

Master Data Analysis with Python

by
Ted Petrou

© 2020 Ted Petrou All Rights Reserved

Contents

I	Intro to Pandas	1
1	What is pandas?	3
1.1	pandas operates on tabular data	4
1.2	pandas examples	4
1.3	Which pandas version to use?	5
1.4	Reading data	5
1.5	Filtering data	6
1.6	Aggregating methods	8
1.7	Non-aggregating methods	9
1.8	Aggregating within groups	9
1.9	Tidying	12
1.10	Joining Data	14
1.11	Time Series Analysis	15
1.12	Visualization	16
1.13	Much More	17
2	The DataFrame and Series	19
2.1	Reading external data with pandas	19
2.2	Components of a DataFrame	22
2.3	What type of object is <code>bikes</code> ?	23
2.4	Select a single column from a DataFrame - a Series	24
2.5	Components of a Series	24
2.6	Changing display options	25
2.7	Exercises	26
3	Data Types and Missing Values	29
3.1	Common data types	29
3.2	String data type - major enhancement to pandas 1.0	29
3.3	Missing value representation	30
3.4	New Integers and booleans data types in pandas 1.0	30
3.5	Recommendation for Pandas 1.0 - Avoid the new data types	30
3.6	Finding the data type of each column	31
3.7	Getting more metadata	32
3.8	More data types	33
3.9	Exercises	34
4	Setting a Meaningful Index	35
4.1	Setting an index of a DataFrame	35
4.2	Accessing the index, columns, and data	37
4.3	Accessing the components of a Series	39
4.4	Setting an index on read	40

4.5 Choosing a good index	42
4.6 Exercises	43
5 Five-Step Process for Data Exploration	45
II Selecting Subsets of Data	49
6 Selecting Subsets of Data from DataFrames with just the brackets	51
6.1 pandas dual references: by label and by integer location	53
6.2 The three indexers [], loc, iloc	53
6.3 Begin with <i>just the brackets</i>	53
6.4 Select multiple columns with a list	54
6.5 Summary of <i>just the brackets</i>	56
6.6 Exercises	57
7 Selecting Subsets of Data from DataFrames with loc	59
7.1 Simultaneous row and column subset selection	59
7.2 loc with slice notation	61
7.3 Summary of the loc indexer	66
7.4 Exercises	66
8 Selecting Subsets of Data from DataFrames with iloc	69
8.1 Simultaneous row and column subset selection	69
8.2 Summary of iloc	74
8.3 Exercises	74
9 Selecting Subsets of Data from a Series	77
9.1 Series indexer rules	77
9.2 Series subset selection with loc	78
9.3 Series subset selection with iloc	79
9.4 Summary of Series subset selection	81
9.5 Exercises	81
10 Boolean Selection Single Conditions	85
10.1 Manually filtering the data	86
10.2 Operator overloading	87
10.3 Practical boolean selection	87
10.4 Boolean selection in one line	89
10.5 Single condition expression	89
10.6 Summary of single condition boolean selection	90
10.7 Exercises	90
11 Boolean Selection Multiple Conditions	93
11.1 Logical operators	93
11.2 Multiple conditions in one line	94
11.3 Using an or condition	95
11.4 Inverting a condition with the not operator	95
11.5 Many equality conditions in a single column	97
11.6 Exercises	99
12 Boolean Selection More	103
12.1 Boolean selection on a Series	103
12.2 The between method	105

12.3 Simultaneous boolean selection of rows and column labels with <code>loc</code>	105
12.4 Column to column comparisons	107
12.5 Finding Missing Values with <code>isna</code>	108
12.6 Exercises	108
13 Filtering with the <code>query</code> Method	111
13.1 The <code>query</code> method	111
13.2 Summary	118
13.3 Exercises	118
14 Miscellaneous Subset Selection	121
14.1 Selecting a column with dot notation	121
14.2 Slicing rows with just the brackets	123
14.3 Selecting a single cell with <code>at</code> and <code>iat</code>	124
III Essential Series Commands	127
15 Series Attributes and Statistical Methods	129
15.1 Calling methods on a Series	129
15.2 Series Attributes and Methods	129
15.3 Core Series attributes	130
15.4 Arithmetic operators	131
15.5 Comparison operations	132
15.6 Boolean and bitwise operators	133
15.7 Statistical methods	133
15.8 Aggregation methods	134
15.9 Non-Aggregation methods	136
15.10 Series with a non-default index	138
15.11 Operations on a boolean Series	139
15.12 Exercises	140
16 Series Missing Value Methods	143
16.1 Methods for handling missing values	143
16.2 The <code>isna</code> method	143
16.3 Dropping missing values with <code>dropna</code>	145
16.4 Filling missing values with the <code>fillna</code> method	146
16.5 Filling missing values with <code>interpolate</code>	148
16.6 Graphing interpolation methods	149
16.7 Interpolation methods use the index	150
16.8 Exercises	151
17 Series Sorting, Ranking, and Uniqueness	155
17.1 Sorting	155
17.2 Ranking	157
17.3 Uniqueness	159
17.4 Exercises	161
18 Series Methods More	163
18.1 The <code>agg</code> method	163
18.2 Index of maximum and minimum	164
18.3 Differencing methods <code>diff</code> and <code>pct_change</code>	165
18.4 The <code>nlargest</code> and <code>nsmallest</code> methods	167

18.5 Randomly sample a Series	168
18.6 The <code>replace</code> method	169
18.7 Exercises	171
19 Series String Methods	173
19.1 The <code>value_counts</code> method	174
19.2 Special methods just for object columns	175
19.3 The <code>count</code> string method	176
19.4 The <code>contains</code> str method	176
19.5 The <code>len</code> str method	177
19.6 The <code>split</code> str method	177
19.7 The <code>replace</code> str method	178
19.8 Selecting substrings with the brackets	178
19.9 Regular expressions	179
19.10 Exercises	180
20 Series Datetime Methods	183
20.1 The <code>dt</code> accessor	183
20.2 Datetime Attributes	184
20.3 Datetime methods	186
20.4 Format time as a string with <code>strftime</code>	187
20.5 Convert to period	187
20.6 Timedeltas	189
20.7 Exercises	189
21 Project - Testing Normality of Stock Market Returns	191
21.1 Results discussion	194
21.2 Exercises	194
IV Essential DataFrame Commands	197
22 Introduction to DataFrame	199
22.1 DataFrames vs Series	199
22.2 Core DataFrame attributes	200
22.3 Arithmetic DataFrame operations	202
22.4 DataFrames comparison operators	205
22.5 Overlap of DataFrame and Series methods	205
22.6 Data Dictionaries	206
22.7 Exercises	208
23 DataFrame Statistical Methods	209
23.1 Aggregation methods	209
23.2 Changing the direction of the operation	211
23.3 Non-Aggregation methods	214
23.4 Nuisance Columns	218
23.5 Exercises	220
24 DataFrame Missing Value Methods	223
24.1 Methods for handling missing values	223
24.2 The <code>isna</code> method	223
24.3 Dropping rows and columns with the <code>dropna</code> method	225
24.4 Filling missing values with the <code>fillna</code> method	226

24.5 The <code>interpolate</code> method	229
24.6 Exercises	231
25 DataFrame Sorting, Ranking, and Uniqueness	233
25.1 Sorting	233
25.2 Ranking	236
25.3 Uniqueness	237
25.4 Finding the maximum/minimum of a group	239
25.5 Exercises	242
26 DataFrame Structure Methods	245
26.1 Adding a new column to the DataFrame	245
26.2 Copying a DataFrame	247
26.3 Column and Row Dropping and Renaming	250
26.4 Inserting columns in the middle of a DataFrame	253
26.5 The <code>pop</code> method	254
26.6 Exercises	255
27 DataFrame Methods More	257
27.1 The <code>agg</code> method	257
27.2 The <code>idxmax</code> and <code>idxmin</code> methods	258
27.3 The <code>diff</code> and <code>pct_change</code> methods	259
27.4 The <code>sample</code> method	260
27.5 The <code>nlargest/nsmallest</code> methods	261
27.6 The <code>corr</code> method	261
27.7 The <code>replace</code> method	262
27.8 Methods available only to Series and not DataFrames	266
27.9 Exercises	267
28 Assigning Subsets of Data	269
28.1 Setting new data with <code>loc</code>	269
28.2 Setting new data with <code>iloc</code>	271
28.3 Boolean selection assignment	272
28.4 Improper Assignment	273
28.5 Exercises	274
V Data Types	275
29 Integer, Float, and Boolean Data types	277
29.1 Constructing a Series	277
29.2 Integer data type	277
29.3 Changing Data Types with <code>astype</code>	278
29.4 Unsigned Integers	279
29.5 Nullable integer data type	280
29.6 Summary of integer data type	283
29.7 Float data types	283
29.8 Changing from float to int	284
29.9 Boolean data type	286
29.10 Nullable boolean data type	288
29.11 Changing data types with an arithmetic operation	289
29.12 Setting data types in numpy arrays	290
29.13 Different syntax for data types	291

29.14 Boolean, integer, and float data type summary	292
29.15 Exercises	292
30 Object, String, and Categorical Data Types	295
30.1 Object data types	295
30.2 Categorical data type	297
30.3 Why the categorical data type is useful	299
30.4 The <code>cat</code> accessor	300
30.5 Modifying categories	300
30.6 Massive reduction in memory used	302
30.7 Speeding up operations	303
30.8 The <code>str</code> accessor is still available	304
30.9 Ordered categories	304
30.10 Integers can be categories	308
30.11 The new string data type	308
30.12 Converting strings to numeric	310
30.13 Force conversion with <code>pd.to_numeric</code>	310
30.14 Object, String, and Categorical data type summary	311
30.15 Exercises	311
30.162. Object, String, and Categorical Data Types	311
31 Datetime, Timedelta, and Period Data Types	315
31.1 The numpy datetime64 data type	315
31.2 The pandas datetime64 data type	317
31.3 The numpy timedelta64 data type	319
31.4 The pandas timedelta64 data type	319
31.5 The pandas Period data type	320
31.6 Datetime, Timedelta, and Period data type summary	321
31.7 Exercises	322
32 DataFrame Data Type Conversion	325
32.1 The <code>astype</code> method for DataFrames	327
32.2 Reading in data with known missing values	328
32.3 More data type conversion with the housing dataset	328
32.4 Exercises	330
VI Grouping Data	333
33 Grouping Aggregation Basics	335
33.1 Group into independent DataFrames, then aggregate	335
33.2 Grouping with the <code>groupby</code> method	336
33.3 Syntax for using the <code>groupby</code> method	336
33.4 The index when grouping	339
33.5 More on method chaining with <code>groupby</code>	340
33.6 <code>GroupBy</code> objects	340
33.7 Exercises	341
34 Grouping and Aggregating with Multiple Columns	345
34.1 Review grouping and aggregating with a single column	345
34.2 Grouping with multiple columns	346
34.3 Aggregating multiple columns	347
34.4 Multiple grouping columns, aggregating columns, and aggregating functions	348

34.5 Getting the size of each group	348
34.6 Exercises	351
35 Grouping with Pivot Tables	353
35.1 Creating the pivot table above with pandas	353
35.2 Comparison to <code>groupby</code>	355
35.3 Styling pivot tables	356
35.4 Getting the size of each group	360
35.5 Add margins to get row and column totals	361
35.6 Non-standard pivot tables	361
35.7 Exercises	367
36 Counting with Crosstabs	369
36.1 Frequency counting with a Series	371
36.2 Counting the mental health occurrences by country	372
36.3 Counting frequency with the <code>crosstab</code> function	374
36.4 Exercises	377
37 Alternate Groupby Syntax	379
37.1 Aggregating a single column	379
37.2 No Aggregating Columns	382
38 Custom Aggregation	385
38.1 Using a customized aggregation function	385
38.2 Find the mean salary for the five highest paid employees per department	388
38.3 Percentage of employees by department with salaries greater than 100,000	391
38.4 Optimizing a Custom Aggregation function	393
38.5 Summary of Custom Aggregation Functions	398
38.6 Exercises	398
39 Transform and Filter with Groupby	401
39.1 The <code>groupby filter</code> method	401
39.2 Getting a nicer display	404
39.3 Finding actors that appear in at least 25 movies	406
39.4 Multiple conditions	407
39.5 The <code>groupby transform</code> method	408
39.6 Transforming multiple columns	412
39.7 Summary of the <code>groupby transform</code> method	415
39.8 Exercises	415
40 Other Groupby Methods	417
40.1 Kinds of <code>groupby</code> attributes and methods	417
40.2 Finding all available attributes and methods	420
40.3 Calling single aggregation methods	421
40.4 <code>head</code> , <code>tail</code> , and <code>nth</code> <code>groupby</code> methods	422
40.5 Non-aggregating methods	425
40.6 Exercises	428
41 Create Your Own Data Analysis	431
41.1 Overview	431
41.2 Begin Asking and Answering Questions	432

VII Time Series	433
42 Datetime and Timedelta	435
42.1 Date vs Time vs Datetime	435
42.2 Creating single datetime objects in pandas	435
42.3 Timestamp attributes and methods	439
42.4 Timedelta - an amount of time	440
42.5 Timedelta attributes and methods	441
42.6 Creating timedeltas by subtracting datetimes	441
42.7 Exercises	442
43 Introduction to Time Series	445
43.1 Set the datetime column as the index	445
43.2 Easy subset selection with a DateTimeIndex	446
43.3 Sampling specific times	447
43.4 Upsampling - Increasing the number of rows	449
43.5 Exercises	453
44 Grouping by Time	455
44.1 The PeriodIndex	457
44.2 The Period data type	457
44.3 Anchored offsets	459
44.4 Calling <code>resample</code> on a datetime column	460
44.5 Calling <code>resample</code> on a Series	461
44.6 Exercises	462
45 Rolling Windows	465
45.1 Keep window size the same with an integer	467
45.2 Plotting	469
45.3 Exercises	470
46 Grouping by Time and another Column	471
46.1 Grouping by an amount of time and another column	473
46.2 Group independently	474
46.3 Using a pivot table with <code>Grouper</code> for easier comparisons	475
46.4 Exercises	477
VIII Regular Expressions	479
47 Introduction to Regular Expressions	481
47.1 The <code>contains</code> and <code>extract</code> methods	481
47.2 Mini-Programming Language	482
47.3 Special Characters	484
47.4 The dot metacharacter	484
47.5 The caret metacharacter <code>^</code>	484
47.6 The dollar sign metacharacter <code>\$</code>	485
47.7 Combining special characters	486
47.8 Exercises	486
48 Quantifiers	489
48.1 The asterisk metacharacter <code>*</code>	489
48.2 The plus metacharacter <code>+</code>	490

48.3 The question mark metacharacter ?	490
48.4 The curly braces metacharacter {m,n}	491
48.5 Exercises	491
49 Or Conditions	493
49.1 The pipe metacharacter 	493
49.2 The brackets metacharacter []	494
49.3 Combining Special Characters	495
49.4 Exercises	496
50 Character Sets and Grouping	499
50.1 Special Characters lose their meaning within the brackets	499
50.2 The backslash \ metacharacter	500
50.3 The parentheses metacharacters ()	501
50.4 Using parentheses to change operator precedence	501
50.5 Using capture groups with the <code>extract</code> string method	503
50.6 Many other string methods take regexes	505
50.7 Other Dialects of Regex	505
50.8 More to Regex	505
50.9 Regex Summary	505
50.10 Exercises	506
51 Project - Explore Newsgroups with Regexes	509
51.1 Can you do the following?	510
52 Project - Feature Engineering on the Titanic	511
52.1 Exercises	512
IX Tidy Data	515
53 Tidy Data with <code>melt</code>	517
53.1 Tidy Data	517
53.2 Melting	518
53.3 Exercises	521
54 Reshaping by Pivoting	523
54.1 Inverting melted data with <code>pivot</code>	523
54.2 Pivoting with duplicate values	525
54.3 Using <code>pivot_table</code> to aggregate those values	527
54.4 Exercises	528
55 Common Messy Datasets	529
55.1 Most common messy data problems	529
55.2 Multiple variables are stored in one column	529
55.3 Two or more values are stored in the same cell	531
55.4 Variables are stored in both rows and columns	533
55.5 Steps to produce tidy data	537
55.6 Exercises	538
X Joining Data	539
56 Automatic Index Alignment	541

