
# note the double square brackets similar to the syntax used to
# subset multiple columns
print(df.loc[[0, 99, 999]])

      country continent    year  lifeExp    pop  gdpPercap
0   Afghanistan     Asia  1952    28.801  8425333  779.445314
99  Bangladesh      Asia  1967    43.453  62821884  721.186086
999 Mongolia       Asia  1967    51.253  1149500  1226.041130

```

1.3.2.2 Subset Rows by Row Number - iloc

`iloc` does the same thing as `loc` but it is used to subset by the row index number. In our current example `iloc` and `loc` will behave exactly the same since the index labels are the row numbers. However, keep in mind that the index labels do not necessarily have to be row numbers.

```

# get the 2nd row
print(df.iloc[1])

country      Afghanistan
continent        Asia
year            1957
lifeExp         30.332
pop             9240934
gdpPercap       820.853
Name: 1, dtype: object

## get the 100th row
print(df.iloc[99])

country      Bangladesh
continent        Asia
year            1967
lifeExp         43.453
pop             62821884
gdpPercap       721.186
Name: 99, dtype: object

```

You can see when we put `1` into the list, we actually get the second row, and not the first. This follows Python's zero indexed behavior, meaning, the first item of a container is index 0 (i.e., 0th item of the container). More details about this kind of behavior can be found in ([Appendix I](#), [L](#), and [P](#))

With `iloc` we can now pass in the `-1` to get the last row. Something we couldn't do with `loc`

```

# using -1 to get the last row
print(df.iloc[-1])

country      Zimbabwe
continent        Africa
year            2007
lifeExp         43.487
pop             12311143
gdpPercap       469.709
Name: 1703, dtype: object

```

Just like before, we can pass in a list of integers to get multiple rows.

```

## get the first, 100th, and 1000th row
print(df.iloc[[0, 99, 999]])

      country continent    year  lifeExp    pop  gdpPercap
0   Afghanistan     Asia  1952    28.801  8425333  779.445314
99  Bangladesh      Asia  1967    43.453  62821884  721.186086
999 Mongolia       Asia  1967    51.253  1149500  1226.041130

```

1.3.2.3 Subsetting Rows With ix No Longer Work In pandas v0.20

.**ix** no longer work after **pandas v0.20**, since it can be confusing. This section will quickly go over **ix** for completeness.

.**ix** can be thought of as a combination of **loc** and **iloc**, it allows us to subset by label or integer.

By default it will search for labels, and if it cannot find the corresponding label, it will fall back to using integer indexing. This can be the cause for a lot of confusion, which is why it has been taken out. The code will look exactly like **loc** or **iloc**.

```
# first row  
df.ix[0]  
  
# 100th row  
df.ix[99]  
  
# 1st, 100th, and 1000th row  
df.ix[[0, 99, 999]]
```

1.3.3 Mixing It Up

The **loc** and **iloc** attributes can be used to take subsets of columns, rows, or both. The general syntax for **loc** and **iloc** uses a square bracketed with a comma. The part to the left of the comma will be the row values to subset, the part to the right of the comma will be the column values to subset.

That is, **df.loc [[row], [column]]** or **df.iloc [[row], [column]]**

1.3.3.1 Subsetting Columns

If we want to use these methods to just subset columns we have to use the python slicing syntax. We need to do this because if we are subsetting columns, we are getting all the rows for the specified column. So, we need a method to capture all the rows.

The python slicing syntax uses a colon, `[:]`. If we have just a colon, it refers to everything. So, if we just want to get the first column using the **loc** or **iloc** syntax, we can write something like **df.loc [:, [column]]** to subset column(s).

```
# subset columns with loc  
# note the position of the colon  
# it is used to select all rows  
subset = df.loc[:, ['year', 'pop']]  
print(subset.head())  
  
      year      pop  
0    1952  8425333  
1    1957  9240934  
2    1962 10267083  
3    1967 11537966  
4    1972 13079460  
  
# subset columns with iloc  
# iloc will allow us to use integers  
# -1 will select the last column  
subset = df.iloc[:, [2, 4, -1]]  
print(subset.head())  
      year      pop   gdpPerCap  
0    1952  8425333  779.445314  
1    1957  9240934  820.853030  
2    1962 10267083  853.100710  
3    1967 11537966  836.197138  
4    1972 13079460  739.981106
```

Note how we will get an error if we don't specify `loc` and `iloc` correctly

```
# subset columns with loc
# but pass in integer values
# this will cause an error
subset = df.loc[:, [2, 4, -1]]
print(subset.head())

Traceback (most recent call last):
  File "<ipython-input-1-719bcb04e3c1>", line 2, in <module>
    subset = df.loc[:, [2, 4, -1]]
KeyError: 'None of [[2, 4, -1]] are in the [columns]'

# subset columns with iloc
# but pass in index names
# this will cause an error
subset = df.iloc[:, ['year', 'pop']]
print(subset.head())

Traceback (most recent call last):
  File "<ipython-input-1-43f52fcab49<", line 2, in <module>
    subset = df.iloc[:, ['year', 'pop']]
TypeError: cannot perform reduce with flexible type
```

1.3.3.2 Subsetting Columns by Range

You can use the built-in `range` function to create a range of values in Python. This way you can specify a beginning and end value, and python will automatically create a range of values in between. By default, every value between the beginning and end (inclusive left, exclusive right - [Appendix L](#)) will be created, unless you specify a step ([Appendix L](#) and [P](#)). In Python 3 the `range` function returns a generator ([Appendix P](#)). If you are using Python 2, the `range` function returns a list ([Appendix I](#)), and the `xrange` function returns a generator.

If we look at the code above ([Section 1.3.1.2](#)), we see that we subset columns using a list of integers. since `range` returns a generator, we have to convert the generator to a list first.

```
# create a range of integers from 0 - 4 inclusive
small_range = list(range(5))
print(small_range)

[0, 1, 2, 3, 4]

# subset the dataframe with the range
subset = df.iloc[:, small_range]
print(subset.head())

   country continent  year  lifeExp      pop
0  Afghanistan     Asia  1952    28.801  8425333
1  Afghanistan     Asia  1957    30.332  9240934
2  Afghanistan     Asia  1962    31.997 10267083
3  Afghanistan     Asia  1967    34.020 11537966
4  Afghanistan     Asia  1972    36.088 13079460
```

Note that when `range(5)` is called, 5 integers are returned from 0 - 4.

```
# create a range from 3 - 5 inclusive
small_range = list(range(3, 6))
print(small_range)

[3, 4, 5]

subset = df.iloc[:, small_range]
print(subset.head())

   lifeExp      pop  gdpPercap
0    28.801  8425333  779.445314
1    30.332  9240934  820.853030
2    31.997 10267083  853.100710
```

```
3      34.020    11537966    836.197138
4      36.088    13079460    739.981106
```

Question

What happens when you specify a range that's beyond the number of columns you have?

Again, note that the values are specified in a way such that it is inclusive on the left, and exclusive on the right.

```
# create a range from 0 - 5 inclusive, every other integer
small_range = list(range(0, 6, 2))
subset = df.iloc[:, small_range]
print(subset.head())

      country      year      pop
0  Afghanistan  1952  8425333
1  Afghanistan  1957  9240934
2  Afghanistan  1962 10267083
3  Afghanistan  1967 11537966
4  Afghanistan  1972 13079460
```

Converting a generator to a list is a bit awkward, we can use the python slicing syntax to fix this.

1.3.3.3 Slicing Columns

Python's slicing syntax, `:`, is similar to the `range` syntax. Instead of a function that specifies, start, stop, step values delimited by a comma, we separate the values with the colon.

If you understand what was going on with the `range` function earlier, then slicing can be seen as a shorthand means to the same thing.

```
small_range = list(range(3))
subset = df.iloc[:, small_range]
print(subset.head())

      country continent      year
0  Afghanistan      Asia  1952
1  Afghanistan      Asia  1957
2  Afghanistan      Asia  1962
3  Afghanistan      Asia  1967
4  Afghanistan      Asia  1972

# slice the first 3 columns
subset = df.iloc[:, :3]
print(subset.head())

      country continent      year
0  Afghanistan      Asia  1952
1  Afghanistan      Asia  1957
2  Afghanistan      Asia  1962
3  Afghanistan      Asia  1967
4  Afghanistan      Asia  1972

small_range = list(range(3, 6))
subset = df.iloc[:, small_range]
print(subset.head())

  lifeExp      pop      gdpPercap
0   28.801    8425333    779.445314
1   30.332    9240934    820.853030
2   31.997   10267083    853.100710
3   34.020   11537966    836.197138
4   36.088   13079460    739.981106
```

```

# slice columns 3 to 5 inclusive
subset = df.iloc[:, 3:6]
print(subset.head())

    lifeExp      pop   gdpPercap
0    28.801    8425333    779.445314
1    30.332    9240934    820.853030
2    31.997   10267083    853.100710
3    34.020   11537966    836.197138
4    36.088   13079460    739.981106

small_range = list(range(0, 6, 2))
subset = df.iloc[:, small_range]
print(subset.head())

      country      year      pop
0  Afghanistan  1952  8425333
1  Afghanistan  1957  9240934
2  Afghanistan  1962 10267083
3  Afghanistan  1967 11537966
4  Afghanistan  1972 13079460

# slice every other first 5 columns
subset = df.iloc[:, 0:6:2]
print(subset.head())

```

Question

What happens if you use the slicing method with 2 colons, but leave a value out. For example:

- df . iloc [:, 0:6:]
 - df . iloc [:, 0::2]
 - df . iloc [:, :6:2]
 - df . iloc [:, ::2]
 - df . iloc [:, ::]
-

1.3.3.4 Subsetting Rows and Columns

We've been using the colon, :, in loc and iloc to the left of the comma. Meaning, we are selecting all the rows in our dataframe. However, we can choose to put values to the left of the comma if we want to select specific rows along with specific columns.

```

# using loc
print(df.loc[42, 'country'])

Angola

# using iloc
print(df.iloc[42, 0])

Angola

```

Just make sure you don't confuse the differences between `loc` and `iloc`.

```
# will cause an error
print(df.loc[42, 0])

Traceback (most recent call last):
  File "<ipython-input-1-2b69d7150b5e>", line 2, in <module>
    print(df.loc[42, 0])
TypeError: cannot do label indexing on <class
'pandas.core.indexes.base.Index'> with these indexers [0] of <class
'int'>
```

Now, look at how confusing `ix` can be. Good thing it no longer works.

```
# get the 43rd country in our data
df.ix[42, 'country']

# instead of 'country' I used the index 0
df.ix[42, 0]
```

1.3.3.5 Subsetting Multiple Rows and Columns

We can combine the row and column subsetting syntax with the multiple row and column subsetting syntax to get various slices of our data.

```
# get the 1st, 100th, and 1000th rows
# from the 1st, 4th, and 6th column
# note the columns we are hoping to get are:
# country, lifeExp, and gdpPercap
print(df.iloc[[0, 99, 999], [0, 3, 5]])

      country  lifeExp    gdpPercap
0   Afghanistan    28.801  779.445314
99  Bangladesh     43.453  721.186086
999 Mongolia       51.253  1226.041130
```

I personally try to pass in the actual column names when subsetting data if possible. It makes the code more readable since you do not need to look at the column name vector to know which index is being called. Additionally, using absolute indexes can lead to problems if the column order gets changed for whatever reason.

```
# if we use the column names directly,
# it makes the code a bit easier to read
# note now we have to use loc, instead of iloc
print(df.loc[[0, 99, 999], ['country', 'lifeExp', 'gdpPercap']])

      country  lifeExp    gdpPercap
0   Afghanistan    28.801  779.445314
99  Bangladesh     43.453  721.186086
999 Mongolia       51.253  1226.041130
```

Remember, you can use the slicing syntax on the row portion of the `loc` and `iloc` attributes.

```
print(df.loc[10:13, ['country', 'lifeExp', 'gdpPercap']])

      country  lifeExp    gdpPercap
10  Afghanistan    42.129  726.734055
11  Afghanistan    43.828  974.580338
12    Albania      55.230  1601.056136
13    Albania      59.280  1942.284244
```

1.4 Grouped and Aggregated Calculations

If you've worked with other numeric libraries or languages, many basic statistic calculations

either come with the library, or are built into the language. Looking at our gapminder data again

```
print(df.head(n=10))

   country continent    year  lifeExp      pop  gdpPercap
0  Afghanistan      Asia  1952  28.801  8425333  779.445314
1  Afghanistan      Asia  1957  30.332  9240934  820.853030
2  Afghanistan      Asia  1962  31.997 10267083  853.100710
3  Afghanistan      Asia  1967  34.020 11537966  836.197138
4  Afghanistan      Asia  1972  36.088 13079460  739.981106
5  Afghanistan      Asia  1977  38.438 14880372  786.113360
6  Afghanistan      Asia  1982  39.854 12881816  978.011439
7  Afghanistan      Asia  1987  40.822 13867957  852.395945
8  Afghanistan      Asia  1992  41.674 16317921  649.341395
9  Afghanistan      Asia  1997  41.763 22227415  635.341351
```

There are several initial questions that we can ask ourselves:

1. For each year in our data, what was the average life expectancy? what about population and GDP?
2. What if we stratify by continent?
3. How many countries are listed in each continent?

1.4.1 Grouped Means

In order to answer the questions posed above, we need to perform a grouped (aka aggregate) calculation. That is, we need to perform a calculation, be it an average, or frequency count, but apply it to each subset of a variable. Another way to think about grouped calculations is split-apply-combine. We first split our data into various parts, apply a function (or calculation) of our choosing to each of the split parts, and finally combine all the individual split calculation into a single dataframe. We accomplish grouped/aggregate computations by using the `groupby` method on dataframes.

```
# For each year in our data, what was the average life expectancy?
# To answer this question,
# we need to split our data into parts fby year
# then we get the 'lifeExp' column and calculate the mean
print(df.groupby('year')['lifeExp'].mean())

year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Let's unpack the statement above. We first create a grouped object. Notice that if we printed the grouped dataframe, pandas only returns us the memory location

```
grouped_year_df = df.groupby('year')
print(type(grouped_year_df))

<class 'pandas.core.groupby.DataFrameGroupBy'>
```

```

print(grouped_year_df)
<pandas.core.groupby.DataFrameGroupBy object at 0x7fe424583438>

```

From the grouped data, we can subset the columns of interest we want to perform calculations on. In our case our question needs the `lifeExp` column. We can use the subsetting methods described in [Section 1.3.1.1](#).

```

grouped_year_df_lifeExp = grouped_year_df['lifeExp']
print(type(grouped_year_df_lifeExp))

<class 'pandas.core.groupby.SeriesGroupBy'>

print(grouped_year_df_lifeExp)

<pandas.core.groupby.SeriesGroupBy object at 0x7fe423c9f208>

```

Notice we now are given a series (because we only asked for 1 column) where the contents of the series are grouped (in our example by year).

Finally, we know the `lifeExp` column is of type `float64`. An operation we can perform on a vector of numbers is to calculate the mean to get our final desired result.

```

mean_lifeExp_by_year = grouped_year_df_lifeExp.mean()
print(mean_lifeExp_by_year)

year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64

```

We can perform a similar set of calculations for population and GDP since they are of types `int64` and `float64`, respectively. However, what if we want to group and stratify by more than one variable? and perform the same calculation on multiple columns? We can build on the material earlier in this chapter by using a list!

```

# the backslash allows us to break up 1 long line of python code
# into multiple lines
# df.groupby(['year', 'continent'])[['lifeExp', 'gdpPercap']].mean()
# is the same as
multi_group_var = df.\
    groupby(['year', 'continent'])\
    [['lifeExp', 'gdpPercap']].\
    mean()
print(multi_group_var)

      lifeExp      gdpPercap
year continent
1952 Africa        39.135500     1252.572466
          Americas     53.279840     4079.062552
          Asia         46.314394     5195.484004
          Europe        64.408500     5661.057435
          Oceania       69.255000     10298.085650
1957 Africa        41.266346     1385.236062
          Americas     55.960280     4616.043733
          Asia         49.318544     5787.732940
          Europe        66.703067     6963.012816
          Oceania       70.295000    11598.522455
1962 Africa        43.319442     1598.078825

```

	Americas	58.398760	4901.541870
	Asia	51.563223	5729.369625
	Europe	68.539233	8365.486814
	Oceania	71.085000	12696.452430
1967	Africa	45.334538	2050.363801
	Americas	60.410920	5668.253496
	Asia	54.663640	5971.173374
	Europe	69.737600	10143.823757
	Oceania	71.310000	14495.021790
1972	Africa	47.450942	2339.615674
	Americas	62.394920	6491.334139
	Asia	57.319269	8187.468699
	Europe	70.775033	12479.575246
	Oceania	71.910000	16417.333380
1977	Africa	49.580423	2585.938508
	Americas	64.391560	7352.007126
	Asia	59.610556	7791.314020
	Europe	71.937767	14283.979110
	Oceania	72.855000	17283.957605
1982	Africa	51.592865	2481.592960
	Americas	66.228840	7506.737088
	Asia	62.617939	7434.135157
	Europe	72.806400	15617.896551
	Oceania	74.290000	18554.709840
1987	Africa	53.344788	2282.668991
	Americas	68.090720	7793.400261
	Asia	64.851182	7608.226508
	Europe	73.642167	17214.310727
	Oceania	75.320000	20448.040160
1992	Africa	53.629577	2281.810333
	Americas	69.568360	8044.934406
	Asia	66.537212	8639.690248
	Europe	74.440100	17061.568084
	Oceania	76.945000	20894.045885
1997	Africa	53.598269	2378.759555
	Americas	71.150480	8889.300863
	Asia	68.020515	9834.093295
	Europe	75.505167	19076.781802
	Oceania	78.190000	24024.175170
2002	Africa	53.325231	2599.385159
	Americas	72.422040	9287.677107
	Asia	69.233879	10174.090397
	Europe	76.700600	21711.732422
	Oceania	79.740000	26938.778040
2007	Africa	54.806038	3089.032605
	Americas	73.608120	11003.031625
	Asia	70.728485	12473.026870
	Europe	77.648600	25054.481636
	Oceania	80.719500	29810.188275

The output data is grouped by year and continent. For each year-continent set, we calculated the average life expectancy and GDP. The data is also printed out a little differently. Notice the year and continent ‘column names’ are not on the same line as the life expectancy and GPD ‘column names’. There is some hierachal structure between the year and continent row indices. More about working with these types of data in [Chapter 10](#). But if you need to “flatten” the dataframe, you can use the `resetIndex` method.

```
flat = multi_group_var.reset_index()
print(flat.head(15))

      year continent    lifeExp      gdpPercap
0   1952    Africa  39.135500  1252.572466
1   1952  Americas  53.279840  4079.062552
2   1952    Asia  46.314394  5195.484004
3   1952   Europe  64.408500  5661.057435
4   1952  Oceania  69.255000  10298.085650
5   1957    Africa  41.266346  1385.236062
6   1957  Americas  55.960280  4616.043733
7   1957    Asia  49.318544  5787.732940
8   1957   Europe  66.703067  6963.012816
9   1957  Oceania  70.295000  11598.522455
10  1962    Africa  43.319442  1598.078825
11  1962  Americas  58.398760  4901.541870
```

```
12    1962      Asia    51.563223    5729.369625
13    1962     Europe   68.539233    8365.486814
14    1962   Oceania   71.085000   12696.452430
```

Question

Does the order of the list we used to group matter?

1.4.2 Grouped Frequency Counts

Another common data task is to calculate frequencies. We can use the `nunique` or `value_counts` methods to get a count of unique values, or frequency counts, on a Pandas Series, respectively.

```
# use the nunique (number unique)
# to calculate the number of unique values in a series
print(df.groupby('continent')['country'].nunique())

continent
Africa      52
Americas    25
Asia        33
Europe      30
Oceania     2
Name: country, dtype: int64
```

Question

What do you get if you use ‘value.counts’ instead of ‘nunique’?

1.5 Basic Plot

Visualizations are extremely important in almost every step of the data process. They help identify trends in data when we are trying to understand and clean it, and they help convey our final findings.

Let’s look at the yearly life expectancies of the world again.

```
global_yearly_life_expectancy = df.groupby('year')['lifeExp'].mean()
print(global_yearly_life_expectancy)

year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

We can use pandas to do some basic plots as shown in [Figure 1-1](#).

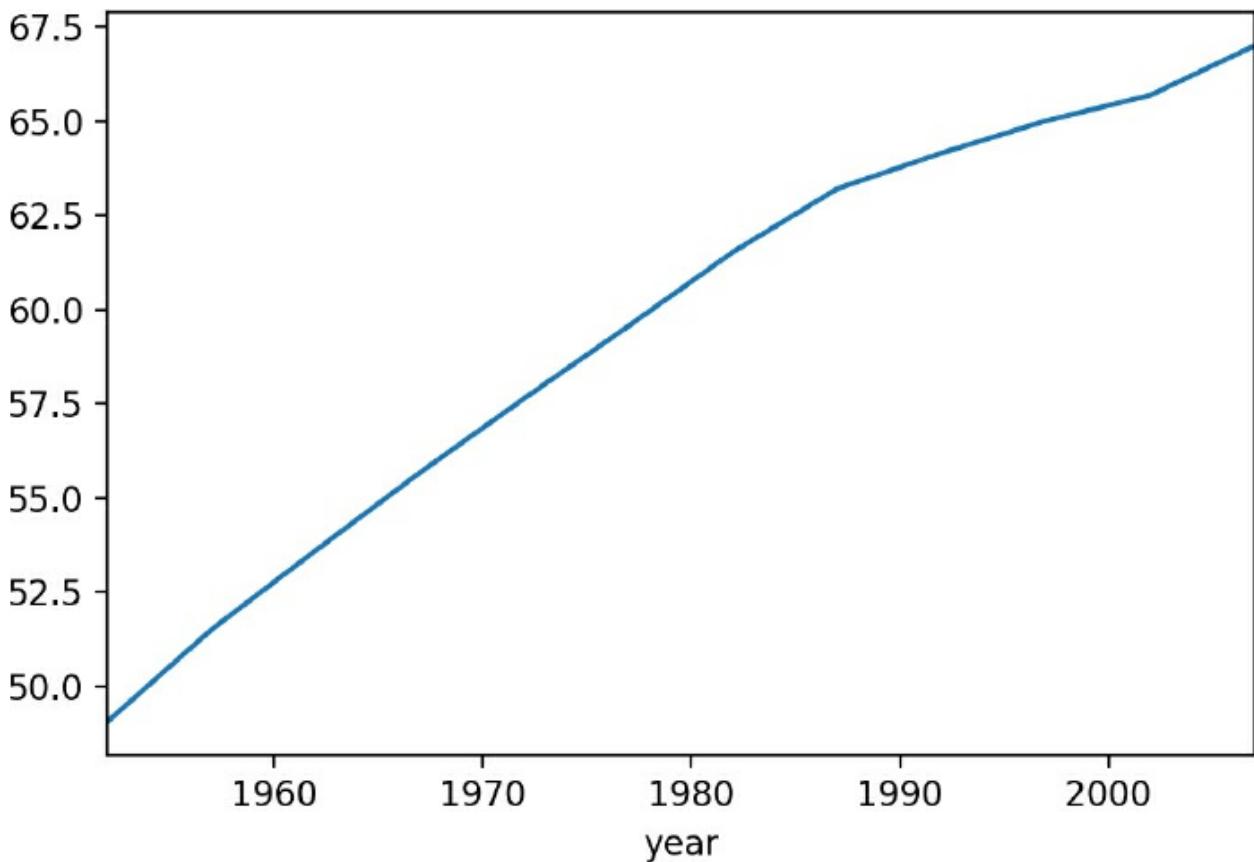
```
global_yearly_life_expectancy.plot()
```

1.6 Conclusion

This chapter showed how to load up a simple dataset and start looking at specific observations. It may seem tedious at first to look at observations this way, especially if you have been coming from a spreadsheet program. Keep in mind, when doing data analytics, the goal is to be reproducible, and not repeat repetitive tasks. Scripting languages give you that ability and flexibility.

Along the way you learned some of the fundamental programming abilities and data structures Python has to offer. As well as a quick way to go aggregated statistics and plots.

Figure 1-1: Basic plots in pandas showing average life expectancy over time



The next chapter will go into more detail about the Pandas `DataFrame` and `Series` object, as well as other ways you can subset and visualize your data.

As you work your way though the book, if there is a concept or data structure that is foreign to you, check the various Appendix chapters for the relevant topic. There are many fundamental programming features of Python there.

Chapter 2. Pandas Data Structures

2.1 Introduction

[Chapter 1](#), mentions the Pandas `DataFrame` and `Series` data structures. These data structures will resemble the primitive Python data containers (lists and dictionaries) for indexing and labeling, but have additional features to make working with data easier.

Concept map

1. Prior knowledge
 - (a) Containers
 - (b) Using functions
 - (c) Subsetting and indexing
2. load in manual data
3. Series
 - (a) creating a series
 - i. dict
 - ii. ndarray
 - iii. scalar
 - iv. lists
 - (b) slicing

Objectives

This chapter will cover:

1. load in manual data
2. learn about the Series object
3. basic operations on Series objects
4. learn about the DataFrame object
5. conditional subsetting and fancy slicing and indexing
6. save out data

2.2 Creating Your Own Data

Whether you are manually inputting data, or creating a small test example, knowing how to create dataframes without loading data from a file is a useful skill. Especially when asking a question on StackOverflow.

2.2.1 Creating A series

The Pandas Series is a one-dimensional container, similar to the built in python `list`. It is the datatype that represents each column of the `DataFrame`. [Table 1-1](#) lists the possible `dtypes` for Pandas `DataFrame` columns. Each column in a dataframe must be of the same `dtype`. Since a dataframe can be thought of a dictionary of `Series` objects, where each `key` is the column name, and the `value` is the `Series`, we can conclude that a series is very similar to a python `list`, except each element must be the same `dtype`. Those who have used the `numpy` library will realize this is the same behavior as the `ndarray`.

The easiest way to create a `series` is to pass in a Python `list`. If we pass in a list of mixed types, the most common representation of both will be used. Typically the `dtype` will be `object`.

```
import pandas as pd
s = pd.Series(['banana', 42])
print(s)
0    banana
1        42
dtype: object
```

You'll notice on the left the 'row number' is shown. This is actually the `index` for the series. It is similar to the row name and row index we saw in [Section 1.3.2](#) for dataframes. This implies that we can actually assign a 'name' to values in our series.

```
# manually assign index values to a series
# by passing a Python list
s = pd.Series(['Wes McKinney', 'Creator of Pandas'],
              index=['Person', 'Who'])
print(s)
Person      Wes McKinney
Who        Creator of Pandas
dtype: object
```

Questions

1. What happens if you use other Python containers like `list`, `tuple`, `dict`, or even the `ndarray` from the `numpy` library?
 2. What happens if you pass an `index` along with the containers?
 3. Does passing in an `index` when you use a `dict` overwrite the index? Or does it sort the values?
-

2.2.2 Creating a Data Frame

As mentioned in [Section 1.1](#), a `DataFrame` can be thought of as a dictionary of `Series` objects. This

is why dictionaries are the the most common way of creating a `DataFrame`. The `key` will represent the column name, and the `values` will be the contents of the column.

```
scientists = pd.DataFrame({
    'Name': ['Rosaline Franklin', 'William Gosset'],
    'Occupation': ['Chemist', 'Statistician'],
    'Born': ['1920-07-25', '1876-06-13'],
    'Died': ['1958-04-16', '1937-10-16'],
    'Age': [37, 61])
print(scientists)
```

	Age	Born	Died	Name	Occupation
0	37	1920-07-25	1958-04-16	Rosaline Franklin	Chemist
1	61	1876-06-13	1937-10-16	William Gosset	Statistician

Notice that order is not guaranteed.

If we look at the documentation for `DataFrame`,¹ we can use the `columns` parameter or specify the column order. If we wanted to use the `name` column for the row index, we can use the `index` parameter.

¹DataFrame documentation: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html>

```
scientists = pd.DataFrame(
    data={'Occupation': ['Chemist', 'Statistician'],
          'Born': ['1920-07-25', '1876-06-13'],
          'Died': ['1958-04-16', '1937-10-16'],
          'Age': [37, 61]},
    index=['Rosaline Franklin', 'William Gosset'],
    columns=['Occupation', 'Born', 'Died', 'Age'])
print(scientists)
```

	Occupation	Born	Died	Age
Rosaline Franklin	Chemist	1920-07-25	1958-04-16	37
William Gosset	Statistician	1876-06-13	1937-10-16	61

The order is not guaranteed because python dictionaries are not ordered, If we want an ordered dictionary, we need to use the `OrderedDict` from the `collections` module. But it's not as simple as wrapping the `OrderedDict` function around our dictionary, since the dictionary would've already lost its order by the time it is created and passed into our `OrderedDict` function.

```
from collections import OrderedDict

# note the round brackets after OrderedDict
# then we pass a list of 2-tuples
scientists = pd.DataFrame(OrderedDict([
    ('Name', ['Rosaline Franklin', 'William Gosset']),
    ('Occupation', ['Chemist', 'Statistician']),
    ('Born', ['1920-07-25', '1876-06-13']),
    ('Died', ['1958-04-16', '1937-10-16']),
    ('Age', [37, 61])
]))
print(scientists)

      Name      Occupation      Born      Died      Age
0 Rosaline Franklin      Chemist 1920-07-25 1958-04-16      37
1 William Gosset      Statistician 1876-06-13 1937-10-16      61
```

2.3 The Series

In [Section 1.3.2.1](#), we saw how the slicing method effects the `type` of the result. If we use the `loc` method to subset the first row of our `scientists` dataframe, we will get a `series` object back.

First let's re-create our example dataframe

```
# create our example dataframe
# with a row index label
scientists = pd.DataFrame(
    data={'Occupation': ['Chemist', 'Statistician'],
          'Born': ['1920-07-25', '1876-06-13'],
          'Died': ['1958-04-16', '1937-10-16'],
          'Age': [37, 61]},
    index=['Rosaline Franklin', 'William Gosset'],
    columns=['Occupation', 'Born', 'Died', 'Age'])
print(scientists)
```

	Occupation	Born	Died	Age
Rosaline Franklin	Chemist	1920-07-25	1958-04-16	37
William Gosset	Statistician	1876-06-13	1937-10-16	61

Select a scientist by the row index label.

```
# select by row index label
first_row = scientists.loc['William Gosset']
print(type(first_row))

<class 'pandas.core.series.Series'>

print(first_row)

Occupation      Statistician
Born            1876-06-13
Died            1937-10-16
Age              61
Name: William Gosset, dtype: object
```

When a series is printed (i.e., the string representation), the index is printed down as the first ‘column’, and the values are printed as the second ‘column’. There are many attributes and methods associated with a series object.² Two examples of attributes are `index` and `values`.

²<http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html>

```
print(first_row.index)
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
print(first_row.values)
['Statistician' '1876-06-13' '1937-10-16' 61]
```

An example of a series method is `keys`, which is an alias for the `index` attribute.

```
print(first_row.keys())
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

By now, you may have questions about the syntax between `index`, `values`, and `keys`. More about attributes and methods are described in [Appendix S](#) on classes. Attributes can be thought of as properties of an object (in this example our object is a `series`). Methods can be thought of as some calculation or operation that is performed. The subsetting syntax for `loc`, `iloc`, and `ix` (from [Section 1.3.2](#)) are all attributes. This is why the syntax does not have a set of round parenthesis, (), but rather, a set of square brackets, [], for subsetting. Since `keys` is a method, if we wanted to get the first key (which is also the first index) we would use the square brackets *after* the method call. Some attributes for the series are listed in Table ??.

```
# get the first index using an attribute
print(first_row.index[0])
```

```

Occupation

# get the first index using a method
print(first_row.keys()[0])

Occupation

```

Table 2–1: Some of the attributes within a series

Series attributes	Description
loc	Subset using index value
iloc	Subset using index position
ix	Subset using index value and/or position
dtype or dtypes	The type of the Series contents
T	Transpose of the series
shape	Dimensions of the data
size	Number of elements in the Series
values	ndarray or ndarray-like of the Series

2.3.1 The series Is ndarray-like

The `Pandas. Series` is very similar to the `numpy.ndarray` ([Appendix R](#)). This means, that many methods and functions that operate on a `ndarray` will also operate on a `series`. People will also refer to a `series` as a ‘vector’.

2.3.1.1 series Methods

Let’s first get a series of ‘Age’ column from our `scientists` dataframe.

```

# get the 'Age' column
ages = scientists['Age']
print(ages)

Rosaline Franklin    37
William Gosset       61
Name: Age, dtype: int64

```

`Numpy` is a scientific computing library that typically deals with numeric vectors. Since a `series` can be thought of as an extension to the `numpy.ndarray`, there is an overlap of attributes and methods. When we have a vector of numbers, there are common calculations we can perform.³

³Descriptive Statistics: <http://pandas.pydata.org/pandas-docs/stable/basics.html#descriptive-statistics>

```

print(ages.mean())
49.0
print(ages.min())
37
print(ages.max())
61

```

```

print(ages.std())
16.9705627485

```

The mean, min, max, and std are also methods in the numpy.ndarray.⁴ Some series methods are listed in [Table 2-2](#).

⁴numpy ndarray documentation: <http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html>

Table 2–2: Some of the methods that can be performed on a Series. * indicates missing values will be automatically dropped

Series methods	Description
append	Concatenates 2 or more Series
corr	Calculate a correlation with another Series*
cov	Calculate a covariance with another Series*
describe	Calculate summary statistics*
drop_duplicates	Returns a Series without duplicates
equals	Sees if a Series has the same elements
get_values	Get values of the Series, same as the values attribute
hist	Draw a histogram
isin	Checks to see if values are contained in a series
min	Return the minimum value
max	Returns the maximum value
mean	Returns the arithmetic mean
median	Returns the median
mode	Returns the mode(s)
quantile	Returns the value at a given quantile
replace	Replaces values in the Series with a specified value
sample	Returns a random sample of values from the Series
sort_values	Sort values
to_frame	Converts Series to DataFrame
transpose	Return the transpose
unique	Returns a numpy.ndarray of unique values

2.3.2 Boolean Subsetting Series

[Chapter 1](#) showed how we can use specific indices to subset our data. However, it is rare that we know the exact row or column index to subset the data. Typically you are looking for values that meet (or don't meet) a particular calculation or observation.

First, let's use a larger dataset

```
scientists = pd.read_csv('.../data/scientists.csv')
```

We just saw how we can calculate basic descriptive metrics of vectors

```

ages = scientists['Age']
print(ages)

0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64

# get basic stats
print(ages.describe())

  count      8.000000
  mean      59.125000
  std       18.325918
  min      37.000000
  25%     44.000000
  50%     58.500000
  75%     68.750000
  max     90.000000
Name: Age, dtype: float64

# mean of all ages
print(ages.mean())

59.125

```

What if we wanted to subset our ages by those above the mean?

```

print(ages[ages > ages.mean()])
1    61
2    90
3    66
7    77
Name: Age, dtype: int64

```

If we tease out this statement and look at what `ages > ages.mean()` returns

```

print(ages > ages.mean())

0    False
1    True
2    True
3    True
4    False
5    False
6    False
7    True
Name: Age, dtype: bool

print(type(ages > ages.mean()))
<class 'pandas.core.series.Series'>

```

The statement returns a `Series` with a `dtype` of `bool`.

This means we can not only subset values using labels and indices, we can also supply a vector of boolean values. Python has many functions and methods. Depending on how it is implemented, it may return labels, indices, or booleans. Keep this in mind as you learn new methods and have to piece together various parts for your work.

If we wanted to, we could manually supply a vector of `bools` to subset our data.

```

# get index 0, 1, 4, and 5
manual_bool_values = [True, True, False, False, True, True, False, True]
print(ages[manual_bool_values])

```

```
0    37
1    61
4    56
5    45
7    77
Name: Age, dtype: int64
```

2.3.3 Operations Are Automatically Aligned and Vectorized (Broadcasting)

If you're familiar with programming, you would find it strange `ages > ages.mean()` returns a vector without any `for` loops ([Appendix M](#)). Many of the methods that work on series (and also dataframes) are vectorized, meaning, they work on the entire vector simultaneously. It makes the code easier to read, and typically there are optimizations to make calculations faster.

2.3.3.1 Vectors Of Same Length

If you perform an operation between 2 vectors of the same length, the resulting vector will be an element-by-element calculation of the vectors.

```
print(ages + ages)

0    74
1   122
2   180
3   132
4   112
5    90
6    82
7   154
Name: Age, dtype: int64

print(ages * ages)
0   1369
1   3721
2   8100
3   4356
4   3136
5   2025
6   1681
7   5929
Name: Age, dtype: int64
```

2.3.3.2 Vectors With Integers (Scalars)

When you perform an operation on a vector using a scalar, the scalar will be recycled across all the elements in the vector.

```
print(ages + 100)

0    137
1    161
2    190
3    166
4    156
5    145
6    141
7    177
Name: Age, dtype: int64

print(ages * 2)

0    74
1   122
2   180
3   132
```

```
4    112
5     90
6     82
7    154
Name: Age, dtype: int64
```

2.3.3.3 Vectors With Different Lengths

When you are working with vectors of different lengths, the behavior will depend on the `type` of the vectors.

With a `Series`, the vectors will perform an operation matched by the index. The rest of the resulting vector will be filled with a ‘missing’ value, this is denoted with a `NaN`, for ‘not a number’.

This type of behavior is called ‘broadcasting’ and it differs between languages. Broadcasting in Pandas refers to how operations are calculated between arrays with different shapes.

```
print(ages + pd.Series([1, 100]))  
  
0    38.0
1   161.0
2     NaN
3     NaN
4     NaN
5     NaN
6     NaN
7     NaN
dtype: float64
```

With other `types`, the shapes must match.

```
import numpy as np  
  
# this will cause an error
print(ages + np.array([1, 100]))  
  
Traceback (most recent call last):
  File "<ipython-input-1-daaf3fc48315>", line 2, in <module>
    print(ages + np.array([1, 100]))
ValueError: operands could not be broadcast together with shapes (8, )
(2, )
```

2.3.3.4 Vectors With Common Index Labels (Automatic Alignment)

What’s cool about Pandas is how data alignment is almost always automatic. If possible, things will always align themselves with the index label when actions are performed.

```
# ages as they appear in the data
print(ages)  
  
0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64  
  
rev_ages = ages.sort_index(ascending=False)
print(rev_ages)  
  
7      77
6      41
```

```

5      45
4      56
3      66
2      90
1      61
0      37
Name: Age, dtype: int64

```

If we perform an operation using the `ages` and `rev_ages`, it will still be conducted element-by-element, however, the vectors will be aligned first before the operation is carried out.

```

# reference output to show index label alignment
print(ages * 2)

0    74
1   122
2   180
3   132
4   112
5    90
6    82
7   154
Name: Age, dtype: int64

# note how we get the same values
# even though the vector is reversed
print(ages + rev_ages)

0    74
1   122
2   180
3   132
4   112
5    90
6    82
7   154
Name: Age, dtype: int64

```

2.4 The DataFrame

The `DataFrame` is the most common `Pandas` object. It can be thought of as Python's way of storing spreadsheet-like data.

Many of the common features with the `Series` carry over into the `DataFrame`.

2.4.1 Boolean Subsetting Data Frame[[h4]]s

Just like how we were able to subset a `Series` with a boolean vector, we can subset a `DataFrame` with a `bool`.

```

# Boolean vectors will subset rows
print(scientists[scientists['Age'] > scientists['Age'].mean()])
      Name          Born        Died   Age Occupation
1  William Gosset  1876-06-13  1937-10-16  61  Statistician
2  Florence Nightingale  1820-05-12  1910-08-13  90       Nurse
3    Marie Curie  1867-11-07  1934-07-04  66     Chemist
7    Johann Gauss  1777-04-30  1855-02-23  77 Mathematician

```

Because of how broadcasting works, if we supply a bool vector that is not the same as the number of rows in the dataframe, the maximum possible rows returned would be the length of the bool vector.

```

# 4 values passed as a bool vector
# 3 rows returned
print(scientists.loc[[True, True, False, True]])

```

	Name	Born	Died	Age	Occupation
0	Rosaline Franklin	1920-07-25	1958-04-16	37	Chemist
1	William Gosset	1876-06-13	1937-10-16	61	Statistician
3	Marie Curie	1867-11-07	1934-07-04	66	Chemist

[Table 2-3](#) summarizes all the various subsetting methods:

Table 2–3: Table of DataFrame subsetting methods. Note that ix no longer works after pandas v0.20

Syntax	Selection Result
<code>df[column_name]</code>	Single column
<code>df[[column1, column2, ...]]</code>	Multiple columns
<code>df.loc [row_label]</code>	Row by row index label (row name)
<code>df.loc [[label1 , label2 , ...]]</code>	Multiple rows by index label
<code>df.iloc [row_number]</code>	Row by row number
<code>df.iloc [[row1, row2, ...]]</code>	Multiple rows by row number
<code>df.ix [label_or_number]</code>	Row by index label or number
<code>df.ix [[lab_num1, lab_num2, ...]]</code>	Multiple rows by index label or number
<code>df[bool]</code>	Row based on bool
<code>df[[bool1, bool2, ...]]</code>	Multiple rows based on bool
<code>df[start :stop :step]</code>	Rows based on slicing notation

2.4.2 Operations Are Automatically Aligned and Vectorized (Broadcasting)

Pandas supports *broadcasting*, which comes from the `numpy` library.⁵ In essence, it describes what happens when performing operations between array-like objects, which the pandas `Series` and `DataFrame` are. These behaviors are dependent on the type of object, its length, and any labels associated with the object.

⁵Numpy library: <http://www.numpy.org/>

First let's create a subset of our dataframes.

```
first_half = scientists[:4]
second_half = scientists[4:]

print(first_half)

      Name      Born      Died   Age Occupation
0  Rosaline  Franklin  1920-07-25  1958-04-16  37  Chemist
1    William  Gosset  1876-06-13  1937-10-16  61  Statistician
2  Florence  Nightingale  1820-05-12  1910-08-13  90  Nurse
3     Marie  Curie  1867-11-07  1934-07-04  66  Chemist

print(second_half)

      Name      Born      Died   Age Occupation
4  Rachel  Carson  1907-05-27  1964-04-14  56  Biologist
5    John  Snow  1813-03-15  1858-06-16  45  Physician
6  Alan  Turing  1912-06-23  1954-06-07  41  Computer Scientist
7  Johann  Gauss  1777-04-30  1855-02-23  77  Mathematician
```

When we perform an action on a dataframe with a scalar, it will try to apply the operation on

each cell of the dataframe. In this example, numbers will be multiplied by 2, and strings will be doubled (this is Python’s normal behavior with strings).

```
# multiply by a scalar
print(scientists * 2)
```

	Name	Born
0	Rosaline Franklin	1920-07-25
1	William Gosset	1876-06-13
2	Florence Nightingale	1820-05-12
3	Marie Curie	1867-11-07
4	Rachel Carson	1907-05-27
5	John Snow	1813-03-15
6	Alan Turing	1912-06-23
7	Johann Gauss	1777-04-30

	Died	Age	Occupation
0	1958-04-16	74	Chemist
1	1937-10-16	122	Statistician
2	1910-08-13	180	Nurse
3	1934-07-04	132	Chemist
4	1964-04-14	112	Biologist
5	1858-06-16	90	Physician
6	1954-06-07	82	Computer Scientist
7	1855-02-23	154	Mathematician

If your dataframes are all numeric values and you want to “add” the values cell-by-cell, you can use the `add` method. The automatic alignment can be better seen in [Chapter 4](#), when we concatenate dataframes together.

2.5 Making Changes to Series and DataFrames

Now that we know various ways of subsetting and slicing our data (See [Table 2-3](#)), we should now be able to alter our data objects.

2.5.1 Add Additional Columns

The `type` of the `Born` and `Died` columns are `objects`, meaning they are strings.

```
print(scientists['Born'].dtype)
object
print(scientists['Died'].dtype)
object
```

We can convert the strings to a proper `datetime` type so we can perform common datetime operations (e.g., take differences between dates or calculate the age). You can provide your own `format` if you have a date that has a specific format. A list of `format` variables can be found in the Python `datetime` module documentation.⁶ The format of our date looks like “YYYY-MM-DD”, so we can use the `%Y-%m-%d` format.

⁶datetime module documentation: <https://docs.python.org/3.5/library/datetime.html#strftime-and-strptime-behavior>

```
# format the 'Born' column as a datetime
born_datetime = pd.to_datetime(scientists['Born'], format='%Y-%m-%d')
print(born_datetime)

0    1920-07-25
1    1876-06-13
```

```

2    1820-05-12
3    1867-11-07
4    1907-05-27
5    1813-03-15
6    1912-06-23
7    1777-04-30
Name: Born, dtype: datetime64[ns]

# format the 'Died' column as a datetime
died_datetime = pd.to_datetime(scientists['Died'], format='%Y-%m-%d')

If we wanted, we can create a new set of columns that contain the datetime representations of the object (string) dates.

scientists['born_dt'], scientists['died_dt'] = (born_datetime,
                                                died_datetime)

print(scientists.head())

      Name      Born      Died  Age Occupation \
0  Rosaline Franklin  1920-07-25  1958-04-16  37     Chemist
1  William Gosset   1876-06-13  1937-10-16  61 Statistician
2  Florence Nightingale  1820-05-12  1910-08-13  90       Nurse
3  Marie Curie     1867-11-07  1934-07-04  66     Chemist
4  Rachel Carson   1907-05-27  1964-04-14  56 Biologist

      born_dt      died_dt
0  1920-07-25  1958-04-16
1  1876-06-13  1937-10-16
2  1820-05-12  1910-08-13
3  1867-11-07  1934-07-04
4  1907-05-27  1964-04-14

print(scientists.shape)
(8, 7)

```

2.5.2 Directly Change a Column

We can also assign a new value directly over the existing column. This example will show how to randomize the contents of a column. More complex calculations that involve multiple columns can be seen in [Chapter 9](#) on the `apply` method.

First, the original `Age` values,

```

print(scientists['Age'])

0    37
1    61
2    90
3    66
4    56
5    45
6    41
7    77
Name: Age, dtype: int64

```

Now let's shuffle the values.

```

import random

# set a seed so the randomness is always the same
random.seed(42)
random.shuffle(scientists['Age'])

/home/dchen/anaconda3/envs/book36/lib/python3.6/random.py:274:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

```

```
See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
    x[i], x[j] = x[j], x[i]
0      66
1      56
2      41
3      77
4      90
5      45
6      37
7      61
Name: Age, dtype: int64
```

The `SettingWithCopyWarning`⁷ in the previous code will tell us that the proper way of handling the statement would be to write it using `loc`, or we can use the built-in `sample` method to randomly sample the length of the column. In this example you need to `resetIndex` since `sample` only picks out the row index, so if you try to re-assign it or use it again, the “scrambled” values will automatically align to the index and order themselves back to the pre-`sample` order. The `drop=True` parameter in `resetIndex`, tells pandas not to insert the index into the dataframe columns, so only the values are kept.

⁷Indexing view versus Copy: <https://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy>

```
# the random_state is used to keep the 'randomization' less random
scientists['Age'] = scientists['Age'].\
    sample(len(scientists['Age']), random_state=24).\
    reset_index(drop=True) # values stay randomized

# we shuffled this column twice
print(scientists['Age'])

0      61
1      45
2      37
3      90
4      56
5      66
6      77
7      41
Name: Age, dtype: int64
```

You’ll notice that the `random.shuffle` method seems to work directly on the column. If you look at the documentation for `random.shuffle`,⁸ it will mention that the sequence will be shuffled ‘in place’. Meaning it will work directly on the sequence. Contrast this with the previous method where we assigned the newly calculated values to a separate variable before we can assign it to the column.

⁸Random shuffle: <https://docs.python.org/3.6/library/random.html#random.shuffle>

We can recalculate the ‘real’ age using `datetime` arithmetic. More about datetimes can be found in [Chapter 11](#).

```
# subtracting dates will give us number of days
scientists['age_days_dt'] = (scientists['died_dt'] - \
                           scientists['born_dt'])
print(scientists)

      Name      Born      Died  Age
0  Rosaline Franklin  1920-07-25  1958-04-16   61
1  William Gosset   1876-06-13  1937-10-16   45
2  Florence Nightingale  1820-05-12  1910-08-13   37
3  Marie Curie     1867-11-07  1934-07-04   90
4  Rachel Carson   1907-05-27  1964-04-14   56
5  John Snow       1813-03-15  1858-06-16   66
```

```

6      Alan Turing    1912-06-23    1954-06-07    77
7      Johann Gauss   1777-04-30    1855-02-23    41

0      Occupation      born_dt      died_dt age_days_dt
0      Chemist        1920-07-25  1958-04-16  13779 days
1      Statistician    1876-06-13  1937-10-16  22404 days
2      Nurse          1820-05-12  1910-08-13  32964 days
3      Chemist        1867-11-07  1934-07-04  24345 days
4      Biologist       1907-05-27  1964-04-14  20777 days
5      Physician       1813-03-15  1858-06-16  16529 days
6      Computer Scientist 1912-06-23  1954-06-07  15324 days
7      Mathematician   1777-04-30  1855-02-23  28422 days

# we can convert the value to just the year
# using the astype method
scientists['age_years_dt'] = scientists['age_days_dt'].\
    astype('timedelta64[Y]')
print(scientists)

      Name      Born      Died  Age \
0  Rosaline Franklin  1920-07-25  1958-04-16  61
1  William Gosset    1876-06-13  1937-10-16  45
2  Florence Nightingale  1820-05-12  1910-08-13  37
3  Marie Curie       1867-11-07  1934-07-04  90
4  Rachel Carson     1907-05-27  1964-04-14  56
5  John Snow          1813-03-15  1858-06-16  66
6  Alan Turing        1912-06-23  1954-06-07  77
7  Johann Gauss       1777-04-30  1855-02-23  41

      Occupation      born_dt      died_dt age_days_dt      \
0      Chemist        1920-07-25  1958-04-16  13779 days
1      Statistician    1876-06-13  1937-10-16  22404 days
2      Nurse          1820-05-12  1910-08-13  32964 days
3      Chemist        1867-11-07  1934-07-04  24345 days
4      Biologist       1907-05-27  1964-04-14  20777 days
5      Physician       1813-03-15  1858-06-16  16529 days
6      Computer Scientist 1912-06-23  1954-06-07  15324 days
7      Mathematician   1777-04-30  1855-02-23  28422 days

      age_years_dt
0            37.0
1            61.0
2            90.0
3            66.0
4            56.0
5            45.0
6            41.0
7            77.0

```

Many functions and methods in pandas will have an `inplace` parameter that you can set to `True`, if you want to perform the action “`inplace`”, meaning it will directly change the given column without returning anything.

Note

We could've directly assigned the column to the `datetime` converted, but the point is an assignment still needed to be preformed. The `random.shuffle` example performs its method ‘in place’, so there is nothing that is explicitly returned from the function. The value passed into the function is directly manipulated.

2.5.3 Dropping Values

To drop a column we can either select all the columns we want to keep or we can use the `drop` method on our dataframe.⁹

⁹drop method: <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html>

```
# all the current columns in our data
print(scientists.columns)

Index(['Name', 'Born', 'Died', 'Age', 'Occupation', 'born_dt',
       'died_dt', 'age_days_dt', 'age_years_dt'],
      dtype='object')

# drop the shuffled age column
# you provide the axis=1 argument to drop column-wise
scientists_dropped = scientists.drop(['Age'], axis=1)

# columns after dropping our column
print(scientists_dropped.columns)

Index(['Name', 'Born', 'Died', 'Occupation', 'born_dt', 'died_dt',
       'age_days_dt', 'age_years_dt'],
      dtype='object')
```

2.6 Exporting and Importing Data

So far we've been importing data. However it's common practice to export or save out datasets while processing them. Datasets are either saved out as final cleaned versions of data or intermediate steps. Both of these can be used for analysis or another part of the data processing pipeline.

2.6.1 pickle

Python has a way to **pickle** data. This is python's way to serialize and save a binary format of data.

2.6.1.1 series

Many of the export methods for a **Series** are also available for a **DataFrame**. Those who have experience with **numpy** will know there is a **save** method on **ndarrays**. This method has been deprecated, and the replacement is to use the **to_pickle** method in its place.

```
names = scientists['Name']
print(names)

0      Rosaline Franklin
1      William Gosset
2    Florence Nightingale
3        Marie Curie
4        Rachel Carson
5        John Snow
6        Alan Turing
7        Johann Gauss
Name: Name, dtype: object

# pass in a string to the path you want to save
names.to_pickle('../output/scientists_names_series.pickle')
```

The pickle output is in a binary format, meaning if you try to open it in a text editor, you will see a bunch of garbled characters.

If the object you are saving is an intermediate step in a set of calculations that you want to save, or if you know your data will stay in the Python world, saving objects to a **pickle**, will be optimized for Python as well as disk storage space. However, this means that people who do not

use Python, will not be able to read the data.

2.6.1.2 DataFrame

The same method can be used on `DataFrame` objects.

```
scientists.to_pickle('../output/scientists_df.pickle')
```

2.6.1.3 Reading pickle Data

To read in `pickle` data we can use the `pd.read_pickle` function.

```
# for a Series
scientist_names_from_pickle = pd.read_pickle(
    '../output/scientists_names_series.pickle')
print(scientist_names_from_pickle)

0      Rosaline Franklin
1      William Gosset
2    Florence Nightingale
3      Marie Curie
4      Rachel Carson
5      John Snow
6      Alan Turing
7      Johann Gauss
Name: Name, dtype: object

# for a DataFrame
scientists_from_pickle = pd.read_pickle(
    '../output/scientists_df.pickle')
print(scientists_from_pickle)

          Name        Born       Died   Age \
0  Rosaline Franklin  1920-07-25  1958-04-16  61
1  William Gosset  1876-06-13  1937-10-16  45
2  Florence Nightingale  1820-05-12  1910-08-13  37
3  Marie Curie  1867-11-07  1934-07-04  90
4  Rachel Carson  1907-05-27  1964-04-14  56
5  John Snow  1813-03-15  1858-06-16  66
6  Alan Turing  1912-06-23  1954-06-07  77
7  Johann Gauss  1777-04-30  1855-02-23  41

          Occupation      born_dt      died_dt age_days_dt \
0        Chemist  1920-07-25  1958-04-16  13779 days
1  Statistician  1876-06-13  1937-10-16  22404 days
2        Nurse  1820-05-12  1910-08-13  32964 days
3        Chemist  1867-11-07  1934-07-04  24345 days
4     Biologist  1907-05-27  1964-04-14  20777 days
5     Physician  1813-03-15  1858-06-16  16529 days
6  Computer Scientist  1912-06-23  1954-06-07  15324 days
7  Mathematician  1777-04-30  1855-02-23  28422 days

       age_years_dt
0            37.0
1            61.0
2            90.0
3            66.0
4            56.0
5            45.0
6            41.0
7            77.0
```

You will see `pickle` files saved as `.p`, `.pkl`, or `.pickle`.

2.6.2 CSV

Comma-separated values (CSV) are the most flexible data storage type. For each row, the

column information will be separated with a comma. The comma is not the only type of delimiter. Some files will be delimited by a tab (tsv), or even a semi-colon. The main reason why CSVs are a preferred data format when collaborating and sharing data is because any program can open it. It can even be opened in a text editor.

The `Series` and `DataFrame` have a `to_csv` method to write a CSV file. The documentation for `Series`¹⁰ and `DataFrame`¹¹ have many different ways you can modify the resulting CSV file. For example, if you wanted to save a `tsv` file because there are commas in your data, you can change the `sep` parameter ([Appendix O](#)).

¹⁰Saving a Series to csv: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.to_csv.html

¹¹Saving a DataFrame to a CSV: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_csv.html

```
# save a series into a CSV  
names.to_csv('../output/scientist_names_series.csv')  
  
# save a dataframe into a TSV,  
# a tab-separated value  
scientists.to_csv('../output/scientists_df.tsv', sep='\t')
```

2.6.2.1 Removing row number from output

If you open the CSV or TSV file created, you will notice that the first ‘column’ will look like the row number of the dataframe. Many times this is not needed, especially when collaborating with other people. However, keep in mind, it is really saving the ‘row label’, which *may* be important. The documentation will show that there is an `index` parameter that to write row names (`index`).

```
# do not write the row names in the CSV output  
scientists.to_csv('../output/scientists_df_no_index.csv', index=False)
```

2.6.2.2 Importing CSV Data

Importing CSV files was shown in Chapter [1.2](#). It uses the `pd.read_csv` function. From the documentation,¹² you can see there are various ways you can read in a `csv`. Look at [Appendix O](#) if you need more information on using function parameters.

¹²read csv documentation: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

2.6.3 Excel

Excel, probably the most common data type (or second most common, next to CSVs). Excel has a bad reputation within the data science community. Some of the reasons of why are listed in Chapter [1.1](#). The goal of this book isn’t to bash Excel, but to teach a reasonable alternative tool for data analytics. In short, the more you can do your work in a scripting language, the easier it will be to scale up to larger projects, catch and fix mistakes, and collaborate. Excel has its own scripting language if you absolutely have to work in it.

2.6.3.1 Series

The `Series` does not have an explicit `to_excel` method. If you have a `Series` that needs to be exported to an Excel file. One way is to convert the `Series` into a 1 column `DataFrame`.

```
# convert the Series into a DataFrame
# before saving it to an excel file
names_df = names.to_frame()

import xlwt # this needs to be installed
# xls file
names_df.to_excel('../output/scientists_names_series_df.xls')

import openpyxl # this needs to be installed
# newer xlsx file
names_df.to_excel('../output/scientists_names_series_df.xlsx')
```

2.6.3.2 DataFrame

From above, you can see how to export a `DataFrame` to an Excel file. The documentation¹³ does show ways on how to further fine tune the output. For example, you can output to a specific ‘sheet’ using the `sheet_name` parameter

¹³DataFrame to Excel Documentation: http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to_excel.html

```
# saving a DataFrame into Excel format
scientists.to_excel('../output/scientists_df.xlsx',
                    sheet_name='scientists',
                    index=False)
```

2.6.4 Feather Format to Interface with R

There is a format called “feather” that is used to save a binary object that can also be loaded into the R language. The main benefit here is that it is faster than writing and reading a CSV between the languages. The general rule of thumb for using this data format is to use it only as an intermediate data format, and to not use the `feather` format for long-term storage. That is, only use it in your code to pass in data into R, and not use it to save a final version of your data.

It is installed via `conda install -c conda-forge feather-format` OR `pip install feather-format`, you can use the `to_feather` method on a dataframe to save the feather object. Not every dataframe will be able to be converted into a feather object. For example, our current dataset has a column of `date` values in it, which is not currently supported by feather.¹⁴

¹⁴Feather dates, ArrowNotImplementedError: <https://github.com/wesm/feather/issues/121>

2.6.5 Many Data Output Types

There are many ways `Pandas` can export and import data, `to_pickle`, `to_csv`, and `to_excel`, and `to_feather` are only some of the data formats that can make its way into `Pandas DataFrame`s. [Table 2-4](#) lists some of these output formats.

Table 2-4: Various DataFrame export methods

Export method	Description
<code>to_clipboard</code>	save data into the system clipboard for pasting
<code>to_dense</code>	convert data into a regular ‘dense’ <code>DataFrame</code>
<code>to_dict</code>	convert data into a Python dict
<code>to_gbq</code>	convert data into a Google BigQuery table
<code>to_hdf</code>	save data into a hierachal data format (HDF)
<code>to_msgpack</code>	save data into a portable JSON-like binary
<code>to_html</code>	convert data to a HTML table
<code>to_json</code>	convert data into a JSON string
<code>to_latex</code>	convert data as a L ^A T _E Xtabular environment
<code>to_records</code>	convert data into a record array
<code>to_string</code>	show <code>DataFrame</code> as a string for <code>stdout</code>
<code>to_sparse</code>	convert data into a <code>SparceDataFrame</code>
<code>to_sql</code>	save data into a SQL database
<code>to_stata</code>	convert data into a Stata <code>dta</code> file

For more complicated and general data conversions (not necessarily just exporting), the `odo` library¹⁵ has a consistent way to convert between data formats ([Appendix T](#)).

¹⁵Odo library <http://odo.readthedocs.org/en/latest/>

2.7 Conclusion

This chapter went in a little more detail about how the `Pandas Series` and `DataFrame` objects work in `Python`. There were some simpler examples of data cleaning shown, and a few common ways to export data to share with others. Chapters 1 and 2 should give you a good basis on how `Pandas` as a library works.

The next chapter will cover the basics of plotting in `Python` and `Pandas`. Data visualization is not only used in the end of an analysis to plot results, it is heavily utilized throughout the entire data pipeline.

Chapter 3. Introduction to Plotting

3.1 Introduction

Data visualization is as much a part of the data processing step as the data presentation step. It is much easier to compare values when they are plotted than numeric values. By visualizing data we are able to get a better intuitive sense of our data, than by looking at tables of values alone. Additionally, visualizations can also bring to light, hidden patterns in data, that you, the analyst, can exploit for model selection.

Concept map

1. Prior knowledge
 - (a) Containers
 - (b) Using functions
 - (c) Subsetting and indexing
 - (d) Classes
2. matplotlib
3. seaborn

Objectives

This chapter will cover:

1. matplotlib
2. seaborn
3. plotting in pandas

The quintessential example for making visualizations of data is Anscombe's quartet. This was a dataset created by English statistician Frank Anscombe to show the importance of statistical graphs.

The Anscombe dataset contains 4 sets of data, where each set contains 2 continuous variables. Each set has the same mean, variance, correlation, and regression line. However, only when the data are visualized is it obvious that each set does not follow the same pattern. This goes to show the benefits of visualizations and the pitfalls of only looking at summary statistics.

```
# the anscombe dataset can be found in the seaborn library
import seaborn as sns
anscombe = sns.load_dataset("anscombe")
print(anscombe)
```

	dataset	x	y
0	I	10.0	8.04
1	I	8.0	6.95
2	I	13.0	7.58
3	I	9.0	8.81
4	I	11.0	8.33
5	I	14.0	9.96
6	I	6.0	7.24
7	I	4.0	4.26
8	I	12.0	10.84
9	I	7.0	4.82
10	I	5.0	5.68
11	II	10.0	9.14
12	II	8.0	8.14
13	II	13.0	8.74
14	II	9.0	8.77
15	II	11.0	9.26
16	II	14.0	8.10
17	II	6.0	6.13
18	II	4.0	3.10
19	II	12.0	9.13
20	II	7.0	7.26
21	II	5.0	4.74
22	III	10.0	7.46
23	III	8.0	6.77
24	III	13.0	12.74
25	III	9.0	7.11
26	III	11.0	7.81
27	III	14.0	8.84
28	III	6.0	6.08
29	III	4.0	5.39
30	III	12.0	8.15
31	III	7.0	6.42
32	III	5.0	5.73
33	IV	8.0	6.58
34	IV	8.0	5.76
35	IV	8.0	7.71
36	IV	8.0	8.84
37	IV	8.0	8.47
38	IV	8.0	7.04
39	IV	8.0	5.25
40	IV	19.0	12.50
41	IV	8.0	5.56
42	IV	8.0	7.91
43	IV	8.0	6.89

3.2 matplotlib

matplotlib is Python's fundamental plotting library. It is extremely flexible and gives the user full control of all elements of the plot.

Importing **matplotlib**'s plotting features is a little different from our previous package imports. You can think of it as the package **matplotlib** and all the plotting utilities are under a subfolder (or sub package) called **pyplot**. Just like how we imported a package and gave it an abbreviated name, we can do the same with **matplotlib . pyplot**.

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

Most of the basic plots will start with **plt . plot** . In our example it takes a vector for the x-values, and a corresponding vector for the y-values ([Figure 3–1](#)).

```
# create a subset of the data
# contains only dataset 1 from anscombe
dataset_1 = anscombe[anscombe['dataset'] == 'I']

plt.plot(dataset_1['x'], dataset_1['y'])
```

By default, **plt . plot** will draw lines. If we want it to draw circles (points) instead we can pass an

'o' parameter to tell `plt.plot` to use points ([Figure 3-2](#))

```
plt.plot(dataset_1['x'], dataset_1['y'], 'o')
```

We can repeat this process for the rest of the `datasets` in our anscombe data.

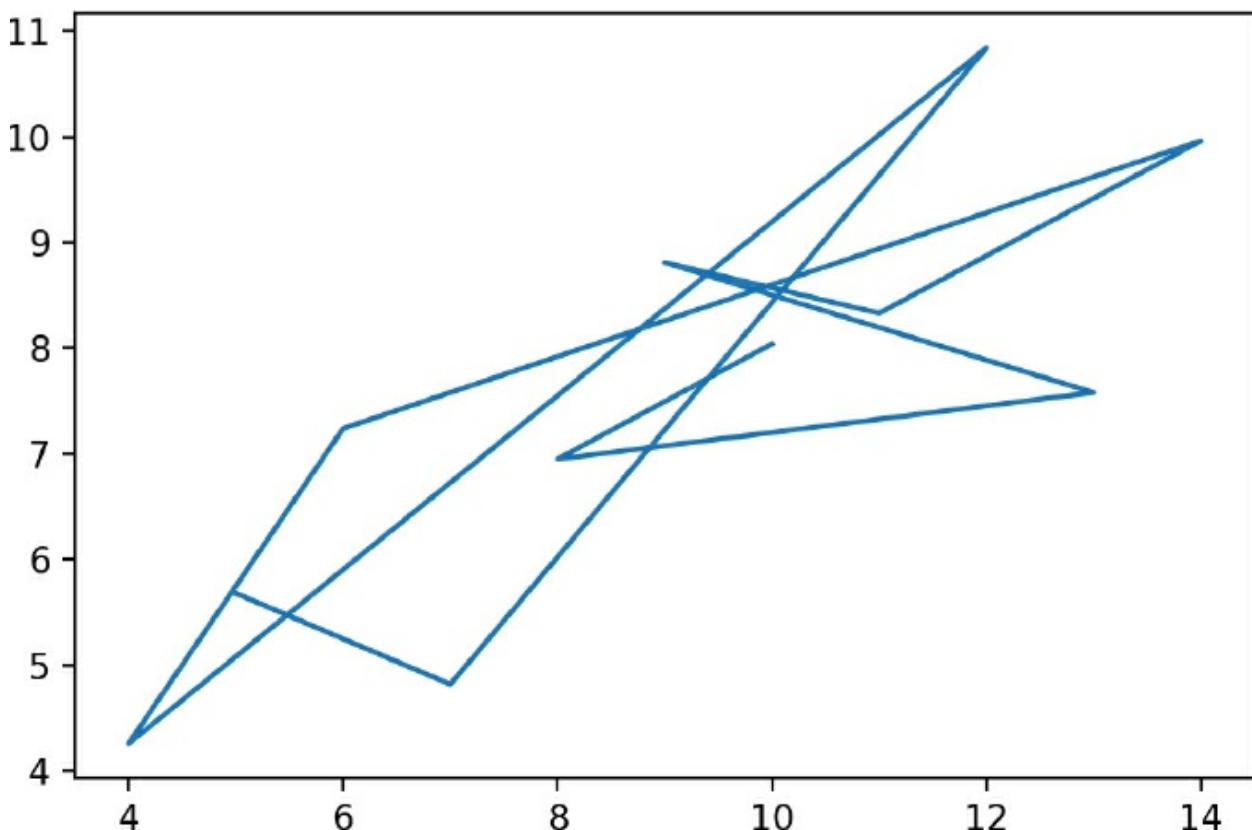
```
# create subsets of the anscombe data
dataset_2 = anscombe[anscombe['dataset'] == 'II']
dataset_3 = anscombe[anscombe['dataset'] == 'III']
dataset_4 = anscombe[anscombe['dataset'] == 'IV']
```

Now, we could make these plots individually, one at a time, but `matplotlib` has a way to create subplots. That is, you can specify the dimensions of your final figure, and put in smaller plots to fit the specified dimensions. This way you can present your results in a single figure, instead of completely separate ones.

The `subplot` syntax takes 3 parameters.

1. number of rows in figure for subplots
2. number of columns in figure for subplots
3. subplot location

Figure 3-1: Anscombe dataset I



The subplot location is sequentially numbered and plots are placed left-to-right then top-to-bottom. If we try to plot this now (by just running the code below), we will get an empty figure ([Figure 3-3](#)). All we have done so far is create a figure and split the figure into a 2x2 grid where plots can be placed. Since no plots were created and inserted, nothing will show up.

```

# create the entire figure where our subplots will go
fig = plt.figure()

# tell the figure how the subplots should be laid out
# in the example below we will have
# 2 row of plots, each row will have 2 plots

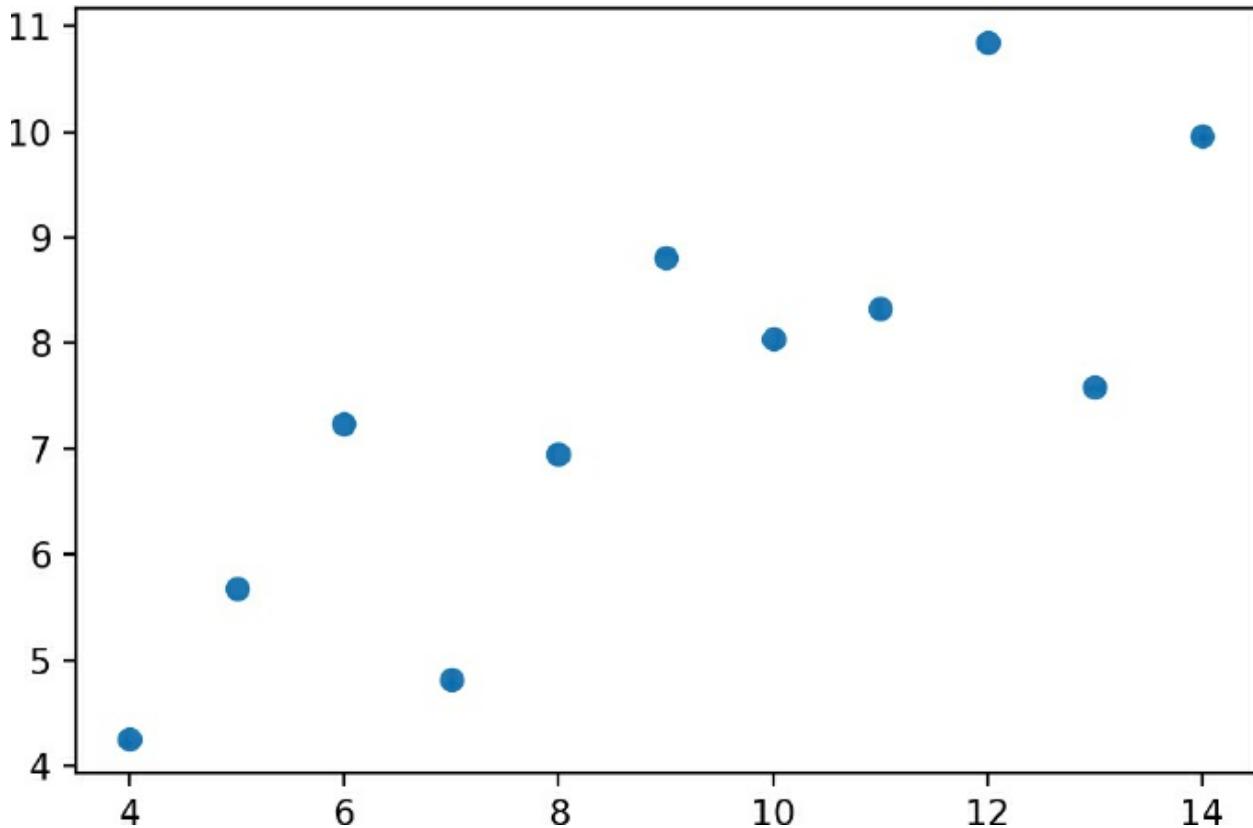
# subplot has 2 rows and 2 columns, plot location 1
axes1 = fig.add_subplot(2, 2, 1)

# subplot has 2 rows and 2 columns, plot location 2
axes2 = fig.add_subplot(2, 2, 2)

# subplot has 2 rows and 2 columns, plot location 3
axes3 = fig.add_subplot(2, 2, 3)

```

Figure 3-2: Anscombe dataset I Using Points



```

# subplot has 2 rows and 2 columns, plot location 4
axes4 = fig.add_subplot(2, 2, 4)

```

We can use the `plot` method on each axes to create our plot ([Figure 3-4](#)).

```

# add a plot to each of the axes created above
axes1.plot(dataset_1['x'], dataset_1['y'], 'o')
axes2.plot(dataset_2['x'], dataset_2['y'], 'o')
axes3.plot(dataset_3['x'], dataset_3['y'], 'o')
axes4.plot(dataset_4['x'], dataset_4['y'], 'o')

[<matplotlib.lines.Line2D at 0x7f8f96598b70>]

```

Finally, we can add a label to our subplots, and use the `tightLayout` to make sure the axes are spread apart from one another ([Figure 3-5](#)).

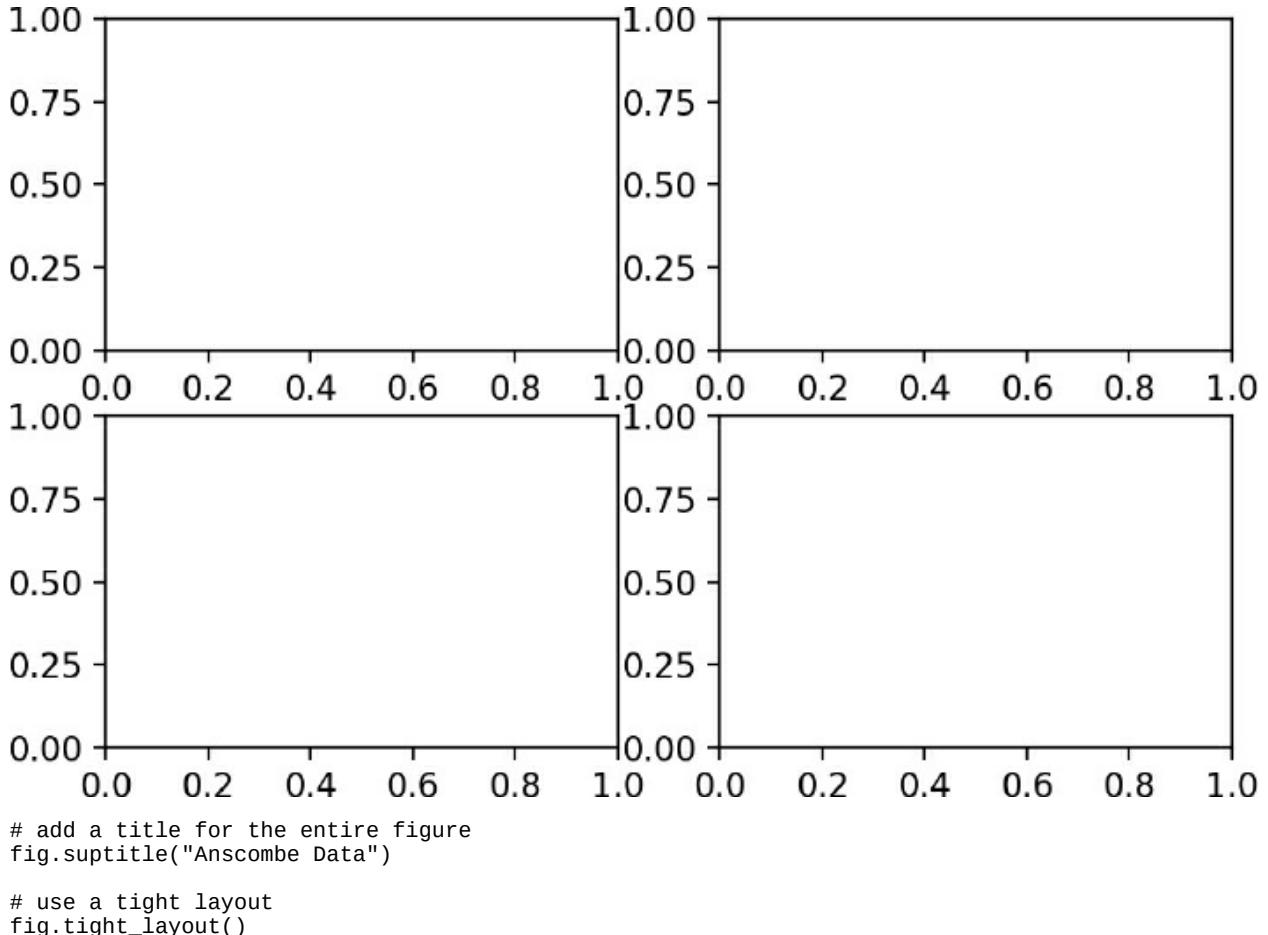
```

# add a small title to each subplot
axes1.set_title("dataset_1")
axes2.set_title("dataset_2")
axes3.set_title("dataset_3")

```

```
axes4.set_title("dataset_4")
```

Figure 3-3: Matplotlib figure with 4 empty axes'



The anscombe data visualizations should depict why just looking at summary statistic values can be misleading. The moment the points were visualized, it becomes clear that even though each dataset has the same summary statistic values, the relationship between points vastly differ across datasets.

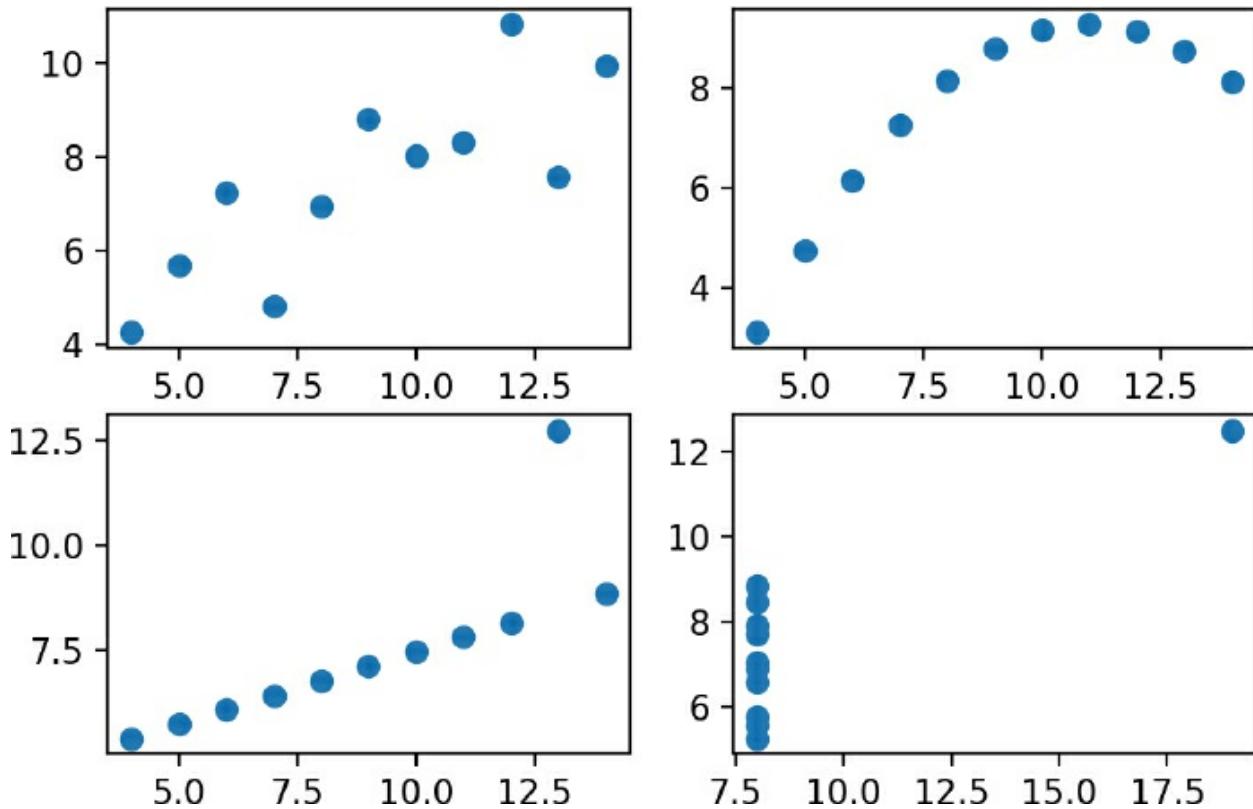
To finish off the anscombe example, we can add `set_xlabel()` and `set_ylabel()` to each of the subplots to add x and y labels, just like how we added a title to the figure.

Before moving on and showing how to create more statistical plots, be familiar with the `matplotlib` documentation on “Parts of a Figure”¹. I have reproduced their oldfigure in [Figure 3-6](#), and the newer figure in [Figure 3-7](#)

¹Parts of a matplotlib figure: http://matplotlib.org/faq/usage_faq.html#parts-of-a-figure

One of the most confusing parts of plotting in Python is the use of ‘axis’ and ‘axes’. Especially when trying to verbally describe the different parts (since they are pronounced the same). In the anscombe example, each individual subplot plot was an `axes`. An `axes` has both an x and y axis. All 4 subplots make the figure.

Figure 3-4: Matplotlib figure with 4 scatter plots'



The remainder of the chapter will show you how to create statistical plots, first with `matplotlib` and later using a higher-level plotting library based on `matplotlib` specifically made for statistical graphics, `seaborn`.

3.3 Statistical Graphics using `matplotlib`

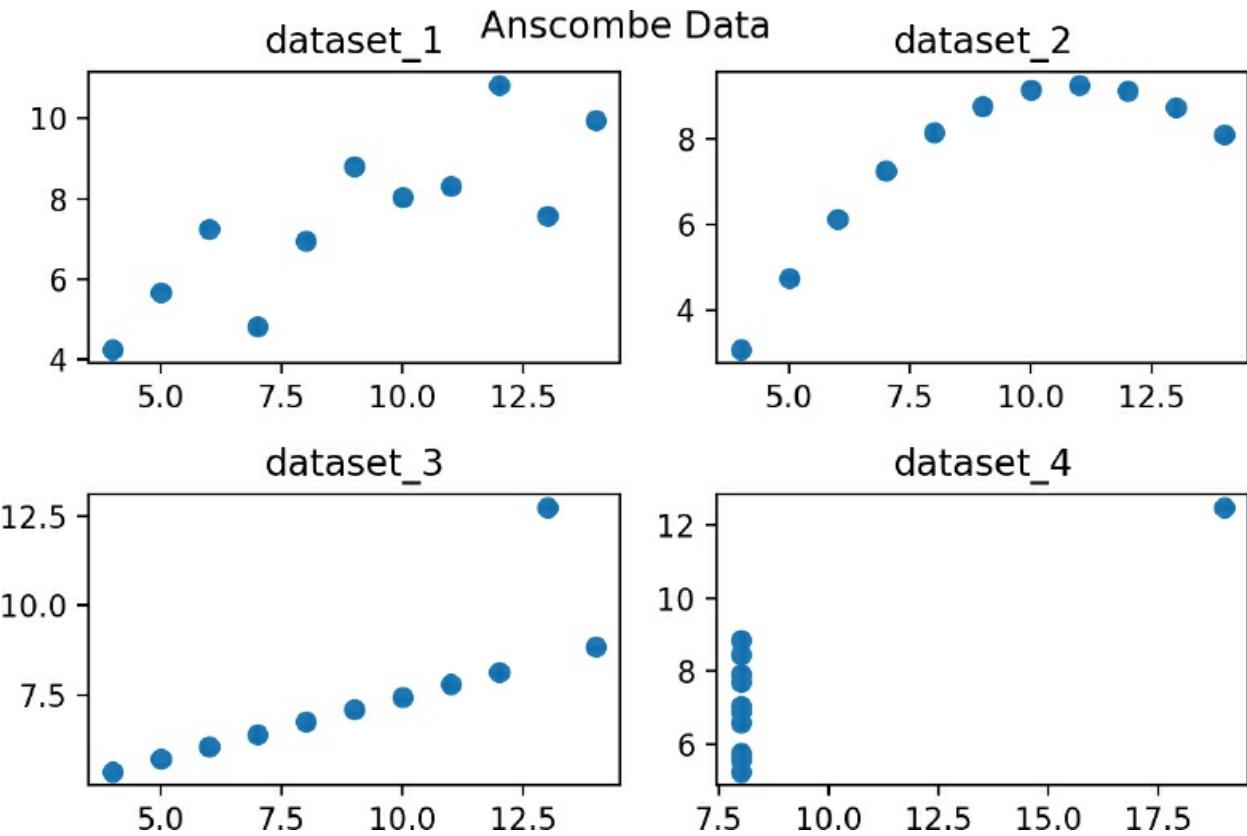
The tips data we will be using for the next series of visualizations come from the `seaborn` library. This dataset contains the amount of tip people leave for various variables. For example, the total cost of the bill, the size of the party, the day of the week, the time of day, etc.