56.1 Adding two Series - Not as simple as it sounds	541
56.2 Adding a numpy array to a Series	542
56.3 Adding Series that don't have the same index labels	543
56.4 Adding Series with duplicate values in the index	544
56.5 An exception to Cartesian Product rule	546
56.6 DataFrames align on both their index and columns	547
57 Combining Data	551
57.1 Concatenating Data	551
57.2 Beware! Automatic Alignment of Index	553
57.3 Column names align first	553
57.4 Use <code>axis=1</code> to change the direction of concatenation	554
58 SQL Databases	555
58.1 Connecting to a SQL database	555
58.2 The Chinook Database	555
58.3 Primary and Foreign Keys	556
58.4 Preparing the connection	556
58.5 Joining tables in Pandas with <code>merge</code>	558
58.6 Exercises	561
59 Data Normalization	563
59.1 Create a primary key to uniquely identify each row	564
59.2 We just created a dimension	565
59.3 Replacing original data with primary keys	567
59.4 Replace all the other dimensions with primary key columns	568
59.5 Fact Table	570
59.6 Data Model Diagram	570
59.7 Exercises	571
XI Visualization with Matplotlib	573
60 Introduction to matplotlib	575
60.1 Two interfaces of matplotlib	575
60.2 Figure - Axes Hierarchy	576
60.3 Setting the size of the figure upon creation	579
60.4 Axes methods	580
60.5 Change tick label and tick line properties with <code>tick_params</code>	587
60.6 Setting multiple properties at the same time with <code>set</code>	588
60.7 Exercises	589
61 Matplotlib Text and Lines	591
61.1 The axes <code>text</code> method	591
61.2 Creating horizontal lines with <code>hlines</code>	597
61.3 Create vertical lines with <code>vlines</code>	600
61.4 Add grid lines with the <code>grid</code> method	601
61.5 Aligning text horizontally and vertically	603
61.6 Add text with arrows using the <code>annotate</code> method	605
61.7 Exercises	609
62 Matplotlib Resolution	611
62.1 Matplotlib inches	611

62.2 Creating figures with custom DPI	614
62.3 Text and line “points”	615
62.4 Run configuration settings	616
62.5 Creating style sheets	620
62.6 Exercises	623
63 Matplotlib Patches and Colors	625
63.1 Adding matplotlib patches	625
63.2 Circle patches	625
63.3 Ellipse patches	627
63.4 Rectangle patches	628
63.5 Polygon patches	628
63.6 Arc patches	629
63.7 Wedge patches	630
63.8 Matplotlib colors	632
63.9 Color transparency	637
63.10 Layering with zorder	638
63.11 Gray scale	639
63.12 Colormaps	640
63.13 Filling between two lines	643
63.14 Creating a basketball court	644
63.15 Exercises	648
64 Matplotlib Line Plots	651
64.1 Axes API	651
64.2 Line plots with the plot method	651
64.3 Integration with pandas	654
64.4 Color cycle	658
64.5 More line plots	660
64.6 Adding a legend	663
64.7 Exercises	668
65 Matplotlib Scatter and Bar Plots	669
65.1 Scatter plots	669
65.2 Change scatter plot point size	679
65.3 Bar plots	682
65.4 Exercises	690
66 Matplotlib Distribution Plots	691
66.1 Histograms	692
66.2 Box and whisker plots	698
66.3 Exercises	700
67 Best of the Rest of Matplotlib	701
67.1 Axes spines	701
67.2 The <code>xaxis</code> and <code>yaxis</code> objects	703
67.3 Tick locators	704
67.4 Tick formatters	705
67.5 Minor ticks	706
67.6 Horizontal and vertical lines that span the axes	707
67.7 Plotting with dates	708
67.8 Using a different scale for the axis	711
67.9 Adding images	715

67.10 Coordinate systems	720
67.11 Figure methods	723
67.12 Creating a grid of axes	726
67.13 Exercises	729
XII Visualization with Pandas and Seaborn	731
68 Plotting with pandas Series	733
68.1 Line plots	734
68.2 Bar plots	736
68.3 Distribution plots	737
68.4 Pie Charts	738
68.5 Area Plots	739
68.6 Adding a plot to a previously made axes	739
69 Plotting with pandas DataFrames	741
69.1 Line plots	741
69.2 Bar plots	743
69.3 Plotting on separate axes	746
69.4 Distribution plots	747
69.5 Scatter and Hexbin	749
69.6 Area plots	752
70 Seaborn Axes Plots	761
70.1 The seaborn API	761
70.2 seaborn integration with pandas	762
70.3 Distribution Plots	762
70.4 Seaborn style sheets	767
70.5 Other distribution plots	767
70.6 Automatic grouping by category	768
70.7 Grouping within groups with hue	770
70.8 Tidy data	771
70.9 Grouping and Aggregating Plots	772
70.10 Raw data plots	780
70.11 Scatter plots with linear regression lines using <code>regplot</code>	785
70.12 Ordered categorical data	787
70.13 Exercises	789
71 Seaborn Grid Plots	791
71.1 Grids by categories	791
71.2 Scatter and line plot grids	795
71.3 Regression grid plots	797
71.4 Bivariate distributions grids	798
71.5 Scatter plot grids of multiple column combinations	799
71.6 Hierarchical cluster map	800

Part I

Intro to Pandas

Chapter 1

What is pandas?

In this chapter, we introduce the pandas library by providing many of the most popular and powerful data analysis tasks that it can complete.

What is pandas?

pandas is one of the most popular open source data exploration libraries currently available. It gives its users the power to explore, manipulate, query, aggregate, and visualize **tabular** data. Tabular refers to data that is two-dimensional, consisting of rows and columns. Commonly, we refer to this organized structure of data as a **table**.



Why pandas and not xyz?

In this current age of data explosion, there are now dozens of other tools that have many of the same capabilities as the pandas library. However, there are many aspects of pandas that make it an attractive choice for data analysis and it continues to have one of the fastest growing user bases.

- It's a Python library and integrates well with the other popular data science libraries such as numpy, scikit-learn, statsmodels, matplotlib, and seaborn.
- It is nearly self-contained in that lots of functionality is built into one package. This contrasts with R, where many packages are needed to obtain the same functionality.
- The community is excellent. Looking at Stack Overflow, for example, there are [many ten's of thousands of](#) pandas questions. If you need help, you are nearly guaranteed to find it quickly.

Why is it named after an East Asian bear?

The pandas library was begun by Wes McKinney in 2008 while working at the hedge fund AQR. In the financial world, it is common to refer to tabular data as ‘panel data’ which smashed together becomes pandas. If you are really interested in the history, hear it from the creator [himself](#).

Python already has data structures to handle data, why do we need another one?

Even though Python is a high-level language, its primary built-in data structure to contain a sequence of values, lists, are not built for scientific computing. Lists are a general purpose data structure that can store any object of any type and are not optimized for tabular data analysis. Python lacks a built-in data structure that contains homogeneous data types for fast numerical computation. This data structure, usually referred to as an ‘array’ in most languages, is provided by the numpy third-party library.

pandas is built directly on numpy

`numpy` (‘numerical Python’) is the most popular third-party Python library for scientific computing and forms the foundation for dozens of others, including pandas. `numpy`’s primary data structure is an n-dimensional array which is much more powerful than a Python list and with much better performance for numerical operations.

All of the data in pandas is stored in numpy arrays. That said, it isn’t necessary to know much about numpy when learning pandas. You can think of pandas as a higher-level, easier to use interface for doing data analysis than numpy. It is a good idea to eventually learn numpy, but for most tasks, pandas will be the right tool.

1.1 pandas operates on tabular data

There are numerous formats for data, such as XML, JSON, CSV, Parquet, raw bytes, and many others. pandas has the capability to read in many different formats of data and always converts it to tabular form. pandas is built just for analyzing this rectangular, deceptively normal concept of data. pandas is not a suitable library for handling data in more than two-dimensions. Its focus is strictly on data that is one or two dimensions.

The DataFrame and Series

The DataFrame and Series are the two primary pandas objects that we use throughout this book.

- **DataFrame** - A two-dimensional data structure that looks like any other rectangular table of data with rows and columns.
- **Series** - A single dimension of data. It is analogous to a single column of data or a one-dimensional array.

1.2 pandas examples

The rest of this chapter contains examples of common data analysis tasks with pandas. There are one or two examples from each of the following major areas of the library:

- Reading data
- Filtering data
- Aggregating methods
- Non-Aggregating methods
- Aggregating within groups
- Tidying data

- Joining data
- Time series analysis
- Visualization

The goal is to give you a broad overview of what pandas is capable of doing. You are not expected to understand the syntax, but rather, get a few ideas of what you can expect to accomplish when using pandas. Explanations will be brief, but hopefully provide just enough information so that you can understand the end result.

The `head` method

Many of the last lines of code end with the `head` method. By default, this method returns the first five rows of the DataFrame or Series that calls it. The purpose of this method is to limit the output so that it easily fits on a screen or page in a book. If the `head` method is not used, then pandas displays the first and last 5 rows of data by default (or all the rows if the DataFrame has 60 or less). To reduce output even further (to save space on the screen), an integer (usually 3) will be passed to the `head` method. This integer controls the number of rows returned.

1.3 Which pandas version to use?

The pandas library is under constant development and regularly releases new versions. Currently, pandas is on version 1.0, which was released in January, 2020. Before version 1.0, pandas was on 0.25. Python libraries use the form `a.b.c` for version numbering where `a` represents the **major** version number. It is increased whenever there are major changes, with some being backward incompatible. `b` represents the **minor** version number and is incremented for smaller backward-compatible changes and enhancements. `c` represents the **micro** version number and is incremented mainly for bug fixes.

Often, only the major and minor version are written when speaking about the version of pandas as the micro version isn't all that important. pandas has a history of releasing around two or three minor versions per year. In order to run all of the code in this book, you need to be running **pandas 1.0**.

Import pandas and verify version number

Let's import pandas and verify it's version by accessing the special attribute `__version__`. If you are running any version of pandas less than 1.0 (0.25 or below), then you'll need to exit the Jupyter Notebook, return to the command line and run `conda update pandas`.

```
[1]: import pandas as pd  
pd.__version__
```

```
[1]: '1.0.2'
```

1.4 Reading data

Multiple datasets are used during the rest of this chapter. The `read_csv` function is able to read in text data that is separated by a delimiter. By default, the delimiter is a comma. Below, we read in public bike usage data from the city of Chicago into a pandas DataFrame named `bikes`.

```
[2]: bikes = pd.read_csv('../data/bikes.csv')  
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

1.5 Filtering data

pandas can filter the rows of a DataFrame based on whether the values in that row meet a condition. For instance, we can select only the rides that had a `tripduration` greater than 5,000 (seconds).

Single Condition

This example is a single condition that gets tested for each row. Only the rows that meet this condition are returned.

```
[3]: filt = bikes['tripduration'] > 5000
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy

Multiple Conditions

We can test for multiple conditions in a single row. The following example returns rides by females **and** have a `tripduration` greater than 5,000.

```
[4]: filt1 = bikes['tripduration'] > 5000
filt2 = bikes['gender'] == 'Female'
filt = filt1 & filt2
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy
1954	1103416	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	...	44.1	10.0	12.7	-9999.0	clear

The next example has multiple conditions but only requires that one of the conditions is true. It returns all the rows where either the rider is female **or** the tripduration is greater than 5,000.

```
[5]: filt = filt1 | filt2
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
9	23558	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	...	81.0	10.0	12.7	-9999.0	mostlycloudy
14	31121	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	...	82.0	10.0	5.8	-9999.0	mostlycloudy
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy

Using the query method

The `query` method provides an alternative and often more readable way to filter data than the above. All three filtering examples from above may be duplicated with `query`. A string representing the condition is passed to the `query` method to filter the data.

```
[6]: bikes.query('tripduration > 5000').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy

```
[7]: bikes.query('tripduration > 5000 and gender=="Female"').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy
1954	1103416	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	...	44.1	10.0	12.7	-9999.0	clear

```
[8]: bikes.query('tripduration > 5000 or gender=="Female"').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
9	23558	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	...	81.0	10.0	12.7	-9999.0	mostlycloudy
14	31121	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	...	82.0	10.0	5.8	-9999.0	mostlycloudy
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy

1.6 Aggregating methods

The technical definition of an **aggregation** is when a sequence of values is summarized by a **single** number. For example, `sum`, `mean`, `median`, `max`, and `min` are all examples of aggregation methods. By default, calling these methods on a pandas DataFrame applies the aggregation to each column. Below, we use a dataset containing San Francisco employee compensation information. Only a subset of the columns are initially read into the DataFrame.

```
[9]: cols = ['salaries', 'overtime', 'other salaries', 'retirement', 'health and dental']
sf_emp = pd.read_csv('../data/sf_employee_compensation.csv', usecols=cols)
sf_emp.head(3)
```

	salaries	overtime	other salaries	retirement	health and dental
0	71414.01	0.00	0.0	14038.58	12918.24
1	67941.06	0.00	0.0	13030.23	10047.52
2	116956.72	59975.43	19037.3	24796.44	15788.97

Calling the `mean` method returns the mean of each column. The result is then rounded to the nearest thousand.

```
[10]: sf_emp.mean()
```

salaries	53715.441133
overtime	4201.272687
other salaries	2816.296542
retirement	10484.755614

```
health and dental    9382.390735
dtype: float64
```

pandas allows you to aggregate rows as well. The `axis` parameter may be used to change the direction of the aggregation. This returns the total compensation for each employee.

```
[11]: sf_emp.sum(axis=1).head(3)
```

```
[11]: 0    98370.83
1    91018.81
2   236554.86
dtype: float64
```

1.7 Non-aggregating methods

There are methods that perform some calculation on the DataFrame that do not aggregate the data and usually preserve the shape of the DataFrame. For example, the `round` method rounds each number to a given decimal place. Here, we round each value in the DataFrame to the nearest thousand.

```
[12]: sf_emp.round(-3).head(3)
```

	salaries	overtime	other salaries	retirement	health and dental
0	71000.0	0.0	0.0	14000.0	13000.0
1	68000.0	0.0	0.0	13000.0	10000.0
2	117000.0	60000.0	19000.0	25000.0	16000.0

1.8 Aggregating within groups

Above, we performed aggregations on the entire DataFrame. We can instead perform aggregations within groups of the data. Below we use an insurance dataset.

```
[13]: ins = pd.read_csv('../data/insurance.csv')
ins.head(3)
```

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.90	0	yes	southwest	16884.9240
1	18	male	33.77	1	no	southeast	1725.5523
2	28	male	33.00	3	no	southeast	4449.4620

One of the simplest aggregations is the frequency of occurrence of all the unique values within a single column. This is performed below with the `value_counts` method.

Frequency of unique values in a single column

Here, we count the occurrence of each individual `region`.

```
[14]: ins['region'].value_counts()
```

```
[14]: southeast    364
northwest     325
southwest     325
northeast     324
Name: region, dtype: int64
```

Single aggregation function

Let's say we wish to find the mean charges for each of the unique values in the `sex` column. The `groupby` method creates groups based on the given grouping column before applying the aggregation. In this example, we return the mean charges for each sex.

```
[15]: ins.groupby('sex').agg(mean_charges=('charges', 'mean')).round(-3)
```

mean_charges	
sex	
female	13000.0
male	14000.0

Multiple aggregation functions

pandas allows us to perform multiple aggregations at the same time. Below, we calculate the mean and max of the `charges` column as well as count the number of non-missing values.

```
[16]: ins.groupby('sex').agg(mean_charges=('charges', 'mean'),
                           max_charges=('charges', 'max'),
                           count_charges=('charges', 'count')).round(0)
```

	mean_charges	max_charges	count_charges
sex			
female	12570.0	63770.0	662
male	13957.0	62593.0	676

Multiple grouping columns

pandas allows us to form groups based on multiple columns. In the below example, each unique combination of `sex` and `region` form a group. For each of these groups, the same aggregations as above are performed on the `charges` column.

```
[17]: ins.groupby(['sex', 'region']).agg(mean_charges=('charges', 'mean'),
                                         max_charges=('charges', 'max'),
                                         count_charges=('charges', 'count')).round(0)
```

			mean_charges	max_charges	count_charges
	sex	region			
female		northeast	12953.0	58571.0	161
		northwest	12480.0	55135.0	164
		southeast	13500.0	63770.0	175
		southwest	11274.0	48824.0	162
male		northeast	13854.0	48549.0	163
		northwest	12354.0	60021.0	161
		southeast	15880.0	62593.0	189
		southwest	13413.0	52591.0	163

Pivot Tables

We can reproduce the exact same output as above in a different shape with the `pivot_table` method. It groups and aggregates the same way as `groupby`, but places the unique values of one of the grouping columns as the new columns in the resulting DataFrame. Notice that pivot tables make for easier comparisons across groups.

```
[18]: pt = ins.pivot_table(index='sex', columns='region',
                           values='charges', aggfunc='mean').round(0)
pt
```

	region	northeast	northwest	southeast	southwest
	sex				
female		12953.0	12480.0	13500.0	11274.0
		13854.0	12354.0	15880.0	13413.0

Styling DataFrames

pandas enables you to style DataFrames in various ways to provide emphasis on particular cells. Below, the maximum value of each column is highlighted, a comma is added to separate the digits, and decimals are removed.

```
[19]: pt.style.highlight_max().format(r'{:,.0f}')
```

	region	northeast	northwest	southeast	southwest
	sex				
female		12,953	12,480	13,500	11,274
		13,854	12,354	15,880	13,413

1.9 Tidying

Many datasets need to be cleaned and tidied before analyzed. pandas provides many tools to prepare data for further analysis.

Options in the `read_csv` function

Below, we read in a new dataset on plane crashes. Notice all the question marks. They represent missing values, but pandas will read them in as strings.

```
[20]: pc = pd.read_csv('../data/tidy/planecrashinfo.csv')
pc.head(3)
```

	date	time	location	operator	flight_no	...	cn_ln	aboard	fatalities	ground	summary
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	?	...	1	(passenger:1 crew:1)	2	1	During a demonstration flight, a U.S. Army fly...
1	September 07, 1909	?	Juvisy-sur-Orge, France	?	?	...	?	(passenger:0 crew:1)	1	1	Eugene Lefebvre was the first pilot to ever be...
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	?	...	?	(passenger:0 crew:5)	5	5	First U.S. dirigible Akron exploded just offsh...

The `read_csv` has dozens of options to help read in messy data. One of the options allows you to convert a particular string to missing values. Notice that all of the question marks are now labeled as `NaN` (not a number).

```
[21]: pc = pd.read_csv('../data/tidy/planecrashinfo.csv', na_values='?')
pc.head(3)
```

	date	time	location	operator	flight_no	...	cn_ln	aboard	fatalities	ground	summary
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	NaN	...	1	(passenger:1 crew:1)	2	1	During a demonstration flight, a U.S. Army fly...
1	September 07, 1909	NaN	Juvisy-sur-Orge, France	NaN	NaN	...	NaN	(passenger:0 crew:1)	1	1	Eugene Lefebvre was the first pilot to ever be...
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	NaN	...	NaN	(passenger:0 crew:5)	5	5	First U.S. dirigible Akron exploded just offsh...

String manipulation

Often times there is data stuck within a string column that you will need to extract. The `aboard` column appears to have three distinct pieces of information; the total number of people on board, the number of

passengers, and the number of crew.

```
[22]: aboard = pc['aboard']
aboard.head()
```

```
[22]: 0      2  (passenger:1 crew:1)
1      1  (passenger:0 crew:1)
2      5  (passenger:0 crew:5)
3      1  (passenger:0 crew:1)
4     20  (passenger:? crew:?)
Name: aboard, dtype: object
```

pandas has special functionality for manipulating strings. Below, we use a regular expression to extract the pertinent numbers from the `aboard` column.

```
[23]: aboard.str.extract(r'(\d+)?\D*(\d+)?\D*(\d+)?') .head()
```

	0	1	2
0	2	1	1
1	1	0	1
2	5	0	5
3	1	0	1
4	20	NaN	NaN

Reshaping into tidy form

Occasionally, you will have several columns of data that all belong in a single column. Take a look at the DataFrame below on the average arrival delay of airlines at different airports. All of the columns with three-letter airport codes could be placed in the same column as they all contain the arrival delay which has the same units.

```
[24]: aad = pd.read_csv('../data/tidy/average_arrival_delay.csv').head()
aad
```

	airline	ATL	DEN	DFW	IAH	...	LAX	MSP	ORD	PHX	SFO
0	AA	4.0	9.0	5.0	11.0	...	3.0	1.0	8.0	5.0	3.0
1	AS	6.0	-3.0	-5.0	1.0	...	-3.0	6.0	2.0	-9.0	4.0
2	B6	NaN	12.0	4.0	NaN	...	2.0	NaN	23.0	20.0	5.0
3	DL	0.0	-3.0	10.0	3.0	...	3.0	-1.0	7.0	-4.0	0.0
4	EV	7.0	14.0	10.0	3.0	...	NaN	10.0	8.0	-14.0	NaN

The `melt` method stacks columns one on top of the other. Here, it places all of the three-letter airport code columns into a single column. The first two airports (ATL and DEN) are shown below in the new tidy DataFrame.

```
[25]: aad.melt(id_vars='airline', var_name='airport', value_name='delay').head(10)
```

	airline	airport	delay
0	AA	ATL	4.0
1	AS	ATL	6.0
2	B6	ATL	NaN
3	DL	ATL	0.0
4	EV	ATL	7.0
5	AA	DEN	9.0
6	AS	DEN	-3.0
7	B6	DEN	12.0
8	DL	DEN	-3.0
9	EV	DEN	14.0

1.10 Joining Data

pandas can join multiple DataFrames together by matching values in one or more columns. If you are familiar with SQL, then pandas performs joins in a similar fashion. Below, we make a connection to a database and read in two of its tables.

```
[26]: from sqlalchemy import create_engine
engine = create_engine('sqlite:///../data/databases/neurIPS.db')
authors = pd.read_sql('Authors', engine)
pa = pd.read_sql('PaperAuthors', engine)
```

Output the first three rows of each DataFrame.

```
[27]: authors.head(3)
```

	Id	Name
0	178	Yoshua Bengio
1	200	Yann LeCun
2	205	Avrim Blum

```
[28]: pa.head(3)
```

	Id	PaperId	AuthorId
0	1	5677	7956
1	2	5677	2649
2	3	5941	8299

We can now join these tables together using the `merge` method. The `AuthorID` column from the `pa` table is aligned with the `Id` column of the `authors` table.

```
[29]: pa.merge(authors, how='left', left_on='AuthorId', right_on='Id').head(3)
```

	Id_x	PaperId	AuthorId	Id_y	Name
0	1	5677	7956	7956	Nihar Bhadresh Shah
1	2	5677	2649	2649	Denny Zhou
2	3	5941	8299	8299	Brendan van Rooyen

1.11 Time Series Analysis

One of the original purposes of pandas was to do time series analysis. Below, we read in 20 years of Microsoft's closing stock price data.

```
[30]: msft = pd.read_csv('../data/stocks/msft20.csv', parse_dates=['date'], index_col='date')
msft.head()
```

	open	high	low	close	adjusted_close	volume	dividend_amount
	date						
1999-10-19	88.250	89.250	85.250	86.313	27.8594	69945600	0.0
1999-10-20	91.563	92.375	90.250	92.250	29.7758	88090600	0.0
1999-10-21	90.563	93.125	90.500	93.063	30.0381	60801200	0.0
1999-10-22	93.563	93.875	91.750	92.688	29.9171	43650600	0.0
1999-10-25	92.000	93.563	91.125	92.438	29.8364	30492200	0.0

Select a period of time

pandas allows us to easily select a period of time. Below, we select all of the trading data from February 27, 2017 through March 2, 2017.

```
[31]: msft['2017-02-27':'2017-03-02']
```

	open	high	low	close	adjusted_close	volume	dividend_amount
	date						
2017-02-27	64.54	64.54	64.045	64.23	61.4355	15871500	0.0
2017-02-28	64.08	64.20	63.760	63.98	61.1964	23239800	0.0
2017-03-01	64.13	64.99	64.022	64.94	62.1146	26937500	0.0
2017-03-02	64.69	64.75	63.880	64.01	61.2251	24539600	0.0

Group by time

We can group by some length of time. Here, we group together every month of trading data and return the average closing price of that month.

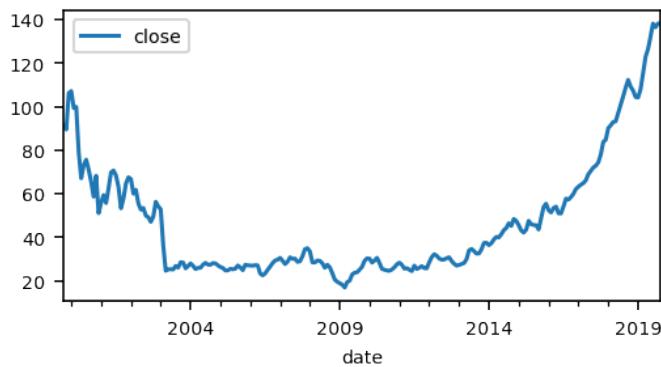
```
[32]: msft_mc = msft.resample('M').agg({'close':'mean'})
msft_mc.head(3)
```

close	
	date
1999-10-31	91.382222
1999-11-30	89.463762
1999-12-31	106.190545

1.12 Visualization

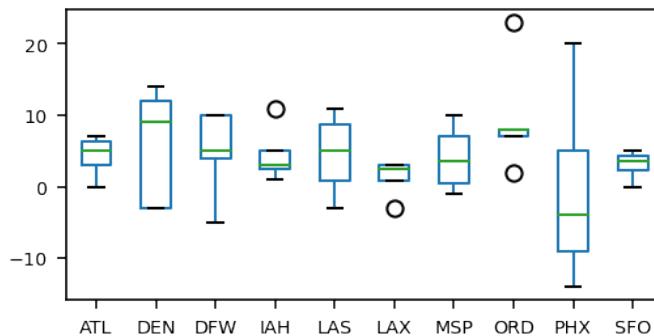
pandas provides basic visualization abilities by giving its users a few default plots. Below, we plot the average monthly closing price of Microsoft for the last 20 years.

```
[33]: import matplotlib.pyplot as plt
plt.style.use('.../.../mdap.mplstyle')
msft_mc.plot(kind='line');
```



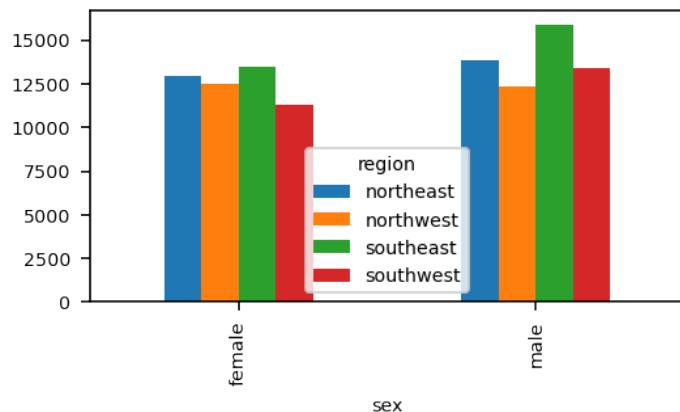
Below, we create a box plot of the average arrival delay by airport.

```
[34]: aad.plot(kind='box');
```



The pivot table of average insurance cost by region and sex is made into a bar graph.

```
[35]: pt.plot(kind='bar');
```



1.13 Much More

The above was a small sample from many of the major sections of the pandas library. The rest of the book focuses on going into great detail on how to effectively use the pandas library to complete nearly any kind of data analysis.

Chapter 2

The DataFrame and Series

The DataFrame and Series are the two primary pandas objects used throughout this book. In this chapter, we will learn how to read in data into a DataFrame and understand its components. We will also learn how to select a single column of data as a Series to understand its components.

2.1 Reading external data with pandas

The one thing you need for data analysis is **data**. If you do not have any data, then you won't be able to use pandas to analyze it. This book contains many data sets stored externally in the **data** directory one level above where this notebook resides. Most of these datasets are stored as comma-separated value (**CSV**) files. These CSVs are human-readable and separate each individual piece of data with a comma. The comma is referred to as the **delimiter**. Despite its name, CSVs can use other one-character delimiters besides commas.

City of Chicago bike rides

We begin our data analysis adventure with a dataset on public bike rides from the city of Chicago. The data is contained in the **bikes.csv** file. There are about 50,000 recorded rides from 2013 through 2017. Each row of the dataset represents a single ride from a single person using the city's public bike stations. There are 19 columns of data containing information on gender, start time, trip duration, bike station name, temperature, wind speed, and more. Let's print out the first three lines of the **bikes.csv** file using Python's built-in capabilities for reading files. This does not use pandas. Take note of the commas separating each value on each line.

```
[1]: with open('../data/bikes.csv') as f:  
    for i in range(3):  
        print(f.readline())
```

```
trip_id,gender,starttime,stoptime,tripduration,from_station_name,dpcapacity_start,to_station_name,dpcapacity_end,temperature,visibility,wind_speed,precipitation,events
```

```
7147,Male,2013-06-28 19:01:00,2013-06-28 19:17:00,993,Lake Shore Dr & Monroe  
St,11.0,Michigan Ave & Oak St,15.0,73.9,10.0,12.7,-9999.0,mostlycloudy
```

```
7524,Male,2013-06-28 22:53:00,2013-06-28 23:03:00,623,Clinton St & Washington  
Blvd,31.0,Wells St & Walton St,19.0,69.1,10.0,6.9,-9999.0,partlycloudy
```

Understanding the file location

Above, the string `../data/bikes.csv` was used to represent the file location of the data. This location is relative to the directory where this notebook resides on your machine. Let's cover every part of this string to ensure we understand what it means.

The file location string begins with two dots, `..`. This translates as 'move one level above the current working directory' to the **Jupyter Notebooks** directory. Appearing next in the string is `/data`, which translates as 'move down into the `data` directory'.

Note that the forward slash was written to separate the directories. Both macOS and Linux operating systems use this forward slash to separate directories and files from one another. On the other hand, the Windows operating system uses the backslash. Fortunately, we can always use a forward slash regardless of our operating system, as Python will automatically handle the file location string for us.

The string ends with `/bikes.csv` which translates as 'reference the filename `bikes.csv`'. In summary, the file location `../data/bikes.csv` represents a relative location to where the dataset resides.

Import pandas

To use the pandas library, we need to import it into our namespace. By convention, pandas is imported and aliased to the name `pd`. After running the import statement below, we will have access to all pandas objects with variable name `pd`. It is possible to use any other valid variable name as an alias, but it's best to use `pd` as the official documentation uses it along with most everyone else.

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

Reading a CSV with the `read_csv` function

pandas provides the `read_csv` function to read in CSV files into a pandas DataFrame. The first argument passed to `read_csv` needs to be the location of the file relative to the current directory as a string, which is the same string from above. Let's call this function to read in our data.