We can load this data just like the `anscombe` data above.

```
tips = sns.load_dataset("tips")
print(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

Figure 3-5: Anscombe data visualization



3.3.1 Univariate

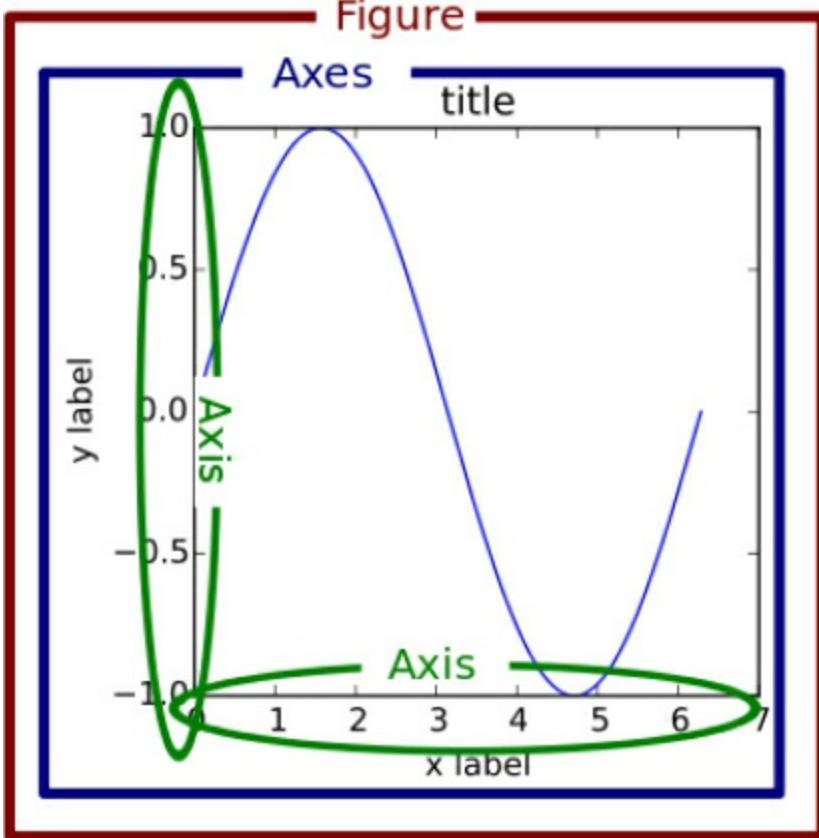
In statistics jargon, ‘univariate’ refers to a single variable.

3.3.1.1 Histograms

Histograms are the most common means of looking at a single variable. The values are ‘binned’, meaning they are grouped together and plotted to show the distribution of the variable ([Figure 3-8](#)).

```
fig = plt.figure()
axes1 = fig.add_subplot(1, 1, 1)
axes1.hist(tips['total_bill'], bins=10)
axes1.set_title('Histogram of Total Bill')
axes1.set_xlabel('Frequency')
axes1.set_ylabel('Total Bill')
fig.show()
```

Figure 3-6: One of the most confusing parts of plotting in Python is the use of ‘axis’ and ‘axes’ since they are pronounced the same but refer to different parts of a figure. This was the older version of the “Parts of a Figure” figure from the matplotlib documentation.



3.3.2 Bivariate

In statistics jargon, ‘bivariate’ refers to a two variables.

3.3.2.1 Scatter plot

Scatter plots are used when a continuous variable is plotted against another continuous variable ([Figure 3-9](#)).

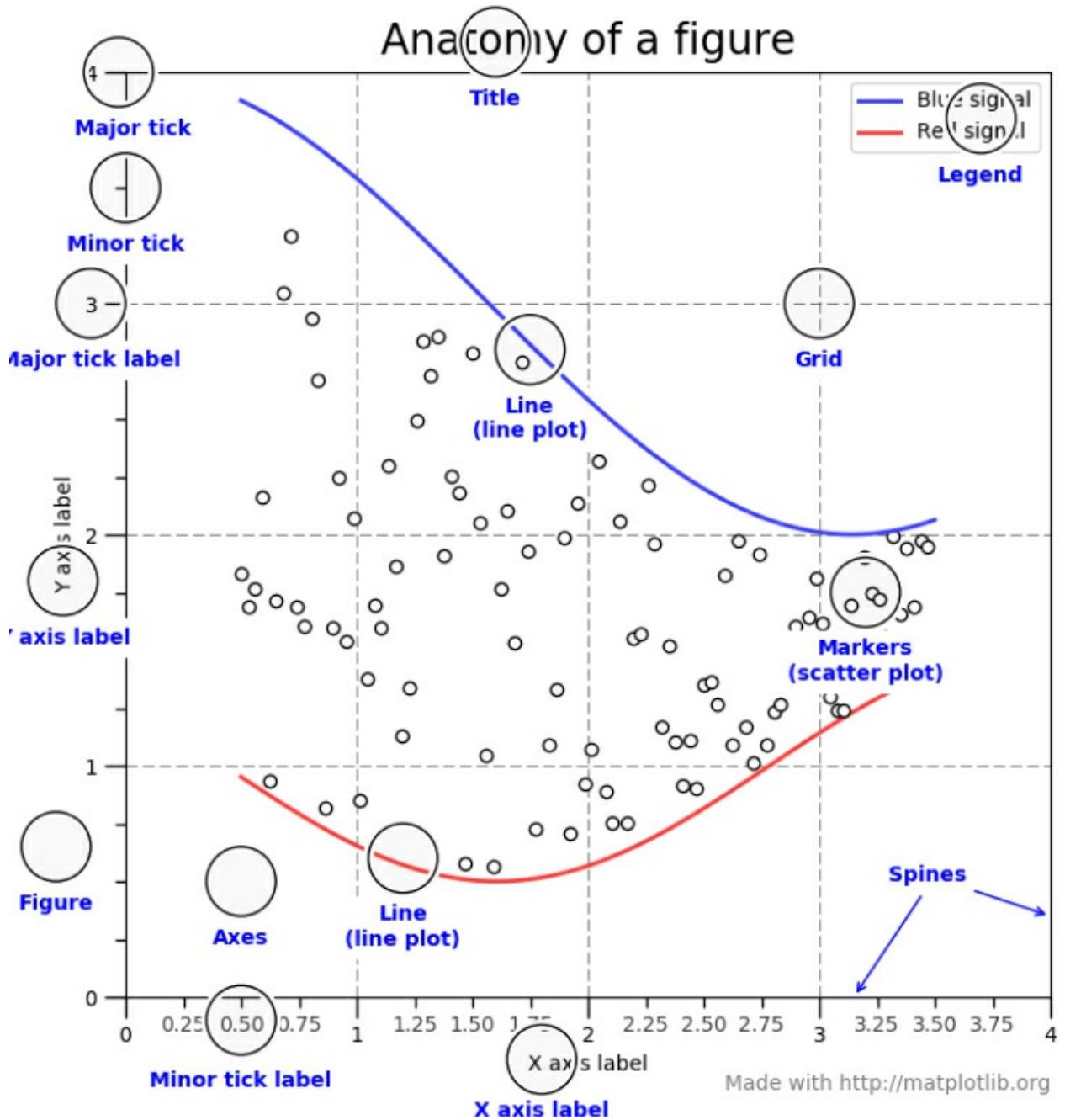
```
scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)
axes1.scatter(tips['total_bill'], tips['tip'])
axes1.set_title('Scatterplot of Total Bill vs Tip')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')
scatter_plot.show()
```

3.3.2.2 Box plot

Boxplots are used when a discrete variable is plotted against a continuous variable ([Figure 3-10](#)).

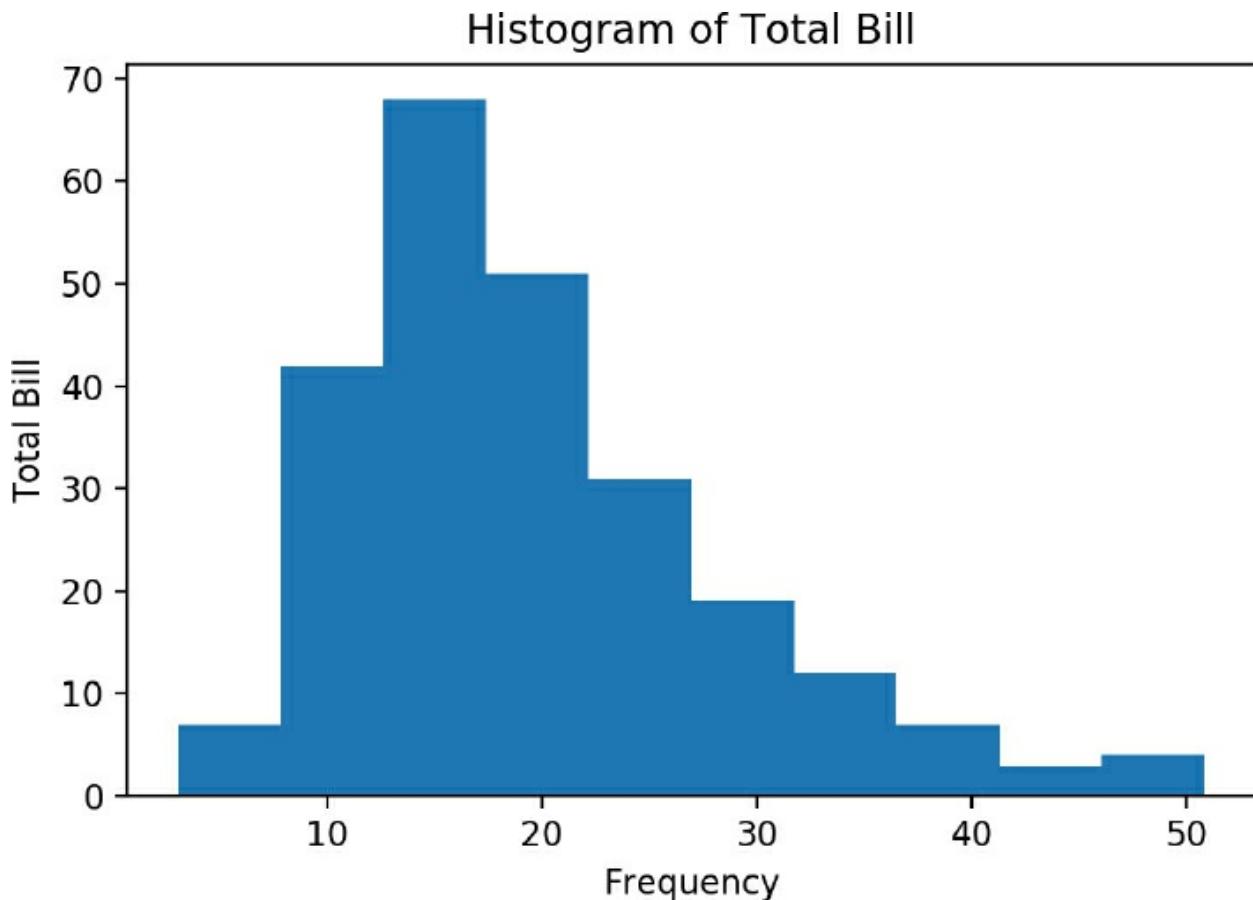
```
boxplot = plt.figure()
axes1 = boxplot.add_subplot(1, 1, 1)
axes1.boxplot()
```

Figure 3-7: A newer version of the “Parts of a Figure”, with more details to the other aspects of a figure. Unlike the older figure, the newer one is also completely created using matplotlib.



```
# first argument of boxplot is the data
# since we are plotting multiple pieces of data
# we have to put each piece of data into a list
[tips[tips['sex'] == 'Female']['tip'],
 tips[tips['sex'] == 'Male']['tip']],
# We can then pass in an optional labels parameter
# to label the data we passed
labels=['Female', 'Male'])
```

Figure 3-8: Histogram using matplotlib.



```
axes1.set_xlabel('Sex')
axes1.set_ylabel('Tip')
axes1.set_title('Boxplot of Tips by Sex')
boxplot.show()
```

3.3.3 multivariate

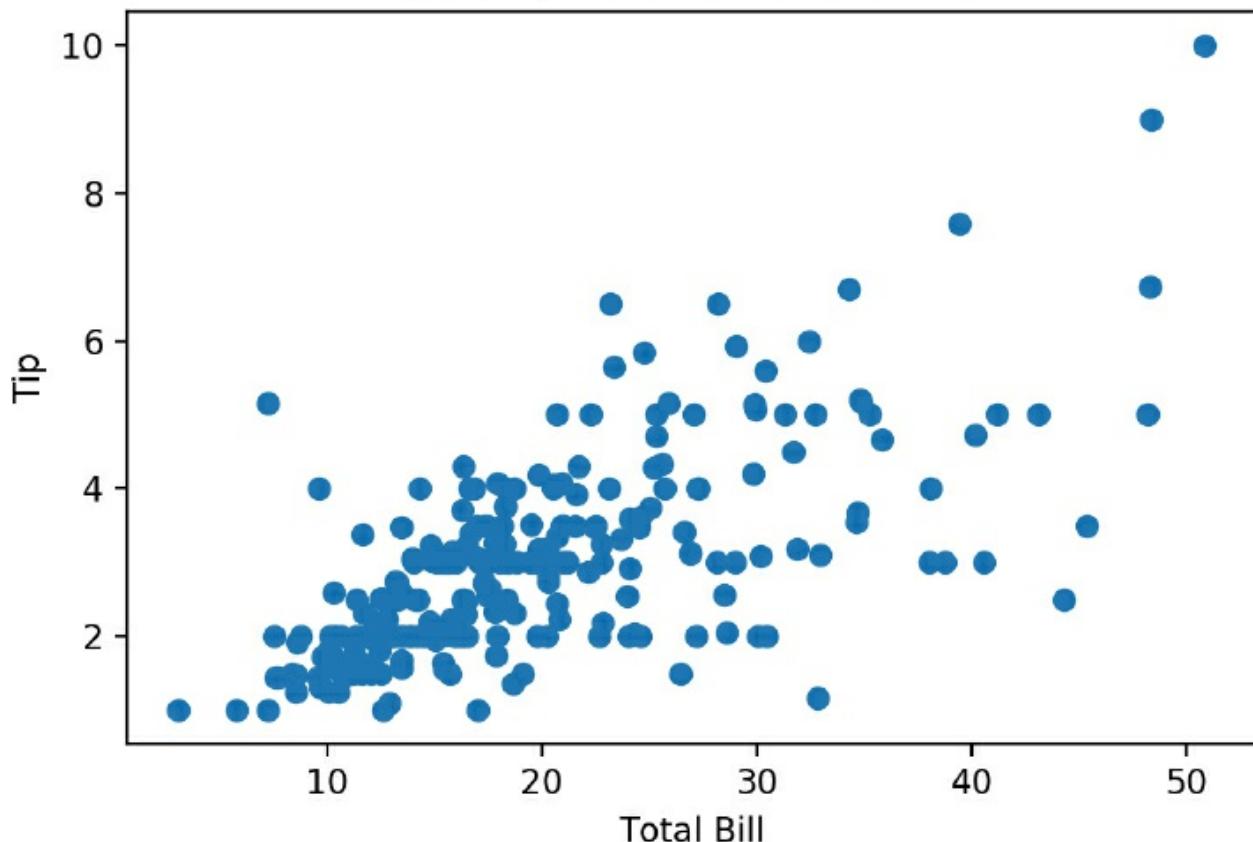
Plotting multivariate data is tricky. There isn't a panacea or template that can be used for every case. Let's build on the scatter plot above. If we wanted to add another variable, say `sex`, one option would be to color the points by the third variable.

If we wanted to add a fourth variable, we could add size to the dots. The only caveat with using size as a variable is humans are not very good at differentiating areas. Sure, if there's an enormous dot next to a tiny one, your point will be conveyed, but smaller differences are hard to distinguish, and may add clutter to your visualization. One way to reduce clutter is to add some value of transparency to the individual points, this way many overlapping points will show a darker region of a plot than less crowded areas.

The general rule of thumb is different colors are much easier to distinguish than changes in size. If you have to use areas, be sure that you are actually plotting relative areas. A common pitfall is to use map a value to the radius of a circle for plots, but since the formula for a circle is πr^2 , your areas are actually on a squared scale, which is not only misleading, but wrong.

Figure 3-9: Scatter plot using matplotlib.

Scatterplot of Total Bill vs Tip



Colors are also difficult to pick. Humans do not perceive hues on a linear scale, so though also needs to go into picking color palettes. Luckily matplotlib² and seaborn³ come with their own set of color pallets, and tools like colorbrewer⁴ help with picking good color palettes.

²Matplotlib colormaps: <http://matplotlib.org/users/colormaps.html>

³Seaborn color palettes: http://stanford.edu/~mwaskom/software/seaborn-dev/tutorial/color_palettes.html

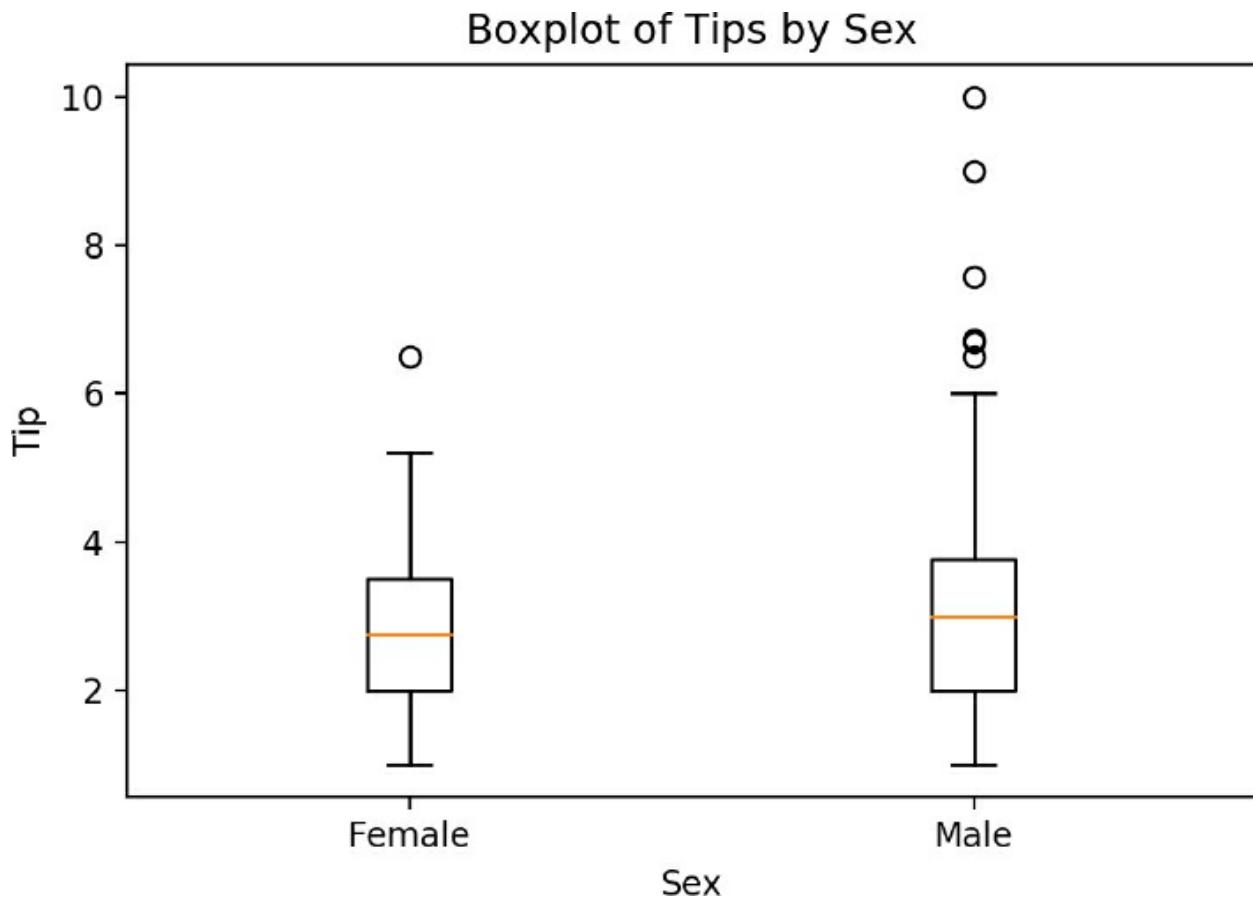
⁴Colorbrewer color palettes <http://colorbrewer2.org/>

Figure 3-11 uses color to add a third variable, sex, to our scatter plot.

```
# create a color variable based on the sex
def recode_sex(sex):
    if sex == 'Female':
        return 0
    else:
        return 1

tips['sex_color'] = tips['sex'].apply(recode_sex)
```

Figure 3-10: Boxplot plot using matplotlib.



```

scatter_plot = plt.figure()
axes1 = scatter_plot.add_subplot(1, 1, 1)
axes1.scatter(
    x=tips['total_bill'],
    y=tips['tip'],

    # set the size of the dots based on party size
    # we multiply the values by 10 to make the points bigger
    # and also to emphasize the difference
    s=tips['size'] * 10,

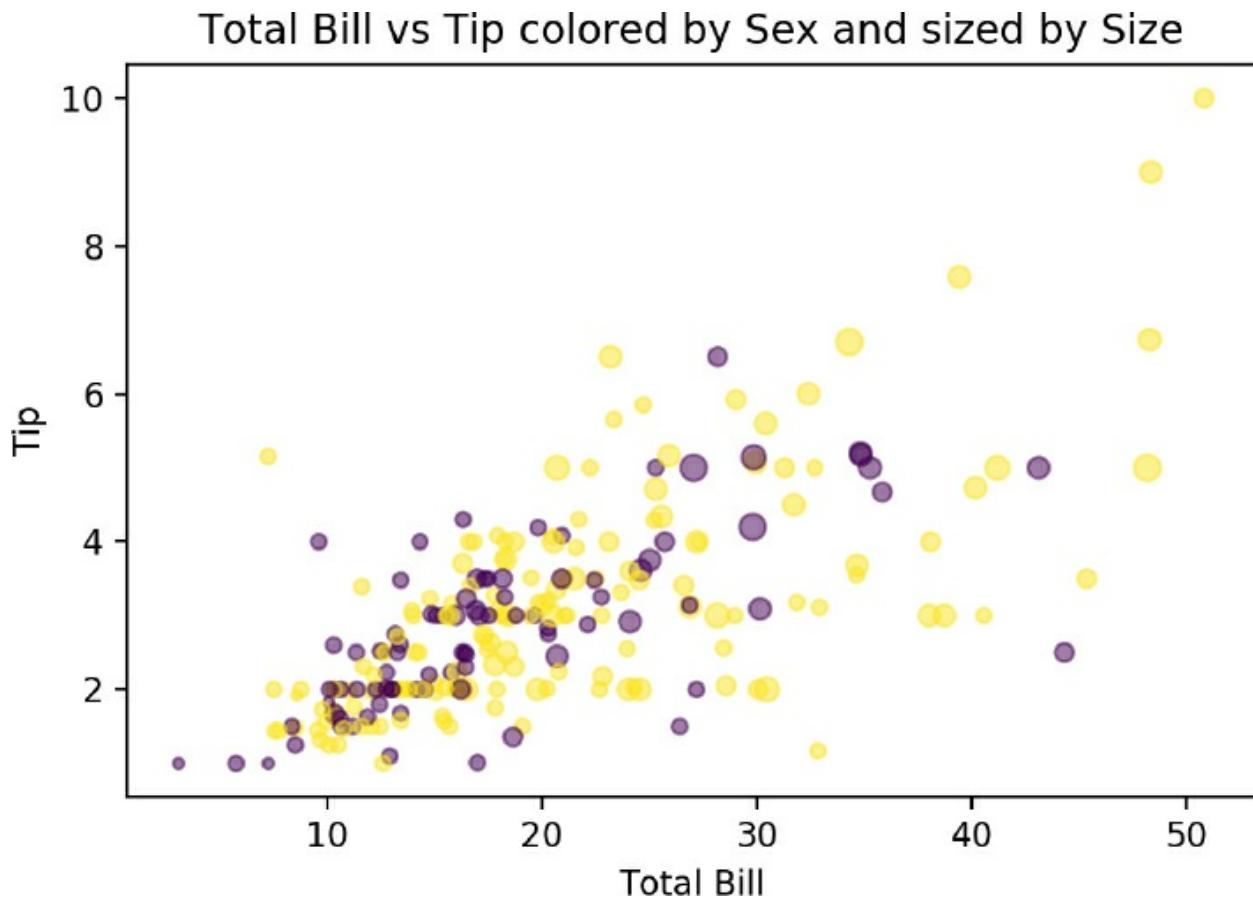
    # set the color for the sex
    c=tips['sex_color'],

    # set the alpha so points are more transparent
    # this helps with overlapping points
    alpha=0.5)

axes1.set_title('Total Bill vs Tip colored by Sex and sized by Size')
axes1.set_xlabel('Total Bill')
axes1.set_ylabel('Tip')
scatter_plot.show()

```

Figure 3-11: Scatter plot using matplotlib with color.



3.4 seaborn

The `matplotlib` library can be thought of as the core foundational plotting tool in Python. `seaborn` builds on `matplotlib` by providing a higher level interface for statistical graphics. It provides an interface to produce prettier and more complex visualizations with fewer lines of code.

The `seaborn` library is tightly integrated with `pandas` and the rest of the PyData stack (numpy, pandas, scipy, statsmodels), making visualizations from any part of the data analysis process a breeze. Since `seaborn` is built on top of `matplotlib`, the user still has the ability to fine tune the visualizations.

We've already loaded the `seaborn` library for its datasets.

```
# load seaborn if you have not done so already
import seaborn as sns

tips = sns.load_dataset("tips")
```

3.4.1 Univariate

Just like we did with the `matplotlib` examples, we will make a series of univariate plots.

3.4.1.1 Histograms

Histograms are created using `sns . distplot5` ([Figure 3-12](#)).

⁵seaborn distplot documentation:

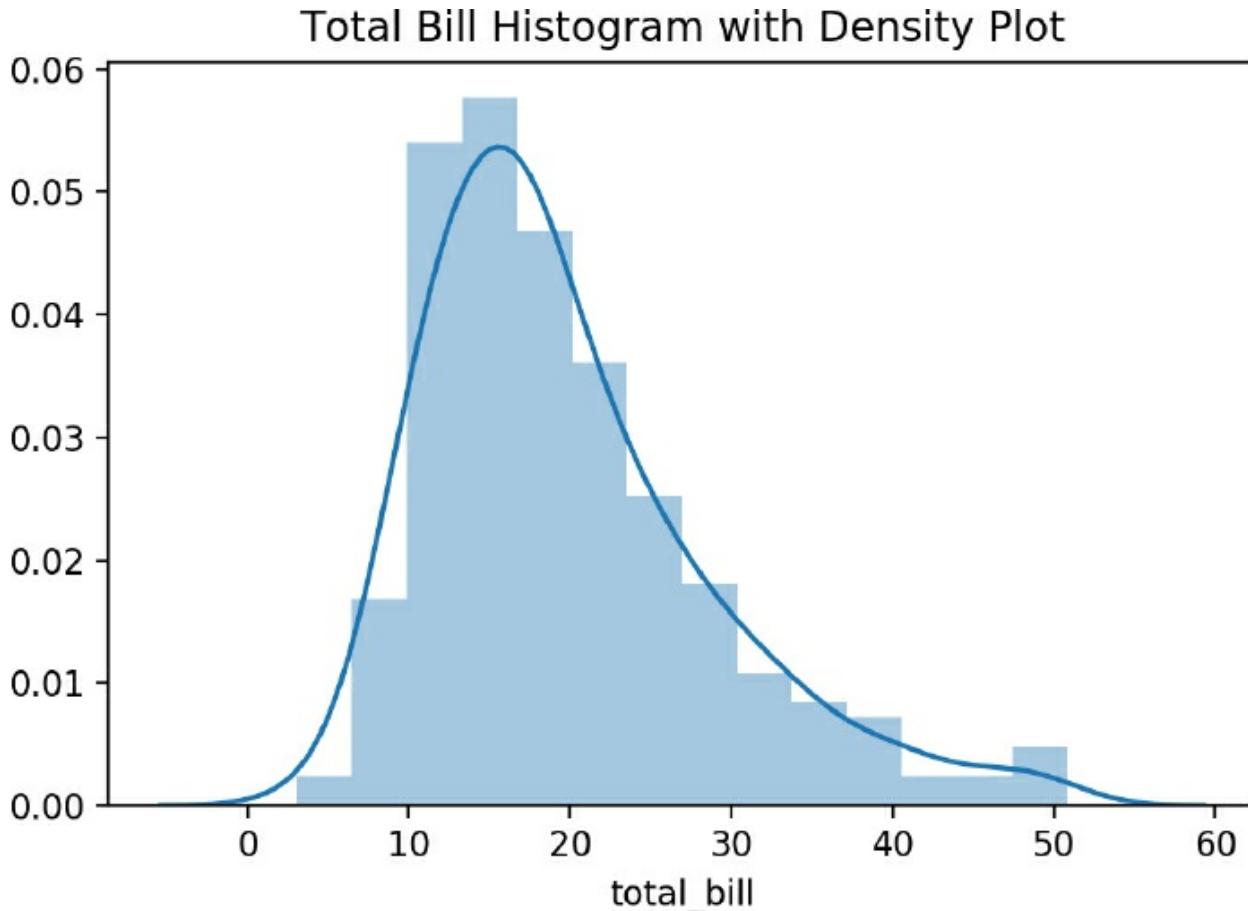
<https://stanford.edu/~mwaskom/software/seaborn/generated/seaborn.distplot.html#seaborn.distplot>

```
# this subplots function is a short cut for
# how we created separate figure objects and
# added individual subplots (axes) to the figure
hist, ax = plt.subplots()

# use the distplot function from seaborn to create our plot
ax = sns.distplot(tips['total_bill'])
ax.set_title('Total Bill Histogram with Density Plot')

plt.show() # note we still need matplotlib.pyplot to show the figure
```

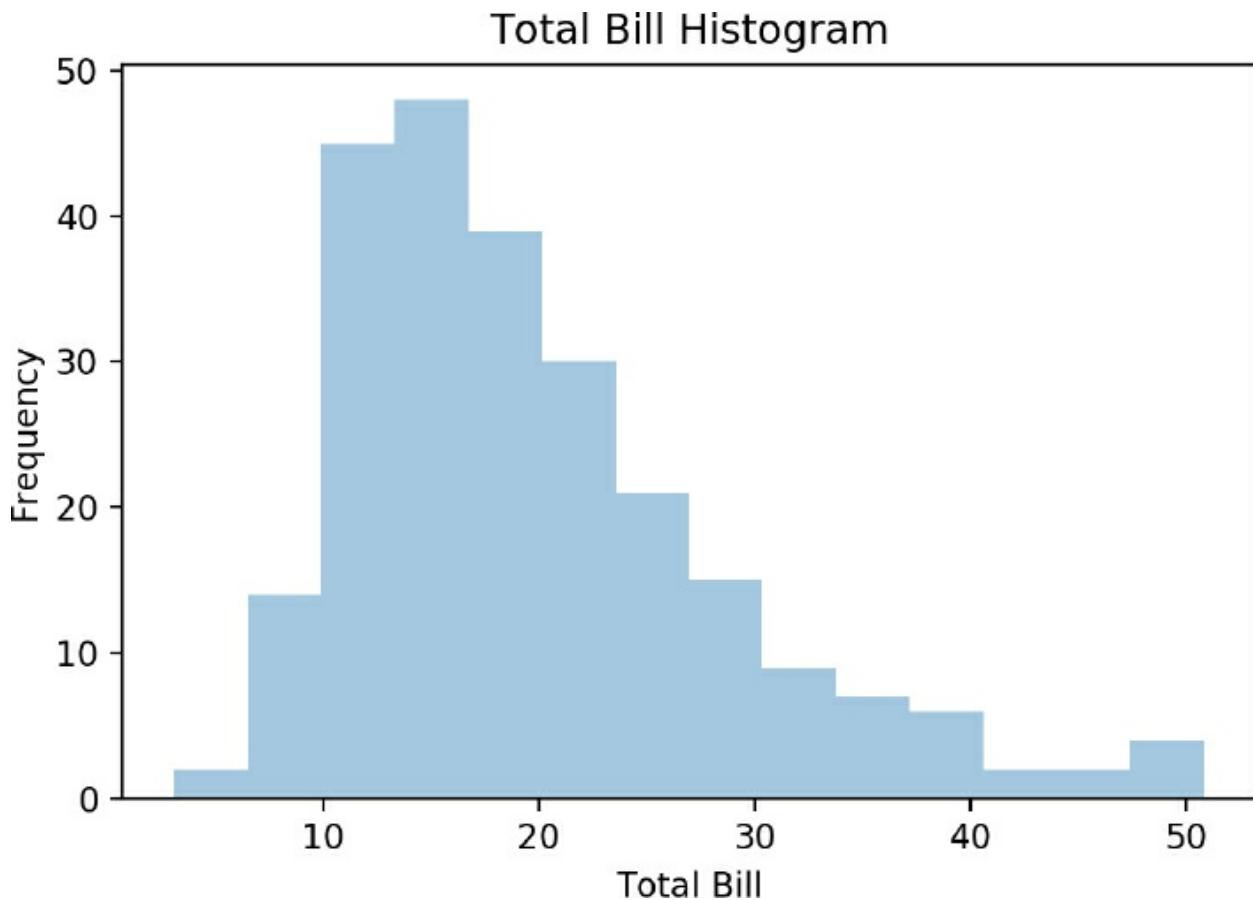
Figure 3-12: Seaborn distplot



The default `distplot` will plot both a histogram and a density plot (using kernel density estimation). If we just wanted the histogram we can set the `kde` parameter to `False`. The results are shown in [Figure 3-13](#)

```
hist, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], kde=False)
ax.set_title('Total Bill Histogram')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Frequency')
plt.show()
```

Figure 3-13: Seaborn distplot

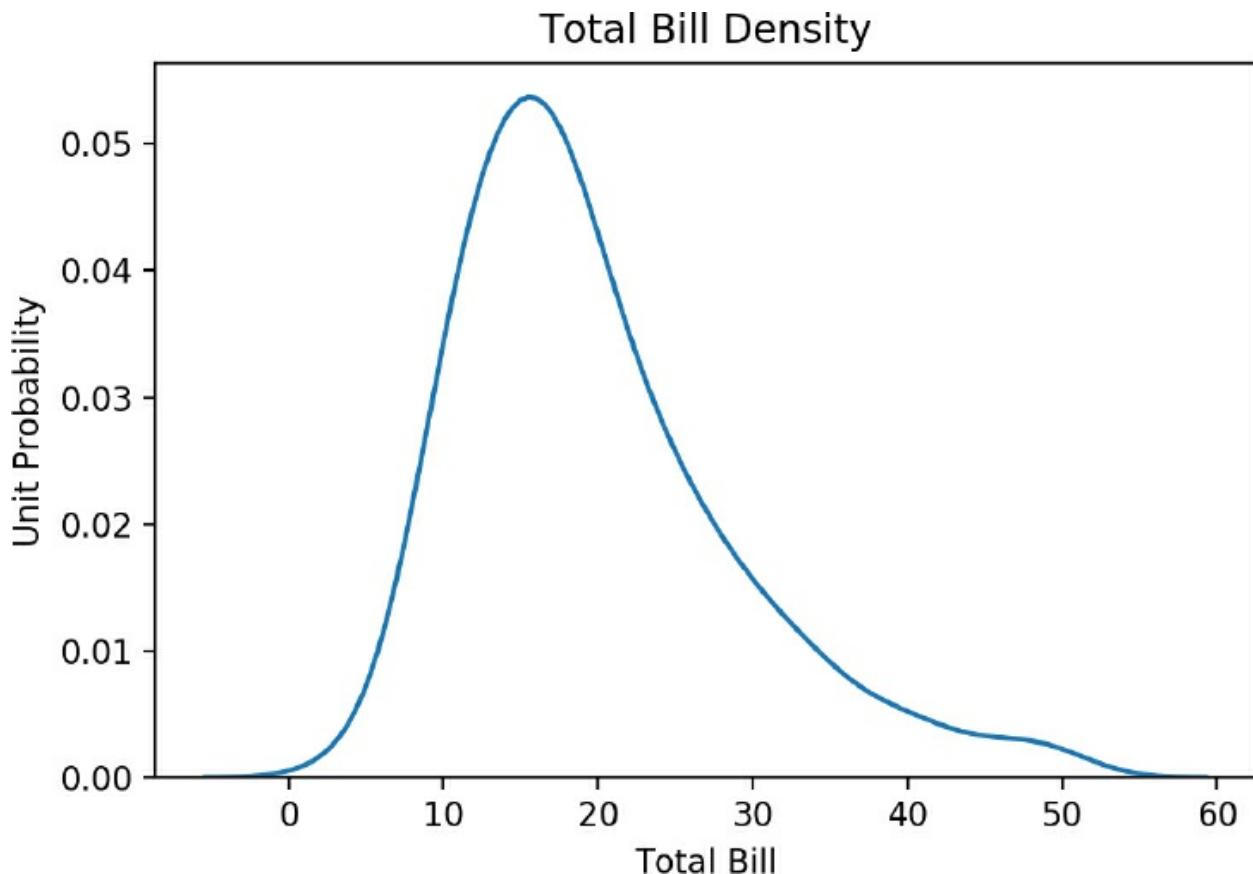


3.4.1.2 Density Plot (Kernel Density Estimation)

Density plots are another way to visualize a univariate distribution ([Figure 3-14](#)). It essentially works by drawing a normal distribution centered at each data point, and smooths out the overlapping plots such that the area under the curve is 1.

```
den, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], hist=False)
ax.set_title('Total Bill Density')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Unit Probability')
plt.show()
```

Figure 3-14: Seaborn density plot using distplot



There is also a `sns.kdeplot` that can be used if you just want a density plot.

3.4.1.3 Rug plot

Rug plots are a 1-dimensional representation of a variable's distribution. They are typically used with other plots to enhance a visualization. [Figure 3-15](#) shows a histogram overlaid with a density plot and a rug plot on the bottom.

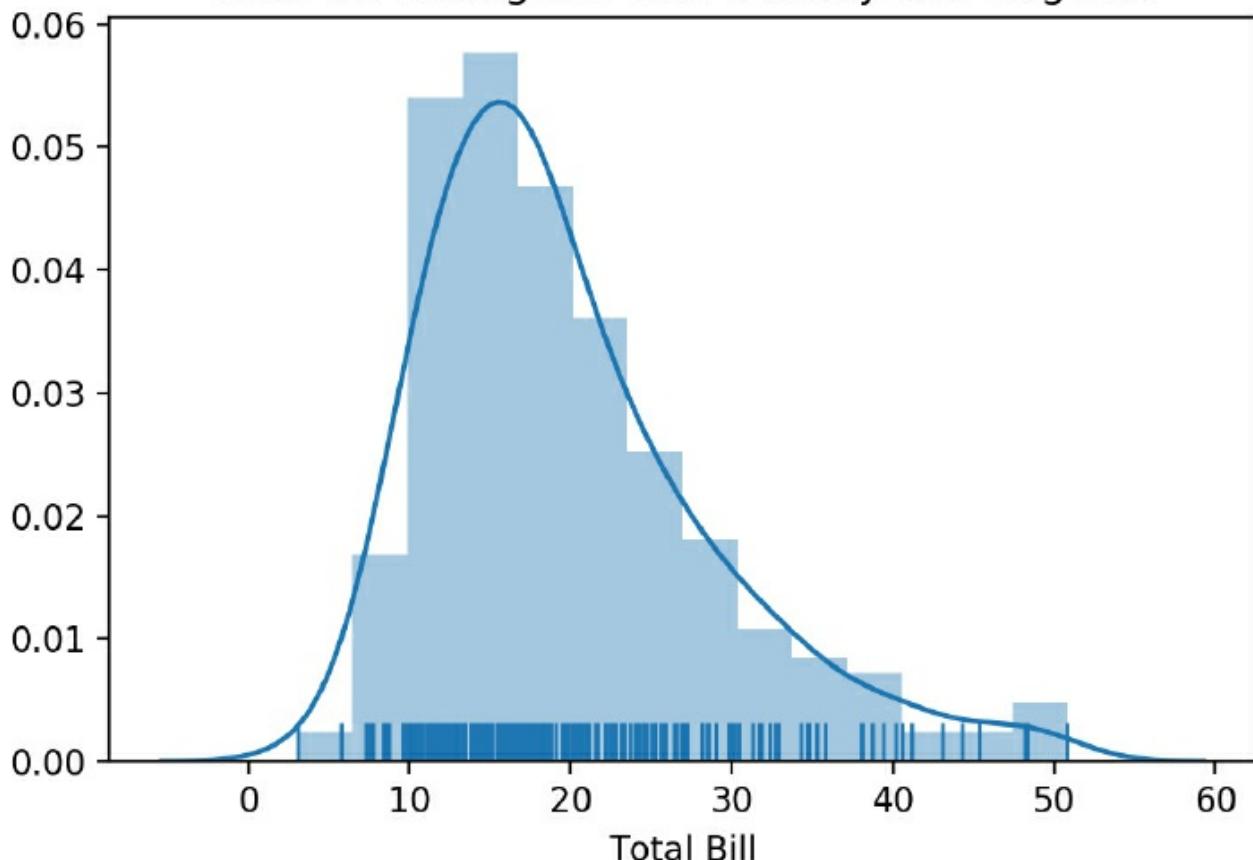
```
hist_den_rug, ax = plt.subplots()
ax = sns.distplot(tips['total_bill'], rug=True)
ax.set_title('Total Bill Histogram with Density and Rug Plot')
ax.set_xlabel('Total Bill')
plt.show()
```

3.4.1.4 Count plot (Bar plot)

Bar plots are very similar to histograms, but instead of binning values to produce a distribution, bar plots can be used to count discrete variables. A `countplot` ([Figure 3-16](#)) is used for this purpose.

Figure 3-15: Seaborn distplot with rugs

Total Bill Histogram with Density and Rug Plot



```
count, ax = plt.subplots()
ax = sns.countplot('day', data=tips)
ax.set_title('Count of days')
ax.set_xlabel('Day of the Week')
ax.set_ylabel('Frequency')
plt.show()
```

3.4.2 Bivariate

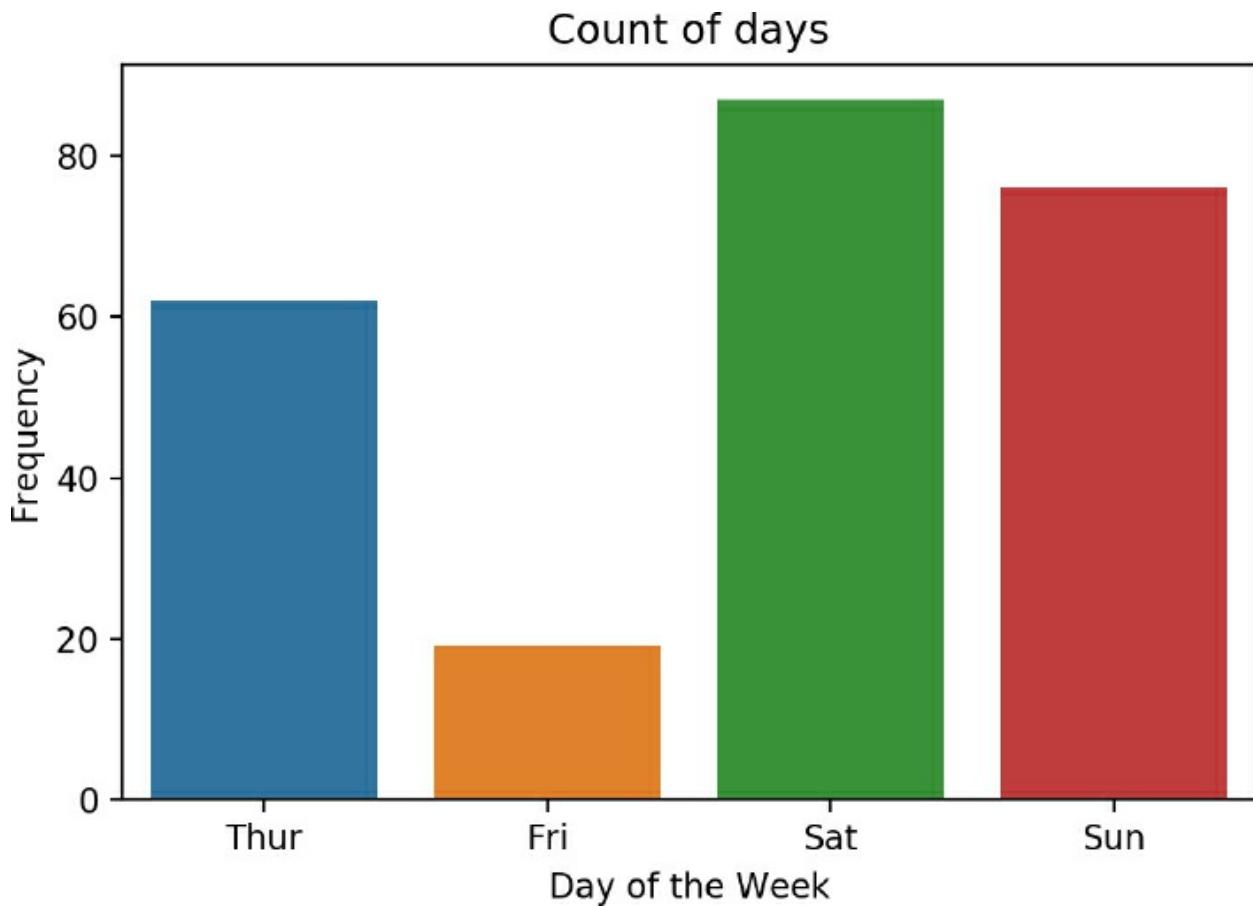
We will not use the `seaborn` library to plot 2 variables.

3.4.2.1 Scatter plot

There are a few ways to create a scatter plot in `seaborn`. There is no explicit function named `scatter`. Instead, we use `regplot`. It will plot a scatter plot and also fit a regression line. We can set `fit_reg =False` so it only shows the scatter plot ([Figure 3-17](#)).

```
scatter, ax = plt.subplots()
ax = sns.regplot(x='total_bill', y='tip', data=tips)
```

Figure 3-16: Seaborn countplot



```
ax.set_title('Scatterplot of Total Bill and Tip')
ax.set_xlabel('Total Bill')
ax.set_ylabel('Tip')
plt.show()
```

There is a similar function, `lmplot`, that can also plot scatter plots. Internally, `lmplot` calls `regplot`, so `regplot` is a more general plot function. The main difference is that `regplot` creates an axes (See [Figure 3-6](#)) and `lmplot` creates a figure ([Figure 3-18](#)).

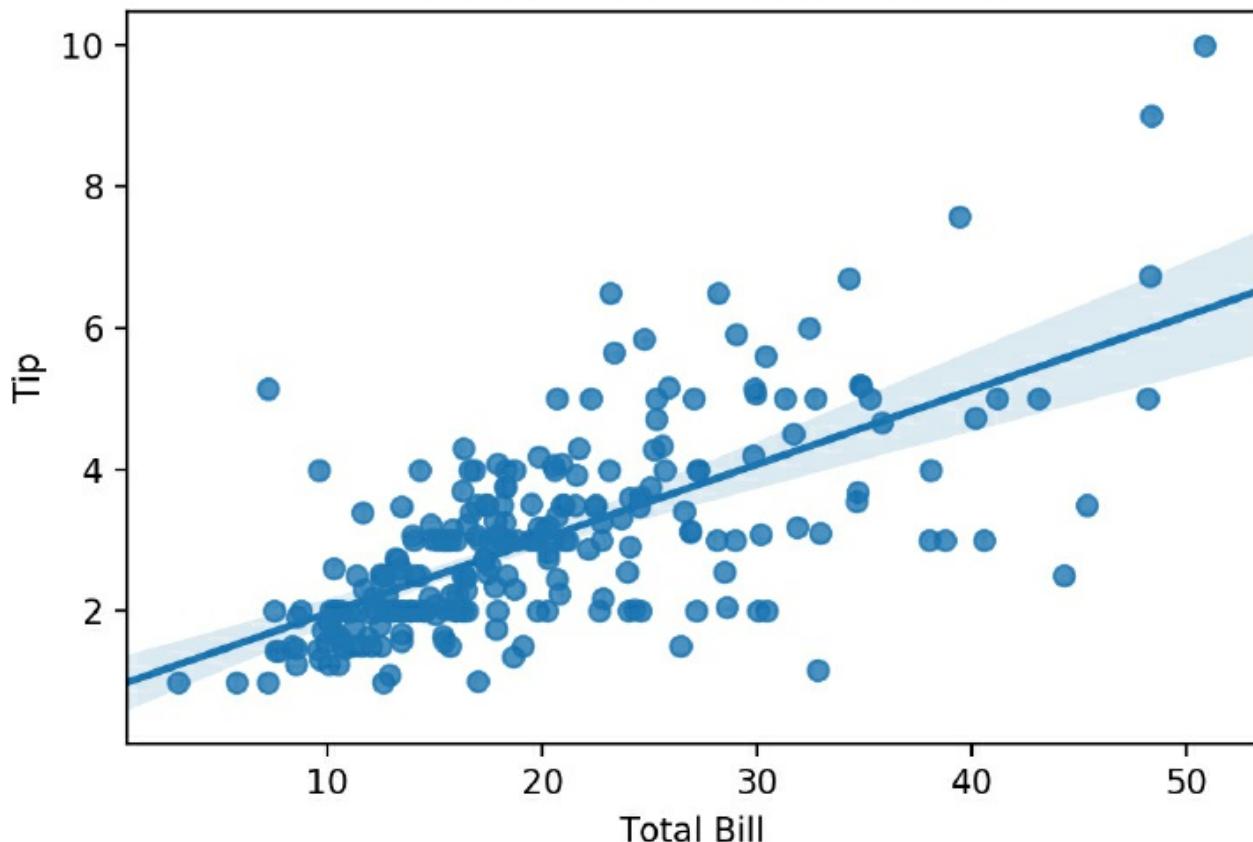
```
fig = sns.lmplot(x='total_bill', y='tip', data=tips)
plt.show()
```

We can also plot our scatter plot with a univariate plot on each axis using `jointplot` ([Figure 3-19](#)). One major difference is that `jointplot` does not return an `axes`, so we do not need to create the figure with an axes to place our plot. It technically creates a `JointGrid` object.

```
joint = sns.jointplot(x='total_bill', y='tip', data=tips)
joint.set_axis_labels(xlabel='Total Bill', ylabel='Tip')
# add a title, set font size,
```

Figure 3-17: Seaborn scatter plot using regplot

Scatterplot of Total Bill and Tip



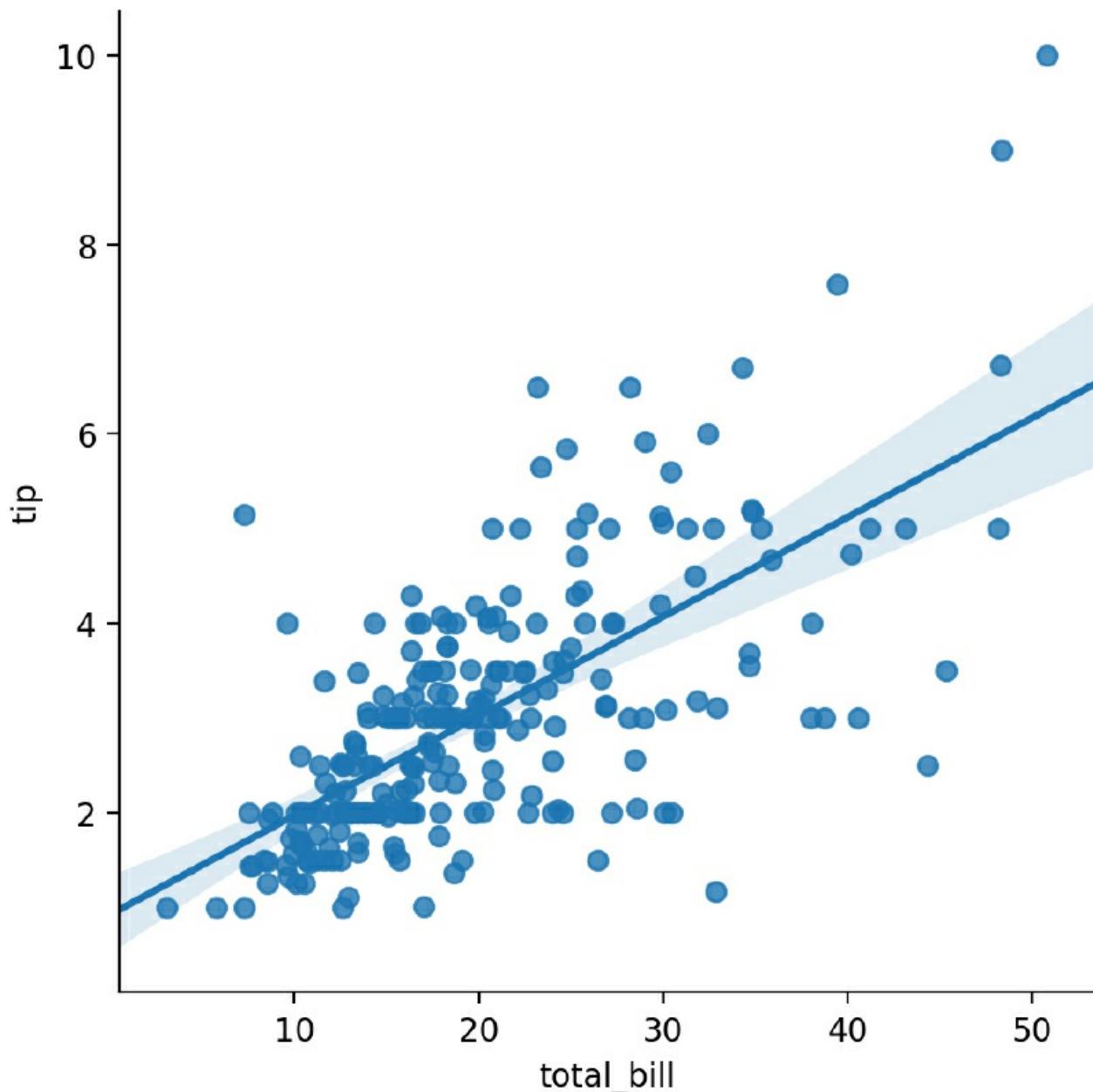
```
# and move the text above the total bill axes
joint.fig.suptitle('Joint plot of Total Bill and Tip',
                    fontsize=10, y=1.03)
```

3.4.2.2 Hexbin plot

Scatter plots are great for comparing two variables. However, sometimes there are too many points for a scatter plot to be meaningful. One way to get around this is to bin points on the plot together. Just like how histograms can bin a variable to create a bar, `hexbin` can bin two variables ([Figure 3-20](#)). A hexagon is used because it is the most efficient shape to cover an arbitrary 2D surface. This is an example of `seaborn` building on top of `matplotlib` as `hexbin` is a `matplotlib` function.

```
hex = sns.jointplot(x="total_bill", y="tip", data=tips, kind="hex")
hex.set_axis_labels(xlabel='Total Bill', ylabel='Tip')
hex.fig.suptitle('Hexbin Joint plot of Total Bill and Tip',
                  fontsize=10, y=1.03)
```

Figure 3-18: Seaborn scatter plot using Implot



3.4.2.3 2D Density plot

You can also have a 2D kernel density plot. It is similar to how `sns.kdeplot` works, except it can plot a density plot across 2 variables. You can have the bivariate plot on its own ([Figure 3-21](#)), or have the two univariate plots alongside using `jointplot` ([Figure 3-22](#)).

```
kde, ax = plt.subplots()  
ax = sns.kdeplot(data=tips['total_bill'],  
                  data2=tips['tip'],  
                  shade=True) # shade will fill in the contours
```

Figure 3-19: Seaborn scatter plot using jointplot

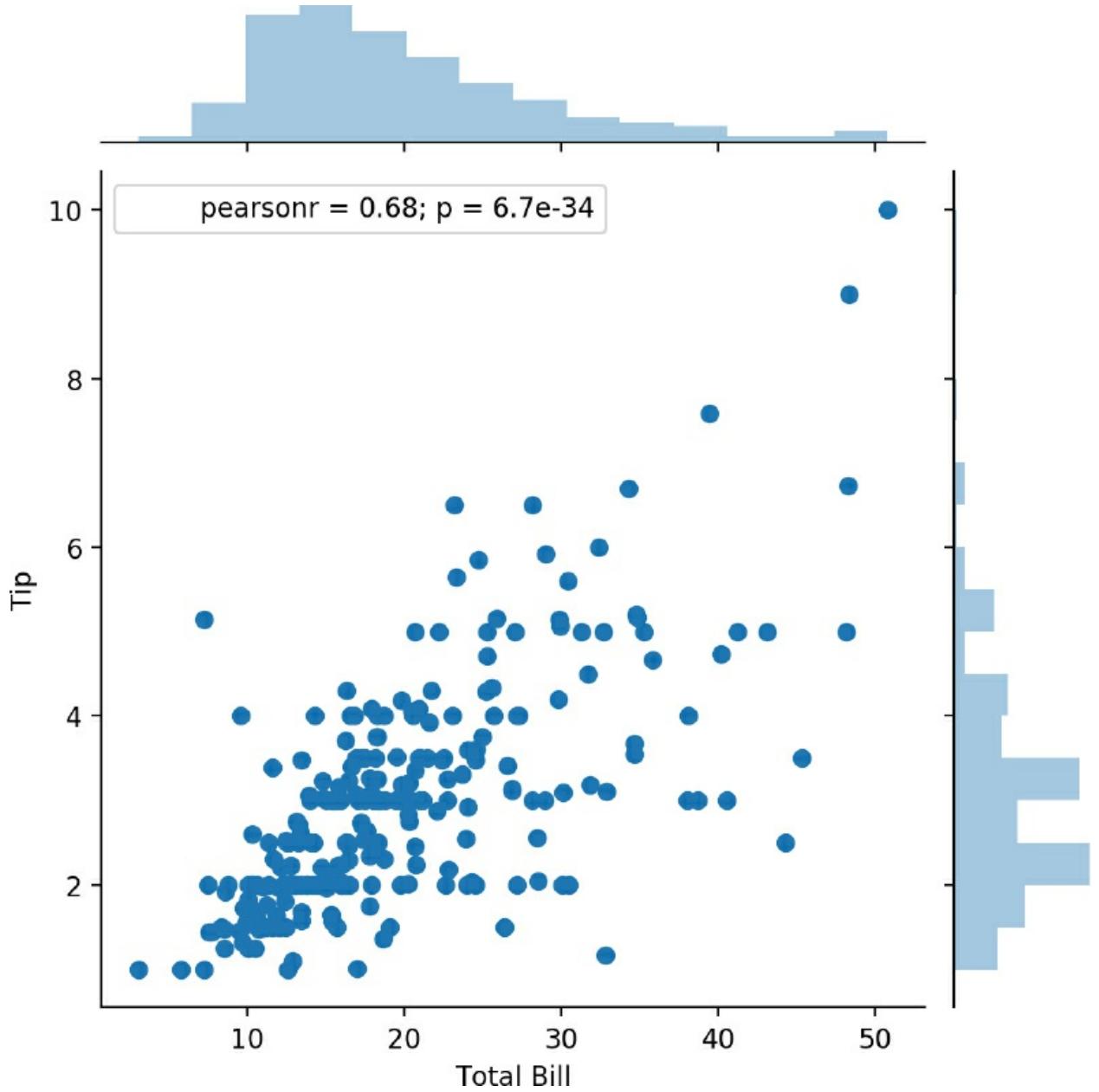
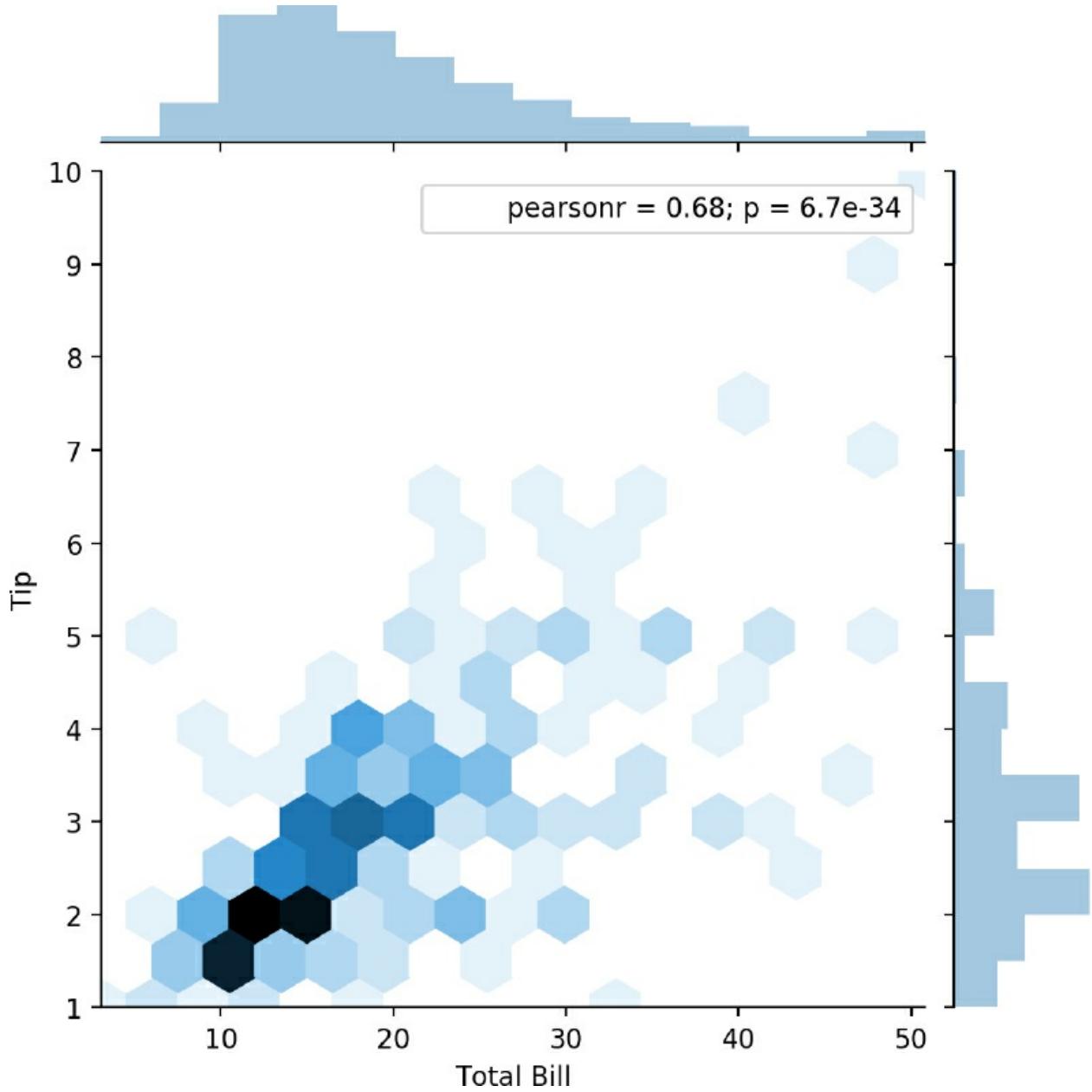


Figure 3-20: Seaborn hexbin plot using jointplot



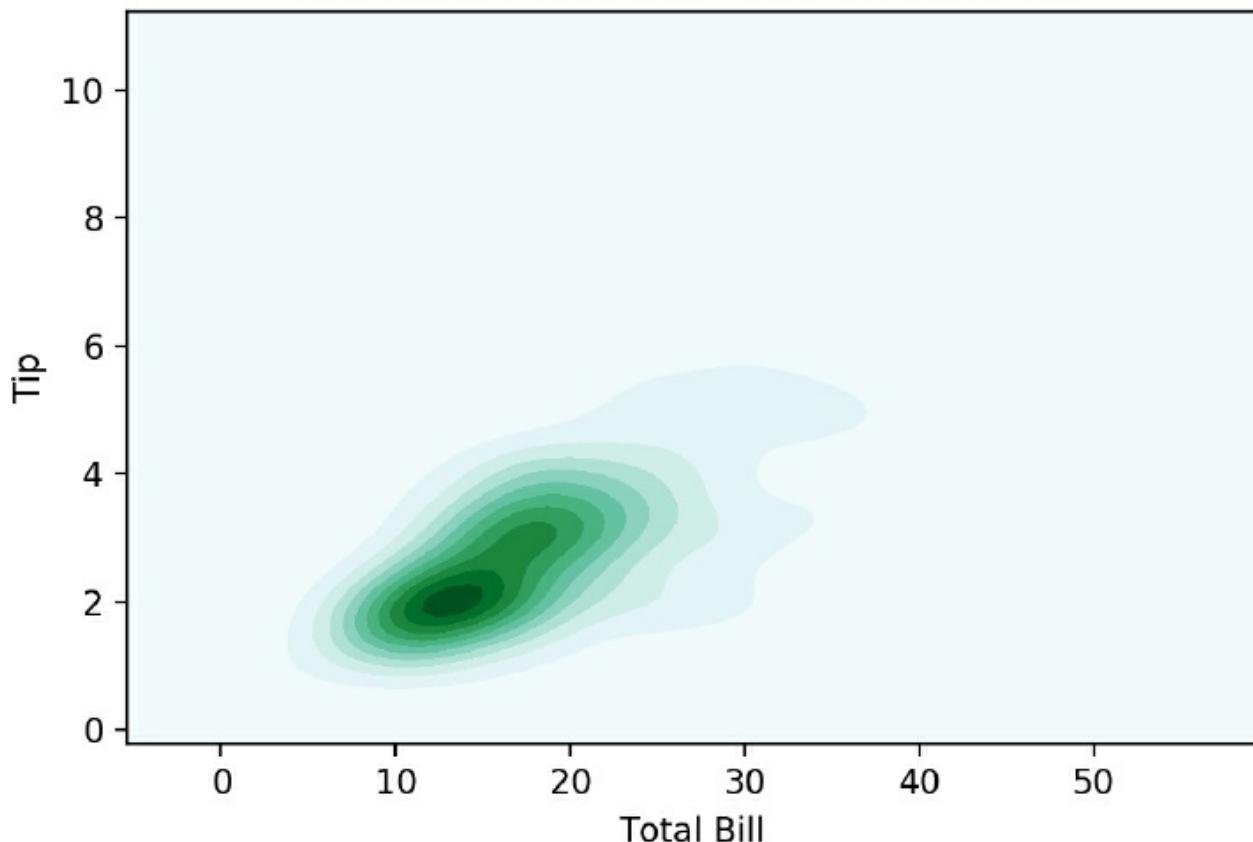
3.4.2.4 Bar plot

Bar plots can also be used to show multiple variables. By default, `barplot` will calculate a mean ([Figure 3-23](#)), but you can pass any function into the `estimator` parameter, for example, the `numpy.std` function to calculate the standard deviation.

```
bar, ax = plt.subplots()
ax = sns.barplot(x='time', y='total_bill', data=tips)
```

Figure 3-21: Seaborn KDE plot

Kernel Density Plot of Total Bill and Tip



```
ax.set_title('Barplot of average total bill for time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Average total bill')
plt.show()
```

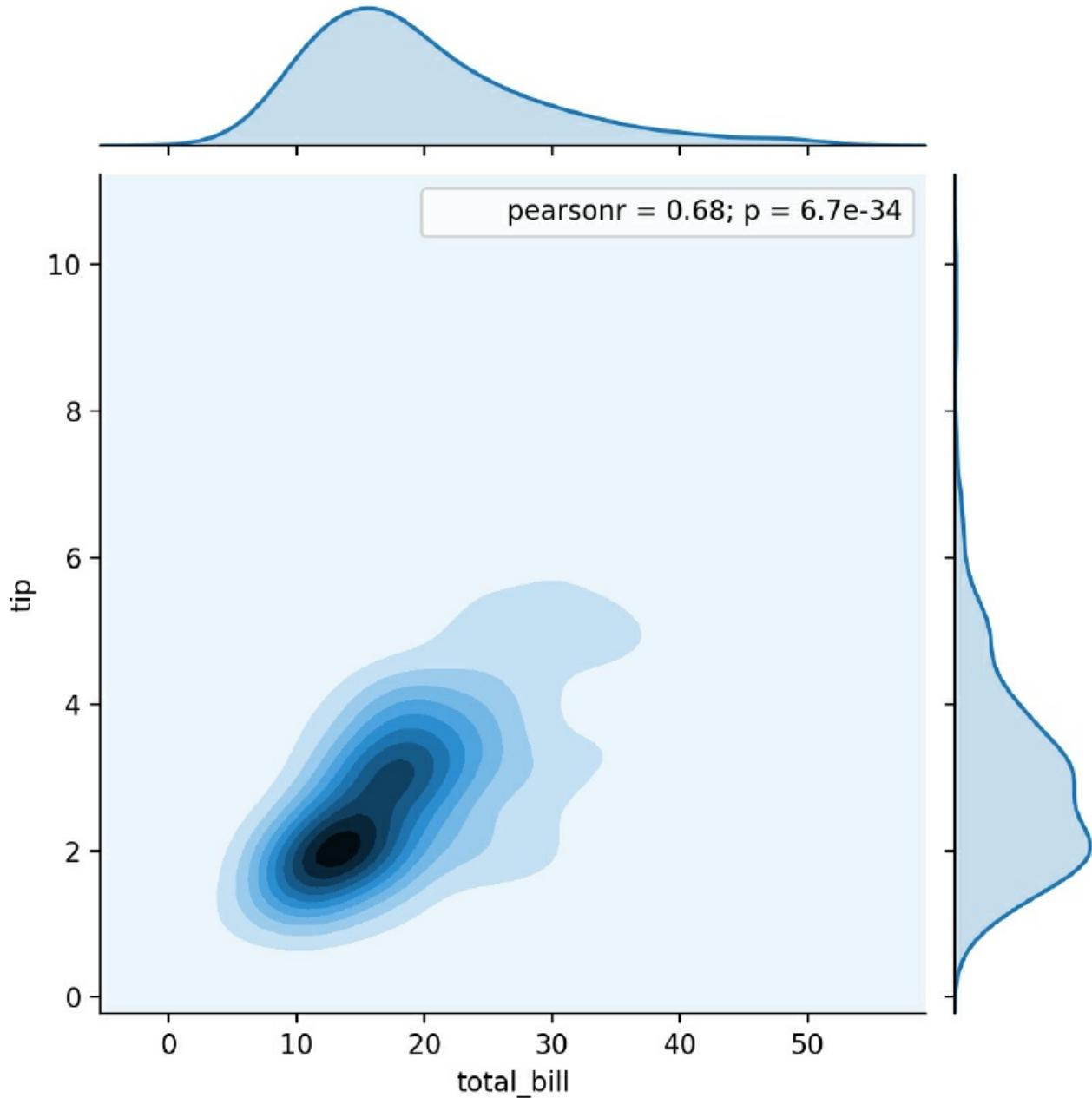
3.4.2.5 Box plot

Unlike previous plots, a box plot ([Figure 3-24](#)) shows multiple statistics: the minimum, first quartile, median, third quartile, maximum, and if applicable, outliers based on the interquartile range.

The y parameter is optional, meaning, if it is left out, it will create a single box in the plot.

```
box, ax = plt.subplots()
ax = sns.boxplot(x='time', y='total_bill', data=tips)
ax.set_title('Box plot of total bill by time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')
plt.show()
```

Figure 3-22: Seaborn KDE plot using jointplot

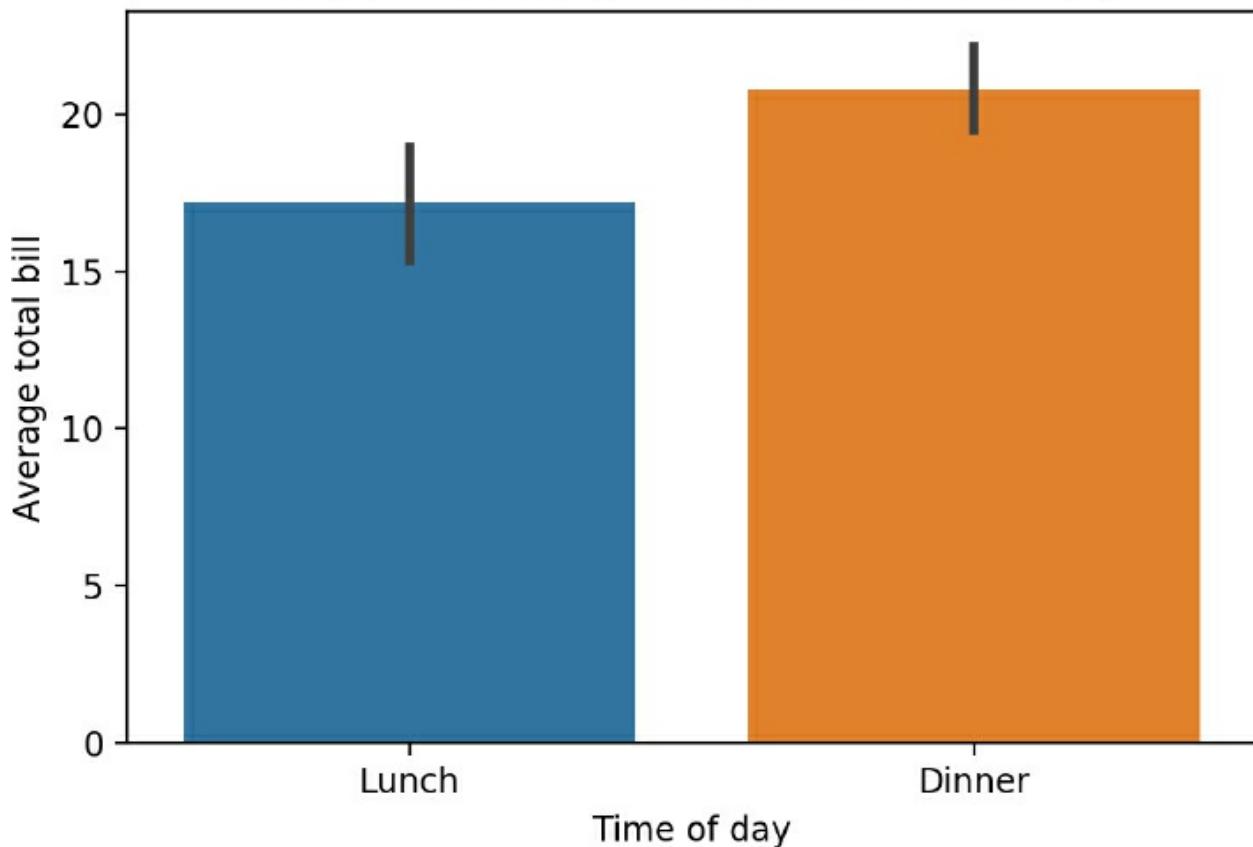


3.4.2.6 Violin plot

Box plots are a classical statistical visualization. However, they can obscure the underlying distribution of the data. Violin plots ([Figure 3-25](#)) are able to show the same values as the box plot, but plots the “boxes” as a kernel density estimation. This can help retain more visual information about your data since only plotting summary statistics can be misleading, as seen by the Anscombe’s quartets.

Figure 3-23: Seaborn barplot using the default mean calculation

Barplot of average total bill for time of day



```
violin, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill', data=tips)
ax.set_title('Violin plot of total bill by time of day')
ax.set_xlabel('Time of day')
ax.set_ylabel('Total Bill')
plt.show()
```

3.4.2.7 Pairwise relationships

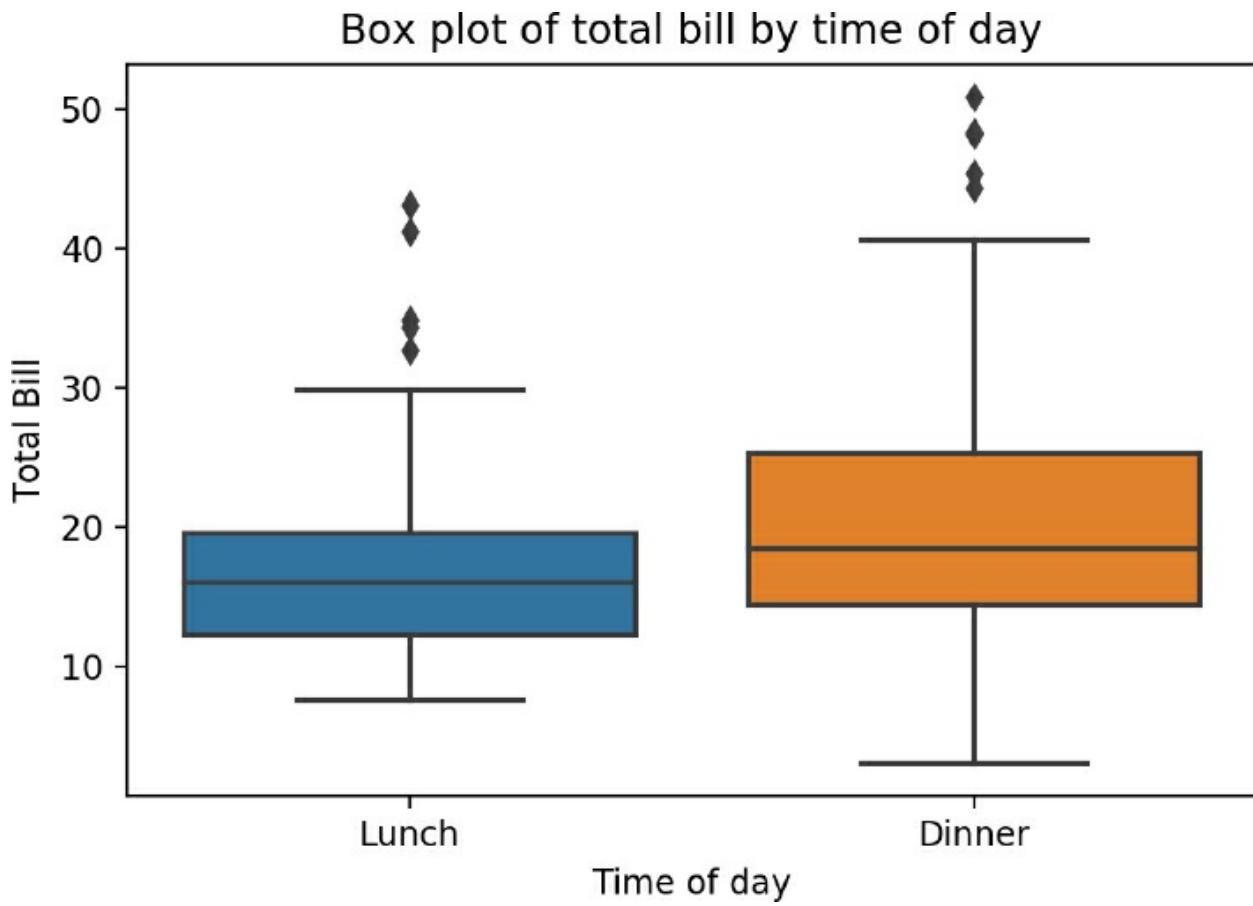
When you have mostly numeric data, visualizing all the pairwise relationships can be easily performed using `pairplot`. This will plot a scatter plot between each pair of variables, and a histogram for the univariate ([Figure 3-26](#)).

```
fig = sns.pairplot(tips)
```

One thing about `pairplot` is that there is redundant information. The top half of the visualization is the same as the bottom half. We can use `pairgrid` to manually assign the plots for the top half and bottom half. This plot is shown in Figure ??.

```
pair_grid = sns.PairGrid(tips)
# can also use plt.scatter instead of sns.regplot
```

Figure 3-24: Seaborn boxplot of total bill by time of day



```
i
i
i
3.4 seaborn 77
Figure 3{24}: Seaborn boxplot of total bill by time of day
pair_grid = pair_grid.map_upper(sns.regplot)
pair_grid = pair_grid.map_lower(sns.kdeplot)
pair_grid = pair_grid.map_diag(sns.distplot, rug=True)
plt.show()
```

3.4.3 multivariate

I mentioned in [Section 3.3.3](#), that there is no de facto template for plotting multivariate data.

Possible ways to include more information is to use color, size, and shape to add more information to a plot

3.4.3.1 Colors

In a `violinplot`, we can pass the `hue` parameter to color the plot by `sex`. We can reduce the redundant information by having each half of the violins represent the different `sex`, which is shown in [Figure 3-28](#). Try the following code with and without the `split` parameter.