```
[3]: bikes = pd.read_csv('..../data/bikes.csv')
```

Display DataFrame in Jupyter Notebook

We assigned the output from the `read_csv` function to the `bikes` variable name which now refers to a DataFrame object. To visually display the DataFrame, place the variable name as the last line in a code cell. By default, pandas outputs the first and last 5 rows and first and last 10 columns. If there are less than 60 total rows, it displays all rows. We cover how to change these display options in an upcoming chapter.

head and tail methods

A very useful and simple method is `head`, which returns the first 5 rows of the DataFrame by default. This avoids long default output and is something I highly recommend when doing data analysis within a notebook. The `tail` method returns the last 5 rows by default. There will only be a few instances in the book where the `head` method is not used, as displaying up to 60 rows is far too many and will take up a lot of space on a screen or page.

```
[4]: bikes.head()
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
3	12907	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...	72.0	10.0	16.1	-9999.0	mostlycloudy
4	13168	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130	...	73.0	10.0	17.3	-9999.0	partlycloudy

The last five rows of the DataFrame may be displayed with the `tail` method.

[5] : `bikes.tail()`

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
50084	17534938	Male	2017-12-30 13:07:00	2017-12-30 13:34:00	1625	...	5.0	10.0	16.1	-9999.0	partlycloudy
50085	17534969	Male	2017-12-30 13:34:00	2017-12-30 13:44:00	585	...	5.0	10.0	16.1	-9999.0	partlycloudy
50086	17534972	Male	2017-12-30 13:34:00	2017-12-30 13:48:00	824	...	5.0	10.0	16.1	-9999.0	partlycloudy
50087	17535645	Female	2017-12-31 09:30:00	2017-12-31 09:33:00	178	...	7.0	10.0	11.5	-9999.0	partlycloudy
50088	17536246	Male	2017-12-31 15:22:00	2017-12-31 15:26:00	214	...	10.9	10.0	15.0	-9999.0	partlycloudy

First and last n rows

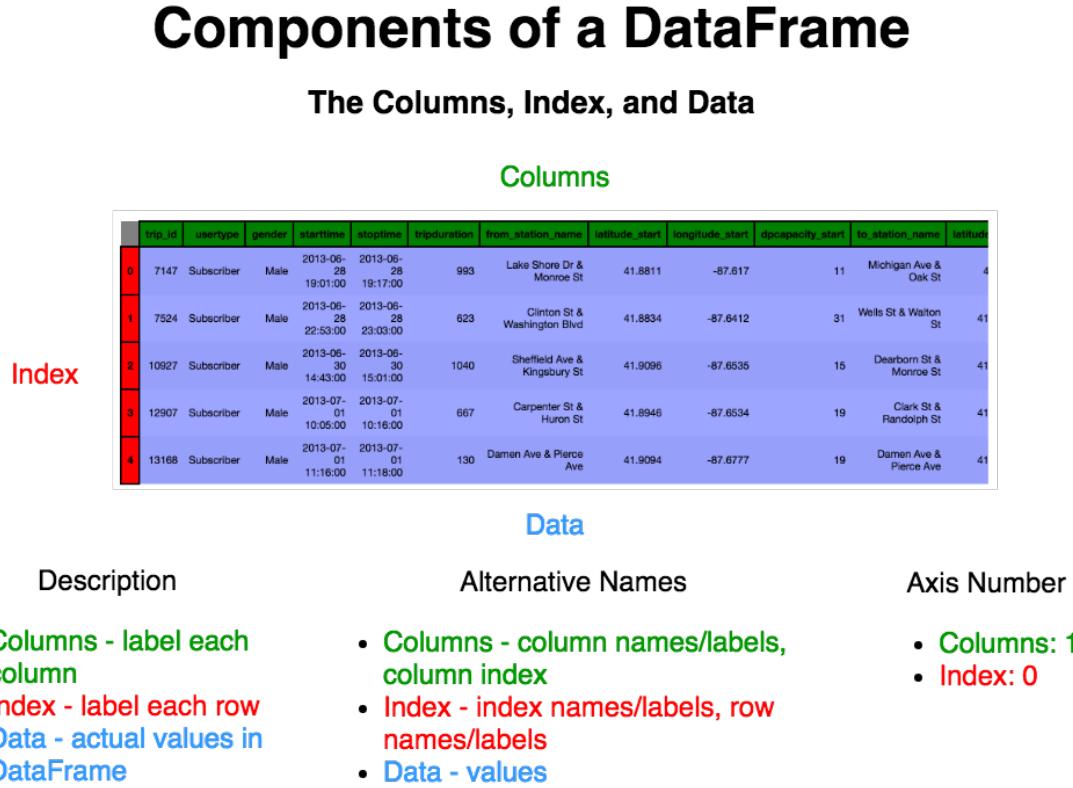
Both the `head` and `tail` methods accept a single integer parameter `n` controlling the number of rows returned. Here, we output the first three rows.

[6] : `bikes.head(3)`

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

2.2 Components of a DataFrame

The DataFrame is composed of three separate components - the **columns**, the **index**, and the **data**. These terms will be used throughout the book and understanding them is vital to your ability to use pandas. Take a look at the following graphic of our `bikes` DataFrame stylized to put emphasis on each component.



The columns

The columns provide a **label** for each column and are always displayed in **bold** font above the data. A column is a single vertical sequence of data. In the above DataFrame, the column name `tripduration` references all the values in that column (993, 623, 1040, etc...).

The columns are also referred to as the **column names** or the **column labels** with individual values referred to as a **column name** or **column label**.

Most DataFrames, like the one above, use strings for column names, but it is possible that they can be other types such as integers. The column names are not required to be unique, though having duplicate columns would be bad practice, as it's vital to be able to uniquely identify each column.

The index

The index provides a **label** for each row and is always displayed to the left of the data. A row is a single horizontal sequence of data. For instance, the index label **3** references all the values in its row (12907, Subscriber, Male, etc...)

The index is also referred to as the **index names/labels** or the **row names/labels** with the individual values referred to as a(n) **index name/label** or **row name/label**.

In the above DataFrame, the index is simply a sequence of integers beginning at 0. The values in the index are not limited to integers. Strings are a common type that are used in the index and make for more descriptive labels.

Surprisingly, values in the index are not required to be unique. In fact, all of the index values can be the same. A row label does not guarantee a one-to-one mapping to one specific row.

The data

The actual data is to the right of the index and below the columns and is displayed with normal font. The data is also referred to as the **values**. The data represents all the values for all the columns. It is important to note that the index and the columns are NOT part of the data. They are separate objects that act as **labels** for either rows or columns.

The Axes

The index and columns are known collectively as the **axes**, each representing a single **axis** of the two-dimensional DataFrame. pandas uses the integer **0** to reference the index and **1** for the columns.

2.3 What type of object is bikes?

Let's verify that `bikes` is indeed a DataFrame with the `type` function.

```
[7]: type(bikes)
```

```
[7]: pandas.core.frame.DataFrame
```

Fully-qualified name

The above output is something called the **fully-qualified name**. Only the word after the last dot is the name of the type. We have now verified that the `bikes` variable has type `DataFrame`.

The fully-qualified name always returns the package and module name of where the type was defined. The package name is the first part of the fully-qualified name and, in this case, is `pandas`. The module name is the word immediately preceding the name of the type. Here, it is `frame`.

Package vs Module

A Python **package** is a directory containing other directories or modules that contain Python code. A Python **module** is a file (typically a text file ending in `.py`) that contains Python code.

Sub-packages

Any directory containing other directories or modules within a Python package is considered a **sub-package**. In this case, `core` is the sub-package.

Where are the packages located on my machine?

Third-party packages are installed in the `site-packages` directory which itself is set up during Python installation. We can get the actual location with help from the standard library's `site` module's `getsitepackages` function.

```
[8]: import site  
site.getsitepackages()
```

```
[8]: ['/Users/Ted/miniconda3/envs/minimal_ds/lib/python3.7/site-packages']
```

If you navigate to this directory in your file system, you'll find the 'pandas' directory. Within it will be a 'core' directory which will contain the 'frame.py' file. It is this file which contains Python code where the DataFrame class is defined.

2.4 Select a single column from a DataFrame - a Series

To select a single column from a DataFrame, append a set of square brackets, [], to the end of the DataFrame variable name. Place the column name as a string within those brackets to select it. This returns a single column of data as a pandas **Series**. This is a separate (but similar) type of object than a DataFrame.

Let's select the column name `tripduration`, assign it to a variable name, and output the first few values to the screen. The `head` and `tail` methods work the same as they do with DataFrames.

```
[9]: td = bikes['tripduration']
td.head()
```

```
[9]: 0      993
1      623
2     1040
3      667
4      130
Name: tripduration, dtype: int64
```

Select the last three values in the Series by passing the `tail` method the integer 3.

```
[10]: td.tail(3)
```

```
[10]: 50086    824
50087    178
50088    214
Name: tripduration, dtype: int64
```

Let's verify that `td` has the type Series.

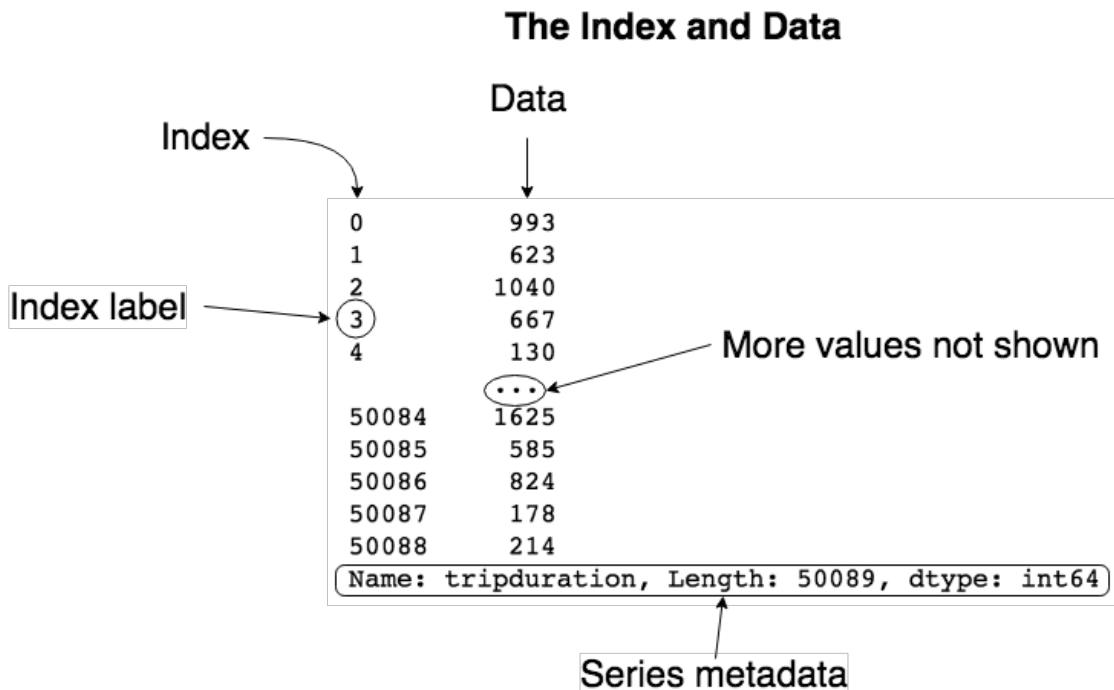
```
[11]: type(td)
```

```
[11]: pandas.core.series.Series
```

2.5 Components of a Series

A Series is a similar type of object as a DataFrame but only contains a single dimension of data. It has two components - the **index** and the **data**. Let's take a look at a stylized Series graphic.

Components of a Series



It's important to note that a Series has no rows and no columns. In appearance, it resembles a one-column DataFrame, but it technically has no columns. It just has a sequence of values that are labeled by an index.

The index

A Series index serves as labels for the values. A single **label** or **name** always references a single value. In the above image, the index label **3** corresponds to the value **667**. The Series index is virtually identical to the DataFrame index, so the same rules apply to it. Index values can be duplicated and can be types other than integers, such as strings.

Output of Series vs DataFrame

Notice that there is no nice HTML styling for the Series. It's just plain text. Below the Series display, you will see a few other items printed to the screen - the **name**, **length**, and **dtype**. These other items are NOT part of the Series itself and are just extra pieces of information to help you understand the Series.

- The **name** is not important right now. If the Series was formed from a column of a DataFrame, it will be set to that column name.
- The **length** is the number of values in the Series
- The **dtype** is the data type of the Series, which will be discussed in an upcoming chapter.

2.6 Changing display options

pandas gives you the ability to change how the output on your screen is displayed. For instance, the default number of columns displayed for a DataFrame is 20, meaning that if your DataFrame has more than 20 columns, then only the first and last 10 columns will be shown on the screen. All the other columns will be hidden and unable to be displayed. This is problematic as many DataFrames have more than 20 columns.

Get current option value with `get_option`

There are a few dozen display options you can control to change the visual representation of your DataFrame. It is not necessary to remember the option names as the official documentation provides descriptions for all available options.

Let's first learn how to retrieve each option value with the `get_option` function. This is not a DataFrame method, but instead, a function that is accessed directly from `pd`. Below are three of the most common options to change.

```
[12]: pd.get_option('display.max_columns')
```

```
[12]: 20
```

```
[13]: pd.get_option('display.max_rows')
```

```
[13]: 60
```

```
[14]: pd.get_option('display.max_colwidth')
```

```
[14]: 50
```

Use the `set_option` function to change an option value

To change the option value, use the `set_option` function. You can set as many options as you would like at one time. Its usage is a bit strange. Pass it the option name as a string and follow it immediately with the value you want to set it to. Continue this pattern of option name followed by new value to set as many options as you desire. Below, we set the maximum number of columns to 100 and the maximum number of rows to 4.

```
[15]: pd.set_option('display.max_columns', 100, 'display.max_rows', 4)
```

We now read in the housing dataset which contains 81 columns, all of which will be visible.

```
[16]: housing = pd.read_csv('../data/housing.csv')
housing
```

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	...	MoSold	YrSold	SaleType	SaleCondition	SalePrice
0	1	60	RL	65.0	8450	...	2	2008	WD	Normal	208500
1	2	20	RL	80.0	9600	...	5	2007	WD	Normal	181500
...
1458	1459	20	RL	68.0	9717	...	4	2010	WD	Normal	142125
1459	1460	20	RL	75.0	9937	...	6	2008	WD	Normal	147500

2.7 Exercises

Use the `bikes` DataFrame for the following exercises.

Exercise 1

Select the column `events`, the type of weather that was recorded, and assign it to a variable with the same name. Output the first 10 values of it.

[]:

Exercise 2

What type of object is `events`?

[]:

Exercise 3

Select the last two rows of the `bikes` DataFrame and assign it to the variable `bikes_last_2`. What type of object is `bikes_last_2`?

[]:

Exercise 4

Use `pd.reset_option('all')` to reset the options to their default values. Test that this worked.

[]:

Chapter 3

Data Types and Missing Values

One of the most important pieces of information you can have about your DataFrame is the data type of each column. pandas stores its data such that each column is exactly one data type. A large number of data types are available for pandas DataFrame columns. This chapter focuses only on the most common data types and provides a brief summary of each one. For extensive coverage of each and every data type, see the part **05. Data Types**.

3.1 Common data types

The following are the most common data types that appear frequently in DataFrames.