```
violin, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
hue='sex', data=tips,
```

Figure 3-25: Seaborn violin of total bill by time of day



```
plt.show()
    split=True)
```

The `hue` parameter can be passed into various other plotting functions as well. [Figure 3-29](#) shows it in a `lmplot`

```
# note I'm using lmplot instead of regplot here
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      hue='sex', fit_reg=False)
plt.show()
```

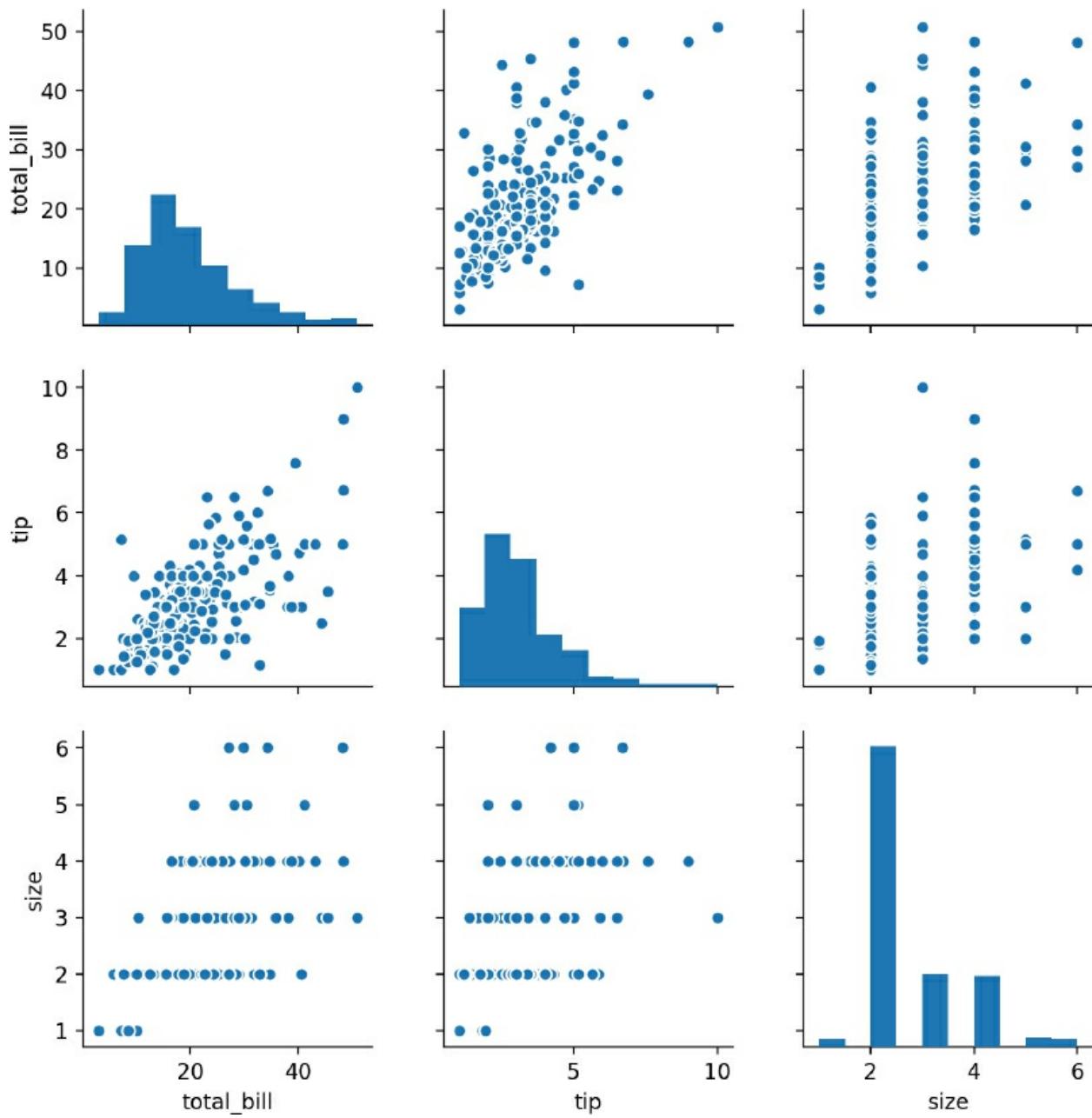
We can make our pairwise plots a little more meaningful by passing one of the categorical variables as a `hue` parameter. [Figure 3-30](#) shows this in our `pairplot`

```
fig = sns.pairplot(tips, hue='sex')
```

3.4.3.2 Size and Shape

Working with point sizes can also be another means to add more information to a plot. However, this should be used sparingly, since the human eye is not very good at comparing areas.

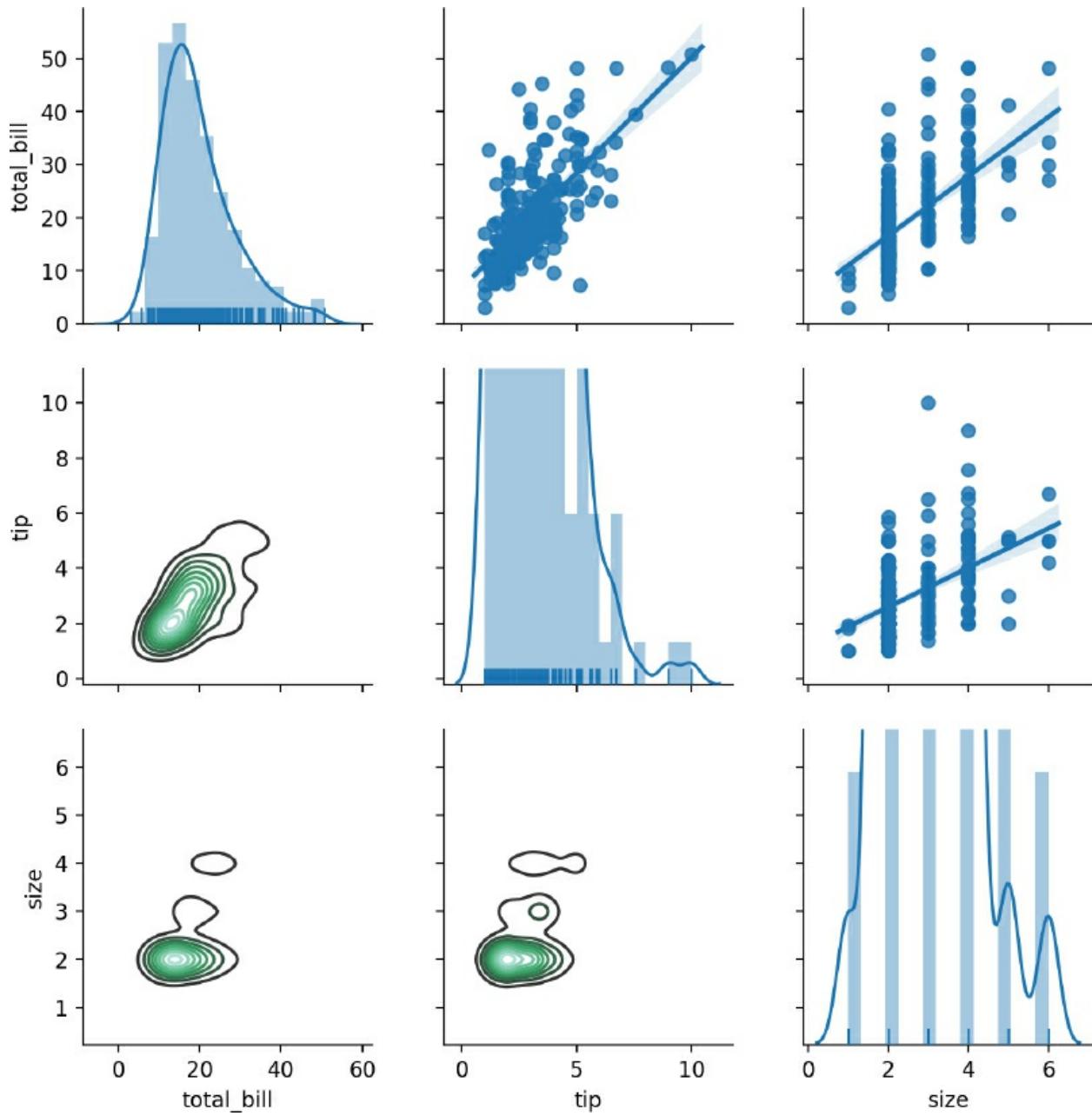
Figure 3-26: Seaborn pairplot



Here, is an example of how `seaborn` works with `matplotlib` function calls. If you look in the documentation for `lmplot`⁶, you'll see that `lmplot` takes a parameter called `catter`, `linescatter`, `line_kws`. This is actually them saying there is a parameter in `lmplot` called `scatter_kws` and `line_kws`. Both of these parameters take a key-value pair, a Python `dict` (dictionary) to be more exact ([Appendix K](#)). Key-value pairs passed into `scatter_kws` is then passed on to the `matplotlib` function `plt.scatter`. This is how we would access the `s` parameter to change the size of the

⁶<https://web.Stanford.edu/~mwaskom/software/seaborn/generated/seaborn.lmplot.html>

Figure 3-27: Seaborn pairplot with different plots on the upper and lower half.

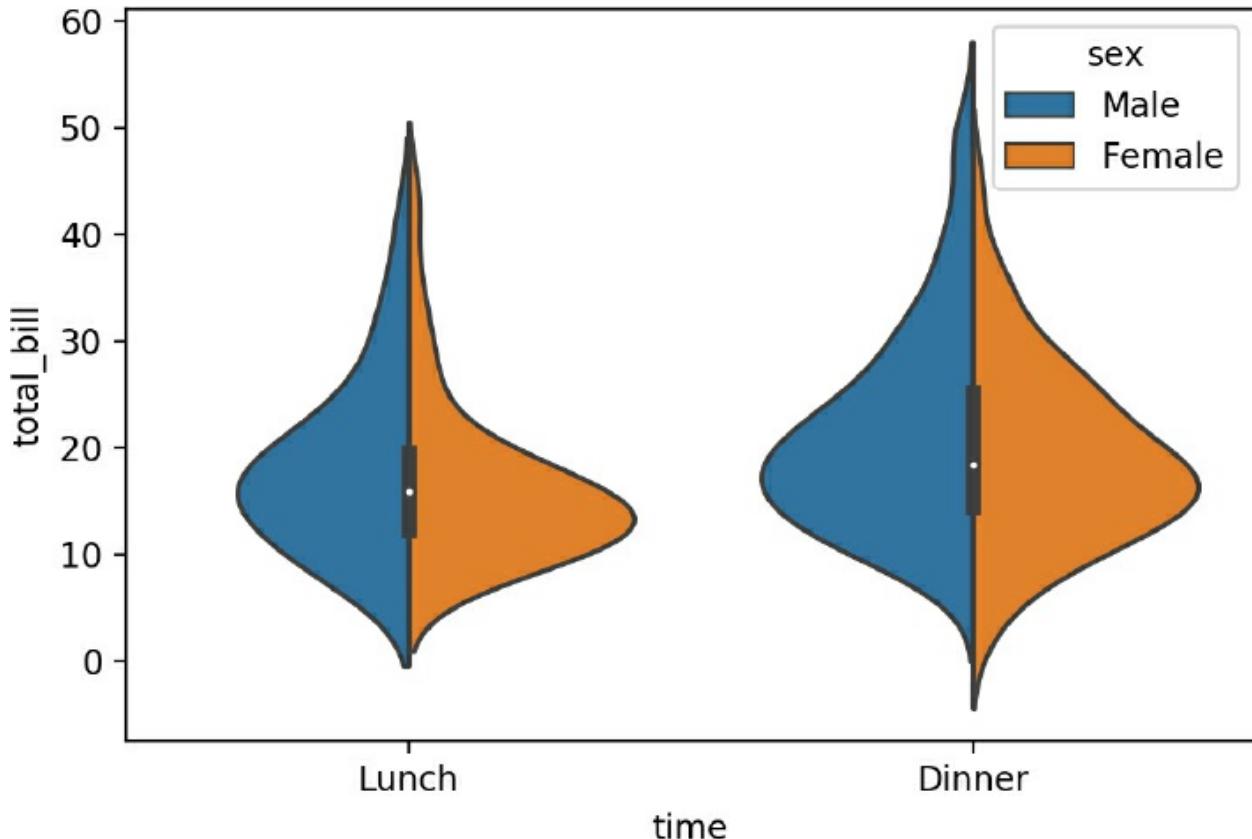


points like we did in [Section 3.3.3](#). This is shown in [Figure 3-31](#).

```
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      fit_reg=False,
                      hue='sex',
                      scatter_kws={'s': tips['size']*10})
plt.show()
```

Also, when working with multiple variables, sometimes having 2 plot elements showing the same information is helpful. [Figure 3-32](#) shows color and shape to distinguish `sex`.

Figure 3-28: Seaborn violin plot with hue parameter



```
scatter = sns.lmplot(x='total_bill', y='tip', data=tips,
                      fit_reg=False, hue='sex', markers=['o', 'x'],
                      scatter_kws={'s': tips['size']*10})
plt.show()
```

3.4.3.3 facets

What if we want to show more variables? Or if we know what plot we want for our visualization, but we want to make multiple plots over a categorical variable? This is what facets are for. Instead of individually subsetting data and laying out the axes in a figure (we did this in [Figure 3-5](#)), facets in `seaborn` handle this for you.

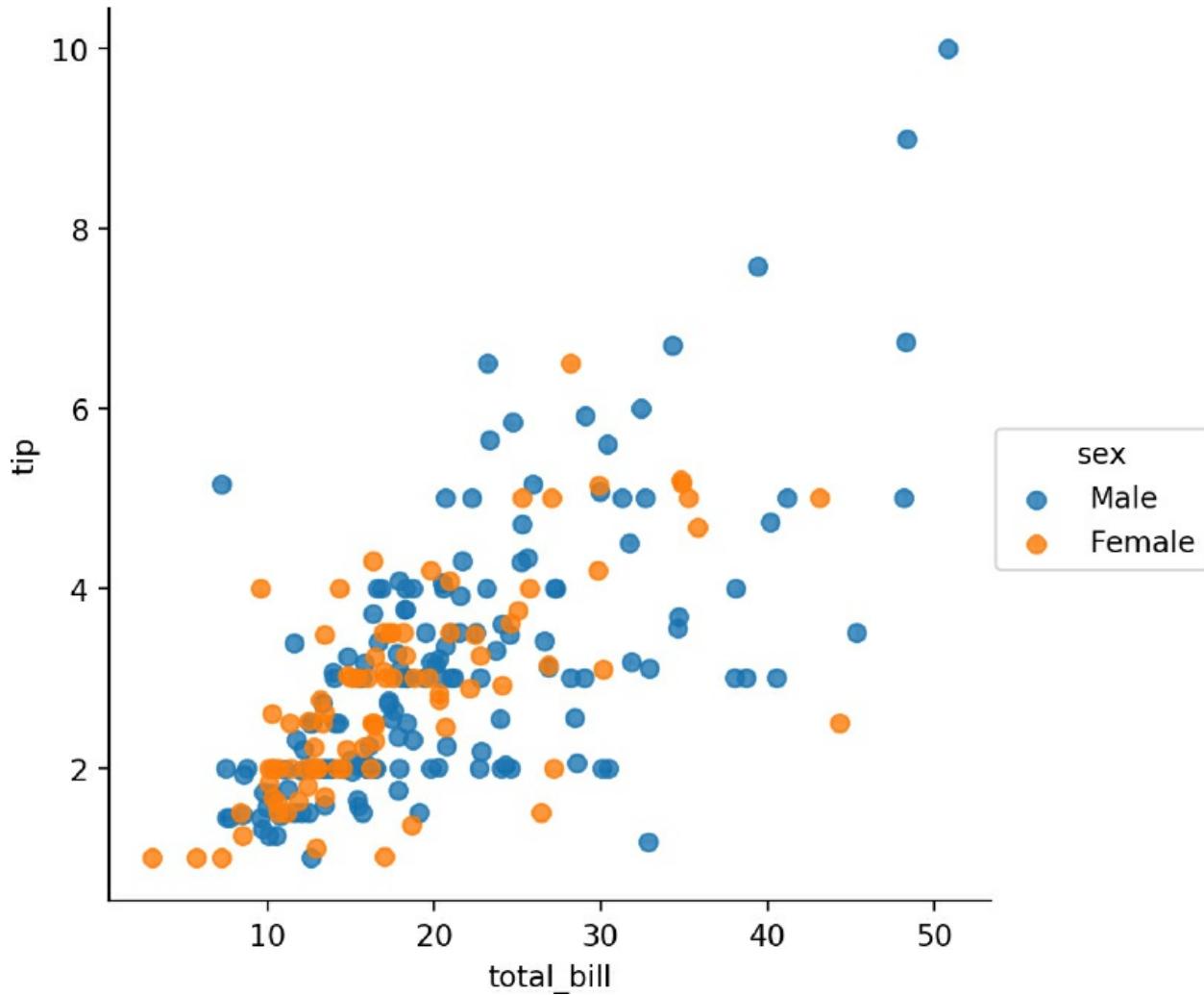
In order to use facets your data needs to be what Hadley Wickham⁷ calls “Tidy Data”⁸, where each row represents an observation in your data, and each column is a variable (it is also known as “long data”).

⁷<http://hadley.nz/>

⁸<http://vita.had.co.nz/papers/tidy-data.pdf>

[Figure 3-33](#) shows a recreation our Anscombe’s quartet from [Figure 3-5](#) in `seaborn`.

Figure 3-29: Seaborn lmplot plot with hue parameter



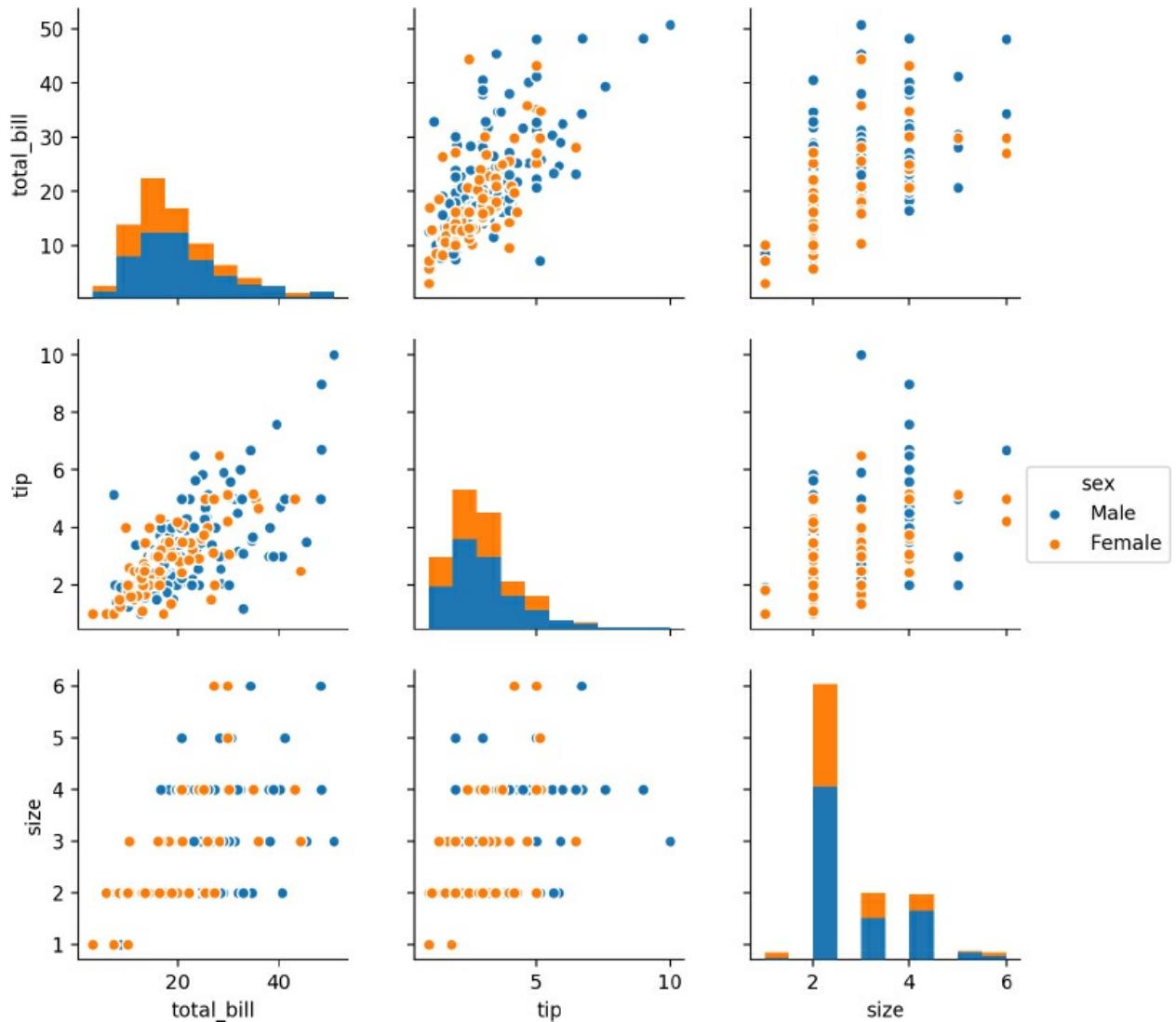
```
anscombe_plot = sns.lmplot(x='x', y='y', data=anscombe,
                           fit_reg=False,
                           col='dataset', col_wrap=2)
```

All we needed to do is pass 2 more parameters into the scatter plot function in `seaborn`. The `col` parameter is the variable the plot will facet by, and the `colwrap` creates a figure that has 2 columns. If we do not use the `colwrap` parameter, all 4 plots will be plotted in the same row.

[Section 3.4.2.1](#) discussed the differences between `lmplot` and `regplot`. `lmplot` is a figure level function. Many of the plots we created in `seaborn` are `axes` level functions. What this means is not every plotting function will have a `col` and `colwrap` parameter for faceting. Instead we have to create a `FacetGrid` that knows what variable to facet on, and then supply the individual plot code for each facet. [Figure 3-34](#) shows our manually created facet plot.

```
# create the FacetGrid
```

Figure 3-30: Seaborn pairplot plot with hue parameter



```

facet = sns.FacetGrid(tips, col='time')
# for each value in time, plot a histogram of total bill
facet.map(sns.distplot, 'total_bill', rug=True)
plt.show()

```

The individual facets need not be univariate plots as seen in [Figure 3-35](#)

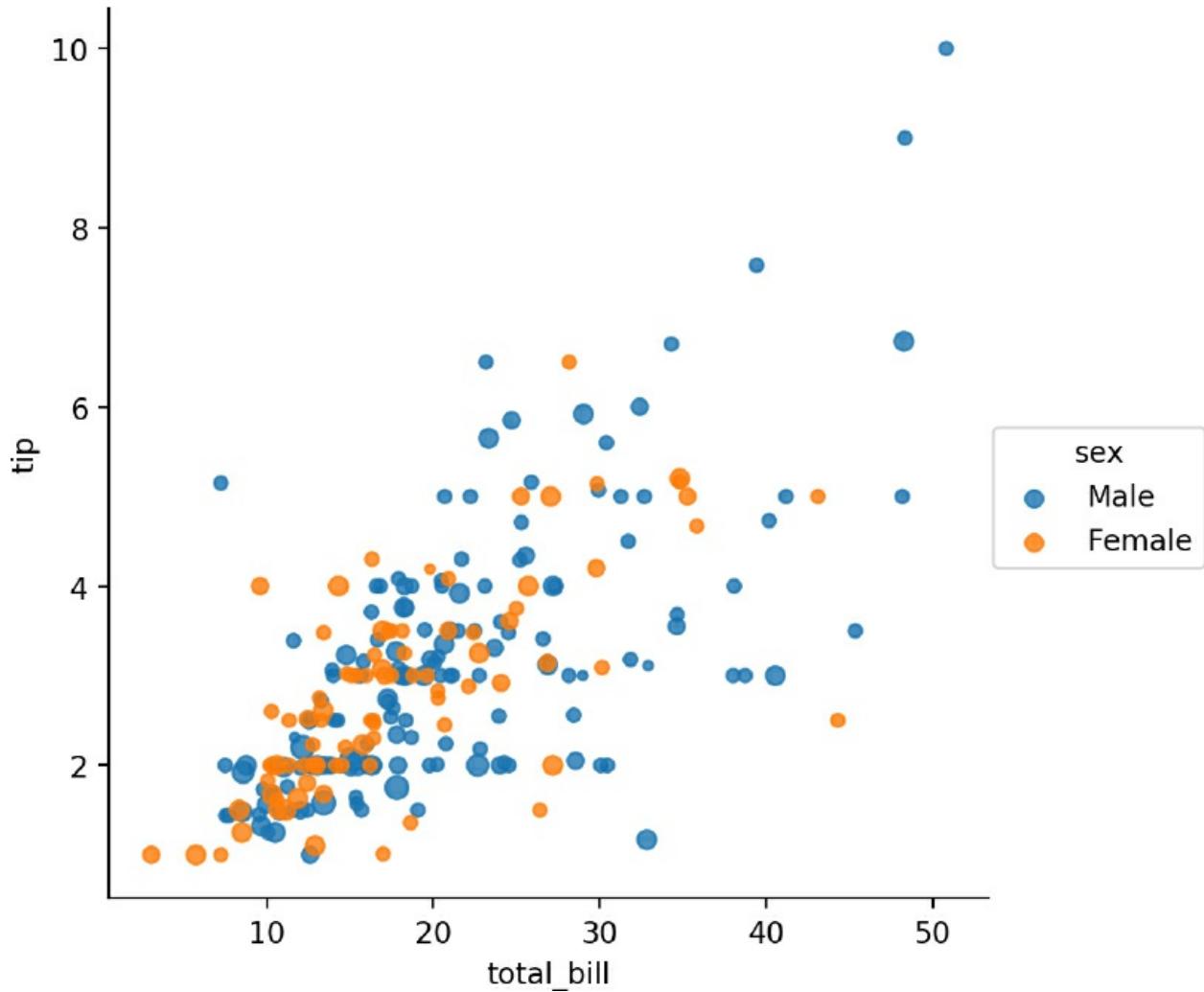
```

facet = sns.FacetGrid(tips, col='day', hue='sex')
facet = facet.map(plt.scatter, 'total_bill', 'tip')
facet = facet.add_legend()
plt.show()

```

If you wanted to stay in `seaborn` you can do the same plot using `lmplot`, as shown in [Figure 3-36](#)

Figure 3-31: Seaborn scatter plot passing scatter_kws



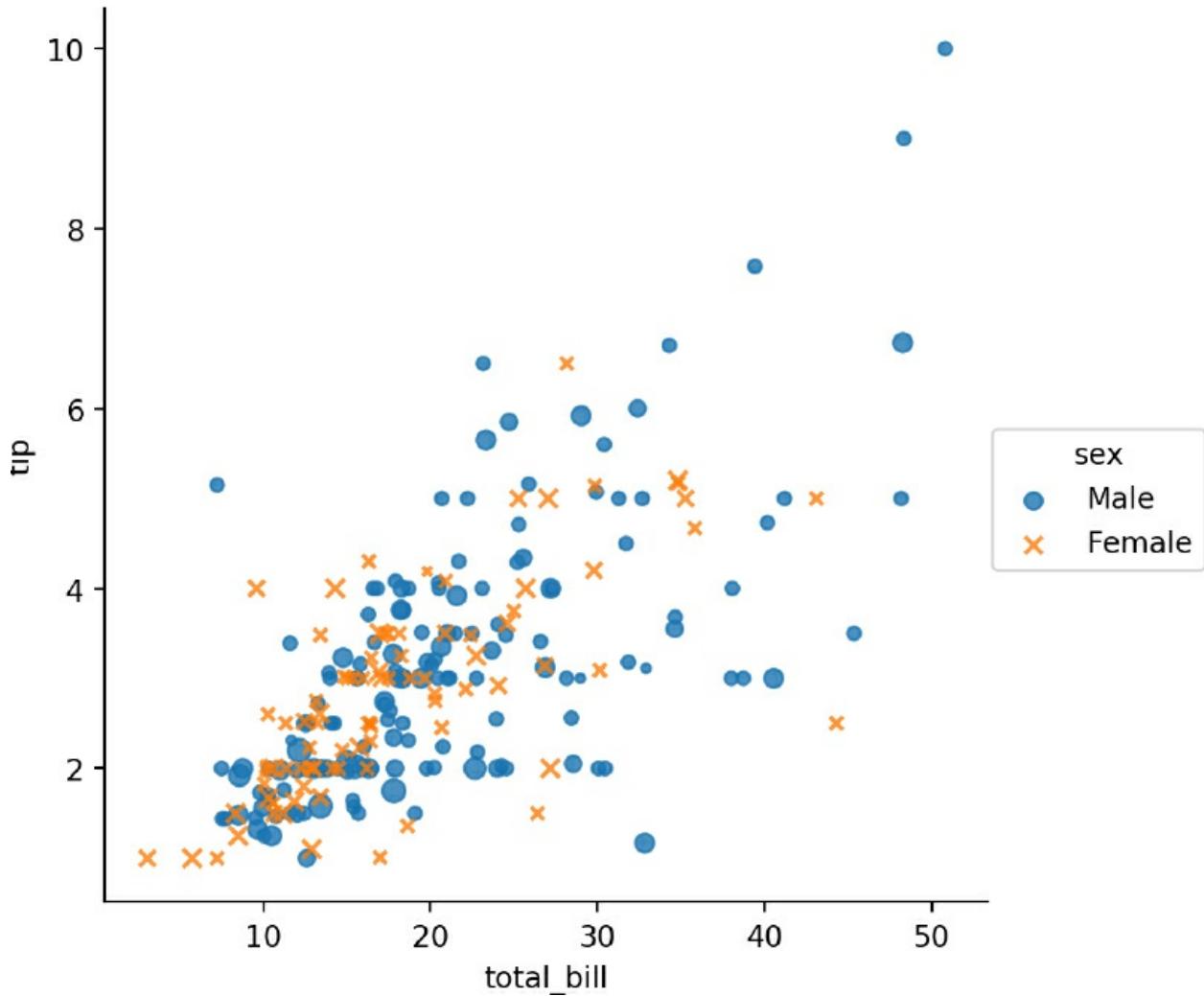
```
fig = sns.lmplot(x='total_bill', y='tip', data=tips, fit_reg=False,
                  hue='sex', col='day')
plt.show()
```

The last thing you can do with facets is to have one variable be faceted on the x axis, and another variable faceted on the y axis. We accomplish this by passing a `row` parameter. This is shown in [Figure 3-37](#)

```
facet = sns.FacetGrid(tips, col='time', row='smoker', hue='sex')
facet.map(plt.scatter, 'total_bill', 'tip')
plt.show()
```

If you do not want all the `hue` elements overlapping each other (i.e., you want this behavior in scatter plots, but not violin plots), you can use the `sns. factorplot` function. This is shown in [Figure 3-38](#).

Figure 3-32: Seaborn scatter plot with markers passing scatter_kws

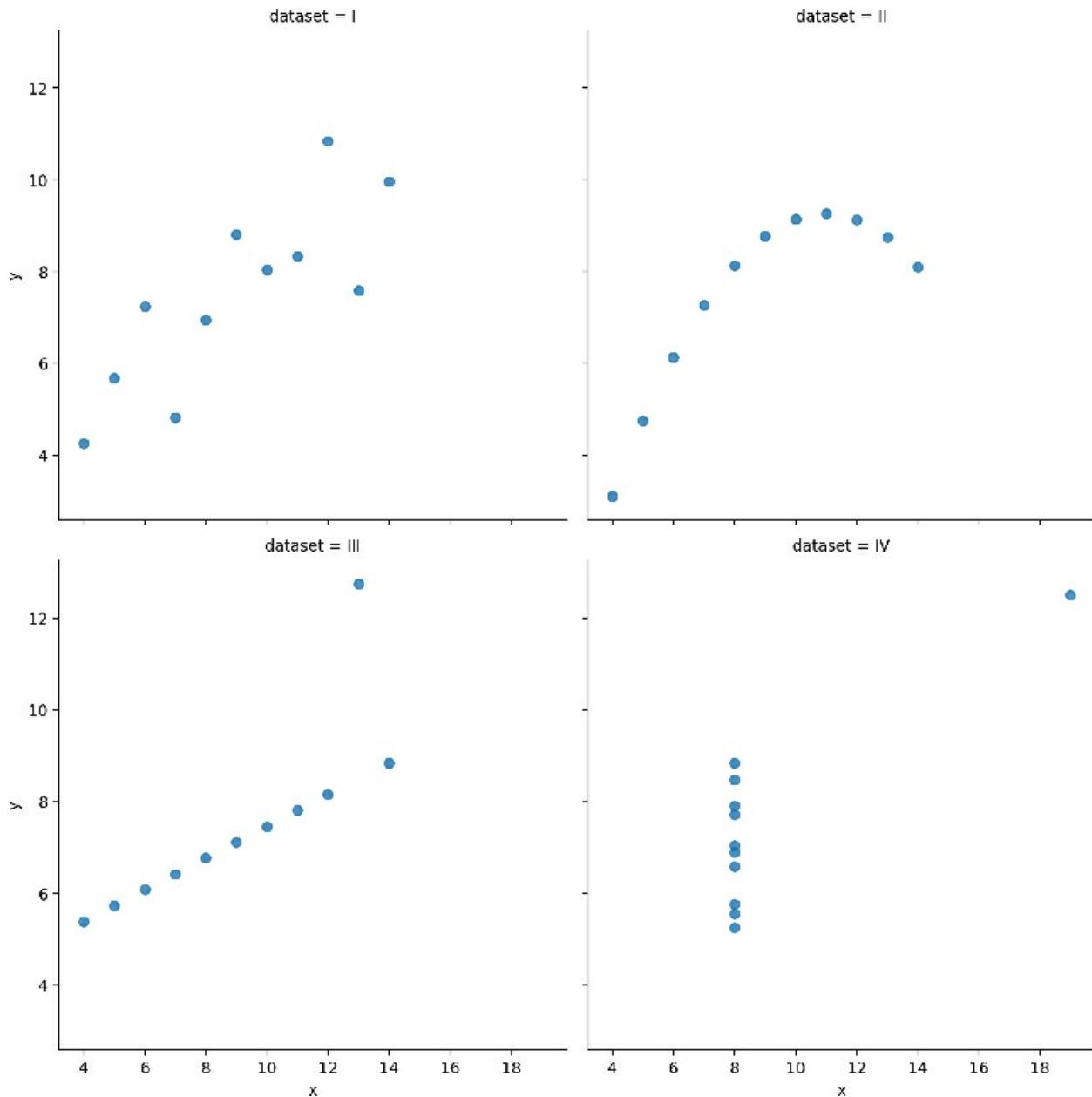


3.5 pandas

`pandas` objects also come equipped with their own plotting functions. Just like `seaborn`, the plotting functions built into `pandas` are just wrappers around `matplotlib` with presets.

In general, plotting using `pandas` follows the `DataFrame.plot.PLOT_TYPE` or `Series . plot. PLOT_TYPE` functions.

Figure 3-33: Seaborn anscombe plot with facets



3.5.1 Histograms

Histograms can be created using the `series.plot.hist` ([Figure 3-39](#)) or `DataFrame.plot.hist` ([Figure 3-40](#)) function.

```
# on a series
fig, ax = plt.subplots()
ax = tips['total_bill'].plot.hist()
plt.show()
```

Figure 3-34: Seaborn plot with manually created facets

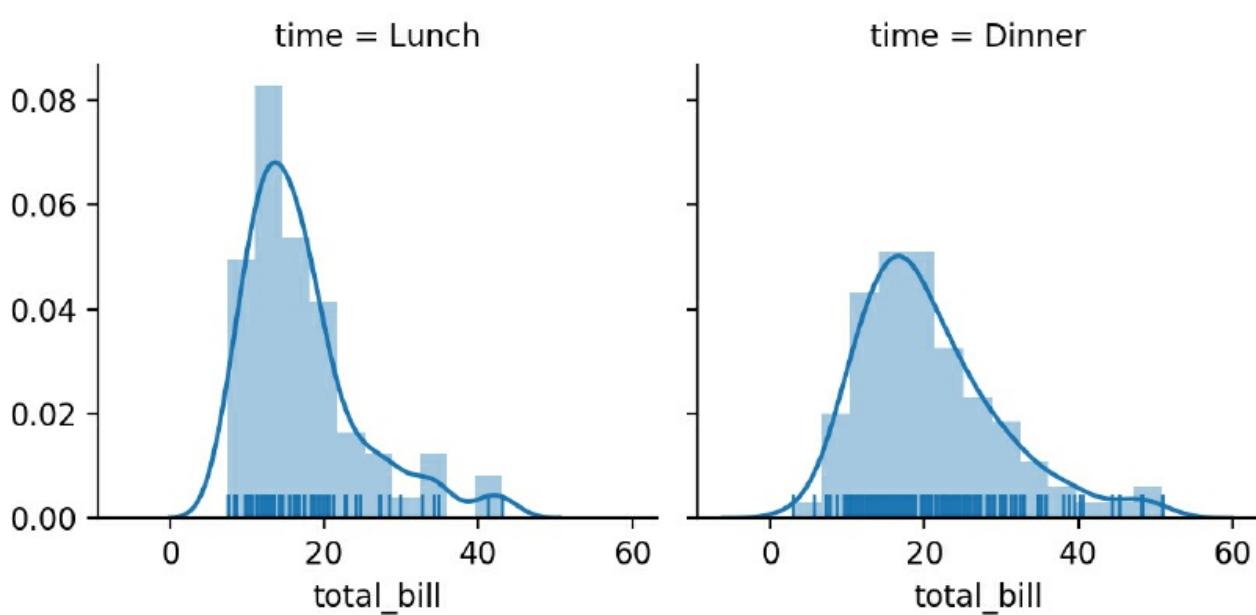


Figure 3-35: Seaborn plot with manually created facets that contain multiple variables

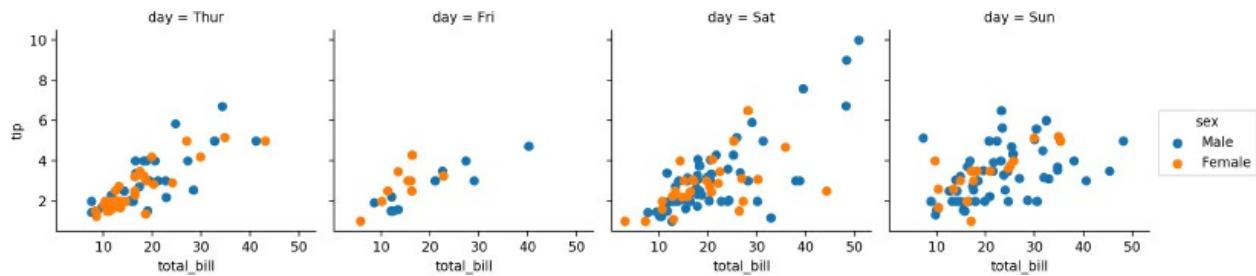
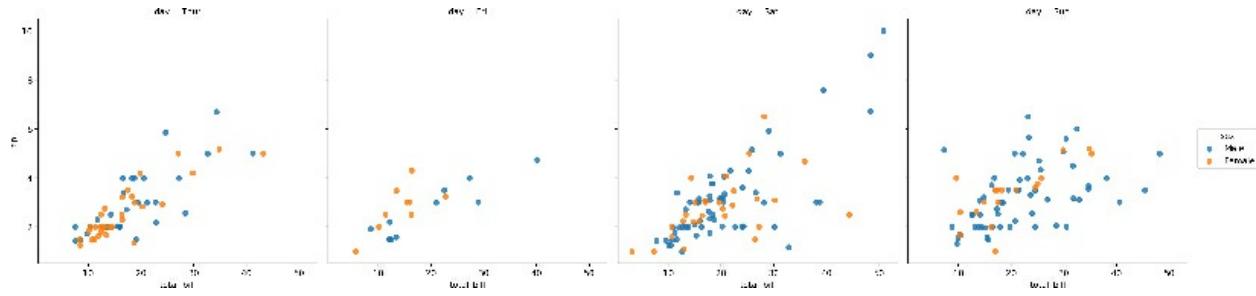
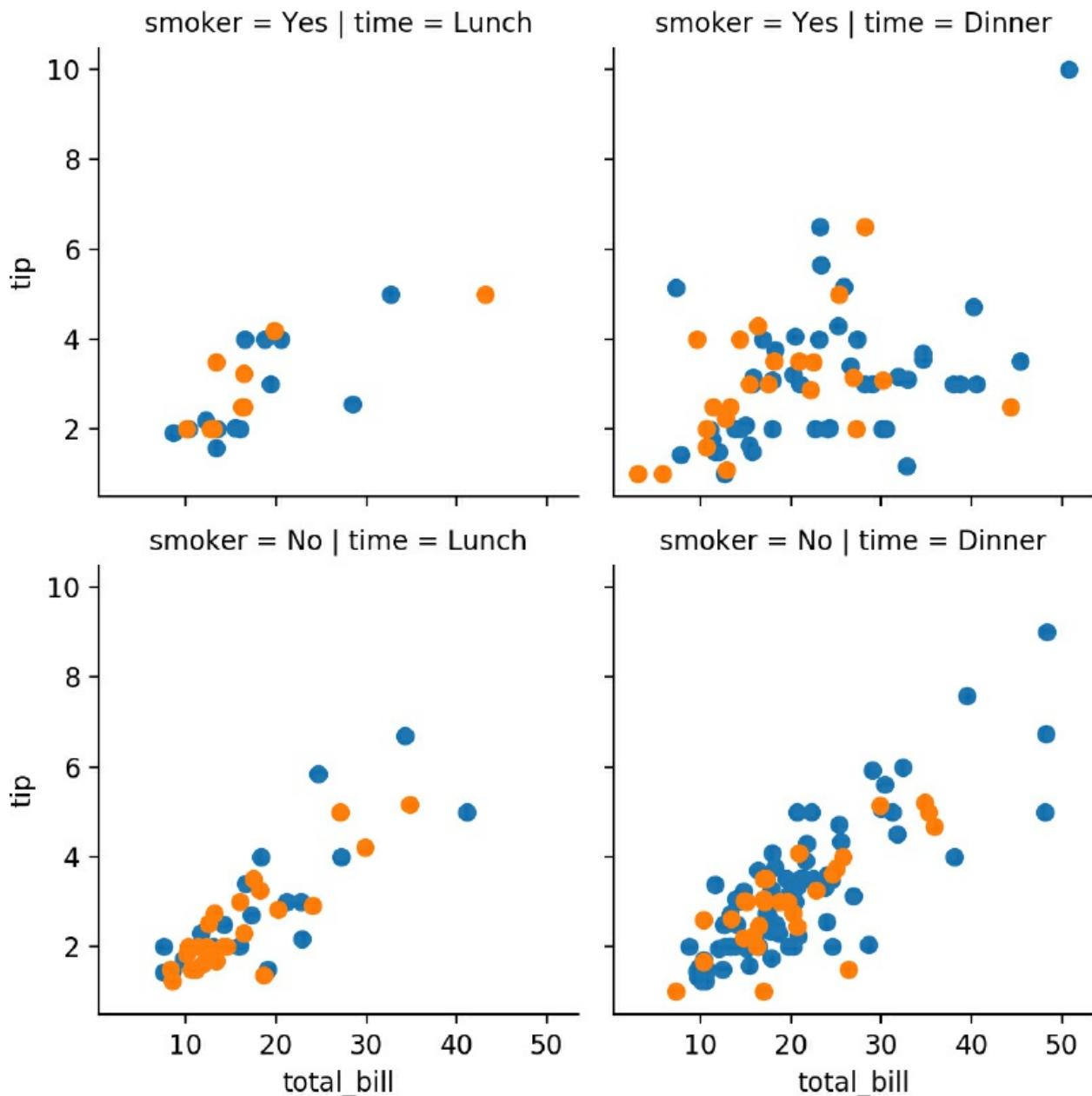


Figure 3-36: Seaborn plot with manually created facets that contain multiple variables



```
# on a dataframe
# set an alpha channel transparency
# so we can see through the overlapping bars
fig, ax = plt.subplots()
ax = tips[['total_bill', 'tip']].plot.hist(alpha=0.5, bins=20, ax=ax)
```

Figure 3-37: Seaborn plot with manually created facets with 2 variables



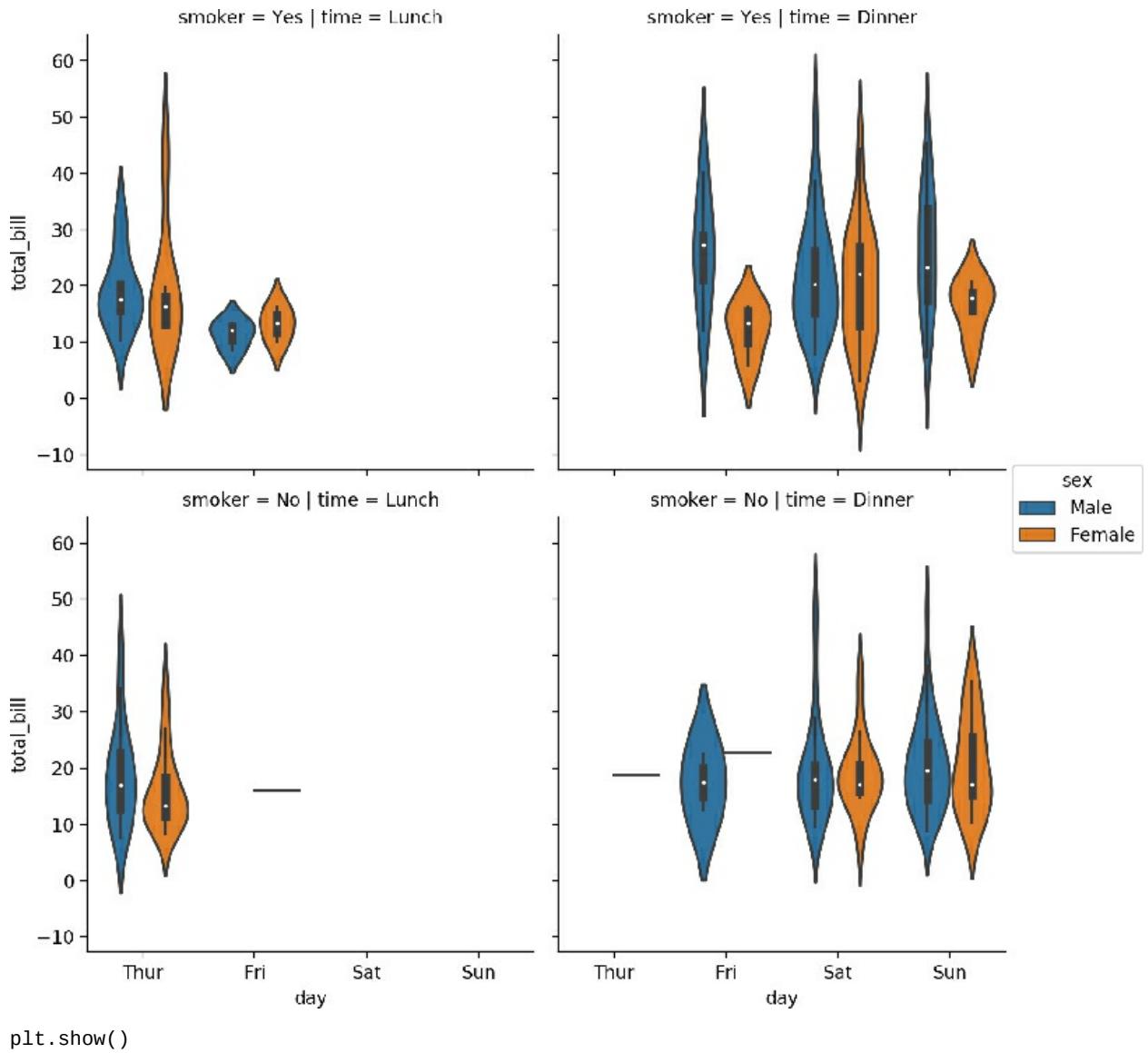
```
plt.show()
```

3.5.2 Density Plot

The kernel density estimation (density) plot can be created with the `DataFrame.plot.kde` function ([Figure 3-41](#)).

```
fig, ax = plt.subplots()
ax = tips['tip'].plot.kde()
```

Figure 3-38: Seaborn plot with manually created facets with 2 variables



```
plt.show()
```

3.5.3 Scatter Plot

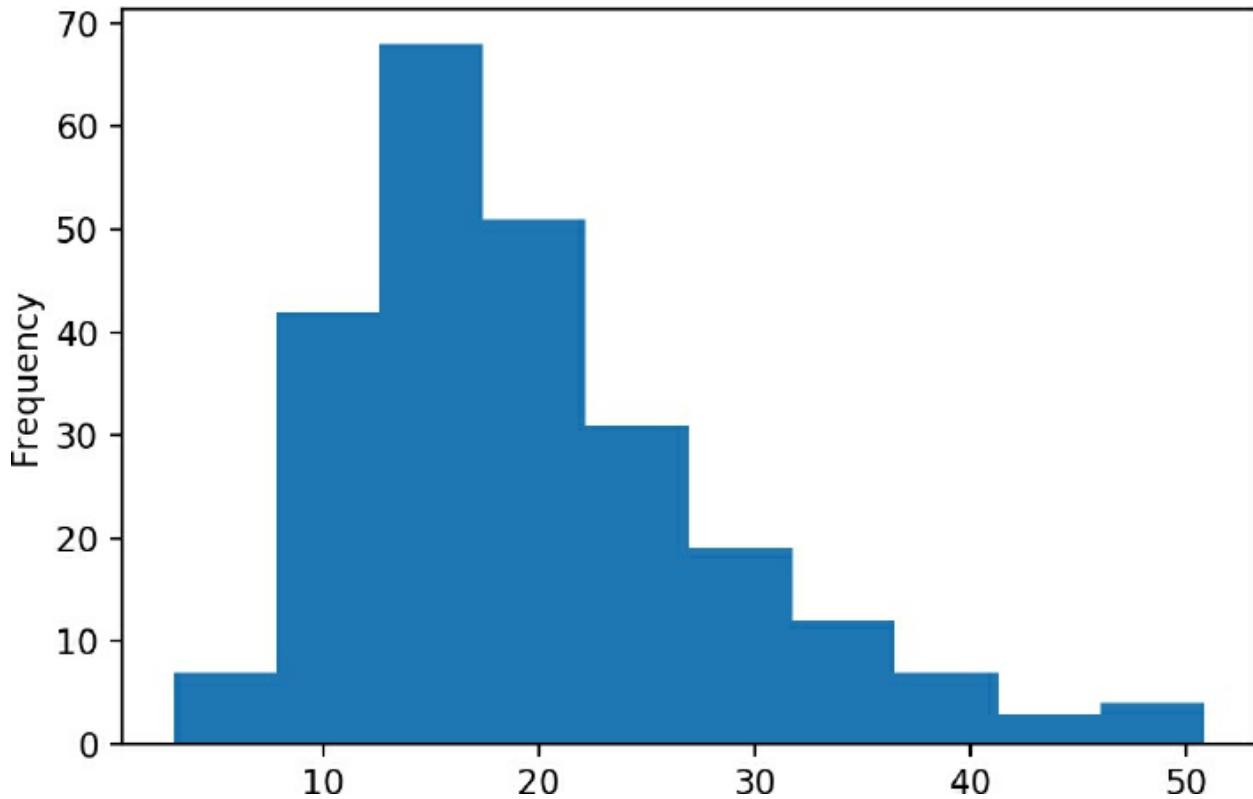
Scatter plots are created by using the `DataFrame.plot.scatter` function ([Figure 3-42](#)).

```
fig, ax = plt.subplots()
ax = tips.plot.scatter(x='total_bill', y='tip', ax=ax)
plt.show()
```

3.5.4 Hexbin Plot

Hexbin plots are created using the `Dataframe=plt.hexbin` function ([Figure 3-43](#)).

Figure 3-39: Histogram of a pandas Series



```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(x='total_bill', y='tip', ax=ax)
plt.show()
```

Gridsize can be adjusted with the `gridsize` parameter ([Figure 3-44](#))

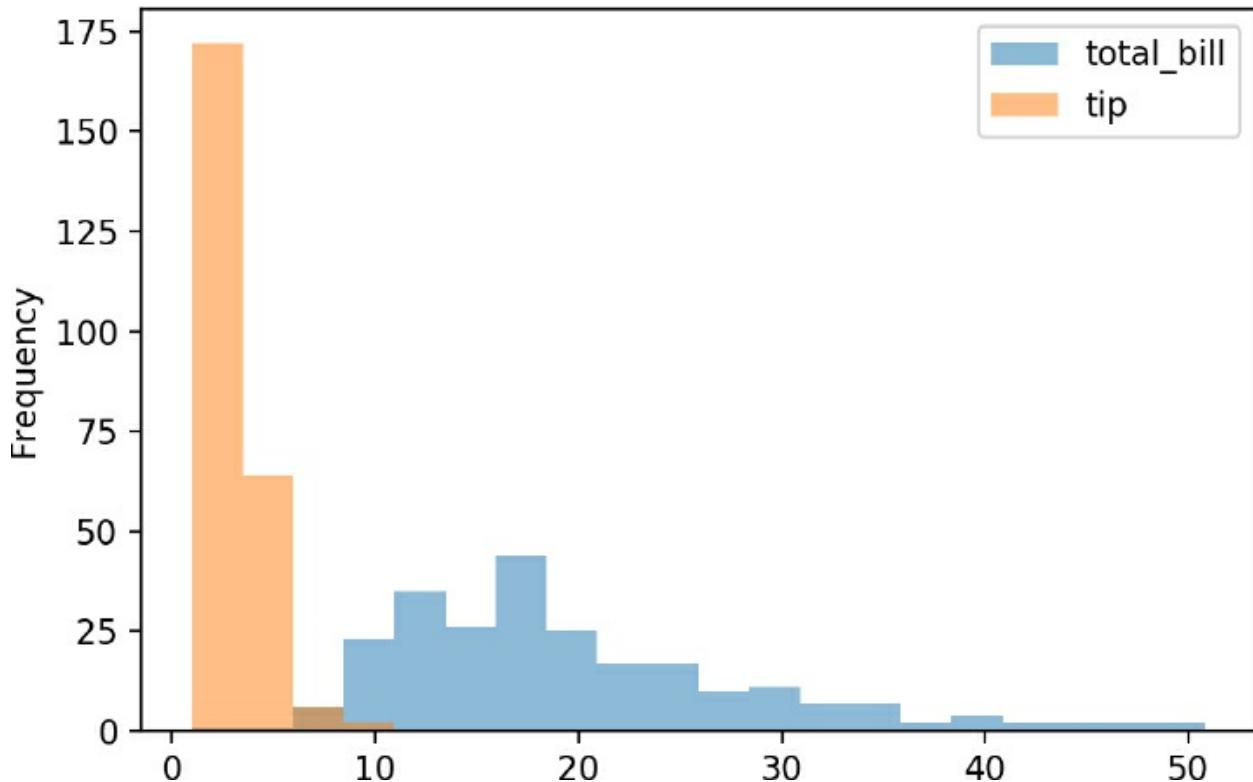
```
fig, ax = plt.subplots()
ax = tips.plot.hexbin(x='total_bill', y='tip', gridsize=10, ax=ax)
plt.show()
```

3.5.5 Box Plot

Box plots are created with the `DataFrame.plot.box` function ([Figure 3-45](#)).

```
fig, ax = plt.subplots()
ax = tips.plot.box(ax=ax)
plt.show()
```

Figure 3-40: Histogram of a pandas DataFrame



3.6 Seaborn Themes and Styles

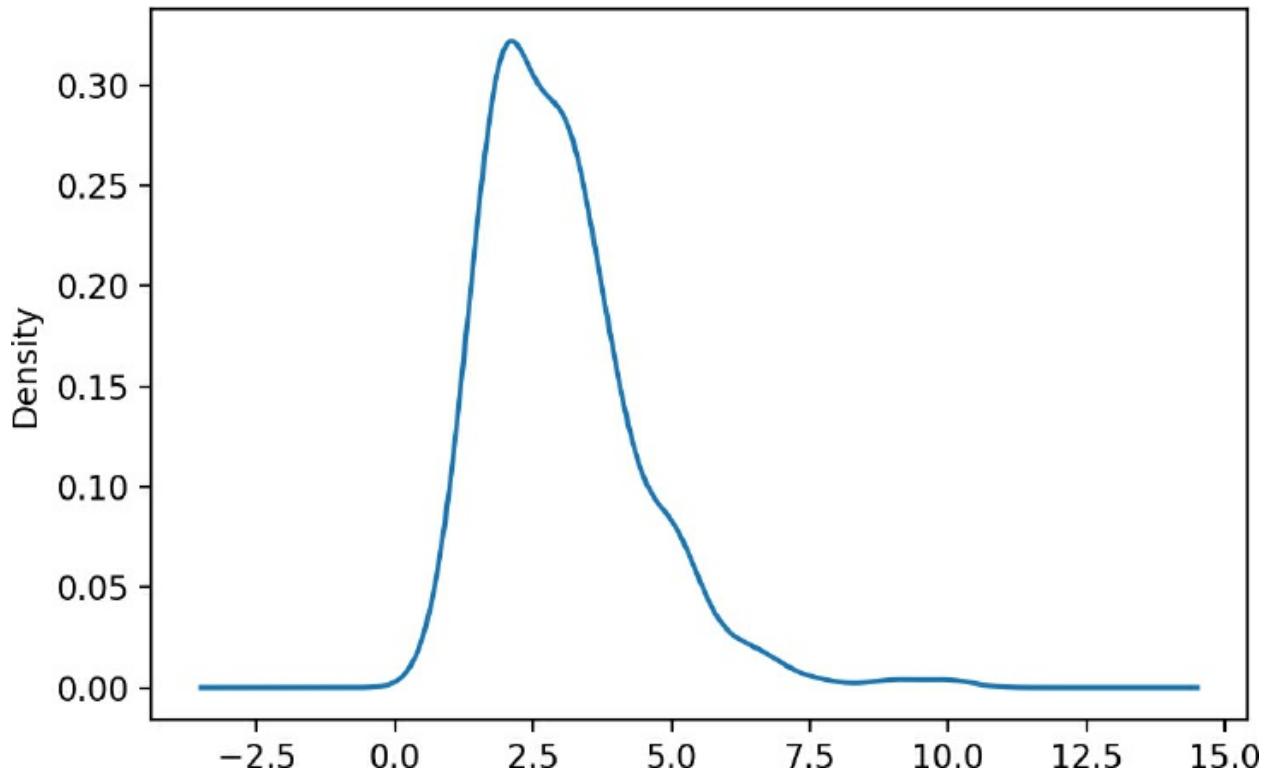
The `seaborn` plots shown in this chapter have all used the default plot styles. We can change the plot style with the `sns.set_style` function. Typically, this function is run just once at the top of your code; all subsequent plots will use the style set.

The styles that come with `seaborn` are `darkgrid`, `whitegrid`, `dark`, `white`, and `ticks`. [Figure 3-46](#) shows a base plot and [Figure 3-47](#) shows a plot with the `whitegrid` style.

```
# initial plot for comparison
fig, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()

# set style and plot
sns.set_style('whitegrid')
fig, ax = plt.subplots()
ax = sns.violinplot(x='time', y='total_bill',
                     hue='sex', data=tips,
                     split=True)
plt.show()
```

Figure 3-41: Pandas KDE plot



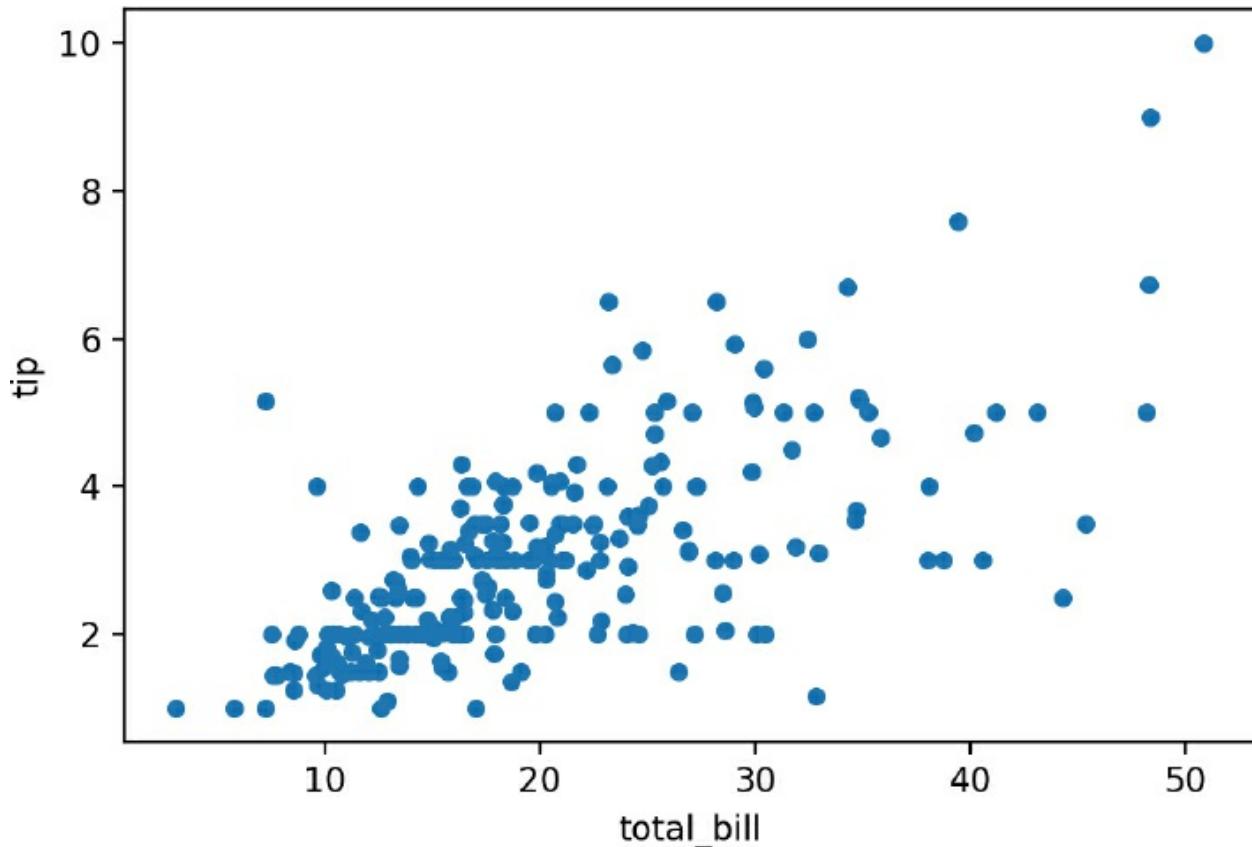
The following code shows what all the styles look like ([Figure 3-48](#)).

```
fig = plt.figure()
seaborn_styles = ['darkgrid', 'whitegrid', 'dark', 'white', 'ticks']
for idx, style in enumerate(seaborn_styles):
    plot_position = idx + 1
    with sns.axes_style(style):
        ax = fig.add_subplot(2, 3, plot_position)
        violin = sns.violinplot(x='time', y='total_bill',
                               data=tips, ax=ax)
        violin.set_title(style)
fig.tight_layout()
plt.show()
```

3.7 Conclusion

Data visualization is an integral part of exploratory data analysis and data presentation. This chapter gives an introduction to start exploring and presenting your data. As we continue through the book, we will learn about more complex visualizations.

Figure 3-42: Pandas scatter plot



There are a myriad of plotting and visualization resources on the internet. The `seaborn` documentation⁹, `pandas` visualization documentation¹⁰, and `matplotlib` documentation¹¹ will all provide ways to further tweak your plots (e.g., colors, line thickness, legend placement, figure annotations, etc.). Other resources include colorbrewer¹² to help pick good color schemes. The plotting libraries mentioned in this chapter also have various color schemes that can be used.

⁹Seaborn documentation: <https://stanford.edu/~mwaskom/software/seaborn/api.html>

¹⁰Pandas plotting documentation: <http://pandas.pydata.org/pandas-docs/stable/visualization.html>

¹¹matplotlib documentation: <http://matplotlib.org/api/index.html>

¹²colorbrewer: <http://colorbrewer2.org/>

Figure 3-43: Pandas hexbin plot

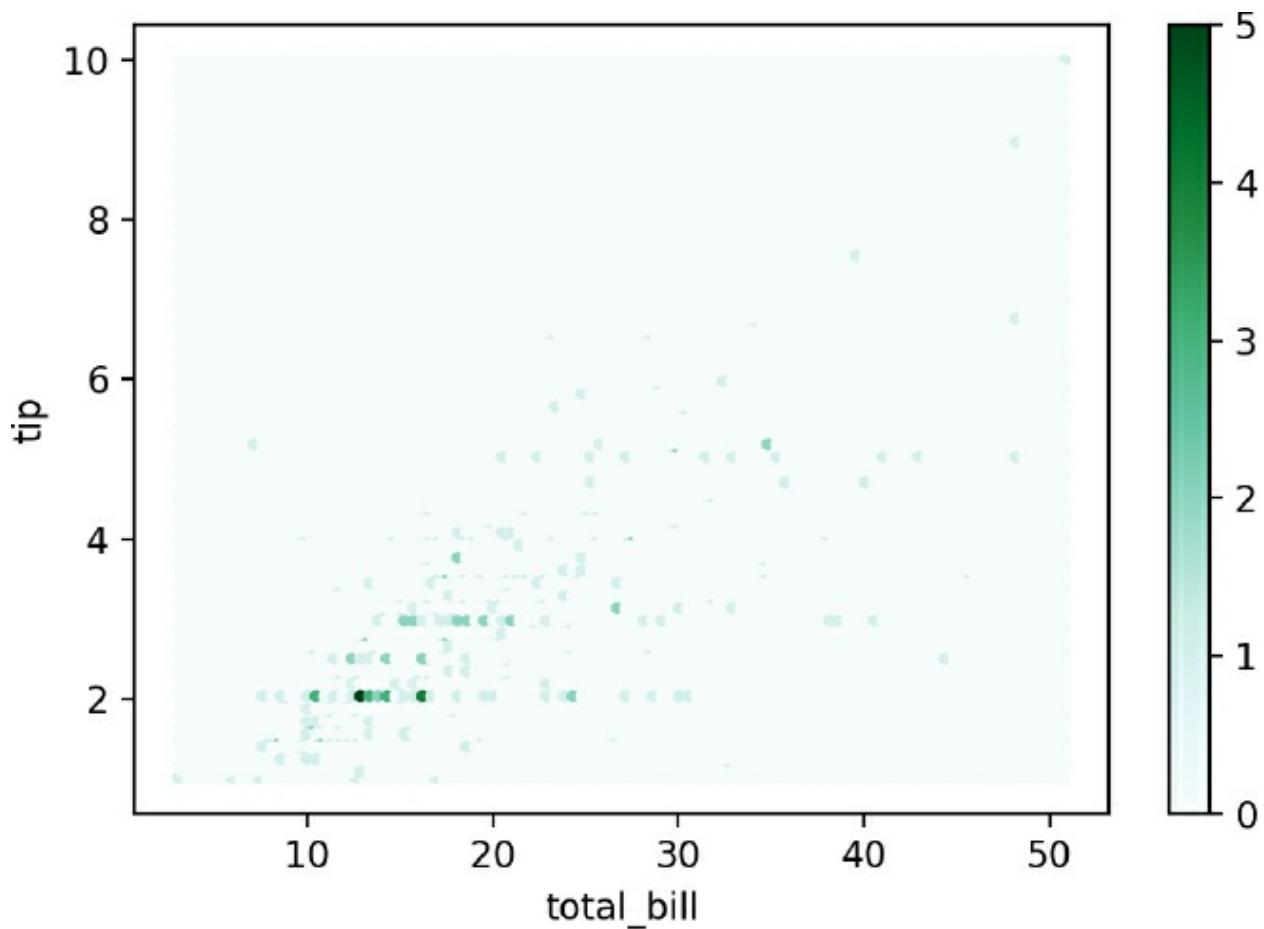


Figure 3-44: Pandas hexbin plot with modified gridszie

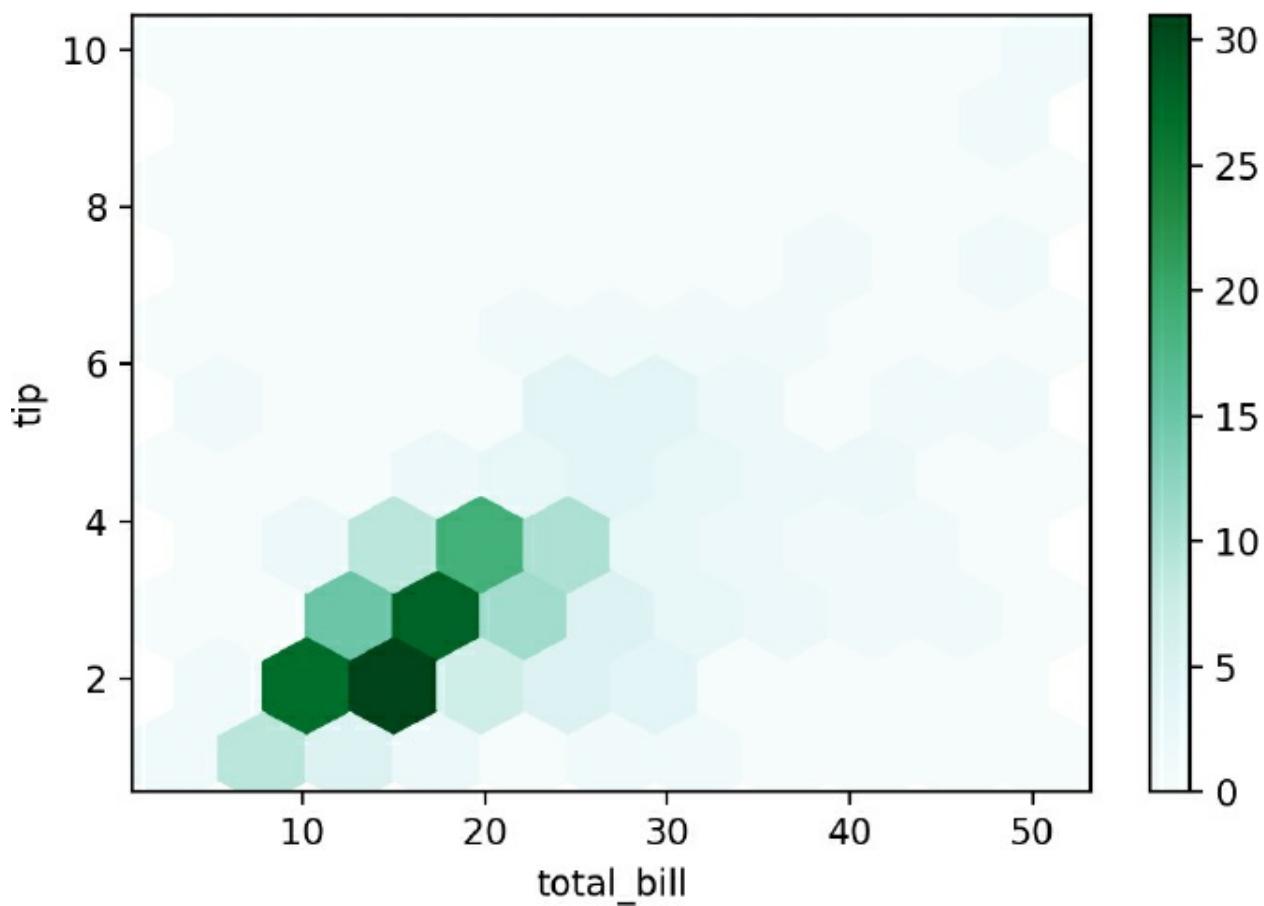


Figure 3-45: Pandas boxplot

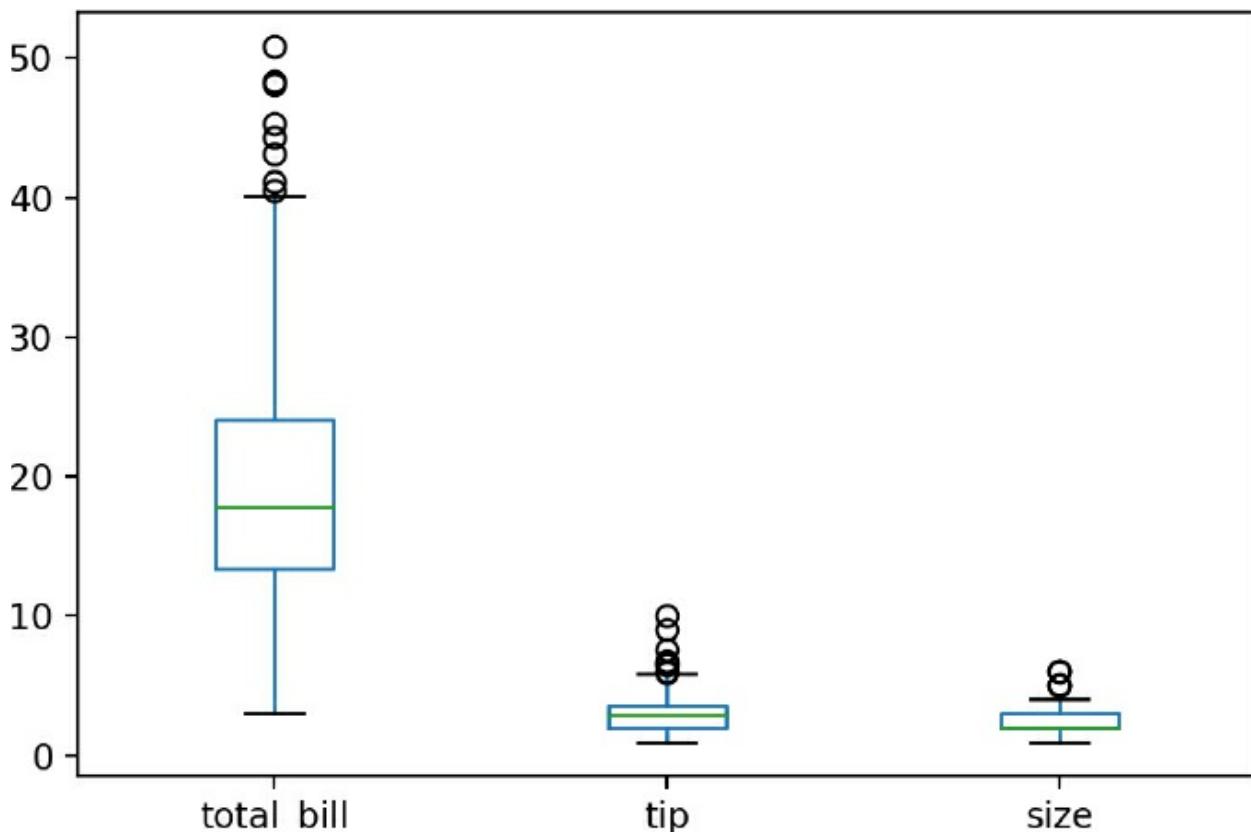


Figure 3-46: Seaborn style baseline

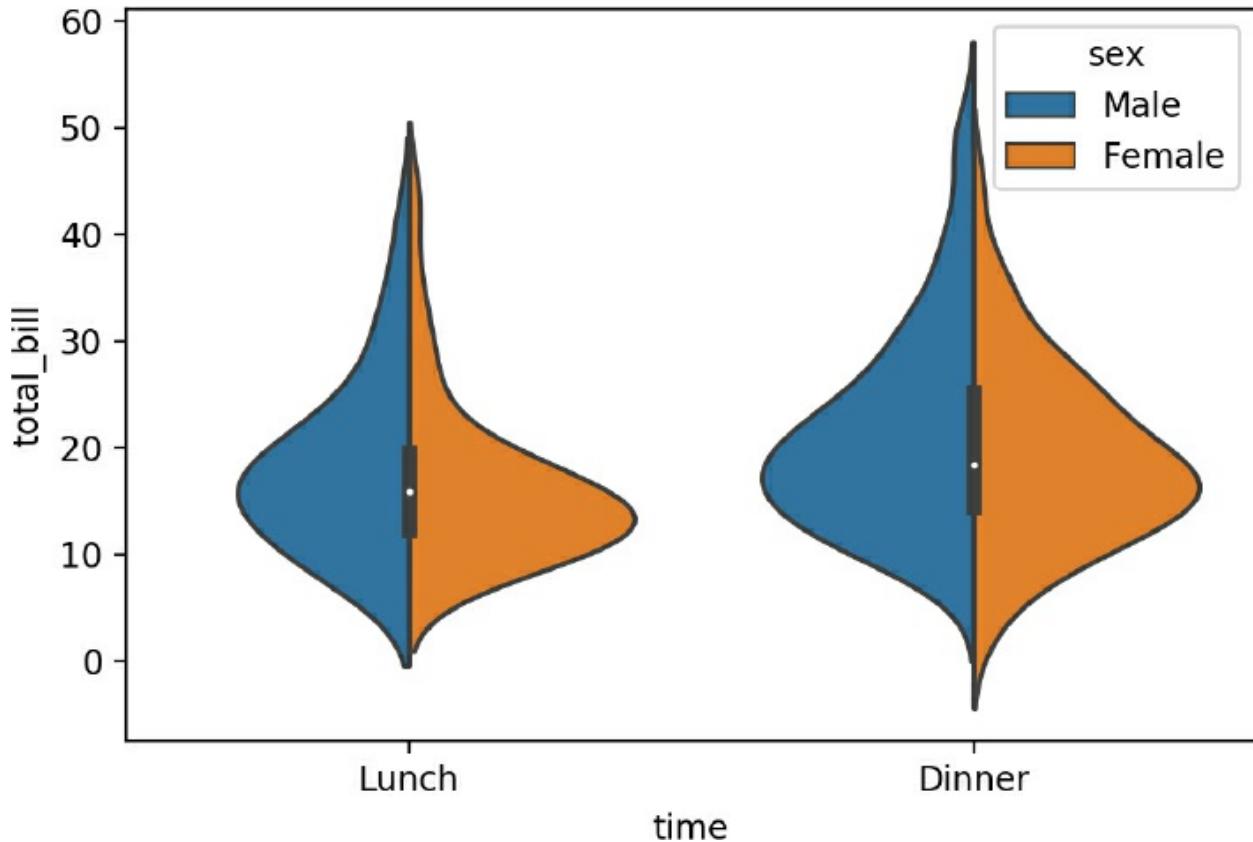


Figure 3-47: Seaborn style baseline

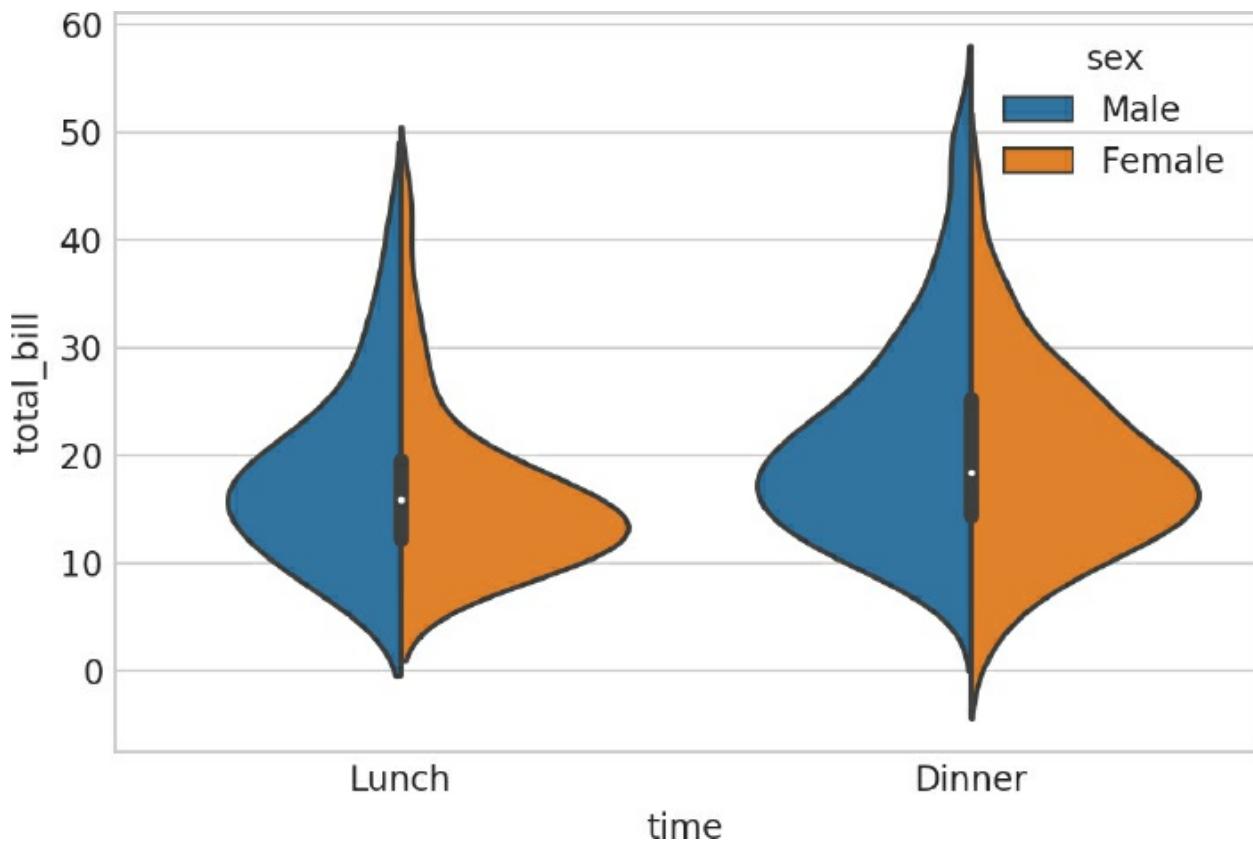
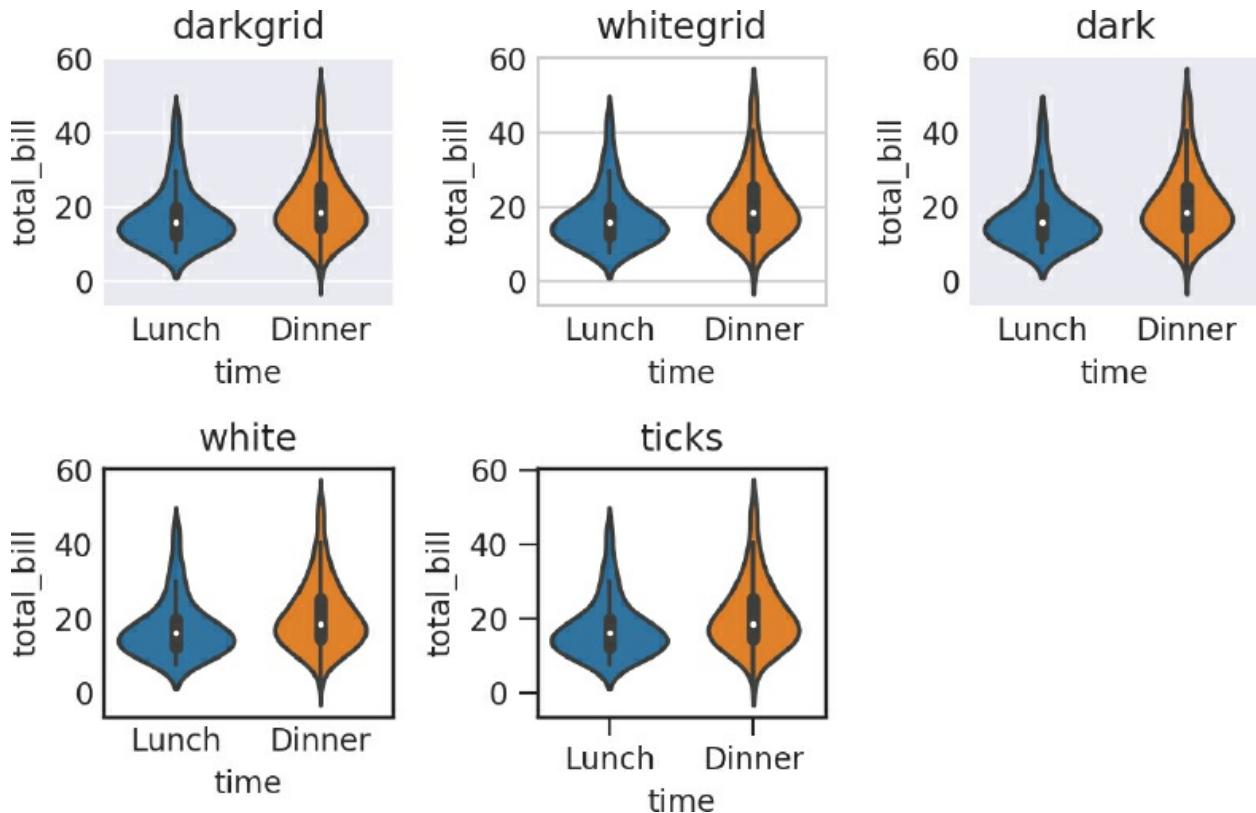


Figure 3-48: All seaborn styles



Part II: Data Manipulation

[Chapter 4](#), “[Data Assembly](#),” Combining and Subsetting data you want for analysis.

[Chapter 5](#), “[Missing Data](#),” Working with missing data.

[Chapter 6](#), “[Tidy Data](#),” Tidying data.

Chapter 4. Data Assembly

4.1 Introduction

Hopefully by now, you are able to load in data into `pandas` and do some basic visualizations. This part of the book will focus on various data cleaning tasks. We begin with assembling a dataset for analysis by combining various datasets together.

Concept map

1. Prior knowledge
 - (a) Loading data
 - (b) Subsetting data
 - (c) functions and class methods

Objectives

This chapter will cover:

1. Tidy data
2. Concatenating data
3. Merging datasets

4.2 Tidy Data

Hadley Wickham¹, one of the more prominent members in the R community, talks about the idea of *tidy* data. He's written a paper about it in the *Journal of Statistical Software*². Tidy data is a framework to structure datasets so they can be easily analyzed. It is mainly used as a goal one should aim for when cleaning data. Once you understand what tidy data is, it will also make data collection much easier.

¹Hadley Wickham's homepage <http://hadley.nz/>

²Tidy Data paper: <http://vita.had.co.nz/papers/tidy-data.pdf>

So what is *tidy* data? Hadley Wickham's paper defines it as such:

- each row is an observation
- each column is a variable
- each type of observational unit forms a table

4.2.1 Combining Datasets

We begin with Hadley Wickham's last tidy data point: "each type of observational unit forms a table". When data is tidy, you need to combine various tables together to answer a question. For example, there may be a separate table on company information and another table on stock prices. If we wanted to look at all the stock prices within the tech industry we may first have to find all the tech companies from the company information table, and then combine it with the stock price data to get the data we need for our question. The data was split up into separate tables to reduce the amount of redundant information (we don't need to store the company information with each stock price entry), but it means we as data analysts must combine the relevant data ourselves for our question.

Other times a single dataset will be split into multiple parts. This may be timeseries data where each date is in a separate file, or a file may have been split into parts to make the individual files smaller. You may also need to combine data from multiple sources to answer a question (e.g., combining latitudes and longitudes with zip codes). In both cases, you will need to combine data into a single dataframe for analysis.

4.3 Concatenation

One of the (conceptually) easier forms of combining data is concatenation. Concatenation can be thought of appending a row or column to your data. This can happen if your data was split into parts or if you made a calculation that you want to append.

Concatenation is all accomplished by using the `concat` function from pandas.

4.3.1 Adding Rows

Let's begin with some example data sets so you can see what is actually happening.

```
import pandas as pd

df1 = pd.read_csv('../data/concat_1.csv')
df2 = pd.read_csv('../data/concat_2.csv')
df3 = pd.read_csv('../data/concat_3.csv')

print(df1)

      A      B      C      D
0   a0    b0    c0    d0
1   a1    b1    c1    d1
2   a2    b2    c2    d2
3   a3    b3    c3    d3

print(df2)

      A      B      C      D
0   a4    b4    c4    d4
1   a5    b5    c5    d5
2   a6    b6    c6    d6
3   a7    b7    c7    d7

print(df3)

      A      B      C      D
0   a8    b8    c8    d8
1   a9    b9    c9    d9
2  a10   b10   c10   d10
3  a11   b11   c11   d11
```

Stacking the datarames on top of each other uses the `concat` function in `pandas` where all the dataframes to be concatenated are passed in a `list`.

```
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

	A	B	C	D
0	a0	b0	c0	d0
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
0	a4	b4	c4	d4
1	a5	b5	c5	d5
2	a6	b6	c6	d6
3	a7	b7	c7	d7
0	a8	b8	c8	d8
1	a9	b9	c9	d9
2	a10	b10	c10	d10
3	a11	b11	c11	d11

You can see `concat` blindly stacks the datarames together. If you look at the row names (a.k.a row index), they are also simply a stacked version of the original row indicies. If we tried the various subsetting methods from [Table 2-3](#), the table will subset as expected.

```
# subset the 4th row of the concatenated dataframe
print(row_concat.iloc[3, :])
```

A	a3
B	b3
C	c3
D	d3

Question

What happens when you use `loc` or `ix` to subset the new dataframe?

[Section 2.2.1](#) showed how to create a `series`. However, if we create a new series to append to a dataframe, you'd quickly see, that it does not append correctly.

```
# create a new row of data
new_row_series = pd.Series(['n1', 'n2', 'n3', 'n4'])
print(new_row_series)

0    n1
1    n2
2    n3
3    n4
dtype: object

# attempt to add the new row to a dataframe
print(pd.concat([df1, new_row_series]))
```

	A	B	C	D	0
0	a0	b0	c0	d0	NaN
1	a1	b1	c1	d1	NaN
2	a2	b2	c2	d2	NaN
3	a3	b3	c3	d3	NaN
0	NaN	NaN	NaN	NaN	n1
1	NaN	NaN	NaN	NaN	n2
2	NaN	NaN	NaN	NaN	n3
3	NaN	NaN	NaN	NaN	n4

The first things we will notice are `NaN` values. This is simply Python's way of representing a 'missing value' ([Chapter 5](#), Missing Values). Next, we were hoping to append our new values as

a row. Not only did our code not append the values as a row, it created a new column completely misaligned with everything else.

If we pause to think about what actually is happening, we can see the results actually make sense. First, if we look at the new indices that were added, It is very similar to how we concatenated dataframes earlier. The indices of the `newrow` series object are analogs to the row numbers of the dataframe. Next, since our series did not have a matching column, our `newrow` was added to a new column.

To fix this, we can turn our series into a dataframe. This data frame would have 1 row of data, and the column names would be the ones the data would bind to.

```
# note the double brackets
new_row_df = pd.DataFrame([['n1', 'n2', 'n3', 'n4']],
                           columns=['A', 'B', 'C', 'D'])
print(new_row_df)

      A      B      C      D
0    n1    n2    n3    n4
print(pd.concat([df1, new_row_df]))


      A      B      C      D
0    a0    b0    c0    d0
1    a1    b1    c1    d1
2    a2    b2    c2    d2
3    a3    b3    c3    d3
0    n1    n2    n3    n4
```

`concat` is a general function that can concatenate multiple things at once. If you just needed to append a single object to an existing dataframe, there's the `append` function for that.

Using a `DataFrame`

```
print(df1.append(df2))

      A      B      C      D
0    a0    b0    c0    d0
1    a1    b1    c1    d1
2    a2    b2    c2    d2
3    a3    b3    c3    d3
0    a4    b4    c4    d4
1    a5    b5    c5    d5
2    a6    b6    c6    d6
3    a7    b7    c7    d7
```

Using a single-row `DataFrame`

```
print(df1.append(new_row_df))

      A      B      C      D
0    a0    b0    c0    d0
1    a1    b1    c1    d1
2    a2    b2    c2    d2
3    a3    b3    c3    d3
0    n1    n2    n3    n4
```

Using a Python Dictionary

```
data_dict = {'A': 'n1',
             'B': 'n2',
             'C': 'n3',
             'D': 'n4'}

print(df1.append(data_dict, ignore_index=True))

      A      B      C      D
```

```

0    a0    b0    c0    d0
1    a1    b1    c1    d1
2    a2    b2    c2    d2
3    a3    b3    c3    d3
4    n1    n2    n3    n4

```

4.3.1.1 Ignoring the index

We saw in the last example when we tried to add a `dict` to a dataframe, we had to use the `ignoreIndex` parameter. If we look closer, you can see the row index also incremented by 1, and did not repeat a previous index value.

If we simply wanted to concatenate or append data together, we can use the `ignoreIndex` to reset the row index after the concatenation.

```

row_concat_i = pd.concat([df1, df2, df3], ignore_index=True)
print(row_concat_i)

```

	A	B	C	D
0	a0	b0	c0	d0
1	a1	b1	c1	d1
2	a2	b2	c2	d2
3	a3	b3	c3	d3
4	a4	b4	c4	d4
5	a5	b5	c5	d5
6	a6	b6	c6	d6
7	a7	b7	c7	d7
8	a8	b8	c8	d8
9	a9	b9	c9	d9
10	a10	b10	c10	d10
11	a11	b11	c11	d11

4.3.2 Adding Columns

Concatenating columns is very similar to concatenating rows. The main difference is the `axis` parameter in the `concat` function. The default value of `axis` has a value of `0`, so it will concatenate row-wise. However, if we pass `axis=1` to the function, it will concatenate column-wise.

```

col_concat = pd.concat([df1, df2, df3], axis=1)
print(col_concat)

```

	A	B	C	D	A	B	C	D	A	B	C	D
0	a0	b0	c0	d0	a4	b4	c4	d4	a8	b8	c8	d8
1	a1	b1	c1	d1	a5	b5	c5	d5	a9	b9	c9	d9
2	a2	b2	c2	d2	a6	b6	c6	d6	a10	b10	c10	d10
3	a3	b3	c3	d3	a7	b7	c7	d7	a11	b11	c11	d11

If we try to subset based on column names, we will get a similar result when we concatenated row-wise and subset by row index.

```

print(col_concat['A'])

```

	A	A	A
0	a0	a4	a8
1	a1	a5	a9
2	a2	a6	a10
3	a3	a7	a11

Adding a single column to a dataframe can be done directly without using any specific pandas function. Simply pass a new column name the vector you want assigned to the new column.

```

col_concat['new_col_list'] = ['n1', 'n2', 'n3', 'n4']
print(col_concat)

```

```

      A    B    C    D    A    B    C    D    A    B    C    D new_col_list
0   a0   b0   c0   d0   a4   b4   c4   d4   a8   b8   c8   d8      n1
1   a1   b1   c1   d1   a5   b5   c5   d5   a9   b9   c9   d9      n2
2   a2   b2   c2   d2   a6   b6   c6   d6   a10  b10  c10  d10     n3
3   a3   b3   c3   d3   a7   b7   c7   d7   a11  b11  c11  d11     n4

col_concat['new_col_series'] = pd.Series(['n1', 'n2', 'n3', 'n4'])
print(col_concat)

      A    B    C    D    A    B    C    D    A    B    C    D new_col_series
0   a0   b0   c0   d0   a4   b4   c4   d4   a8   b8   c8   d8      n1
1   a1   b1   c1   d1   a5   b5   c5   d5   a9   b9   c9   d9      n2
2   a2   b2   c2   d2   a6   b6   c6   d6   a10  b10  c10  d10     n3
3   a3   b3   c3   d3   a7   b7   c7   d7   a11  b11  c11  d11     n4

```

Using the `concat` function still works, as long as you pass it a dataframe. This does require a bit more unnecessary code.

Finally, we can choose to reset the column indices so we do not have duplicated column names.

```

print(pd.concat([df1, df2, df3], axis=1, ignore_index=True))

      0    1    2    3    4    5    6    7    8    9    10   11
0   a0   b0   c0   d0   a4   b4   c4   d4   a8   b8   c8   d8
1   a1   b1   c1   d1   a5   b5   c5   d5   a9   b9   c9   d9
2   a2   b2   c2   d2   a6   b6   c6   d6   a10  b10  c10  d10
3   a3   b3   c3   d3   a7   b7   c7   d7   a11  b11  c11  d11

```

4.3.3 Concatenation With Different Indices

The examples shown so far assume a simple row or column concatenation. It also assumes that the new row(s) had the same column names or the column(s) had the same row indices.

Here I will show you what happens when the row and column indices are not aligned.

4.3.3.1 Concatenate Rows With Different Columns

Let's modify our dataframes for the next few examples.

```

df1.columns = ['A', 'B', 'C', 'D']
df2.columns = ['E', 'F', 'G', 'H']
df3.columns = ['A', 'C', 'F', 'H']

print(df1)
      A    B    C    D
0   a0   b0   c0   d0
1   a1   b1   c1   d1
2   a2   b2   c2   d2
3   a3   b3   c3   d3

print(df2)
      E    F    G    H
0   a4   b4   c4   d4
1   a5   b5   c5   d5
2   a6   b6   c6   d6
3   a7   b7   c7   d7

print(df3)
      A    C    F    H
0   a8   b8   c8   d8
1   a9   b9   c9   d9
2   a10  b10  c10  d10
3   a11  b11  c11  d11

```

If we try to concatenate the dataframes like we did in [Section 4.3.1](#), you will now see the dataframes do much more than simply stack one on top of the other. The columns will align themselves, and a `NaN` value will fill any of the missing areas.

```
row_concat = pd.concat([df1, df2, df3])
print(row_concat)
```

	A	B	C	D	E	F	G	H
0	a0	b0	c0	d0	NaN	NaN	NaN	NaN
1	a1	b1	c1	d1	NaN	NaN	NaN	NaN
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN
3	a3	b3	c3	d3	NaN	NaN	NaN	NaN
0	NaN	NaN	NaN	NaN	a4	b4	c4	d4
1	NaN	NaN	NaN	NaN	a5	b5	c5	d5
2	NaN	NaN	NaN	NaN	a6	b6	c6	d6
3	NaN	NaN	NaN	NaN	a7	b7	c7	d7
0	a8	NaN	b8	NaN	NaN	c8	NaN	d8
1	a9	NaN	b9	NaN	NaN	c9	NaN	d9
2	a10	NaN	b10	NaN	NaN	c10	NaN	d10
3	a11	NaN	b11	NaN	NaN	c11	NaN	d11

One way to not have any `NaN` missing values is to only keep the columns that are in common from the list of objects to be concatenated. There is a parameter named `join` that accomplishes this. By default it has a value of '`outer`', meaning it will keep all the columns. However, we can set `join='inner'` to keep only the columns that

If we try to keep only the columns from all 3 dataframes, we will get an empty dataframe since there are no columns in common.

```
print(pd.concat([df1, df2, df3], join='inner'))
```

Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]

If we use the dataframes that have columns in common, only the columns that all of them share will be returned.

```
print(pd.concat([df1, df3], ignore_index=False, join='inner'))
```

	A	C
0	a0	c0
1	a1	c1
2	a2	c2
3	a3	c3
0	a8	b8
1	a9	b9
2	a10	b10
3	a11	b11

4.3.3.2 Concatenate Columns With Different Rows

Let's take our dataframes and modify them again with different row indices. I am building on the same dataframe modifications from [Section 4.3.3.1](#).

```
df1.index = [0, 1, 2, 3]
df2.index = [4, 5, 6, 7]
df3.index = [0, 2, 5, 7]
```

```
print(df1)

      A    B    C    D
0   a0   b0   c0   d0
1   a1   b1   c1   d1
2   a2   b2   c2   d2
3   a3   b3   c3   d3
```

```

print(df2)

   E   F   G   H
4  a4  b4  c4  d4
5  a5  b5  c5  d5
6  a6  b6  c6  d6
7  a7  b7  c7  d7

print(df3)

   A   C   F   H
0  a8  b8  c8  d8
2  a9  b9  c9  d9
5  a10 b10 c10 d10
7  a11 b11 c11 d11

```

When we concatenate along `axis=1`, we get the same results from concatenating along `axis =0`. The new dataframes will be added column wise and matched against their respective row indices. Missing values will fill in the areas where the indices did not align.

```

col_concat = pd.concat([df1, df2, df3], axis=1)
print(col_concat)

```

	A	B	C	D	E	F	G	H	A	C	F	H
0	a0	b0	c0	d0	NaN	NaN	NaN	NaN	a8	b8	c8	d8
1	a1	b1	c1	d1	NaN							
2	a2	b2	c2	d2	NaN	NaN	NaN	NaN	a9	b9	c9	d9
3	a3	b3	c3	d3	NaN							
4	NaN	NaN	NaN	NaN	a4	b4	c4	d4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	a5	b5	c5	d5	a10	b10	c10	d10
6	NaN	NaN	NaN	NaN	a6	b6	c6	d6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	a7	b7	c7	d7	a11	b11	c11	d11

Lastly, just like we did when we concatenated row-wise, we can choose to only keep the results when there are matching indices by using `join='inner'`.

```

print(pd.concat([df1, df3], axis=1, join='inner'))

```

	A	B	C	D	A	C	F	H
0	a0	b0	c0	d0	a8	b8	c8	d8
2	a2	b2	c2	d2	a9	b9	c9	d9

4.4 Merging Multiple Datasets

The end of the previous section alluded to a few database concepts. The `join='inner'` and the default `join ='outer'` parameters come from working with databases when we want to merge tables.

Instead of simply having a row or column index that we want to concatenate values to, there will be times when you have 2 or more dataframes that you want to combine based on common data values. This is known in the database world as performing a “join”.

Pandas has a `pd.join` command that uses `pd.merge` under the hood. `join` will merge dataframe objects by an index, but the `merge` command is much more explicit and flexible. If you are only planning to merge dataframes by the row index, you can look into the `join` function.³

³Pandas `DataFrame join` function: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.join.html>

We will be using the survey data in this series of examples.

```

person = pd.read_csv('../data/survey_person.csv')
site = pd.read_csv('../data/survey_site.csv')
survey = pd.read_csv('../data/survey_survey.csv')
visited = pd.read_csv('../data/survey_visited.csv')

print(person)

      ident    personal    family
0      dyer     William     Dyer
1       pb        Frank  Pabodie
2      lake   Anderson     Lake
3      roe    Valentina  Roerich
4  danforth      Frank  Danforth

print(site)

      name      lat      long
0  DR-1  -49.85  -128.57
1  DR-3  -47.15  -126.72
2  MSK-4  -48.87  -123.40

print(visited)

      ident    site      dated
0      619  DR-1  1927-02-08
1      622  DR-1  1927-02-10
2      734  DR-3  1939-01-07
3      735  DR-3  1930-01-12
4      751  DR-3  1930-02-26
5      752  DR-3        NaN
6      837  MSK-4  1932-01-14
7      844  DR-1  1932-03-22

print(survey)

      taken person  quant      reading
0       619   dyer    rad      9.82
1       619   dyer    sal      0.13
2       622   dyer    rad      7.80
3       622   dyer    sal      0.09
4       734     pb    rad      8.41
5       734   lake    sal      0.05
6       734     pb  temp     -21.50
7       735     pb    rad      7.22
8       735     NaN    sal      0.06
9       735     NaN  temp     -26.00
10      751     pb    rad      4.35
11      751     pb  temp     -18.50
12      751   lake    sal      0.10
13      752   lake    rad      2.19
14      752   lake    sal      0.09
15      752   lake  temp     -16.00
16      752   roe    sal     41.60
17      837   lake    rad      1.46
18      837   lake    sal      0.21
19      837   roe    sal     22.50
20      844   roe    rad     11.25

```

Currently, our data is split into multiple parts, where each part is an observational unit. If we wanted to look at the dates at each site with the lat long of the site. We would have to combine (and merge) multiple dataframes. We do this with the `merge` function in pandas. `merge` is actually a `DataFrame` method.

When we call this method, the dataframe that is called will be referred to the one on the “`left`”. Within the `merge` function, the first parameter is the “`right`” dataframe. The next parameter is `how` the final merged result looks. See [Table 4-1](#) for more details. The next, we set the `on` parameter. This specifies which columns to match on. If the left and right columns are not the same name, we can use the `left_on` and `right_on` parameters instead.

Table 4-1: How the pandas `how` parameter relates to SQL

Pandas	SQL	Description
left	left outer	Keep all the keys from the left
right	right outer	Keep all the keys from the right
outer	full outer	Keep all the keys from both left and right
inner	inner	keep only the keys that exist in the left and right

4.4.1 One-to-one

The simplest type of merge we can do is when we have 2 dataframes where we want to join one column to another column, and when the columns we want to join do not have duplicate values in them.

For this example I am going to modify the `visited` dataframe so there are no duplicated `site` values.

```
visited_subset = visited.loc[[0, 2, 6], ]
```

We can perform our one-to-one merge as follows:

```
# the default value for 'how' is 'inner'
# so it doesn't need to be specified
o2o_merge = site.merge(visited_subset,
                      left_on='name', right_on='site')
print(o2o_merge)

      name    lat   long     ident    site       dated
0    DR-1 -49.85 -128.57     619  DR-1  1927-02-08
1    DR-3 -47.15 -126.72     734  DR-3  1939-01-07
2  MSK-4 -48.87 -123.40     837  MSK-4  1932-01-14
```

You can see here that we now have a new dataframe from 2 separate dataframes where the rows were matched based on a particular set of columns. In SQL speak, the columns used to match are called ‘key(s)’.

4.4.2 Many-to-one

If we choose to do the same merge, but this time without using the subsetted `visited` dataframe, we would perform a many-to-one merge. This happens when performing a merge and one of the dataframes has key values that repeat.

When this happens, the dataframe that contains the single observations will be duplicated in the merge.

```
m2o_merge = site.merge(visited, left_on='name', right_on='site')
print(m2o_merge)

      name    lat   long     ident    site       dated
0    DR-1 -49.85 -128.57     619  DR-1  1927-02-08
1    DR-1 -49.85 -128.57     622  DR-1  1927-02-10
2    DR-1 -49.85 -128.57     844  DR-1  1932-03-22
3    DR-3 -47.15 -126.72     734  DR-3  1939-01-07
4    DR-3 -47.15 -126.72     735  DR-3  1930-01-12
5    DR-3 -47.15 -126.72     751  DR-3  1930-02-26
6    DR-3 -47.15 -126.72     752  DR-3        NaN
7  MSK-4 -48.87 -123.40     837  MSK-4  1932-01-14
```

As you can see, the `site` information (`name`, `lat`, and `long`) were duplicated and matched to the

`visited` data.

4.4.3 Many-to-many

Lastly, there will be times when we want to perform a match based on multiple columns. This can also be performed.

Let's say we have 2 dataframes that come from the person merged with survey, and another dataframe that comes from visited merged with survey.

```
ps = person.merge(survey, left_on='ident', right_on='person')
vs = visited.merge(survey, left_on='ident', right_on='taken')
```

```
print(ps)
```

	ident	personal	family	taken	person	quant	reading
0	dyer	William	Dyer	619	dyer	rad	9.82
1	dyer	William	Dyer	619	dyer	sal	0.13
2	dyer	William	Dyer	622	dyer	rad	7.80
3	dyer	William	Dyer	622	dyer	sal	0.09
4	pb	Frank	Pabodie	734	pb	rad	8.41
5	pb	Frank	Pabodie	734	pb	temp	-21.50
6	pb	Frank	Pabodie	735	pb	rad	7.22
7	pb	Frank	Pabodie	751	pb	rad	4.35
8	pb	Frank	Pabodie	751	pb	temp	-18.50
9	lake	Anderson	Lake	734	lake	sal	0.05
10	lake	Anderson	Lake	751	lake	sal	0.10
11	lake	Anderson	Lake	752	lake	rad	2.19
12	lake	Anderson	Lake	752	lake	sal	0.09
13	lake	Anderson	Lake	752	lake	temp	-16.00
14	lake	Anderson	Lake	837	lake	rad	1.46
15	lake	Anderson	Lake	837	lake	sal	0.21
16	roe	Valentina	Roerich	752	roe	sal	41.60
17	roe	Valentina	Roerich	837	roe	sal	22.50
18	roe	Valentina	Roerich	844	roe	rad	11.25

```
print(vs)
```

	ident	site	dated	taken	person	quant	reading
0	619	DR-1	1927-02-08	619	dyer	rad	9.82
1	619	DR-1	1927-02-08	619	dyer	sal	0.13
2	622	DR-1	1927-02-10	622	dyer	rad	7.80
3	622	DR-1	1927-02-10	622	dyer	sal	0.09
4	734	DR-3	1939-01-07	734	pb	rad	8.41
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3	NaN	752	lake	rad	2.19
14	752	DR-3	NaN	752	lake	sal	0.09
15	752	DR-3	NaN	752	lake	temp	-16.00
16	752	DR-3	NaN	752	roe	sal	41.60
17	837	MSK-4	1932-01-14	837	lake	rad	1.46
18	837	MSK-4	1932-01-14	837	lake	sal	0.21
19	837	MSK-4	1932-01-14	837	roe	sal	22.50
20	844	DR-1	1932-03-22	844	roe	rad	11.25

We can perform a many-to-many merge by passing the multiple columns to match on in a python list.

```
ps_vs = ps.merge(vs,
                  left_on=['ident', 'taken', 'quant', 'reading'],
                  right_on=['person', 'ident', 'quant', 'reading'])
```

If we just take a look at the first row of data:

```
print(ps_vs.loc[0, :])
```

ident_x	dyer
personal	William
family	Dyer
taken_x	619
person_x	dyer
quant	rad
reading	9.82
ident_y	619
site	DR-1
dated	1927-02-08
taken_y	619
person_y	dyer
Name: 0, dtype:	object

Pandas will automatically add a suffix to a column name if there are collisions in the name. the `jx` refers to values from the left dataframe, and the `_y` suffix comes from values in the right dataframe.

4.5 Conclusion

There will be times when you need to combine various parts or data or multiple datasets depending on the question you are trying to answer. One thing to keep in mind, the data you need for analysis, does not necessarily mean the best shape of data for storage.

The survey data used in the last example came in 4 separate parts that needed to be merged together. After we merged the tables together, you will notice a lot of redundant information across rows. From a data storage and entry point of view, each of these duplications can lead to errors and data inconsistency. This is what Hadley meant by “each type of observational unit forms a table”.

Chapter 5. Missing Data

5.1 Introduction

Rarely will you be given a dataset without any missing values. There are many representations of missing data. In databases they are `NULL` values, Certain programming languages will use `NA`, and depending on where you get your data, missing values can be an empty string, "", or even numeric values such as `88` or `99`. Pandas has displays missing values as `NaN`.

Concept map

1. Prior knowledge
 - (a) importing libraries
 - (b) slicing and indexing data
 - (c) using functions and methods
 - (d) using function parameters

Objectives

This chapter will cover:

1. What is a missing value
2. How are missing values created
3. How to recode and make calculations with missing values

5.2 What is a `NaN` value

We can get the `NaN` value from `numpy`. You may see missing values in python used or displayed in a few ways: `NaN`, `NAN`, or `nan`. They are all equivalent. See [Appendix H](#) on how the missing values are imported below.

```
# Just import the numpy missing values
from numpy import NaN, NAN, nan
```

Missing values are different than other types of data, in that they don't really equal anything. The data is missing, so there is no concept of equality. `NaN` is not be equivalent to `0` or an empty string, `"`.

We can illustrate this in python by testing it's equality.

```
print(NaN == True)
False
```

```
print(NaN == False)
False
print(NaN == 0)
False
print(NaN == '')
False
```

To illustrate the lack of equality, missing values are also not equal to misisng values.

```
print(NaN == NaN)
False
print(NaN == nan)
False
print(NaN == NAN)
False
print(nan == NAN)
False
```

Pandas has built-in methods to test for a missing value.

```
import pandas as pd
print(pd.isnull(NaN))
True
print(pd.isnull(nan))
True
print(pd.isnull(NAN))
True
```

Pandas also has methods for testing non-missing values

```
print(pd.notnull(NaN))
False
print(pd.notnull(42))
True
print(pd.notnull('missing'))
True
```

5.3 Where do missing values come from?

We can get missing values from loading in data with missing values, or from the data munging process.

5.3.1 Load data

The survey data we used in [Chapter 4](#) had a dataset, `visited`, which contained missing data. When we loaded the data, `pandas` automatically found the missing data cell, and gave us a `dataframe` with the `NaN` value in the appropriate cell. In the `read_csv` function, there are three parameters that relate to reading in missing values: `na_values`, `keep_default_na`, and `na_filter`.

`na_values` allow you to specify additional missing or `NaN` values. You can either pass in a python `str` or list-like object for to be automatically coded as missing values when the file is read. There are already default missing values, such as `NA`, `NaN`, or `nan`, which is why this parameter is not always used. Some health data will code `99` as a missing value; an example of a value you would set in this field is `na_values=[99]`.

`keep_default_na` is a `bool` that allows you to specify whether any additional values need to be considered as missing. This parameter is `True` by default, meaning, any additional missing values specified with the `na_values` parameter will be appended to the list of missing values. However, `keep_default_na` can also be set to `keep_default_na =False` to only use the missing values specified in `na_values`

Lastly, `na.filter` is a `bool` that will specify whether or not any values will be read as missing. The default value of `na.filter =True` means that missing values will be coded as a `NaN`. If we assign `na.filter =False`, then nothing will be recoded as missing. This can be thought of as a means to turn off all the parameters set for `na_values` and `keep_default_na`, but it really is used when you want a performance boost loading in data without missing values.

```
# set the location for data
visited_file = '../data/survey_visited.csv'

# load the data with default values
print(pd.read_csv(visited_file))

    ident      site      dated
0       619   DR-1  1927-02-08
1       622   DR-1  1927-02-10
2       734   DR-3  1939-01-07
3       735   DR-3  1930-01-12
4       751   DR-3  1930-02-26
5       752   DR-3        NaN
6       837  MSK-4  1932-01-14
7       844   DR-1  1932-03-22

# load the data without default missing values
print(pd.read_csv(visited_file, keep_default_na=False))

    ident      site      dated
0       619   DR-1  1927-02-08
1       622   DR-1  1927-02-10
2       734   DR-3  1939-01-07
3       735   DR-3  1930-01-12
4       751   DR-3  1930-02-26
5       752   DR-3
6       837  MSK-4  1932-01-14
7       844   DR-1  1932-03-22

# manually specify missing values
print(pd.read_csv(visited_file,
                  na_values=[''],
                  keep_default_na=False))

    ident      site      dated
0       619   DR-1  1927-02-08
1       622   DR-1  1927-02-10
2       734   DR-3  1939-01-07
3       735   DR-3  1930-01-12
4       751   DR-3  1930-02-26
5       752   DR-3        NaN
6       837  MSK-4  1932-01-14
7       844   DR-1  1932-03-22
```

5.3.2 Merged data

[Chapter 4](#) showed how to combine datasets. Some of the examples in the chapter showed missing values in the output. If we recreate the merged table from [Section 4.4.3](#), we will see missing values in the merged output.

```
visited = pd.read_csv('..../data/survey_visited.csv')
survey = pd.read_csv('..../data/survey_survey.csv')

print(visited)

      ident    site      dated
0       619  DR-1  1927-02-08
1       622  DR-1  1927-02-10
2       734  DR-3  1939-01-07
3       735  DR-3  1930-01-12
4       751  DR-3  1930-02-26
5       752  DR-3        NaN
6       837  MSK-4  1932-01-14
7       844  DR-1  1932-03-22

print(survey)

      taken person quant      reading
0       619   dyer   rad      9.82
1       619   dyer   sal      0.13
2       622   dyer   rad      7.80
3       622   dyer   sal      0.09
4       734     pb   rad      8.41
5       734   lake   sal      0.05
6       734     pb  temp     -21.50
7       735     pb   rad      7.22
8       735     NaN   sal      0.06
9       735     NaN  temp     -26.00
10      751     pb   rad      4.35
11      751     pb  temp     -18.50
12      751   lake   sal      0.10
13      752   lake   rad      2.19
14      752   lake   sal      0.09
15      752   lake  temp     -16.00
16      752   roe   sal     41.60
17      837   lake   rad      1.46
18      837   lake   sal      0.21
19      837   roe   sal     22.50
20      844   roe   rad     11.25

vs = visited.merge(survey, left_on='ident', right_on='taken')
print(vs)
```

	ident	site	dated	taken	person	quant	reading
0	619	DR-1	1927-02-08	619	dyer	rad	9.82
1	619	DR-1	1927-02-08	619	dyer	sal	0.13
2	622	DR-1	1927-02-10	622	dyer	rad	7.80
3	622	DR-1	1927-02-10	622	dyer	sal	0.09
4	734	DR-3	1939-01-07	734	pb	rad	8.41
5	734	DR-3	1939-01-07	734	lake	sal	0.05
6	734	DR-3	1939-01-07	734	pb	temp	-21.50
7	735	DR-3	1930-01-12	735	pb	rad	7.22
8	735	DR-3	1930-01-12	735	NaN	sal	0.06
9	735	DR-3	1930-01-12	735	NaN	temp	-26.00
10	751	DR-3	1930-02-26	751	pb	rad	4.35
11	751	DR-3	1930-02-26	751	pb	temp	-18.50
12	751	DR-3	1930-02-26	751	lake	sal	0.10
13	752	DR-3	NaN	752	lake	rad	2.19
14	752	DR-3	NaN	752	lake	sal	0.09
15	752	DR-3	NaN	752	lake	temp	-16.00
16	752	DR-3	NaN	752	roe	sal	41.60
17	837	MSK-4	1932-01-14	837	lake	rad	1.46
18	837	MSK-4	1932-01-14	837	lake	sal	0.21
19	837	MSK-4	1932-01-14	837	roe	sal	22.50
20	844	DR-1	1932-03-22	844	roe	rad	11.25

5.3.3 User input values

Missing values could also be created by the user. This can come from creating a vector of values from a calculation or a manually curated vector. To build on the examples from [Section 2.2](#), we can create our own data with missing values. `Nans` are valid values for `Series` and `DataFrames`.

```
# missing value in a series
num_legs = pd.Series({'goat': 4, 'amoeba': nan})
print(num_legs)
amoeba    NaN
goat      4.0
dtype: float64
# missing value in a dataframe
scientists = pd.DataFrame({
    'Name': ['Rosaline Franklin', 'William Gosset'],
    'Occupation': ['Chemist', 'Statistician'],
    'Born': ['1920-07-25', '1876-06-13'],
    'Died': ['1958-04-16', '1937-10-16'],
    'missing': [NaN, nan]})
print(scientists)
   Born          Died           Name       Occupation      missing
0  1920-07-25  1958-04-16  Rosaline Franklin     Chemist      NaN
1  1876-06-13  1937-10-16  William Gosset  Statistician      NaN
```

You can also assign a column of missing values to a dataframe directly.

```
# create a new dataframe
scientists = pd.DataFrame({
    'Name': ['Rosaline Franklin', 'William Gosset'],
    'Occupation': ['Chemist', 'Statistician'],
    'Born': ['1920-07-25', '1876-06-13'],
    'Died': ['1958-04-16', '1937-10-16']})

# assign a columns of missing values
scientists['missing'] = nan

print(scientists)
   Born          Died           Name       Occupation      missing
0  1920-07-25  1958-04-16  Rosaline Franklin     Chemist      NaN
1  1876-06-13  1937-10-16  William Gosset  Statistician      NaN
```

5.3.4 Re-indexing

Lastly, another way to introduce missing values into your data is to reindex your dataframe. This is useful when you want to add new indicies to your dataframe, but still want to retain its original values. A common useage is when your index represents some time interval, and you want to add more dates.

If we wanted to only look at the years from 2000 to 2010 from the gapminder plot in [Section 1.5](#), we can perform the same grouped operations, subset the data and then re-index it.

```
gapminder = pd.read_csv('../data/gapminder.tsv', sep='\t')

life_exp = gapminder.groupby(['year'])['lifeExp'].mean()
print(life_exp)

year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
```

```
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

We can re-index by slicing the data (See [Section 1.3](#))

```
# note you can continue to chain the `loc` from the code above
print(life_exp.loc[range(2000, 2010), ])

year
2000      NaN
2001      NaN
2002    65.694923
2003      NaN
2004      NaN
2005      NaN
2006      NaN
2007    67.007423
2008      NaN
2009      NaN
Name: lifeExp, dtype: float64
```

Or subset the data separately, and use the `reindex` method.

```
# subset
y2000 = life_exp[life_exp.index > 2000]
print(y2000)

year
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64

# reindex
print(y2000.reindex(range(2000, 2010)))

year
2000      NaN
2001      NaN
2002    65.694923
2003      NaN
2004      NaN
2005      NaN
2006      NaN
2007    67.007423
2008      NaN
2009      NaN
Name: lifeExp, dtype: float64
```

5.4 Working with missing data

Now that we know how missing values can be created, let's see how they behave when working with data.

5.4.1 Find and Count missing data

```
ebola = pd.read_csv('../data/country_timeseries.csv')
```

One way to look at the number of missing values is to count them.

```
# count the number of non-missing values
print(ebola.count())
```

Date	122
Day	122
Cases_Guinea	93

```

Cases_Liberia          83
Cases_SierraLeone      87
Cases_Nigeria          38
Cases_Senegal           25
Cases_UnitedStates      18
Cases_Spain              16
Cases_Mali                12
Deaths_Guinea            92
Deaths_Liberia           81
Deaths_SierraLeone       87
Deaths_Nigeria           38
Deaths_Senegal            22
Deaths_UnitedStates       18
Deaths_Spain              16
Deaths_Mali                12
dtype: int64

```

If we wanted, we can subtract the number of non-missing from the total number of rows.

```

num_rows = ebola.shape[0]
num_missing = num_rows - ebola.count()
print(num_missing)

Date                  0
Day                   0
Cases_Guinea          29
Cases_Liberia          39
Cases_SierraLeone      35
Cases_Nigeria          84
Cases_Senegal           97
Cases_UnitedStates     104
Cases_Spain             106
Cases_Mali              110
Deaths_Guinea           30
Deaths_Liberia          41
Deaths_SierraLeone      35
Deaths_Nigeria          84
Deaths_Senegal           100
Deaths_UnitedStates      104
Deaths_Spain             106
Deaths_Mali              110
dtype: int64

```

If you wanted to count the total number of missing values in your data, or count the number of missing values for a particular columns, you can use the `count_nonzero` function from `numpy` in conjunction with the `isnull` method.

```

import numpy as np

print(np.count_nonzero(ebola.isnull()))
1214

print(np.count_nonzero(ebola['Cases_Guinea'].isnull()))

```

Another way to get missing data counts is to use the `value_counts` method on a series. This will print a frequency table of values, if you use the `dropna` parameter, you can also get a missing value count.

```

# get the first 5 value counts from the Cases_Guinea column
print(ebola.Cases_Guinea.value_counts(dropna=False).head())

NaN      29
86.0      3
495.0      2
112.0      2
390.0      2
Name: Cases_Guinea, dtype: int64

```

5.4.2 Cleaning missing data

There are many different ways we can deal with missing data. Either by replacing it with another value, filling them using existing data, or dropping them from our dataset.

5.4.2.1 Recode/Replace

We Can use the `fillna` method to recode the missing values to another value. For example, if we wanted the missing values to be recoded as a 0.

```
print(ebola.fillna(0).iloc[0:10, 0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	0.0	10030.0
1	1/4/2015	288	2775.0	0.0	9780.0
2	1/3/2015	287	2769.0	8166.0	9722.0
3	1/2/2015	286	0.0	8157.0	0.0
4	12/31/2014	284	2730.0	8115.0	9633.0
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	0.0	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	0.0	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

You can see if we use `fillna`, we can recode the values to a specific value. If you look into the documentation, `fillna`, like many other pandas functions, have a parameter for `inplace`. This simply means, the underlying data will be automatically changed without creating a new copy with the changes. This is a parameter you will want to use when your data gets larger and you want to be more memory efficient.

5.4.2.2 Fill Forwards

We can use built-in methods to fill forwards or backwards. When we fill data forwards, it means take the last known value, and use that value for the next missing value. This way, missing values are replaced with the last known/recoded value.

```
print(ebola.fillna(method='ffill').iloc[0:10, 0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	NaN	10030.0
1	1/4/2015	288	2775.0	NaN	9780.0
2	1/3/2015	287	2769.0	8166.0	9722.0
3	1/2/2015	286	2769.0	8157.0	9722.0
4	12/31/2014	284	2730.0	8115.0	9633.0
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	8018.0	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	7977.0	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

If a column begins with a missing value, then it will remain missing because there is no previous value to fill in.

5.4.2.3 Fill Backwards

We can also have pandas fill data backwards. When we fill data backwards, the newest value is used to replace missing. This way, missing values are replaced with the newest value.

```
print(ebola.fillna(method='bfill').iloc[:, 0:5].tail())
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
117	3/27/2014	5	103.0	8.0	6.0
118	3/26/2014	4	86.0	NaN	NaN

```

119    3/25/2014      3     86.0      NaN      NaN
120    3/24/2014      2     86.0      NaN      NaN
121    3/22/2014      0     49.0      NaN      NaN

```

If a column ends with a missing value, then it will remain missing because there is no new value to fill in.

5.4.2.4 interpolate

Interpolation uses existing values values to fill in missing values. There are many ways to fill in missing values, the interpolation in pandas fills in missing values linearly. Specifically, it treats the missing values to be equally spaced apart.

```
print(ebola.interpolate().iloc[0:10, 0:5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
0	1/5/2015	289	2776.0	NaN	10030.0
1	1/4/2015	288	2775.0	NaN	9780.0
2	1/3/2015	287	2769.0	8166.0	9722.0
3	1/2/2015	286	2749.5	8157.0	9677.5
4	12/31/2014	284	2730.0	8115.0	9633.0
5	12/28/2014	281	2706.0	8018.0	9446.0
6	12/27/2014	280	2695.0	7997.5	9409.0
7	12/24/2014	277	2630.0	7977.0	9203.0
8	12/21/2014	273	2597.0	7919.5	9004.0
9	12/20/2014	272	2571.0	7862.0	8939.0

The `interpolate` method has a `method` parameter that can change the interpolation method.¹

¹Series `interpolate` documentation: <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.interpolate.html>

5.4.2.5 Drop Missing values

The last way to work with missing data is to drop observations or variables with missing data. Depending on how much data is missing, only keeping complete case data can leave you with a useless dataset. Either the missing data is not random, and dropping missing values will leave you with a biased dataset, or keeping only complete data will leave you with not enough data to run your analysis.

We can use the `dropna` method to drop missing data. There are a few ways we can control how data can be dropped. The `dropna` method has a `how` parameter that lets you specify whether a row (or column) is dropped when 'any' or 'all' the data is missing.

The `thresh` parameter lets you specify how many non-NA values you have before dropping the row or column.

```
print(ebola.shape)
(122, 18)
```

If we only keep complete cases in our ebola dataset, we are only left with 1 row of data.

```
ebola_dropna = ebola.dropna()
print(ebola_dropna.shape)
(1, 18)

print(ebola_dropna)
```

```

19      Date      Day    Cases_Guinea      Cases_Liberia      Cases_SierraLeone      \
19      11/18/2014 241     2047.0          7082.0           6190.0
19      Cases_Nigeria 20.0      Cases_Senegal 1.0      Cases_UnitedStates 4.0      Cases_Spain 1.0      \
19      Cases_Mali 6.0      Deaths_Guinea 1214.0      Deaths_Liberia 2963.0      Deaths_SierraLeone 1267.0      \
19      Deaths_Nigeria 8.0      Deaths_Senegal 0.0      Deaths_UnitedStates 1.0      \
19      Deaths_Spain 0.0      Deaths_Mali 6.0

```

5.4.3 Calculations with missing data

Let's say we wanted to look at the case counts for multiple regions. We can add multiple regions together to get a new column of case counts.

```
ebola['Cases_multiple'] = ebola['Cases_Guinea'] + \
                           ebola['Cases_Liberia'] + \
                           ebola['Cases_SierraLeone']
```

We can look at the results by looking at the first 10 lines of the calculation.

```
ebola_subset = ebola.loc[:, ['Cases_Guinea', 'Cases_Liberia',
                             'Cases_SierraLeone', 'Cases_multiple']]
print(ebola_subset.head(n=10))
```

	Cases_Guinea	Cases_Liberia	Cases_SierraLeone	Cases_multiple
0	2776.0	Nan	10030.0	Nan
1	2775.0	Nan	9780.0	Nan
2	2769.0	8166.0	9722.0	20657.0
3	Nan	8157.0	Nan	Nan
4	2730.0	8115.0	9633.0	20478.0
5	2706.0	8018.0	9446.0	20170.0
6	2695.0	Nan	9409.0	Nan
7	2630.0	7977.0	9203.0	19810.0
8	2597.0	Nan	9004.0	Nan
9	2571.0	7862.0	8939.0	19372.0

You can see that the only time a value for `Cases_multiple` was calculated, was when there was no missing value for `Cases_Guinea`, `Cases_Liberia`, and `Cases_SierraLeone`. Calculations with missing values will typically return a missing value, unless the function or method called has a means to ignore missing values in its calculations.

An example of a built-in method that can ignore missing values is `mean` or `sum`. These functions will typically have a `skipna` parameter that will still calculate a value by skipping over the missing values.

```
# skipping missing values is True by default
print(ebola.Cases_Guinea.sum(skipna = True))

84729.0

print(ebola.Cases_Guinea.sum(skipna = False))

nan
```

5.5 Conclusion

It is rare to have a dataset without any missing values. It is important to know how to work with missing values because even when you are working with data that is complete, missing values

can still arise from your own data munging. Here I began some of the basic methods of the data analysis process that pertains to data validity. By looking at your data, and tabulating missing values, you can start the process of assessing if the data you are given is of enough quality for making decisions and inferences from your data.

Chapter 6. Tidy Data

6.1 Introduction

Hadley Wickham¹, one of the more prominent members in the R community, talks about *tidy data* in a paper² in the *Journal of Statistical Software*. Tidy data is a framework to structure datasets so they can be easily analyzed and visualized. It can be thought of as a goal one should aim for when cleaning data. Once you understand what tidy data is, it will make your data analysis, visualization, and collection much easier. What is *tidy* data? Hadley Wickham's paper defines it as such:

¹Hadley Wickham: <http://hadley.nz/>

²Tidy Data paper: <http://vita.had.co.nz/papers/tidy-data.pdf>

- each row is an observation
- each column is a variable
- each type of observational unit forms a table

This chapter will go through the various ways to tidy data from the *Tidy Data* paper.

Concept Map

Prior knowledge:

1. function and method calls
2. subsetting data
3. loops
4. list comprehension

This Chapter:

- reshaping data
1. unpivot/melt/gather
 2. pivot/cast/spread
 3. subsetting
 4. combining
 - (a) globbing

(b) concatenation

Objectives

This chapter will cover:

1. unpivot/melt/gather columns into rows
2. pivot/cast/spread rows into columns
3. normalize data by separating a dataframe into multiple tables
4. assembling data from multiple parts

6.2 Columns Contain Values, Not Variables

Data can have columns that contain values instead of variables. This is usually a convenient format for data collection and presentation.

6.2.1 Keep 1 Column Fixed

We can use the data on income and religion in the United States from the Pew Research Center to illustrate this example.

```
import pandas as pd
pew = pd.read_csv('../data/pew.csv')
```

If we look at the data, we can see that not every column is a variable. The values that relate to income are spread across multiple columns. The format shown is great when presenting data in a table, but for data analytics, the table needs to be reshaped such that we have a religion, income, and count variables.

```
# only show the first few columns
print(pew.iloc[:, 0:6])
```

	religion	<\$10k	\$10-20k	\$20-30k	\$30-40k	\
0	Agnostic	27	34	60	81	
1	Atheist	12	27	37	52	
2	Buddhist	27	21	30	34	
3	Catholic	418	617	732	670	
4	Dont know/refused	15	14	15	11	
5	Evangelical Prot	575	869	1064	982	
6	Hindu	1	9	7	9	
7	Historically Black Prot	228	244	236	238	
8	Jehovah's Witness	20	27	24	24	
9	Jewish	19	19	25	25	
10	Mainline Prot	289	495	619	655	
11	Mormon	29	40	48	51	
12	Muslim	6	7	9	10	
13	Orthodox	13	17	23	32	
14	Other Christian	9	7	11	13	
15	Other Faiths	20	33	40	46	
16	Other World Religions	5	2	3	4	
17	Unaffiliated	217	299	374	365	

	\$40-50k
0	76
1	35
2	33
3	638

```

4      10
5     881
6     11
7    197
8     21
9     30
10    651
11     56
12      9
13     32
14     13
15     49
16      2
17    341

```

This view of the data is also known as “wide” data. In order to turn it into the ‘long’ tidy data format, we will have to unpivot/melt/gather (depending on which statistical programming language you use) our dataframe. Pandas has a function called melt that will reshape the dataframe into a tidy format. melt takes a few parameters:

- **ic_vars** is a container (list, tuple, ndarray) that represents the variables that will remain **as-is**
- **value_vars** are the columns you want to melt down (or unpivot) By default it will melt all the columns not specified in the **ic_vars** parameter
- **var_name** is a string for the new column name when the **value_vars** is melted down. By default it will be called **variable**
- **value_name** is a string for the new column name that represents the values for the **var_name**. By default it will be called **value**

```

# we do not need to specify a value_vars since we want to pivot
# all the columns except for the 'religion' column
pew_long = pd.melt(pew, id_vars='religion')
print (pew_long.head())

```

	religion	variable	value
0	Agnostic	<\$10k	27
1	Atheist	<\$10k	12
2	Buddhist	<\$10k	27
3	Catholic	<\$10k	418
4	Dont know/refused	<\$10k	15

```

print(pew_long.tail())

```

	religion	variable	value
175	Orthodox	Don't know/refused	73
176	Other Christian	Don't know/refused	18
177	Other Faiths	Don't know/refused	71
178	Other World Religions	Don't know/refused	8
179	Unaffiliated	Don't know/refused	597

We can change the defaults so that the melted/unpivoted columns are named.

```

pew_long = pd.melt(pew,
                   id_vars='religion',
                   var_name='income',
                   value_name='count')
print(pew_long.head())

```

	religion	income	count
0	Agnostic	<\$10k	27
1	Atheist	<\$10k	12
2	Buddhist	<\$10k	27
3	Catholic	<\$10k	418
4	Dont know/refused	<\$10k	15

```

print(pew_long.tail())

```

	religion	income	count
175	Orthodox	Don't know/refused	73
176	Other Christian	Don't know/refused	18
177	Other Faiths	Don't know/refused	71
178	Other World Religions	Don't know/refused	8
179	Unaffiliated	Don't know/refused	597

6.2.2 Keep Multiple Columns Fixed

Not every dataset will have one column to hold still while you unpivot the rest. If you look at the Billboard dataset:

```
billboard = pd.read_csv('..../data/billboard.csv')

# look at the first few rows and columns
print(billboard.iloc[0:5, 0:16])
   year      artist          track    time date.entered \
0  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26
1  2000  2Ge+her  The Hardest Part Of ...  3:15  2000-09-02
2  2000  3 Doors Down           Kryptonite  3:53  2000-04-08
3  2000  3 Doors Down            Loser  4:24  2000-10-21
4  2000      504 Boyz          Wobble Wobble  3:35  2000-04-15

   wk1    wk2    wk3    wk4    wk5    wk6    wk7    wk8    wk9    wk10   wk11
0  87.0  82.0  72.0  77.0  87.0  94.0  99.0  NaN  NaN  NaN  NaN
1  91.0  87.0  92.0  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
2  81.0  70.0  68.0  67.0  66.0  57.0  54.0  53.0  51.0  51.0  51.0
3  76.0  76.0  72.0  69.0  67.0  65.0  55.0  59.0  62.0  61.0  61.0
4  57.0  34.0  25.0  17.0  31.0  36.0  49.0  53.0  57.0  64.0
```

You can see here that each week is it's own column. Again, there is nothing *wrong* with this form of data. It maybe easy to enter the data in this form, and it is much quicker to understand when presented in a table. However, there may be a time when you will need to melt the data. An example would be when plotting weekly ratings in a faceted plot, since the facet variable needs to be a columns in the dataframe.

```
billboard_long = pd.melt(
    billboard,
    id_vars=['year', 'artist', 'track', 'time', 'date.entered'],
    var_name='week',
    value_name='rating')

print(billboard_long.head())
   year      artist          track    time date.entered \
0  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26
1  2000  2Ge+her  The Hardest Part Of ...  3:15  2000-09-02
2  2000  3 Doors Down           Kryptonite  3:53  2000-04-08
3  2000  3 Doors Down            Loser  4:24  2000-10-21
4  2000      504 Boyz          Wobble Wobble  3:35  2000-04-15

   week  rating
0  wk1    87.0
1  wk1    91.0
2  wk1    81.0
3  wk1    76.0
4  wk1    57.0

print(billboard_long.tail())
   year      artist          track    time date.entered \
24087  2000  Yankee Grey  Another Nine Minutes  3:10
24088  2000  Yearwood, Trisha  Real Live Woman  3:55
24089  2000  Ying Yang Twins  Whistle While You Tw...  4:19
24090  2000  Zombie Nation  Kernkraft 400  3:30
24091  2000  matchbox twenty          Bent  4:12

   date.entered week  rating
24087  2000-04-29  wk76  NaN
24088  2000-04-01  wk76  NaN
24089  2000-03-18  wk76  NaN
```

```
24090    2000-09-02  wk76      NaN
24091    2000-04-29  wk76      NaN
```

6.3 Columns Contain Multiple Variables

There will be times when the columns represent multiple variables. This is something that is common when working with health data. To illustrate this, let's look at the Ebola dataset.

```
ebola = pd.read_csv('../data/country_timeseries.csv')
print(ebola.columns)

Index(['Date', 'Day', 'Cases_Guinea', 'Cases_Liberia',
       'Cases_SierraLeone', 'Cases_Nigeria', 'Cases_Senegal',
       'Cases_UnitedStates', 'Cases_Spain', 'Cases_Mali',
       'Deaths_Guinea', 'Deaths_Liberia', 'Deaths_SierraLeone',
       'Deaths_Nigeria', 'Deaths_Senegal', 'Deaths_UnitedStates',
       'Deaths_Spain', 'Deaths_Mali'],
      dtype='object')

# print select rows
print(ebola.iloc[:5, [0, 1, 2, 3, 10, 11]])

      Date   Day  Cases_Guinea  Cases_Liberia  Deaths_Guinea \
0    1/5/2015  289        2776.0           NaN          1786.0
1    1/4/2015  288        2775.0           NaN          1781.0
2    1/3/2015  287        2769.0         8166.0          1767.0
3    1/2/2015  286           NaN         8157.0           NaN
4  12/31/2014  284        2730.0         8115.0          1739.0

  Deaths_Liberia
0             NaN
1             NaN
2            3496.0
3            3496.0
4            3471.0
```

The column names `Cases_Guinea` and `Deaths_Guinea` actually contain 2 variables. The individual status, cases and deaths, and the county, Guinea. The data is also in wide format that needs to be unpivoted.

```
ebola_long = pd.melt(ebola, id_vars=['Date', 'Day'])
print(ebola_long.head())

      Date   Day     variable  value
0    1/5/2015  289  Cases_Guinea  2776.0
1    1/4/2015  288  Cases_Guinea  2775.0
2    1/3/2015  287  Cases_Guinea  2769.0
3    1/2/2015  286  Cases_Guinea      NaN
4  12/31/2014  284  Cases_Guinea  2730.0

print(ebola_long.tail())

      Date   Day     variable  value
1947  3/27/2014    5  Deaths_Mali    NaN
1948  3/26/2014    4  Deaths_Mali    NaN
1949  3/25/2014    3  Deaths_Mali    NaN
1950  3/24/2014    2  Deaths_Mali    NaN
1951  3/22/2014    0  Deaths_Mali    NaN
```

6.3.1 Split and Add Columns Individually (Simple Method)

Conceptually, the column of interest can be split by the underscore, `_`. The first part will be the new status column, and the second part will be the new country column. This will require some string parsing and splitting in Python (more on this in [Chapter 8](#)). In Python, a string is an object, similar to how Pandas has a `Series` and `DataFrame` object. [Chapter 2](#) showed how `Series` can have various methods, such as `mean`, and `DataFrames` have methods such as `to_csv`. Strings have

methods as well, in this case we will use the split method that takes a string and will split the string up by a given delimiter. By default split will split the string by a space, but we can pass in the underscore, _, in our example. In order to get access to the string methods, we need to use the str accessor (See [Chapter 8](#) for more on strings). This will give us access to the python string methods and work across the entire column.

```
# get the variable column
# access the string methods
# and split the column by a delimiter
variable_split = ebola_long.variable.str.split('_')

print(variable_split[:5])

0    [Cases, Guinea]
1    [Cases, Guinea]
2    [Cases, Guinea]
3    [Cases, Guinea]
4    [Cases, Guinea]
Name: variable, dtype: object

print(variable_split[-5:])
1947    [Deaths, Mali]
1948    [Deaths, Mali]
1949    [Deaths, Mali]
1950    [Deaths, Mali]
1951    [Deaths, Mali]
Name: variable, dtype: object
```

We can see that after we split on the underscore, the values are returned in a list. We know it's a list because that's how the split method works,³ but the visual cue is that the results are surrounded by square brackets.

³String `split` documentation: <https://docs.python.org/3.6/library/stdtypes.html#str.split>

```
# the entire container
print(type(variable_split))

<class 'pandas.core.series.Series'>

# the first element in the container
print(type(variable_split[0]))

<class 'list'>
```

Now that we have column split into the various pieces, the next step is to assign them to a new column. But first, we need to extract all the 0 index elements for the `status` column and the 1 index elements for the country column. To do so, we need to access the string methods again, and then use the get method to get the index we want for each row.

```
status_values = variable_split.str.get(0)
country_values = variable_split.str.get(1)

print(status_values[:5])

0    Cases
1    Cases
2    Cases
3    Cases
4    Cases
Name: variable, dtype: object

print(status_values[-5:])

1947    Deaths
1948    Deaths
1949    Deaths
1950    Deaths
1951    Deaths
```

```

Name: variable, dtype: object
print(country_values[:5])

0    Guinea
1    Guinea
2    Guinea
3    Guinea
4    Guinea
Name: variable, dtype: object
print(country_values[-5:])

1947   Mali
1948   Mali
1949   Mali
1950   Mali
1951   Mali
Name: variable, dtype: object

```

Now that we have the vectors we want, we can add them to our dataframe

```

ebola_long['status'] = status_values
ebola_long['country'] = country_values

print(ebola_long.head())

```

	Date	Day	variable	value	status	country
0	1/5/2015	289	Cases_Guinea	2776.0	Cases	Guinea
1	1/4/2015	288	Cases_Guinea	2775.0	Cases	Guinea
2	1/3/2015	287	Cases_Guinea	2769.0	Cases	Guinea
3	1/2/2015	286	Cases_Guinea	NaN	Cases	Guinea
4	12/31/2014	284	Cases_Guinea	2730.0	Cases	Guinea

6.3.2 Split and Combine in a Single Step (Simple Method)

We can do the same thing as before, and exploit the fact that the vector returned is in the same order as our data. We can concatenate ([Chapter 4](#)) the new vector or our original data.

```

variable_split = ebola_long.variable.str.split('_', expand=True)
variable_split.columns = ['status', 'country']
ebola_parsed = pd.concat([ebola_long, variable_split], axis=1)

print(ebola_parsed.head())

      Date  Day      variable    value status country status \
0  1/5/2015  289  Cases_Guinea  2776.0  Cases  Guinea  Cases
1  1/4/2015  288  Cases_Guinea  2775.0  Cases  Guinea  Cases
2  1/3/2015  287  Cases_Guinea  2769.0  Cases  Guinea  Cases
3  1/2/2015  286  Cases_Guinea    NaN  Cases  Guinea  Cases
4 12/31/2014  284  Cases_Guinea  2730.0  Cases  Guinea  Cases

      country
0    Guinea
1    Guinea
2    Guinea
3    Guinea
4    Guinea

print(ebola_parsed.tail())

      Date  Day      variable    value status country status \
1947 3/27/2014    5  Deaths_Mali    NaN Deaths    Mali Deaths
1948 3/26/2014    4  Deaths_Mali    NaN Deaths    Mali Deaths
1949 3/25/2014    3  Deaths_Mali    NaN Deaths    Mali Deaths
1950 3/24/2014    2  Deaths_Mali    NaN Deaths    Mali Deaths
1951 3/22/2014    0  Deaths_Mali    NaN Deaths    Mali Deaths

      country
1947    Mali
1948    Mali
1949    Mali

```

```
1950    Mali
1951    Mali
```

6.3.3 Split and Combine in a Single Step (More Complicated Method)

We can do the same thing as before, and exploit the fact that the vector returned is in the same order as our data. We can concatenate ([Chapter 4](#)) the new vector or our original data.

We can accomplish the same result in a single step by taking advantage of the fact that the split results return a list of 2 elements, where each element will be a new column. We can combine the list of split items with the built-in `zip` function. `zip` takes a set of iterators (lists, tuples, etc.) and creates a new container that is made of the input iterators, but each new container created is the same index from the input containers. For example, if we have 2 lists of values:

```
constants = ['pi', 'e']
values = ['3.14', '2.718']
```

we can zip the values together as such:

```
# we have to call list on the zip function
# to show the contents of the zip object
# this is because in Python 3 zip returns an iterator.
print(list(zip(constants, values)))

[('pi', '3.14'), ('e', '2.718')]
```

Each element now has the constant matched with its corresponding value. Conceptually, each container is like a side of a zipper. When we `zip` the containers, the indices are matched up and returned.

Another way to visualize what `zip` is doing is taking each container passed into `zip` and stacking them on top of each other (think row wise concatenation in [Section 4.3.1](#)) creating a dataframe of sorts. `zip` then returns the values column-by-column in a tuple.

We can use the same `ebolaJong.variable.str.split('')` to split the values in the column. However, since the result is already a container (a `Series` object), we need to unpack it such that it is the contents of the container (each status-country list) not the container itself (the series)

The asterisk, `*`, in python is used to unpack containers⁴. When we `zip` the unpacked containers, it is the same as creating the status values and country values above. We can then assign the vectors to the columns simultaneously using multiple assignment ([Appendix Q](#)).

⁴Unpacking Argument Lists: <https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists>

```
ebola_long['status'], ebola_long['country'] = \
    zip(*ebola_long.variable.str.split('_'))

print(ebola_long.head())

      Date  Day   variable   value status country
0  1/5/2015  289  Cases_Guinea  2776.0  Cases  Guinea
1  1/4/2015  288  Cases_Guinea  2775.0  Cases  Guinea
2  1/3/2015  287  Cases_Guinea  2769.0  Cases  Guinea
3  1/2/2015  286  Cases_Guinea     NaN  Cases  Guinea
4 12/31/2014  284  Cases_Guinea  2730.0  Cases  Guinea
```

6.4 Variables in Both Rows and Columns

At times data will be in a shape where variables are in both rows and columns. That is, some combination of the previous sections of this chapter. Most of the methods to tidy up the data have already been presented. What is left to show is what happens if a column of data actually holds 2 variables instead of 1. In this case, we will have to pivot or cast the variable into separate columns.

```
weather = pd.read_csv('../data/weather.csv')
print(weather.iloc[:5, :11])

    id  year  month element  d1  d2  d3  d4  d5  d6  d7
0  MX17004  2010        1  tmax  NaN  NaN  NaN  NaN  NaN  NaN
1  MX17004  2010        1  tmin  NaN  NaN  NaN  NaN  NaN  NaN
2  MX17004  2010        2  tmax  NaN  27.3  24.1  NaN  NaN  NaN
3  MX17004  2010        2  tmin  NaN  14.4  14.4  NaN  NaN  NaN
4  MX17004  2010        3  tmax  NaN  NaN  NaN  NaN  32.1  NaN  NaN
```

In the weather data, there are minimum and maximum (`tmin` and `tmax` values in the `element` column, respectively) temperatures recorded for each day (`d1`, `d2`, `d3`) of the month (month). The `element` column contains variables that need to be casted/pivoted to become new columns, and the day variables, need to be melted into row values. Again, there is nothing wrong with the data in the current format. It is simply not in a shape for analysis, but can be helpful when presenting data in reports. Let's first melt/unpivot the day values.

```
weather_melt = pd.melt(weather,
                       id_vars=['id', 'year', 'month', 'element'],
                       var_name='day',
                       value_name='temp')
print(weather_melt.head())

    id  year  month element  day  temp
0  MX17004  2010        1  tmax  d1  NaN
1  MX17004  2010        1  tmin  d1  NaN
2  MX17004  2010        2  tmax  d1  NaN
3  MX17004  2010        2  tmin  d1  NaN
4  MX17004  2010        3  tmax  d1  NaN

print(weather_melt.tail())

    id  year  month element  day  temp
677  MX17004  2010       10  tmin  d31  NaN
678  MX17004  2010       11  tmax  d31  NaN
679  MX17004  2010       11  tmin  d31  NaN
680  MX17004  2010       12  tmax  d31  NaN
681  MX17004  2010       12  tmin  d31  NaN
```

The next, we need to pivot up the variables stored in the `element` column. This is also referred to as casting or spreading in other statistical languages. One of the main differences from `pivot_table` and `melt`, is that `melt` is a function within pandas and `pivot_table` is a method we call on a `DataFrame` object.

```
weather_tidy = weather_melt.pivot_table(
    index=['id', 'year', 'month', 'day'],
    columns='element',
    values='temp')
```

If we look at the pivoted table, we will notice that each value in the `element` column is now a separate column. We can leave it in its current state, but we can also flatten the hierarchical columns

```
weather_tidy_flat = weather_tidy.reset_index()
print(weather_tidy_flat.head())
```

	element	id	year	month	day	tmax	tmin
0	MX17004	2010	1	d1	NaN	NaN	
1	MX17004	2010	1	d10	NaN	NaN	

```

2      MX17004  2010      1    d11    NaN    NaN
3      MX17004  2010      1    d12    NaN    NaN
4      MX17004  2010      1    d13    NaN    NaN

```

Likewise, we can perform those methods without the intermediate dataframe as such:

```

weather_tidy = weather_melt.\
    pivot_table(
        index=['id', 'year', 'month', 'day'],
        columns='element',
        values='temp').\
    reset_index()

print(weather_tidy.head())

```

element	id	year	month	day	tmax	tmin	
0	MX17004	2010		1	d1	NaN	NaN
1	MX17004	2010		1	d10	NaN	NaN
2	MX17004	2010		1	d11	NaN	NaN
3	MX17004	2010		1	d12	NaN	NaN
4	MX17004	2010		1	d13	NaN	NaN

6.5 Multiple Observational Units in a Table (Normalization)

One of the simplest ways of knowing if multiple observational units are represented in a table is by looking at each of the rows, and taking note of any cells or values that are being repeated from row to row. This is very common in government education administration data where student demographics are reported for each student for each year the student is enrolled.

If we look at the billboard data we cleaned in [Section 6.2.2](#):

```

print(billboard_long.head())

      year      artist          track   time date.entered \
0  2000       2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26
1  2000  2Ge+her  The Hardest Part Of ...  3:15  2000-09-02
2  2000  3 Doors Down           Kryptonite  3:53  2000-04-08
3  2000  3 Doors Down            Loser  4:24  2000-10-21
4  2000     504 Boyz            Wobble  3:35  2000-04-15

      week  rating
0    wk1    87.0
1    wk1    91.0
2    wk1    81.0
3    wk1    76.0
4    wk1    57.0

```

and if we subset ([Section 2.4.1](#)) on a particular track:

```

print(billboard_long[billboard_long.track == 'Loser'].head())

      year      artist  track   time date.entered week  rating
3  2000  3 Doors Down  Loser  4:24  2000-10-21  wk1    76.0
320 2000  3 Doors Down  Loser  4:24  2000-10-21  wk2    76.0
637 2000  3 Doors Down  Loser  4:24  2000-10-21  wk3    72.0
954 2000  3 Doors Down  Loser  4:24  2000-10-21  wk4    69.0
1271 2000  3 Doors Down  Loser  4:24  2000-10-21  wk5    67.0

```

We can see that this table actually holds 2 types of data, the track information and weekly ranking. It would be better to store the track information in a separate table. This way, the information stored in the `year`, `artist`, `track`, and `time` columns are not repeated in the dataset. This is particularly important if the data is manually entered. By repeating the same values over and over during data entry, one risks having inconsistent data.

What we should do in this case is to have the `year`, `artist`, `track`, `time`, and `date.entered` in a new

dataframe and each unique set of values be assigned a unique ID. We can then use this unique ID in a second dataframe that represents a song, date, week number, and ranking. This entire process can be thought of as reversing the steps in concatenating and merging data in [Chapter 4](#).

```
billboard_songs = billboard_long[['year', 'artist', 'track', 'time']]
print(billboard_songs.shape)

(24092, 4)
```

We know there are duplicate entries in this dataframe, so we need to drop the duplicate

```
billboard_songs = billboard_songs.drop_duplicates()
print(billboard_songs.shape)

(317, 4)
```

We can then assign a unique value to each row of data.

```
billboard_songs['id'] = range(len(billboard_songs))
print(billboard_songs.head(n=10))

   year      artist          track    time  id
0  2000      2 Pac  Baby Don't Cry (Keep...  4:22  0
1  2000  2Ge+her  The Hardest Part Of ...  3:15  1
2  2000      3 Doors Down           Kryptonite  3:53  2
3  2000      3 Doors Down           Loser  4:24  3
4  2000      504 Boyz           Wobble Wobble  3:35  4
5  2000      98^0  Give Me Just One Nig...  3:24  5
6  2000      A*Teens           Dancing Queen  3:44  6
7  2000      Aaliyah           I Don't Wanna  4:15  7
8  2000      Aaliyah           Try Again  4:03  8
9  2000  Adams, Yolanda        Open My Heart  5:30  9
```

Now that we have a separate dataframe about songs, we can use the newly created id column to match a song to its weekly ranking.

```
# Merge the song dataframe to the original dataset
billboard_ratings = billboard_long.merge(
    billboard_songs, on=['year', 'artist', 'track', 'time'])
print(billboard_ratings.shape)

(24092, 8)

print(billboard_ratings.head())

   year      artist          track    time date.entered  week \
0  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26  wk1
1  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26  wk2
2  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26  wk3
3  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26  wk4
4  2000      2 Pac  Baby Don't Cry (Keep...  4:22  2000-02-26  wk5

   rating  id
0    87.0  0
1    82.0  0
2    72.0  0
3    77.0  0
4    87.0  0
```

Finally, we subset the columns to the ones we want in our ratings dataframe.

```
billboard_ratings = \
    billboard_ratings[['id', 'date.entered', 'week', 'rating']]
print(billboard_ratings.head())

   id date.entered  week  rating
0   0  2000-02-26  wk1    87.0
1   0  2000-02-26  wk2    82.0
2   0  2000-02-26  wk3    72.0
3   0  2000-02-26  wk4    77.0
```

6.6 Observational Units Across Multiple Tables

The last bit of data tidying involves having the same type of data being spread across multiple datasets. This has already been covered in [Chapter 4](#) when we discussed data concatenation and merging. A reason why data would be split across multiple files would be size. By splitting up data into various parts, each part would be smaller. This may be good to share data on the Internet or email since many services limit the size of a file that can be opened or shared. Another reason why a dataset would be split into multiple parts would be from the data collection process. For example, a separate data containing stock information could be created for each day.

Since merging and concatenation has already been covered, this example will show techniques on how to quickly load multiple data sources and assemble them together.

The Unified New York City Taxi and Uber Data is a good example to show this. The entire dataset has over 1.3 billion taxi and Uber trips from New York City, and has over 140 files. Here for illustration purposes, we only work with 5 of these data files. When the same data is broken into multiple parts, they typically have a structured naming pattern associated with it.

First let's download the data. Do not worry too much about the details in the following block of code. The `raw_data_urls.txt` file contain a list of URLs where each URL is the download link to a part of the taxi data. We begin by opening and reading the file, and iterating through each line of the file (i.e., each data URL). We only download the first 5 data sets since the files are fairly large, and can take a while to download. We use some string manipulation ([Chapter 8](#)) to create the path where the data will be saved, and use the `urllib` library to download our data.

```
import os
import urllib

# code to download the data
# download only the first 5 datasets from the list of files
with open('../data/raw_data_urls.txt', 'r') as data_urls:
    for line, url in enumerate(data_urls):
        if line == 5:
            break
        fn = url.split('/')[-1].strip()
        fp = os.path.join('..', 'data', fn)
        print(url)
        print(fp)
        urllib.request.urlretrieve(url, fp)
```

In this example, all of the raw taxi trips have the pattern `fhv_tripdata_YYYY_XX.csv`, where `YYYY` represents the year (e.g., 2015), and `XX` represents the part number. We can use the a simple pattern matching function from the `glob` library in Python to get a list of all the filenames that match a particular pattern.

```
import glob
# get a list of the csv files from the nyc-taxi data folder
nyc_taxi_data = glob.glob('../data/fhv_*')
print(nyc_taxi_data)

[ '../data/fhv_tripdata_2015-04.csv',
  '../data/fhv_tripdata_2015-05.csv',
  '../data/fhv_tripdata_2015-03.csv',
  '../data/fhv_tripdata_2015-01.csv',
  '../data/fhv_tripdata_2015-02.csv']
```

Now that we have a list of filenames we want to load, we can load each file into a dataframe. We can choose to load each file individually like we have been doing so far.

```
taxi1 = pd.read_csv(nyc_taxi_data[0])
taxi2 = pd.read_csv(nyc_taxi_data[1])
taxi3 = pd.read_csv(nyc_taxi_data[2])
taxi4 = pd.read_csv(nyc_taxi_data[3])
taxi5 = pd.read_csv(nyc_taxi_data[4])
```

We can look at our data and see how they can be nicely stacked (concatenated) on top of each other.

```
print(taxi1.head(n=2))
print(taxi2.head(n=2))
print(taxi3.head(n=2))
print(taxi4.head(n=2))
print(taxi5.head(n=2))

   Dispatching_base_num      Pickup_date  locationID
0              B00001  2015-04-01 04:30:00      NaN
1              B00001  2015-04-01 06:00:00      NaN
   Dispatching_base_num      Pickup_date  locationID
0              B00001  2015-05-01 04:30:00      NaN
1              B00001  2015-05-01 05:00:00      NaN
   Dispatching_base_num      Pickup_date  locationID
0              B00029  2015-03-01 00:02:00    213.0
1              B00029  2015-03-01 00:03:00     51.0
   Dispatching_base_num      Pickup_date  locationID
0              B00013  2015-01-01 00:30:00      NaN
1              B00013  2015-01-01 01:22:00      NaN
   Dispatching_base_num      Pickup_date  locationID
0              B00013  2015-02-01 00:00:00      NaN
1              B00013  2015-02-01 00:01:00      NaN
```

We can concatenate them just like in [Chapter 4](#).

```
# shape of each dataframe
print(taxi1.shape)
print(taxi2.shape)
print(taxi3.shape)
print(taxi4.shape)
print(taxi5.shape)

(3917789, 3)
(4296067, 3)
(3281427, 3)
(2746033, 3)
(3126401, 3)

# concatenate the dataframes together
taxi = pd.concat([taxi1, taxi2, taxi3, taxi4, taxi5])

# shape of final concatenated taxi data
print(taxi.shape)

(17367717, 3)
```

However, manually saving each dataframe will get tedious when there are many parts the data is split into. Instead we can automate the process using loops and list comprehensions.

6.6.1 Load Multiple Files Using a Loop

The easier way is to first create an empty list, use a loop to iterate through each of the csv files, load the csv file into a pandas dataframe, and finally append the dataframe to the list.

The final type of data we want is a list of dataframes because the concat function takes a list of dataframes to concatenate.

```

# create an empty list to append to
list_taxi_df = []

# loop though each csv filename
for csv_filename in nyc_taxi_data:
    # you can choose to print the filename for debugging
    # print(csv_filename)

    # load the csv file into a dataframe
    df = pd.read_csv(csv_filename)

    # append the dataframe to the list that will hold the dataframes
    list_taxi_df.append(df)

# print the length of the dataframe
print(len(list_taxi_df))

5

# type of the first element
print(type(list_taxi_df[0]))

<class 'pandas.core.frame.DataFrame'>

# look at the head of the first dataframe
print(list_taxi_df[0].head())

   Dispatching_base_num      Pickup_date  locationID
0              B00001  2015-04-01 04:30:00        NaN
1              B00001  2015-04-01 06:00:00        NaN
2              B00001  2015-04-01 06:00:00        NaN
3              B00001  2015-04-01 06:00:00        NaN
4              B00001  2015-04-01 06:15:00        NaN

```

Now that we have a list of dataframes, we can concatenate them.

```

taxi_loop_concat = pd.concat(list_taxi_df)
print(taxi_loop_concat.shape)

(17367717, 3)

# Did we get the same results as the manual load and concatenation?
print(taxi.equals(taxi_loop_concat))

True

```

6.6.2 Load Multiple Files Using a List Comprehension

Python has an idiom for looping though something and adding it to a list. It is called a list comprehension. The loop above which, I will show again without the comments, can be written in a list comprehension ([Appendix N](#)).

```

# the loop code without comments
list_taxi_df = []
for csv_filename in nyc_taxi_data:
    df = pd.read_csv(csv_filename)
    list_taxi_df.append(df)

# same code in a list comprehension
list_taxi_df_comp = [pd.read_csv(data) for data in nyc_taxi_data]

```

The result from our list comprehension is a list, just like the loop example above.

```

print(type(list_taxi_df_comp))

<class 'list'>

```

Finally, we can concatenate the results just like before.

```
taxi_loop_concat_comp = pd.concat(list_taxi_df_comp)

# are the concatenated dataframes the same?
print(taxi_loop_concat_comp.equals(taxi_loop_concat))

True
```

6.7 Conclusion

Here I showed you how we can reshape data to a format that is conducive for data analysis, visualization, and collection. We followed Hadley Wickham's *Tidy Data* paper to show the various functions and methods to reshape our data. This is an important skill since various functions will need data in a certain shape, tidy or not, in order to work. Knowing how to reshape your data will be an important still as a data scientist and analyst.

Part III: Data Munging

[Chapter 7, “Data Types,”](#) Converting data types.

[Chapter 8, “Strings and Text Data,”](#) Working with strings.

[Chapter 9, “Apply,”](#) Applying Functions.

[Chapter 10, “Groupby operations: split-apply-combine,”](#) Aggregate, transform, and filter data.

[Chapter 11, “The datetime Data Type,”](#) Working with dates and times.

Chapter 7. Data Types

7.1 Introduction

Data types determine what can and cannot be done to a variable (i.e., column). For example, when numeric data types are added together, the result will be a sum of the values, whereas if strings (in pandas they are called `object`) are added, the strings will be concatenated together.

This chapter will be a quick overview of the various data types you may encounter in pandas, and how you may want to convert from one data type to another.

Objectives

This chapter will cover:

1. find the data types of columns in a dataframe
2. convert between various data types
3. learn about the categorical data type
4. working with dates and times are covered in [Chapter 11](#)

7.2 Data Types

We'll be using the built-in `tips` dataset from `seaborn`

```
import pandas as pd
import seaborn as sns

tips = sns.load_dataset("tips")
```

To get a list of data types stored in each column of our dataframe, we call the `dtypes` attribute, as shown previously in [Section 1.2](#).

```
print(tips.dtypes)

total_bill      float64
tip            float64
sex            category
smoker         category
day            category
time           category
size           int64
dtype: object
```

[Table 1-1](#) in [Chapter 1](#) listed the various types of data that can be stored in a pandas column. In our dataset, you see we have `int64`, `float64`, and `category`. The `int64` and `float64` represent numeric values without and with decimal points, respectively. The number following the numeric data type represent the number of bits of information that will be stored for that particular number.

The `category` data type represents categorical variables. It differs from the generic `object` data type that stores arbitrary strings. We will cover the differences later on in this chapter. Since the `tips` dataset is a fully prepared and cleaned dataset, variables that store strings were saved as a category.

7.3 Converting Types

The data type that is stored in a column will govern what kinds of functions and calculations you can perform on them. So it's important to know how to convert between data types.

We'll be showing you how to convert between one data type to another, but keep in mind, you need not do all your data type conversions when you first get your data. Remember, data analytics is not a linear process, and you can choose to convert types on-the-fly as needed. I briefly showed you this in Chapter [2.5.2](#). When we converted a date value into just the number of years.

7.3.1 Converting to String Object

In our `tips` data, the `sex`, `smoker`, `day`, and `time` variables are stored as a `category`. In general, it's much easier to work with string `object` types when the variable is not a numeric number.

Some datasets may have an `id` column where the `id` may be stored as a number. However, but has no meaning if you perform a calculation on it, e.g., the mean. Unique identifiers or `id` numbers are typically coded this way and you may want to convert them to string `object` types depending on what you need.

To convert values into strings, we use the `astype`¹ method on the column. `astype` takes a parameter, `dtype`, that will be the new data type the column will take on. In this case, we want to convert the `sex` variable to a string `object`, `str`.

¹Converting `Series` types: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.astype.html>

```
tips['sex_str'] = tips['sex'].astype(str)
```

Python has built-in `str`, `float`, `int`, `complex`, and `bool` types. However, you can also specify any `dtype` from the `numpy` library.

If we look at the `dtypes` now, you will see the `unique` key will now have a `dtype` of `object`

```
print(tips.dtypes)
total_bill      float64
tip            float64
sex           category
smoker        category
day           category
time          category
size          int64
sex_str       object
dtype: object
```

7.3.2 Converting to Numeric Values

The `astype` method is a generic function that can be used to convert any column in a dataframe to

another `dtype`.

Remember, each column is a pandas Series object, that's why the `astype` documentation is listed under `pandas.Series.astype`. The example here shows how to change the type of a dataframe column, but if you are working with a `Series` object, you can use the same `astype` method to convert the `Series` as well.

We can provide any built-in or `numpy` type to the `astype` method to convert the `dtype` of the column. For example, if we wanted to convert the `total_bill` column to a string `object` and back to its original `float64`, we can pass in `str` and `float` into `astype`, respectively.

```
# convert total_bill into a string
tips['total_bill'] = tips['total_bill'].astype(str)
print(tips.dtypes)

total_bill      object
tip            float64
sex            category
smoker         category
day            category
time           category
size           int64
sex_str        object
dtype: object

# convert it back to a float
tips['total_bill'] = tips['total_bill'].astype(float)
print(tips.dtypes)

total_bill      float64
tip            float64
sex            category
smoker         category
day            category
time           category
size           int64
sex_str        object
dtype: object
```

7.3.2.1 `to_numeric`

When converting variables into numeric values (e.g., `int`, `float`), pandas also has a `to_numeric` function that handles non-numeric values better.

Since each column in a dataframe has to have the same `dtype`, there will be times when you will have a numeric column, but some values will be strings. A common example would be a numeric column but instead of a `NaN` value that represents a missing value in `pandas`, a dataset may have the string '`missing`' or '`null`' in it instead. This would make the entire column a string `object` type instead of a numeric type.

Let's subset our `tips` dataframe and also put in a '`missing`' value in the `total_bill` column to illustrate how the `to_numeric` function works.

```
# subset the tips data
tips_sub_miss = tips.head(10)

# assign some 'missing' values
tips_sub_miss.loc[[1, 3, 5, 7], 'total_bill'] = 'missing'

print(tips_sub_miss)

   total_bill    tip     sex smoker  day   time  size sex_str
0      16.99  1.01  Female    No   Sun Dinner     2  Female
1      missing  1.66    Male    No   Sun Dinner     3    Male
```

```

2    21.01  3.50   Male    No  Sun Dinner      3   Male
3  missing  3.31   Male    No  Sun Dinner      2   Male
4    24.59  3.61 Female   No  Sun Dinner      4 Female
5  missing  4.71   Male    No  Sun Dinner      4   Male
6    8.77  2.00   Male    No  Sun Dinner      2   Male
7  missing  3.12   Male    No  Sun Dinner      4   Male
8    15.04  1.96   Male    No  Sun Dinner      2   Male
9    14.78  3.23   Male    No  Sun Dinner      2   Male

```

If we looked at the `dtypes`, you will see the `total_bill` column will now be a string `object`.

```

print(tips_sub_miss.dtypes)

total_bill      object
tip            float64
sex            category
smoker         category
day            category
time           category
size           int64
sex_str        object
dtype: object

```

If we used the `astype` method to convert the column back to a `float`, we will now get an error because pandas does not know how to convert '`missing`' into a `float`.

```

# this will cause an error
tips_sub_miss['total_bill'].astype(float)

Traceback (most recent call last):
  File "<ipython-input-1-98a540fd2fa7>", line 2, in <module>
    tips_sub_miss['total_bill'].astype(float)
ValueError: could not convert string to float: 'missing'

```

If we use the `to_numeric` function from the `pandas` library, we get a similar error.

```

# this will cause an error
pd.to_numeric(tips_sub_miss['total_bill'])

Traceback (most recent call last):
  File "pandas/_libs/src/inference.pyx", line 1021, in
pandas._libs.lib.maybe_convert_numeric (pandas/_libs/lib.c:56156)
ValueError: Unable to parse string "missing"

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<ipython-input-1-fcf2f6d55ed>", line 2, in <module>
    pd.to_numeric(tips_sub_miss['total_bill'])
ValueError: Unable to parse string "missing" at position 1

```

The `to_numeric` has a parameter called `errors` that governs what happens when the function encounters a value that it is unable to convert to a numeric value. By default, this value is set to '`raise`', meaning if it does encounter a value it is unable to convert to a numeric value, it will '`raise`' an error.

From the documentation², we can give `errors` three possible values:

²Converting to numeric values: https://pandas-docs.github.io/pandas-docs-travis/generated/pandas.to_numeric.html

1. '`raise`' (default) will raise an error if it cannot convert to a numeric value
2. '`coerce`' will return a `NaN` for values it cannot convert to numeric

3. 'ignore' will return the vector without converting the column into a numeric (i.e., will do nothing)

Going out of order from the documentation, if we pass `errors` the 'ignore' value, nothing will change in our column. But we also do not get an error.

```
tips_sub_miss['total_bill'] = pd.to_numeric(  
    tips_sub_miss['total_bill'], errors='ignore')  
  
print(tips_sub_miss)  
  
   total_bill     tip      sex smoker  day    time    size  sex_str  
0      16.99  1.01  Female    No  Sun  Dinner      2  Female  
1    missing  1.66   Male    No  Sun  Dinner      3   Male  
2      21.01  3.50   Male    No  Sun  Dinner      3   Male  
3    missing  3.31   Male    No  Sun  Dinner      2   Male  
4      24.59  3.61 Female    No  Sun  Dinner      4 Female  
5    missing  4.71   Male    No  Sun  Dinner      4   Male  
6      8.77  2.00   Male    No  Sun  Dinner      2   Male  
7    missing  3.12   Male    No  Sun  Dinner      4   Male  
8      15.04  1.96   Male    No  Sun  Dinner      2   Male  
9      14.78  3.23   Male    No  Sun  Dinner      2   Male  
  
print(tips_sub_miss.dtypes)  
  
total_bill      object  
tip        float64  
sex       category  
smoker     category  
day        category  
time       category  
size      int64  
sex_str    object  
dtype: object
```

However if we pass in the 'coerce' value, we will get `NaN` values for the 'missing' string.

```
tips_sub_miss['total_bill'] = pd.to_numeric(  
    tips_sub_miss['total_bill'], errors='coerce')  
  
print(tips_sub_miss)  
  
   total_bill     tip      sex smoker  day    time    size  sex_str  
0      16.99  1.01  Female    No  Sun  Dinner      2  Female  
1      NaN  1.66   Male    No  Sun  Dinner      3   Male  
2      21.01  3.50   Male    No  Sun  Dinner      3   Male  
3      NaN  3.31   Male    No  Sun  Dinner      2   Male  
4      24.59  3.61 Female    No  Sun  Dinner      4 Female  
5      NaN  4.71   Male    No  Sun  Dinner      4   Male  
6      8.77  2.00   Male    No  Sun  Dinner      2   Male  
7      NaN  3.12   Male    No  Sun  Dinner      4   Male  
8      15.04  1.96   Male    No  Sun  Dinner      2   Male  
9      14.78  3.23   Male    No  Sun  Dinner      2   Male  
  
print(tips_sub_miss.dtypes)  
  
total_bill      float64  
tip        float64  
sex       category  
smoker     category  
day        category  
time       category  
size      int64  
sex_str    object  
dtype: object
```

This is a useful trick when you know a column must contain numeric values, and for some reason non-numeric values are in the data.

7.3.2.2 `to_numeric` downcast

The `to_numeric` function has another parameter called `downcast`, that allows you to change (i.e., ‘downcast’) the numeric `dtype` to the smallest numerical `dtype` after a column (or vector) has been successfully converted to a numeric vector.

By default the value is set to `None`, but other possible values are '`integer`', '`signed`', '`unsigned`', and '`float`'.

Compare the `dtypes` after we provide a `downcast` argument.