- **boolean** - Only two possible values, `True` and `False`
- **integer** - Whole numbers without decimals
- **float** - Numbers with decimals
- **object** - Almost always strings, but can technically contain any Python object
- **datetime** - Specific date and time with nanosecond precision

The importance of knowing the data type

Knowing the data type of each column of your pandas DataFrame is very important. The main reason for this is that every value in each column will be of the same type. For instance, if you select a single value from a column that has an integer data type, then you are guaranteed that this value is also an integer. Knowing the data type of a column is one of the most fundamental pieces of knowledge of your DataFrame.

The exception with the object data type

The object data type is the most confusing and deserves a longer discussion. It is an exception to the message in the last section. Each value in an object column can be any Python object. Object columns can contain integers, floats, or even data structures such as lists or dictionaries. Anything can be contained in object columns. But, nearly all of the time, columns with the object data type only contain strings. When you see a column with the object data type, you should expect the values to be strings. If you do have strings in your columns, the data type will be object, but you are not guaranteed that all values will be strings.

3.2 String data type - major enhancement to pandas 1.0

Before the release of pandas version 1.0, there was no dedicated string data type. This was a huge limitation and caused numerous problems. pandas still has the ‘object’ data type, which is capable of holding strings. But, the object data type can hold **ANY** Python object regardless of its type. Each value in an object column can be of a different type. This is NOT a good thing.

With the addition of the string data type, we are guaranteed that every value will be a string in a column with string data type. This new data type is still labeled as “experimental” in the pandas documentation, so I do not suggest using it for serious work yet. There are many bugs that need to be fixed and behavior sorted out before it is ready to use. Until then, this book will continue to use the object data type for columns containing strings.

3.3 Missing value representation

Datasets often have missing values and need to have some representation to identify them. Pandas uses the object `NaN` and `NaT` to represent them.

- `NaN` - “Not a Number”
- `NaT` - “Not a Time”

Missing values for each data type

The missing value representation depends on the data type of the column. For our common data types, we have the following missing value representation for each.

- `boolean` - No missing value representation
- `integer` - No missing value representation
- `float` - `NaN`
- `object` - `NaN`
- `datetime` - `NaT`

Missing values in boolean and integer columns

Knowing that a column is either a boolean or integer column guarantees that there are no missing values in that column, as pandas does not allow for it. If, for instance, you would like to place a missing value in a boolean or integer column, then pandas would convert the entire column to float. This is because a float column can accommodate missing values. When booleans are converted to floats, `False` becomes 0 and `True` becomes 1.

3.4 New Integers and booleans data types in pandas 1.0

Two new data types, the `nullable integer` and `nullable boolean` are now available in pandas 1.0. These are completely different data types than the original integer and boolean data types and have slightly different behavior. The main difference is that they do have missing value representation.

Pandas NA - A new missing value representation for pandas 1.0

Previously, pandas relied on the numpy library to supply its primary missing value, `NaN`, which continues to exist. With the release of version 1.0, pandas created its own missing value representation, `NA`. This is a new and experimental addition, so its behavior can change.

3.5 Recommendation for Pandas 1.0 - Avoid the new data types

I recommend not using the new string, nullable integer, and nullable boolean data types along with the pandas `NA` until there has been more development with them. They are still experimental and their behavior can change. I’ve personally found several bugs and strange behavior using them and would wait until they are more stable. There will be a chapter dedicated to these new data types in part **05. Data Types** with more information.

3.6 Finding the data type of each column

The `dtypes` DataFrame attribute (NOT a method) returns the data type of each column and is one of the first commands you should execute after reading in your data. Let's begin by using the `read_csv` function to read in the bikes dataset.

```
[1]: import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Let's get the data types of each column in our `bikes` DataFrame. This returns a Series object with the data types as the values and the column names as the index.

```
[2]: bikes.dtypes
```

```
[2]: trip_id          int64
gender           object
starttime        object
stoptime         object
tripduration     int64
from_station_name   object
dpcapacity_start    float64
to_station_name    object
dpcapacity_end      float64
temperature       float64
visibility        float64
wind_speed         float64
precipitation      float64
events            object
dtype: object
```

Object data types hold string columns

By default, pandas reads in columns containing strings as the object data type. When you see object as the data type, you should think “string”.

The `starttime` and `stoptime` columns are not datetimes

From the visual display of the bikes DataFrame above, it appears that both the `starttime` and `stoptime` columns are datetimes. The result of the `dtypes` attribute shows that they are strings. Unfortunately, the `read_csv` function does not automatically read in these columns as datetimes. It requires that you provide

it a list of columns that are datetimes to the `parse_dates` parameter, otherwise it will read them in as strings. Let's reread the data using the `parse_dates` parameter.

```
[3]: bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])
bikes.dtypes.head()
```

```
[3]: trip_id          int64
gender            object
starttime    datetime64[ns]
stoptime     datetime64[ns]
tripduration      int64
dtype: object
```

What are all those 64's at the end of the data types?

Booleans, integers, floats, and datetimes all use a particular amount of memory for each of their values. The memory is measured in **bits**. The number of bits used for each value is the number appended to the end of the data type name. For instance, integers can be either 8, 16, 32, or 64 bits while floats can be 16, 32, 64, or 128. A 128-bit float column will show up as `float128`.

Technically a `float128` is a different data type than a `float64` but generally you will not have to worry about such a distinction as the operations between different float columns will be the same. Booleans are always stored as 8-bits. There is no set bit size for object columns as each value can be of any size.

3.7 Getting more metadata

Metadata can be defined as data on the data. The data type of each column is an example of metadata. The number of rows and columns is another piece of metadata. We find this with the `shape` attribute, which returns a tuple of integers representing the number of rows and columns of the DataFrame.

```
[4]: bikes.shape
```

```
[4]: (50089, 14)
```

Use the `len` function to get the number of rows

Pass the DataFrame to the built-in `len` function to return the number of rows as an integer.

```
[5]: len(bikes)
```

```
[5]: 50089
```

You can also get the number of rows as an integer by selecting the first item of the tuple return from `shape`. Either way is acceptable.

```
[6]: bikes.shape[0]
```

```
[6]: 50089
```

Similarly, you can get the number of columns as an integer by selecting the second item.

```
[7]: bikes.shape[1]
```

[7]: 14

Total number of values with the `size` attribute

The `size` attribute returns the total number of values (the number of columns multiplied by the number of rows) in the DataFrame.

[8]: `bikes.size`

[8]: 701246

Get data types plus more with the `info` method

The `info` DataFrame method provides output similar to `dtypes`, but also shows the number of non-missing values in each column along with more info such as:

- Type of object (always a DataFrame)
- The type of index and number of rows
- The number of columns
- The data types of each column and the number of non-missing (a.k.a non-null)
- The frequency count of all data types
- The total memory usage

[9]: `bikes.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
 ---  --  
 0   trip_id          50089 non-null   int64  
 1   gender           50089 non-null   object  
 2   starttime        50089 non-null   datetime64[ns]
 3   stoptime         50089 non-null   datetime64[ns]
 4   tripduration     50089 non-null   int64  
 5   from_station_name 50089 non-null   object  
 6   dpcapacity_start 50083 non-null   float64 
 7   to_station_name  50089 non-null   object  
 8   dpcapacity_end   50077 non-null   float64 
 9   temperature       50089 non-null   float64 
 10  visibility        50089 non-null   float64 
 11  wind_speed        50089 non-null   float64 
 12  precipitation     50089 non-null   float64 
 13  events            50089 non-null   object  
dtypes: datetime64[ns](2), float64(6), int64(2), object(4)
memory usage: 5.4+ MB
```

3.8 More data types

There are many more data types available in pandas. An extensive and formal discussion on all data types is available in the part **05. Data Types**.

3.9 Exercises

Use the `bikes` DataFrame for the following:

Exercise 1

What type of object is returned from the `dtypes` attribute?

[]:

Exercise 2

What type of object is returned from the `shape` attribute?

[]:

Exercise 3

What type of object is returned from the `info` method?

[]:

Exercise 4

The memory usage from the `info` method isn't correct when you have objects in your DataFrame. Read the docstrings from it and get the true memory usage.

[]:

Chapter 4

Setting a Meaningful Index

The index of a DataFrame provides a label for each of the rows. If not explicitly provided, pandas uses the sequence of consecutive integers beginning at 0 as the index. In this chapter, we learn how to set one of the columns of the DataFrame as the new index so that it provides a more meaningful label for each row.

4.1 Setting an index of a DataFrame

Instead of using the default index for your pandas DataFrame, you can use the `set_index` method to use one of the columns as the index. Let's read in a small dataset to show how this is done.

```
[1]: import pandas as pd  
df = pd.read_csv('../data/sample_data.csv')  
df
```

	name	state	color	food	age	height	score
0	Jane	NY	blue	Steak	30	165	4.6
1	Niko	TX	green	Lamb	2	70	8.3
2	Aaron	FL	red	Mango	12	120	9.0
3	Penelope	AL	white	Apple	4	80	3.3
4	Dean	AK	gray	Cheese	32	180	1.8
5	Christina	TX	black	Melon	33	172	9.5
6	Cornelia	TX	red	Beans	69	150	2.2

The `set_index` method

Pass the `set_index` method the name of the column to use it as the index. This column will no longer be part of the data of the returned DataFrame.

```
[2]: df.set_index('name')
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

A new DataFrame copy is returned

The `set_index` method returns an entire new DataFrame copy by default, and does not modify the original calling DataFrame. Let's verify this by outputting the DataFrame referenced by `df`. It has not changed.

[3] : df

	name	state	color	food	age	height	score
0	Jane	NY	blue	Steak	30	165	4.6
1	Niko	TX	green	Lamb	2	70	8.3
2	Aaron	FL	red	Mango	12	120	9.0
3	Penelope	AL	white	Apple	4	80	3.3
4	Dean	AK	gray	Cheese	32	180	1.8
5	Christina	TX	black	Melon	33	172	9.5
6	Cornelia	TX	red	Beans	69	150	2.2

Assigning the result of `set_index` to a variable name

We must assign the result of the `set_index` method to a variable name if we are to use this new DataFrame with its new index.

[4] : df2 = df.set_index('name')
df2

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Number of columns decreased

The new DataFrame, `df2`, has one less column than the original as the `name` column was set as the index. Let's verify this by accessing the `shape` attribute of the original and new DataFrames.

```
[5]: df.shape
```

```
[5]: (7, 7)
```

```
[6]: df2.shape
```

```
[6]: (7, 6)
```

4.2 Accessing the index, columns, and data

The index, columns, and data are each separate objects that can be accessed from the DataFrame as attributes and NOT methods. Let's assign each of them to their own variable name beginning with the index and output it to the screen.

```
[7]: index = df2.index
index
```

```
[7]: Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina', 'Cornelia'],
      dtype='object', name='name')
```

```
[8]: columns = df2.columns
columns
```

```
[8]: Index(['state', 'color', 'food', 'age', 'height', 'score'], dtype='object')
```

```
[9]: data = df2.values
data
```

```
[9]: array([['NY', 'blue', 'Steak', 30, 165, 4.6],
       ['TX', 'green', 'Lamb', 2, 70, 8.3],
       ['FL', 'red', 'Mango', 12, 120, 9.0],
```

```
[['AL', 'white', 'Apple', 4, 80, 3.3],
 ['AK', 'gray', 'Cheese', 32, 180, 1.8],
 ['TX', 'black', 'Melon', 33, 172, 9.5],
 ['TX', 'red', 'Beans', 69, 150, 2.2]], dtype=object)
```

Find the type of these objects

The output of these objects looks correct, but we don't know the exact type of each one. Let's find out the types of each object.

```
[10]: type(index)
```

```
[10]: pandas.core.indexes.base.Index
```

```
[11]: type(columns)
```

```
[11]: pandas.core.indexes.base.Index
```

```
[12]: type(data)
```

```
[12]: numpy.ndarray
```

Accessing the components does not change the DataFrame

Accessing these components does nothing to our DataFrame. It merely gives us a variable to reference each of these components. Let's verify that the DataFrame remains unchanged.

```
[13]: df2
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

pandas Index type

Both the index and columns are a special type of object named `Index`. This `Index` object is somewhat similar to a Python list. It is a sequence of labels for either the rows or the columns. You will not deal with this object much directly, so we will not go into further details about it here.

Two-dimensional numpy array

The values are returned as a single two-dimensional numpy array.

Operating with the DataFrame and not its components

You rarely need to operate with these components directly and instead will be working with the entire DataFrame. But, it is important to understand that they are separate components and you can access them directly if needed.

4.3 Accessing the components of a Series

Similarly, we can access the two Series components - the index and the data. Let's first select a single column from our DataFrame so that we have a Series. When we select a column from the DataFrame as a Series, the index remains the same.

```
[14]: color = df2['color']
color
```

```
[14]: name
Jane      blue
Niko      green
Aaron     red
Penelope  white
Dean      gray
Christina black
Cornelia  red
Name: color, dtype: object
```

Let's access the index and the data from the `color` Series without assigning them to separate variables.

```
[15]: color.index
```

```
[15]: Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina', 'Cornelia'],
dtype='object', name='name')
```

```
[16]: color.values
```

```
[16]: array(['blue', 'green', 'red', 'white', 'gray', 'black', 'red'],
dtype=object)
```

The default index

If you don't specify an index when first reading in a DataFrame, then pandas creates one for you as the sequence of integers integers beginning at 0. Let's read in the movie dataset and keep the default index.

```
[17]: movie = pd.read_csv('../data/movie.csv')
movie.head(3)
```

	title	year	color	content_rating	duration	...	plot_keywords	language	country	budget	imdb_score
0	Avatar	2009.0	Color	PG-13	178.0	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
1	Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
2	Spectre	2015.0	Color	PG-13	148.0	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Integers in the index

The integers you see above in the index are the labels for each of the rows. Let's examine the underlying index object.

```
[18]: idx = movie.index
idx
```

```
[18]: RangeIndex(start=0, stop=4916, step=1)
```

We can also verify its type.

```
[19]: type(idx)
```

```
[19]: pandas.core.indexes.range.RangeIndex
```

The RangeIndex

pandas has various types of index objects. A `RangeIndex` is the simplest index and represents the sequence of consecutive integers beginning at 0. It is similar to a Python `range` object in that the values are not actually stored in memory.

A numpy array underlies the index

The index has a `values` attribute just like the DataFrame. Use it to retrieve the underlying index values as a numpy array.

```
[20]: idx.values
```

```
[20]: array([ 0, 1, 2, ..., 4913, 4914, 4915])
```

It's not necessary to assign the index to a variable name to access its attributes and methods. You can access it beginning from the DataFrame.

```
[21]: movie.index.values
```

```
[21]: array([ 0, 1, 2, ..., 4913, 4914, 4915])
```

4.4 Setting an index on read

The `read_csv` function provides dozens of parameters that allow us to read in a wide variety of text files. The `index_col` parameter may be used to select a particular column as the index. We can either use the column name or its integer location.

Reread the movie dataset with the movie title as the index

There's a column in the movie dataset named `title`. Let's reread the data using it as the index.

```
[22]: movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title						...					
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Notice that now the titles of each movie serve as the label for each row. Also notice that the word **title** appears directly above the index. This is a bit confusing. The word **title** is NOT a column name. Technically, it is the **name** of the index, but this isn't important at the moment.

Access the new index and output its type

Let's access this new index, output its values, and verify that type is now **Index** instead of **RangeIndex**.

```
[23]: idx2 = movie.index
idx2
```

```
[23]: Index(['Avatar', 'Pirates of the Caribbean: At World's End', 'Spectre',
       'The Dark Knight Rises', 'Star Wars: Episode VII - The Force Awakens',
       'John Carter', 'Spider-Man 3', 'Tangled', 'Avengers: Age of Ultron',
       'Harry Potter and the Half-Blood Prince',
       ...
       'Primer', 'Cavite', 'El Mariachi', 'The Mongol King', 'Newlyweds',
       'Signed Sealed Delivered', 'The Following', 'A Plague So Pleasant',
       'Shanghai Calling', 'My Date with Drew'],
      dtype='object', name='title', length=4916)
```

```
[24]: type(idx2)
```

```
[24]: pandas.core.indexes.base.Index
```

Select a value from the index

The index is a complex object on its own and has many attributes and methods. The minimum we should know about an index is how to select values from it. We can select single values from an index just like we do with a Python list, by placing the integer location of the item we want within the square brackets. Here, we select the 4th item (integer location 3) from the index.