```
tips_sub_miss['total_bill'] = pd.to_numeric(
    tips_sub_miss['total_bill'],
    errors='coerce',
    downcast='float')

print(tips_sub_miss)

   total_bill  tip      sex smoker  day    time  size sex_str
0      16.99  1.01  Female     No  Sun  Dinner    2  Female
1        NaN  1.66    Male     No  Sun  Dinner    3   Male
2      21.01  3.50    Male     No  Sun  Dinner    3   Male
3        NaN  3.31    Male     No  Sun  Dinner    2   Male
4      24.59  3.61  Female     No  Sun  Dinner    4  Female
5        NaN  4.71    Male     No  Sun  Dinner    4   Male
6       8.77  2.00    Male     No  Sun  Dinner    2   Male
7        NaN  3.12    Male     No  Sun  Dinner    4   Male
8      15.04  1.96    Male     No  Sun  Dinner    2   Male
9      14.78  3.23    Male     No  Sun  Dinner    2   Male
/home/dchen/anaconda3/envs/book36/bin/pweave:4:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/indexing.html#indexing-view-versus-copy
import re

print(tips_sub_miss.dtypes)

total_bill      float32
tip            float64
sex           category
smoker         category
day            category
time           category
size          int64
sex_str        object
dtype: object
```

You can see the `dtype` for our column is no longer a `float64`, it is now taking up a smaller amount of space (i.e., memory) now that it has been `downcast` to a `float32`.

7.4 Categorical Data

Not all data values are numeric. Pandas has a `category3` `dtype` that can encode categorical values. There are a few use cases for categorical data

1. can be memory⁴ and speed⁵ efficient to store data in this manner. Especially if there are many repeated string values
2. when a column of values has an order (e.g., a likert scale)
3. some python libraries understand how to deal with categorical data (e.g., when fitting statistical models)

³Categorical Data: <http://pandas.pydata.org/pandas-docs/stable/categorical.html>

⁴Categorical memory efficiency: <https://pandas.pydata.org/pandas-docs/stable/categorical.html#memory>

⁵Categorical performance: <https://www.anaconda.com/blog/developer-blog/pandas-categoricals/>

7.4.1 Convert to categorical

To convert a column into a categorical type, we can pass in `category` into the `astype` method.

```
# convert the sex column into a string object first
tips['sex'] = tips['sex'].astype('str')
print(tips.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null object
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
sex_str       244 non-null object
dtypes: category(3), float64(2), int64(1), object(2)
memory usage: 10.7+ KB
None

# convert the sex column back into a categorical
tips['sex'] = tips['sex'].astype('category')
print(tips.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):
total_bill    244 non-null float64
tip          244 non-null float64
sex           244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
sex_str       244 non-null object
dtypes: category(4), float64(2), int64(1), object(1)
memory usage: 9.1+ KB
None
```

7.4.2 Manipulating Categorical Data

The API reference⁶ has a list of what operations can be performed on a categorical `Series` which has been reproduced in [Table 7-1](#).

⁶Categorical API reference: <https://pandas.pydata.org/pandas-docs/stable/api.html#api-categorical>

Table 7-1: Categorical API

Attribute or Method	Description
<code>Series.cat.categories</code>	the categories
<code>Series.cat.ordered</code>	whether or not the categories are ordered
<code>Series.cat.codes</code>	return the integer code of the category
<code>Series.cat.rename_categories()</code>	rename categories
<code>Series.cat.reorder_categories()</code>	reorder categories
<code>Series.cat.add_categories()</code>	add new categories
<code>Series.cat.remove_categories()</code>	remove categories
<code>Series.cat.remove_unused_categories()</code>	removes unused categories
<code>Series.cat.set_categories()</code>	sets new categories
<code>Series.cat.as_ordered()</code>	makes the category ordered
<code>Series.cat.as_unordered()</code>	makes the category unordered

7.5 Conclusion

This chapter covered how to convert from one data type to another. `dtypes` govern what operations can and cannot be performed on a column. While the chapter was relatively short, converting types will become an important skill when working with data and when using other pandas methods.

Chapter 8. Strings and Text Data

8.1 Introduction

Most of the data in the world will be stored as text and strings. Even values that may eventually be numeric data, may initially come in the form of text. It's important to be able to work with text data. This chapter won't be pandas specific, We will mainly explore how you manipulate strings within Python without Pandas, first. The following chapters will cover some more pandas materials. Then we will come back to strings and how it all ties back with pandas.

As an aside, many of the string examples in this chapter come from “Monty Python and the Holy Grail”. The quotes were taken from the wikiquote page¹.

¹WikiQuote Monty Python and the Holy Grail:
https://en.wikiquote.org/wiki/Monty_Python_and_the_Holy_Grail

Objectives

This chapter will cover:

1. Subsetting strings
2. String methods
3. String formatting
4. regular expressions

8.2 Strings

In python a **string** is simply a series of characters. They are created by a set of opening and matching single or double quotes. For example,

```
word = 'grail'  
sent = 'a scratch'
```

will create the strings, **grail** and **a scratch**, and assign them to the variables **word** and **sent**, respectively.

8.2.1 Subsetting and Slicing Strings

Strings can be thought of as a container of characters. You can subset it like any other Python container (e.g., **list** or **pandas. Series**).

[Tables 8-1](#) and [8-2](#) show the strings with their associated index. It should help with the examples below when slicing values using the index.

Table 8–1: Index Positions for the string “grail”

index	0	1	2	3	4
string	g	r	a	i	l
neg index	-5	-4	-3	-2	-1

Table 8–2: Index Positions for the string “a scratch”

index	0	1	2	3	4	5	6	7	8
string	a		s	c	r	a	t	c	h
neg index	-9	-8	-7	-6	-5	-4	-3	-2	-1

8.2.1.1 Single Letter

So, to get the first letter of our strings, we can use the square bracket notation, []. This notation is the same method we used in Chapter [1.3](#) where we looked at various slices of data.

```
print(word[0])
g
print(sent[0])
a
```

8.2.1.2 Slicing Multiple Letters

Or we can use slicing notation to get ranges from our strings

```
# get the first 3 characters
# note index 3 is really the 4th character
print(word[0:3])

gra
```

Remember that when using slicing notation in Python, it is left-side inclusive, right-side exclusive. Meaning, it will include the index value specified first, and it will not include the index value specified second.

When we use `[0:3]`, it will include the characters from `0` to `3`, but not index `3`. Another way to say this is `[0:3]` will include the indices from `0` to `2`, inclusive.

8.2.1.3 Negative Numbers

Recall that in python, passing in a negative index actually starts the count from the **end** of a container

```
# get the last letter
print(sent[-1])

h
```

The negative index only refers to the index position, so you can also use them to slice values too.

```
# get 'a'
print(sent[-9:-8])
```

```
a
```

You can combine non-negative numbers with negative numbers.

```
# get 'a'  
print(sent[0:-8])  
  
a
```

Note that you can't actually get the last letter when using negative index slicing

```
# scratch  
print(sent[2:-1])  
  
scratc  
  
# scratch  
print(sent[-7:-1])  
  
scratc
```

8.2.2 Getting the Last Character in a String

Just getting the last element in a string (or any container) can be done with the negative index, `-1`. However, it becomes problematic when we want to use slicing notation also include the last character. For example, if we wanted to use the slicing notation to get the word “scratch” from the `sent` variable, they will all return 1 letter short.

Since python is right side exclusive, we need to specify an index position that is one greater than the last index. To do this, we can get the `len` (length) of the string and use that value to pass into the slicing notation.

```
# note that the last index is position is smaller than  
# the number returned for len  
s_len = len(sent)  
print(s_len)  
  
9  
  
print(sent[2:s_len])  
  
scratch
```

8.2.2.1 Slicing From the Beginning or to the End

A very common task is to slice a value from the beginning to a certain point in the string (or container). The first element will always be `0`, so we can always write something like `word[0:3]` to get the first 3 elements. or `word[-3:length(word)]` to get the last 3 elements.

Another short cut for this is to leave out the left or right side of the `:`. If the left side of the `:` is empty, then the slice will start from the beginning and end at the index on the right (non-inclusive). If the right side of the `:` is empty, then the slice will start from the index on the left, and end at the end of the string. For example, these slices are equivalent:

```
print(word[0:3])  
gra  
  
print(word[ :3])  
gra
```

as well as these:

```
print(sent[2:len(sent)])
scratch
print(sent[2: ])
scratch
```

Another way to specify the entire string is to leave both values empty `print(sent[:])` |a scratch

```
print(sent[:])
a scratch
```

8.2.2.2 Slicing Increments

The final notation while slicing allows you to slice in increments. To do this, you use a second colon, `:`, to provide a third number. The third number allows you to specify the increment to pull values out.

For example you can get every other string by passing in `2` for every second character.

```
print(sent[::-2])
asrth
```

Any integer can be passed here, so if you wanted every third character (or value in a container), you can pass in `3`

```
print(sent[::-3])
act
```

8.3 String Methods

Python strings have many methods that are also commonly used when processing data. A list of all the string methods can be found on the “String Methods” documentation page.² [Tables 8-3](#) and [8-4](#) summarizes some common python string methods.

²String methods: <https://docs.python.org/3.6/library/stdtypes.html#string-methods>

8.4 More String Methods

There are a few more string methods that are useful (but hard to convey in a table).

Table 8–3: Description of Common Python string methods

Method	Description
capitalize	capitalizes the first character
count	counts the number of occurrences of a string
startswith	True if the string begins with specified value
endswith	True if the string ends with specified value
find	lowest index of where the string matched, -1 if no match
index	same as find but returns ValueError if no match
isalpha	True if all characters are alphabetic
isdecimal	True if all characters are decimal numbers (see documentation as well as isdigit , isnumeric, isalnum)
isalnum	True if all characters are alphanumeric (alphabetic or numeric)
lower	copy of a string with all lower-case letters
upper	copy of string with all upper-case letters
replace	copy of a string with the old values replaced with new
strip	removes leading and trailing white space, also see lstrip and rstrip
split	Returns a list of values split by the delimiter (separator)
partition	similar to split (maxsplit=1) but also returns the separator
center	centers the string to a given width
zfill	copy of string left filled with '0'

8.4.1 join

The `join` method takes a container (e.g., a `list`) and returns a new string where each element in the list For example, if we wanted to combine coordinates in the DMS degrees minute ' seconds" notation.

```
d1 = '40'
m1 = "46"
s1 = '52.837'
u1 = 'N'

d2 = '73'
m2 = "58"
s2 = '26.302'
u2 = 'W'
```

We can join all the values with a space, ' ', by using the `join` method on the space.

```
coords = ' '.join([d1, m1, s1, u1, d2, m2, s2, u2])
print(coords)
```

```
40 46' 52.837" N 73 58' 26.302" W
```

This method is also useful if you have a list of strings that you want to put your own delimiter between (e.g., tabs \t, or commas ,). If we wanted, we can now `split` on a space, ' ', and get the individual parts from `coords`.

Table 8–4: Examples of Common Python string methods

Code	Results
"black Knight".capitalize()	'Black knight'
"It's just a flesh wound!".count('u')	2
"Halt! Who goes there?".startswith ('Halt')	True
"coconut".endswith('nut')	True
"It's just a flesh wound!".find('u')	7
"It's just a flesh wound!".index('scratch ')	ValueError
"old woman".isalpha()	False (there's a whitespace)
"37".isdecimal()	True
"I'm 37".isalnum()	False (apostrophe and space)
"Black Knight".lower()	'black knight'
"Black Knight".upper()	'BLACK KNIGHT'
"flesh wound!".replace(' flesh wound', ' scratch ')	'scratch !'
" I'm not dead. ".strip ()	" I'm not dead."
"NI! NI! NI! NI!".split (sep=' ')	['NI!', 'NI!', 'NI!', 'NI!']
"3,4. partition (',')	('3', ',', '4')
"nine".center(width=10)	' nine '
"9".zfill (width=5)	'00009'

8.4.2 splitlines

The `splitlines` method is similar to the `split` method. It is usually used on strings that are multiple lines long, and will return a list where each element of the list is a line in the multi-line string.

```
multi_str = """Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're bangin' 'em together.
"""
print(multi_str)

Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're bangin' 'em together.
```

We can get every line as a separate element in a list using `splitlines`

```
multi_str_split = multi_str.splitlines()
print(multi_str_split)

['Guard: What? Ridden on a horse?', 'King Arthur: Yes!', "Guard:
You're using coconuts!", 'King Arthur: What?', "Guard: You've got ...
coconut[s] and you're bangin' 'em together."]
```

Finally, if we just wanted the text from the `Guard`, we see this is a 2 person dialog, so the `Guard` speaks every other line

```
guard = multi_str_split[::2]
print(guard)
```

```
[ 'Guard: What? Ridden on a horse?', "Guard: You're using coconuts!",  
  "Guard: You've got ... coconut[s] and you're bangin' 'em together."]
```

There's a few ways to just get the lines from the `Guard`. One way would be to use the `replace` method on the string and `replace` the '`Guard:`' string with an empty string '' . And then use the `splitlines` .

```
guard = multi_str.replace("Guard: ", "").splitlines()[:2]  
print(guard)  
  
['What? Ridden on a horse?', "You're using coconuts!", "You've got ...  
coconut[s] and you're bangin' 'em together."]
```

8.5 String Formatting

Formatting strings allow you to specify a generic template for a sting, and insert variables into the pattern. It can also handle various ways to visually represent some string, for example, showing 2 decimal values in a `float`, or represent a number as a percentage, instead of a decimal value.

It can even help when you want to print something to the console, and instead of just printing out the variable, you can print a string that hints to the value that is printed. There are many use cases for string formatting, and python has 2 ways to do so.

8.5.1 Custom String Formatting

A newer way of string formatting is described in the string formatting documentation³ with plenty of examples⁴

³Custom String Formatting: <https://docs.python.org/3.6/library/string.html#string-formatting>

⁴String formatting examples: <https://docs.python.org/3.6/library/string.html#format-examples>

8.5.2 Formating Character Strings

You essentially write a string with special place holder characters and use the `format` method on the string to insert values into the placeholder.

```
var = 'flesh wound'  
s = "It's just a {}!"  
  
print(s.format(var))  
  
It's just a flesh wound!  
  
print(s.format('scratch'))  
  
It's just a scratch!
```

The placeholders can also refer to variables multiple times

```
# using variables multiple times by index  
s = """Black Knight: 'Tis but a {0}.  
King Arthur: A {0}? Your arm's off!  
"""  
print(s.format('scratch'))  
  
Black Knight: 'Tis but a scratch.
```

King Arthur: A scratch? Your arm's off!

You can also give the placeholders a variable.

```
s = 'Hayden Planetarium Coordinates: {lat}, {lon}'  
print(s.format(lat='40.7815 N', lon='73.9733 W'))  
  
Hayden Planetarium Coordinates: 40.7815 N, 73.9733 W
```

8.5.3 Formating Numbers

Numbers can also be formatted.

```
print('Some digits of pi: {}'.format(3.14159265359))  
  
Some digits of pi: 3.14159265359
```

You can even format numbers and use 1000 place comma separators.

```
print("In 2005, Lu Chao of China recited {:,} digits of pi".\n      format(67890))  
  
In 2005, Lu Chao of China recited 67,890 digits of pi
```

Numbers can be used to perform a calculation and formatted to a certain number of decimal values. Here we can calculate a proportion and format it into a percentage

```
# the 0 in {0:.4} and {0:.4%} refer to the 0 index in format  
# the .4 refers to how many decimal values, 4  
# if we provide a %, it will format the decimal into a percentage  
print("I remember {0:.4} or {0:.4%} of what Lu Chao recited".\n      format(7/67890))  
  
I remember 0.0001031 or 0.0103% of what Lu Chao recited
```

Finally, you can also use string formatting to pad a number with zeros, similar to how `zfill` works on strings. When working with data, this may be useful when working with ID numbers that were read in as an number but should be a string.

```
# the first 0 refers to the index in format  
# the second zero refers to the character to fill  
# the 5 in this case refers to how many characters in total  
# the d signals a digit will used  
# this'll pad a number with 0s so the entire string has 5 characters  
print("My ID number is {0:05d}".format(42))  
  
My ID number is 00042
```

8.5.4 C `printf` style formatting

Another way that you will find string formatting performed in Python is using the `%` operator. This follows the C `printf` style formatting. According to the documentation⁵ The `str .format` method presented in the previous section is preferred. Nonetheless, you may still find code examples that use this formatting style.

⁵C `printf` style formatting: <https://docs.python.org/3.6/library/stdtypes.html#old-string-formatting>

I won't go too much into detail with this method, but here are some of the previous examples recreated using the C `printf` style formatting.

```

# the d represents an integer digit
s = 'I only know %d digits of pi' % 7
print(s)
I only know 7 digits of pi

# the s represents a string
# note the string patter uses round brackets ( )
# instead of curly brackets { }
# the variables passed is a python dict, which uses { }
print('Some digits of %(cont)s: %(value).2f' % \
{'cont': 'e', 'value': 2.718})

Some digits of e: 2.72

```

8.5.5 The new f-strings in Python 3.6+

Formatted literal strings, **f- strings**, are a new feature in Python. The syntax is very similar to ones used in [Section 8.5.1](#). The differences is that we have to begin our string with the letter **f**. This tells Python we have a formatted literal string. We can then use the variable directly in the place holder, **{ }**, without calling **format**.

```

var = 'flesh wound'
s = f"It's just a {var}!"
print(s)
It's just a flesh wound!

lat='40.7815 N'
lon='73.9733 W'
s = f'Hayden Planetarium Coordinates: {lat}, {lon}'
print(s)

Hayden Planetarium Coordinates: 40.7815 N, 73.9733 W

```

The main benefits of using **f-strings** is that they are more readable, and can be faster and more performant.

8.6 Regular Expressions (RegEx)

When the base python string methods that search for patterns isn't enough, you can throw the kitchen sink at the problem by using regular expressions. While extremely powerful, regular expressions provide a (non trivial) way to find and match patterns in strings. The downside is that after you finish writing a complex regular expression, it becomes difficult to figure out what the pattern does by looking at it. That is, the syntax is difficult to read.

For many data tasks, such as matching a telephone number, or address field validation, it's almost easier to Google what type of pattern you are trying to match, and paste what someone already has written (don't forget to document where you got the pattern from).

Before continuing, <https://regex101.com/> is a great place and reference for regular expressions and testing patterns on test strings. It even has a python mode, so you can directly copy/paste your pattern from the site into your python code.

Regular Expressions in Python use the **re** module⁶. The module also has a great How To⁷ that can be used as an additional resource.

⁶Python regular expressions: <https://docs.python.org/3.6/library/re.html>

[Python regular expressions HowTo: <https://docs.python.org/3.6/howto/regex.html>](#)

[Tables 8-5](#) and [8-6](#) show some common RegEx syntax and special characters that will be used in this section.

To use regular expressions, we write a string that contains the regex pattern, and provide a string for the pattern to match. There are various functions within `re`. Some common tasks are provided in [Table 8-7](#)

8.6.1 Match a Pattern

We will be using the `re` module to write our regular expression pattern, to match in a string. Let's write a pattern that will match 10 digits (the digits for a US telephone number).

```
import re
tele_num = '1234567890'
```

There are many ways we can match 10 consecutive digits. We can use the `match` function to see if the pattern matches a string. The output of many `re` functions is a `match` object.

Table 8-5: Basic RegEx Syntax

Syntax	Description
.	Matches any 1 character
^	Matches from the beginning of a string
\$	Matches from the end of a string
*	Matches 0 or more repetitions of the previous character
+	Matches 1 or more repetitions of the previous character
?	Matches 0 or 1 repetition of the previous character
{m}	Matches m repetitions of the previous character
{m,n}	Matches any number from m to n of the previous character
\	Escape character
[]	A set of characters, e.g., [a-z] will match all letters from a to z
	OR, A B will match A or B
()	Matches the pattern specified within the parenthesis exactly

Table 8-6: Common RegEx special characters

Sequence	Description
\d	a digit
\D	any character NOT a digit (opposite of \d)
\s	any whitespace character
\S	any character NOT a whitespace (opposite of \s)
\w	word characters
\W	any character NOT a word character (opposite of \w)

Table 8–7: Common RegEx Functions

Function	Description
search	find the first occurrence of a string
match	match from the beginning of a string
fullmatch	matches the entire string
split	split string by the pattern
findall	find all non-overlapping matches of of a string
finditer	similar to findall but returns a python iterator
sub	substitutes the matched pattern with a provided string

```
m = re.match(pattern='\d\d\d\d\d\d\d\d\d', string=tele_num)
print(type(m))

<class '_sre.SRE_Match'>

print(m)

<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
```

If we look at the printed match object, you can see that if there was a match, the `span` tells us the index of the string where the matches occurred, and the `match` tells us the exact string that got matched.

Many times when we are matching a pattern to a string, we simply want a `True` or `False` value on whether there was a match. If you just need a `True/False` value you can run the built-in `bool` function to get the boolean value of the match object

```
print(bool(m))
True
```

However, many times a regular expression match will be part of an `if`-statement, so the above `bool` casting is unnecessary.

```
# should print match
if m:
    print('match')
else:
    print('no match')

match
```

If we wanted to extract some of the match object values, such as the index positions or the actual string that matched, we can use a few methods on the match object.

```
# get the first index of the string match
print(m.start())
0

# get the last index of the string match
print(m.end())
10

# get the first and last index of the string match
print(m.span())
(0, 10)

# the string that matched the pattern
print(m.group())
1234567890
```

Telephone numbers can be a little more complex than a series of 10 consecutive digits. Here's another common representation.

```
tele_num_spaces = '123 456 7890'
```

If we use the previous pattern on this example

```
# we can simplify the previous pattern
m = re.match(pattern='\d{10}', string=tele_num_spaces)
print(m)
None
```

You can tell the pattern did not match because the match object returned `None`. If we ran our `if` statement again, it will print '`no match`'

```
if m:
    print('match')
else:
    print('no match')

no match
```

Let's modify our pattern this time, the new string has 3 digits, a space, another 3 digits, another space, followed by 4 digits. If we want to make it general to the original example, the spaces can be matched 0 or 1 times. The new regex pattern will look as follows:

```
# you may see regex pattern as a separate variable
# because it can get long and
# make the actual match function call hard to read
p = '\d{3}\s?\d{3}\s?\d{4}'
m = re.match(pattern=p, string=tele_num_spaces)
print(m)

<_sre.SRE_Match object; span=(0, 12), match='123 456 7890'>
```

Area codes can also be surrounded by parenthesis and a dash between the 7 main digits.

```
tele_num_space_paren_dash = '(123) 456-7890'
p = '(\?\d{3}\)?\s?\d{3}\s?-?\d{4}'
m = re.match(pattern=p, string=tele_num_space_paren_dash)
print(m)

<_sre.SRE_Match object; span=(0, 14), match='(123) 456-7890'>
```

Finally, there could be a country code before the number

```
cnty_tele_num_space_paren_dash = '+1 (123) 456-7890'
p = '\+?1\s?\(?d{3}\)\?\s?d{3}\s?-?d{4}'
m = re.match(pattern=p, string=cnty_tele_num_space_paren_dash)
print(m)

<_sre.SRE_Match object; span=(0, 17), match='+1 (123) 456-7890'>
```

Hopefully you can see in these examples that although powerful, regular expressions can easily become unwieldy. Even something as simple as a telephone number almost ends up in a daunting series of symbols and numbers. But sometimes, regular expressions are the only way to get something done.

8.6.2 Find a Pattern

We can use the `.findall` function to find all matches within a pattern. Let's write a pattern that matches digits and use it to find all the digits from a string.

```
p = '\d+'
# python will concatenate 2 strings next to each other
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \
      "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = re.findall(pattern=p, string=s)
print(m)

['13', '12', '11', '10', '9']
```

8.6.3 Substituting a Pattern

In our `str.replace` example we wanted to get all the lines from the `Guard`, we ended up doing a direct string replacement on the script. However, using regular expressions, we can generalize the pattern so we can get the line from the `Guard OR King Arthur`

```
multi_str = """Guard: What? Ridden on a horse?
King Arthur: Yes!
Guard: You're using coconuts!
King Arthur: What?
Guard: You've got ... coconut[s] and you're bangin' 'em together.
"""

p = '\w+\s?\w+: \s?'

s = re.sub(pattern=p, string=multi_str, repl='')
print(s)

What? Ridden on a horse?
Yes!
You're using coconuts!
What?
You've got ... coconut[s] and you're bangin' 'em together.
```

Now if we wanted we can get either party's line using string slicing with increments.

```
guard = s.splitlines()[ ::2]
kinga = s.splitlines()[1::2] # skip the first element

print(guard)

['What? Ridden on a horse?', "You're using coconuts!", "You've got ...
coconut[s] and you're bangin' 'em together."]
print(kinga)

['Yes!', 'What?']
```

Don't be afraid to mix and match regular expressions with the more simpler pattern match and string methods.

8.6.4 Compiling a Pattern

Just thinking ahead a little. When we work with data, typically many operations will occur on a column-by-column or row-by-row basis. Python's `re` module allows you to `compile` a pattern so it can be reused. This can give you performance benefits, especially if your dataset is large. Here we will see how to compile a pattern and use it just like the previous examples in this section.

The syntax is almost the same. We write our regular expression pattern, but this time instead of saving it to a variable directly, we pass the string into the `compile` function and save that result. We can then use the other `re` functions on the compiled pattern. Also, since the pattern is already compiled, you no longer need to specify the pattern parameter in the method.

Here is the `match` example:

```
p = re.compile('\d{10}')
s = '1234567890'
# note: calling match on the compiled pattern
# not using the re.match function
m = p.match(s)
print(m)

<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
```

The `findall` example:

```
p = re.compile('\d+')
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, "
    "11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston"
m = p.findall(s)
print(m)

['13', '12', '11', '10', '9']
```

Finally, the `sub` or substitution example:

```
p = re.compile('\w+\s?\w+:\s?')
s = "Guard: You're using coconuts!"
m = p.sub(string=s, repl='')
print(m)

You're using coconuts!
```

8.7 The regex Library

The `re` library is a very popular regular expression library in Python because it is a built-in and the default regular expression engine in the language. However, the hardcore regular expression writers may find the `regex` library to be far superior and feature complete. It is backwards compatible with the `re` library, so all the from [Section 8.6](#) will still work. The documentation for the library can be found on the PiPI page⁸.

⁸`regex` documentation: <https://pypi.python.org/pypi/regex/>

```
import regex

# a re example using regex
p = regex.compile('\d+')
```

```
s = "13 Jodie Whittaker, war John Hurt, 12 Peter Capaldi, \"\n    \"11 Matt Smith, 10 David Tennant, 9 Christopher Eccleston\"\nprint(m)\n['13', '12', '11', '10', '9']
```

I will defer to the examples and writeup on www.rexegg.com for more details:

- <http://www.rexegg.com/regex-python.html>
- <http://www.rexegg.com/regex-best-trick.html>

8.8 Conclusion

The world is filled with data stored as text. Understanding how to manipulate text strings will be a fundamental skill as a data scientist. Python has many built-in string methods and libraries that can make string and text manipulation easier. This chapter covered some of the fundamental methods of string manipulations that we can build on when working with data.

Chapter 9. Apply

9.1 Introduction

Learning about `apply` is fundamental in the data cleaning process. It also encapsulates key concepts in programming, mainly writing functions. `apply` takes a function, and ‘applies’ (i.e., runs it) across each row or column of a dataframe, ‘simultaneously’. If you’ve programmed before, then the concept of an ‘apply’ should be familiar. It is similar to writing a `for` loop across each row or column and calling the function, `apply` just does it ‘simultaneously’. In general, this is the preferred way to `apply` functions across dataframes, because it typically is much faster than writing a `for` loop in Python.

Objectives

This chapter will cover:

1. Functions
2. Applying functions across columns or rows of data

9.2 Functions

Functions are core to writing `apply` statements. There’s a lot more information about functions in [Appendix Q](#), but here’s a quick introduction.

Functions are a way to group and reuse Python code. If you are ever in a situation where you are copy/pasting code and changing a few parts of the code, then chances are the copied code can be written into a function. To create a function, we need to “define” it, The basic function skeleton looks like this:

```
def my_function():
    # indent 4 spaces
    # function code here
```

Since Pandas is used for data analysis, let’s write a more “useful” function, one that squares a given value and another that takes 2 numbers and calculates the average between them.

```
def my_sq(x):
    """Squares a given value
    """
    return x ** 2

def avg_2(x, y):
    """Calculates the average between 2 numbers
    """
    return (x + y) / 2
```

The text within the triple quotes is a `docstring`. This is the text that shows up when you look up the help documentation about a function. You can use these docstrings to write your own documentation for functions you write as well.

We’ve been using functions throughout the book, if we want to use functions that we’ve created

ourselves, we can call them just like a function we've loaded from a library.

```
print(my_sq(4))
16
print(avg_2(10, 20))
15.0
```

9.3 Apply (Basics)

Now that we know how to write a function, how would we use them in Pandas? When working with dataframes, it's more likely that you want to use a function across rows or columns of your data.

Here's a mock dataframe of 2 columns.

```
import pandas as pd

df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})

print(df)

      a    b
0    10   20
1    20   30
2    30   40
```

We can **apply** our functions over a **series** (i.e., an individual column or row).

For didactic purposes, let's use the function we wrote to square the '**a**' column. In this overly simplified example, we could've directly squared the column as such:

```
print(df['a'] ** 2)

0    100
1    400
2    900
Name: a, dtype: int64
```

But that would not allow us to use a function we wrote ourselves.

9.3.1 Apply Over a Series

In our example, if we subset a single column or row, the **type** of the object we get back is a Pandas **series**.

```
# get the first column
print(type(df['a']))

<class 'pandas.core.series.Series'>

# get the first row
print(type(df.iloc[0]))

<class 'pandas.core.series.Series'>
```

The **series** has a method called **apply**¹. The way we use the **apply** is we pass the function we want to use across each element in the **series**.

¹`series apply` documentation: <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.apply.html#pandas.Series.apply>

For example, if we wanted to square each value in column `a` we can do the following:

```
# apply our square function on the 'a' column
sq = df['a'].apply(my_sq)
print(sq)

0    100
1    400
2    900
Name: a, dtype: int64
```

Note we do not need the round brackets, (), when we pass the function into the `apply`. Let's build on this example by writing a function that takes 2 parameters, the first parameter will be a value, and the second parameter will be the exponent to raise the value to. So far in our `my_sq` function, we've "hard-coded" the exponent, `2`, to raise our value to.

```
def my_exp(x, e):
    return x ** e
```

Now if we want to use our function, we have to provide it 2 parameters

```
cb = my_exp(2, 3)
print(cb)

8
```

However, if we want to `apply` the function on our series, we will need to pass in the second parameter. To do this, we simply pass in the second argument as a **keyword argument** into the `apply`

```
ex = df['a'].apply(my_exp, e=2)
print(ex)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

```
ex = df['a'].apply(my_exp, e=3)
print(ex)
```

```
0     1000
1     8000
2    27000
Name: a, dtype: int64
```

9.3.2 Apply Over a DataFrame

Now that we've seen how to `apply` functions over a 1-dimensional `Series`, let's see how the syntax changes when working with `DataFrames`. Here is the example `DataFrame` from earlier:

```
df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})
print(df)

      a    b
0   10   20
1   20   30
2   30   40
```

`DataFrames` typically have at least 2 dimensions, so when we `apply` a function over a dataframe,

we first need to specify which `axis` to `apply` the function over, e.g., column-by-column or row-by-row.

Let's first write a function that takes a single value, and prints out the given value.

```
def print_me(x):
    print(x)
```

Let's `apply` this function on our `df`. The syntax is similar to the `apply` on a `Series`, however, this time we need to specify whether or not we want the function to be applied column-wise or row-wise.

If you want the function to work column-wise, we can pass in the `axis=0` parameter into `apply`. If you want the function to work row-wise, we can pass in the `axis=1` parameter into `apply`.

9.3.2.1 Column-wise Operations

Use the `axis=0` parameter (the default value) in `apply` to functions column-wise

```
df.apply(print_me, axis=0)
```

```
0    10
1    20
2    30
Name: a, dtype: int64
0    20
1    30
2    40
Name: b, dtype: int64

a    None
b    None
dtype: object
```

Compare this output to the following:

```
print(df['a'])

0    10
1    20
2    30
Name: a, dtype: int64

print(df['b'])

0    20
1    30
2    40
Name: b, dtype: int64
```

You can see that the outputs are exactly the same. When you `apply` a function across a `DataFrame` (in this case column-wise with `axis=0`), the entire axis (e.g., column) is passed into the first argument of the function. To illustrate this further, Let's write a function that calculates the mean (average) of 3 numbers. I'm picking 3 because each column has 3 values in it.

```
def avg_3(x, y, z):
    return (x + y + z) / 3
```

If we want to `apply` this function across our columns, you will get an error.

```
# will cause an error
print(df.apply(avg_3))
```

```

Traceback (most recent call last):
  File "<ipython-input-1-5ebf32ddae32>", line 2, in <module>
    print(df.apply(avg_3))
TypeError: ("avg_3() missing 2 required positional arguments: 'y' and
'z'", 'occurred at index a')

```

From the (last line of the) error message you can see that it the function takes 3 arguments, but we failed to pass in the `y` and `z` (i.e., the second and third) arguments. Again, this is because when we use `apply`, the **entire** column got passed into the **first** argument. For the function to work in the `apply`, we will have to re-write parts of it

```

def avg_3_apply(col):
    x = col[0]
    y = col[1]
    z = col[2]
    return (x + y + z) / 3

print(df.apply(avg_3_apply))

a    20.0
b    30.0
dtype: float64

```

9.3.2.2 Row-wise Operations

Row-wise operations work just like column-wise operations. The parts that differ is the `axis`, we will now use `axis=1`, in the `apply`, and instead of the entire column being passed into the first argument of the function, it is now the entire row of the dataframe.

Since our example dataframe has 2 columns and 3 rows, the `avg_3_apply` function we just wrote will not work row-wise.

```

# will cause an error
print(df.apply(avg_3_apply, axis=1))

Traceback (most recent call last):
  File "/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/pandas/core/indexes/base.py", line 2477, in get_value
    tz=getattr(series.dtype, 'tz', None))
KeyError: 2

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<ipython-input-1-8e6ba41f3975>", line 2, in <module>
    print(df.apply(avg_3_apply, axis=1))
IndexError: ('index out of bounds', 'occurred at index 0')

```

The main issue here is the '`index out of bounds`', we passed in the row of data into the first argument, but in our function we begin indexing out of range (i.e., we only have 2 values in each row, but we tried to get index `2`, aka the 3rd element, that does not exist). If we wanted to calculate our averages row-wise, we'd have to write a new function.

```

def avg_2_apply(row):
    x = row[0]
    y = row[1]
    return (x + y) / 2

print(df.apply(avg_2_apply, axis=0))

a    15.0
b    25.0
dtype: float64

```

9.4 Apply

The previous examples used a small toy dataset illustrate how `apply` works. It showed how you can write a function that can be tested before converting it into something to be used for `apply`, by writing the function that takes as many inputs as you need, and then converting it into a function that takes one parameter, the entire row or entire column, and then subsetting the components within the function body. [Section 9.5](#) shows another way to get an existing function to work with `apply`, but for now, let's use a more realistic example.

The `seaborn` library has a built-in `titanic` dataset. It contains data about whether an individual survived during the sinking of the Titanic.

```
import seaborn as sns
titanic = sns.load_dataset("titanic")
print(titanic.info())
print(titanic.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
survived    891 non-null int64
pclass      891 non-null int64
sex         891 non-null object
age         714 non-null float64
sibsp       891 non-null int64
parch       891 non-null int64
fare         891 non-null float64
embarked    889 non-null object
class        891 non-null category
who          891 non-null object
adult_male   891 non-null bool
deck         203 non-null category
embark_town  889 non-null object
alive        891 non-null object
alone        891 non-null bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.6+ KB
None
```

This dataset has 891 rows and 15 columns. Almost all of the cells in our dataset have a value in them. Of the 891 values, `age` has 714 complete cases, and `deck` has 203 complete cases. One way we can use `apply` is to calculate how many `null` or `NaN` values there are in our data, as well as the percentage of complete cases across each column or across each row. Let's write a few functions.

1. number of missing values

```
# we'll be using the numpy sum function
import numpy as np

def count_missing(vec):
    """Counts the number of missing values in a vector
    """
    # get a vector of True/False values
    # depending if the value is missing or not
    null_vec = pd.isnull(vec)

    # take the sum of the null_vec
    # since null values do not contribute to the sum
    null_count = np.sum(null_vec)

    # return the number of missing values in the vector
```

```
    return null_count
```

2. proportion of missing values

```
def prop_missing(vec):
    """Percent of missing values in a vector
    """
    # numerator aka number of missing values
    # we can use the count_missing function we just wrote!
    num = count_missing(vec)

    # denominator aka the total number of values in the vector
    # remember we also need to count the missing values
    dem = vec.size

    # return the proportion/percent of missing
    return num / dem
```

3. proportion of complete values

```
def prop_complete(vec):
    """Percent of non missing values in a vector
    """
    # we can utilize the percent_missing function we just wrote
    # by subtracting it's value from 1
    return 1 - prop_missing(vec)
```

The beauty about many (if not all) of the functions from `numpy` and `pandas` is that they work on vectors. Unlike our original set of functions that calculated the mean of 2 or 3 values, we can pass in an arbitrarily number of items into `pd. isnull` or `np.sum` and the function will calculate the corresponding value. These “vectorized” functions ([Section 9.5](#)) work across a “vector” and can handle any arbitrary amount of information.

9.4.1 Column-wise

Let’s use our newly created functions on each column of our data.

```
cmis_col = titanic.apply(count_missing)
pmis_col = titanic.apply(prop_missing)
pcom_col = titanic.apply(prop_complete)

print(cmis_col)

survived      0
pclass        0
sex           0
age          177
sibsp         0
parch         0
fare          0
embarked      2
class         0
who           0
adult_male    0
deck          688
embark_town   2
alive         0
alone         0
dtype: int64

print(pmis_col)

survived      0.000000
pclass       0.000000
sex          0.000000
age         0.198653
```

```

sibsp      0.000000
parch     0.000000
fare      0.000000
embarked  0.002245
class     0.000000
who       0.000000
adult_male 0.000000
deck      0.772166
embark_town 0.002245
alive     0.000000
alone     0.000000
dtype: float64

print(pcom_col)

survived    1.000000
pclass      1.000000
sex         1.000000
age         0.801347
sibsp       1.000000
parch       1.000000
fare         1.000000
embarked   0.997755
class       1.000000
who         1.000000
adult_male  1.000000
deck        0.227834
embark_town 0.997755
alive       1.000000
alone       1.000000
dtype: float64

```

What can we do with this information? Since we have counts of missing values, we can access whether or not a column is a viable option to be used in an analysis. For example, there are only 2 missing values in the `embark_town` column. We can easily check those rows to see if these values are missing randomly, or if there is a special reason for them to be missing.

```

print(titanic.loc[pd.isnull(titanic.embark_town), :])

      survived  pclass     sex   age  sibsp  parch  fare embarked \
61          1       1  female  38.0      0      0  80.0      NaN
829         1       1  female  62.0      0      0  80.0      NaN

      class   who  adult_male deck embark_town alive  alone
61  First  woman     False     B        NaN  yes  True
829 First  woman     False     B        NaN  yes  True

```

Another observation is that the `deck` variable has 688 (77.2%) of it's values missing. Barring further investigation, it's save to say this is a variable we would not use for an analysis.

9.4.2 Row-wise

Since our functions are vectorized, we can `apply` them across the rows of our data without changing it.

```

cmis_row = titanic.apply(count_missing, axis=1)
pmis_row = titanic.apply(prop_missing, axis=1)
pcom_row = titanic.apply(prop_complete, axis=1)

print(cmis_row.head())

0      1
1      0
2      1
3      0
4      1
dtype: int64

```

```
print(pmis_row.head())
```

```
0    0.066667  
1    0.000000  
2    0.066667  
3    0.000000  
4    0.066667  
dtype: float64
```

```
print(pcom_row.head())
```

```
0    0.933333  
1    1.000000  
2    0.933333  
3    1.000000  
4    0.933333  
dtype: float64
```

One thing we can do with this analysis is to see if we have any rows in our data that have multiple missing values.

```
print(cmis_row.value_counts())
```

```
1    549  
0    182  
2    160  
dtype: int64
```

Since we are using `apply` row-wise, we can actually create a new column with these values

```
titanic['num_missing'] = titanic.apply(count_missing, axis=1)
```

```
print(titanic.head())
```

```
survived  pclass      sex   age  sibsp  parch      fare embarked  \
0         0       3 male  22.0     1     0    7.2500      S
1         1       1 female  38.0     1     0   71.2833      C
2         1       3 female  26.0     0     0    7.9250      S
3         1       1 female  35.0     1     0   53.1000      S
4         0       3 male  35.0     0     0    8.0500      S

class     who  adult_male deck  embark_town alive  alone  \
0  Third    man      True   NaN  Southampton   no  False
1  First  woman     False     C  Cherbourg   yes  False
2  Third  woman     False   NaN  Southampton   yes  True
3  First  woman     False     C  Southampton   yes  False
4  Third    man      True   NaN  Southampton   no  True

num_missing
0        1
1        0
2        1
3        0
4        1
```

We can then look at the rows with multiple missing values. Since there are too many rows with multiple values to print in this book, we can randomly sample the results.

```
print(titanic.loc[titanic.num_missing > 1, :].sample(10))
```

```
survived  pclass      sex   age  sibsp  parch      fare embarked  \
470        0       3 male  NaN     0     0    7.2500      S
468        0       3 male  NaN     0     0    7.7250      Q
464        0       3 male  NaN     0     0    8.0500      S
65         1       3 male  NaN     1     1   15.2458      C
330        1       3 female  NaN     2     0   23.2500      Q
109        1       3 female  NaN     1     0   24.1500      Q
121        0       3 male  NaN     0     0    8.0500      S
639        0       3 male  NaN     1     0   16.1000      S
48         0       3 male  NaN     2     0   21.6792      C
837        0       3 male  NaN     0     0    8.0500      S
```

```

    class   who  adult_male  deck  embark_town  alive  alone  \
470  Third   man      True   NaN  Southampton   no   True
468  Third   man      True   NaN  Queenstown   no   True
464  Third   man      True   NaN  Southampton   no   True
65   Third   man      True   NaN  Cherbourg  yes  False
330  Third  woman     False  NaN  Queenstown  yes  False
109  Third  woman     False  NaN  Queenstown  yes  False
121  Third   man      True   NaN  Southampton   no   True
639  Third   man      True   NaN  Southampton   no  False
48   Third   man      True   NaN  Cherbourg  no  False
837  Third   man      True   NaN  Southampton   no   True

num_missing
470      2
468      2
464      2
65       2
330      2
109      2
121      2
639      2
48       2
837      2

```

9.5 Vectorized Functions

When we use `apply`, we are able to make a function work column-by-column or row-by-row. However, in [Section 9.3](#), we had to re-write our function when we had to `apply` it since the entire column or row got passed into the first parameter of the function. However, there might be times where its not feasible to re-write a function like we did. We can leverage the `vectorize` function and decorator to vectorize any function. Vectorizing your code can also have performance gains (See [17.2.1](#)) Here's our toy dataframe:

```

df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})
print(df)

      a    b
0    10   20
1    20   30
2    30   40

```

And our average function that we can apply on a row-by-row basis:

```

def avg_2(x, y):
    return (x + y) / 2

```

For a vectorized function, we'd like to be able to pass in a vector of values for `x`, a vector of values for `y`, and the results should be the average of a given `x` and `y` in the same order. What we want is to be able to write `avg_2(df['a'], df['b'])` and get `[15, 25, 35]` as a result.

```

print(avg_2(df['a'], df['b']))

0    15.0
1    25.0
2    35.0
dtype: float64

```

This works because the actual calculations within our function are inherently vectorized. That is, if we add 2 numeric columns together, pandas (and numpy) will automatically perform element-wise addition, and when we divide by a scalar, it will broadcast the scalar, and divide each element by the scalar.

Let's change our function and make a non-vectorizable calculation.

```
import numpy as np
def avg_2_mod(x, y):
    """Calculate the average, unless x is 20
    """
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2
```

If we run this function, it will cause an error.

```
# will cause an error
print(avg_2_mod(df['a'], df['b']))

Traceback (most recent call last):
  File "<ipython-input-1-cb2743ef2888>", line 2, in <module>
    print(avg_2_mod(df['a'], df['b']))
ValueError: The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

However, if we give it individual numbers, instead of a vector, it will work as expected

```
print(avg_2_mod(10, 20))

15.0

print(avg_2_mod(20, 30))

nan
```

9.5.1 Using numpy

We want to change our function so when it is given a vector of values, it will perform the calculations element-wise. We can do this by using the `vectorize` function from `numpy`. We pass `np. vectorize` the `function` we want to vectorize, to create a new function.

```
# np.vectorize actually creates a new function
avg_2_mod_vec = np.vectorize(avg_2_mod)
print(avg_2_mod_vec(df['a'], df['b']))

[ 15.  nan  35.]
```

The above method is great if you do not have the source code for an existing function. However, if you are writing your own function, you can actually use what is called a python decorator to “automatically” vectorize the function without having to create a new function. Decorators are “functions” that take another function as input, and modifies how it’s output behaves.

```
# to use the vectorize decorator
# we use the @ symbol before our function definition
@np.vectorize
def v_avg_2_mod(x, y):
    """Calculate the average, unless x is 20
    Same as before, but we are using the vectorize decorator
    """
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

# we can then directly use the vectorized function
# without having to create a new function
print(v_avg_2_mod(df['a'], df['b']))

[ 15.  nan  35.]
```

9.5.2 Using numba

The `numba` library² is designed to optimize python code, especially calculations on arrays performing mathematical calculations. Just like `numpy`, it also has a `vectorize` decorator.

²Numba: <https://numba.pydata.org/>

```
import numba

@numba.vectorize
def v_avg_2_numba(x, y):
    """Calculate the average, unless x is 20
    Using the numba decorator.
    """
    # note we now have to add type information to our function
    if (int(x) == 20):
        return(np.NaN)
    else:
        return (x + y) / 2
```

`numba` does not understand pandas objects,

```
print(v_avg_2_numba(df['a'], df['b']))

Traceback (most recent call last):
  File "<ipython-input-1-b03c5b533ae5>", line 2, in <module>
    print(v_avg_2_numba(df['a'], df['b']))
ValueError: cannot determine Numba type of <class
'pandas.core.series.Series'>
```

We actually have to pass in the numpy array representation of our data ([Appendix R](#)).

```
# passing in the numpy array
print(v_avg_2_numba(df['a'].values, df['b'].values))

[ 15.      nan     35.]
```

9.6 Lambda Functions

Sometimes the function used in the `apply` is simple enough that there is no need to create a separate function.

```
docs = pd.read_csv('../data/doctors.csv', header=None)
```

We can write a pattern that extracts all the letters from the row, and assign those values to a new 'name' column in our data.

We could write our function and `apply` it as we have done

```
import regex

p = regex.compile('\w+\s+\w+')

def get_name(s):
    return p.match(s).group()

docs['name_func'] = docs[0].apply(get_name)
print(docs)

          0           name_func
0  William Hartnell (1963-66)  William Hartnell
1  Patrick Troughton (1966-69)  Patrick Troughton
2    Jon Pertwee (1970-74)      Jon Pertwee
3     Tom Baker (1974-81)       Tom Baker
```

```

4      Peter Davison (1982-84)      Peter Davison
5          Colin Baker (1984-86)      Colin Baker
6      Sylvester McCoy (1987-89)      Sylvester McCoy
7          Paul McGann (1996)      Paul McGann
8  Christopher Eccleston (2005)  Christopher Eccleston
9          David Tennant (2005-10)      David Tennant
10         Matt Smith (2010-13)      Matt Smith
11      Peter Capaldi (2014-2017)      Peter Capaldi
12      Jodie Whittaker (2017)      Jodie Whittaker

```

You can see the actual function is a simple 1-liner. Usually when this happens people will opt to write the 1-liner directly in the apply. This method is called using **lambda functions**. We can perform the same operation as above in the following manner.

```

docs['name_lamb'] = docs[0].apply(lambda x: p.match(x).group())
print(docs)

          0           name_func \
0  William Hartnell (1963-66)  William Hartnell
1  Patrick Troughton (1966-69)  Patrick Troughton
2  Jon Pertwee (1970-74)  Jon Pertwee
3  Tom Baker (1974-81)  Tom Baker
4  Peter Davison (1982-84)  Peter Davison
5  Colin Baker (1984-86)  Colin Baker
6  Sylvester McCoy (1987-89)  Sylvester McCoy
7  Paul McGann (1996)  Paul McGann
8  Christopher Eccleston (2005)  Christopher Eccleston
9  David Tennant (2005-10)  David Tennant
10  Matt Smith (2010-13)  Matt Smith
11  Peter Capaldi (2014-2017)  Peter Capaldi
12  Jodie Whittaker (2017)  Jodie Whittaker

name_lamb
0  William Hartnell
1  Patrick Troughton
2  Jon Pertwee
3  Tom Baker
4  Peter Davison
5  Colin Baker
6  Sylvester McCoy
7  Paul McGann
8  Christopher Eccleston
9  David Tennant
10  Matt Smith
11  Peter Capaldi
12  Jodie Whittaker

```

To write the lambda function, we use the `lambda` keyword. Since apply functions will pass the entire axis as the first argument, our `lambda` function only takes one parameter, `x`. We can then write our function directly, without having to define it. The calculated result is automatically returned.

Although you can write complex multi-line lambda functions, typically, people will use this method when small 1-liner calculations are needed. The code can become hard to read if the lambda function tries to do too much at once.

9.7 Conclusion

This chapter covered an important concept of creating functions that can be used on our data. Not all data cleaning steps or manipulations will be able to be done using built-in functions. There will be (many) times where you will have to write your own custom functions to process and analyze data.

Chapter 10. Groupby operations: split-apply-combine

10.1 Introduction

Grouped operations are a powerful way to aggregate, transform, and filter our data. It uses the mantra of “split-apply-combine”, meaning:

1. Data is split into separate parts based on key(s)
2. A function is applied to each part of data
3. The results from each part is finally combined to create a new dataset

This is a powerful concept where parts of your original data can be split up into independent parts to perform a calculation. If you worked with databases in the past, then the pandas `groupby` works just like the SQL `GROUP BY`. The split-apply-combine concept is also heavily used in “big data” systems that use distributed computing. Where the data is split into independent parts and dispatched to a separate server where a function is applied and the results are all combined together.

The techniques shown in this chapter can all be done without using the `groupby` method. For example,

- Aggregation can be done by using conditional subsetting on a dataframe
- Transformation can be done by passing a column into a separate function
- Filtering can be done with conditional subsetting

However, by thinking about your data in `groupby` statements, your code can be faster, is more flexible when you want to create multiple groups, and help set your mind to work with larger datasets on distributed or parallel systems.

Objectives

This chapter will cover:

1. Groupby operations to aggregate, transform, and filter data
2. Built-in and custom user functions to perform groupby operations

10.2 Aggregate

Aggregation is the process of taking multiple values and retiring a single value. Calculating an arithmetic mean is an example of this, where the average of multiple values is a single value.

10.2.1 Basic 1 Variable Grouped Aggregation

[Section 1.4.1](#) showed how to calculate grouped means on Gapminder data. We calculated the average life expectancy for each year of the data and plotted it. This is an example of using groupby operation for data aggregation, that is, we used the `groupby` statement to calculate a summary statistic, the mean, for all the values in each year.

Aggregation may sometimes be referred to as summarization. Both mean that there is some form of data reduction involved. For example, when you calculate a summary statistic, such as the mean, you are taking multiple values and replacing it with 1 value. The amount of data is now smaller.

```
# load the gapminder data
import pandas as pd
df = pd.read_csv('../data/gapminder.tsv', sep='\t')

# calculate the average life expectancy for each year
avg_life_exp_by_year = df.groupby('year').lifeExp.mean()
print(avg_life_exp_by_year)

year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Recall, the above statement uses dot notation to subset the `lifeExp` column, it is exactly the same as subsetting using bracket notation

```
avg_life_exp_by_year = df.groupby('year')['lifeExp'].mean()
```

Groupby statements can be thought of as creating a subset of each unique value of a column (or unique pairs from columns).

For example, we could get a list of unique values in the column

```
# get a list of unique years in the data
years = df.year.unique()
print(years)

[1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007]
```

We can go through each of the years, and subset the data.

```
# subset the data for the year 1952
y1952 = df.loc[df.year == 1952, :]
print(y1952.head())

   country continent  year  lifeExp    pop  gdpPerCap
0  Afghanistan     Asia  1952    28.801  8425333  779.445314
12  Albania      Europe  1952    55.230  1282697 1601.056136
24  Algeria      Africa  1952    43.077  9279525 2449.008185
36  Angola       Africa  1952    30.015  4232095 3520.610273
48 Argentina     Americas  1952    62.485 17876956 5911.315053
```

Finally, we can perform a function on the subset data. Here I'll be taking the mean of the `lifeExp`.

```
y1952_mean = y1952.lifeExp.mean()
```

```

print(y1952_mean)
49.0576197183

```

What the `groupby` does is essentially repeat this process for every year column, and conveniently return all the results into a single dataframe. The mean is not the only type of aggregation function you can use. There are many built-in methods in Pandas you can use with the `groupby` statement.

10.2.2 Built-in aggregation methods

[Table 10-1](#) contains a non-exclusive list of built-in pandas methods you can use to aggregate your data.

For example, we can calculate multiple summary statistics simultaneously with `describe`.

```

# group by continent and describe each group
continent_describe = df.groupby('continent').lifeExp.describe()
print(continent_describe)

    count      mean       std      min      25%      50%  \
continent
Africa     624.0  48.865330   9.150210  23.599  42.37250  47.7920
Americas   300.0  64.658737   9.345088  37.579  58.41000  67.0480
Asia       396.0  60.064903  11.864532  28.801  51.42625  61.7915
Europe     360.0  71.903686   5.433178  43.585  69.57000  72.2410
Oceania    24.0   74.326208   3.795611  69.120  71.20500  73.6650

```

Table 10-1: Table of methods and functions that

pandas method	numpy/scipy function	description
count	np.count_nonzero	frequency count <i>not</i> including NaN values
size		frequency count <i>with</i> NaN values
mean	np.mean	mean of the values
std	np.std	sample standard deviation
min	np.min	minimum values
quantile (q=0.25)	np.percentile (q=0.25)	25th percentile of the values
quantile (q=0.50)	np.percentile (q=0.50)	50th percentile of the values
quantile (q=0.75)	np.percentile (q=0.75)	75th percentile of the values
max	np.max	maximum value
sum	np.sum	sum of the values
var	np.var	unbiased variance
sem	scipy.stats.sem	unbiased standard error of the mean
describe	scipy.stats.describe	count, mean, std, min, 25%, 50%, 75%, and max
first		returns the first row
last		returns the last row
nth		returns the nth row (python starts counting from 0)
	75%	max
continent		
Africa	54.41150	76.442
Americas	71.69950	80.653

Asia	69.50525	82.603
Europe	75.45050	81.757
Oceania	77.55250	81.235

10.2.3 Aggregation Functions

You can also use an aggregation function that is not listed “pandas method” column in [Table 10-1](#). Instead of directly calling the aggregation method, we call the `agg` or `aggregate` method, and pass the aggregation function we want in there. When using `agg` or `aggregate`, you will use the functions listed in the “numpy/scipy function” column in [Table 10-1](#).

10.2.3.1 Functions from other libraries

We can use the `mean` function from the `numpy` library.

```
# import the numpy library
import numpy as np

# calculate the average life expectancy by continent
# but use the np.mean function
cont_le_agg = df.groupby('continent').lifeExp.agg(np.mean)
print(cont_le_agg)

continent
Africa      48.865330
Americas    64.658737
Asia        60.064903
Europe      71.903686
Oceania     74.326208
Name: lifeExp, dtype: float64

# agg and aggregate do the same thing
cont_le_agg2 = df.groupby('continent').lifeExp.aggregate(np.mean)
print(cont_le_agg2)

continent
Africa      48.865330
Americas    64.658737
Asia        60.064903
Europe      71.903686
Oceania     74.326208
Name: lifeExp, dtype: float64
```

10.2.3.2 Custom user functions

There will be times when we will want to make a calculation that is not provided by pandas or another library. We can write our own function that performs the calculation we want and use it in `aggregate` as well.

Let's create our own mean function. Recall the mean function:

$$\text{mean} = \bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (10.1)$$

```
def my_mean(values):
    """My version of calculating a mean
    """
    # get the total number of numbers for the denominator
    n = len(values)

    # start the sum at 0
    sum = 0
```

```

for value in values:
    # for each value we are adding it to the running sum
    sum += value

# return the summed values divided by the number of values
return(sum / n)

```

Note that the function we wrote only takes 1 parameter, `values`. What gets passed into the function is the entire `series` of values. This is why we need to iterate through them to take the sum.

Also, I could have also calculated the `sum` in the function by using `values.sum()`, which can actually handle missing values better than the way the `for` loop is written.

We can pass our custom function straight into the `agg` or `aggregate` method like before with ‘`np.mean`’.

```

agg_my_mean = df.groupby('year').lifeExp.agg(my_mean)
print(agg_my_mean)

year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64

```

Finally, we can also write functions that take multiple parameters. So long as the first parameter takes the `series` of values from the data frame, you pass the other arguments as keywords into `agg` or `aggregate`.

In the following example, we will calculate the global average for average life expectancy, `diff_value`, and subtract it from each grouped value.

```

def my_mean_diff(values, diff_value):
    """Difference between the mean and diff_value
    """
    n = len(values)
    sum = 0
    for value in values:
        sum += value
    mean = sum / n
    return(mean - diff_value)

# calculate the global average life expectancy mean
global_mean = df.lifeExp.mean()
print(global_mean)

59.4744393662

# custom aggregation function with multiple parameters
agg_mean_diff = df.groupby('year').lifeExp.\
    agg(my_mean_diff, diff_value=global_mean)
print(agg_mean_diff)

year
1952    -10.416820
1957     -7.967038
1962     -5.865190
1967     -3.796150

```

```

1972      -1.827053
1977       0.095718
1982       2.058758
1987       3.738173
1992       4.685899
1997       5.540237
2002       6.220483
2007       7.532983
Name: lifeExp, dtype: float64

```

10.2.4 Multiple functions simultaneously

When you want to calculate multiple aggregation functions, we can pass the individual functions into `agg` or `aggregate` as a python `list`. Examples of functions you can use here are listed in the “numpy/scipy function” column in [Table 10-1](#)

```

# calculate the count, mean, sd of the lifeExp by continent
gdf = df.groupby('year').lifeExp.\
    agg([np.count_nonzero, np.mean, np.std])
print(gdf)
   count_nonzero      mean        std
year
1952           142.0  49.057620  12.225956
1957           142.0  51.507401  12.231286
1962           142.0  53.609249  12.097245
1967           142.0  55.678290  11.718858
1972           142.0  57.647386  11.381953
1977           142.0  59.570157  11.227229
1982           142.0  61.533197  10.770618
1987           142.0  63.212613  10.556285
1992           142.0  64.160338  11.227380
1997           142.0  65.014676  11.559439
2002           142.0  65.694923  12.279823
2007           142.0  67.007423  12.073021

```

10.2.5 Using a dict in agg/aggregate

There are some other ways you can apply functions in the `agg` and `aggregate` methods. You can pass `agg` a python dictionary. However, the results will differ depending on if you are aggregating directly on a `DataFrame`, or a `Series` object. The latter is deprecated.

10.2.5.1 On a DataFrame

When specifying a `dict` on a grouped `DataFrame`, the keys are the columns of the `DataFrame`, and the values are the functions used in the aggregated calculation. This allows you to group on a variable/variables and have a different aggregation function performed on different columns simultaneously.

```

# use a dictionary on a dataframe to agg different columns
# for each year, calculate the
# average lifeExp, median pop, and median gdpPerCap
gdf_dict = df.groupby('year').agg({
    'lifeExp': 'mean',
    'pop': 'median',
    'gdpPerCap': 'median'
})
print(gdf_dict)

   lifeExp        pop      gdpPerCap
year
1952  49.057620  3943953.0  1968.528344
1957  51.507401  4282942.0  2173.220291
1962  53.609249  4686039.5  2335.439533
1967  55.678290  5170175.5  2678.334741
1972  57.647386  5877996.5  3339.129407

```

```

1977  59.570157  6404036.5  3798.609244
1982  61.533197  7007320.0  4216.228428
1987  63.212613  7774861.5  4280.300366
1992  64.160338  8688686.5  4386.085502
1997  65.014676  9735063.5  4781.825478
2002  65.694923  10372918.5  5319.804524
2007  67.007423  10517531.0  6124.371109

```

10.2.5.2 On a Series

Passing a `dict` into a `Series` after a `groupby`, used to allow you to directly calculate aggregate statistics as the value, and the key of the `dict` would be the new column name. However, this notation is not consistent with the behavior when `dicts` are passed into grouped `DataFrames` like the example in [10.2.5.1](#). In order to have user-defined column names in the output of a grouped series calculation, you need to `rename` them after the fact.

```

gdf = df.groupby('year')['lifeExp'].\
agg([np.count_nonzero,
     np.mean,
     np.std,]).\
rename(columns={'count_nonzero': 'count',
               'mean': 'avg',
               'std': 'std_dev'}).\
reset_index() # return a flat dataframe
print(gdf)

   year  count      avg    std_dev
0  1952  142.0  49.057620  12.225956
1  1957  142.0  51.507401  12.231286
2  1962  142.0  53.609249  12.097245
3  1967  142.0  55.678290  11.718858
4  1972  142.0  57.647386  11.381953
5  1977  142.0  59.570157  11.227229
6  1982  142.0  61.533197  10.770618
7  1987  142.0  63.212613  10.556285
8  1992  142.0  64.160338  11.227380
9  1997  142.0  65.014676  11.559439
10 2002  142.0  65.694923  12.279823
11 2007  142.0  67.007423  12.073021

```

10.3 Transform

When we transform data, we pass values from our dataframe into a function. The function will then “transform” the data. Unlike `aggregate` that would take multiple values, and return a single (aggregated) value, `transform` takes multiple values, and returns one-to-one transformation of the values. That is, it does not reduce the amount of data.

10.3.1 Z-score Example

Let’s calculate the z-score of our life expectancy data by year. The z-score will calculate the number of standard deviations from the mean of our data. It centers our data around 0 with a standard deviation of 1. This is a technique that standardizes our data and makes it easier to compare different variables to each other. Here’s the formula for calculating z-score

$$z = \frac{x - \mu}{\sigma} \quad (10.2)$$

- x is the a data point in our dataset
- μ is the average of our dataset, as calculated by Equation 10.1

- σ is the standard deviation as calculated by 10.3

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2} \quad (10.3)$$

Let's write a python function that calculates a z-score

```
def my_zscore(x):
    '''calculates the z-score of provided data
    `x` is a vector or series of values.
    '''
    return((x - x.mean()) / x.std())
```

Now we can use this function to `transform` our data by group.

```
transform_z = df.groupby('year').lifeExp.transform(my_zscore)
```

Note the shape of our original dataframe, and the `transformz`, they have the same number of rows and data.

```
# note the number of rows in our data
print(df.shape)
(1704, 6)

# note the number of values in our transformation
print(transform_z.shape)
(1704, )
```

The `scipy` library has its own `zscore` function. Let's use the `zscore` function in a `groupby transform` and not in a `groupby`.

```
# import the zscore function from scipy.stats
from scipy.stats import zscore

# calculate a grouped zscore
sp_z_grouped = df.groupby('year').lifeExp.transform(zscore)

# calculate a non grouped zscore
sp_z_nogroup = zscore(df.lifeExp)
```

You will see that not all the zscore values are the same

```
# grouped z-score
print(transform_z.head())

0    -1.656854
1    -1.731249
2    -1.786543
3    -1.848157
4    -1.894173
Name: lifeExp, dtype: float64

# grouped z-score using scipy
print(sp_z_grouped.head())

0    -1.662719
1    -1.737377
2    -1.792867
3    -1.854699
4    -1.900878
Name: lifeExp, dtype: float64
```

```
# non grouped z-score
print(sp_z_nogroup[:5])

[-2.37533395 -2.25677417 -2.1278375 -1.97117751 -1.81103275]
```

We can see our grouped results are similar. But when we calculate the z-score outside the `groupby`, we get the z-score calculated on the entire data, not broken up by group.

10.3.1.1 Missing value example

[Chapter 5](#) covered missing values and how we can fill in missing values. In the Ebola example from the chapter, it made more sense to fill our data using the `interpolate` method, or forward/backward filling our data.

In certain datasets, filling the missing values with the mean of the column could also make sense, however, sometimes it may make more sense to fill missing data based on a particular group. Let's work with the `tips` dataset that comes from the `seaborn` library.

```
import seaborn as sns
import numpy as np

# set the seed so results are deterministic
np.random.seed(42)

# sample 10 rows from tips
tips_10 = sns.load_dataset('tips').sample(10)

# randomly pick 4 `total_bill` values and turn them into missing
tips_10.loc[np.random.permutation(tips_10.index)[:4],
            'total_bill'] = np.Nan

print(tips_10)

   total_bill  tip    sex smoker  day time  size
24      19.82  3.18   Male     No Sat Dinner     2
6       8.77  2.00   Male     No Sun Dinner     2
153      NaN  2.00   Male     No Sun Dinner     4
211      NaN  5.16   Male    Yes Sat Dinner     4
198      NaN  2.00 Female   Yes Thur Lunch     2
176      NaN  2.00   Male    Yes Sun Dinner     2
192     28.44  2.56   Male    Yes Thur Lunch     2
124     12.48  2.52 Female   No Thur Lunch     2
9      14.78  3.23   Male     No Sun Dinner     2
101     15.38  3.00 Female   Yes Fri Dinner     2
```

[Chapter 5](#) showed how you can use the `fillna` method to fill in the missing values. However, we may not want to just fill the missing values with the `mean` of `totalbill`. Maybe the `Male` and `Female` values in the `sex` column have different spending habits, or `totalbill` values differ between time of day, `time`, and `size` of the table. These are all valid concerns when processing our data.

We can use the `groupby` statement to calculate a statistic to fill in missing values. Instead of using `agg` or `aggregate`, we use the `transform` method. First let's count the non-missing values by `sex`.

```
count_sex = tips_10.groupby('sex').count()
print(count_sex)

   total_bill  tip  smoker  day  time  size
sex
Male          4    7      7    7      7      7
Female        2    3      3    3      3      3
```

You see we have 3 missing values for `Male`, and 1 missing value for `Female`. Now let's calculate a grouped average, and use the grouped average to fill in the missing values.

```

def fill_na_mean(x):
    '''Returns the average of a given vector
    '''
    avg = x.mean()
    return(x.fillna(avg))

# calculate a mean `total_bill` by `sex`
total_bill_group_mean = tips_10.\n
    groupby('sex').\n
    total_bill.\n
    transform(fill_na_mean)

# assign to a new column in the original data
# you can also replace the original column by using `total_bill`
tips_10['fill_total_bill'] = total_bill_group_mean

print(tips_10)

      total_bill   tip     sex smoker  day   time  size  \
24        19.82  3.18   Male     No  Sat Dinner     2
6          8.77  2.00   Male     No  Sun Dinner     2
153       NaN  2.00   Male     No  Sun Dinner     4
211       NaN  5.16   Male    Yes  Sat Dinner     4
198       NaN  2.00  Female    Yes Thur Lunch     2
176       NaN  2.00   Male    Yes  Sun Dinner     2
192       28.44  2.56   Male    Yes Thur Lunch     2
124       12.48  2.52  Female    No Thur Lunch     2
9          14.78  3.23   Male     No  Sun Dinner     2
101       15.38  3.00  Female    Yes  Fri Dinner     2

      fill_total_bill
24            19.8200
6             8.7700
153           17.9525
211           17.9525
198           13.9300
176           17.9525
192           28.4400
124           12.4800
9             14.7800
101           15.3800

```

If we just look at the two `totalbill` column, you can see a different value was filled in for the `NaN` missing values

```

print(tips_10[['sex', 'total_bill', 'fill_total_bill']])
      sex  total_bill  fill_total_bill
24  Male      19.82      19.8200
6   Male       8.77      8.7700
153  Male       NaN      17.9525
211  Male       NaN      17.9525
198 Female      NaN      13.9300
176  Male       NaN      17.9525
192  Male      28.44      28.4400
124 Female      12.48      12.4800
9   Male      14.78      14.7800
101 Female      15.38      15.3800

```

10.4 Filter

The last type of action you can perform with a `groupby` is filtering. This allows you to split your data by keys, and then perform some kind of boolean subsetting on the data. Like all the examples with the `groupby`, you can accomplish the same thing by using regular subsetting as described in Chapter [1.3](#) and [2.4.1](#). Let's use the full tips dataset, and look at the number of observations for the various `size`

```
# load the tips dataset
tips = sns.load_dataset('tips')
```

```

# note the number of rows in the original data
print(tips.shape)
(244, 7)
# look at the frequency counts for the table size
print(tips['size'].value_counts())
2    156
3     38
4     37
5      5
6      4
1      4
Name: size, dtype: int64

```

You see here that table sizes of **1**, **5**, and **6** are infrequent. Depending on your needs, you may want to filter those data points out. In this example, we want each group to greater than or equal to 30 observations.

To do so we can use the **filter** method on a grouped operation.

```

# filter the data such that each group has more than 30 observations
tips_filtered = tips.\
    groupby('size').\
    filter(lambda x: x['size'].count() >= 30)

```

And you can see that our dataset was filtered down.

```

print(tips_filtered.shape)
(231, 7)

print(tips_filtered['size'].value_counts())
2    156
3     38
4     37
Name: size, dtype: int64

```

10.5 The pandas.core .groupby.DataFrameGroupBy object

aggregate, **transform**, and **filter** are common ways to work with grouped objects in pandas. In this section, we will go into some of the inner working of grouped objects. The **groupby** documentation¹ is an excellent resource on some of the more nuanced features of **groupby**

¹Groupby documentation: <http://pandas.pydata.org/pandas-docs/stable/groupby.html>

10.5.1 Groups

Throughout the chapter, we've directly chained **aggregate**, **transform**, or **filter** after the **groupby**. However, we can actually save the results of **groupby** before performing those other methods. We will start with the subsetted **tips** dataset.

```

tips_10 = sns.load_dataset('tips').sample(10, random_state=42)
print(tips_10)

   total_bill  tip    sex smoker  day time  size
24       19.82  3.18   Male     No  Sat Dinner     2
6        8.77  2.00   Male     No  Sun Dinner     2
153      24.55  2.00   Male     No  Sun Dinner     4
211      25.89  5.16   Male    Yes  Sat Dinner     4
198      13.00  2.00 Female   Yes Thur Lunch     2
176      17.89  2.00   Male    Yes  Sun Dinner     2
192      28.44  2.56   Male    Yes Thur Lunch     2
124      12.48  2.52 Female   No Thur Lunch     2

```

```

9      14.78  3.23   Male    No   Sun Dinner     2
101     15.38  3.00 Female   Yes   Fri Dinner     2

```

We can choose to save just the `groupby` object without running any other `aggregate`, `transform`, or `filter` method on it.

```

# save just the grouped object
grouped = tips_10.groupby('sex')

# note that we just get back the object and its memory location
print(grouped)

<pandas.core.groupby.DataFrameGroupBy object at 0x7fd0ddd73588>

```

You can see that when we try to print out the `grouped` result, we only get a memory reference back and the data type is a pandas `DataFrameGroupBy` object. Under the hood nothing has been actually calculated yet, because we never performed an action that requires a calculation. If we want to actually see the calculated groups, we can call the `groups` attribute.

```

# see the actual groups of the groupby
# note it only returns the index
print(grouped.groups)

{'Male': Int64Index([24, 6, 153, 211, 176, 192, 9], dtype='int64'),
 'Female': Int64Index([198, 124, 101], dtype='int64')}

```

Notice that even when we ask for the `groups` from our `grouped` object, we only get the `index` of the dataframe back. Think of this as the row numbers. This are mainly done to optimize performance. Again, we haven't calculated anything yet.

However, this allows you to save just the grouped result, and then perform multiple `aggregate`, `transform`, or `filter` statements without having to recalculate the `groupby` statement again.

10.5.2 Group calculations of multiple variables

One of the nice things about Python is that it follows the EAFP² mantra of: “easier to ask for forgiveness than permission”. Throughout the chapter we have been performing `groupby` calculations on a single column. But if we specified the calculation we want right after the `groupby` it will perform the calculation on all the columns it can, and silently drop the rest. Here's an example of a grouped mean on all the columns by `sex`.

²EAFP: <https://docs.python.org/3/glossary.html#term-eafp>

```

# calculate the mean on relevant columns
avgs = grouped.mean()
print(avgs)

      total_bill      tip      size
sex
Male        20.02  2.875714  2.571429
Female      13.62  2.506667  2.000000

```

You can see not all the columns reported a mean.

```

# list all the columns
print(tips_10.columns)

Index(['total_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size'],
      dtype='object')

```

The `smoker`, `day`, and `time` columns were not returned in the results because it makes no sense to

take the average between `Dinner` and `Lunch`.

10.5.3 Selecting a group

If we want to get a particular group out, we can use the `get_group` method, and pass in the group that we want. For example, if we wanted the `Female` values:

```
# get the `Female` group
female = grouped.get_group('Female')
print(female)

   total_bill  tip    sex smoker  day    time  size
198      13.00  2.00  Female    Yes Thur  Lunch     2
124      12.48  2.52  Female     No Thur  Lunch     2
101      15.38  3.00  Female    Yes   Fri Dinner     2
```

10.5.4 Iterating through groups

Another benefit for saving just the `groupby` object is that you will be able to iterate though the groups individually. There might be times where it's easier to conceptualize a question using a 'for' loop, rather than trying to formulate an `aggregate`, `transform`, or `filter` statement. It's more important to get something done, then you can work on optimizing for speed.

We can iterate through our `grouped` values just like any other container in python, with a `for` loop.

```
for sex_group in grouped:
    print(sex_group)

('Male',   total_bill  tip    sex smoker  day    time  size
24      19.82  3.18  Male    No Sat Dinner     2
6       8.77  2.00  Male    No Sun Dinner     2
153     24.55  2.00  Male    No Sun Dinner     4
211     25.89  5.16  Male    Yes Sat Dinner     4
176     17.89  2.00  Male    Yes Sun Dinner     2
192     28.44  2.56  Male    Yes Thur Lunch     2
9        14.78  3.23  Male    No Sun Dinner     2)
('Female', total_bill  tip    sex smoker  day    time  size
198      13.00  2.00  Female    Yes Thur  Lunch     2
124      12.48  2.52  Female     No Thur  Lunch     2
101      15.38  3.00  Female    Yes   Fri Dinner     2)
```

If you try to just get the first index from the `grouped` object, you will get an error because it is still a `pandas.core.groupby.DataFrameGroupBy` object, and not really a python container.

```
# you can't really get the 0 element out of it like a container
print(grouped[0])

Traceback (most recent call last):
  File "<ipython-input-1-acdbc5d1f67a>", line 2, in <module>
    print(grouped[0])
  KeyError: 'Column not found: 0'
```

For now, let's modify the `for` loop, to just show the first element, and some of the things we get when we loop over the `grouped` object.

```
for sex_group in grouped:
    # get the type of the object (tuple)
    print('the type is: {}\\n'.format(type(sex_group)))

    # get the len of the object (2 elements)
    print('the length is: {}\\n'.format(len(sex_group)))

    # get the first element
    first_element = sex_group[0]
```

```

print('the first element is: {}'.format(first_element))

# the type of the first element (string)
print('it has a type of: {}'.format(type(sex_group[0])))

# get the second element
second_element = sex_group[1]
print('the second element is:\n{}'.format(second_element))

# get the type of the second element (dataframe)
print('it has a type of: {}'.format(type(second_element)))

# print what we have
print('what we have:')
print(sex_group)

# stop after first iteration
break

the type is: <class 'tuple'>
the length is: 2
the first element is: Male
it has a type of: <class 'str'>
the second element is:
   total_bill  tip  sex smoker  day    time  size
24      19.82  3.18  Male     No  Sat  Dinner    2
6       8.77  2.00  Male     No  Sun  Dinner    2
153     24.55  2.00  Male     No  Sun  Dinner    4
211     25.89  5.16  Male    Yes  Sat  Dinner    4
176     17.89  2.00  Male    Yes  Sun  Dinner    2
192     28.44  2.56  Male    Yes Thur  Lunch    2
9       14.78  3.23  Male     No  Sun  Dinner    2

it has a type of: <class 'pandas.core.frame.DataFrame'>
what we have:
('Male',      total_bill  tip  sex smoker  day    time  size
24      19.82  3.18  Male     No  Sat  Dinner    2
6       8.77  2.00  Male     No  Sun  Dinner    2
153     24.55  2.00  Male     No  Sun  Dinner    4
211     25.89  5.16  Male    Yes  Sat  Dinner    4
176     17.89  2.00  Male    Yes  Sun  Dinner    2
192     28.44  2.56  Male    Yes Thur  Lunch    2
9       14.78  3.23  Male     No  Sun  Dinner    2)

```

We have a 2 element **tuple** where the first element is a **str**, string, that represents the **Female** key, and the second element is a **DataFrame** of the **Female** data.

If you want you can forgo all the techniques from this chapter and iterate through your grouped values in this manner to perform your calculations. Again, there may be times where this is the only way to get something done. Maybe you have a complicated conditional you want to check for each group, or you want to write out each group into separate files, the option is available to you if you need to iterate through the groups one at a time.

10.5.5 Multiple groups

Throughout the chapter I only showed one variable in the **groupby** statement. However, we can add multiple variables during the **groupby** process. [Section 1.4.1](#) briefly showed this.

Let's say we want to calculate the **mean** of our data by **sex**, time of day (**time**), and day of week (**day**) in the **tips** dataset. We can pass in `['sex', 'time']` as a python **list** instead of the single string we have been using.

```
# mean by sex and time
```

```

bill_sex_time = tips_10.groupby(['sex', 'time'])

group_avg = bill_sex_time.mean()
print(group_avg)

      total_bill      tip      size
sex   time
Male  Lunch    28.440000  2.560000  2.000000
      Dinner   18.616667  2.928333  2.666667
Female Lunch   12.740000  2.260000  2.000000
      Dinner   15.380000  3.000000  2.000000

```

10.5.6 Flattening the results

The final topic I want to cover is the results from the `groupby`. Let's look at the type of the `group_avg` we just calculated.

```

# type of the group_avg
print(type(group_avg))

<class 'pandas.core.frame.DataFrame'>

```

We have a `DataFrame` however, the results look a little strange: we have what appears to be empty cells in your dataframe.

If we look at the `columns`, we get what we expect

```

print(group_avg.columns)

Index(['total_bill', 'tip', 'size'], dtype='object')

```

However, the interesting things come from when we look at the `index`

```

print(group_avg.index)

MultiIndex(levels=[[ 'Male', 'Female'], ['Lunch', 'Dinner']],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
           names=['sex', 'time'])

```

We actually have something called a `MultiIndex`. If we wanted to get a regular flat dataframe back, we can either call the `resetIndex` method on the results.

```

group_method = tips_10.groupby(['sex', 'time']).mean().reset_index()
print(group_method)

      sex      time  total_bill      tip      size
0     Male    Lunch    28.440000  2.560000  2.000000
1     Male   Dinner   18.616667  2.928333  2.666667
2   Female    Lunch   12.740000  2.260000  2.000000
3   Female   Dinner   15.380000  3.000000  2.000000

```

Or we can use the `as_index=False` parameter in the `groupby` method (it is `True` by default).

```

group_param = tips_10.groupby(['sex', 'time'], as_index=False).mean()
print(group_param)

      sex      time  total_bill      tip      size
0     Male    Lunch    28.440000  2.560000  2.000000
1     Male   Dinner   18.616667  2.928333  2.666667
2   Female    Lunch   12.740000  2.260000  2.000000
3   Female   Dinner   15.380000  3.000000  2.000000

```

10.6 Working with a MultiIndex

There will be times when you want to chain calculations after a `groupby`. You can always flatten the results, and perform another `groupby` statement, but that may not always be the most efficient way of performing the calculation.

We begin with epidemiological simulation data of Influenza in Chicago (note this is a fairly large dataset).

```
intv_df = pd.read_csv('..../data/epi_sim.txt')

# the number of rows is over 9 million!
print(intv_df.shape)

(9434653, 6)
```

There are 6 columns in the dataset:

1. `ig-type` : edge type (type of relationship between 2 nodes in the network, e.g., school, work)
2. `intervened` : time in the simulation an intervention occurred for a given person (pid).
3. `pid`: simulated person's ID number
4. `rep`: replication run (each set of simulation parameters were run multiple times)
5. `sid` : simulation ID
6. `tr` : transmissibility value of the Influenza virus

```
print(intv_df.head())

   ig_type  intervened      pid  rep    sid      tr
0         3          40  294524448    1    201  0.000135
1         3          40  294571037    1    201  0.000135
2         3          40  290699504    1    201  0.000135
3         3          40  288354895    1    201  0.000135
4         3          40  292271290    1    201  0.000135
```

About the epidemiological simulation dataset

This dataset comes from a simulation using a program called Indemics (Interactive Epidemic Simulation):

<http://ndssl.vbi.vt.edu/apps/Modeling.html>. It was developed by the Network Dynamics and Simulation Science Laboratory at Virginia Tech: <https://www.bi.vt.edu/ndssl>.

The references for the program are:

- Bisset KR, Chen J, Deodhar S, Feng X, Ma Y, Marathe MV. Indemics: An Interactive High-Performance Computing Framework for Data Intensive Epidemic Modeling. ACM transactions on modeling and computer simulation: a publication of the Association for Computing Machinery. 2014;24(1):10.1145/2501602. doi:10.1145/2501602.
- Suruchi Deodhar, Keith Bisset, Jiangzhuo Chen, Yifei Ma, and Madhav V. Marathe. Enhancing software capability through integration of distinct software in epidemiological systems. To appear in the 2nd ACM SIGHIT International Health Informatics Symposium, 2012.
- Keith R. Bisset, Jiangzhuo Chen, Xizhou Feng, Yifei Ma, and Madhav V. Marathe. Indemics:

an interactive data intensive framework for high performance epidemic simulation. In Proc. the 24rd international conference on Conference on Supercomputing, pages 233242, 2010.

Let's do a count of number of interventions for each replicate, intervention time, and treatment value. Here, we are counting the `ig_type` arbitrarily. We just need a value to get a count of observations for the groups.

```
count_only = intv_df.\n    groupby(['rep', 'intervened', 'tr'])['ig_type'].\\n    count()\nprint(count_only.head(n=10))\n\nrep           rep\n      intervened      tr\n0   8       0.000166     1\n    9       0.000152     3\n    0       0.000166     1\n  10      0.000152     1\n    0       0.000166     1\n  12      0.000152     3\n    0       0.000166     5\n  13      0.000152     1\n    0       0.000166     3\n  14      0.000152     3\nName: ig_type, dtype: int64
```

Now that we've done a `groupby count`, we can do an additional `groupby` that calculates the average value. However, the results of our initial `groupby`, does not return a regular flat dataframe back.

```
print(type(count_only))\n<class 'pandas.core.series.Series'>
```

It results are actually in a multi-index series. If we want to do another groupby, we have to pass in the `levels` parameter to refer to the multi-index levels. Here we pass in `[0, 1, 2]` for the first, second, and third index levels.

```
count_mean = count_only.\n    groupby(level=[0, 1, 2]).\\n    mean()\nprint(count_mean.head())\n\nrep      intervened      tr\n0   8       0.000166     1\n    9       0.000152     3\n    0       0.000166     1\n  10      0.000152     1\n    0       0.000166     1\nName: ig_type, dtype: int64
```

We can combine these all in a single command.