```
[25]: idx2[3]
```

```
[25]: 'The Dark Knight Rises'
```

We can select this same index label without actually assigning the index to a variable first.

```
[26]: movie.index[3]
```

```
[26]: 'The Dark Knight Rises'
```

Selection with slice notation

As with Python lists, you can select a range of values using slice notation. Provide the start, stop, and step components of slice notation separated by a colon within the brackets.

```
[27]: idx2[100:120:4]
```

```
[27]: Index(['The Fast and the Furious', 'The Sorcerer's Apprentice', 'Warcraft',
           'Transformers', 'Hancock'],
           dtype='object', name='title')
```

Selection with a list of integers

You can select multiple individual values with a list of integers. This type of selection does not exist for Python lists.

```
[28]: nums = [1000, 453, 713, 2999]
idx2[nums]
```

```
[28]: Index(['The Life Aquatic with Steve Zissou', 'Daredevil', 'Daddy Day Care',
           'The Ladies Man'],
           dtype='object', name='title')
```

4.5 Choosing a good index

Before even considering using one of the columns as an index, know that it's not a necessity. You can complete all of your analysis tasks with just the default `RangeIndex`. The reason the index is mentioned in this book, is that there are some tasks that become easier with a custom index. Also, many other pandas users do analysis with the index, so it's important to understand how it works.

If you do choose to set an index for your `DataFrame`, I suggest using columns that are both **unique** and **descriptive**. Pandas does not enforce uniqueness for its index allowing the same value to repeat multiple times. That said, a good index will have unique values to identify each row.

Verifying uniqueness in the index

The `set_index` method has the ability to verify that all values used for the index are unique by setting the `verify_integrity` parameter to `True`.

```
[29]: movie2 = pd.read_csv('../data/movie.csv')
movie2.set_index('title', verify_integrity=True).head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Attempting to set the index to a column with duplicate values will raise an error. The `color` column has only a few unique values and fails to be set as the index when `verify_integrity` is set to `True`.

```
[30]: movie2.set_index('color', verify_integrity=True).head(3)
```

```
      ValueError: Index has duplicate keys: Index(['Color', 'Black and White', nan],  
      ↪dtype='object', name='color')
```

4.6 Exercises

You may wish to change the display options before completing the exercises.

Exercise 1

Read in the movie dataset and set the index to be something other than movie title. Are there any other good columns to use as an index?

```
[ ]:
```

Exercise 2

Use `set_index` to set the index and keep the column as part of the data. Read the docstrings to find the parameter that controls this functionality.

```
[ ]:
```

Exercise 3

Read in the movie DataFrame and set the index as the title column. Assign the index to its own variable and output the last 10 movies titles.

```
[ ]:
```

Exercise 4

Use an integer instead of the column name for `index_col` when reading in the data using `read_csv`. What does it do?

```
[ ]:
```


Chapter 5

Five-Step Process for Data Exploration

In this chapter, we discuss a simple process for exploring data in a Jupyter Notebook. This is the same process that I use when exploring data for the first time and should help you understand data better and write better, cleaner code.

Major issues arise for beginners when too many lines of code are written in a single cell of a notebook. It's important to get feedback on every single line of code that you write and verify that it is, in fact, correct. Only once you have verified the result should you move on to the next line of code. To help increase your ability to do data exploration in Jupyter Notebooks, I recommend the following five-step process:

1. Write and execute a single line of code to explore your data
2. Verify that this line of code works by inspecting the output
3. Assign the result to a variable
4. Within the same cell, in a second line, output the head of the DataFrame or Series
5. Continue to the next cell. Do not add more lines of code to the cell

Apply to every part of the analysis

You can apply this five-step process to every part of your data analysis. Let's begin by reading in the bikes dataset and applying the five-step process for setting the index of our DataFrame as the `trip_id` column.

```
[1]: import pandas as pd  
bikes = pd.read_csv('../data/bikes.csv')  
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Step 1: Write and execute a single line of code to explore your data

In this step, we call the `set_index` method to use the `trip_id` column as the index.

[2]: `bikes.set_index('trip_id').head(3)`

	gender	starttime	stoptime	tripduration	from_station_name	...	temperature	visibility	wind_speed	precipitation	events
trip_id											
7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	73.9	10.0	12.7	-9999.0	mostlycloudy
7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	69.1	10.0	6.9	-9999.0	partlycloudy
10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Step 2: Verify that this line of code works by inspecting the output

Looking above, the output appears to be correct. The `trip_id` column is set as the index and is no longer a column.

Step 3: Assign the result to a variable

You would normally do this step in the same cell, but for this demonstration, we will place it in the cell below. Assign the above operation to a new variable name. Here, we use `bikes2`.

[3]: `bikes2 = bikes.set_index('trip_id')`

Step 4: Within the same cell, in a second line, output the head of the DataFrame or Series

Again, all these steps would be combined in the same cell. Output the `head` of the DataFrame.

[4]: `bikes2.head(3)`

	gender	starttime	stoptime	tripduration	from_station_name	...	temperature	visibility	wind_speed	precipitation	events
trip_id											
7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	73.9	10.0	12.7	-9999.0	mostlycloudy
7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	69.1	10.0	6.9	-9999.0	partlycloudy
10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Step 5: Continue to the next cell. Do not add more lines of code to the cell

It is tempting to do more analysis in a single cell. I advise against doing so when you are a beginner. By limiting your analysis to a single main line of code per cell, and outputting that result, you can easily trace your work from one step to the next. Most lines of code in a notebook apply some operation to the data. It is vital that you can see exactly what this operation is doing. If you put multiple lines of code in a single cell, you lose track of what is happening and can't easily determine the veracity of each operation.

All steps in one cell

The five-step process was shown above one step at a time in different cells. When you actually explore data with this process, you complete it in a single cell.

```
[5]: bikes2 = bikes.set_index('trip_id')
bikes2.head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	temperature	visibility	wind_speed	precipitation	events
trip_id											
7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	73.9	10.0	12.7	-9999.0	mostlycloudy
7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	69.1	10.0	6.9	-9999.0	partlycloudy
10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	73.0	10.0	16.1	-9999.0	mostlycloudy

More examples

Let's see a more complex example of the five-step process. Let's find the `from_station_name` that has the longest average trip duration. This example will be completed with two rounds of the five-step process. First we find the average trip duration for each station and then sort it. This example uses the `groupby` method which is covered in the **Grouping Data** part of the book. It's not important that you understand how `groupby` works here. The point is to show the five-step process for data exploration.

```
[6]: avg_td = bikes.groupby('from_station_name').agg({'tripduration':'mean'})
avg_td.head(3)
```

tripduration	
from_station_name	
2112 W Peterson Ave	911.625000
63rd St Beach	1027.666667
900 W Harrison	495.500000

After grouping, we can sort from greatest to least. A second round of the five-step process is completed to get the following result:

```
[7]: top_stations = avg_td.sort_values('tripduration', ascending=False)
top_stations.head(3)
```

tripduration	
from_station_name	
Western Blvd & 48th Pl	7902.000000
Kedzie Ave & Lake St	5474.823529
Ridge Blvd & Howard St	4839.666667

While it is possible to complete this example in a single cell, I recommend executing only a single main line of code that explores the data.

No strict requirement for one line of code

The above examples each had a single main line of code followed by the head of the DataFrame being output. Often times there will be a few more simple lines of code that can be written in the same cell. You should not adhere strictly to writing a single line of code, but instead, think about keeping the amount of code written in a single cell to a minimum.

For instance, the following cell selects a subset of the data with three lines of code. The first line is simple and creates a list of column names as strings. This is an instance where multiple lines of code are easily interpreted.

```
[8]: cols = ['gender', 'tripduration']
bikes_gt = bikes[cols]
bikes_gt.head(3)
```

	gender	tripduration
0	Male	993
1	Male	623
2	Male	1040

When to assign the result to a variable

Not all operations on our data need to be assigned to a variable. We might just be interested in seeing the results. But, for many operations, you will want to continue with the new transformed data. By assigning the result to a variable, you will have immediate access to the result.

When to create a new variable name

During step 3 of the first example, the result of our new dataset was assigned to `bikes2`. We could have reassigned the result back to `bikes` and continued on with our analysis. When first exploring data, I recommend creating a new variable name for each major result. By doing so, you have preserved each step of your work and are able to inspect it at a later date. Creating new variables makes it much easier to find errors at different places in your analysis.

When to reuse variable names

The downside to using new variable names is that each variable can hold a copy of the data and if your dataset is large, you might take up unnecessary amounts of memory slowing down the performance of your machine. By reassigning a result to the same variable name, you'll reduce memory used.

Another time to reuse variable names is when you are confident that the analysis you have produced is correct and no longer needed to preserve all the previous results.

Continuously verifying results

Regardless of how adept you become at doing data explorations, it is good practice to verify each line of code. Data science is difficult and it is easy to make mistakes. Data is also messy and it is good to be skeptical while proceeding through an analysis. Getting visual verification that each line of code produces the desired result is important. Doing this provides feedback to help you think about what avenues to explore next.

Part II

Selecting Subsets of Data

Chapter 6

Selecting Subsets of Data from DataFrames with just the brackets

One of the most common tasks during a data analysis is to select a subset of the dataset. In pandas, this means selecting particular rows and/or columns from a DataFrame or selecting values from a Series. Although subset selection sounds like an easy task, and is an easy task for many other data manipulation tools, it's actually quite complex with pandas.

Examples of selections of subsets of data

The following images show different types of subset selection that are possible. We first highlight the values we want to select and then show the corresponding DataFrame after the completed selection.

Selection of columns

The most common subset selection, involves selecting one or more columns of a DataFrame. In this example, we select the `color`, `age`, and `height` columns.

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

The selection returns the following DataFrame:

	color	age	height
Jane	blue	30	165
Niko	green	2	70
Aaron	red	12	120
Penelope	white	4	80
Dean	gray	32	180
Christina	black	33	172
Cornelia	red	69	150

Selection of rows

Subsets of rows are a less frequent selection. In this example, we select the rows labeled Aaron and Dean.

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

The selection returns the following DataFrame:

	state	color	food	age	height	score
Aaron	FL	red	Mango	12	120	9.0
Dean	AK	gray	Cheese	32	180	1.8

Simultaneous selection of rows and columns

The last type of subset selection involves selecting rows and columns simultaneously. In this example, we select the color, age, and height columns along with the rows labeled Aaron and Dean.

	state	color	food	age	height	score
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

The selection returns the following DataFrame:

	color	age	height
Aaron	red	12	120
Dean	gray	32	180

6.1 pandas dual references: by label and by integer location

As previously mentioned, the index of each DataFrame provides a label to reference each individual row. Similarly, the column names provide a label to reference each column. What hasn't been mentioned, is that each row and column may be referenced by an integer as well. I call this **integer location**. The integer location begins at 0 for the first row and continues sequentially one integer at a time until the last row. The last row will have integer location $n - 1$, where n is the total number of rows in the DataFrame.

Take a look above at our DataFrame one more time. The rows with labels `Aaron` and `Dean` can also be referenced by their respective integer locations 2 and 4. Similarly, the columns `color`, `age`, and `height` can be referenced by their integer locations 1, 3, and 4.

The official pandas documentation refers to integer location as **position**. I don't particularly like this terminology as it's not as explicit as integer location. The key term here is the word **integers**.

What's the difference between indexing and selecting subsets of data?

The documentation uses the term **indexing** frequently. This term is a shorter, more technical term that implies **subset selection**. I prefer the term subset selection, as, again, it is more descriptive of what is actually happening. Indexing is also the term used in the official Python documentation (for selecting subsets of lists or strings for example).

6.2 The three indexers [], loc, iloc

pandas provides three **indexers** to select subsets of data. An indexer is a term for one of `[]`, `loc`, or `iloc` and is what makes the subset selection. All the details on how to make selections with each of these indexers will be covered. Each indexer has different rules for how they work. All of our selections will look similar to the following, except they will have something placed within the brackets.

```
>>> df[]  
>>> df.loc[]  
>>> df.iloc[]
```

Terminology

When the brackets are placed directly after the DataFrame variable name, the term **just the brackets** will be used to differentiate them from the brackets after `loc` and `iloc`.

Square brackets instead of parentheses

One of the most common mistakes when using `loc` and `iloc` is to append parentheses to them, instead of square brackets. One of the main reasons this mistake is done is because `loc` and `iloc` appear to be methods and all methods are called with parentheses. Both `loc` and `iloc` are not methods, but are accessed in the same manner as methods through dot notation, which leads to the mistake.

Few objects accessed through dot notation use brackets instead of parentheses. In Python, the brackets are a universal operator for selecting subsets of data regardless of the type of object. The brackets select subsets of lists, strings, and select a single value in a dictionary. numpy arrays use the brackets operator for subset selection. If you are doing subset selection, it's likely that you need brackets and not parentheses.

6.3 Begin with *just the brackets*

As we saw in a previous chapter, just the brackets are used to select a single column as a Series. We place the column name inside the brackets to return the Series. Let's read in a simple, small DataFrame and

select a single column from it. We use the `index_col` parameter to set the index to the first column (integer location 0) on read.

```
[1]: import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Append square brackets directly to the DataFrame variable name and then place the name of the column within those brackets. This selects a single column of data as a Series.

```
[2]: df['color']
```

```
[2]: name
Jane      blue
Niko      green
Aaron     red
Penelope  white
Dean      gray
Christina black
Cornelia  red
Name: color, dtype: object
```

6.4 Select multiple columns with a list

Select multiple columns by placing the column names in a list inside of just the brackets. Notice that a DataFrame and NOT a Series is returned.

```
[3]: df[['color', 'age', 'score']]
```

	color	age	score
name			
Jane	blue	30	4.6
Niko	green	2	8.3
Aaron	red	12	9.0
Penelope	white	4	3.3
Dean	gray	32	1.8
Christina	black	33	9.5
Cornelia	red	69	2.2

You must use an inner set of brackets

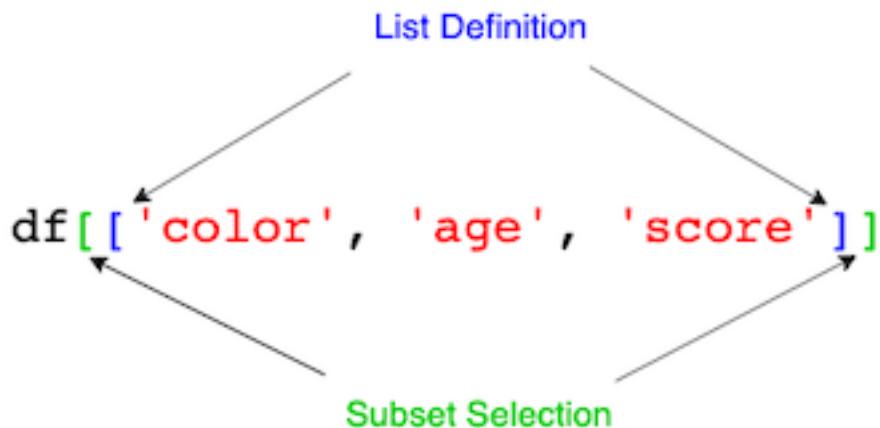
You might be tempted to do the following, which will NOT work. When selecting multiple columns, you must use a **list** to contain the names. Remember, that a list is defined by a set of square brackets, so the following raises an error.