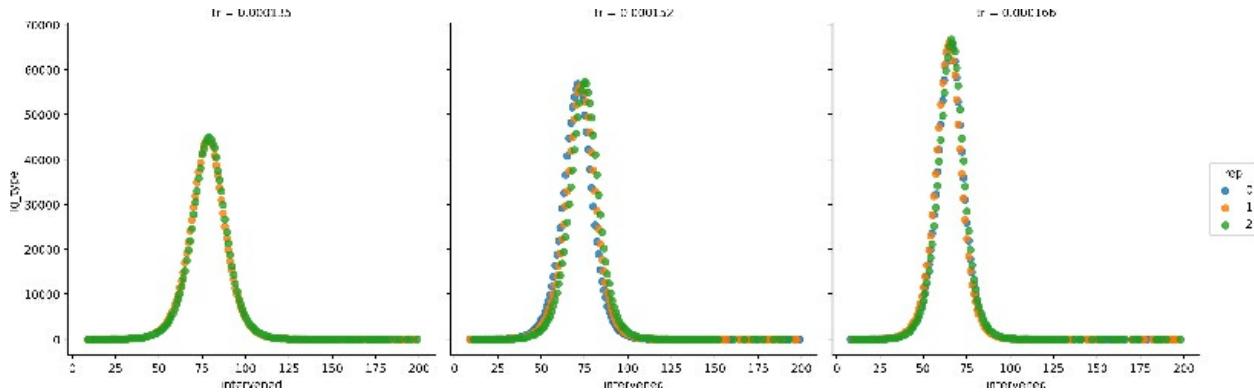
```
count_mean = intv_df.\n    groupby(['rep', 'intervened', 'tr'])['ig_type'].\\n    count().\\n    groupby(level=[0, 1, 2]).\\n    mean()
```

[Figure 10-1](#) show what our results.

```
import seaborn as sns\nimport matplotlib.pyplot as plt\n\nfig = sns.lmplot(x='intervened', y='ig_type', hue='rep', col='tr',\n                  fit_reg=False, data=count_mean.reset_index(),)
```

```
plt.show()
```

Figure 10-1: Grouped counts and mean

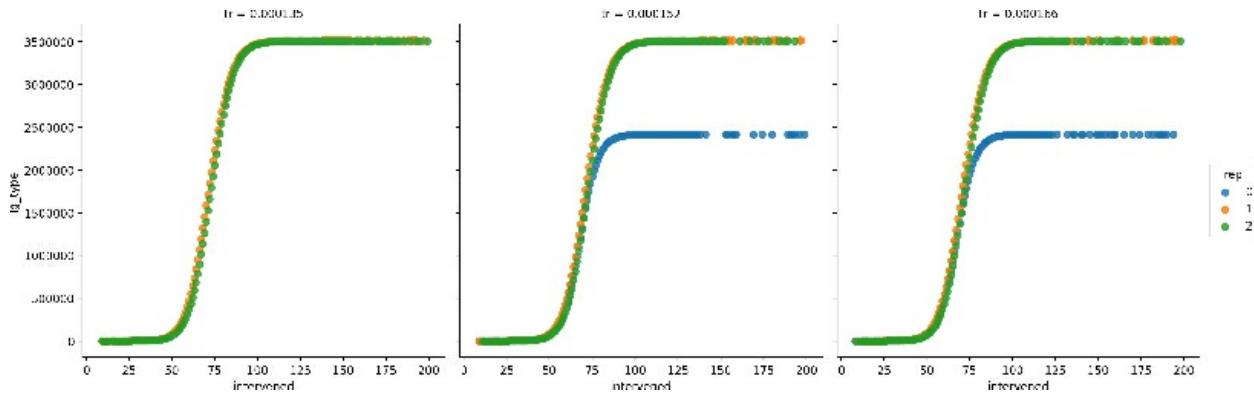


The previous example showed how we can pass in a `level` to do an additional `groupby` calculation. However, it used integer positions, we can also pass in the string of the level to make our code a bit more readable.

Here, instead of looking at the `mean`, we will be using `cumsum`, for the cumulative sum. [Figure 10-2](#) shows our results.

```
cumulative_count = intv_df.\n    groupby(['rep', 'intervened', 'tr'])['ig_type'].\\\n    count().\\\n    groupby(level=['rep']).\\\n    cumsum().\\\n    reset_index()\nfig = sns.lmplot(x='intervened', y='ig_type', hue='rep', col='tr',\n                  fit_reg=False, data=cumulative_count)\nplt.show()
```

Figure 10-2: Grouped cumulative counts. The plot shows that one of the replicates did not run in our simulation.



10.7 Conclusion

The `groupby` follows the mantra of “split-apply-combine”. It is a powerful concept that is not necessarily new to data analytics, but will help you think about your data and pipelines in a different way that will scale to more “big data” systems, such as distributed computing.

I urge you to check out the documentation for the `groupby` method³ and the general

documentation for `groupby`⁴, as there are many more complex things you can do with `groupby` statements. What I've covered in this chapter should serve the vast majority of needs and use cases.

³Groupby method: <http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.groupby.html>

⁴Groupby: <http://pandas.pydata.org/pandas-docs/stable/groupby.html>

Chapter 11. The `dateti` Data Type

11.1 Introduction

One of the biggest reasons for using pandas is its ability to work with timeseries data. We've already seen some of this capability when we concatenated data in [chapter 4](#) and saw how the indices automatically aligned themselves. This chapter will focus on the more common tasks when working with data that involve dates and times.

Objectives

This chapter will cover:

1. Python's built-in `datetime` library
2. Converting strings into a date
3. Formating dates
4. Extracting date components
5. Calculations with dates
6. How to work with dates in a `DataFrame`
7. Resampling
8. Working with timezones

11.2 Python's `datetime`

Python has a built-in `datetime` object from the `datetime` library.

```
from datetime import datetime
```

We can use `datetime` to get the current date and time.

```
now = datetime.now()
print(now)

2017-09-14 23:16:37.647327
```

We can also create our own `datetime` manually.

```
t1 = datetime.now()
t2 = datetime(1970, 1, 1)
```

And do datetime math.

```
diff = t1 - t2
print(diff)
```

```
17423 days, 23:16:37.671703
```

You can see here the data type of a date calculation is a `timedelta`

```
print(type(diff))
<class 'datetime.timedelta'>
```

We can perform these types of actions when working within a pandas dataframe.

11.3 Converting to datetime

Converting an object type into a `datetime` type is done with the `to_datetime` function¹. We'll load up our Ebola dataset and convert the `Date` column into a proper `datetime` object.

```
import pandas as pd
ebola = pd.read_csv('../data/country_timeseries.csv')

# top left corner of the data
print(ebola.iloc[:5, :5])

      Date    Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0  1/5/2015   289     2776.0        NaN          10030.0
1  1/4/2015   288     2775.0        NaN          9780.0
2  1/3/2015   287     2769.0      8166.0          9722.0
3  1/2/2015   286        NaN      8157.0          NaN
4 12/31/2014   284     2730.0      8115.0          9633.0
```

You can see the first `Date` column contains date information, however, the `info` tells us it is actually encoded as a generic string `object` in pandas.

¹`to_datetime` documentation: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.to_datetime.html

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
Date              122 non-null object
Day               122 non-null int64
Cases_Guinea      93 non-null float64
Cases_Liberia     83 non-null float64
Cases_SierraLeone 87 non-null float64
Cases_Nigeria     38 non-null float64
Cases_Senegal      25 non-null float64
Cases_UnitedStates 18 non-null float64
Cases_Spain        16 non-null float64
Cases_Mali         12 non-null float64
Deaths_Guinea      92 non-null float64
Deaths_Liberia     81 non-null float64
Deaths_SierraLeone 87 non-null float64
Deaths_Nigeria     38 non-null float64
Deaths_Senegal      22 non-null float64
Deaths_UnitedStates 18 non-null float64
Deaths_Spain        16 non-null float64
Deaths_Mali         12 non-null float64
dtypes: float64(16), int64(1), object(1)
memory usage: 17.2+ KB
None
```

We can create a new column, `date_dt`, that converts the `Date` column into a `datetime`.

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

We can be a little more explicit with how we convert into a `datetime` object. The `to_datetime` has a parameter called `format` that allows you to manually specify the `format` of the date you are hoping to parse. Since our date is in the format of `month/day/year` we can pass in the string `%m/%d/%Y`.

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'], format='%m/%d/%Y')
```

In both cases, we end up with a new column with a `datetime` type.

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 19 columns):
Date          122 non-null object
Day           122 non-null int64
Cases_Guinea   93 non-null float64
Cases_Liberia   83 non-null float64
Cases_SierraLeone 87 non-null float64
Cases_Nigeria   38 non-null float64
Cases_Senegal    25 non-null float64
Cases_UnitedStates 18 non-null float64
Cases_Spain     16 non-null float64
Cases_Mali       12 non-null float64
Deaths_Guinea    92 non-null float64
Deaths_Liberia   81 non-null float64
Deaths_SierraLeone 87 non-null float64
Deaths_Nigeria   38 non-null float64
Deaths_Senegal    22 non-null float64
Deaths_UnitedStates 18 non-null float64
Deaths_Spain     16 non-null float64
Deaths_Mali       12 non-null float64
date_dt         122 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(16), int64(1), object(1)
memory usage: 18.2+ KB
None
```

There are some built-in convenient options in the `to_datetime` function. There is a `dayfirst` and `yearfirst` that you can set to `True` if the date format begins with a day, e.g., `31-03-2014`, or if the date begins with a year, e.g., `2014-03-31`.

For other date formats you can manually specify how they are represented using the syntax specified by python's `strptime`² which has been replicated in [Table 11-1](#).

²`strptime` behavior: <https://docs.python.org/3.6/library/datetime.html#strftime-and-strptime-behavior>

11.4 Loading data with dates

For the most part, many of the datasets used in the book are in a `csv` format, or comes from the `seaborn` library. The `gapminder` dataset was an exception, in that it was a tab separated file (`tsv`). The `read_csv` function has a lot of parameters³. There are `parse_dates`, `infer_datetime_format`, `keep_date_col`, `date_parser`, `dayfirst`. We can parse the `Date` column directly by specifying the column we want in the `parse_dates` parameter.

³`read_csv` documentation: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

```
ebola = pd.read_csv('../data/country_timeseries.csv', parse_dates=[0])
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
```

```

RangeIndex: 122 entries, 0 to 121
Data columns (total 18 columns):
Date                122 non-null datetime64[ns]
Day                 122 non-null int64
Cases_Guinea        93 non-null float64
Cases_Liberia       83 non-null float64
Cases_SierraLeone   87 non-null float64
Cases_Nigeria       38 non-null float64
Cases_Senegal        25 non-null float64
Cases_UnitedStates  18 non-null float64
Cases_Spain          16 non-null float64
Cases_Mali           12 non-null float64

```

Table 11–1: Python strftime behavior

Directive	Meaning	Example
%a	Weekday abbreviated name	Sun, Mon, ..., Sat
%A	Weekday full name	Sunday, Monday, ..., Saturday
%w	Weekday as a number, where 0 is Sunday	0, 1, ..., 6
%d	Day of the month as a 2-digit number	01, 02, ..., 31
%b	Month abbreviated name	Jan, Feb, ..., Dec
%B	Month full name	January, February, ..., December
%m	Month as a 2-digit number	01, 02, ..., 12
%y	Year as a 2-digit number	00, 01, ..., 99
%Y	Year as a 4-digit number	0001, 0002, ..., 2013, 2014, ..., 9999
%H	Hour (24-hour clock) as a 2-digit number	00, 01, ..., 23
%I	Hour (12-hour clock) as a 2-digit number	01, 02, ..., 12
%p	AM or PM	AM, PM
%M	Minute as a 2-digit number	00, 01, ..., 59
%S	Second as a 2-digit number	00, 01, ..., 59
%f	Microsecond as a number	000000, 000001, ..., 999999
%z	UTC offset in the form of +HHMM or -HHMM	(empty), +0000, -0400, +1030
%Z	Time zone name	(empty), UTC, EST, CST
%j	Day of the year as a 3-digit number	001, 002, ..., 366
%U	Week number of the year (Sunday first)	00, 01, ..., 53
%W	Week number of the year (Monday first)	00, 01, ..., 53
%c	Date and time representation	Tue Aug 16 21:30:00 1988
%x	Date representation	08/16/88 (None);08/16/1988
%X	Time representation	21:30:00
%%	Literal '%' character	%
%G	ISO 8601 year	0001, 0002, ..., 2013, 2014, ..., 9999
%u	ISO 8601 weekday	1, 2, ..., 7
%V	ISO 8601 week	01, 02, ..., 53
Deaths_Guinea	92	non-null float64
Deaths_Liberia	81	non-null float64
Deaths_SierraLeone	87	non-null float64

```

Deaths_Nigeria           38    non-null   float64
Deaths_Senegal            22    non-null   float64
Deaths_UnitedStates        18    non-null   float64
Deaths_Spain               16    non-null   float64
Deaths_Mali                12    non-null   float64
dtypes: datetime64[ns](1), float64(16), int64(1)
memory usage: 17.2 KB
None

```

You can see this way we can automatically convert columns into dates directly when the data is loaded.

11.5 Extracting date components

Now that we have a `datetime` object, we can extract various parts of the date, such as year, month, day, etc. Here's an example `datetime`.

```

d = pd.to_datetime('2016-02-29')
print(d)

2016-02-29 00:00:00

```

We can see, that if we pass in a single string, we get a `Timestamp`.

```

print(type(d))
<class 'pandas._libs.tslib.Timestamp'>

```

Now that we have a proper `datetime`, we can access various date components as attributes

```

print(d.year)
2016
print(d.month)
2
print(d.day)
29

```

Similar to when tidied our data in [Chapter 6](#) when we needed to parse a column that stored multiple bits of information and used the `str` accessor to use string methods like `split`. We can do something similar here with `datetime` objects by accessing `datetime` methods using the `dt` accessor⁴. Let's first re-create our `date_dt` column.

⁴Datetime-like properties: <https://pandas.pydata.org/pandas-docs/stable/api.html#datetimelike-properties>

```
ebola['date_dt'] = pd.to_datetime(ebola['Date'])
```

Since we know we can get date components such as year, month, and day by using the `year`, `month`, and `day` attributes, we can do this on a column basis like how we parsed strings in a column using `str`. Here's the `head` of the `Date` and `date_dt` columns.

```
print(ebola[['Date', 'date_dt']].head())
```

	Date	date_dt
0	2015-01-05	2015-01-05
1	2015-01-04	2015-01-04
2	2015-01-03	2015-01-03

```
3      2015-01-02  2015-01-02
4      2014-12-31  2014-12-31
```

We can create a new `year` column based on the `Date` column.

```
ebola['year'] = ebola['date_dt'].dt.year
print(ebola[['Date', 'date_dt', 'year']].head())
```

	Date	date_dt	year
0	2015-01-05	2015-01-05	2015
1	2015-01-04	2015-01-04	2015
2	2015-01-03	2015-01-03	2015
3	2015-01-02	2015-01-02	2015
4	2014-12-31	2014-12-31	2014

Let's finish parsing out our date.

```
ebola['month'], ebola['day'] = (ebola['date_dt'].dt.month,
                                 ebola['date_dt'].dt.day)

print(ebola[['Date', 'date_dt', 'year', 'month', 'day']].head())

    Date      date_dt    year  month  day
0  2015-01-05  2015-01-05  2015      1     5
1  2015-01-04  2015-01-04  2015      1     4
2  2015-01-03  2015-01-03  2015      1     3
3  2015-01-02  2015-01-02  2015      1     2
4  2014-12-31  2014-12-31  2014     12    31
```

You can see when we parse out our dates, but the datatype is not preserved.

```
print(ebola.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 22 columns):
Date           122 non-null datetime64[ns]
Day            122 non-null int64
Cases_Guinea   93 non-null float64
Cases_Liberia  83 non-null float64
Cases_SierraLeone  87 non-null float64
Cases_Nigeria  38 non-null float64
Cases_Senegal   25 non-null float64
Cases_UnitedStates  18 non-null float64
Cases_Spain    16 non-null float64
Cases_Mali     12 non-null float64
Deaths_Guinea  92 non-null float64
Deaths_Liberia 81 non-null float64
Deaths_SierraLeone  87 non-null float64
Deaths_Nigeria 38 non-null float64
Deaths_Senegal  22 non-null float64
Deaths_UnitedStates  18 non-null float64
Deaths_Spain   16 non-null float64
Deaths_Mali    12 non-null float64
date_dt        122 non-null datetime64[ns]
year           122 non-null int64
month          122 non-null int64
day            122 non-null int64
dtypes: datetime64[ns](2), float64(16), int64(4)
memory usage: 21.0 KB
None
```

11.6 Date Calculations and `TimedeltaS`

The benefits of having date objects is to be able to do date calculations. In our data, we have a column named `day` which is how many days into the Ebola outbreak. We can recreate this column using date arithmetic. Here's the bottom left corner of our data.

```
print(ebola.iloc[-5:, :5])
```

	Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
117	2014-03-27	5	103.0	8.0	6.0
118	2014-03-26	4	86.0	NaN	NaN
119	2014-03-25	3	86.0	NaN	NaN
120	2014-03-24	2	86.0	NaN	NaN
121	2014-03-22	0	49.0	NaN	NaN

The first day of the outbreak (in this dataset) is on **2015-03-22**. So, if we wanted to calculate number of days into the outbreak, we can subtract this date from each date. We picked this date because it's the earliest date in our dataset. We can accomplish this by calling the `min` of the column

```
print(ebola['date_dt'].min())
2014-03-22 00:00:00
```

We can use this date in our calculation.

```
ebola['outbreak_d'] = ebola['date_dt'] - ebola['date_dt'].min()
print(ebola[['Date', 'Day', 'outbreak_d']].head())
      Date    Day  outbreak_d
0  2015-01-05   289   289 days
1  2015-01-04   288   288 days
2  2015-01-03   287   287 days
3  2015-01-02   286   286 days
4  2014-12-31   284   284 days
print(ebola[['Date', 'Day', 'outbreak_d']].tail())
      Date    Day  outbreak_d
117  2014-03-27     5      5 days
118  2014-03-26     4      4 days
119  2014-03-25     3      3 days
120  2014-03-24     2      2 days
121  2014-03-22     0      0 days
```

You can see here we when we perform date calculations like this we actually end up with a `timedelta` object.

```
print(ebola.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 122 entries, 0 to 121
Data columns (total 23 columns):
Date                  122 non-null datetime64[ns]
Day                   122 non-null int64
Cases_Guinea          93 non-null float64
Cases_Liberia         83 non-null float64
Cases_SierraLeone    87 non-null float64
Cases_Nigeria        38 non-null float64
Cases_Senegal         25 non-null float64
Cases_UnitedStates   18 non-null float64
Cases_Spain           16 non-null float64
Cases_Mali            12 non-null float64
Deaths_Guinea         92 non-null float64
Deaths_Liberia        81 non-null float64
Deaths_SierraLeone   87 non-null float64
Deaths_Nigeria        38 non-null float64
Deaths_Senegal        22 non-null float64
Deaths_UnitedStates  18 non-null float64
Deaths_Spain          16 non-null float64
Deaths_Mali           12 non-null float64
date_dt               122 non-null datetime64[ns]
year                  122 non-null int64
month                 122 non-null int64
day                   122 non-null int64
outbreak_d            122 non-null timedelta64[ns]
dtypes: datetime64[ns](2), float64(16), int64(4), timedelta64[ns](1)
memory usage: 22.0 KB
None
```

We get `timedeltas` when we perform calculations with `datetime` objects.

11.7 datetime methods

Here's a dataset on bank failures.

```
banks = pd.read_csv('../data/banklist.csv')
print(banks.head())

          Bank Name \
0      Fayette County Bank
1  Guaranty Bank, (d/b/a BestBank in Georgia & Mi...
2                  First NBC Bank
3                  Proficio Bank
4      Seaway Bank and Trust Company

      City      ST   CERT \
0    Saint Elmo    IL  1802
1  Milwaukee    WI 30003
2   New Orleans   LA 58302
3  Cottonwood Hts  UT 35495
4      Chicago    IL 19328

      Acquiring Institution Closing Date Updated Date
0  United Fidelity Bank, fsb  26-May-17  26-Jul-17
1  First-Citizens Bank & Trust Company  5-May-17  26-Jul-17
2           Whitney Bank  28-Apr-17  26-Jul-17
3  Cache Valley Bank  3-Mar-17  18-May-17
4  State Bank of Texas  27-Jan-17  18-May-17
```

Again, we can import our data with the dates directly parsed.

```
banks = pd.read_csv('../data/banklist.csv', parse_dates=[5, 6])
print(banks.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 553 entries, 0 to 552
Data columns (total 7 columns):
Bank Name      553 non-null object
City          553 non-null object
ST            553 non-null object
CERT          553 non-null int64
Acquiring Institution 553 non-null object
Closing Date    553 non-null datetime64[ns]
Updated Date    553 non-null datetime64[ns]
dtypes: datetime64[ns](2), int64(1), object(4)
memory usage: 30.3+ KB
None
```

We can parse out the date by getting the quarter and year the bank closed

```
banks['closing_quarter'], banks['closing_year'] = \
(banks['Closing Date'].dt.quarter,
 banks['Closing Date'].dt.year)
```

We can calculate how many banks closed in each year.

```
closing_year = banks.groupby(['closing_year']).size()
```

Or we can then calculate how many banks close in each quarter of each year.

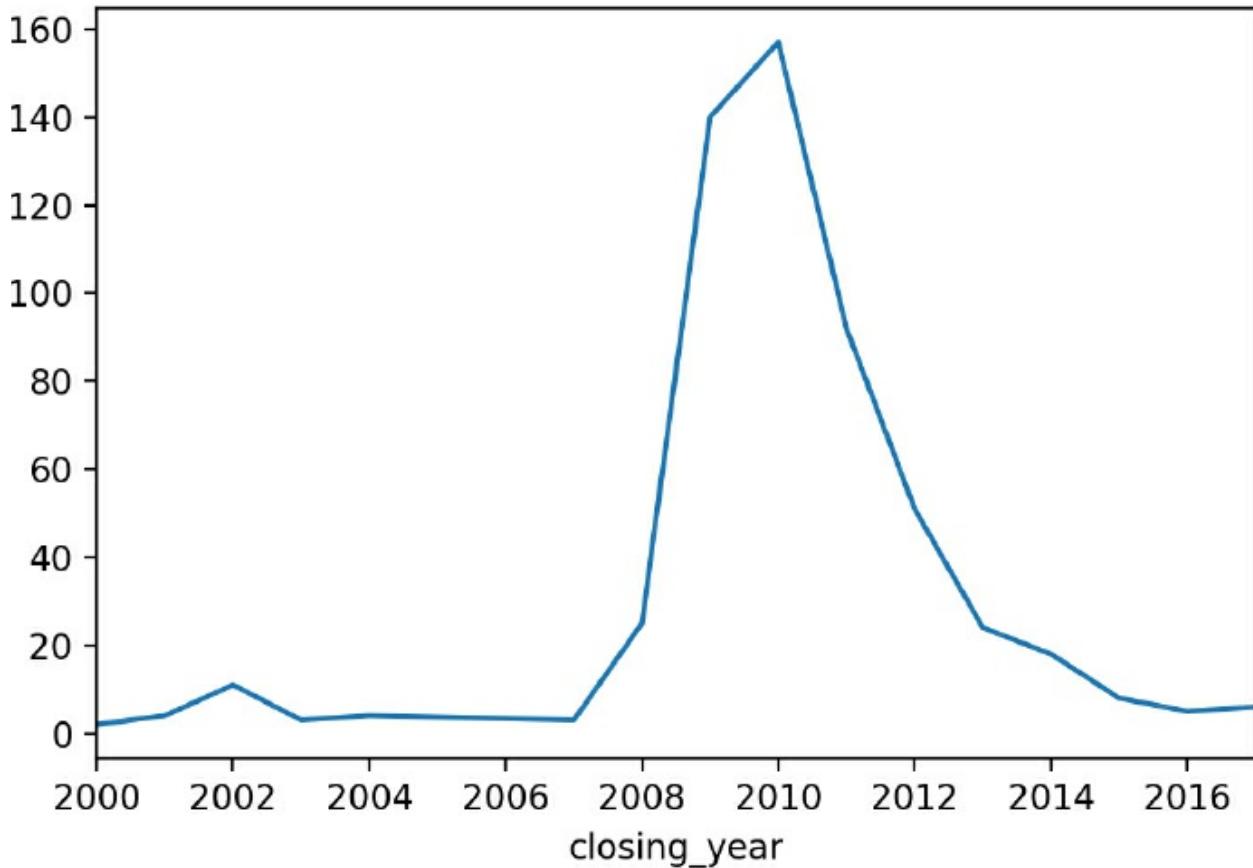
```
closing_year_q = banks.groupby(['closing_year', 'closing_quarter']).size()
```

We can then plot these results as shown in [Figure 11-1](#) and [11-2](#).

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

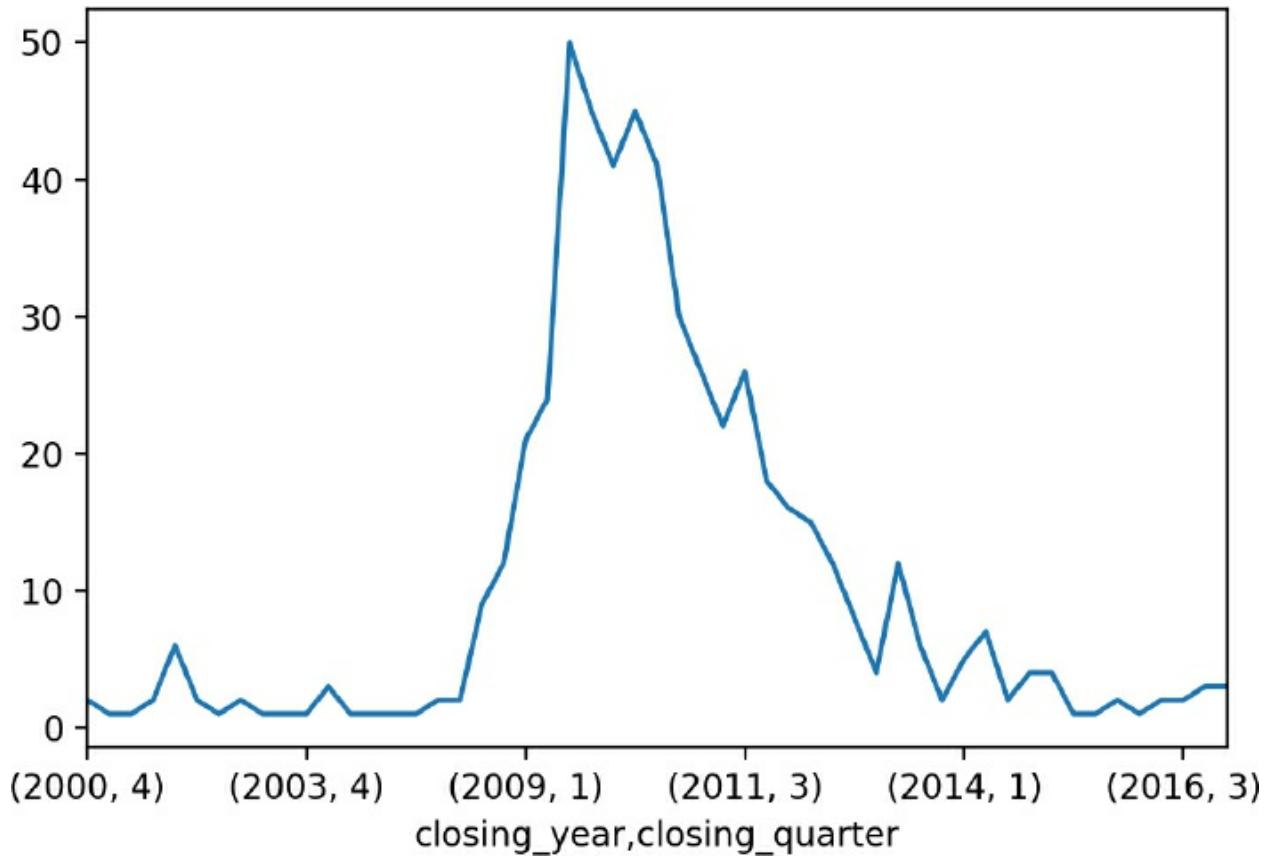
```
fig, ax = plt.subplots()  
ax = closing_year.plot()  
plt.show()
```

Figure 11-1: Number of banks closing each year



```
fig, ax = plt.subplots()  
ax = closing_year_q.plot()  
plt.show()
```

Figure 11-2: Number of banks closing each year by quarter



11.8 Getting stock data

One common type of data that contains dates are stock prices. Luckily python has a way of getting this type of data programmatically.

```
# we can install and use the pandas_datareader
# to get data from the internet
import pandas_datareader as pdr

# in this example we are getting stock information about Tesla
tesla = pdr.get_data_yahoo('TSLA')

# the stock data was saved
# so we do not need to rely on the internet again
# so we can load the same dataset as a file
tesla = pd.read_csv('../data/tesla_stock_yahoo.csv', parse_dates=[0])
```

Here's what the stock data looks like.

```
print(tesla.head())

      Date      Open      High       Low     Close   Adj Close \\
0  2010-06-29  19.000000    25.00  17.540001  23.889999  23.889999
1  2010-06-30  25.790001    30.42  23.299999  23.830000  23.830000
2  2010-07-01  25.000000    25.92  20.270000  21.959999  21.959999
3  2010-07-02  23.000000    23.10  18.709999  19.200001  19.200001
4  2010-07-06  20.000000    20.00  15.830000  16.110001  16.110001

      Volume
0  18766300
1  17187100
2  8218800
3  5139800
4  6866900
```

```

print(tesla.tail())

```

	Date	Open	High	Low	Close	\
1786	2017-08-02	318.940002	327.119995	311.220001	325.890015	
1787	2017-08-03	345.329987	350.000000	343.149994	347.089996	
1788	2017-08-04	347.000000	357.269989	343.299988	356.910004	
1789	2017-08-07	357.350006	359.480011	352.750000	355.170013	
1790	2017-08-08	357.529999	368.579987	357.399994	365.220001	

	Adj Close	Volume
1786	325.890015	13091500
1787	347.089996	13535000
1788	356.910004	9198400
1789	355.170013	6276900
1790	365.220001	7449837

11.9 Subsetting data based on dates

Since we now know how to extract parts of a date out of a column ([Section 11.5](#)), we can incorporate these methods to subset our data without having to parse out the individual components manually.

For example if we were only looking for data from June 2010 we can use boolean sub-setting like it was shown in Chapters 1 and 2

```

print(tesla.loc[(tesla.Date.dt.year == 2010) & \
                (tesla.Date.dt.month == 6)])

```

	Date	Open	High	Low	Close	Adj Close	\
0	2010-06-29	19.000000	25.00	17.540001	23.889999	23.889999	
1	2010-06-30	25.790001	30.42	23.299999	23.830000	23.830000	

	Volume
0	18766300
1	17187100

11.9.1 The DatetimeIndex

What ends up being fairly common when working with datetime data is setting the datetime object to be the dataframe's index. Thus far we've mainly left the dataframe row index to be the row number. And we've seen some side-effects where the row index may not always be the row number such as concatenating dataframes in [Chapter 4](#).

First let's assign the `Date` column as the index.

```

tesla.index = tesla['Date']
print(tesla.index)

DatetimeIndex(['2010-06-29', '2010-06-30', '2010-07-01',
               '2010-07-02', '2010-07-06', '2010-07-07',
               '2010-07-08', '2010-07-09', '2010-07-12',
               '2010-07-13',
               ...
               '2017-07-26', '2017-07-27', '2017-07-28',
               '2017-07-31', '2017-08-01', '2017-08-02',
               '2017-08-03', '2017-08-04', '2017-08-07',
               '2017-08-08'],
              dtype='datetime64[ns]', name='Date', length=1791,
              freq=None)

```

With the index set as a date object, we can now use the date directly to subset rows. For example, we can subset based on the year

```
print(tesla['2015'].iloc[:5, :5])
```

Date	Date	Open	High	Low	\
2015-01-02	2015-01-02	222.869995	223.250000	213.259995	
2015-01-05	2015-01-05	214.550003	216.500000	207.160004	
2015-01-06	2015-01-06	210.059998	214.199997	204.210007	
2015-01-07	2015-01-07	213.350006	214.779999	209.779999	
2015-01-08	2015-01-08	212.809998	213.800003	210.009995	

Date	Close
2015-01-02	219.309998
2015-01-05	210.089996
2015-01-06	211.279999
2015-01-07	210.949997
2015-01-08	210.619995

Or year and month

```
print(tesla['2010-06'].iloc[:, :5])
```

Date	Date	Open	High	Low	Close
2010-06-29	2010-06-29	19.000000	25.00	17.540001	23.889999
2010-06-30	2010-06-30	25.790001	30.42	23.299999	23.830000

11.9.2 The TimedeltaIndex

Just like how we set the index of a dataframe to a `datetime` to create a `DatetimeIndex`, we can do the same thing with `timedeltas` and create a `TimedeltaIndex`.

Let's create a `timedelta`.

```
tesla['ref_date'] = tesla['Date'] - tesla['Date'].min()
```

Now we can assign the `timedelta` to the index.

```
tesla.index = tesla['ref_date']
```

```
print(tesla.iloc[:5, :5])
```

ref_date	Date	Open	High	Low	Close
0 days	2010-06-29	19.000000	25.00	17.540001	23.889999
1 days	2010-06-30	25.790001	30.42	23.299999	23.830000
2 days	2010-07-01	25.000000	25.92	20.270000	21.959999
3 days	2010-07-02	23.000000	23.10	18.709999	19.200001
7 days	2010-07-06	20.000000	20.00	15.830000	16.110001

We can now select our data based on these deltas.

```
print(tesla['0 day': '5 day'].iloc[:5, :5])
```

ref_date	Date	Open	High	Low	Close
0 days	2010-06-29	19.000000	25.00	17.540001	23.889999
1 days	2010-06-30	25.790001	30.42	23.299999	23.830000
2 days	2010-07-01	25.000000	25.92	20.270000	21.959999
3 days	2010-07-02	23.000000	23.10	18.709999	19.200001

11.10 Date Ranges

Not every dataset will have a fixed frequency of values. For example, in our `ebola` dataset, we do not have an observation for every day in the date range.

```
ebola = pd.read_csv('../data/country_timeseries.csv',
```

```

        parse_dates=[0])
print(ebola.iloc[:5, :5])

      Date    Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
0  2015-01-05   289       2776.0           NaN        10030.0
1  2015-01-04   288       2775.0           NaN         9780.0
2  2015-01-03   287       2769.0       8166.0        9722.0
3  2015-01-02   286           NaN       8157.0           NaN
4  2014-12-31   284       2730.0       8115.0        9633.0

```

Here, **2015-01-01** is missing from the **head** of the data,

```

print(ebola.iloc[-5:, :5])

      Date    Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
117  2014-03-27     5       103.0          8.0            6.0
118  2014-03-26     4        86.0           NaN           NaN
119  2014-03-25     3        86.0           NaN           NaN
120  2014-03-24     2        86.0           NaN           NaN
121  2014-03-22     0        49.0           NaN           NaN

```

and **2014-03-23** is missing from the **tail** of the data.

It's common to create a date range to **reindex** our data. We can create a date range with the **date_range** function⁵.

⁵**date_range** documentation: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.date_range.html

For example we can create a date range for the **head** of our data.

```

head_range = pd.date_range(start='2014-12-31', end='2015-01-05')
print(head_range)

DatetimeIndex(['2014-12-31', '2015-01-01', '2015-01-02',
               '2015-01-03', '2015-01-04', '2015-01-05'],
              dtype='datetime64[ns]', freq='D')

```

We'll just work with the first 5 rows in this example.

```
ebola_5 = ebola.head()
```

If we wanted to set this date range as the index, we need to first set the date as the index.

```
ebola_5.index = ebola_5['Date']
```

Next we can **reindex** our data

```

ebola_5.reindex(head_range)
print(ebola_5.iloc[:, :5])

```

Date	Date	Day	Cases_Guinea	Cases_Liberia	\
2015-01-05	2015-01-05	289	2776.0	NaN	
2015-01-04	2015-01-04	288	2775.0	NaN	
2015-01-03	2015-01-03	287	2769.0	8166.0	
2015-01-02	2015-01-02	286	NaN	8157.0	
2014-12-31	2014-12-31	284	2730.0	8115.0	
Cases_SierraLeone					
2015-01-05			10030.0		
2015-01-04			9780.0		
2015-01-03			9722.0		
2015-01-02			NaN		
2014-12-31			9633.0		

11.10.1 Frequencies

When we created our `head_range`, there is a part in the printed statement called `freq`. In that example, the `freq` is '`d`' for 'day'. That is, the values in our date range are stepping day-by-day. A table of these frequencies is reproduced in [Table 11-2](#).⁶

⁶Frequency offset aliases: <https://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>

These values can be passed into the `freq` parameter when calling `date_range`. For example, January 1, 2017 is a Sunday, and we can create a range of business days of that week.

```
# business days during the week of Jan 1, 2017
print(pd.date_range('2017-01-01', '2017-01-07', freq='B'))  
  
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04',  
               '2017-01-05', '2017-01-06'],  
              dtype='datetime64[ns]', freq='B')
```

11.10.2 Offsets

Offsets are variations on a base frequency. For example, we can take our business days created earlier, and add an offset such that instead of *every* business day, we can have *every other* business day.

```
# every other business day during the week of Jan 1, 2017
print(pd.date_range('2017-01-01', '2017-01-07', freq='2B'))  
  
DatetimeIndex(['2017-01-02', '2017-01-04', '2017-01-06'],  
              dtype='datetime64[ns]', freq='2B')
```

You can see this offset was created by putting some multiplying value before the base frequency. These offsets can be combined with other base frequencies as well. For example, we can specify the first Thursday of each month in the year 2017.

```
print(pd.date_range('2017-01-01', '2017-12-31', freq='WOM-1THU'))  
  
DatetimeIndex(['2017-01-05', '2017-02-02', '2017-03-02',  
               '2017-04-06', '2017-05-04', '2017-06-01',  
               '2017-07-06', '2017-08-03', '2017-09-07',  
               '2017-10-05', '2017-11-02', '2017-12-07'],  
              dtype='datetime64[ns]', freq='WOM-1THU')
```

Table 11–2: Table of frequencies

Alias	Description
B	business day frequency
C	custom business day frequency (experimental)
D	calendar day frequency
W	weekly frequency
M	month end frequency
SM	semi-month end frequency (15th and end of month)
BM	business month end frequency
CBM	custom business month end frequency
MS	month start frequency
SMS	semi-month start frequency (1st and 15th)
BMS	business month start frequency
CBMS	custom business month start frequency
Q	quarter end frequency
BQ	business quarter end frequency
QS	quarter start frequency
BQS	business quarter start frequency
A	year end frequency
BA	business year end frequency
AS	year start frequency
BAS	business year start frequency
BH	business hour frequency
H	hourly frequency
T	min
S	secondly frequency
L	ms
U	us
N	nanoseconds

Or the third Friday of each month.

```
print(pd.date_range('2017-01-01', '2017-12-31', freq='WOM-3FRI'))

DatetimeIndex(['2017-01-20', '2017-02-17', '2017-03-17',
               '2017-04-21', '2017-05-19', '2017-06-16',
               '2017-07-21', '2017-08-18', '2017-09-15',
               '2017-10-20', '2017-11-17', '2017-12-15'],
              dtype='datetime64[ns]', freq='WOM-3FRI')
```

11.11 Shifting Values

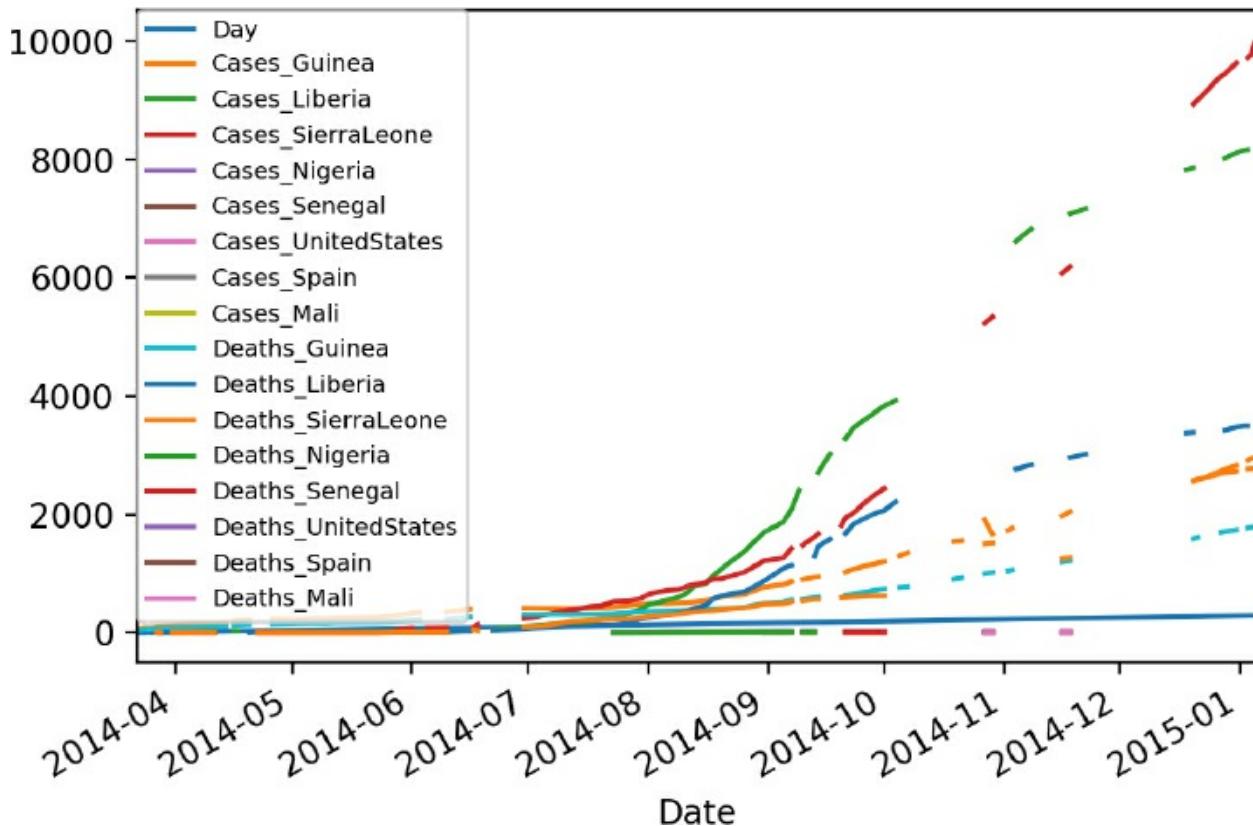
There are a few reasons why you'd want to shift your dates by a certain value. One could be correcting some kind of measurement error in your data. Another could be standardizing the start

dates for your data so you can compare trends.

Even though our `ebola` data isn't "tidy", one of the benefits of the data in its current format is its ability to plot the outbreak. This plot is shown in [Figure 11-3](#).

```
import matplotlib.pyplot as plt  
  
ebola.index = ebola['Date']  
  
fig, ax = plt.subplots()  
ax = ebola.plot(ax=ax)  
ax.legend(fontsize=7,  
          loc=2,  
          borderaxespad=0.)  
plt.show()
```

Figure 11-3: Ebola plot of cases and deaths (unshifted dates)



When we're looking at an outbreak, one useful thing to look at is how fast an outbreak is spreading relative to other countries.

Let's look at just a few columns from our `ebola` dataset.

```
ebola_sub = ebola[['Day', 'Cases_Guinea', 'Cases_Liberia']]  
print(ebola_sub.tail(10))
```

Date	Day	Cases_Guinea	Cases_Liberia
2014-04-04	13	143.0	18.0
2014-04-01	10	127.0	8.0
2014-03-31	9	122.0	8.0
2014-03-29	7	112.0	7.0
2014-03-28	6	112.0	3.0
2014-03-27	5	103.0	8.0
2014-03-26	4	86.0	NaN
2014-03-25	3	86.0	NaN

2014-03-24	2	86.0	Nan
2014-03-22	0	49.0	Nan

You can see that each country's starting date is different, this makes comparing the actual slopes between countries difficult when a new outbreak occurs later in time.

In this example, we want all our dates to start from a common 0 day. There are multiple steps to this process.

1. Since not every date is listed, we need to create a date range of all the dates in our data
2. We need to calculate the difference between the earliest date in our dataset, and the earliest valid (non `NaN`) date in each column
3. We can then shift each of the columns by this calculated value

Before we begin, let's start off with a fresh copy of our `ebola` dataset. We'll parse the `Date` column as a proper `date` object, and assign this date to the `index`. In this example, we are parsing the date and setting it as the index directly.

```
ebola = pd.read_csv('../data/country_timeseries.csv',
                    index_col='Date',
                    parse_dates=['Date'])
print(ebola.head().iloc[:, :4])

      Date      Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
2015-01-05    289     2776.0          NaN        10030.0
2015-01-04    288     2775.0          NaN         9780.0
2015-01-03    287     2769.0       8166.0        9722.0
2015-01-02    286          NaN       8157.0          NaN
2014-12-31    284     2730.0       8115.0        9633.0

print(ebola.tail().iloc[:, :4])

      Date      Day  Cases_Guinea  Cases_Liberia  Cases_SierraLeone
2014-03-27      5      103.0          8.0          6.0
2014-03-26      4       86.0          NaN         NaN
2014-03-25      3       86.0          NaN         NaN
2014-03-24      2       86.0          NaN         NaN
2014-03-22      0       49.0          NaN         NaN
```

First, we need to create the date range to fill in all the missing dates in our data. This way when we shift down our date values, the number of days the data will shift will be the same as the number of rows that will be shifted.

```
new_idx = pd.date_range(ebola.index.min(), ebola.index.max())
```

If we look at our `new_idx`, you'll see that the dates are in the wrong order than what we want

```
print(new_idx)

DatetimeIndex(['2014-03-22', '2014-03-23', '2014-03-24',
               '2014-03-25', '2014-03-26', '2014-03-27',
               '2014-03-28', '2014-03-29', '2014-03-30',
               '2014-03-31',
               ...
               '2014-12-27', '2014-12-28', '2014-12-29',
               '2014-12-30', '2014-12-31', '2015-01-01',
               '2015-01-02', '2015-01-03', '2015-01-04',
               '2015-01-05'],
              dtype='datetime64[ns]', length=290, freq='D')
```

To fix this, we can reverse the order of the index.

```
new_idx = reversed(new_idx)
```

Now we can properly `reindex` our data (Chapter [5.3.4](#)). This will create rows of `NaN` values if the index does not exist already in our dataset.

```
ebola = ebola.reindex(new_idx)
```

Now if we look at the `head` and `tail` of our data, dates that were originally not listed, have been added into our dataset along with a row of `NaN` missing values. Additionally, the `Date` column is filled with the `NaT` value, which is the `NaN` missing value equivalent for missing times.

```
print(ebola.head().iloc[:, :4])
```

	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
Date				
2015-01-05	289.0	2776.0	NaN	10030.0
2015-01-04	288.0	2775.0	NaN	9780.0
2015-01-03	287.0	2769.0	8166.0	9722.0
2015-01-02	286.0	NaN	8157.0	NaN
2015-01-01	NaN	NaN	NaN	NaN

```
print(ebola.tail().iloc[:, :4])
```

	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone
Date				
2014-03-26	4.0	86.0	NaN	NaN
2014-03-25	3.0	86.0	NaN	NaN
2014-03-24	2.0	86.0	NaN	NaN
2014-03-23	NaN	NaN	NaN	NaN
2014-03-22	0.0	49.0	NaN	NaN

Now that we've created our date range and assigned it to the `index`. Our next step is calculating the difference between the earliest date in our dataset, and the earliest valid (non-missing) date in each column.

We can use the `Series` method, `last_valid_index` ⁷, that returns the label (`index`) of the last non-missing or non-null value. There is an analogous method called `first_valid_index` ⁸ that returns the first non-missing or non-null value. Since we want to perform this calculation across all the columns, we can `apply` it.

⁷`last_valid_index` documentation https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.last_valid_index.html#pandas.Series.last_valid_index

⁸`first_valid_index` documentation https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.first_valid_index.html#pandas.Series.first_valid_index

```
last_valid = ebola.apply(pd.Series.last_valid_index)
print(last_valid)
```

Day	2014-03-22
Cases_Guinea	2014-03-22
Cases_Liberia	2014-03-27
Cases_SierraLeone	2014-03-27
Cases_Nigeria	2014-07-23
Cases_Senegal	2014-08-31
Cases_UnitedStates	2014-10-01
Cases_Spain	2014-10-08
Cases_Mali	2014-10-22
Deaths_Guinea	2014-03-22
Deaths_Liberia	2014-03-27
Deaths_SierraLeone	2014-03-27
Deaths_Nigeria	2014-07-23
Deaths_Senegal	2014-09-07

```

Deaths_UnitedStates      2014-10-01
Deaths_Spain             2014-10-08
Deaths_Mali              2014-10-22
dtype: datetime64[ns]

```

Next we want to get the earliest date in our dataset.

```

earliest_date = ebola.index.min()
print(earliest_date)

2014-03-22 00:00:00

```

We can use this date and subtract it from each of our `last_valid` dates.

```

shift_values = last_valid - earliest_date
print(shift_values)

Day          0    days
Cases_Guinea 0    days
Cases_Liberia 5    days
Cases_SierraLeone 5    days
Cases_Nigeria 123   days
Cases_Senegal 162   days
Cases_UnitedStates 193   days
Cases_Spain 200   days
Cases_Mali 214   days
Deaths_Guinea 0    days
Deaths_Liberia 5    days
Deaths_SierraLeone 5    days
Deaths_Nigeria 123   days
Deaths_Senegal 169   days
Deaths_UnitedStates 193   days
Deaths_Spain 200   days
Deaths_Mali 214   days
dtype: timedelta64[ns]

```

Finally, we can iterate through each column and use the `shift` method on it to shift our columns down by the corresponding value in `shift_values`. Note that the values in `shift_values` are all positive, if they were negative (if we flipped the order of our subtraction), it would shift the values up.

```

ebola_dict = {}
for idx, col in enumerate(ebola):
    d = shift_values[idx].days
    shifted = ebola[col].shift(d)
    ebola_dict[col] = shifted

```

Since we have a `dict` of values, we can convert it to a dataframe using the pandas `DataFrame` function.

```
ebola_shift = pd.DataFrame(ebola_dict)
```

Since `dict` objects are unordered, we can pass in the original `ebola` columns to reorder the values back.

```
ebola_shift = ebola_shift[ebola.columns]
```

Now you can see the last row in each column has a value, That is the columns have been shifted down accordingly.

```
print(ebola_shift.tail())
```

Date	Day	Cases_Guinea	Cases_Liberia	Cases_SierraLeone	\
2014-03-26	4.0	86.0	8.0	2.0	
2014-03-25	3.0	86.0	NaN	NaN	

```

2014-03-24    2.0      86.0      7.0      NaN
2014-03-23    NaN       NaN      3.0      2.0
2014-03-22    0.0      49.0      8.0      6.0
                                         Cases_Nigeria   Cases_Senegal   Cases_UnitedStates \
Date
2014-03-26    1.0       NaN       1.0
2014-03-25    NaN       NaN       NaN
2014-03-24    NaN       NaN       NaN
2014-03-23    NaN       NaN       NaN
2014-03-22    0.0       1.0       1.0
                                         Cases_Spain     Cases_Mali    Deaths_Guinea   Deaths_Liberia \
\
Date
2014-03-26    1.0       NaN      62.0      4.0
2014-03-25    NaN       NaN      60.0      NaN
2014-03-24    NaN       NaN      59.0      2.0
2014-03-23    NaN       NaN      NaN       3.0
2014-03-22    1.0       1.0      29.0      6.0
                                         Deaths_SierraLeone  Deaths_Nigeria  Deaths_Senegal \
Date
2014-03-26    2.0       1.0       NaN
2014-03-25    NaN       NaN       NaN
2014-03-24    NaN       NaN       NaN
2014-03-23    2.0       NaN       NaN
2014-03-22    5.0       0.0       0.0
                                         Deaths_UnitedStates  Deaths_Spain  Deaths_Mali
Date
2014-03-26    0.0       1.0       NaN
2014-03-25    NaN       NaN       NaN
2014-03-24    NaN       NaN       NaN
2014-03-23    NaN       NaN       NaN
2014-03-22    0.0       1.0       1.0

```

Finally, since the index that are no longer valid across each row, we can remove them, and assign the correct index, which is the `Day`. Now `Day` no longer represents the first day of the entire outbreak, it represents the day of an outbreak for a given country.

```

ebola_shift.index = ebola_shift['Day']
ebola_shift = ebola_shift.drop(['Day'], axis=1)

print(ebola_shift.tail())
                                         Cases_Guinea  Cases_Liberia  Cases_SierraLeone  Cases_Nigeria \
\
Day
4.0      86.0      8.0      2.0      1.0
3.0      86.0      NaN      NaN      NaN
2.0      86.0      7.0      NaN      NaN
NaN      NaN      3.0      2.0      NaN
0.0      49.0      8.0      6.0      0.0
                                         Cases_Senegal  Cases_UnitedStates  Cases_Spain  Cases_Mali \
\
Day
4.0      NaN          1.0      1.0      NaN
3.0      NaN          NaN      NaN      NaN
2.0      NaN          NaN      NaN      NaN
NaN      NaN          NaN      NaN      NaN
0.0      1.0          1.0      1.0      1.0
                                         Deaths_Guinea  Deaths_Liberia  Deaths_SierraLeone \
\
Day
4.0      62.0          4.0      2.0
3.0      60.0          NaN      NaN
2.0      59.0          2.0      NaN
NaN      NaN          3.0      2.0
0.0      29.0          6.0      5.0
                                         Deaths_Nigeria  Deaths_Senegal  Deaths_UnitedStates \
\
Day
4.0      1.0          NaN          0.0

```

3.0	NaN	NaN	NaN
2.0	NaN	NaN	NaN
NaN	NaN	NaN	NaN
0.0	0.0	0.0	0.0
	Deaths_Spain	Deaths_Mali	
Day			
4.0	1.0	NaN	
3.0	NaN	NaN	
2.0	NaN	NaN	
NaN	NaN	NaN	
0.0	1.0	1.0	

11.12 Resampling

Resampling converts a datetime from one frequency to another frequency. There are 3 types of resampling that can occur,

1. downsampling: higher frequency to a lower frequency (e.g., daily to monthly)
2. upsampling: lower frequency to a higher frequency (e.g., monthly to daily)
3. no change: frequency does not change (e.g., every first Thursday of the month to the last Friday of the month)

The values we can pass into `resample` are listed in [Table 11-2](#)

```
# downsample daily values to monthly values
# since we have multiple values we need to aggregate the results
# here we will use the mean
down = ebola.resample('M').mean()
print(down.iloc[:5, :5])

          Day      Cases_Guinea      Cases_Liberia   \
Date
2014-03-31    4.500000    94.500000    6.500000
2014-04-30   24.333333  177.818182  24.555556
2014-05-31   51.888889  248.777778  12.555556
2014-06-30   84.636364  373.428571  35.500000
2014-07-31  115.700000  423.000000  212.300000

          Cases_SierraLeone      Cases_Nigeria
Date
2014-03-31        3.333333            NaN
2014-04-30        2.200000            NaN
2014-05-31        7.333333            NaN
2014-06-30     125.571429            NaN
2014-07-31     420.500000     1.333333

# here we will upsample our downsampled value
# notice how we get missing dates populated,
# but they are filled in with missing values
up = down.resample('D').mean()
print(up.iloc[:5, :5])

          Day      Cases_Guinea      Cases_Liberia      Cases_SierraLeone   \
Date
2014-03-31      4.5           94.5           6.5           3.333333
2014-04-01      NaN           NaN           NaN           NaN
2014-04-02      NaN           NaN           NaN           NaN
2014-04-03      NaN           NaN           NaN           NaN
2014-04-04      NaN           NaN           NaN           NaN

          Cases_Nigeria
Date
2014-03-31            NaN
2014-04-01            NaN
2014-04-02            NaN
2014-04-03            NaN
```

11.13 Timezones

Don't try to write your own timezone converter. As Tom Scott explains in a "Computerphile" video: "that way lies madness"⁹. There are many things you probably did not even think to consider when working with different time zones. For example, not every country follows daylight savings, and those that do, do not change the clocks on the same day of the year. And sometimes, countries that usually observe daylight savings. Don't forget about leap years and **leap seconds!**. Luckily Python has a library specifically designed to work with timezones, `pytz`¹⁰. Pandas also wraps this library when working with time zones.

⁹The problem with time and timezones - Computerphile: <https://www.youtube.com/watch?v=-5wpm-gesOY>

¹⁰`pytz` documentation: <http://pytz.sourceforge.net/>

```
import pytz
```

There are many time zones available in the library.

```
print(len(pytz.all_timezones))  
593
```

Here are the US time zones

```
import re  
regex = re.compile(r'^US')  
selected_files = filter(regex.search, pytz.common_timezones)  
print(list(selected_files))  
['US/Alaska', 'US/Arizona', 'US/Central', 'US/Eastern', 'US/Hawaii',  
'US/Mountain', 'US/Pacific']
```

The easiest way to interact with timezones in pandas is to use the string names given in `pytz.all_timezones`.

One way to illustrate time zones is to create 2 time stamps using the pandas `Timestamp` function. At the time of writing, there is a flight from JFK to LAX that departs at 7:00 AM and lands at 9:57 AM. We can encode these times with the proper timezone.

```
# 7AM Eastern  
depart = pd.Timestamp('2017-08-29 07:00', tz='US/Eastern')  
print(depart)  
2017-08-29 07:00:00-04:00
```

Another way we can encode a time zone is using the `tz_localize` method on an "empty" timestamp.

```
arrive = pd.Timestamp('2017-08-29 09:57')  
print(arrive)  
2017-08-29 09:57:00  
arrive = arrive.tz_localize('US/Pacific')  
print(arrive)
```

```
2017-08-29 09:57:00-07:00
```

We can convert the arrival time back to eastern to see what the time would be on the east coast when the flight arrives.

```
print(arrive.tz_convert('US/Eastern'))
```

```
2017-08-29 12:57:00-04:00
```

We can also perform operations on timezones. Here we can look at the difference between the times to get the flight duration

```
# will cause an error
duration = arrive - depart

Traceback (most recent call last):
  File "<ipython-input-1-0db03cba0b30>", line 2, in <module>
    duration = arrive - depart
TypeError: Timestamp subtraction must have the same timezones or no
timezones
```

You can see the `TypeError` saying that we must have the same time zones (or no timezones) in order to perform these calculations

```
# get the flight duration
duration = arrive.tz_convert('US/Eastern') - depart
print(duration)

0 days 05:57:00
```

11.14 Conclusion

Pandas provides a series of convenient methods and functions when working with dates and time because they are constantly used when working with timeseries data. A common example of timeseries data is stock prices, but can also be observational data, or simulated data. These convenient functions and methods allow you to quickly work with date objects without having to resort to string manipulation and parsing.

Part IV: Data Modeling

[Chapter 12](#), “[Linear Models](#),” Linear regression.

[Chapter 13](#), “[Generalized Linear Models](#),” Generalized Linear Models and Survival Analysis.

[Chapter 14](#), “[Model Diagnostics](#),” Comparing Models.

[Chapter 15](#), “[Regularization](#),” Regularization.

[Chapter 16](#), “[Clustering](#),” Clustering.

Chapter 12. Linear Models

12.1 Introduction

This part of the book will follow the methods described in Jared Lander’s “R for Everyone” book. The rationale is that since you have learned the methods of data manipulation in Python using pandas, you can save out the cleaned dataset if you need to use a method from another analytics language. Also, this part covers many of the basic modeling techniques that serves as an introduction to data analytics/machine learning. Another great reference is Andreas Muller’s and Sarah Guido’s book “Introduction to Machine Learning with Python”.

12.2 Simple Linear Regression

The goal of linear regression is to draw a straight line relationship between a response variable (aka outcome or dependent variable) and a predictor variable (aka feature, covariate, or independent variable).

Let’s take a look at our `tips` dataset,

```
import pandas as pd
import seaborn as sns

tips = sns.load_dataset('tips')
print(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
```

In our simple linear regression, we’d like to see how the `total_bill` relates to or predicts the `tip`.

12.2.1 Using statsmodels

We can use the `statsmodels` library to perform our simple linear regression. We will use the formula api from `statsmodels`.

```
import statsmodels.formula.api as smf
```

To perform our simple linear regression we use the `ols` function, that performs ordinary least squares, which is one method to estimate parameters in a linear regression. Recall, that the formula for a line is $y = mx + b$, where y is our response variable, x is our predictor, b is the intercept, and m , the slope, is the parameter we are estimating.

The formula notation has 2 parts, separated by a tilde, \sim , the left of the tilde, is the response variable, and the tilde is the predictor.

```
model = smf.ols(formula='tip ~ total_bill', data=tips)
```

Once we specified our model, we can fit the data to the model by using the `fit` method.

```
results = model.fit()
```

To look at our results, we can call the `summary` method on the `results`

```
print(results.summary())
OLS Regression Results
=====
Dep. Variable:          tip      R-squared:     0.457
Model:                 OLS      Adj. R-squared:  0.454
Method:                Least Squares   F-statistic:   203.4
Date:        Tue, 12 Sep 2017   Prob (F-statistic):  6.69e-34
Time:             06:25:09      Log-Likelihood: -350.54
No. Observations:    244      AIC:            705.1
Df Residuals:        242      BIC:            712.1
Df Model:             1      Covariance Type: nonrobust
=====
              coef    std err       t   P>|t|      [0.025  0.975]
-----
Intercept    0.9203    0.160    5.761    0.000    0.606    1.235
total_bill   0.1050    0.007   14.260    0.000    0.091    0.120
=====
Omnibus:           20.185 Durbin-Watson:   2.151
Prob(Omnibus):    0.000   Jarque-Bera (JB): 37.750
Skew:             0.443   Prob(JB):      6.35e-09
Kurtosis:          4.711   Cond. No.      53.0
=====
Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

Here we can see the `Intercept` of the model and the `total_bill`. We can use these parameters in our formula for a line, $y = (0.105)x + 0.920$. To interpret these numbers, we can say, for every 1 unit increase in `total_bill` (i.e., every time the bill increases by a dollar) the tip will increase by 0.105 (i.e., 10.5 cents).

If we just wanted the coefficients, we can call the `params` attribute on the results

```
print(results.params)
Intercept    0.920270
total_bill   0.105025
dtype: float64
```

Depending on your field, you will also need to report a confidence interval, these are the possible values the estimated value can take on. The confidence interval can be seen from the values under `[0.025 0.975]`. We can also extract these values using the `conf_int` method,

```
print(results.conf_int())
          0            1
Intercept  0.605622  1.234918
total_bill  0.090517  0.119532
```

12.2.2 Using sklearn

Scikit learn is another library used to fit various machine learning models. To perform the same analysis as above, we need to import the `linear_model` module from the library.

```
from sklearn import linear_model
```

From there we can create our linear regression object

```
# create our LinearRegression object
lr = linear_model.LinearRegression()
```

Just like how we fit our model before, we now need to specify the predictors, `x`, and response, `y`. To do this we pass in the columns we want to use for the model

```
# note it is a upper case X
# and a lower case y
# this will fail because our X only has 1 variable
predicted = lr.fit(X=tips['total_bill'], y=tips['tip'])

Traceback (most recent call last):
  File "<ipython-input-1-40e6128e301f>", line 2, in <module>
    predicted = lr.fit(X=tips['total_bill'], y=tips['tip'])
ValueError: Expected 2D array, got 1D array instead:
array=[ 16.99 10.34 21.01 23.68 24.59 25.29          8.77 26.88 15.04
14.78
 10.27 35.26 15.42 18.43 14.83 21.58 10.33 16.29 16.97 20.65
 17.92 20.29 15.77 39.42 19.82 17.81 13.37 12.69 21.7          19.65
  9.55 18.35 15.06 20.69 17.78 24.06 16.31 16.93 18.69 31.27
 16.04 17.46 13.94      9.68 30.4      18.29 22.23 32.4      28.55 18.04
 12.54 10.29 34.81      9.94 25.56 19.49 38.01 26.41 11.24 48.27
 20.29 13.81 11.02 18.29 17.59 20.08 16.45      3.07 20.23 15.01
 12.02 17.07 26.86 25.28 14.73 10.51 17.92 27.2          22.76 17.29
 19.44 16.66 10.07 32.68 15.98 34.83 13.03 18.28 24.71 21.16
 28.97 22.49      5.75 16.32 22.75 40.17 27.28 12.03 21.01 12.46
 11.35 15.38 44.3      22.42 20.92 15.36 20.49 25.21 18.24 14.31
14.
  7.25 38.07 23.95 25.71 17.31 29.93 10.65 12.43 24.08 11.69
 13.42 14.26 15.95 12.48 29.8      8.52 14.52 11.38 22.82 19.08
 20.27 11.17 12.26 18.26      8.51 10.33 14.15 16.          13.16 17.47
  34.3   41.19 27.05 16.43      8.35 18.64 11.87      9.78   7.51 14.07
 13.13 17.26 24.55 19.77 29.85 48.17 25.          13.39 16.49 21.5
 12.66 16.21 13.81 17.51 24.52 20.76 31.71 10.59 10.63 50.81
 15.81   7.25 31.85 16.82 32.9      17.89 14.48      9.6   34.63 34.65
 23.33 45.35 23.17 40.55 20.69 20.9      30.46 18.15 23.1          15.69
 19.81 28.44 15.48 16.58      7.56 10.34 43.11 13.          13.51 18.71
 12.74 13.          16.4   20.53 16.47 26.59 38.73 24.27 12.76 30.06
 25.89 48.33 13.27 28.17 12.9      28.15 11.59      7.74 30.14 12.16
 13.42   8.58 15.98 13.42 16.27 10.09 20.45 13.28 22.12 24.01
 15.69 11.61 10.77 15.53 10.07 12.6      32.83 35.83 29.03 27.18
 22.67 17.82 18.78].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) if it contains a single sample.
```

Since `sklearn` is built to take numpy arrays, there will be times when you have to do some data manipulations to pass your dataframe into `sklearn`. The Error above essentially tells us the matrix passed is not in the correct shape. We need to `reshape` our inputs, and depending on whether we have a single feature (which we have), or a single sample (we have multiple observations), we have to `reshape(-1, 1)` or `reshape(1, -1)`, respectively.

Calling `reshape` directly on the column will either raise a `DeprecationWarning` (pandas 0.17) or `ValueError` (pandas 0.19) depending on the version of pandas. To properly reshape our data, we use call them on the `values` attribute (else you may get another error or warning). When we call `values` on a pandas dataframe or series, we get the `numpy ndarray` representation of the data.

```
# note it is a upper case X
# and a lower case y
# we fix it by putting it in the correct shape for sklearn
predicted = lr.fit(X=tips['total_bill'].values.reshape(-1, 1),
                    y=tips['tip'])
```

Since `sklearn` works on `numpy ndarray`s, you may see code that explicitly passes in the `numpy` vector into the `x` or `y` parameter as such: `y=tips[' tip ']. values`.

`sklearn` doesn't give us the nice summary tables that `statsmodels` does. This mainly stems from 2 different school of thought, statistics and computer science/machine learning. For us to get the coefficients in `sklearn`, we call the `coef_` attribute on the fitted model.

```
print(predicted.coef_)
[ 0.10502452]
```

To get the intercept we call the `intercept`, attribute

```
predicted.intercept_
0.92026961355467307
```

You see here we get the same results as `statsmodels`, people in our data are tipping about 10%.

12.3 Multiple Regression

Simple linear regression has 1 predictor regressed on a continuous response variable. However, we can use multiple regression to put multiple predictors in a model.

12.3.1 Using `statsmodels`

To fit a multiple regression model is very similar to the simple linear regression model. Using the formula interface, we “add” the other covariates to the right-hand side.

```
model = smf.ols(formula='tip ~ total_bill + size', data=tips).\
    fit()
```

```
print(model.summary())
```

OLS Regression Results						
Dep. Variable:	tip	R-squared:	0.468			
Model:	OLS	Adj. R-squared:	0.463			
Method:	Least Squares	F-statistic:	105.9			
Date:	Tue, 12 Sep 2017	Prob (F-statistic):	9.67e-34			
Time:	06:25:10	Log-Likelihood:	-347.99			
No. Observations:	244	AIC:	702.0			
Df Residuals:	241	BIC:	712.5			
Df Model:	2	Covariance Type:	nonrobust			
	coef	std err	t	P> t	[0.025	0.975]
Intercept	0.6689	0.194	3.455	0.001	0.288	1.050
total_bill	0.0927	0.009	10.172	0.000	0.075	0.111
size	0.1926	0.085	2.258	0.025	0.025	0.361
Omnibus:	24.753	Durbin-Watson:	2.100			
Prob(Omnibus):	0.000	Jarque-Bera (JB):	46.169			
Skew:	0.545	Prob(JB):	9.43e-11			
Kurtosis:	4.831	Cond. No.	67.6			

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

The interpretations are exactly the same as before, however, each parameter is interpreted “with all other variables held constant”. That is, for every one unit increase (dollar) in `totalbill`, then the tip increases by `0.09` (i.e., 9 cents) given the size of the group does not change.

12.3.2 Using `statsmodels` [[h4]]with categorical variables

So far we’ve only been using continuous predictors in our model. If we look at the `info` of our data, we will notice that we have categorical variables in our data.

```

print(tips.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
total_bill    244 non-null float64
tip          244 non-null float64
sex          244 non-null category
smoker        244 non-null category
day           244 non-null category
time          244 non-null category
size          244 non-null int64
dtypes: category(4), float64(2), int64(1)
memory usage: 7.2 KB
None

```

When we want to model categorical variable, we have to create dummy variables. That is, each unique value in the category becomes a new binary feature. For example, `sex` in our data can hold one of 2 values, `Female` or `Male`.

```

print(tips.sex.unique())

[Female, Male]
Categories (2, object): [Female, Male]

```

`statsmodels` will automatically create dummy variables for us. To avoid multicollinearity, we typically drop one of the dummy variables. That is, if we have a column that determines whether or not an individual is female, then we know if the person is not female (in our data) then the person must be male. So we can effectively drop the dummy variable that codes for males and still have the same information.

Here's the model that uses all the variables in our data.

```

model = smf.ols(
    formula='tip ~ total_bill + size + sex + smoker + day + time',
    data=tips).\
    fit()

```

We can see from the summary that `statsmodels` automatically creates dummy variables as well as drops the reference variable to avoid multicollinearity.

```

print(model.summary())
                OLS Regression Results
=====
Dep. Variable:                  tip      R-squared:                 0.470
Model:                          OLS      Adj. R-squared:            0.452
Method: Least Squares          F-statistic:              26.06
Date: Tue, 12 Sep 2017          Prob (F-statistic):       1.20e-28
Time: 06:25:10                  Log-Likelihood:           -347.48
No. Observations:               244      AIC:                      713.0
Df Residuals:                  235      BIC:                      744.4
Df Model:                       8      Covariance Type:            nonrobust
=====
            coef    std err          t      P>|t|      [ 0.025     0.975 ]
-----
Intercept      0.5908      0.256      2.310      0.022      0.087      1.095
sex[T.Female]   0.0324      0.142      0.229      0.819     -0.247      0.311
smoker[T.No]    0.0864      0.147      0.589      0.556     -0.202      0.375
day[T.Fri]      0.1623      0.393      0.412      0.680     -0.613      0.937
day[T.Sat]      0.0408      0.471      0.087      0.931     -0.886      0.968
day[T.Sun]      0.1368      0.472      0.290      0.772     -0.793      1.066
time[T.Dinner]  -0.0681     0.445     -0.153      0.878     -0.944      0.808
total_bill      0.0945      0.010      9.841      0.000      0.076      0.113
size             0.1760      0.090      1.966      0.051     -0.000      0.352
=====
Omnibus:                   27.860      Durbin-Watson:            2.096
Prob(Omnibus):            0.000      Jarque-Bera (JB):         52.555
Skew:                      0.607      Prob(JB):                  3.87e-12
Kurtosis:                  4.923      Cond. No.                   281.

```

```
=====
```

```
Warnings:  
[1] Standard Errors assume that the covariance matrix of the errors is  
correctly specified.
```

The interpretation for these parameters are the same as before. However, when interpreting categorical variables, it's in relation to the reference variable (i.e., the dummy variable that was dropped from the analysis). For example, the coefficient for sex [T.Female] is 0.0324. We interpret this value in relation to the reference value, Male, so we say that when the sex goes from Male to Female, the tip increases by 0.324. For the day variable,

```
print(tips.day.unique())  
  
[Sun, Sat, Thur, Fri]  
Categories (4, object): [Sun, Sat, Thur, Fri]
```

We see that our `summary` is missing `Thur`, so that is the reference variable to use to interpret the coefficients.

12.3.3 Using `sklearn`

Just like before multiple regression in `sklearn` is very similar to the syntax for simple linear regression. To add in more features into the model, we pass in the columns we want to use.

```
lr = linear_model.LinearRegression()  
  
# since we are performing multiple regression  
# we no longer need to reshape our X values  
predicted = lr.fit(X=tips[['total_bill', 'size']],  
                    y=tips['tip'])  
print(predicted.coef_)  
  
[ 0.09271334    0.19259779]
```

We can get the intercept from the model just like before

```
print(predicted.intercept_)  
  
0.668944740813
```

12.3.4 Using `sklearn` with categorical variables

We have to manually create our dummy variables for `sklearn`. Luckily, pandas has a function, `get_dummies` that will do this for us. The function will convert all the categorical variables into dummy variables for us, no need to pass in individual columns one-at-a-time. `sklearn` has a `OneHotEncoder` function that does something similar.¹

¹`sklearn OneHotEncoder` documentation: <http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html>

```
tips_dummy = pd.get_dummies(  
    tips[['total_bill', 'size',  
          'sex', 'smoker', 'day', 'time']])  
print(tips_dummy.head())  
  
      total_bill  size  sex_Male  sex_Female  smoker_Yes  smoker_No  \\\n0        16.99     2        0         1          0           1  
1        10.34     3        1         0          0           1  
2        21.01     3        1         0          0           1  
3        23.68     2        1         0          0           1
```

	24.59	4	0	1	0	1
	day_Thur	day_Fri	day_Sat	day_Sun	time_Lunch	time_Dinner
0	0	0	0	1	0	1
1	0	0	0	1	0	1
2	0	0	0	1	0	1
3	0	0	0	1	0	1
4	0	0	0	1	0	1

To drop the reference variable, we can pass in `drop_first =True`.

```
x_tips_dummy_ref = pd.get_dummies(
    tips[['total_bill', 'size',
          'sex', 'smoker', 'day', 'time']], drop_first=True)
print(x_tips_dummy_ref.head())

      total_bill  size  sex_Female  smoker_No  day_Fri  day_Sat \
0        16.99     2           1           1         0         0
1        10.34     3           0           1         0         0
2        21.01     3           0           1         0         0
3        23.68     2           0           1         0         0
4        24.59     4           1           1         0         0

      day_Sun  time_Dinner
0           1           1
1           1           1
2           1           1
3           1           1
4           1           1
```

We fit the model just like before

```
lr = linear_model.LinearRegression()
predicted = lr.fit(X=x_tips_dummy_ref,
                    y=tips['tip'])

print(predicted.coef_)

[ 0.09448701  0.175992      0.03244094      0.08640832      0.1622592
  0.04080082   0.13677854 -0.0681286 ]
```

and coefficient

```
print(predicted.intercept_)

0.590837425951
```

12.4 Keeping index labels from sklearn

One of the annoying things when trying to interpret a model from `sklearn` is that the coefficients are not labeled. This is because the `numpy ndarray` is unable to store this type of metadata. If we want our output to resemble something from `statsmodels`, we need to manually store the labels and append the coefficients to it.

```
import numpy as np

# create and fit the model
lr = linear_model.LinearRegression()
predicted = lr.fit(X=x_tips_dummy_ref, y=tips['tip'])

# get the intercept along with other coefficients
values = np.append(predicted.intercept_, predicted.coef_)

# get the names of the values
names = np.append('intercept', x_tips_dummy_ref.columns)

# put everything in a labeled dataframe
```

```

results = pd.DataFrame(values, index = names,
    columns=['coef'] # you need the square brackets here
)
print(results)

            coef
intercept      0.590837
total_bill      0.094487
size           0.175992
sex_Female     0.032441
smoker_No       0.086408
day_Fri         0.162259
day_Sat         0.040801
day_Sun         0.136779
time_Dinner    -0.068129

```

12.5 Conclusion

This chapter serves as the introduction to fitting models using `statsmodels` and `sklearn`. The concept of adding features to a model and dummy variables, are constantly used when fitting models. Thus far we only worked with fitting linear models where the response variable is a continuous variable. In later chapters we'll fit models where the response variable is not a continuous variable.

Chapter 13. Generalized Linear Models

13.1 Introduction

Not every response variable will be continuous. So a linear regression will not be the correct model in every circumstance. Some outcomes will contain binary data (e.g., sick, not-sick), or even count data (e.g., how many heads will I get). There is a general class of models call the “generalized linear model” that can account for these types of data but still use a linear combination of predictors.

13.2 Logistic Regression

When you have a binary response variable, a common model used is the logistic regression. Here's some data from the ACS

```
import pandas as pd

acs = pd.read_csv('../data/acs_ny.csv')
print(acs.columns)

Index(['Acres', 'FamilyIncome', 'FamilyType', 'NumBedrooms',
       'NumChildren', 'NumPeople', 'NumRooms', 'NumUnits',
       'NumVehicles', 'NumWorkers', 'OwnRent', 'YearBuilt',
       'HouseCosts', 'ElectricBill', 'FoodStamp', 'HeatingFuel',
       'Insurance', 'Language'],
      dtype='object')

print(acs.head())

   Acres FamilyIncome FamilyType NumBedrooms NumChildren \
0    1-10     150 Married        4         1
1    1-10     180 Female Head      3         2
2    1-10     280 Female Head      4         0
3    1-10     330 Female Head      2         1
4    1-10     330 Male  Head      3         1

   NumPeople NumRooms NumUnits NumVehicles NumWorkers \
0          3         9 Single detached           1          0
1          4         6 Single detached           2          0
2          2         8 Single detached           3          1
3          2         4 Single detached           1          0
4          2         5 Single attached          1          0

   OwnRent YearBuilt HouseCosts ElectricBill FoodStamp \
0 Mortgage 1950-1959     1800        90      No
1 Rented Before 1939      850        90      No
2 Mortgage 2000-2004     2600       260      No
3 Rented 1950-1959     1800       140      No
4 Mortgage Before 1939     860       150      No

   HeatingFuel Insurance Language
0      Gas      2500 English
1      Oil        0 English
2      Oil     6600 Other European
3      Oil        0 English
4      Gas      660 Spanish
```

We first need to create a binary response variable. Here we split the `FamilyIncome` variable into a binary variable.

```
acs['ge150k'] = pd.cut(acs['FamilyIncome'],
[0, 150000, acs['FamilyIncome'].max()],
```

```

        labels=[0, 1])
acs['ge150k_i'] = acs['ge150k'].astype(int)
print(acs['ge150k_i'].value_counts())

0    18294
1    4451
Name: ge150k_i, dtype: int64

```

We can see now we've created a binary (0/1) variable

```

acs.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 22745 entries, 0 to 22744
Data columns (total 20 columns):
Acres            22745 non-null object
FamilyIncome     22745 non-null int64
FamilyType       22745 non-null object
NumBedrooms      22745 non-null int64
NumChildren      22745 non-null int64
NumPeople         22745 non-null int64
NumRooms          22745 non-null int64
NumUnits          22745 non-null object
NumVehicles       22745 non-null int64
NumWorkers        22745 non-null int64
OwnRent           22745 non-null object
YearBuilt          22745 non-null object
HouseCosts        22745 non-null int64
ElectricBill      22745 non-null int64
FoodStamp          22745 non-null object
HeatingFuel        22745 non-null object
Insurance          22745 non-null int64
Language           22745 non-null object
ge150k             22745 non-null category
ge150k_i           22745 non-null int64
dtypes: category(1), int64(11), object(8)
memory usage: 3.3+ MB

```

13.2.1 Using statsmodels

To perform a logistic regression, we can use the `logit` function. The formula syntax is the same from the linear regression in the previous chapter.

```

import statsmodels.formula.api as smf

model = smf.logit('ge150k_i ~ HouseCosts + NumWorkers + \
                   'OwnRent + NumBedrooms + FamilyType',
                   data = acs)
results = model.fit()

Optimization terminated successfully.
    Current function value: 0.391651
    Iterations 7

print(results.summary())

```

Logit Regression Results							
Dep. Variable:	ge150k_i	No. Observations:	22745				
Model:	Logit	Df Residuals:	22737				
Method:	MLE	Df Model:	7				
Date:	Tue, 12 Sep 2017	Pseudo R-squ.:	0.2078				
Time:	04:37:17	Log-Likelihood:	-8908.1				
converged:	True	LL-Null:	-11244.				
		LLR p-value:	0.000				
	coef	std err	z	P> z	[0.025	0.975]	
Intercept	-5.8081	0.120	-48.456	0.000	-6.043	-5.573	
OwnRent[T.Outright]	1.8276	0.208	8.782	0.000	1.420	2.236	
OwnRent[T.Rented]	-0.8763	0.101	-8.647	0.000	-1.075	-0.678	
FamilyType[T.Male Head]	0.2874	0.150	1.913	0.056	-0.007	0.582	
FamilyType[T.Married]	1.3877	0.088	15.781	0.000	1.215	1.560	

HouseCosts	0.0007	1.72e-05	42.453	0.000	0.001	0.001
NumWorkers	0.5873	0.026	22.393	0.000	0.536	0.639
NumBedrooms	0.2365	0.017	13.985	0.000	0.203	0.270

Interpreting results from a logistic regression is not as straightforward as a linear regression. This is because in a logistic regression, as with all glm models, there is a transformation, in the form of a link function, that needs to be undone to interpret.

To interpret our logistic model, we first need to exponentiate our results

```
import numpy as np

odds_ratios = np.exp(results.params)
print(odds_ratios)

Intercept          0.003003
OwnRent[T.OutOfBounds] 6.219147
OwnRent[T.Rented]    0.416310
FamilyType[T.Male Head] 1.332901
FamilyType[T.Married] 4.005636
HouseCosts          1.000731
NumWorkers           1.799117
NumBedrooms          1.266852
dtype: float64
```

The values are then interpreted as odds ratios. You can think of this as how many “times likely” the outcome will be, but that phrasing should only be used as an analogy, as it is not technically correct.

An example interpretation of these numbers would be for every 1 unit increase in `NumBedrooms`, the **odds** of the `FamilyIncome` being greater than 150,000 increases by 1.27 times. A similar interpretation can be done with categorical variables. Remember that categorical variables are interpreted in relation to the reference variable.

The three potential values for `OwnRent` are as follows

```
print(acs.OwnRent.unique())
['Mortgage' 'Rented' 'OutOfBounds']
```

An example interpretation of these dummy variables would be the odds of the `FamilyIncome` being greater than 150,000 increases by 1.82 times when the home is owned `outright` versus being under a `Mortgage`.

13.2.2 Using sklearn

When using `sklearn`, remember dummy variables need to be manually created.

```
predictors = pd.get_dummies(
    acs[['HouseCosts', 'NumWorkers', 'OwnRent', 'NumBedrooms',
         'FamilyType']], 
    drop_first=True)
```

We can use the `LogisticRegression` object from the `linear_model` module.

```
from sklearn import linear_model
lr = linear_model.LogisticRegression()
```

We can fit our model just like before when we fitted a linear regression

```
results = lr.fit(X = predictors, y = acs['ge150k_i'])
```

and get our coefficients

```
print(results.coef_)

[[ 7.09576796e-04      5.59835691e-01      2.22619419e-01      1.18014648e+00
 -7.30046173e-01      3.18642512e-01      1.21313432e+00]]
```

and intercept

```
print(results.intercept_)

[-5.49270525]
```

We can print out our results in a nicer format

```
values = np.append(results.intercept_, results.coef_)
# get the names of the values
names = np.append('intercept', predictors.columns)

# put everything in a labeled dataframe
results = pd.DataFrame(values, index = names,
                        columns=['coef']) # you need the square brackets here
)

print(results)

            coef
intercept    -5.492705
HouseCosts     0.000710
NumWorkers      0.559836
NumBedrooms     0.222619
OwnRent_Outright  1.180146
OwnRent_Rented   -0.730046
FamilyType_Male Head  0.318643
FamilyType_Married  1.213134
```

In order to interpret our coefficients, we still need to exponentiate our values.

```
results['or'] = np.exp(results['coef'])
print(results)

            coef          or
intercept    -5.492705    0.004117
HouseCosts     0.000710    1.000710
NumWorkers      0.559836    1.750385
NumBedrooms     0.222619    1.249345
OwnRent_Outright  1.180146    3.254851
OwnRent_Rented   -0.730046    0.481887
FamilyType_Male Head  0.318643    1.375260
FamilyType_Married  1.213134    3.364012
```

13.3 Poisson Regression

Poisson regression is performed when our response variable involves count data. For example in our `acs` data, the `NumChildren` variable is an example of count data.