```
[4]: df['color', 'age', 'score']
```

`KeyError: ('color', 'age', 'score')`

The inner square brackets define a list, the outer square brackets do subset selection

To help understand the double set of brackets, take a look at the following image. The inner set of brackets define a list of three items. The outer set of brackets mean something completely different. They inform the DataFrame to make a subset selection.



This difference is confusing because the exact same syntax, the brackets, have a different meaning depending on where they are used. When the brackets are appended directly to the right of a variable name, they translate as “subset selection”. When the brackets appear apart from any variable, they translate as “create a list”.

Use two lines of code to select multiple columns

To help clarify the process of making subset selection, I recommend using intermediate variables. In this instance, we assign the columns we would like to select to a list and then pass this list to the brackets.

```
[5]: cols = ['color', 'age', 'score']
df[cols]
```

		color	age	score
	name			
	Jane	blue	30	4.6
	Niko	green	2	8.3
	Aaron	red	12	9.0
	Penelope	white	4	3.3
	Dean	gray	32	1.8
	Christina	black	33	9.5
	Cornelia	red	69	2.2

Columns in any order

The order of the column names in the list is important. The new DataFrame will have the columns in the order given from the list.

```
[6]: cols = ['height', 'age']
df[cols]
```

		height	age
	name		
	Jane	165	30
	Niko	70	2
	Aaron	120	12
	Penelope	80	4
	Dean	180	32
	Christina	172	33
	Cornelia	150	69

6.5 Summary of *just the brackets*

The primary purpose of *just the brackets* is to select either one or more entire columns as either a Series or DataFrame. Providing it a single column returns a Series, while providing it a list of columns returns a DataFrame. In later chapters, we will see how to select rows using *just the brackets* by passing it a boolean Series.

6.6 Exercises

Read in the movie dataset by executing the cell below and use it for the following exercises.

```
[7]: pd.set_option('display.max_columns', 50)
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 1

Select the column with the director's name as a Series

```
[ ]:
```

Exercise 2

Select the column with the director's name and number of Facebook likes.

```
[ ]:
```

Exercise 3

Select a single column as a DataFrame and not a Series

```
[ ]:
```

Exercise 4

Look in the data folder and read in another dataset. Select some columns from it.

```
[ ]:
```


Chapter 7

Selecting Subsets of Data from DataFrames with loc

In this chapter, we use the `loc` indexer to select subsets of data from DataFrames. The `loc` indexer selects data in a different manner than *just the brackets*. It has its own separate set of rules that we must learn.

7.1 Simultaneous row and column subset selection

The `loc` indexer can select rows and columns simultaneously. This is done by separating the row and column selections with a **comma**. The selection will look something like this:

```
df.loc[rows, cols]
```

Just the brackets cannot do this

Simultaneous row and column subset selection is not possible with *just the brackets*. Reiterating from above, the `loc` indexer has a completely different and distinct set of rules that you must abide by to use correctly. It's best to forget about how *just the brackets* works when first learning subset selection with `loc`.

loc primarily selects data by label

Very importantly, `loc` primarily selects subsets by the **label** of the rows and columns. It also makes selections via boolean selection, a topic covered in a later chapter.

Read in data

Let's get started by reading in a sample DataFrame with the first column set as the index.

```
[1]: import pandas as pd  
df = pd.read_csv('../data/sample_data.csv', index_col=0)  
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Select two rows and three columns with loc

Let's make our first selection with `loc` by simultaneously selecting some rows and some columns. Let's select the rows `Dean` and `Cornelia` along with the columns `age`, `state`, and `score`. A list is used to contain both the row and column selections before being placed within the brackets following `loc`. Row and column selection must be separated by a comma.

```
[2]: rows = ['Dean', 'Cornelia']
cols = ['age', 'state', 'score']
df.loc[rows, cols]
```

	age	state	score
name			
Dean	32	AK	1.8
Cornelia	69	TX	2.2

The possible types of row and column selections

In the above example, we used a list of labels for both the row and column selection. You are not limited to just lists. All of the following are valid objects available for both row and column selections with `loc`.

- A single label
- A list of labels
- A slice with labels
- A boolean Series (covered in a later chapter)

Select two rows and a single column

Let's select the rows `Aaron` and `Dean` along with the `food` column. We can use a list for the row selection and a single string for the column selection.

```
[3]: rows = ['Dean', 'Aaron']
cols = 'food'
df.loc[rows, cols]
```

[3]: name

```
Dean      Cheese
Aaron     Mango
Name: food, dtype: object
```

Series returned

In the above example, a Series and not a DataFrame was returned. Whenever you select a single row or a single column using a string label, pandas returns a Series

7.2 loc with slice notation

Let's take a moment to review Python's slice notation, which are used to select subsets from some core Python objects such as lists, tuples, and strings. Slice notation always has three components - the **start**, **stop**, and **step**. Syntactically, each component is separated by a colon like this - `start:stop:step`. All components of slice notation are optional and not necessary to include. Each has a default value if not included in the notation. The start component defaults to the beginning, the stop defaults to the end, and the step size to 1.

Example slices

Let's take a look at several slice notations and the value of each component of the slice

- '`Niko':'Christina':2`' - start is 'Niko', stop is 'Christina', step is 2
- '`Niko':'Christina'`' - start is 'Niko', stop is 'Christina', step is 1
- '`Niko':'2`' - start is 'Niko', stop is the end, step is 2
- '`Niko':`' - start is 'Niko', stop is the end, step is 1
- '`:'Christina':2`' - start is the beginning, stop is 'Christina', step is 2
- '`:`' - start is the beginning, stop is the end, step is 1. All components take their default value.

This same slice notation is allowed with DataFrames. Let's select all of the rows from Jane to Penelope with slice notation along with the columns `state` and `color`.

[4]: cols = ['`state`', '`color`']
`df.loc['Jane':'Penelope', cols]`

		state	color
	name		
	Jane	NY	blue
	Niko	TX	green
	Aaron	FL	red
	Penelope	AL	white

Slice notation is inclusive of the stop label

Slice notation with the `loc` indexer includes the stop label. This behaves differently than slicing done on Python lists, which is exclusive of the stop integer.

Slice notation only works within the brackets attached to the object

Python only allows us to use slice notation within the brackets that are attached to an object. If we try and assign slice notation outside of this, we will get a syntax error like we do below.

```
[5]: rows = 'Jane':'Penelope'
```

SyntaxError: invalid syntax

Slice both the rows and columns

Both row and column selections support slice notation. In the following example, we slice all the rows from the beginning up to and including label `Dean` along with columns from `height` until the end.

```
[6]: df.loc[:'Dean', 'height':]
```

	height	score
name		
Jane	165	4.6
Niko	70	8.3
Aaron	120	9.0
Penelope	80	3.3
Dean	180	1.8

Selecting all of the rows and some of the columns

It is possible to use slice notation to select all of rows or columns. We do so with a single colon, which is sometimes referred to as the **empty slice**. In this example, we select all of the rows and two of the columns.

```
[7]: cols = ['food', 'color']
df.loc[:, cols]
```

	food	color
name		
Jane	Steak	blue
Niko	Lamb	green
Aaron	Mango	red
Penelope	Apple	white
Dean	Cheese	gray
Christina	Melon	black
Cornelia	Beans	red

Could have used *just the brackets*

It isn't necessary to use `loc` for this selection as we are only selecting two distinct columns. This could have been accomplished with *just the brackets*.

```
[8]: cols = ['food', 'color']
df[cols]
```

	food	color
name		
Jane	Steak	blue
Niko	Lamb	green
Aaron	Mango	red
Penelope	Apple	white
Dean	Cheese	gray
Christina	Melon	black
Cornelia	Beans	red

A single colon is slice notation to select all values

That single colon might be intimidating, but it is technically slice notation that selects all items. In the following example, all of the elements of a Python list are selected using a single colon.

```
[9]: a_list = [1, 2, 3, 4, 5, 6]
a_list[:]
```

```
[9]: [1, 2, 3, 4, 5, 6]
```

Use a single colon to select all the columns

It is possible to use a single colon to represent a slice of all of the rows or all of the columns. Below, a colon is used as slice notation for all of the columns.

```
[10]: rows = ['Penelope', 'Cornelia']
df.loc[rows, :]
```

	state	color	food	age	height	score
name						
Penelope	AL	white	Apple	4	80	3.3
Cornelia	TX	red	Beans	69	150	2.2

The above can be shortened

By default, pandas selects all of the columns if you only provide a row selection. Providing the colon is not necessary so the following syntax makes the exact same selection.

```
[11]: rows = ['Penelope', 'Cornelia']
df.loc[rows]
```

	state	color	food	age	height	score
name						
Penelope	AL	white	Apple	4	80	3.3
Cornelia	TX	red	Beans	69	150	2.2

Though it is not syntactically necessary, one reason to use the colon is to reinforce the idea that `loc` may be used for simultaneous column selection. The first object passed to `loc` always selects rows and the second always selects columns.

Use slice notation to select a range of rows with all of the columns

Similarly, we can use slice notation to select several rows at a time. Below, begin at the row labeled by `Niko` and go all the way through `Dean`. We do not provide a specific column selection to return all of the columns.

```
[12]: df.loc['Niko':'Dean']
```

	state	color	food	age	height	score
name						
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8

You could have written the above as `df.loc['Niko':'Dean', :]` to reinforce the fact that `loc` first selects rows and then columns.

Changing the step size

The step size must be an integer when using slice notation with `loc`. In this example, we select every other row beginning at `Niko` and ending at `Christina`.

```
[13]: df.loc['Niko':'Christina':2, :]
```

	state	color	food	age	height	score
name						
Niko	TX	green	Lamb	2	70	8.3
Penelope	AL	white	Apple	4	80	3.3
Christina	TX	black	Melon	33	172	9.5

Select a single row and a single column

If the row and column selections are both a single label, then a scalar value and NOT a DataFrame or Series is returned.

```
[14]: rows = 'Jane'  
       cols = 'state'  
       df.loc[rows, cols]
```

```
[14]: 'NY'
```

Select a single row as a Series with loc

The `loc` indexer returns a single row as a Series when given a single row label. Let's select the row for Niko. Notice that the column names have now become index labels.

```
[15]: df.loc['Niko']
```

```
[15]: state      TX  
color     green  
food      Lamb  
age        2  
height     70  
score      8.3  
Name: Niko, dtype: object
```

Again, the column selection isn't necessary, but does provide clarity.

```
[16]: df.loc['Niko', :]
```

```
[16]: state      TX  
color     green  
food      Lamb  
age        2  
height     70  
score      8.3  
Name: Niko, dtype: object
```

Confusing output

This output is potentially confusing. The original row that was labeled by Niko had horizontal data. Selecting a single row returns a Series that displays the row data vertically.

Selecting a single row as a DataFrame

It is possible to select a single row as a DataFrame instead of a Series. Create the row selection as a one-item list instead of just a string label. The returned result is a DataFrame and maintains the same horizontal position for the row.

```
[17]: rows = ['Niko']  
       df.loc[rows, :]
```

	state	color	food	age	height	score
name						
Niko	TX	green	Lamb	2	70	8.3

7.3 Summary of the loc indexer

- Primarily uses labels
- Selects rows and columns simultaneously with `df.loc[rows, cols]`
- Both row and column selections can be a:
 - single label
 - list of labels
 - slice of labels
 - boolean Series
- A comma separates row and column selections

7.4 Exercises

Read in the movie dataset by executing the cell below and use it for the following exercises.

```
[18]: pd.set_option('display.max_columns', 50)
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 1

Select columns `actor1`, `actor2`, and `actor3` for the movies ‘Home Alone’ and ‘Top Gun’.

```
[ ]:
```

Exercise 2

Select columns `actor1`, `actor2`, and `actor3` for all of the movies beginning at ‘Home Alone’ and ending at ‘Top Gun’.

```
[ ]:
```

Exercise 3

Select just the `director_name` column for the movies ‘Home Alone’ and ‘Top Gun’.

[]:

Exercise 4

Repeat exercise 3, but return a DataFrame instead.

[]:

Exercise 5

Select all columns for the movie ‘The Dark Knight Rises’.

[]:

Exercise 6

Repeat exercise 5 but return a DataFrame instead.

[]:

Exercise 7

Select all columns for the movies ‘Tangled’ and ‘Avatar’.

[]:

Exercise 8

What year was ‘Tangled’ and ‘Avatar’ made and what was their IMBD scores?

[]:

Exercise 9

What is the data type of the `year` column?

[]:

Exercise 10

Use a single method to output the data type and number of non-missing values of `year`. Is it missing any?

[]:

Exercise 11

Select every 300th movie between ‘Tangled’ and ‘Forrest Gump’. Why doesn’t ‘Forrest Gump’ appear in the results?

[]:

Chapter 8

Selecting Subsets of Data from DataFrames with iloc

The `iloc` indexer is very similar to the `loc` indexer but only uses **integer location** to make its subset selections. The word `iloc` itself stands for integer location and can help remind you what it does.

8.1 Simultaneous row and column subset selection

The `iloc` indexer is capable of making simultaneous row and column selections just like `loc`. Selection with `iloc` takes on the following form, with a comma separating the row and column selections.

```
df.iloc[rows, cols]
```

Let's read in some sample data and then begin making selections with integer location using `iloc`.

```
[1]: import pandas as pd
df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

		state	color	food	age	height	score
	name						
	Jane	NY	blue	Steak	30	165	4.6
	Niko	TX	green	Lamb	2	70	8.3
	Aaron	FL	red	Mango	12	120	9.0
	Penelope	AL	white	Apple	4	80	3.3
	Dean	AK	gray	Cheese	32	180	1.8
	Christina	TX	black	Melon	33	172	9.5
	Cornelia	TX	red	Beans	69	150	2.2

What is integer location?

Integer location is the term used to reference a row or column. The first row/column is referenced by the integer 0. Each subsequent row is referenced by the next integer. The last row/column is referenced by `n - 1` where `n` is the number of row/columns.

Select using a list for both rows and columns

Let's select rows with integer location 2 and 4 along with the first and last columns. It is possible to use negative integers in the same manner as Python lists. The integer location -1 refers to the last column below.

```
[2]: rows = [2, 4]
cols = [0, -1]
df.iloc[rows, cols]
```

	state	score
name		
Aaron	FL	9.0
Dean	AK	1.8

The possible types of selections for iloc

In the above example, we used a list of integers for both the row and column selection. You are not limited to just lists. All of the following are valid objects available for both row and column selections with `iloc`. The `iloc` indexer, unlike `loc`, is unable to do boolean selection.

- A single integer
- A list of integers
- A slice with integers

Slice the rows and use a list for the columns

Let's use slice notation to select rows with integer location 2 and 3 and a list to select columns with integer location 4 and 2. Notice that the stop integer location is **excluded** with `iloc`, which is exactly how slicing works with Python lists, tuples, and strings. Slicing with `loc` is **inclusive** of the stop label.

```
[3]: cols = [4, 2]
df.iloc[2:4, cols]
```

	height	food
name		
Aaron	120	Mango
Penelope	80	Apple

Use a list for the rows and a slice for the columns

In this example, we use a list for the row selection and slice notation for the columns.

```
[4]: rows = [5, 2, 4]
df.iloc[rows, 3:]
```

	age	height	score
name			
Christina	33	172	9.5
Aaron	12	120	9.0
Dean	32	180	1.8

Select all of the rows and some of the columns

You can use an empty slice (just the colon) to select all of the rows or columns. In the example below, we select all of the rows and some of the columns with a list.

```
[5]: cols = [2, 4]
df.iloc[:, cols]
```

	food	height
name		
Jane	Steak	165
Niko	Lamb	70
Aaron	Mango	120
Penelope	Apple	80
Dean	Cheese	180
Christina	Melon	172
Cornelia	Beans	150

Cannot do this with *just the brackets*

Just the brackets does select columns, but it only understands **labels** and not **integer location**. The following produces an error as pandas is looking for column names that are the integers 2 and 4.