13.3.1 Using statsmodels

We can perform a Poisson regression using the `poisson` function in `statsmodels`

```
results = smf.poisson(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs).fit()
```

```

Optimization terminated successfully.
    Current function value: 1.348824
    Iterations 7

print(results.summary())
                Poisson Regression Results
=====
Dep. Variable:      NumChildren      No. Observations:      22745
Model:              Poisson          Df Residuals:          22739
Method:             MLE              Df Model:                  5
Date:        Tue, 12 Sep 2017   Pseudo R-squ.:     0.009627
Time:           04:37:18       Log-Likelihood:   -30679.
converged:         True            LL-Null:        -30977.
                           LLR p-value:  1.190e-126
=====
            coef      std err      z  P>|z|      [0.025      0.975]
-----
Intercept      -0.3257      0.021    -15.490  0.000      -0.367     -0.284
FamilyType[T.Male Head] -0.0630      0.038    -1.637  0.102      -0.138     0.012
FamilyType[T.Married]    0.1440      0.021     6.707  0.000      0.102     0.186
OwnRent[T.Outright]     -1.9737      0.230    -8.599  0.000      -2.424     -1.524
OwnRent[T.Rented]        0.4086      0.021    19.772  0.000      0.368     0.449
FamilyIncome      5.42e-07      6.57e-08    8.247  0.000      4.13e-07    6.71e-07
=====
```

The benefit of using a GLM model is that the only thing that needs to be changed is the `family` of the model that needs to be fit, and the `link` function that transforms our data. We can also use the more general `glm` function to perform all the same calculations.

```
import statsmodels
```

```
import statsmodels.api as sm
```

```
import statsmodels.formula.api as smf
```

```

model = smf.glm(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs,
    family=sm.families.Poisson(sm.genmod.families.links.log))

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/statsmodels/compat/pandas.py:56: FutureWarning: The
pandas.core.datetools module is deprecated and will be removed in a
future version. Please use the pandas.tseries module instead.
    from pandas.core import datetools
```

when using the `glm` function, we need to specify the `family`, which also takes a `link`. In this example we are using the `Poisson` family which comes from `sm.families.Poisson`, and the link comes from `sm.genmod.families.links.log`

We get the same values from before using this method.

```

results = model.fit()

print(results.summary())
                Generalized Linear Model Regression Results
=====
Dep. Variable:      NumChildren      No. Observations:      22745
Model:              GLM              Df Residuals:          22739
Model Family:      Poisson          Df Model:                  5
Link Function:     log              Scale:                  1.0
Method:             IRLS             Log-Likelihood:   -30679.
Date:        Tue, 12 Sep 2017   Deviance:        34643.
Time:           04:37:18       Pearson chi2:  3.34e+04
No. Iterations:      6
=====
            coef      std err      z  P>|z|      [0.025      0.975]
-----
```

Intercept	-0.3257	0.021	-15.490	0.000	-0.367	-0.284
FamilyType[T.Male Head]	-0.0630	0.038	-1.637	0.102	-0.138	0.012
FamilyType[T.Married]	0.1440	0.021	6.707	0.000	0.102	0.186
OwnRent[T.Outright]	-1.9737	0.230	-8.599	0.000	-2.424	-1.524
OwnRent[T.Rented]	0.4086	0.021	19.772	0.000	0.368	0.449
FamilyIncome	5.42e-07	6.57e-08	8.247	0.000	4.13e-07	6.71e-07

13.3.2 Negative Binomial Regression for Overdispersion

If our assumptions for Poisson regression are violated, that is our data has overdispersion, we can perform a negative binomial regression instead

```
model = smf.glm(
    'NumChildren ~ FamilyIncome + FamilyType + OwnRent',
    data=acs,
    family=sm.families.NegativeBinomial(sm.genmod.families.links.log))
results = model.fit()

print(results.summary())
      Generalized Linear Model Regression Results
=====
Dep. Variable:      NumChildren   No. Observations:      22745
Model:              GLM            Df Residuals:          22739
Model Family:      NegativeBinomial  Df Model:             5
Link Function:     log            Scale:               0.778781336189
Method:             IRLS           Log-Likelihood:       -29749.
Date:              Tue, 12 Sep 2017 Deviance:            20731.
Time:              04:37:19        Pearson chi2:         1.77e+04
No. Iterations:    6
=====
      coef    std err      z   P>|z|    [0.025    0.975]
-----
Intercept      -0.3345    0.025  -13.226   0.000   -0.384   -0.285
FamilyType[T.Male Head] -0.0468    0.046  -1.025   0.305   -0.136   0.043
FamilyType[T.Married]   0.1529    0.026   5.892   0.000    0.102   0.204
OwnRent[T.Outright]   -1.9737    0.215  -9.193   0.000   -2.394   -1.553
OwnRent[T.Rented]     0.4164    0.027  15.586   0.000    0.364   0.469
FamilyIncome        5.398e-07  8.43e-08   6.405   0.000   3.75e-07  7.05e-07
=====
```

13.4 More Generalized Linear Models

`statsmodels`'s documentation page on GLM¹ lists the various families that can be passed into the `glm` parameter. The families are can all be found under `sm. families .<FAMILY>`

- Binomial
- Gamma
- InverseGaussian
- NegativeBinomial
- Poisson
- Tweedie

¹`statsmodels`GLM documentation: <http://www.statsmodels.org/dev/glm.html>

The link functions are found under `sm. families . family .<FAMILY>.links`. Below is the list of link functions, but note that not all link functions are available for each family.

- CDFLink
- CLogLog
- Log
- Logit
- NegativeBinomial
- Power
- cauchy
- identity
- inverse_power
- inverse_squared

13.5 Survival Analysis

While not technically a regression method, survival analysis is used when time to a certain event is modeled. For example, when looking at medical research and seeing if one treatment prevents a serious adverse event (e.g., death) over standard or a different treatment. Survival analysis, is also used when data is censored. Meaning the exact outcome of an event is not entirely known. For example, patients who follow a treatment regimen who are lost to follow-up.

Survival analysis is performed using the lifelines library². We will be using the `bladder` data from the R `survival` package which contains data on recurrences of bladder cancer for a given treatment.

²lifelines documentation: <https://lifelines.readthedocs.io/en/latest/>

```
bladder = pd.read_csv('~/data/bladder.csv')
print(bladder.head())

      id      rx  number   size   stop  event  enum
0      1       1      1      3      1      0      1
1      1       1      1      3      1      0      2
2      1       1      1      3      1      0      3
3      1       1      1      3      1      0      4
4      2       1      2      1      4      0      1
```

Here are the counts of the different treatments, `rx`.

```
print(bladder['rx'].value_counts())
1    188
2    152
Name: rx, dtype: int64
```

To perform our survival analysis, we import the `KaplanMeierFitter` from the `lifelines` library.

```
# pip install lifelines
```

```
from lifelines import KaplanMeierFitter
```

Creating the model and fitting the data is similar to how models are fit using `sklearn`. The `stop` variable shows when an event occurs, and the `event` variable signals if the event of interest (bladder cancer re-occurrence) occurred. The `event` value can have a value of `0`, because people can be lost to follow-up. This type of data is what is called “censored”.

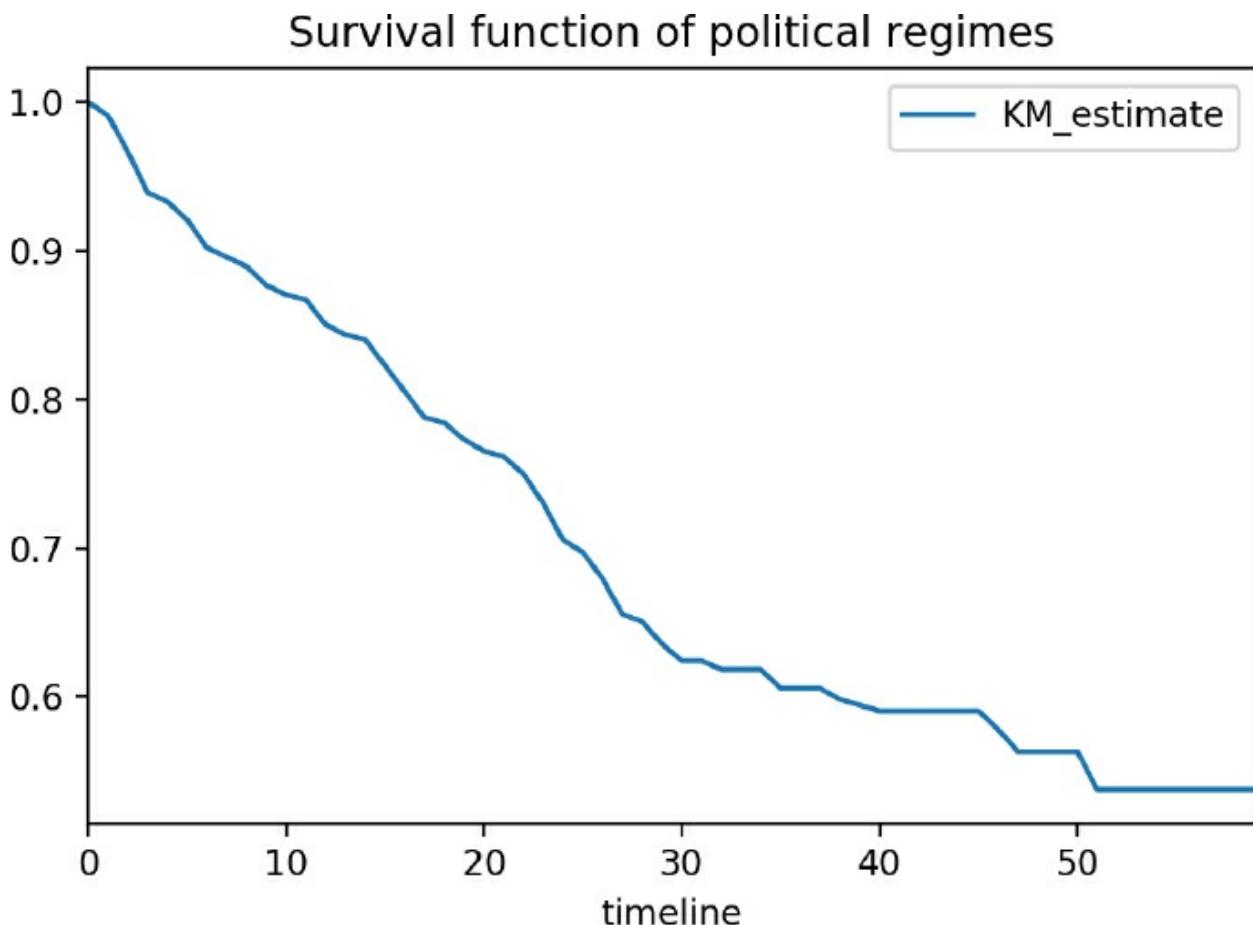
```
kmf = KaplanMeierFitter()  
kmf.fit(bladder['stop'], event_observed=bladder['event'])  
  
<lifelines.KaplanMeierFitter: fitted with 340 observations, 228  
censored>
```

We can plot the survival curve using `matplotlib` as shown in [Figure 13-1](#)

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

```
fig, ax = plt.subplots()  
ax = kmf.survival_function_.plot(ax=ax)  
ax.set_title('Survival function of political regimes')  
plt.show()
```

Figure 13-1: Survival function of political regimes using the KaplanMeierFitter



We can also show the confidence interval of our survival curve as shown in [Figure 13-2](#)

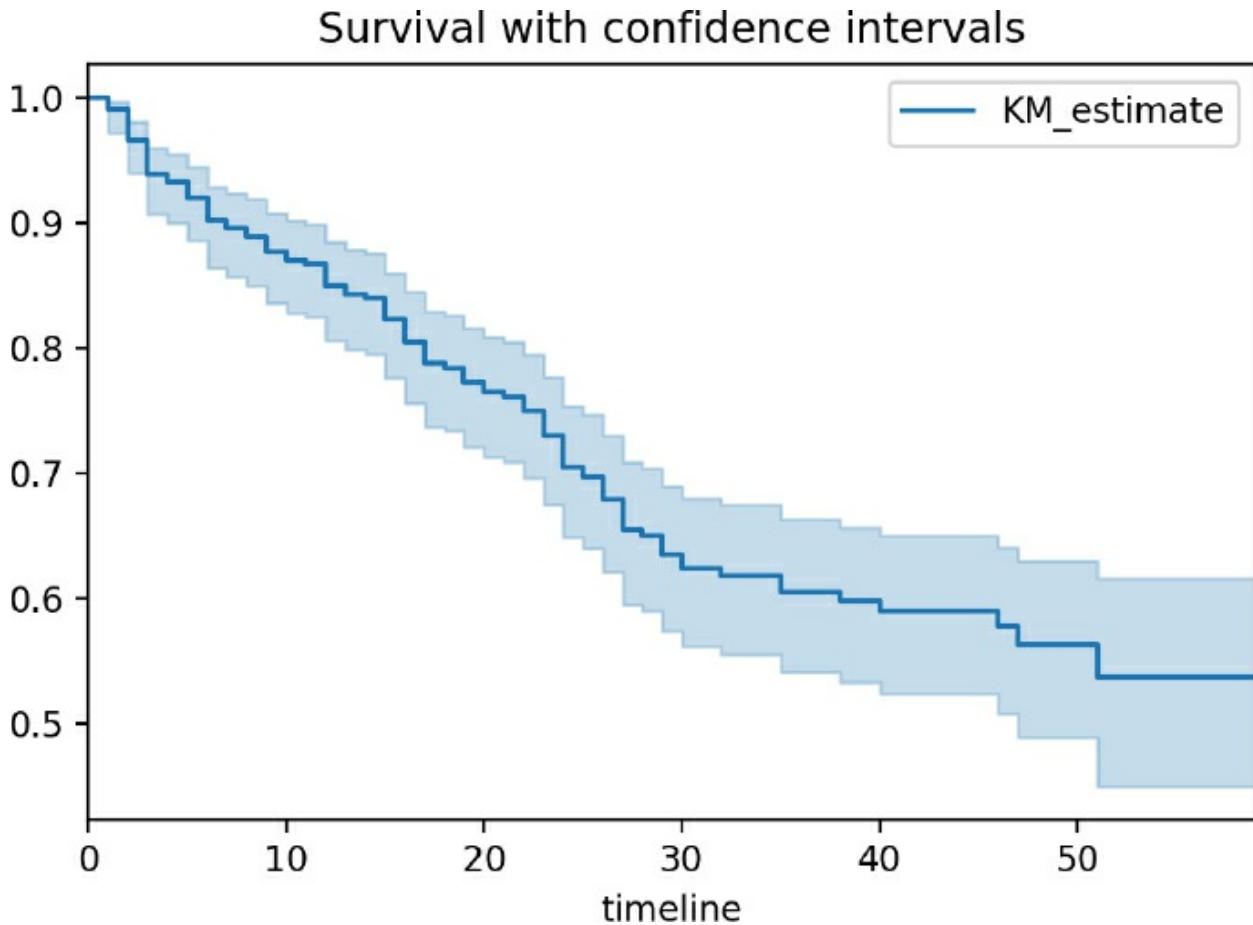
```
fig, ax = plt.subplots()  
ax = kmf.plot(ax=ax)  
ax.set_title('Survival with confidence intervals')  
plt.show()
```

So far we've just plotted the survival curve. We can also fit a model to predict survival rate. One such model is called the cox proportional hazards model. We fit this model using the `CoxPHFitter` class from `lifelines`.

```
from lifelines import CoxPHFitter
```

```
cph = CoxPHFitter()
```

Figure 13-2: Survival function of political regimes with confidence intervals



We then pass in the columns to be used as predictors.

```
cph_bladder_df = bladder[['rx', 'number', 'size',
                           'enum', 'stop', 'event']]
cph.fit(cph_bladder_df, duration_col='stop', event_col='event')

<lifelines.CoxPHFitter: fitted with 340 observations, 228 censored>
```

We can then use the `print_summary` method to print out the coefficients

```
print(cph.print_summary())

n=340, number of events=112

      coef      exp(coef)      se(coef)        z      p      lower 0.95      upper 0.95
rx    -0.5974      0.5502     0.2009 -2.9738 0.0029      -0.9912     -0.2036    **
number  0.2175      1.2430     0.0465  4.6756 0.0000       0.1263      0.3087    ***
size   -0.0568      0.9448     0.0709 -0.8007 0.4233      -0.1958      0.0822
enum   -0.6038      0.5467     0.0940 -6.4231 0.0000      -0.7881     -0.4195    ***
---
Signif. codes:  0 '****' 0.001  ** '***' 0.01  * '0.05 .'  . '0.1 ' ' 1
```

```
Concordance = 0.753
None
```

13.5.1 Testing the Cox model assumptions

One way to check the Cox model assumptions is to plot a separate survival curve by a strata. In our example, our strata will be the `rx` column, meaning we will plot a separate curve for each type of treatment. If the `log(-log(survival curve))` vs `log(time)` cross each other ([Figure 13-3](#)), then it signals that the model needs to be stratified by the variable.

```
rx1 = bladder.loc[bladder['rx'] == 1]
rx2 = bladder.loc[bladder['rx'] == 2]

kmf1 = KaplanMeierFitter()
kmf1.fit(rx1['stop'], event_observed=rx1['event'])

kmf2 = KaplanMeierFitter()
kmf2.fit(rx2['stop'], event_observed=rx2['event'])

fig, axes = plt.subplots()

# put both plots on the same axes
kmf1.plot_loglogs(ax=axes)
kmf2.plot_loglogs(ax=axes)

axes.legend(['rx1', 'rx2'])

plt.show()
```

Since the lines to cross each other, it makes sense to stratify our analysis.

```
cph_strat = CoxPHFitter()
cph_strat.fit(cph_bladder_df, duration_col='stop', event_col='event',
               strata=['rx'])
print(cph_strat.print_summary())
n=340, number of events=112

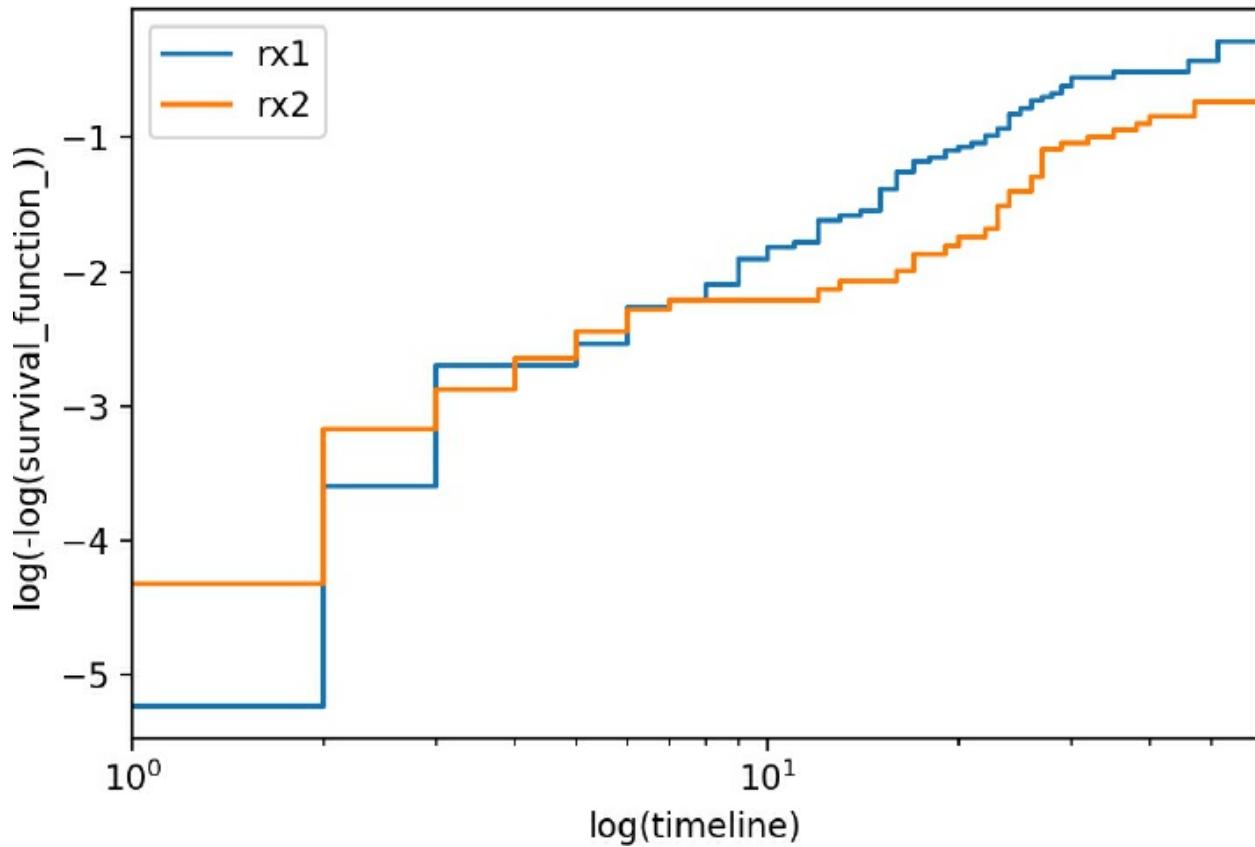
      coef      exp(coef)      se(coef)        z        p      lower 0.95      upper 0.95
number  0.2137      1.2383     0.0465  4.5978 0.0000      0.1226     0.3048    ***
size   -0.0549      0.9466     0.0710 -0.7728 0.4396     -0.1940     0.0843
enum   -0.6070      0.5450     0.0941 -6.4512 0.0000     -0.7914     -0.4225    ***
---
Signif. codes:  0 '****' 0.001  '**' 0.01 '*' 0.05 '.'  0.1 ' ' 1

Concordance = 0.733
None
```

13.6 Conclusion

This chapter covered some of the most basic and common models used. These types of models serve as an interpretable baseline to more complex machine learning models. As we cover more complex models, keep in mind that sometimes simple and tried-and-true interpretable models can outperform the fancy newer models.

Figure 13-3: Plotting separate survival curves to check the Cox model assumption



Chapter 14. Model Diagnostics

14.1 Introduction

Building models is a continuous art. As we start adding and removing variables from our model we need a means to compare models with one another and need a consistent way of measuring model performance. There are many ways we can compare models and this chapter will go about some these methods.

14.2 Residuals

The residuals of a model compare what the model calculates and the actual values in the data. Let's fit some models on a housing dataset.

```
import pandas as pd
```

```
housing = pd.read_csv('../data/housing_renamed.csv')
```

```
print(housing.head())
```

```
      neighborhood          type     units   year_built    sq_ft    income \
0       FINANCIAL  R9-CONDOMINIUM      42    1920.0    36500  1332615
1       FINANCIAL  R4-CONDOMINIUM      78    1985.0    126420  6633257
2       FINANCIAL  RR-CONDOMINIUM     500        NaN  554174  17310000
3       FINANCIAL  R4-CONDOMINIUM     282    1930.0    249076  11776313
4      TRIBECA    R4-CONDOMINIUM     239    1985.0    219495  10004582

  income_per_sq_ft     expense  expense_per_sq_ft    net_income \
0         36.51    342005            9.37      990610
1         52.47   1762295           13.94     4870962
2         31.24   3543000            6.39    13767000
3         47.28   2784670           11.18    8991643
4         45.58   2783197           12.68    7221385

      value  value_per_sq_ft      boro
0  7300000        200.00  Manhattan
1 30690000        242.76  Manhattan
2 90970000        164.15  Manhattan
3 67556006        271.23  Manhattan
4 54320996        247.48  Manhattan
```

We'll begin with a multiple linear regression model with 3 covariates.

```
import statsmodels
```

```
import statsmodels.api as sm
```

```
import statsmodels.formula.api as smf
```

```
house1 = smf.glm('value_per_sq_ft ~ units + sq_ft + boro',
                  data=housing).fit()
print(house1.summary())
```

```
Generalized Linear Model Regression Results
=====
Dep. Variable:      value_per_sq_ft    No. Observations:                 2626
Model:                          GLM    Df Residuals:                   2619
Model Family:                     Gaussian    Df Model:                      6
```

```

Link Function:           identity  Scale:          1879.49193485
Method:                 IRLS    Log-Likelihood:   -13621.
Date:      Tue, 12 Sep 2017 Deviance:        4.9224e+06
Time:       05:07:05     Pearson chi2:      4.92e+06
No. Iterations:          2
=====
              coef    std err      z  P>|z|    [0.025  0.975]
-----
Intercept      43.2909    5.330    8.122  0.000    32.845  53.737
boro[T.Brooklyn] 34.5621    5.535    6.244  0.000    23.714  45.411
boro[T.Manhattan] 130.9924   5.385   24.327  0.000   120.439  141.546
boro[T.Queens]   32.9937    5.663    5.827  0.000    21.895  44.092
boro[T.Staten Island] -3.6303   9.993   -0.363  0.716   -23.216  15.956
units          -0.1881    0.022   -8.511  0.000   -0.231  -0.145
sq_ft           0.0002   2.09e-05  10.079  0.000    0.000  0.000
=====
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/statsmodels/compat/pandas.py:56: FutureWarning: The
pandas.core.datetools module is deprecated and will be removed in a
future version. Please use the pandas.tseries module instead.
  from pandas.core import datetools

```

We can plot the residuals of our model ([Figure 14-1](#)). What we are looking for is a plot with a random scattering of points. If there is a pattern, then we will need to investigate our data and model to see why.

```
import seaborn as sns
```

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

```

fig, ax = plt.subplots()
ax = sns.regplot(x=house1.fittedvalues,
                  y=house1.resid_deviance, fit_reg=False)
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/resid_1')

```

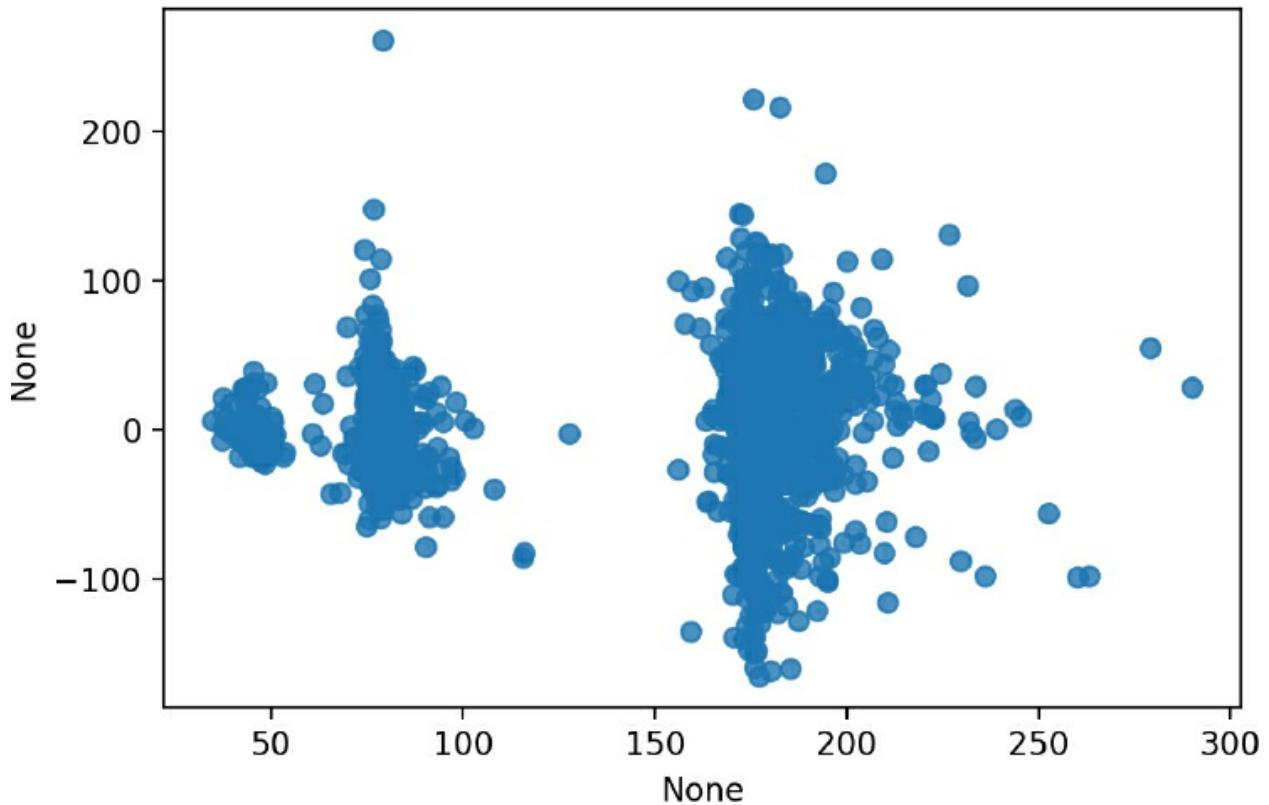
This plot is extremely concerning since there are obvious clusters and groups in the residual plot. However we can color our plot by the `boro` variable ([Figure 14-2](#)).

```

res_df = pd.DataFrame({
    'fittedvalues': house1.fittedvalues,
    'resid_deviance': house1.resid_deviance,
    'boro': housing['boro']
})

```

Figure 14-1: Residuals of the housel model



```
fig = sns.lmplot(x='fittedvalues',
                  y='resid_deviance',
                  data=res_df, hue='boro', fit_reg=False)
plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/resid_boros')
```

When we color our points by boro you can see that the clusters are highly governed by the boro.

14.2.1 Q-Q Plots

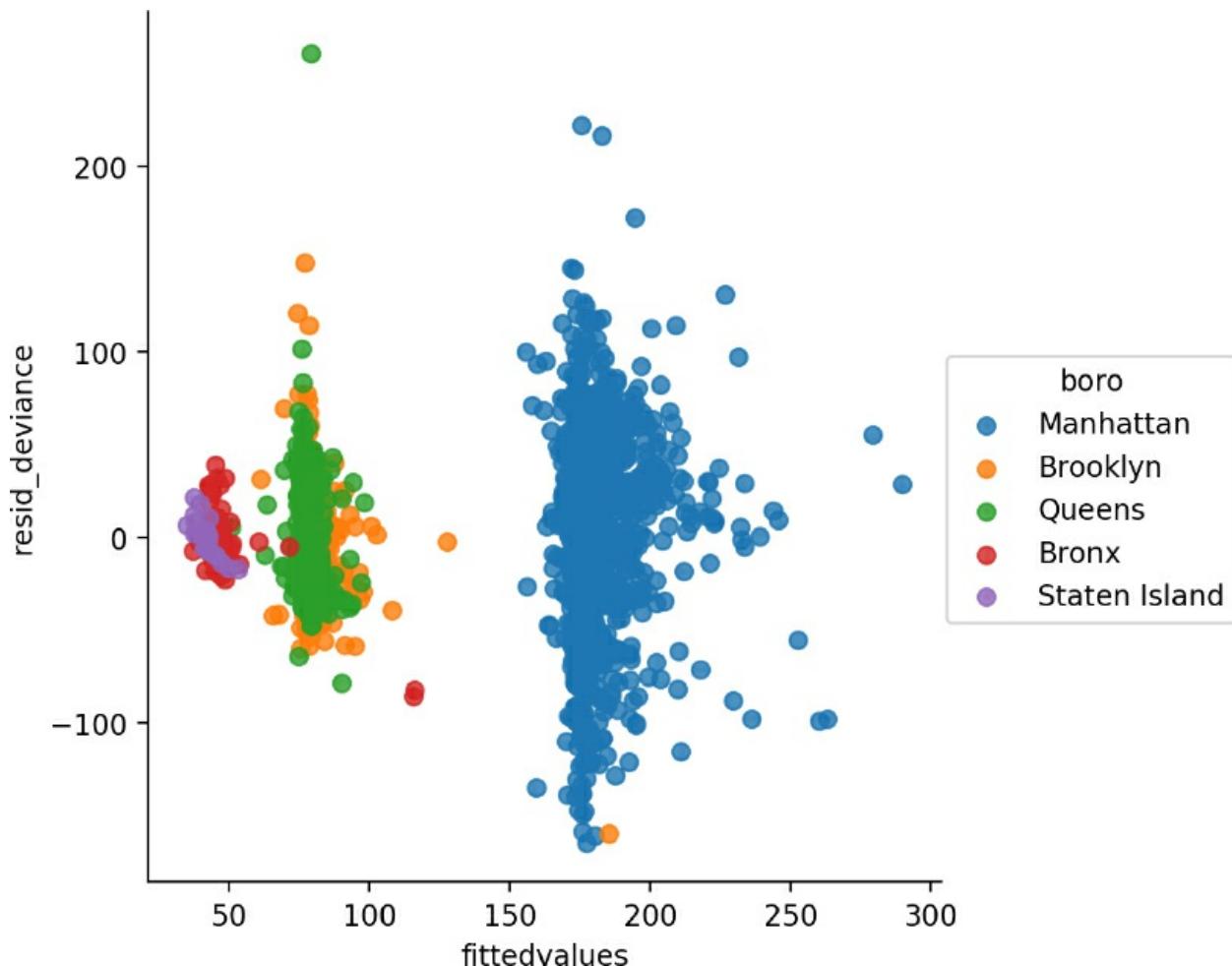
A q-q plot is a graphical technique that compares whether your data conforms to a reference distribution. Since many models assume your data is normally distributed, a q-q plot is one way to make sure your data is normal.

```
from scipy import stats
```

```
resid = house1.resid_deviance.copy()
resid_std = stats.zscore(resid)

fig = statsmodels.graphics.gofplots.qqplot(resid, line='r')
plt.show()
```

Figure 14-2: Residuals of the house model colored by boro



```
fig.savefig('p5-ch-model_diagnostics/figures/house_1_qq')
```

We can also plot a histogram of the residuals to see if our data is normal or not ([Figure 14-4](#)).

```
fig, ax = plt.subplots()
ax = sns.distplot(resid_std)
plt.show()

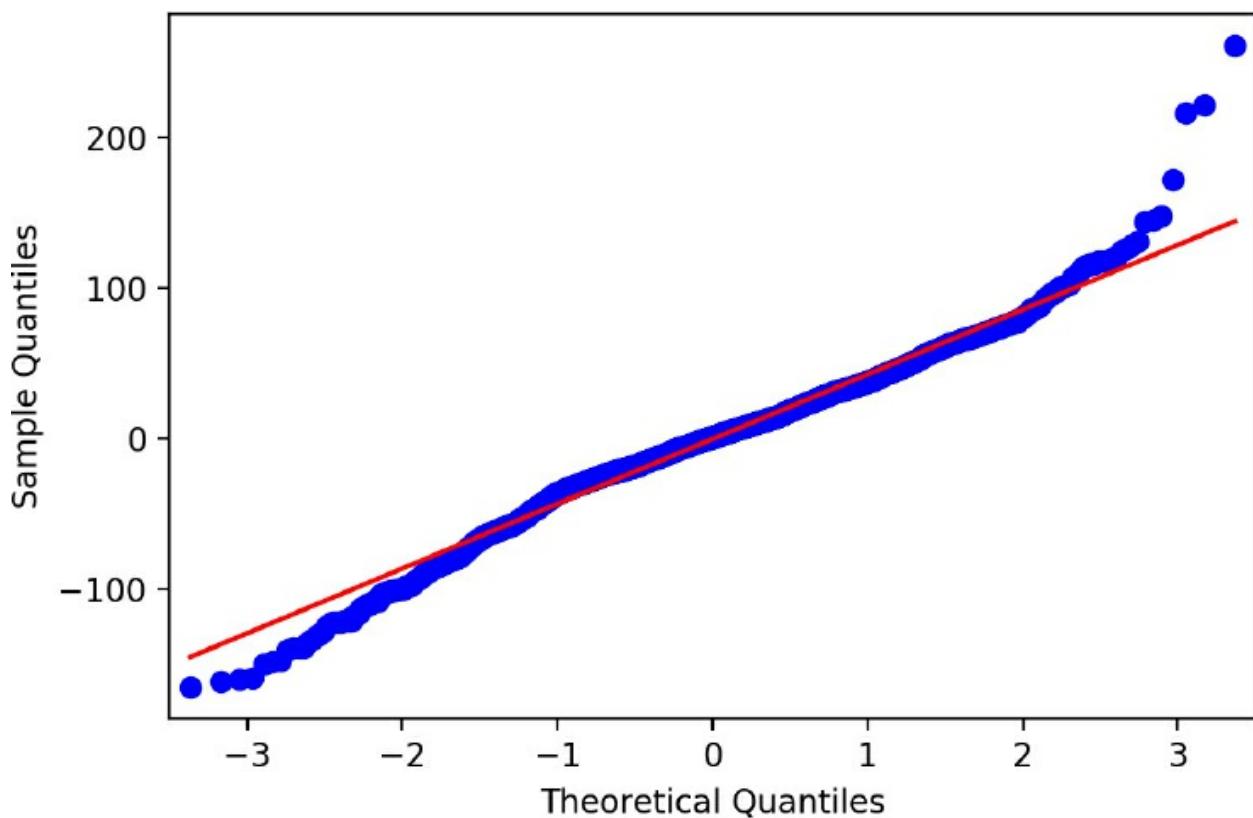
fig.savefig('p5-ch-model_diagnostics/figures/house1_resid_std')
```

If the points on the q-q plot lie on the red line, then that means our data matches our reference distribution. If it doesn't than one thing we can do is apply a transformation to our data. [Table 14-1](#) shows what transformations can be performed on your data. If the q-q plot of points is convex compared to the red reference line, then you can transform your data towards the top of the table. If the q-q plot of points is concave compared to the red reference line, then you can transform your data towards the bottom of the table.

Table 14–1: Table of transformations

x^p	Equivalent	Description
x^2	x^2	square
x^1	x	
$x^{\frac{1}{2}}$	\sqrt{x}	square root
"x" x	$\log(x)$	log
$x^{-\frac{1}{2}}$	$\frac{1}{\sqrt{x}}$	reciprocal square root
x^{-1}	$\frac{1}{x}$	reciprocal
x^{-2}	$\frac{1}{x^2}$	reciprocal square

Figure 14-3: Q-Q plot of the housel model



14.3 Comparing Multiple Models

Now that we know how to assess a single model, we need a means to compare multiple models to each other to be able to pick the “best” one.

14.3.1 Working with Linear Models

We begin by fitting five models. Note that some of the models use the `+` operator to add covariates to the model, and others have the `*` operator. The `*` is how we specify an interaction in our model. That is, the variables that are interacting are behaving in a way that is not independent from one another and are “interacting” in a way that their values affect one another, and are not simply additive.

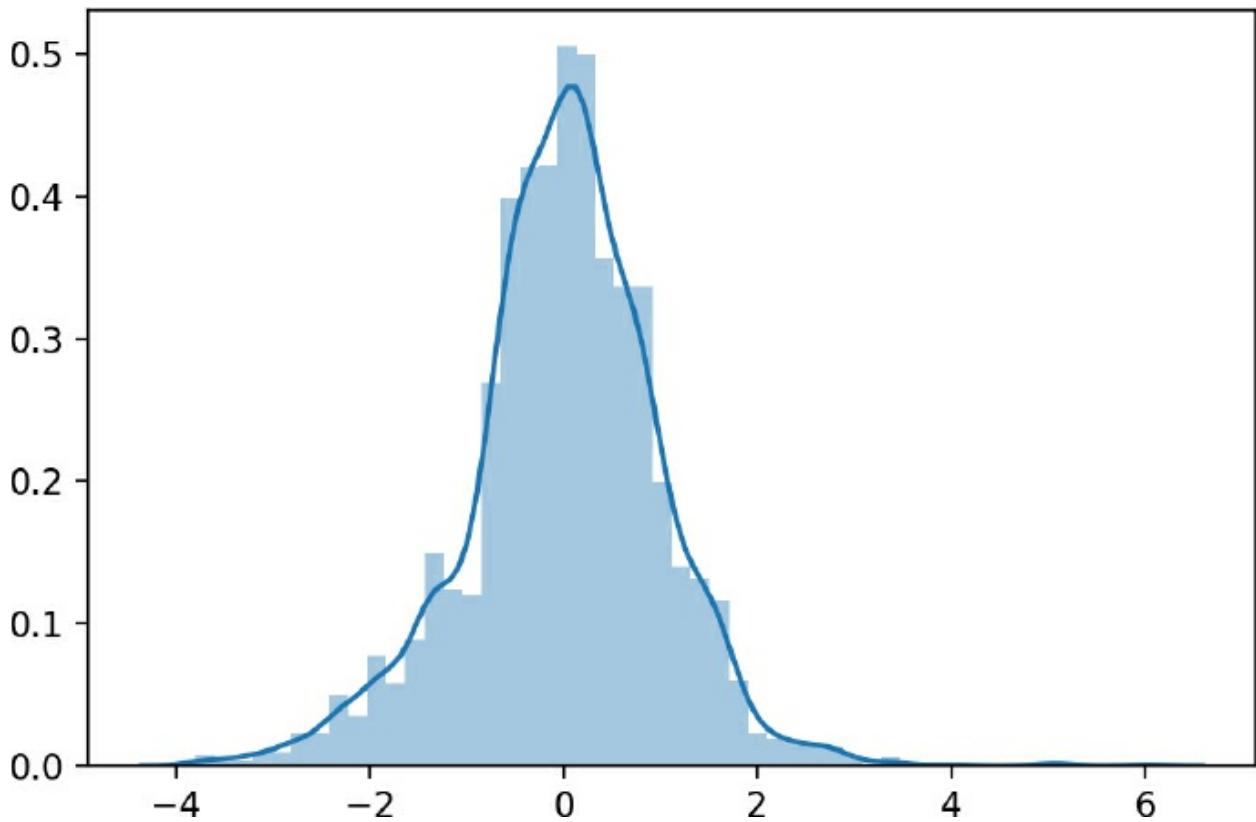
```

# The original housing dataset has a column named class
# this would cause an error if used 'class'
# it would cause an error since 'class' is a python keyword
# the column was renamed to 'type'

f1 = 'value_per_sq_ft ~ units + sq_ft + boro'
f2 = 'value_per_sq_ft ~ units * sq_ft + boro'
f3 = 'value_per_sq_ft ~ units + sq_ft * boro + type'

```

Figure 14-4: Histogram of the the housel model residuals



```

f4 = 'value_per_sq_ft ~ units + sq_ft * boro + sq_ft * type'
f5 = 'value_per_sq_ft ~ boro + type'

housel = smf.ols(f1, data=housing).fit()
house2 = smf.ols(f2, data=housing).fit()
house3 = smf.ols(f3, data=housing).fit()
house4 = smf.ols(f4, data=housing).fit()
house5 = smf.ols(f5, data=housing).fit()

```

With all our models, we can collect all of our coefficients and the model they are associated with.

```
mod_results = pd.concat([house1.params, house2.params, house3.params,
                        house4.params, house5.params], axis=1).\
    rename(columns=lambda x: 'house' + str(x + 1)).\
    reset_index().\
    rename(columns={'index': 'param'}).\
    melt(id_vars='param', var_name='model', value_name='estimate')

print(mod_results.head())

      param      model   estimate
0   Intercept  house1  43.290863
1  boro[T.Brooklyn]  house1  34.562150
2  boro[T.Manhattan]  house1 130.992363
3  boro[T.Queens]  house1  32.993674
4  boro[T.Staten Island]  house1 -3.630251

print(mod_results.tail())

      param      model   estimate
85 type[T.R4-CONDOMINIUM]  house5  20.457035
86 type[T.R9-CONDOMINIUM]  house5  1.293322
87 type[T.RR-CONDOMINIUM]  house5 -11.680515
88          units  house5       NaN
89      units:sq_ft  house5       NaN
```

Since it's not very useful to look at a large column of values, we can plot our coefficients to quickly see how the models are estimating parameters in relation to each other ([Figure 14-5](#)).

```
fig, ax = plt.subplots()
ax = sns.pointplot(x="estimate", y="param", hue="model",
                    data=mod_results,
                    dodge=True, # jitter the points
                    join=False) # don't connect the points

plt.show()

fig.savefig('p5-ch-model_diagnostics/figures/coef_house_1-4')
```

Now that we have our linear models, we can use the ANOVA to compare them. The ANOVA will give us the residual sum of squares (RSS) which is one way we can measure performance (lower the better).

```
model_names = ['house1', 'house2', 'house3', 'house4', 'house5']
house_anova = statsmodels.stats.anova.anova_lm(
    house1, house2, house3, house4, house5)
house_anova.index = model_names
print(house_anova)

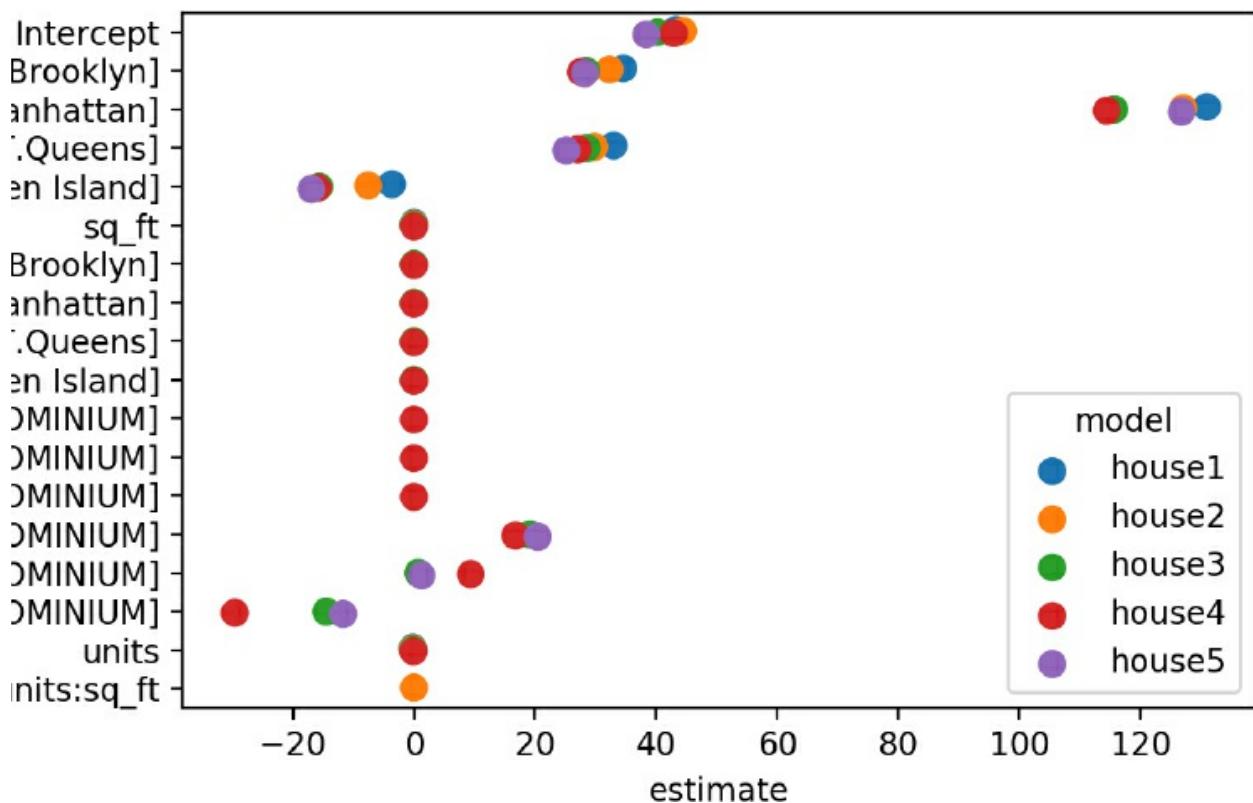
      df_resid        ssr      df_diff      ss_diff           F      \
house1     2619.0  4.922389e+06        0.0        NaN        NaN
house2     2618.0  4.884872e+06        1.0  37517.437605  20.039049
house3     2612.0  4.619926e+06        6.0  264945.539994  23.585728
house4     2609.0  4.576671e+06        3.0   43255.441192   7.701289
house5     2618.0  4.901463e+06       -9.0 -324791.847907  19.275539

      Pr(>F)
house1      NaN
house2  7.912333e-06
house3  2.754431e-27
house4  4.025581e-05
house5      NaN

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:879: RuntimeWarning:
invalid value encountered in greater
    return (self.a < x) & (x < self.b)
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:879: RuntimeWarning:
invalid value encountered in less
    return (self.a < x) & (x < self.b)
```

```
/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/scipy/stats/_distn_infrastructure.py:1818: RuntimeWarning:
invalid value encountered in less_equal
    cond2 = cond0 & (x <= self.a)
```

Figure 14-5: Coefficients of house1 to house4 models



Another way we can calculate model performance is using AIC and BIC. What these methods do is put a penalty for each feature that is added to the model. So it strives to balance performance and parsimony (lower is better).

```
house_models = [house1, house2, house3, house4, house5]

house_aic = list(
    map(statsmodels.regression.linear_model.RegressionResults.aic,
        house_models))
house_bic = list(
    map(statsmodels.regression.linear_model.RegressionResults.bic,
        house_models))

# remember dicts are unordered
abic = pd.DataFrame({
    'model': model_names,
    'aic': house_aic,
    'bic': house_bic
})

print(abic)

      aic          bic   model
0  27256.031113  27297.143632  house1
1  27237.939618  27284.925354  house2
2  27103.502577  27185.727615  house3
3  27084.800043  27184.644733  house4
4  27246.843392  27293.829128  house5
```

14.3.2 Working with GLM models

We can perform the same calculations and model diagnostics on GLM models. However, the ANOVA is simply the deviance of the model.

```
def anova_deviance_table(*models):
    return pd.DataFrame({
        'df_residuals': [i.df_resid for i in models],
        'resid_stddev': [i.deviance for i in models],
        'df': [i.df_model for i in models],
        'deviance': [i.deviance for i in models]
    })

f1 = 'value_per_sq_ft ~ units + sq_ft + boro'
f2 = 'value_per_sq_ft ~ units * sq_ft + boro'
f3 = 'value_per_sq_ft ~ units + sq_ft * boro + type'
f4 = 'value_per_sq_ft ~ units + sq_ft * boro + sq_ft * type'
f5 = 'value_per_sq_ft ~ boro + type'

glm1 = smf.glm(f1, data=housing).fit()
glm2 = smf.glm(f2, data=housing).fit()
glm3 = smf.glm(f3, data=housing).fit()
glm4 = smf.glm(f4, data=housing).fit()
glm5 = smf.glm(f5, data=housing).fit()

glm_anova = anova_deviance_table(glm1, glm2, glm3, glm4, glm5)
print(glm_anova)

      deviance      df      df_residuals      resid_stddev
0   4.922389e+06      6          2619      4.922389e+06
1   4.884872e+06      7          2618      4.884872e+06
2   4.619926e+06     13          2612      4.619926e+06
3   4.576671e+06     16          2609      4.576671e+06
4   4.901463e+06      7          2618      4.901463e+06
```

We can do the same set of calculations in a logistic regression

```
# create a binary variable
housing['high_value'] = (housing['value_per_sq_ft'] >= 150).\
    astype(int)

print(housing['high_value'].value_counts())

0    1619
1    1007
Name: high_value, dtype: int64

# create and fit our logistic regression using glm

f1 = 'high_value ~ units + sq_ft + boro'
f2 = 'high_value ~ units * sq_ft + boro'
f3 = 'high_value ~ units + sq_ft * boro + type'
f4 = 'high_value ~ units + sq_ft * boro + sq_ft * type'
f5 = 'high_value ~ boro + type'

logistic = statsmodels.genmod.families.family.Binomial(
    link=statsmodels.genmod.families.links.logit
)

glm1 = smf.glm(f1, data=housing, family=logistic).fit()
glm2 = smf.glm(f2, data=housing, family=logistic).fit()
glm3 = smf.glm(f3, data=housing, family=logistic).fit()
glm4 = smf.glm(f4, data=housing, family=logistic).fit()
glm5 = smf.glm(f5, data=housing, family=logistic).fit()

# show the deviances from our glm models
print(anova_deviance_table(glm1, glm2, glm3, glm4, glm5))

      deviance      df      df_residuals      resid_stddev
0   1695.631547      6          2619      1695.631547
1   1686.126740      7          2618      1686.126740
2   1636.492830     13          2612      1636.492830
3   1619.431515     16          2609      1619.431515
4   1666.615696      7          2618      1666.615696
```

Finally, we can create a table of AIC and BIC values.

```
mods = [glm1, glm2, glm3, glm4, glm5]

mods_aic = list(
    map(statsmodels.regression.linear_model.RegressionResults.aic,
        mods))
mods_bic = list(
    map(statsmodels.regression.linear_model.RegressionResults.bic,
        mods))

# remember dicts are unordered
abic = pd.DataFrame({
    'model': model_names,
    'aic': house_aic,
    'bic': house_bic
})

print(abic)

      aic          bic   model
0  27256.031113  27297.143632  house1
1  27237.939618  27284.925354  house2
2  27103.502577  27185.727615  house3
3  27084.800043  27184.644733  house4
4  27246.843392  27293.829128  house5
```

Looking at all these measures we can say Model 4 is performing the best so far.

14.4 K-fold Cross Validation

Cross validation is another technique to compare models. One of the main benefits is that it can account for how well your model performs on new data. It does this by partitioning your data into k parts. It will hold one of the parts aside as the “test” set and then fit the model on the remaining k—1 parts, the “training” set. The fitted model will then be used on the “test” and an error rate will be calculated. It will repeat this process until all k parts have been used as a “test” set. The final error of the model will be some average across all the models.

There are many different ways cross validation can be performed. The method just described is called “k-fold cross validation”. But there are other ways of performing cross validation, such as “leave-one-out cross validation”. Where the training data is all the data except 1 observation designated as the test set.

Here we will split our data into a testing and training dataset

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

print(housing.columns)

Index(['neighborhood', 'type', 'units', 'year_built', 'sq_ft',
       'income', 'income_per_sq_ft', 'expense', 'expense_per_sq_ft',
       'net_income', 'value', 'value_per_sq_ft', 'boro',
       'high_value'],
      dtype='object')

# get training and test data
X_train, X_test, y_train, y_test = train_test_split(
    pd.get_dummies(housing[['units', 'sq_ft', 'boro']],
                  drop_first=True),
    housing['value_per_sq_ft'],
    test_size=0.20,
    random_state=42
)
```

We can get a score on how well our model is performing using our test data

```
lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))

0.613712528503
```

Since `sklearn` relies heavily on the `numpy ndarray`, the `patsy` library allows you to specify a formula just like the formula api in `statsmodels`, and it returns back a proper numpy array you can use in `sklearn`

Here is the same code as before, but using the `dmatrices` function in the `patsy` library.

```
from patsy import dmatrices

y, X = dmatrices('value_per_sq_ft ~ units + sq_ft + boro', housing,
                  return_type="dataframe")
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42)
lr = LinearRegression().fit(X_train, y_train)
print(lr.score(X_test, y_test))

0.613712528503
```

To perform a k-fold cross validation we need to import it from `sklearn`

```
from sklearn.model_selection import KFold, cross_val_score

# get a fresh new housing dataset
housing = pd.read_csv('../data/housing_renamed.csv')
```

We have to specify how many folds we want. This depends on how many rows of data you have. If you do not have too many observations in your data, you may opt to select a smaller `k` (e.g., 2). Otherwise a `k` between 5 to 10 is fairly common. However, keep in mind that the trade-off with higher `k` values is more computation time.

```
kf = KFold(n_splits=5)

y, X = dmatrices('value_per_sq_ft ~ units + sq_ft + boro', housing)
```

Next we can train and test our model on each fold

```
coefs = []
scores = []
for train, test in kf.split(X):
    X_train, X_test = X[train], X[test]
    y_train, y_test = y[train], y[test]
    lr = LinearRegression().fit(X_train, y_train)
    coefs.append(pd.DataFrame(lr.coef_))
    scores.append(lr.score(X_test, y_test))
```

And we can view the results

```
coefs_df = pd.concat(coefs)
coefs_df.columns = X.design_info.column_names
coefs_df
   Intercept      boro[T.Brooklyn]      boro[T.Manhattan]      boro[T.Queens] \\
0          0.0           33.369037           129.904011           32.103100
0          0.0           32.889925           116.957385           31.295956
0          0.0           30.975560           141.859327           32.043449
0          0.0           41.449196           130.779013           33.050968
0          0.0          -38.511915            56.069855          -17.557939
                                         \\
   boro[T.Staten Island]          units        sq_ft
0             -4.381085       -0.205890      0.000220
```

```

0           -4.919232      -0.146180      0.000155
0           -4.379916      -0.179671      0.000194
0           -3.430209      -0.207904      0.000232
0            0.000000      -0.145829      0.000202

```

We can take a look at the average coefficient across all folds using `apply` and the `np.mean` function

```

import numpy as np
print(coefs_df.apply(np.mean))
Intercept          0.000000
boro[T.Brooklyn]   20.034361
boro[T.Manhattan]  115.113918
boro[T.Queens]     22.187107
boro[T.Staten Island] -3.422088
units              -0.177095
sq_ft              0.000201
dtype: float64

```

Or look at our scores. Each model has a default scoring method. `LinearRegression` for example uses the R^2 coefficient of determination) regression score function¹².

¹sklearn R^2 scoring:

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html

²wikipedia coefficient of determination:

https://en.wikipedia.org/wiki/Coefficient_of_determination

```

print(scores)

[0.027314162906420303, -0.55383622124079213, -0.15636371688048567,
 -0.32342020619288148, -1.6929655586930985]

```

We can also use `cross_val_scores` (for cross validation scores) to calculate CV scores

```

# use cross_val_scores to calculate CV scores
model = LinearRegression()
scores = cross_val_score(model, X, y, cv=5)
print(scores)

[ 0.02731416 -0.55383622 -0.15636372 -0.32342021 -1.69296556]

```

If we were to compare multiple models with each other, we would compare the average of the scores.

```

print(scores.mean())

-0.53985430802

```

Now we'll refit all our models again using k-fold cross validation.

```

# Create the predictor and response matrices
y1, X1 = dmatrices('value_per_sq_ft ~ units + sq_ft + boro',
                    housing)
y2, X2 = dmatrices('value_per_sq_ft ~ units*sq_ft + boro',
                    housing)
y3, X3 = dmatrices('value_per_sq_ft ~ units + sq_ft*boro + type',
                    housing)
y4, X4 = dmatrices('value_per_sq_ft ~ units + sq_ft*boro + sq_ft?type',
                    housing)
y5, X5 = dmatrices('value_per_sq_ft ~ boro + type', housing)

# Fit our models
model = LinearRegression()

```

```
scores1 = cross_val_score(model, X1, y1, cv=5)
scores2 = cross_val_score(model, X2, y2, cv=5)
scores3 = cross_val_score(model, X3, y3, cv=5)
scores4 = cross_val_score(model, X4, y4, cv=5)
scores5 = cross_val_score(model, X5, y5, cv=5)
```

We can now look at our CV Scores

```
scores_df = pd.DataFrame([scores1, scores2, scores3,
                           scores4, scores5])

print(scores_df.apply(np.mean, axis=1))

0    -5.398543e-01
1    -1.088184e+00
2    -3.569632e+26
3    -1.141180e+27
4    -3.227148e+25
dtype: float64
```

We can see again, that our 4th model is performing the best.

14.5 Conclusion

As we start working with models, it's important to measure their performances. Using the ANOVA for linear models, looking at deviance for GLM models, and using cross validation are all ways we can measure error and performance when trying to pick a best model.

Chapter 15. Regularization

15.1 Introduction

In [Chapter 14](#), we talked about various ways to measure model performance. [Section 14.4](#) talked about cross validation, a technique that tries to measure model performance by looking at how it predicts on test data. This Chapter will cover regularization, one technique to address better performance on test data. That is, it is a method that aims to prevent overfitting.

15.2 Why Regularize?

Let's begin with a base case of linear regression. We will be using the ACS data.

```
import pandas as pd
```

```
acs = pd.read_csv('../data/acs_ny.csv')
```

```
print(acs.columns)
```

```
Index(['Acres', 'FamilyIncome', 'FamilyType', 'NumBedrooms',
       'NumChildren', 'NumPeople', 'NumRooms', 'NumUnits',
       'NumVehicles', 'NumWorkers', 'OwnRent', 'YearBuilt',
       'HouseCosts', 'ElectricBill', 'FoodStamp', 'HeatingFuel',
       'Insurance', 'Language'],
      dtype='object')
```

Now, let's create our design matrices using patsy.

```
from patsy import dmatrices
```

```
response, predictors = dmatrices(
    'FamilyIncome ~ NumBedrooms + NumChildren + NumPeople + '
    'NumRooms + NumUnits + NumVehicles + NumWorkers + OwnRent + '
    'YearBuilt + ElectricBill + FoodStamp + HeatingFuel + '
    'Insurance + Language',
    data=acs
)
```

With our predictor and response matrices created, we can use `sklearn` to split our data into training and testing sets.

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(predictors, response, random_state=0)
```

Now, let's fit our linear model. Here we are normalizing our data so we can compare our coefficients when we use our regularization techniques.

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression(normalize=True).fit(X_train, y_train)

model_coefs = pd.DataFrame(list(zip(predictors.design_info.column_names,
                                      lr.coef_[0])),
                           columns=['variable', 'coef_lr'])
print(model_coefs)
```

	variable	coef_lr
0	Intercept	3.522660e-11
1	NumUnits[T.Single attached]	3.135646e+04
2	NumUnits[T.Single detached]	2.418368e+04
3	OwnRent[T.Ordinary]	2.839186e+04
4	OwnRent[T.Rented]	7.229586e+03
5	YearBuilt[T.1940-1949]	1.292169e+04
6	YearBuilt[T.1950-1959]	2.057793e+04
7	YearBuilt[T.1960-1969]	1.764835e+04
8	YearBuilt[T.1970-1979]	1.756881e+04
9	YearBuilt[T.1980-1989]	2.552566e+04
10	YearBuilt[T.1990-1999]	2.983944e+04
11	YearBuilt[T.2000-2004]	3.012502e+04
12	YearBuilt[T.2005]	4.318648e+04
13	YearBuilt[T.2006]	3.242038e+04
14	YearBuilt[T.2007]	3.562061e+04
15	YearBuilt[T.2008]	3.712470e+04
16	YearBuilt[T.2009]	3.035133e+04
17	YearBuilt[T.2010]	7.364529e+04
18	YearBuilt[T.Before 1939]	1.218711e+04
19	FoodStamp[T.Yes]	-2.745712e+04
20	HeatingFuel[T.Electricity]	1.946552e+04
21	HeatingFuel[T.Gas]	2.588482e+04
22	HeatingFuel[T.None]	2.532452e+04
23	HeatingFuel[T.Oil]	2.535803e+04
24	HeatingFuel[T.Other]	1.734533e+04
25	HeatingFuel[T.Solar]	8.424991e+03
26	HeatingFuel[T.Wood]	8.898002e+02
27	Language[T.English]	-1.873624e+04
28	Language[T.Other]	-4.463333e+03
29	Language[T.Other European]	-1.409466e+04
30	Language[T.Spanish]	-2.603347e+04
31	NumBedrooms	3.443931e+03
32	NumChildren	8.215723e+03
33	NumPeople	-8.203826e+03
34	NumRooms	5.735494e+03
35	NumVehicles	7.484535e+03
36	NumWorkers	2.283630e+04
37	ElectricBill	9.332524e+01
38	Insurance	3.099441e+01

Now, we can look at our model scores.

```
print(lr.score(X_train, y_train))

0.272614046564

print(lr.score(X_test, y_test))

0.269769795685
```

In this particular case, we have poor model performance. Another potential scenario is having a high training score, and a low test score. This would be a sign of overfitting. Regularization solves this overfitting issue, by putting constraints on the coefficients and variables. This will cause the coefficients of our data to be smaller. In the case of LASSO regression, some coefficients can actually be dropped (i.e., become 0), whereas in Ridge regression, coefficients will approach 0, but are never dropped.

15.3 LASSO

The first type of regularization technique is called LASSO (Least Absolute Shrinkage and Selection Operator). This is also known as regression with L1 regularization. We will fit the same model as we did in our linear regression.

```
from sklearn.linear_model import Lasso
lasso = Lasso(normalize=True, random_state=0).\
    fit(X_test, y_test)
```

Now, let's get a dataframe of coefficients, and combine them with our linear regression results.

```
coefs_lasso = pd.DataFrame(  
    list(zip(predictors.design_info.column_names, lasso.coef_)),  
    columns=['variable', 'coef_lasso'])  
  
model_coefs = pd.merge(model_coefs, coefs_lasso, on='variable')  
print(model_coefs)
```

	variable	coef_lr	coef_lasso
0	Intercept	3.522660e-11	0.000000
1	NumUnits[T.Single attached]	3.135646e+04	23847.097905
2	NumUnits[T.Single detached]	2.418368e+04	20278.620009
3	OwnRent[T.Outright]	2.839186e+04	30153.611697
4	OwnRent[T.Rented]	7.229586e+03	1440.140884
5	YearBuilt[T.1940-1949]	1.292169e+04	-6382.312453
6	YearBuilt[T.1950-1959]	2.057793e+04	-905.142030
7	YearBuilt[T.1960-1969]	1.764835e+04	-0.000000
8	YearBuilt[T.1970-1979]	1.756881e+04	-1579.827129
9	YearBuilt[T.1980-1989]	2.552566e+04	7854.066748
10	YearBuilt[T.1990-1999]	2.983944e+04	1355.026160
11	YearBuilt[T.2000-2004]	3.012502e+04	11212.207583
12	YearBuilt[T.2005]	4.318648e+04	8770.315635
13	YearBuilt[T.2006]	3.242038e+04	34814.310436
14	YearBuilt[T.2007]	3.562061e+04	27415.800873
15	YearBuilt[T.2008]	3.712470e+04	10866.123988
16	YearBuilt[T.2009]	3.035133e+04	312.110532
17	YearBuilt[T.2010]	7.364529e+04	10093.244533
18	YearBuilt[T.Before 1939]	1.218711e+04	-4903.325664
19	FoodStamp[T.Yes]	-2.745712e+04	-23717.406880
20	HeatingFuel[T.Electricity]	1.946552e+04	1775.625749
21	HeatingFuel[T.Gas]	2.588482e+04	12410.061671
22	HeatingFuel[T.None]	2.532452e+04	-4153.735420
23	HeatingFuel[T.Oil]	2.535803e+04	10009.595676
24	HeatingFuel[T.Other]	1.734533e+04	-6803.711978
25	HeatingFuel[T.Solar]	8.424991e+03	0.000000
26	HeatingFuel[T.Wood]	8.898002e+02	-9398.444417
27	Language[T.English]	-1.873624e+04	-8076.201004
28	Language[T.Other]	-4.463333e+03	-21403.661071
29	Language[T.Other European]	-1.409466e+04	-9113.511553
30	Language[T.Spanish]	-2.603347e+04	-14321.350716
31	NumBedrooms	3.443931e+03	3976.075383
32	NumChildren	8.215723e+03	5652.313652
33	NumPeople	-8.203826e+03	-5903.547002
34	NumRooms	5.735494e+03	4612.117329
35	NumVehicles	7.484535e+03	7736.529456
36	NumWorkers	2.283630e+04	20346.201513
37	ElectricBill	9.332524e+01	89.504660
38	Insurance	3.099441e+01	31.954902

The main thing to see here is that the coefficients are now smaller than their original linear regression values. Additionally, some of the coefficients are now 0.

Finally let's look at our training and test data scores.

```
print(lasso.score(X_train, y_train))  
0.266701046594  
print(lasso.score(X_test, y_test))  
0.275062046386
```

There isn't much difference here. But you can now see that the test results are now better than the training results. That is, there is an improvement in prediction using new unseen data.

15.4 Ridge

Now let's look at another regularization technique, Ridge regression. This is also known as regression with L2 regularization.

Most of the code will be very similar to the previous methods. We will fit the model on our training data, and combine the results to our ongoing dataframe of results.

```

from sklearn.linear_model import Ridge
ridge = Ridge(normalize=True, random_state=0).\
    fit(X_train, y_train)

coefs_ridge = pd.DataFrame(
    list(zip(predictors.design_info.column_names, ridge.coef_[0])),
    columns=['variable', 'coef_ridge'])

model_coefs = pd.merge(model_coefs, coefs_ridge, on='variable')
print(model_coefs)

      variable      coef_lr      coef_lasso \
0       Intercept  3.522660e-11   0.000000
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009
3       OwnRent[T.Outright]  2.839186e+04  30153.611697
4       OwnRent[T.Rented]  7.229586e+03  1440.140884
5  YearBuilt[T.1940-1949]  1.292169e+04  -6382.312453
6  YearBuilt[T.1950-1959]  2.057793e+04  -905.142030
7  YearBuilt[T.1960-1969]  1.764835e+04   -0.000000
8  YearBuilt[T.1970-1979]  1.756881e+04  -1579.827129
9  YearBuilt[T.1980-1989]  2.552566e+04  7854.066748
10  YearBuilt[T.1990-1999]  2.983944e+04  1355.026160
11  YearBuilt[T.2000-2004]  3.012502e+04  11212.207583
12       YearBuilt[T.2005]  4.318648e+04  8770.315635
13       YearBuilt[T.2006]  3.242038e+04  34814.310436
14       YearBuilt[T.2007]  3.562061e+04  27415.800873
15       YearBuilt[T.2008]  3.712470e+04  10866.123988
16       YearBuilt[T.2009]  3.035133e+04   312.110532
17       YearBuilt[T.2010]  7.364529e+04  10093.244533
18  YearBuilt[T.Before 1939]  1.218711e+04  -4903.325664
19       FoodStamp[T.Yes]  -2.745712e+04  -23717.406880
20  HeatingFuel[T.Electricity]  1.946552e+04  1775.625749
21       HeatingFuel[T.Gas]  2.588482e+04  12410.061671
22       HeatingFuel[T.None]  2.532452e+04  -4153.735420
23       HeatingFuel[T.Oil]  2.535803e+04  10009.595676
24       HeatingFuel[T.Other]  1.734533e+04  -6803.711978
25       HeatingFuel[T.Solar]  8.424991e+03   0.000000
26       HeatingFuel[T.Wood]  8.898002e+02  -9398.444417
27       Language[T.English]  -1.873624e+04  -8076.201004
28       Language[T.Other]  -4.463333e+03  -21403.661071
29  Language[T.Other European]  -1.409466e+04  -9113.511553
30       Language[T.Spanish]  -2.603347e+04  -14321.350716
31       NumBedrooms  3.443931e+03  3976.075383
32       NumChildren  8.215723e+03  5652.313652
33       NumPeople  -8.203826e+03  -5903.547002
34       NumRooms  5.735494e+03  4612.117329
35       NumVehicles  7.484535e+03  7736.529456
36       NumWorkers  2.283630e+04  20346.201513
37       ElectricBill  9.332524e+01   89.504660
38       Insurance  3.099441e+01   31.954902

      coef_ridge
0      0.000000
1     4571.129321
2     4514.956813
3    10674.890982
4   -10180.631863
5   -3672.096659
6    1221.616020
7   -15.801437
8   -1868.746915
9    2664.343363
10   4079.639281
11   5615.285677
12  12607.557029
13   5783.401233
14   8019.076178
15   7964.342869
16   3892.605415
17  28469.966885
18  -4271.925584

```

```

19 -21854.708263
20 -2043.214963
21 2043.550077
22 1376.185561
23 2377.402169
24 -5135.068670
25 589.799008
26 -13652.201413
27 -3003.249668
28 9067.969977
29 3059.003880
30 -6155.075714
31 4690.469564
32 1102.877585
33 -203.132130
34 3489.196546
35 5245.929228
36 10344.202715
37 68.784409
38 15.914804

```

15.5 Elastic Net

The elastic net is a regularization technique that combines the Ridge and LASSO techniques.

```

from sklearn.linear_model import ElasticNet
en = ElasticNet(random_state=42).fit(X_train, y_train)
coefs_en = pd.DataFrame(
    list(zip(predictors.design_info.column_names, en.coef_)),
    columns=['variable', 'coef_en'])
model_coefs = pd.merge(model_coefs, coefs_en, on='variable')
print(model_coefs)

          variable      coef_lr     coef_lasso \
0           Intercept  3.522660e-11  0.000000
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009
3           OwnRent[T.Outright]  2.839186e+04  30153.611697
4           OwnRent[T.Rented]   7.229586e+03  1440.140884
5  YearBuilt[T.1940-1949]   1.292169e+04  -6382.312453
6  YearBuilt[T.1950-1959]   2.057793e+04  -905.142030
7  YearBuilt[T.1960-1969]   1.764835e+04      -0.000000
8  YearBuilt[T.1970-1979]   1.756881e+04  -1579.827129
9  YearBuilt[T.1980-1989]   2.552566e+04  7854.066748
10  YearBuilt[T.1990-1999]   2.983944e+04  1355.026160
11  YearBuilt[T.2000-2004]   3.012502e+04  11212.207583
12           YearBuilt[T.2005]   4.318648e+04  8770.315635
13           YearBuilt[T.2006]   3.242038e+04  34814.310436
14           YearBuilt[T.2007]   3.562061e+04  27415.800873
15           YearBuilt[T.2008]   3.712470e+04  10866.123988
16           YearBuilt[T.2009]   3.035133e+04  312.110532
17           YearBuilt[T.2010]   7.364529e+04  10093.244533
18  YearBuilt[T.Before 1939]   1.218711e+04  -4903.325664
19           FoodStamp[T.Yes]  -2.745712e+04  -23717.406880
20  HeatingFuel[T.Electricity]  1.946552e+04  1775.625749
21           HeatingFuel[T.Gas]  2.588482e+04  12410.061671
22           HeatingFuel[T.None]  2.532452e+04  -4153.735420
23           HeatingFuel[T.Oil]   2.535803e+04  10009.595676
24           HeatingFuel[T.Other]  1.734533e+04  -6803.711978
25           HeatingFuel[T.Solar]  8.424991e+03      0.000000
26           HeatingFuel[T.Wood]  8.898002e+02  -9398.444417
27           Language[T.English] -1.873624e+04  -8076.201004
28           Language[T.Other]  -4.463333e+03  -21403.661071
29  Language[T.Other European] -1.409466e+04  -9113.511553
30           Language[T.Spanish] -2.603347e+04  -14321.350716
31           NumBedrooms   3.443931e+03  3976.075383
32           NumChildren    8.215723e+03  5652.313652
33           NumPeople     -8.203826e+03  -5903.547002
34           NumRooms      5.735494e+03  4612.117329
35           NumVehicles   7.484535e+03  7736.529456

```

```

36          NumWorkers  2.283630e+04  20346.201513
37          ElectricBill 9.332524e+01   89.504660
38          Insurance  3.099441e+01   31.954902

      coef_ridge      coef_en
0       0.000000     0.000000
1      4571.129321  1342.291706
2      4514.956813  168.728479
3     10674.890982  445.533238
4    -10180.631863 -600.673747
5     -3672.096659 -794.239494
6     1221.616020  513.289101
7     -15.801437 -275.576200
8     -1868.746915 -574.365605
9     2664.343363  708.813588
10    4079.639281  1357.944466
11    5615.285677  798.576141
12   12607.557029  445.271666
13    5783.401233  202.040682
14    8019.076178  222.170314
15    7964.342869  153.161478
16    3892.605415  88.228204
17   28469.966885  233.189152
18   -4271.925584 -3053.705550
19  -21854.708263 -4394.455708
20   -2043.214963 -129.968032
21   2043.550077  1924.299033
22   1376.185561     0.000000
23   2377.402169  453.942244
24   -5135.068670  -67.445065
25    589.799008   0.994142
26 -13652.201413 -1894.123724
27  -3003.249668  -955.455328
28   9067.969977  374.835549
29   3059.003880  626.547311
30  -6155.075714 -1367.763935
31   4690.469564  2073.910045
32   1102.877585  2498.719581
33  -203.132130 -2562.412933
34   3489.196546  5685.101939
35   5245.929228  6059.776166
36  10344.202715 12247.547800
37   68.784409   97.566664
38   15.914804   32.484207

```

The `ElasticNet` object has 2 parameters, `alpha` and `l1_ratio` that allow you to control the behavior of the model. The `l1_ratio` parameter specifically controls how much of the L2 or L1 penalty is used. If `l1_ratio = 0`, then the model will behave like ridge regression. If `l1_ratio = 1`, then the model will behave like LASSO regression. Any value in between will give some combination of ridge and LASSO.

15.6 Cross Validation

Cross validation ([Section 14.4](#)) is a common technique when fitting models. It has been mentioned at the beginning of this chapter, as a segue to regularization, and it is also a way to pick optimal parameters for regularization. Since there are parameters that the user must tune (aka as hyper-parameters), cross-validation can be used to try out various combinations of these hyper-parameters to pick the “best” model. The `ElasticNet` has a similar function called `ElasticNetCV`¹ that can iteratively fit the Elastic Net with various hyper-parameter values.

¹ElasticNetCV documentation:

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNetCV.html#sklearn.linear_model.ElasticNetCV

```
from sklearn.linear_model import ElasticNetCV
```

```

en_cv = ElasticNetCV(cv=5, random_state=42).fit(X_train, y_train)

coefs_en_cv = pd.DataFrame(
    list(zip(predictors.design_info.column_names, en_cv.coef_)),
    columns=['variable', 'coef_en_cv'])

model_coefs = pd.merge(model_coefs, coefs_en_cv, on='variable')
print(model_coefs)

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/sklearn/linear_model/coordinate_descent.py:1094:
DataConversionWarning: A column-vector y was passed when a 1d array
was expected. Please change the shape of y to (n_samples, ), for
example using ravel().
    y = column_or_1d(y, warn=True)
        variable      coef_lr      coef_lasso \
0           Intercept  3.522660e-11   0.000000
1  NumUnits[T.Single attached]  3.135646e+04  23847.097905
2  NumUnits[T.Single detached]  2.418368e+04  20278.620009
3           OwnRent[T.Outright]  2.839186e+04  30153.611697
4           OwnRent[T.Rented]   7.229586e+03  1440.140884
5  YearBuilt[T.1940-1949]   1.292169e+04  -6382.312453
6  YearBuilt[T.1950-1959]   2.057793e+04  -905.142030
7  YearBuilt[T.1960-1969]   1.764835e+04  -0.000000
8  YearBuilt[T.1970-1979]   1.756881e+04  -1579.827129
9  YearBuilt[T.1980-1989]   2.552566e+04  7854.066748
10  YearBuilt[T.1990-1999]   2.983944e+04  1355.026160
11  YearBuilt[T.2000-2004]   3.012502e+04  11212.207583
12  YearBuilt[T.2005]       4.318648e+04  8770.315635
13  YearBuilt[T.2006]       3.242038e+04  34814.310436
14  YearBuilt[T.2007]       3.562061e+04  27415.800873
15  YearBuilt[T.2008]       3.712470e+04  10866.123988
16  YearBuilt[T.2009]       3.035133e+04  312.110532
17  YearBuilt[T.2010]       7.364529e+04  10093.244533
18  YearBuilt[T.Before 1939] 1.218711e+04  -4903.325664
19  FoodStamp[T.Yes]      -2.745712e+04  -23717.406880
20  HeatingFuel[T.Electricity]  1.946552e+04  1775.625749
21  HeatingFuel[T.Gas]      2.588482e+04  12410.061671
22  HeatingFuel[T.None]    2.532452e+04  -4153.735420
23  HeatingFuel[T.Oil]     2.535803e+04  10009.595676
24  HeatingFuel[T.Other]   1.734533e+04  -6803.711978
25  HeatingFuel[T.Solar]   8.424991e+03   0.000000
26  HeatingFuel[T.Wood]   8.898002e+02  -9398.444417
27  Language[T.English]   -1.873624e+04  -8076.201004
28  Language[T.Other]    -4.463333e+03  -21403.661071
29  Language[T.Other European] -1.409466e+04  -9113.511553
30  Language[T.Spanish]   -2.603347e+04  -14321.350716
31  NumBedrooms          3.443931e+03  3976.075383
32  NumChildren          8.215723e+03  5652.313652
33  NumPeople            -8.203826e+03  -5903.547002
34  NumRooms             5.735494e+03  4612.117329
35  NumVehicles          7.484535e+03  7736.529456
36  NumWorkers            2.283630e+04  20346.201513
37  ElectricBill         9.332524e+01   89.504660
38  Insurance            3.099441e+01   31.954902

      coef_ridge      coef_en      coef_en_cv
0      0.000000  0.000000  0.000000
1     4571.129321 1342.291706  -0.000000
2     4514.956813 168.728479  0.000000
3     10674.890982 445.533238  0.000000
4    -10180.631863 -600.673747  -0.000000
5     -3672.096659 -794.239494  -0.000000
6     1221.616020  513.289101  0.000000
7    -15.801437 -275.576200  0.000000
8    -1868.746915 -574.365605  -0.000000
9     2664.343363  708.813588  0.000000
10    4079.639281 1357.944466  0.000000
11    5615.285677  798.576141  0.000000
12   12607.557029  445.271666  0.000000
13    5783.401233  202.040682  0.000000
14    8019.076178  222.170314  0.000000
15    7964.342869  153.161478  0.000000
16    3892.605415  88.228204  0.000000
17   28469.966885 233.189152  0.000000

```

18	-4271.925584	-3053.705550	-0.000000
19	-21854.708263	-4394.455708	-0.000000
20	-2043.214963	-129.968032	-0.000000
21	2043.550077	1924.299033	0.000000
22	1376.185561	0.000000	-0.000000
23	2377.402169	453.942244	0.000000
24	-5135.068670	-67.445065	-0.000000
25	589.799008	0.994142	-0.000000
26	-13652.201413	-1894.123724	-0.000000
27	-3003.249668	-955.455328	-0.000000
28	9067.969977	374.835549	0.000000
29	3059.003880	626.547311	0.000000
30	-6155.075714	-1367.763935	-0.000000
31	4690.469564	2073.910045	0.000000
32	1102.877585	2498.719581	0.000000
33	-203.132130	-2562.412933	0.000000
34	3489.196546	5685.101939	0.028443
35	5245.929228	6059.776166	0.000000
36	10344.202715	12247.547800	0.000000
37	68.784409	97.566664	26.166320
38	15.914804	32.484207	38.561748

15.7 Conclusion

Regularization is a technique used to prevent over-fitting of data. It does this by applying some penalty for each feature added to the model. The end result is either dropping variables from the model, or shrinking the coefficients of the model. Both techniques try to fit the training data less accurately but hope to better predict on data that was not seen before. These techniques can be combined (as seen in the elastic net), and can also be iterated over and improved with cross-validation.

Chapter 16. Clustering

16.1 Introduction

Machine learning methods can generally be broken up into two main categories of models, supervised learning and unsupervised learning. Thus far, we have been working on supervised learning models since we train our models with a target y or response variable. In other words, in the training data for our models, we know the “correct” answer. Unsupervised models are techniques where the “correct” answer is unknown. A large part of these methods involve clustering, where the two main methods are k-means clustering and hierarchical clustering.

16.2 K-means

K-means works by first selecting how many clusters, k , exist in the data. The algorithm will randomly select k points in the data and calculate the distance from every data point to the initially selected k points. The closest points to each of the k clusters will be assigned to the same cluster group. The center of each cluster is then designated as the new cluster centroid. The process then repeats, where the distance of each point is calculated to each cluster centroid and assigned to a cluster and a new centroid is picked. This algorithm repeats until the convergence.

There are great visualizations¹ and explanations² on the internet for how k-means works

¹Visualizing k-means: <http://shabalin/visuals.html>

²Visualization and explanation of k-means: <https://www.naftaliharris.com/blog/visualizing-k-means-clustering/>

```
import pandas as pd
wine = pd.read_csv('../data/wine.csv')

# note that the data values are all numeric
print(wine.head())

      Cultivar    Alcohol   Malic acid     Ash Alkalinity of ash \
0            1       14.23      1.71    2.43             15.6
1            1       13.20      1.78    2.14             11.2
2            1       13.16      2.36    2.67             18.6
3            1       14.37      1.95    2.50             16.8
4            1       13.24      2.59    2.87             21.0

      Magnesium   Total phenols   Flavanoids Nonflavanoid phenols \
0          127           2.80        3.06            0.28
1          100           2.65        2.76            0.26
2          101           2.80        3.24            0.30
3          113           3.85        3.49            0.24
4          118           2.80        2.69            0.39

Proanthocyanins   Color intensity     Hue \
0            2.29         5.64      1.04
1            1.28         4.38      1.05
2            2.81         5.68      1.03
3            2.18         7.80      0.86
4            1.82         4.32      1.04

OD280/OD315 of diluted wines   Proline \
0            3.92        1065
1            3.40        1050
2            3.17        1185
```

3	3.45	1480
4	2.93	735

We will drop the `Cultivar` column since it correlates too much with the actual clusters in our data.

```
wine = wine.drop('Cultivar', axis=1)
print(wine.head())

   Alcohol    Malic acid    Ash Alkalinity of ash Magnesium \
0     14.23      1.71  2.43           15.6        127
1     13.20      1.78  2.14           11.2        100
2     13.16      2.36  2.67           18.6        101
3     14.37      1.95  2.50           16.8        113
4     13.24      2.59  2.87           21.0        118

   Total phenols    Flavanoids Nonflavanoid phenols Proanthocyanins \
0            2.80       3.06          0.28             2.29
1            2.65       2.76          0.26             1.28
2            2.80       3.24          0.30             2.81
3            3.85       3.49          0.24             2.18
4            2.80       2.69          0.39             1.82

   Color intensity    Hue OD280/OD315 of diluted wines \
0              5.64   1.04           3.92
1              4.38   1.05           3.40
2              5.68   1.03           3.17
3              7.80   0.86           3.45
4              4.32   1.04           2.93

   Proline
0            1065
1            1050
2            1185
3            1480
4            735
```

`sklearn` has an implementation of `KMeans`. Here we will set `k=3`, and use all the data in our dataset.

```
# create 3 clusters
# use a random seed of 42
# you can opt to leave out the random_state parameter
# or use a different value, the 42 will ensure your results
# are the same as the one printed in the book
kmeans = KMeans(n_clusters=3, random_state=42).fit(wine.values)
```

Here's our `kmeans` object.

```
print(kmeans)

KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
       random_state=42, tol=0.0001, verbose=0)
```

We can see that since we specified 3 clusters, there are only 3 unique labels.

```
import numpy as np
print(np.unique(kmeans.labels_, return_counts=True))

(array([0, 1, 2], dtype=int32), array([69, 47, 62]))
```

We can turn these labels into a dataframe so we can add it to our dataset

```
kmeans_3 = pd.DataFrame(kmeans.labels_, columns=['cluster'])
print(kmeans_3.head())

   cluster
0         1
1         1
2         1
3         1
4         2
```

Finally, we can visualize our clusters. Since humans are only able to visualize things in 3 dimensions we need to reduce the number of dimensions of our data. Our `wine` dataset has 13 columns, and we need to reduce it down to 3 for us to even understand what is going on. Furthermore, since we are trying to plot the points in a book (a non-interactive medium) we should reduce the number of dimensions down to 2, if possible.

16.2.1 Dimension Reduction with PCA

Principal component analysis (PCA) is a projection technique that is used to reduce the number of dimensions of your data. It works by finding a lower dimension in the data such that the variance is maximized. Imagine a 3-dimensional sphere of points. What PCA is doing is shining a light through these points and casting a shadow in the lower 2-dimensional plane. PCA tries to shine this light such that the shadows are spread out as much as possible. It's okay to interpret points that are far apart in PCA without cause for concern, however, points that are far apart in the original 3D sphere can have the light shine through them in such a way that the shadows cast are right next to one another. So, be careful when trying to interpret points that are close to one another. It is possible that these points could not be farther apart in the original space.

We import `PCA` from `sklearn`

```
from sklearn.decomposition import PCA
```

We tell PCA how many dimensions (aka principal components) we want to project our data into. Here we are projecting our data down into 2 components.

```
# project our data into 2 components
pca = PCA(n_components=2).fit(wine)
```

Next, we need to transform our data into the new space and add it to our dataset.

```
# transform our data into the new space
pca_trans = pca.transform(wine)

# give our projections a name
pca_trans_df = pd.DataFrame(pca_trans, columns=['pca1', 'pca2'])

# concatenate our data
kmeans_3 = pd.concat([kmeans_3, pca_trans_df], axis=1)

print(kmeans_3.head())

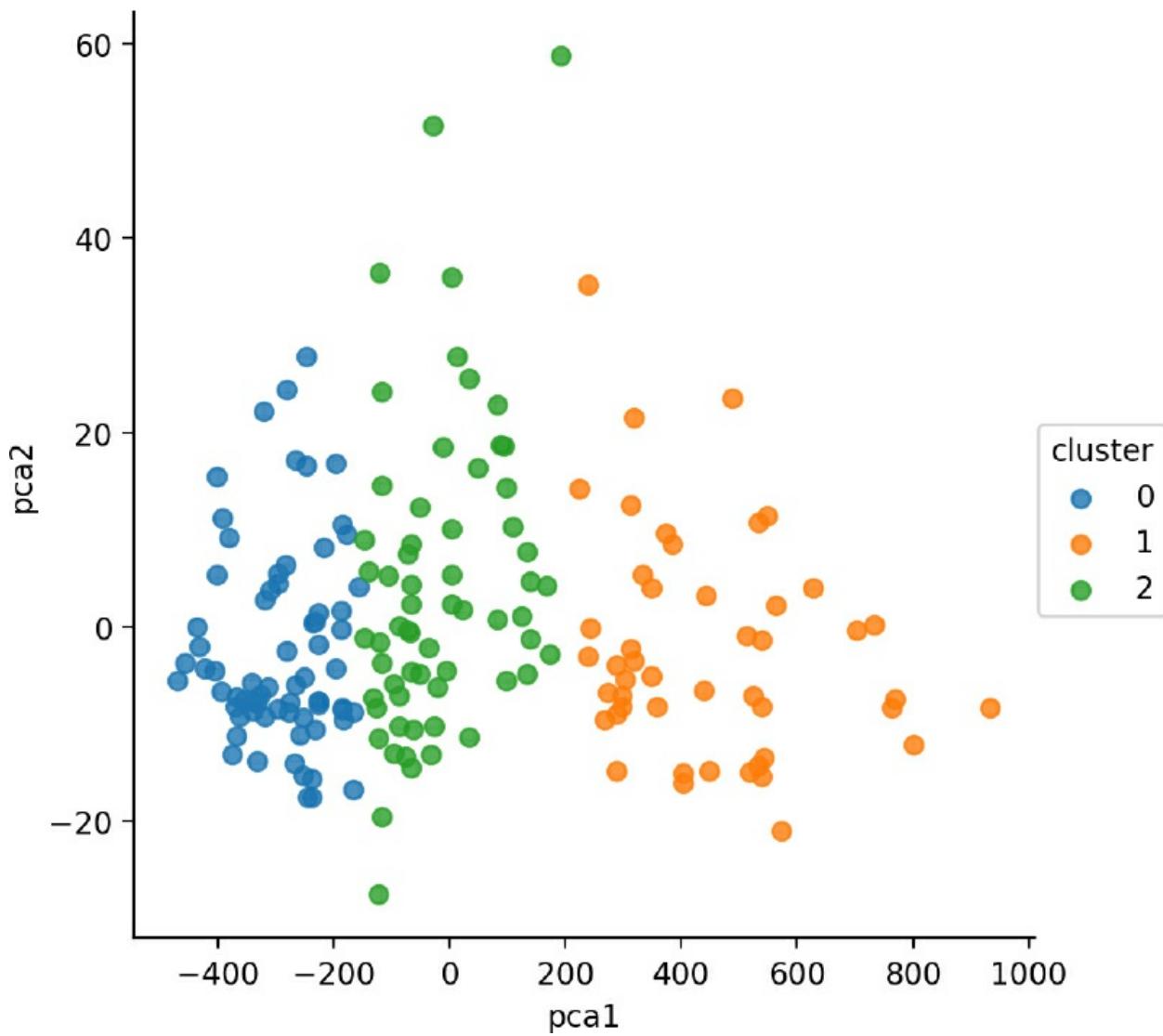
   cluster      pca1      pca2
0         1  318.562979  21.492131
1         1  303.097420 -5.364718
2         1  438.061133 -6.537309
3         1  733.240139  0.192729
4         2 -11.571428  18.489995
```

Finally, we can plot our results ([Figure 16-1](#)).

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

fig = sns.lmplot(x = 'pca1', y='pca2', data=kmeans_3,
                  hue='cluster', fit_reg=False)
plt.show()
```

Figure 16-1: Kmeans plot using PCA



Now that we've seen what k-means does to our wine data, let's load back the original dataset and keep the `cultivar` column we dropped.

```
wine_all = pd.read_csv('../data/wine.csv')
print(wine_all.head())
```

	Cultivar	Alcohol	Malic acid	Ash	Alkalinity of ash	\
0	1	14.23	1.71	2.43	15.6	
1	1	13.20	1.78	2.14	11.2	
2	1	13.16	2.36	2.67	18.6	
3	1	14.37	1.95	2.50	16.8	
4	1	13.24	2.59	2.87	21.0	
	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols		\
0	127	2.80	3.06	0.28		
1	100	2.65	2.76	0.26		
2	101	2.80	3.24	0.30		
3	113	3.85	3.49	0.24		
4	118	2.80	2.69	0.39		
	Proanthocyanins	Color intensity	Hue	\		
0	2.29	5.64	1.04			
1	1.28	4.38	1.05			
2	2.81	5.68	1.03			
3	2.18	7.80	0.86			
4	1.82	4.32	1.04			
	OD280/OD315 of diluted wines		Proline			

0	3.92	1065
1	3.40	1050
2	3.17	1185
3	3.45	1480
4	2.93	735

We'll run PCA on our data, just like before, and compare the clusters from PCA and the variables from `cultivar`.

```
pca_all = PCA(n_components=2).fit(wine_all)
pca_all_trans = pca_all.transform(wine_all)
pca_all_trans_df = pd.DataFrame(pca_all_trans,
                                 columns=['pca_all_1', 'pca_all_2'])

kmeans_3 = pd.concat([kmeans_3,
                      pca_all_trans_df,
                      wine_all['Cultivar']], axis=1)
```

We can compare the groupings by faceting our plot ([Figure 16-2](#)).

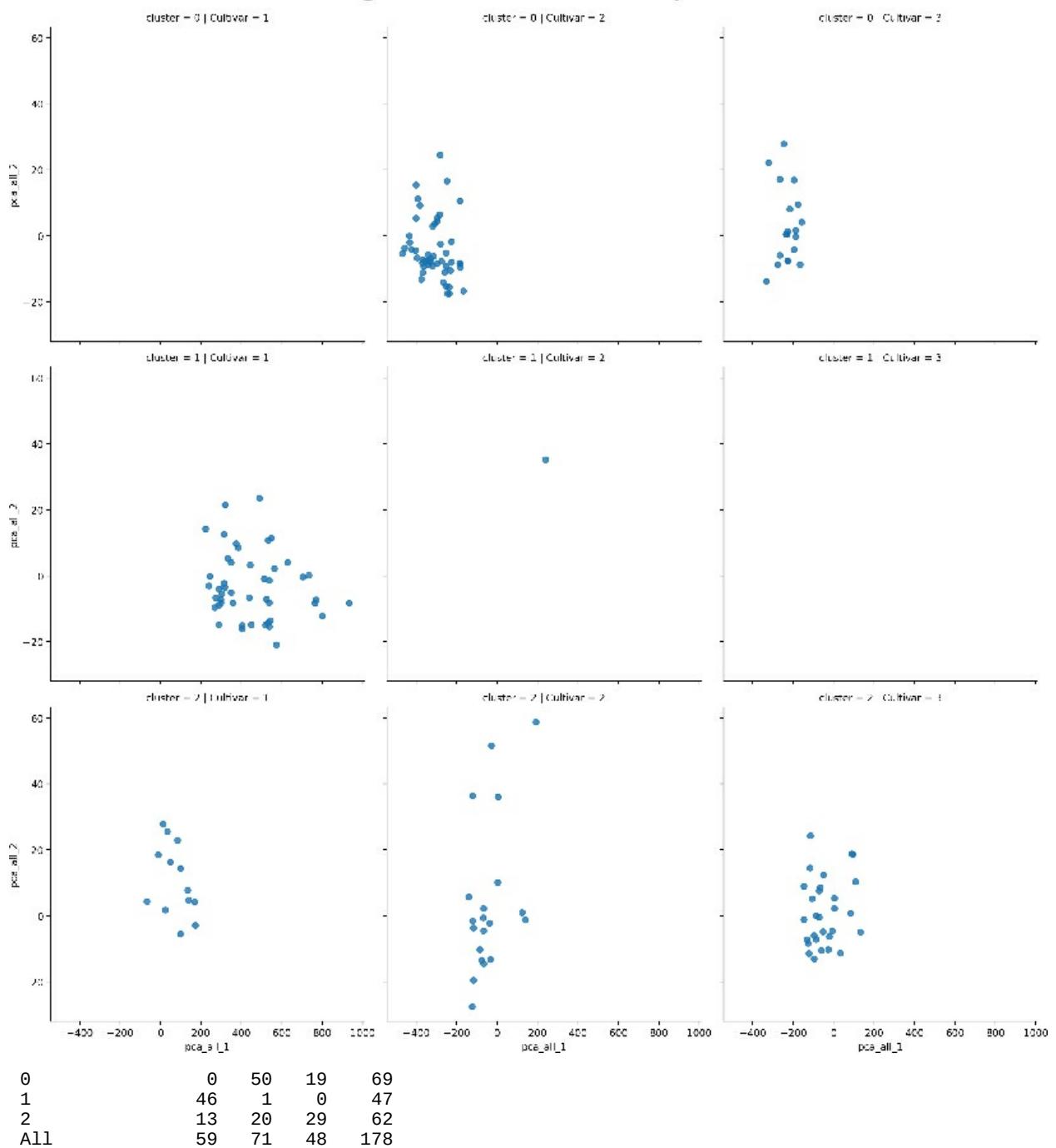
```
with sns.plotting_context(font_scale=5):
    fig = sns.lmplot(x = 'pca_all_1',
                      y='pca_all_2',
                      data=kmeans_3,
                      row='cluster', col='Cultivar',
                      fit_reg=False)
plt.show()
```

Or we can look at a cross tabulated frequency count.

```
print(pd.crosstab(kmeans_3['cluster'],
                   kmeans_3['Cultivar'],
                   margins=True))

Cultivar      1    2    3   All
cluster
```

Figure 16-2: Faceted kmeans plot



16.3 Hierarchical Clustering

As the name suggests, hierarchical clustering aims to build a hierarchy of clusters. It can accomplish this with a bottom-up (agglomerative) or top-down (divisive) approach.

We can perform this type of clustering with the `scipy` library.

```
from scipy.cluster import hierarchy
```

We'll load up a clean wine dataset again, and drop the `cultivar` column.

```
wine = pd.read_csv('../data/wine.csv')
```

```
wine = wine.drop('Cultivar', axis=1)
```

There are many ways the hierarchical clustering algorithm works. We can use `matplotlib` to plot the results

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

16.3.1 Complete

A hierarchical cluster using the complete clustering algorithm is shown in [Figure 16-3](#).

```
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_complete)
plt.show()
```

16.3.2 Single

A hierarchical cluster using the single clustering algorithm is shown in [Figure 16-4](#).

```
wine_single = hierarchy.single(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_single)
plt.show()
```

16.3.3 Average

A hierarchical cluster using the average clustering algorithm is shown in [Figure 16-5](#).

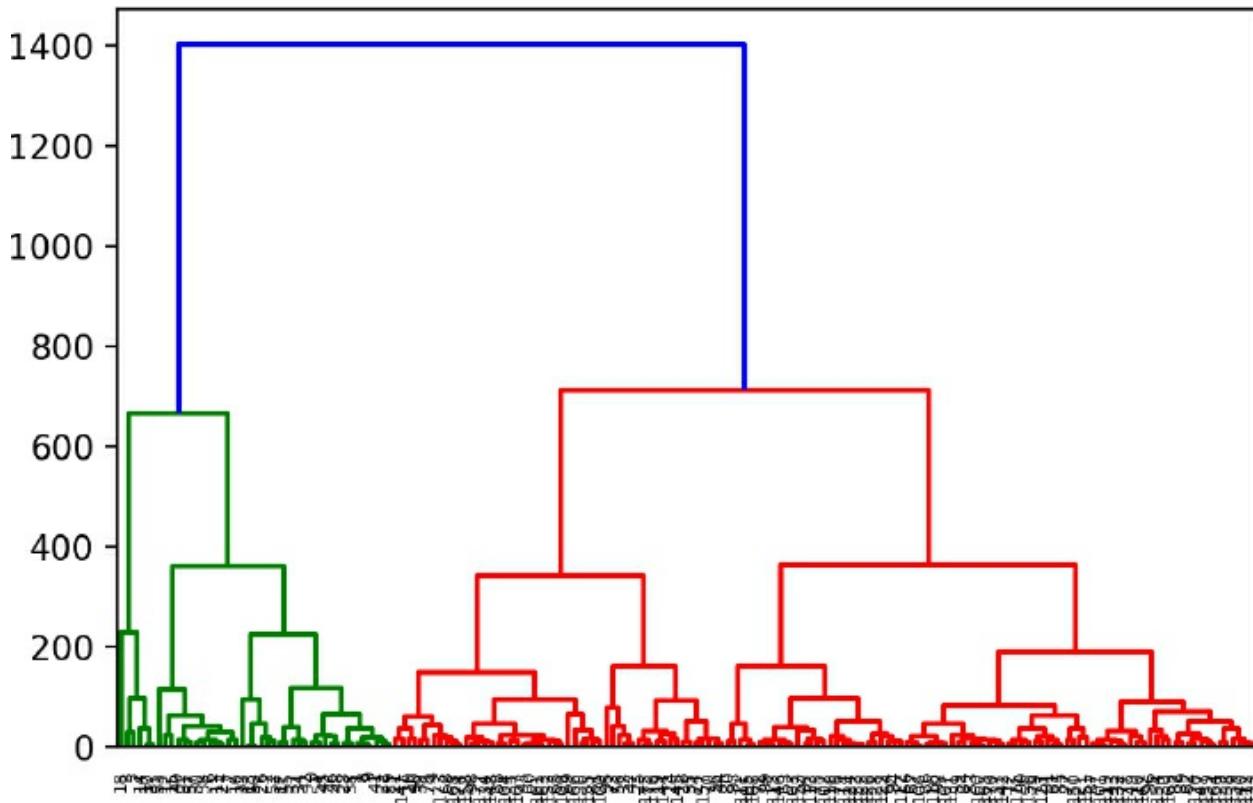
```
wine_average = hierarchy.average(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_average)
plt.show()
```

16.3.4 Centroid

A hierarchical cluster using the centroid clustering algorithm is shown in [Figure 16-6](#).

```
wine_centroid = hierarchy.centroid(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(wine_centroid)
plt.show()
```

Figure 16-3: Hierarchical clustering: complete



16.3.5 Manually setting threshold

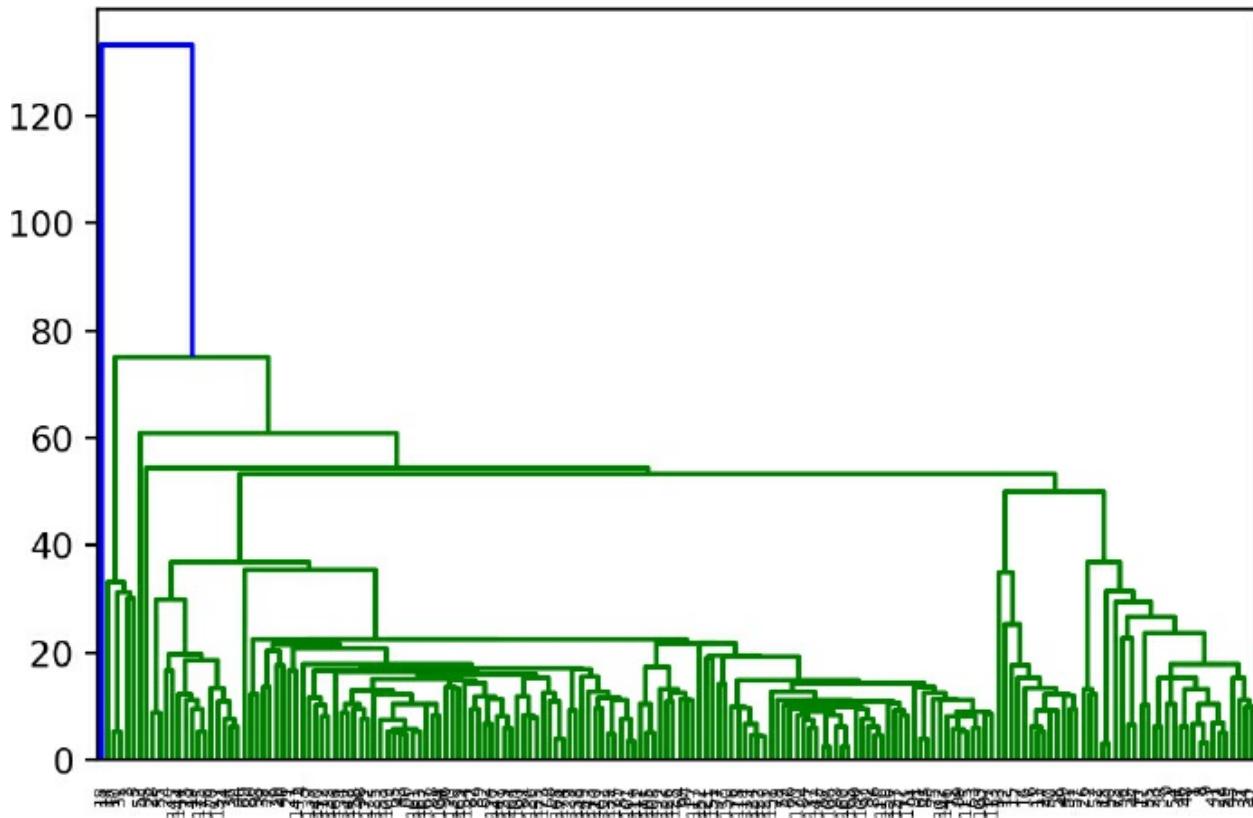
We can pass in a value for `color_threshold` to color the groups by a threshold ([Figure 16-7](#)). By default `scipy` uses the default MATLAB values.

```
wine_complete = hierarchy.complete(wine)
fig = plt.figure()
dn = hierarchy.dendrogram(
    wine_complete,
    # default MATLAB threshold
    color_threshold=0.7 * max(wine_complete[:,2]),
    above_threshold_color='y')
plt.show()
```

16.4 Conclusion

When trying to find underlying structure in a dataset, unsupervised machine learning methods are often used. K-means and hierarchical clustering are two common methods used to solve this problem. The key is to tune your models either by specifying a k in k-means or a threshold value in hierarchical clustering that makes sense for the question you are trying to solve.

Figure 16-4: Hierarchical clustering: single



It is also common to mix multiple types of analysis techniques to solve a problem. For example, it is common to use a unsupervised learning method to cluster your data and use these clusters as features in another analysis method.

Figure 16-5: Hierarchical clustering: average

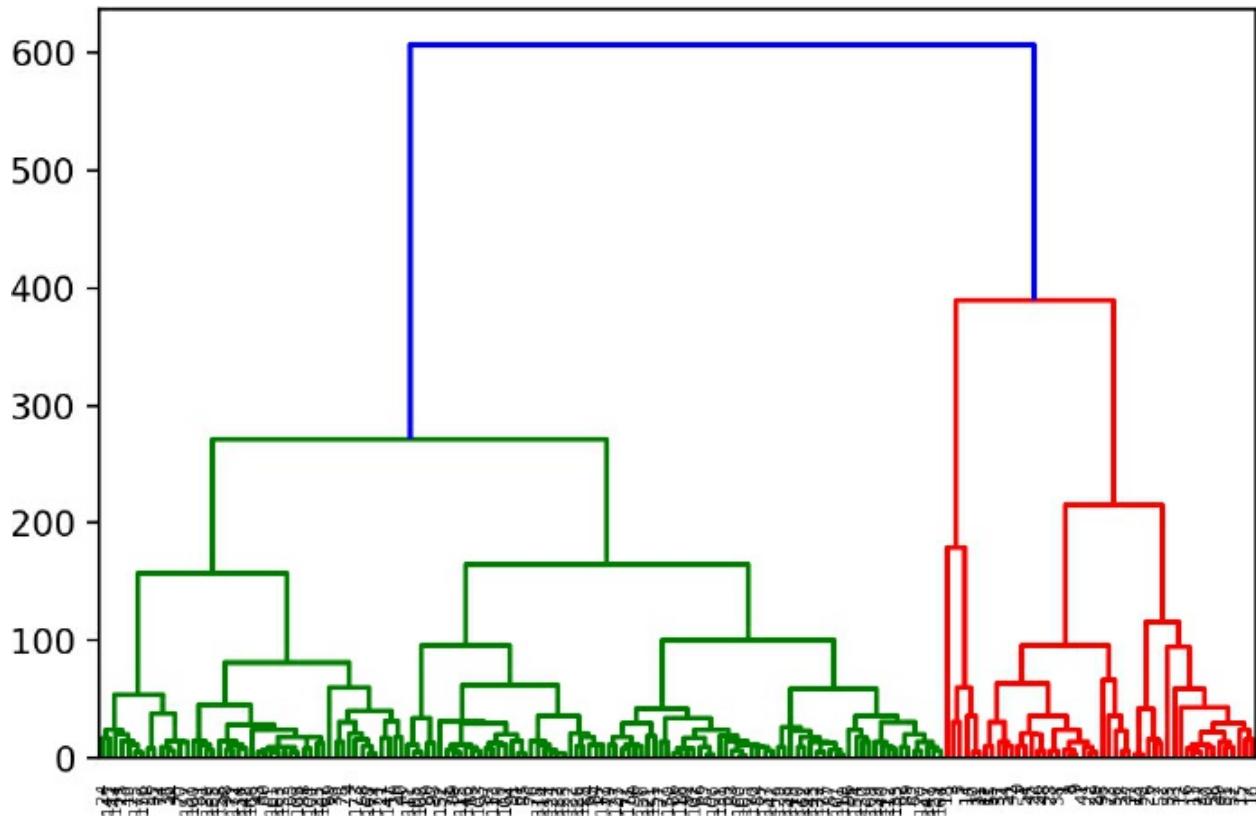


Figure 16-6: Hierarchical clustering: centroid

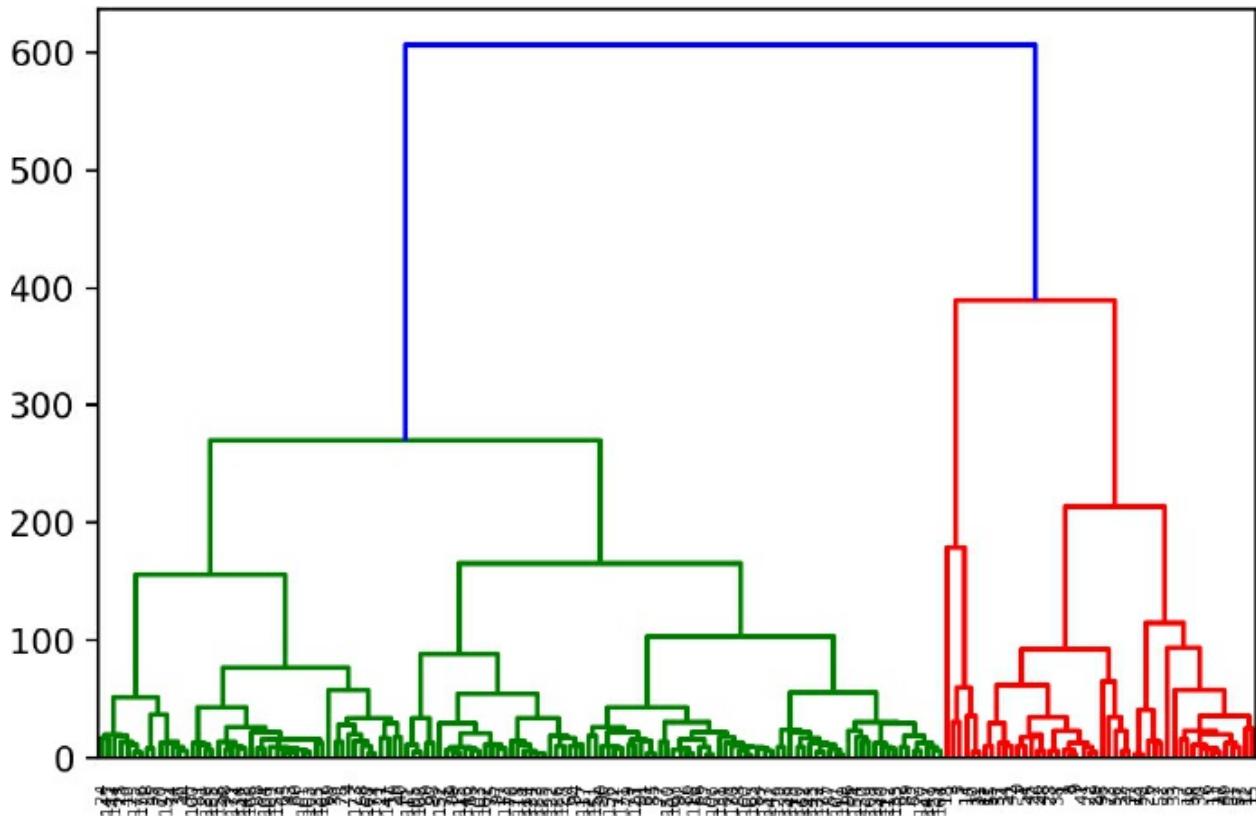
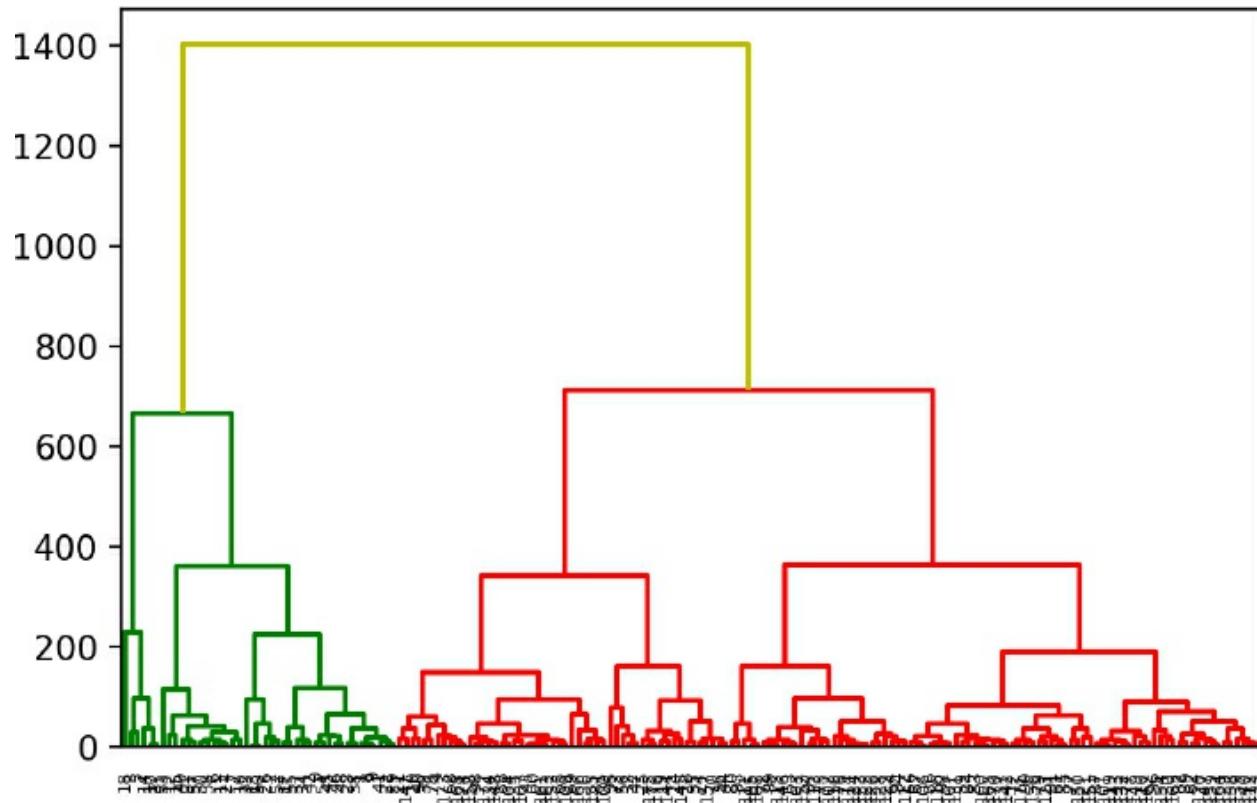


Figure 16-7: Manual hierarchical clustering threshold



Part V: Conclusion

[Chapter 17](#), “[Life Outside of Pandas](#),” The packages and ecosystem surrounding pandas.

[Chapter 18](#), “[Towards a Self-Directed Learner](#),” Additonal learning resources.

Chapter 17. Life Outside of Pandas

17.1 The (Scientific) Computing Stack

Jake VanderPlas¹ gave a Scipy² 2015 keynote³ On “The State of the Stack”. In there he talks about how community of packages that surround the core Python language developed. Python the language was created in the 1980s. Numerical computing began in 1995 and eventually evolved into the NumPy library in 2006. The NumPy library was the basis of the Pandas `series` we've worked with in the book. The core plotting library, Matplotlib, was created in 2002 which is also used within pandas in the `plot` method. The ability for pandas to work with heterogeneous data allows the analyst to clean different types of data to be used for analysis using the scikits, which stemmed from the Scipy package in 2000.

¹Jake VanderPlas: <https://staff.washington.edu/jakevdp/>

²Scipy Conference: <https://conference.scipy.org/>

³Jake's SciPy 2015 Keynote: <https://speakerdeck.com/jakevdp/the-state-of-the-stack-scipy-2015-keynote>

There have also been advances to how we interface with Python. IPython in 2001 was created to give more interactivity with the language and the Shell. Project Jupyter in 2012, created the interactive notebook for python, which further solidified the language as a scientific computing platform, as it gave a easy and highly extensible way to do literate programming and much more.

However, the ecosystem is more than these few libraries and tools. SymPy⁴ is a fully functional computer algebra system (CAS) in Python that can do symbolic manipulation of mathematical formulas and equations. While pandas is great for rectangular flat files and has support for hierarchical indices, the xarray library⁵ gives python the ability to work with N-dimensional arrays. Think of pandas as a 2-dimensional dataframe, array, gives us an n-dimensional dataframe. These types of data are common within the scientific community. If you have to constantly work with various data input and output types, have a look into the odo library ([Appendix T](#)).

⁴SymPy: <http://www.sympy.org/en/index.html>

⁵xarray: <http://xarray.pydata.org/en/stable/>

17.2 Performance

“Premature optimization is the root of all evil”. Write your python code in a way that works first, and gives you a result that you can test. If it's not fast enough, then you can work on optimizing the code. The SciPy ecosystem has libraries that make python faster: cython and numba.

17.2.1 Timing your Code

IPython also comes with “magic commands”⁶ that give even more features to the language. One

of which is the `timeit` magic that times the execution of a python statement or expression. You can use this to benchmark your code to see what aspects are slow. Let's use the examples from [Section 9.5](#).

⁶IPython build-in magic commands:
<http://ipython.readthedocs.io/en/stable/interactive/magics.html>

We begin with applying a function with `axis=1`

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})

def avg_2_apply(row):
    x = row[0]
    y = row[1]
    if (x == 20):
        return np.nan
    else:
        return (x + y) / 2
```

Then we can vectorize our function using numpy.

```
%%timeit
df.apply(avg_2_apply, axis=1)

475 s      7.37 s per loop (mean      std. dev. of 7 runs, 1000 loops
each)

@np.vectorize
def v_avg_2_mod(x, y):
    if (x == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

%%timeit
v_avg_2_mod(df['a'], df['b'])

91.5 s      2.73 s per loop (mean      std. dev. of 7 runs, 10000 loops
each)
```

Finally, we can time our calculations using numba.

```
import numba

@numba.vectorize
def v_avg_2_numba(x, y):
    if (int(x) == 20):
        return(np.NaN)
    else:
        return (x + y) / 2

%%timeit
v_avg_2_numba(df['a'].values, df['b'].values)

10.9 s      70.5 ns per loop (mean std. dev. of 7 runs, 100000 loops
each)
```

You can see how much faster by looking at the amount of time per loop each method takes. Numba is clearly the fastest method *in this example*.

17.2.2 Profiling Your Code

There are other tools such as cProfile⁷ and snakeviz⁸ that can help time entire scripts and blocks of code and give a line-by-line breakdown. Additionally, the snakeviz library comes with an IPython `snakeviz` extension!

⁷cProfile: <https://docs.python.org/3.4/library/profile.html#module-cProfile>

⁸Snakeviz: <https://jiffyclub.github.io/snakeviz/>

17.3 Going Bigger and Faster

There are many different libraries and frameworks to help scale up your computation.

`concurrent.futures`⁹ allows you to essentially re-write the function calls into the built-in `map` function.¹⁰ Dask¹¹ is another library that is geared towards working with large datasets. It allows you to create a computational graph, so only calculations that are “out of date” need to be recalculated. Dask also can help parallelize calculations on your own (single) machine or across multiple machines in a cluster. It creates a system where you can write code on your laptop, and quickly scale your code up to larger compute clusters. The nicest part of dask is that the syntax aims to mimic the syntax from pandas. Thus, lowering the overhead of learning the library.

There's a great set of notebooks¹² that go over these techniques.

⁹concurrent.futures : <https://docs.python.org/3/library/concurrent.futures.html>

¹⁰Python map: <https://docs.python.org/3.6/library/functions.html#map>

¹¹Dask: <https://dask.pydata.org/en/latest/>

¹²Parallel Python Tutorial: <https://github.com/pydata/parallel-tutorial>

Chapter 18. Towards a Self-Directed Learner

18.1 It's Dangerous to Go Alone!

Take this! One of the best ways to learn a language is to work on a problem with other people. You can either do this in the form of pair-programming where two people program together, one person does the typing and the other person talks though the code. This allows two sets of eyes to look at the code, improves communication between people, and gives a sense of ownership. All of these contribute to higher quality code, and makes programming fun, which means you're more likely to improve by doing it more.

18.2 Local Meetups

Many cities have a Meetup culture¹ where people can find a common hobby or topic and have a place to “meetup”. Python specific meetups exist, but it’s worth going to others that focus on data cleaning, visualization, or machine learning. Even meetups in other languages can be helpful. The more you expose yourself to the community and field, the more connections you can make with your own work.

¹Meetup: <https://www.meetup.com/>

If there isn’t a meetup in your city, create one! You can start with friends and people who are interested, and begin to host regular times to meet and talk. Keep it fun. Talk about topics of interest at a bar. Again, the more enjoyable something is, the more likely you will do it.

18.3 Conferences

Conferences are a great way to learn about the latest libraries and techniques. You also get to meet new people as well as library maintainers. Many conferences will also have a “sprint day” where people are encouraged to work on code and contribute to a library. This is a great way to learn about the library itself, improve your programming skills, and also contributing to the community.

Pycon² is the main python conference. There are topics across the entire Python ecosystem, such as Django³ and Flask⁴ for web development. The talks for these conferences are usually recorded and freely available.⁵ SciPy⁶ and EuroSciPy⁷ are the conferences that focus more on the scientific and analytics stack of Python. I personally have been attending SciPy the past few years, and the tutorials cover a vast set of topics. I’ve tried my best to compile a list of talks and the corresponding video or materials.⁸ You can always find the YouTube playlists for these conferences.^{9 10} AnacondaCon¹¹ is a newer conference that also has videos posted online.¹² Jupyter also hosts its own conferences. Jupyter Days and JupyterCon have videos¹³ and you can hear when the next conference is on the main Jupyter blog.¹⁴ Finally, PyData, the non-profit that supports many open-source projects also have conferences with videos.¹⁵

²Pycon: <https://us.pycon.org/2018/>

³Django: <https://www.djangoproject.com/>

⁴Flask: <http://flask.pocoo.org/>

⁵Python 2017 Talks: https://www.youtube.com/channel/UCrJhliKNQ8q0qoE_zvL8eVg

⁶Scipy Conference: <https://conference.scipy.org/>

⁷EuroSciPy Conference: <https://www.euroscipy.org/>

⁸SciPy 2017 Links and Videos: https://github.com/chendaniely/scipy_2017_notes

⁹SciPy 2017 Videos: <https://www.youtube.com/playlist?list=PLYx7XA2nY5GfdAFycPLBdUDOUTdQIVoMf>

¹⁰EuroSciPy 2017 Videos: https://www.youtube.com/watch?v=ToYFc_AcKU0&list=PLYHT2hHT8PFC03qijYx1HqEIpVov0tmXy

¹¹AnacondaCon 2018: <https://anacondacon18.io/>

¹²AnacondaCon Videos: <https://www.anaconda.com/videos/>

¹³JupyterCon: <https://www.youtube.com/playlist?list=PL055Epbe6d5aP6Ru42r7hk68GTSaclYqi>

¹⁴Jupyter Blog: <https://blog.jupyter.org/>

¹⁵PyData: <https://pydata.org/>

18.4 The Internet

The internet is a great place to find additional resources. DataCamp¹⁶ is a great place to get a deep dive into a particular topic. I've personally got my start from Software-Carpentry¹⁷ and Data Carpentry.¹⁸ The Yhat blog also have regular posts¹⁹ about python and data analytics.

¹⁶DataCamp: <https://www.datacamp.com/>

¹⁷Software-Carpentry Lessons: <https://software-carpentry.org/lessons/>

¹⁸Data Carpentry Lessons: <http://www.datacarpentry.org/lessons/>

¹⁹YHat blog: <http://blog.yhat.com/>

18.5 Podcasts

Data science related podcasts are plenty. Here are some that I listen to (in no particular order)

1. Not So Standard Deviations: <https://soundcloud.com/nssd-podcast>

2. Partially Derivative: <http://partiallyderivative.com/>

3. Linear Digressions: <http://lineardigressions.com/>
4. Data Skeptic: <https://dataskeptic.com/>
5. Becoming a Data Scientist: <https://www.becomingadatascientist.com/category/podcast/>
6. Talk Python to Me: <https://talkpython.fm/>

While this isn't an exhaustive list, these will give you a good sense of the python community and the tools, news, and thinking behind many data science methods.

18.6 Conclusion

Hopefully this book provided a solid foundation to learn more about pandas and it's related libraries. Be sure to checkout the accompanying github repository for the book for updates and additional resources: https://github.com/chendaniely/pandas_for_everyone

Part VI: Appendix

[Chapter A](#), “[Installation](#),” Installing Python.

[Chapter B](#), “[Command line](#),” Using the Command Line.

[Chapter C](#), “[Project Templates](#),” Project Templates for your data and code.

[Chapter D](#), “[Using Python](#),” Running and Using Python.

[Chapter E](#), “[Working Directories](#),” Integrating Project Templates with Running and Using Python.

[Chapter F](#), “[Environments](#),” Having different environments to run your code.

[Chapter G](#), “[Install packages](#),” Installing libraries for more functionality.

[Chapter H](#), “[Importing Libraries](#),” Importing libraries.

[Chapter I](#), “[Lists](#),” Working with Python list objects.

[Chapter J](#), “[Tuples](#),” Working with Python tuple objects.

[Chapter K](#), “[Dictionaries](#),” Working with Python dictionary objects.

[Chapter L](#), “[Slicing Values](#),” Getting multiple values from container objects.

[Chapter M](#), “[Loops](#),” Iterating through objects using loops.

[Chapter N](#), “[Comprehensions](#),” A Python way to loop and create lists and dicts.

[Chapter O](#), “[Functions](#),” Writing functions.

[Chapter P](#), “[Ranges and Generators](#),” Ranges and generators.

[Chapter Q](#), “[Multiple Assignment](#),” Assigning multiple variables simultaneously.

[Chapter R](#), “[numpy ndarray](#),” Getting the numpy array from pandas objects.

[Chapter S](#), “[Classes](#),” A quick introduction to classes, methods, and attributes.

[Chapter T](#), “[Odo: The Shapeshifter](#),” Converting various input output datatypes.

Chapter U, “Datasets,” Datasets used in the book.

Appendix A. Installation

There are two Python distributions that have been gaining popularity over the years mainly due to the ease of installing Python and its various modules.

1. **Anaconda:** <https://www.anaconda.com/download/>

2. **Enthought Canopy:** <https://store.enthought.com/downloads/>

Both distributions work on Windows, Mac, and Linux operating systems. The main benefits of using these scientific python distributions are

1. **local installations:** allow you to install the distributions without needing administrative privileges

2. **python package manager:** helps with the installation of various python packages that have non-python dependencies

Since Software-Carpentry has been using the Anaconda distribution, I will be using it for the installation instructions below. You can also look at the generic workshop template installation instructions for Python here:

<https://swcarpentry.github.io/workshop-template/#python>

A.1 Installing Anaconda

For the most part, the directions listed on the main Anaconda download site¹ will be the same as the ones listed in this book. You can also look at the Anaconda installation documentation.² Be sure to use the Python 3 version. If you also need to have Python 2 follow the instructions in [Appendix F](#) on creating Python environments.

¹<https://www.anaconda.com/download/>

²<https://docs.continuum.io/anaconda/install/>

A.1.1 Windows

Install Anaconda using the Windows installer with all the default settings. At the end make sure you make Anaconda the default Python on the system.

A.1.2 Mac

Install Anaconda using the Mac installer with all the default settings. Make sure you make Anaconda the default Python on the system.

A.1.3 Linux

Installing on Linux involves downloading the .sh file and running it from the command line. You

can either do this by navigating to the Anaconda download site and downloading the `.sh` file there, or if you are on a server, for example, you can use the `wget` command. Assuming the `.sh` file is in your ‘Downloads’ folder:

```
$ cd ~/Downloads  
$ bash Anaconda3 ...*.sh # your version number will differ
```

Note, the version of Anaconda will be different by the time this book is published.

Keeping the default options are good, and when it asks you to read the license agreement, you can press `q` to exit to accept by typing `yes`.

Type `yes` when the installer asks to prepend Anaconda to the `PATH`, this makes Anaconda the default python on the system.

When you are done, close the current terminal window, and any new terminal moving forward will default to the Anaconda Python distribution.

A.2 Uninstall Anaconda

Since Anaconda will create an `Anaconda3` folder in your home directory, deleting this folder will completely remove anything associated with anaconda on the machine. This is one of my favorite features of using Anaconda, If I install a bad Python package, I can reset everything back to “normal” by deleting the `Anaconda3` folder

Appendix B. Command line

Having some familiarity with the command line can go a very long way. My main suggestion is to go through the Software-Carpentry Unix Shell lesson¹. The “Navigating Files and Directories” is probably the most important lesson there for this book, but learning about “Shell Scripts” is also important when running your python code from the command line.

¹<http://swcarpentry.github.io/shell-novice/>

Since this book is mainly a Python book about Pandas, I won’t be able to go over all of the topics in learning the Unix Shell. The main takeaway I want to convey in this chapter is the notion of a “working directory”.

B.1 Installation

For the most part, if you are on a Mac or Linux system, you will already have access to the Bash Shell. By default, Windows does not have it installed.

B.1.1 Windows

In Windows, the best way would be to follow the Software-Carpentry Bash Shell instructions². You will be installing Git for Windows³ that will also provide the Bash Shell. The SWC Windows Installer⁴ is also a good idea, since it will also provide a terminal based text editor and various other useful tools.

²<https://swcarpentry.github.io/workshop-template/#setup>

³<https://git-for-windows.github.io/>

⁴<https://github.com/swcarpentry/windows-installer/releases>

If you do not want to use Git for Windows, Anaconda also comes with its own Anaconda Prompt that you can use to run Python code from the command line. The only difference here is that the Anaconda Prompt will use Windows command line commands, instead of the UNIX-like ones on a Mac or Linux system. However, running your Python scripts from the command line will be the same.

B.1.2 Mac

You can find the `Terminal` application in `Applications/ Utilities`. That is, in your main application folder, there will be a folder called `Utilities`, where you can find the `Terminal`.

iTerm⁵ is a popular alternative to the default Mac Terminal application.

⁵<https://www.iterm2.com/>

B.1.3 Linux

The terminal and **bash** is setup on Linux systems by default. There is nothing to install or setup.

B.2 Basics

At minimum, you should know the following commands:

- Where you currently are in your file system (Windows: **cd**, Mac/Linux: **pwd**)
- List the contents of the current folder you are in (Windows: **dir**, Mac/Linux: **ls**)
- Change to a different folder (Windows: **cd <folder name>**, Mac/Linux: **cd <folder name>**)
- Run a python script (Windows/Mac/Linux: **python <python script>**)

The last other useful “command” is the .. (two dots) which refers to the parent folder of where you are now (**cd** in Windows/**pwd** in Mac/Linux).

Appendix C. Project Templates

It is very easy and convenient to put all the data, code, and outputs in the same folder. However, this convenience is negated by a messy project folder. That is, by putting everything into a single folder, it can easily lead to a folder on your computer with tens or hundreds of files, and can become unmanageable and confusing for not only others, but yourself.

At minimum, I suggest the following folder structure for any analysis project:

```
my_project/
  |- data/
  |- src/
  +- output/
```

I put all my datasets in the **data** folder, any code I write for analysis in the **src** (sometimes I call this **code**) folder, and finally cleaned datasets or other outputs such as figures will go into the **output** folder. You can adapt this general folder structure as you need.

Here is a paper reference that discusses the theory a bit further: **Noble WS (2009) A Quick Guide to Organizing Computational Biology Projects. PLoS Comput Biol 5(7): e1000424.**
<https://doi.org/10.1371/journal.pcbi.1000424>

Appendix D. Using Python

There are many different ways to use Python. the ‘simplest’ way would be using a text editor and terminal. However, projects like IPython and Jupyter have enhanced Python’s REPL (Read-Evaluate-Print-Loop) interface, making it one of the standards interfaces in the data analytics and scientific Python communities.

D.1 Command line and text editor

To use Python from the command line and text editor, all you need is a plain text editor and a terminal. Although any plain text editor would work, a ‘good’ one would be able to have a Python feature that will do syntax highlighting, and auto-completion. Popular multi platform text editors include Sublime Text¹, and Atom². Textmate³ and TextWrangler⁴ are other popular text editors for Macs. Notepad++⁵ could be another for Windows users.

¹<http://www.sublimetext.com/3>

²<https://atom.io/>

³<https://macromates.com/>

⁴<http://www.barebones.com/products/textwrangler/>

⁵<https://notepad-plus-plus.org/>

If you are on Windows, just be careful not to do too much editing using the default Notepad application, especially if you plan to collaborate with users on other operating systems. The reason is mainly because line endings are different from Windows and *nix machines (Linux and Macs). If you ever open up a Python file and all the indentations and new lines are not showing up correctly, it’s probably because of how Windows is interpreting the new line endings of the file.

Once you have a text editor, all your python code will be saved in a `.py` script. You can run the script by executing it from the command line. For example if your script’s name is `my_script.py`, you can execute all the code in the script, line-by-line:

```
$ python my_script.py
```

More about running python scripts from the command line in [Appendix B](#) and [E](#)

D.2 Python and IPython

Under Windows, Anaconda will provide a “Anaconda command prompt”. This is just like the regular windows command prompt, but it configured to use the Anaconda Python distribution. typing `python` or `ipython` here will open the `python` or `ipython` command prompt, respectively

For OSX and Linux, you can run the `python` or `ipython` command prompt by typing the respective command in a terminal.

There are a few differences between the `python` and `ipython` command prompt. the regular `python` prompt only takes python commands, whereas the `ipython` prompt provides some useful additional commands you can type to enhance your python experience. My personal suggestion is to use the ipython prompt.

You can either directly type python commands into either prompt, or save your code in a file, and copy/paste commands into the prompt to run your code.

D.3 Jupyter

Instead of running `python` or `ipython` in the command prompt to run python, you can run the `jupyter notebook`. This will open another python interface in a web browser. Even though a web browser is opened, it does not actually need any internet connection to run. Nor is any information being sent across the internet.

the `jupyter notebook` will open up in a location on your computer. You can create a new “notebook” by clicking the “new” button on the top right corner, and selecting “python”.

This will open up a “notebook” for you to type your python commands. Each cell provides a means for you to type your code, and you can run the cell by either using the commands in the “Cell” menu bar, or you can type `Shift + Enter` to run the cell and create a new cell below, or `Ctrl + Enter` to simply run the cell.

What is useful in the notebook is the ability to interweave your python code, its output, along with regular prose text.

To change the cell type, make sure you have the cell selected, then on the top right below the menu bar, there is a drop-down menu that should say “Code”, if you change this to “Markdown” you can write regular prose text that is not python code to help interpret your results, or record notes about what your code is doing.

D.4 Integrated Development Environments (IDEs)

Anaconda comes with an IDE called Spyder. Those who are familiar with Matlab or RStudio might take comfort to a similar interface.

Other IDEs include

1. rodeo: <https://www.yhat.com/products/rodeo>
2. nteract: <https://nteract.io/>
3. pycharm: <https://www.jetbrains.com/pycharm/>

I would suggest exploring the various ways to use python and see which works for you. The ipython/script, jupyter notebook, and Spyder come pre-installed with Anaconda, so those would be the most accessible, but the other IDEs might work better for you.

Appendix E. Working Directories

Building on the [Appendix Chapters B, C, and D](#), this appendix covers working directories, especially when working with project templates ([Chapter C](#)).

A working directory simply tells the program where the base or reference location is. It's common to have all of your code, data, output, figures, etc all in the same folder because the working directory is easy to figure out. However, this can easily lead to a messy folder as mentioned in [Appendix C](#).

We like fully documented project templates that tell us where and how to run our scripts so all our scripts have a predictable and consistent working directory.

There are a few ways to figure out what your current working directory is. If you are using `IPython`, then you can type `pwd` into the `IPython` prompt and it will return you the folder path of your current working directory. This method also works if you are using the `jupyter notebook`.

If you are executing your python code as scripts directly in the command line, then the working directory is the out after you run `cd` on Windows (note there is nothing else after the command), and `pwd` on OSX and Linux.

Here is an example of how working directories effect your code.

Let's say you have have the following project structure, I will denote the current working directory with the *

```
my_project/
    |- data/
    |    + data.csv
    |- *src/
    |    + script.py
    +- output/
```

If my `script .py` wants to read in a data set from the `data` folder, it would have to do something like `data = pd.read_csv ('.. / data / data.csv')`. Note that because my current working directory is in the `src` folder, to navigate to the `data.csv`, I need to go up one level .. then to the `data` folder to get to my dataset. The benefit of this is that I can run my code by tying `python script .py`, but this can lead to issues I'll discuss below.

Let's use a different working directory:

```
*my_project/
    |- data/
    |    + data.csv
    |- src/
    |    + script.py
    +- output/
```

Now that my working directory is on the top level, my `script . py` can reference my dataset by typing `data = pd.read_csv('data/data.csv')`. Note that I no longer need to go up a level to reference my data. However, now if I want to run my code, I have to reference the file as such: `python src/script .py`. This may be annoying, but this allows you to create any amount of subfolders, and the `data` and `output` will always be referenced the same exact way across all the files.

This also means I as a user have one and only one working directory to execute any script in this project.

Appendix F. Environments

Using environments is a great way to work with different versions of Python and/or packages. It also provides an isolated environment to install everything so if something goes wrong, it won't affect the rest of the system. Python environments are particular handy when you need both Python 3 and Python 2 installed on your system to work with different python projects. You can also use environments to see if all the package dependencies.

I personally use the Anaconda python distribution, which comes with `conda`. The “Getting Started” guide will be a useful resource here.¹ If you installed Anaconda with Python 3 ([Appendix A](#)), this appendix will show you how you can create a separate environment that has Python 2 in it. If we run `python` in the command line, we will begin with Python 3, your exact version will differ.

¹<https://conda.io/docs/user-guide/getting-started.html>

```
$ python
Python 3.6.2 |Continuum Analytics, Inc.| (default, Jul 20 2017)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To create a new environment we run the `conda` command in the command line. We use the `create` command within `conda` and specify a `--name` we want to give the environment. Here we are naming our python environment `py2`. By default, it will create a Python 3 evnriomment, so we have to specify our python version with `python=2`

```
$ conda create --name py2 python=2
```

After running the command you will see the following output

```
Fetching package metadata .....
Solving package specifications: .

Package plan for installation in environment "/anaconda3/envs/py2" :

The following NEW packages will be INSTALLED:

certifi :    2016.2.28 - py27 0
openssl :    1.0.2l - 0
pip :        9.0.1 - py27 1
python :      2.7.13 - 0
readline :    6.2 - 2
setuptools : 36.4.0 - py27 0
sqlite :     3.13.0 - 0
tk :          8.5.18 - 0
wheel :      0.29.0 - py27 0
zlib :       1.2.11 - 0

Proceed ([y]/n)? y

certifi -2016.2.28 100% [=====] Time : 0:00:00 3.76 MB/s
setuptools -36.4.0 100% [=====] Time : 0:00:00 6.23 MB/s
#
# To activate this environment , use :
# > source activate py2
#
# To deactivate an active environment , use :
# > source deactivate
#
```

The last few lines of the output tell you how you can use your newly created environment. If we run `source activate py2` in the command line now, Our prompt will now be prepended with our environment name. and now if we run `python` in the terminal to launch python, you can see a different version of python being used

```
$ python
Python 2.7.13 |ContinuumAnalytics, Inc.| (default , Dec 20 2016)
[GCC 4.4.7 20120313 (Red Hat 4.4.7 - 1)] on linux2
Type "help" , "copyright" , "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

To delete an environment, you can navigate to your `anaconda3` folder. There is a folder there called `envs` that store all your environments, In this example if we delete the `py2` folder within `envs`, it's as if we never created our environment, and it will be removed.

Within a given environment any package or library we install ([Appendix G](#)) within it will be specific to that particular environment. So we can not only have different versions of python between environments, but also different versions of libraries too. You can create a separate python environment (`pfe` for “Pandas For Everyone”) for this book as well.

```
$ conda create --name pfe python=3
```

You can install the libraries needed by following [Appendix G](#).

Appendix G. Install packages

There will be times when you have to install a python package that did not come with your distribution. If you used Anaconda to install python, then you will have a package manager called **conda**.

conda has gained popularity over the past few years because of its ability to install python packages that require non-python dependencies. You may have heard of other package managers, such as **pip**.

This book uses a few packages that need to be installed. If you installed the entire Anaconda distribution, then libraries like **pandas** are already installed. But there's no harm in running the command to reinstall it.

Please check the accompanying repository¹ for all the commands to install the relevant libraries for the book.

¹https://github.com/chendaniely/pandas_for_everyone

We can use **conda** to install python libraries. If you created a separate environment for the book, then we can **source activate pfe** to get into the “pandas for everyone” environment.

conda's default repository is maintained by Anaconda (formerly known as Continuum Analytics). We can install **pandas** using **conda**

```
$ conda install pandas
```

For certain packages that are not listed in the default channel, or if the default channel does not have the latest version of a package, we can use the user and community maintained conda-forge channel².

²<https://conda-forge.org/>

```
$ conda install -c conda-forge pandas
```

Lastly, if the package isn't listed in conda you can also use **pip** to install packages.

```
$ pip install pandas
```

For example to install all the libraries used in the book you can run the following lines:

```
$ conda install pandas xlwt openpyxl feather -format seaborn numpy  
$ conda install ipython jupyter statsmodels scikit-learn regex wget  
$ conda install -c conda-forge pweave  
$ pip install lifelines  
$ pip install pandas-datareader
```

Again, best to check the accompanying repository for the most recent installation and setup instructions.

G.1 Updating Packages

You can update `conda` itself with

```
$ conda update conda
```

and to update all the packages in a given conda environment

```
$ conda update --all
```

Appendix H. Importing Libraries

Libraries provide additional functionality in an organized and packaged way. We mainly work with the Pandas library throughout the book, but there are times where we will import other libraries. You will see many different ways to import a library. The most basic way is to simply import the library by its name.

```
import pandas
```

This way we can use functions within pandas using dot notation.

```
pandas.read_csv('..../data/concat_1.csv')
```

```
   A   B   C   D  
0 a0 b0 c0 d0  
1 a1 b1 c1 d1  
2 a2 b2 c2 d2  
3 a3 b3 c3 d3
```

Python gives us a way to alias libraries. This allows us to use an abbreviation for longer library names. To do this, we specify the alias after the `as` statement.

```
import pandas as pd
```

Now, instead of referring to the library as `pandas`, we can use our abbreviation, `pd`

```
pd.read_csv('..../data/concat_1.csv')
```

```
   A   B   C   D  
0 a0 b0 c0 d0  
1 a1 b1 c1 d1  
2 a2 b2 c2 d2  
3 a3 b3 c3 d3
```

Sometimes if there is only a few function that are needed from a library, we can import them directly

```
from pandas import read_csv
```

This will allow us to use the `read_csv` function directly, without specifying the library it is coming from

```
read_csv('..../data/concat_1.csv')
```

```
   A   B   C   D  
0 a0 b0 c0 d0  
1 a1 b1 c1 d1  
2 a2 b2 c2 d2  
3 a3 b3 c3 d3
```

Finally, there is a method that enables users to import all the functions of a library directly into the namespace.

```
from pandas import *  
from numpy import *  
from scipy import *
```

However this method is not recommended since libraries contain many functions, and a function can “overwrite” an existing function. For example, if we import all the functions from `numpy`, and

`scipy`, which mean function is used? It's not as clear as saying `np.mean` and `sp. mean`.

Appendix I. Lists

Lists are a fundamental data structure in Python. They are used to store heterogeneous data, and are created with a pair of square brackets, [].

```
my_list = ['a', 1, True, 3.14]
```

We can subset the list using square brackets and provide the index of the item we want

```
# get the first item
print(my_list[0])
a
```

We can also pass in a range of values ([Appendix L](#))

```
# get the first 3 values
print(my_list[:3])
['a', 1, True]
```

We can also re-assign values when we subset values from the list.

```
# reassign the first value
my_list[0] = 'zzzzz'
print(my_list)
['zzzzz', 1, True, 3.14]
```

Lists are **objects** in python ([Appendix S](#)) so they will have methods that they can perform. For example, we can **append** values to the list.

```
my_list.append('appended a new value!')
print(my_list)
['zzzzz', 1, True, 3.14, 'appended a new value!']
```

More about lists and its various methods can be found in the documentation.¹

¹<https://docs.python.org/3/tutorial/datastructures.html#more-on-lists>

Appendix J. Tuples

A **tuple** is similar to a **list**, in that they both can hold heterogeneous bits of information. The main difference is the contents of a tuple are “immutable”, meaning they cannot be changed. They are also created with a pair of round brackets, ()

```
my_tuple =('a', 1, True, 3.14)
```

Subsetting items is exactly the same as a list (i.e., you use square brackets)

```
# get the first item
print(my_tuple[0])
a
```

However, if we try to change the contents of an index, we will get an error.

```
# this will cause an error
my_tuple[0] = 'zzzzz'
Traceback (most recent call last):
  File "<ipython-input-1-3689669e7d2b>", line 2, in <module>
    my_tuple[0] = 'zzzzz'
TypeError: 'tuple' object does not support item assignment
```

More about tuples can be found in the documentation.¹

¹<https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences>

Appendix K. Dictionaries

Python dictionaries (`dict`) are efficient ways of storing information. Just like how an actual dictionary stores a word and its corresponding definition, a python `dict` stores some key and a corresponding value. This also means dictionaries can make your code more readable since there will be a label assigned to each value in the dictionary. Contrast this with `list` objects that are unlabeled. Dictionaries are created by using a set of curly-brackets, {}.

```
my_dict = {}
print(my_dict)
[]

print(type(my_dict))
<class 'dict'>
```

When we have a `dict`, we can add values to it by using square brackets, []. Where we put the key inside the square brackets. Usually, it is some string, but it technically can be any immutable type (e.g., a python tuple, which is the immutable form of a python `list`). Here we create two keys, `fname` and `lname`, for a first-name and last-name, respectively.

```
my_dict['fname'] = 'Daniel'
my_dict['lname'] = 'Chen'
```

We can also create a dictionary directly, with key-value pairs instead of adding them one-at-a-time. To do this, we use our curly brackets, and the key-value pairs are specified by a colon.

```
my_dict = {'fname': 'Daniel', 'lname': 'Chen'}
print(my_dict)

{'fname': 'Daniel', 'lname': 'Chen'}
```

In order to get the values from our keys, we can use the square brackets with the key inside.

```
fn = my_dict['fname']
print(fn)

Daniel
```

We can also use the `get` method

```
ln = my_dict.get('lname')
print(ln)

| Chen
```

The main difference between these two ways of getting the values from the dictionary is the behavior when you try to get a key that does not exist. When using the square bracket notation, trying to get a key that does not exist will return an error

```
# will return an error
print(my_dict['age'])

Traceback (most recent call last):
  File "<ipython-input-1-404b91316179>", line 2, in <module>
    print(my_dict['age'])
KeyError: 'age'
```

Whereas, the `get` method will return `None`.

```
# will return None
print(my_dict.get('age'))
```

None

To get all the **keys** from the **diet**, we can use the **keys** method.

```
# get all the keys in the dictionary
print(my_dict.keys())
dict_keys(['fname', 'lname'])
```

To get all the **values** form the **diet**, we can use the **values** method

```
# get all the values in the dictionary
print(my_dict.values())
dict_values(['Daniel', 'Chen'])
```

To get every key-value pair you can use the **items** method. This can be useful if you need to loop through a dictionary.

```
print(my_dict.items())
dict_items([('fname', 'Daniel'), ('lname', 'Chen')])
```

Each key-value pair is returned in a form of a **tuple**, you can tell because of the use of round brackets, ()

More on dictionaries can be found in the official documentation on data structures.¹

¹<https://docs.python.org/3/tutorial/datastructures.html#dictionaries>

Appendix L. Slicing Values

Python is a 0-indexed language (things start counting from zero), and also left inclusive, right exclusive when specifying range of values. This applies to objects like `lists` and `pandas.Series` where the first element has a position (index) of 0. When creating `ranges` or slicing a range of values from a list-like object, we need to specify the beginning index, and the ending index. This is where the left inclusive right exclusive terminology comes in. The left index will be included in the returned range or slice, and the right index will not.

Think of items in a list-like object to be fenced in. The index will represent the fence post. When we are specifying a range or a slice, we are actually referring to the fence posts and everything between the posts are returned.

[Figure 12-1](#) is an image depicting why this may be the case. This is why when slice from 0 to 1, we only get one value back, and when we slice 1 to 3 we get two values back.

```
l = ['one', 'two', 'three']

print(l[0 : 1])
['one']

print(l[1:3])
['two', 'three']
```

The slicing notation used, `:`, comes in 2 parts. The value on the left denotes the starting value (left inclusive), the value of the right denotes the ending value (right exclusive). We can leave one of these values blank, and it will start from the beginning (if we leave the left value blank) or go to the end (if we leave the right value blank).

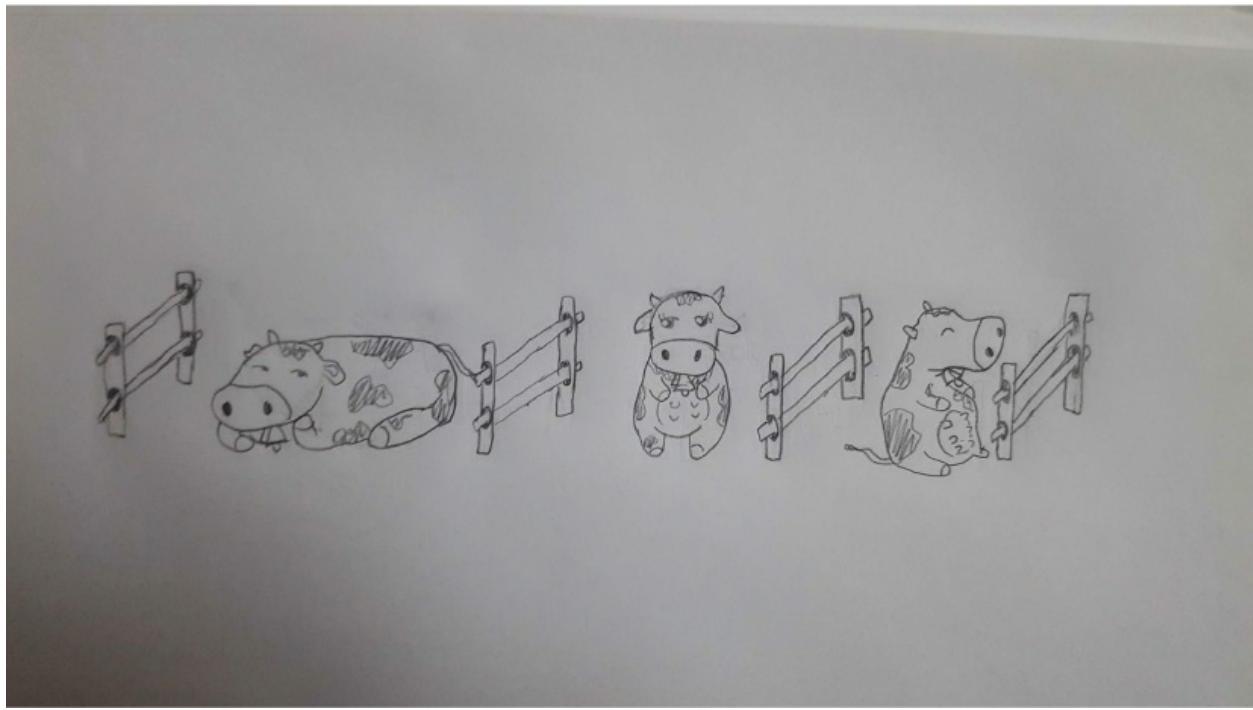
```
print (l[1:])
['two', 'three']

print(l[:3])
['one', 'two', 'three']
```

We can add a second colon, and that refers to the “step”. For example, if we have a step value of `2`, then whatever range we specified from the first colon, the returned value will be every other value from the specified range.

```
# get every other value starting from the first value
print(l[::-2])
['one', 'three']
```

Figure 12-1: Figure of labeled fence posts to depict slicing syntax



(283)

Appendix M. Loops

Loops provide a means to perform the same action across multiple items. Multiple items are typically stored in a python `list` object. Any list-like object can be iterated over (e.g., tuples, arrays, dataframes, dictionaries, etc). More information on loops can be found in the Software-Carpentry Python lesson on Loops.¹

To loop over a list we use a `for` statement. The basic for loop looks like such:

```
for item in container:  
    # do something
```

The `container` represents some iterable of values (e.g., a `list`). The `item` represents a temporary variable that represents each item in the iterable. What happens in the `for` statement, the first element of the container is assigned the temporary variable (in this example `item`). Everything in the indentation block after the colon will be performed. When it gets to the end of the loop, the code will assign the next element in the iterable to the temporary variable and perform the steps over again.

```
# an example list of values to iterate over  
l = [1, 2, 3]  
  
# write a for loop that prints the value and its squared value  
for i in l:  
    # print the current value  
    print('the current value is: {}'.format(i))  
  
    # print the square of the value  
    print("it's squared value is: {}".format(i*i))  
  
    # end of the loop, the \n at the end creates a new line  
    print('end of loop, going back to the top\n')  
  
the current value is: 1  
it's squared value is: 1  
end of loop, going back to the top  
  
the current value is: 2  
it's squared value is: 4  
end of loop, going back to the top  
  
the current value is: 3  
it's squared value is: 9  
end of loop, going back to the top
```

Appendix N. Comprehensions

A typical task in python is to iterate over a list, run some function on each value, and save the results into a new list.

```
# create a list
l = [1, 2, 3, 4, 5]

# list of newly calculated results
r = []

# iterate over the list
for i in l:
    # square each number and add the new value to a new list
    r.append(i ** 2)

print(r)
[1, 4, 9, 16, 25]
```

However, you can see this requires a few lines of code to do a relatively simple task. Another way to re-write the above loop is using a python list-comprehension. It is a shortcut and concise way of performing the same action.

```
# note the square brackets around on the right hand side
# this saves the final results as a list
rc = [i ** 2 for i in l]
print(rc)

[1, 4, 9, 16, 25]

print(type(rc))

<class 'list'>
```

Our final results will be a list, so the right hand side will have a pair of square brackets. From there, we write what looks very similar to a for loop. Starting from the center to the right side, we write **for i in l**, which is very similar to the first line of our original for loop. To the right side, we write **i ** 2**, which is similar to the body of the for loop. Since we are using a list comprehension, we no longer need to specify the list we want to append our new values to.

Appendix O. Functions

Functions are one of the cornerstones in programming. It provides a way to reuse code. If you've ever copy-pasted lines of code just to change a few parameters, then turning those lines of code into a function not only makes your code more readable, but also prevents you from mistakes later on. Every time code is copy-pasted, it adds another place to look if a correction is needed, and puts that burden on the programmer. By using a function, a correction can be made once, and it will be applied every time the function is called.

I highly suggest the Software-Carpentry Python episode on functions for more details.¹

¹<http://swcarpentry.github.io/python-novice-inflammation/06-func/>

An empty function looks like this:

```
def empty_function():
    pass
```

They begin with the `def` keyword, then the function name (i.e., how the function will be called and used), a set of round brackets, and a colon. The body of the function is indented (1 tab or 4 spaces). This indentation is *extremely* important. Otherwise you will get an error. In this example, `pass` is used as a place holder to do nothing.

Typically, functions will have what's called a "docstring". These are multi-line comments that describe the function's purpose, parameters, output, and can also contain testing code. When looking up help documentation about a function in Python, it is usually what is contained in the function docstring that shows up. This allows the function's documentation and code to travel together, which makes the documentation easier to maintain.

```
def empty_function():
    """This is an empty function with a docstring.
    These docstrings are used to help document the function.
    They can be created by using 3 single quotes or 3 double quotes.
    The PEP-8 style guide says to use the double quotes.
    """
    pass # this function still does nothing
```

Functions need not have parameters to be called.

```
def print_value():
    """Just prints the value 3
    """
    print(3)

# call our print_value function
print_value()
```

Functions can take parameters as well. We can modify our `print_value` function such that it prints whatever value we pass into the function.

```
def print_value(value):
    """Prints the value passed into the parameter 'value'
    """
    print(value)

print_value(3)
```

```

print_value("Hello!")
Hello!

Functions can take multiple values as well.

def person(fname, lname, sex):
    """A function that takes 3 values, and prints them
    """
    print(fname)
    print(lname)
    print(sex)

person('Daniel', 'Chen', 'Male')
Daniel
Chen
Male

```

The examples thus far only created functions that printed values. What makes functions powerful are its ability to take inputs and return an output, not just print values to the screen. To accomplish this, we can use the `return` statement

```

def my_mean_2(x, y):
    """A function that returns the mean between 2 values
    """
    mean_value = (x + y) / 2
    return mean_value

m = my_mean_2(0, 10)
print(m)
5.0

```

O.1 Default Parameters

Functions can also have default values. Many functions in various libraries have default values. It allows users to type less by only specifying the minimal amount of information into the function, but also gives the user flexibility to make changes to the function's behavior. Default values are also useful if you have your own functions and want to add more features without breaking your existing code.

```

def my_mean_3(x, y, z=20):
    """A function with a parameter z that has a default value
    """
    # you can also directly return values without having to create
    # an intermediate variable
    return (x + y + z) / 3

```

Here we only **need** to specify **x** and **y**.

```

print(my_mean_3(10, 15))
15.0

```

But we can also specify **z** if we want to override it's default value

```

print(my_mean_3(0, 50, 100))
50.0

```

O.2 Arbitrary Parameters

Sometimes you will see in function documentations the use of `*args` or `**kwargs`. These stand for “arguments” and “keyword arguments”, respectively. They allow the function author to capture an arbitrary number of arguments into the function. Or allow a means for the user to pass arguments into another function that is called within the current function.

O.2.1 *args

Let's write a more generic `mean` function that can take an arbitrary number of values.

```
def my_mean(*args):
    """Calculate the mean for an arbitrary number of values
    """
    # add up all the values
    sum = 0
    for i in args:
        sum += i
    return sum / len(args)
print(my_mean(0, 10))

5.0

print(my_mean(0, 50, 100))

50.0

print(my_mean(3, 10, 25, 2))

10.0
```

O.2.2 **kwargs

`**kwargs` are similar to `*args`, but instead of acting like an arbitrary list of values, they are used like a dictionary by specifying arbitrary pairs of key-value stores.

```
def greetings(welcome_word, **kwargs):
    """Prints out a greeting to a person,
    where the person's fname and lname are provided by the kwargs
    """
    print(welcome_word)
    print(kwargs.get('fname'))
    print(kwargs.get('lname'))

greetings('Hello!', fname='Daniel', lname='Chen')

Hello!
Daniel
Chen
```

Appendix P. Ranges and Generators

The python `range` function allows the user to create a sequence of values, by providing a starting value, and ending value, and if needed, a step value. It is very similar to the slicing syntax in [Appendix L](#). By default, if we give `range` a single number, it will create a sequence of values starting from `0`.

```
# create a range of 5
r = range(5)
```

However, the `range` function doesn't just return a list of numbers. In Python 3 it actually returns a generator (In Python 2 this is the behavior of the `xrange` function).

```
print(r)
range(0, 5)
print(type(r))
<class 'range'>
```

If we wanted an actual `list` of the range, we can convert the generator to a list.

```
lr = list(range(5))
print(lr)
[0, 1, 2, 3, 4]
```

However, before converting generators, think about what you plan to use them for. If you plan to create a generator to write a look over ([Appendix M](#)), then there is no need to convert the generator.

```
for i in lr:
    print(i)
0
1
2
3
4
```

Generators create the next value in the sequence on-the-fly. This means the entire contents of the generator does not need to be loaded into memory before using it. Since generators only know the current position, and how to calculate the next item in the sequence, you cannot reuse generators a second time. This is an example from the built-in `itertools` library in Python.¹

¹<https://docs.python.org/2/library/itertools.html>

It creates a cartesian product of values provided into the function.

```
import itertools
prod = itertools.product([1, 2, 3], ['a', 'b', 'c'])

for i in prod:
    print(i)
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
```

```
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
```

If you need to reuse the cartesian product again, then you'd have to either re-create the generator object again, or convert the generator into something more static, e.g., a `list`

```
# this will also not work because we already spent the generator
for i in prod:
    print(i)

# create a new generator
prod = itertools.product([1, 2, 3], ['a', 'b', 'c'])
for i in prod:
    print(i)

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

Appendix Q. Multiple Assignment

Multiple assignment in Python is a form of syntactic sugar. It provides the programmer to express something succinctly while making it easier to express and understand for others.

Let's have a list of values

```
l = [1, 2, 3]
```

If we wanted to assign a variable to each element of the list, we can subset the list and assign the value.

```
a = l[0]
b = l[1]
c = l[2]

print(a)
1
print(b)
2
print(c)
3
```

However, with multiple assignment, if the statement to the right is some kind of container, we can directly assign its values to multiple variables on the left. So, the above code can be re-written as such

```
a1, b1, c1 = l

print(a1)
1
print(b1)
2
print(c1)
3
```

A common use case of multiple assignments is generating figures and axes while plotting.

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
```

This will create the figure and the axes in a single 1-line command. Other use cases can be seen in the following stack-overflow question: <https://stackoverflow.com/questions/5182573/multiple-assignment-semantics>

Appendix R. numpy ndarray

The `numpy` library¹ gives Python the ability to work with matrices and arrays.

¹<https://docs.scipy.org/doc/numpy/index.html>

```
import numpy as np
```

Pandas started off as an extension to the `numpy.ndarray` by providing more features suitable for data analysis. These days, Pandas has evolved to the point where it shouldn't be thought of as a collection of numpy arrays, since the two libraries are different.

```
import pandas as pd
df = pd.read_csv('../data/concat_1.csv')
print(df)

   A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

If you do need to get the `numpy.ndarray` values from a `Series` OR `DataFrame`, you can use the `values` attribute.

```
a = df['A']
print(a)
0    a0
1    a1
2    a2
3    a3
Name: A, dtype: object

print(type(a))
<class 'pandas.core.series.Series'>

print(a.values)
['a0' 'a1' 'a2' 'a3']

print(type(a.values))
<class 'numpy.ndarray'>
```

This is particularly helpful when cleaning data in pandas, and then using your newly cleaned data in other Python libraries that do not fully support pandas `Series` and `DataFrame` objects. The Software-Carpentry Python Inflammation lesson² uses `numpy` and can be another good reference to learn about the library and Python as a whole.

²<http://swcarpentry.github.io/python-novice-inflammation/>

Appendix S. Classes

Python is an object-oriented language. Meaning that every thing you create or use is a “class”. Classes allow the programmer to group relevant functions and methods together. In pandas, the `Series` and `DataFrame` are classes, and they each have attributes (e.g., `shape`) and methods (e.g., `apply`). While it’s not my intention to give a lesson of object-oriented programming here, I want to very quickly cover classes, with the hope that it can be used to help navigate official documentation and understand why things the way they are.

What’s nice about classes, is that the programmer can define any class for his/her intended purpose. Here is a class that represents a person, there is a first name (`fname`), last name (`lname`), and age (`age`) associated with the person. When the person celebrates his/her birthday (`celebrate_birthday`), the age increases by 1.

```
class Person(object):
    def __init__(self, fname, lname, age):
        self.fname = fname
        self.lname = lname
        self.age = age

    def celebrate_birthday(self):
        self.age += 1
```

With the `Person` class created, we can use it in our code. Let’s create an instance of our `Person`

```
ka = Person(fname='King', lname='Authur', age=39)
```

This created a `Person`, King Authur age 39, and saved him to a variable named `ka`.

We can then get some attributes from `ka` (note attributes are not functions or methods so they do not have round brackets)

```
print(ka.fname)
King
print(ka.lname)
Authur
print(ka.age)
39
```

Finally, we can call the method on our class to increment the `age`

```
ka.celebrate_birthday()
print(ka.age)
```

40

The pandas `Series` and `DataFrame` objects are more complex versions of our `Person` class. But the general concepts are the same. We can instantiate any new class to a variable, and access its attributes or call its methods.

Appendix T. Odo: The Shapeshifter

Odo¹ is a Python library² that is able to convert one type of data into another. For example, we can tell it to load a csv file into a pandas `DataFrame`.

¹[https://en.wikipedia.org/wiki/Odo_\(Star_Trek\)](https://en.wikipedia.org/wiki/Odo_(Star_Trek))

²<https://odo.readthedocs.io/en/latest/>

```
from odo import odo
import pandas as pd

df = odo('../data/concat_1.csv', pd.DataFrame)
print(df)

/home/dchen/anaconda3/envs/book36/lib/python3.6/site-
packages/odo/backends/pandas.py:102: FutureWarning: pandas.tslib is
deprecated and will be removed in a future version.
You can access NaTType as type(pandas.NaT)
@convert.register((pd.Timestamp, pd.Timedelta), (pd.tslib.NaTType,
type(None)))
   A   B   C   D
0  a0  b0  c0  d0
1  a1  b1  c1  d1
2  a2  b2  c2  d2
3  a3  b3  c3  d3
```

What `odo` is doing is that it knows all the various ways one data format can be converted to another using a variety of python libraries and functions. It creates a graph that converts one type to another, and runs the series of conversions for you.

Loading `csvs` are not the only file type it can work with, `json`, `hdf5`, `xls`, and `sas7bdat` are just some other types of files it can load. It can also make database connections using the `SQLAlchemy` library, so you can pull down SQL tables into a `csv` or `DataFrame`. You can even use `odo` to upload `DataFrames` into a SQL table. There are connections to Spark/SparkSQL as well as AWS and Hive.

The library is definitely worth looking into if you are constantly converting data formats.