```
[6]: df[cols]
```

`KeyError: "None of [Int64Index([2, 4], dtype='int64')] are in the [columns]"`

Select some of the rows and all of the columns

We can again use an empty slice, but do so to select all of the columns. We use a list to select some of the rows.

```
[7]: rows = [-3, -1, -2]
df.iloc[rows, :]
```

	state	color	food	age	height	score
name						
Dean	AK	gray	Cheese	32	180	1.8
Cornelia	TX	red	Beans	69	150	2.2
Christina	TX	black	Melon	33	172	9.5

It is possible to rewrite the above without the column selection. pandas defaults to selecting all of the columns if a selection for them is not explicitly present.

[8]: `df.iloc[rows]`

	state	color	food	age	height	score
name						
Dean	AK	gray	Cheese	32	180	1.8
Cornelia	TX	red	Beans	69	150	2.2
Christina	TX	black	Melon	33	172	9.5

Select a single row and a single column

We can select a single value in our DataFrame using `iloc` by providing a single integer for both the row and column selection. This returns the actual value by itself completely outside of a DataFrame or Series.

[9]: `df.iloc[3, 2]`

[9]: 'Apple'

Select a single row and a single column as a DataFrame

It is possible to select the above value as a DataFrame by using one-item lists for the row and column selections. The output looks a little bizarre, but it's just a DataFrame with one row and one column.

[10]: `rows = [3]
cols = [2]
df.iloc[rows, cols]`

	food
name	
Penelope	Apple

Select some rows and a single column

In this example, a list of integers is used for the rows and a single integer for the columns. pandas returns a Series when a single integer is used to select either a row or column.

```
[11]: rows = [2, 3, 5]
       cols = 4
       df.iloc[rows, cols]
```

```
[11]: name
      Aaron      120
      Penelope    80
      Christina   172
      Name: height, dtype: int64
```

Select a single row or column as a DataFrame and NOT a Series

You can select a single row (or column) and return a DataFrame and not a Series if you use a list to make the selection. Let's replicate the selection from the previous example, but use a one-item list for the column selection.

```
[12]: rows = [2, 3, 5]
       cols = [4]
       df.iloc[rows, cols]
```

height	
	name
	Aaron 120
	Penelope 80
	Christina 172

Select a single row as a Series

We can select a single row by providing a single integer as the row selection for `iloc`. We use an empty slice to select all of the columns. Because we are selecting a single row, a Series is returned. Just as with `loc`, the returned output can be confusing as the original horizontal row is now displayed vertically.

```
[13]: df.iloc[2, :]
```

```
[13]: state      FL
       color     red
       food     Mango
       age      12
       height    120
       score      9
       Name: Aaron, dtype: object
```

To maintain the original orientation, we can select the row using a one-item list which returns a DataFrame.

```
[14]: df.iloc[[2], :]
```

	state	color	food	age	height	score
name						
Aaron	FL	red	Mango	12	120	9.0

8.2 Summary of iloc

The `iloc` indexer is analogous to `loc` but only uses **integer location** for selection. The official pandas documentation refers to it as selection by **position**.

- Uses only integer location
- Selects rows and columns simultaneously with `df.iloc[rows, cols]`
- Selection can be a
 - single integer
 - a list of integers
 - a slice of integers
- A comma separates row and column selections

8.3 Exercises

Read in the movie dataset by executing the cell below and use it for the following exercises.

```
[15]: pd.set_option('display.max_columns', 50)
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 1

Select the columns with integer location 10, 5, and 1.

```
[ ]:
```

Exercise 2

Select the rows with integer location 10, 5, and 1.

```
[ ]:
```

Exercise 3

Select rows with integer location 100 to 104 along with the column integer location 5.

[]:

Exercise 4

Select the value at row integer location 100 and column integer location 4.

[]:

Exercise 5

Return the result of exercise 4 as a DataFrame.

[]:

Exercise 6

Select the last 5 rows of the last 5 columns.

[]:

Exercise 7

Select every 25th row between rows with integer location 100 and 200 along with every fifth column.

[]:

Exercise 8

Select the column with integer location 7 as a Series.

[]:

Exercise 9

Select the rows with integer location 999, 99, and 9 and the columns with integer location 9 and 19.

[]:

Chapter 9

Selecting Subsets of Data from a Series

Selecting subsets of data from a Series is accomplished similarly to how it's done with DataFrames.

9.1 Series indexer rules

The same three indexers, `[]`, `loc`, and `iloc`, are available for the Series. Because there are no columns in a Series, the rules for each indexer are slightly different than they are for a DataFrame. Let's begin by reading in the movie dataset and setting the index to the title.

```
[1]: import pandas as pd  
movie = pd.read_csv('../data/movie.csv', index_col='title')  
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Let's select a single column of data so that we can have access to a Series. Here, we select the `imdb_score` column.

```
[2]: imdb = movie['imdb_score']  
imdb.head(3)
```

```
[2]: title  
Avatar                      7.9  
Pirates of the Caribbean: At World's End    7.1  
Spectre                      6.8  
Name: imdb_score, dtype: float64
```

Series subset selection with just the brackets

For DataFrames, we learned that *just the brackets* accepted either a single label or a list of labels and used this input to select one or more DataFrame columns. For a Series, *just the brackets* has different rules

that you must follow to use it correctly. It allows selection by index label. For instance, we can select the `imdb_score` for the movie Avatar like this:

```
[3]: imdb['Avatar']
```

```
[3]: 7.9
```

Interestingly enough, it's possible to use integer location as well with *just the brackets*. The movie Avatar is at integer location 0 and we can duplicate our previous result by using it.

```
[4]: imdb[0]
```

```
[4]: 7.9
```

9.2 Series subset selection with `loc`

The `loc` indexer selects by **label** just as it does with a DataFrame. Since there are no columns, it only accepts a single selection object which can be any of the following:

- A single label
- A list of labels
- A slice with labels
- A boolean Series (covered in a later chapter)

Select a single value with `loc`

Select a single value by providing the `loc` indexer the name of the index. Here, we select the `imdb_score` of the movie Forrest Gump.

```
[5]: imdb.loc['Forrest Gump']
```

```
[5]: 8.8
```

Select multiple values using a list with `loc`

Provide the `loc` indexer a list of index labels to select multiple values.

```
[6]: names = ['Good Will Hunting', 'Home Alone', 'Meet the Parents']
imdb.loc[names]
```

```
[6]: title
Good Will Hunting    8.3
Home Alone          7.5
Meet the Parents    7.0
Name: imdb_score, dtype: float64
```

Select multiple values using slice notation with `loc`

Provide the `loc` indexer index labels for the start and stop components of slice notation to select all the values between those two labels. The results are **inclusive** of the stop label.

```
[7]: imdb.loc['Home Alone':'Top Gun']
```

[7]: title

```
Home Alone      7.5
3 Men and a Baby 5.9
Tootsie        7.4
Top Gun         6.9
Name: imdb_score, dtype: float64
```

As with any slice notation, all components are optional. Here, we select every `imdb_score` from the movie ‘Twins’ to the end.

[8]: `imdb.loc['Twins':].head()`

[8]: title

```
Twins          6.0
Scream: The TV Series 7.3
The Yellow Handkerchief 6.8
The Color Purple    7.8
Tidal Wave         5.7
Name: imdb_score, dtype: float64
```

In this example, we select every 300th `imdb_score` beginning at the movie Twins to the end.

[9]: `imdb.loc['Twins)::300]`

[9]: title

```
Twins          6.0
Ernest & Celestine 7.9
Welcome to the Rileys 7.0
Alpha and Omega 4: The Legend of the Saw Toothed Cave 6.0
Fast Times at Ridgemont High 7.2
Young Frankenstein 8.0
Neal 'N' Nikki    3.3
Rise of the Entrepreneur: The Search for a Better Way 8.2
Name: imdb_score, dtype: float64
```

9.3 Series subset selection with iloc

The Series `iloc` indexer is analogous to `loc` except that it only makes selection via integer location. Here are the valid kinds of selections.

- A single integer location
- A list of integer locations
- A slice with integer locations

Select a single value with iloc

Let’s select the `imdb_score` for the movie with integer location 499.

[10]: `imdb.iloc[499]`

[10]: 4.2

Selecting with a single integer always return the value by itself and not within a Series. If we want to return a one-item Series, so that we can see the index, we can use a one-item list as our selection.

```
[11]: imdb.iloc[[499]]
```

```
[11]: title  
A Sound of Thunder    4.2  
Name: imdb_score, dtype: float64
```

Select multiple values using a list with iloc

Provide `iloc` a list of integer locations to select multiple values.

```
[12]: ints = [499, 599, 699]  
imdb.iloc[ints]
```

```
[12]: title  
A Sound of Thunder    4.2  
The Abyss            7.6  
Blades of Glory      6.3  
Name: imdb_score, dtype: float64
```

Select multiple values using slice notation with iloc

Provide `iloc` with slice notation using integers as the stop and start components to select all the values between those two locations. The results are **exclusive** of the last integer. Here, we select integer locations 145 through, but not including 148.

```
[13]: imdb.iloc[145:148]
```

```
[13]: title  
Mr. Peabody & Sherman        6.9  
Troy                          7.2  
Madagascar 3: Europe's Most Wanted 6.9  
Name: imdb_score, dtype: float64
```

Let's select the last three values using slice notation.

```
[14]: imdb.iloc[-3:]
```

```
[14]: title  
A Plague So Pleasant     6.3  
Shanghai Calling         6.3  
My Date with Drew       6.6  
Name: imdb_score, dtype: float64
```

Let's select every 200th value from integer location 1,000 to 2,000

```
[15]: imdb.iloc[1000:2000:200]
```

```
[15]: title  
The Life Aquatic with Steve Zissou    7.3  
Ride Along 2                      5.9
```

```
Trainwreck          6.3
Down to Earth      5.4
The Duchess         6.9
Name: imdb_score, dtype: float64
```

Use loc and iloc instead of just the brackets

For a Series, *just the brackets* is flexible and can take either a label or integer location. This might make it seem like `loc` and `iloc` would be unnecessary, but the opposite is actually the case. Using *just the brackets* for a Series is ambiguous and not explicit. It's not clear whether the label or integer location are being used.

I suggest only using `loc` and `iloc` for clarity. Whenever the `loc` indexer is used, we are certain it selects by label. Likewise, whenever the `iloc` indexer is used, we are certain it selects by integer location.

9.4 Summary of Series subset selection

The three indexers, `[]`, `loc`, and `iloc` are available to make subset selections on a Series. They work similarly as they do on DataFrames

- The `loc` indexer makes selections by label using a:
 - single label
 - list of labels
 - slice of labels
 - boolean Series
- The `loc` indexer makes selections by label using a:
 - single integer location
 - list of integer locations
 - slice of integer locations
- Use `loc` and `iloc` instead of *just the brackets* to be explicit
- There are no columns in a Series, so selection is only based on the index

9.5 Exercises

Execute the cell below to select the `duration` column (length of movie in minutes) as a Series and use it for the first few exercises.

```
[16]: duration = movie['duration']
duration.head()
```

```
[16]: title
Avatar                  178.0
Pirates of the Caribbean: At World's End 169.0
Spectre                  148.0
The Dark Knight Rises    164.0
Star Wars: Episode VII - The Force Awakens   NaN
Name: duration, dtype: float64
```

Exercise 1

How long was the movie ‘Titanic’?

```
[ ]:
```

Exercise 2

How long was the movie at the 999th integer location?

[]:

Exercise 3

Select the duration for the movies ‘Hulk’, ‘Toy Story’, and ‘Cars’.

[]:

Exercise 4

Select the duration for every 100th movies from ‘Hulk’ to ‘Cars’.

[]:

Exercise 5

Select the duration for every 10th movie beginning from the 100th from the end.

[]:

Read in bikes dataset

Read in the bikes dataset and select the `wind_speed` column by executing the cell below and use it for the rest of the exercises. Notice that the index labels are integers, meaning that when you use `loc` you will be using integers.

```
[17]: bikes = pd.read_csv('../data/bikes.csv')
wind = bikes['wind_speed']
wind.head()
```

```
[17]: 0    12.7
      1    6.9
      2    16.1
      3    16.1
      4    17.3
Name: wind_speed, dtype: float64
```

Exercise 6

What kind of index does the `wind` Series have?

[]:

Exercise 7

From the `wind` Series, select the integer locations 4 through, but not including 10.

[]:

Exercise 8

Copy and paste your answer to Exercise 7 below but use `loc` instead. Do you get the same result? Why not?

[]:

Chapter 10

Boolean Selection Single Conditions

Boolean Selection, also referred to as **boolean indexing**, is the process of selecting subsets of rows from DataFrames (or Series) based on the actual data values and NOT by their labels or integer locations. All of the previous subset selections were done using either labels or integer location. Those selections had nothing to do with the actual values.

Examples of boolean selection

Let's see some examples of actual questions (in plain English) that boolean selection can help us answer from the bikes dataset. The term **query** is used to refer to these sorts of questions.

- Find all rides by males
- Find all rides with a duration longer than 2 hours
- Find all rides that took place between March and June of 2015
- Find all rides with a duration longer than 2 hours by females with temperature higher than 90 degrees

All queries have a logical condition

Each of the above queries have a strict logical condition that must be checked one row at a time.

Keep or discard an entire row of data

If you were to manually answer the above queries, you would need to scan each row and determine whether the row, as a whole, meets the condition. If so, then it is kept in the result, otherwise it is discarded.

Each row will have a True or False value associated with it

When you perform boolean selection, each row of the DataFrame (or value of a Series) has a **True** or **False** value associated with it that corresponds to the outcome of the logical condition.

Begin with a small DataFrame

Let's perform our first boolean selection on our sample DataFrame. Let's read it in now.

```
[1]: import pandas as pd  
df = pd.read_csv('../data/sample_data.csv', index_col=0)  
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

10.1 Manually filtering the data

Let's find all the people who are younger than 30 years of age. We will do this manually by inspecting the data.

Create a list of booleans

By inspecting the data, we see that **Niko**, **Aaron**, and **Penelope** are all under 30 years of age. To signify which people are under 30, we create a list of 7 boolean values corresponding to the 7 rows in the DataFrame. The values in the list that correspond with the positions of **Niko**, **Aaron**, and **Penelope** are **True**. All other values are **False**. **Niko**, **Aaron**, and **Penelope** are the 2nd, 3rd, and 4th rows, so these are the locations in the list that are **True**.

```
[2]: filt = [False, True, True, True, False, False]
```

Variable name `filt`

The variable name `filt` will be used throughout the book to refer to the sequence of booleans. You are free to use any variable name you like for the sequence of booleans, but being consistent makes your code easier to understand. I chose `filt` because it is short for the word 'filter'. Boolean selection filters the data for a particular condition, which is why this variable name makes sense to me.

Place this list in just the brackets

The above list has **True** in the 2nd, 3rd, and 4th positions. These will be the rows that are kept in the resulting boolean selection. Place the list inside *just the brackets* to complete the selection.

```
[3]: df[filt]
```

	state	color	food	age	height	score
name						
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3

Wait a second... Isn't [] just for column selection?

The primary purpose of *just the brackets* for a DataFrame is to select one or more columns by using either a string or a list of strings. All of a sudden, this example shows entire rows being selected with boolean values. This is what makes pandas, unfortunately, a confusing library to learn and use.

10.2 Operator overloading

Just the brackets is **overloaded**. Depending on the inputs, pandas will do something completely different. Here are the rules for the different objects passed to *just the brackets*.

- **string**—return a column as a Series
- **list of strings**—return all those column names as a DataFrame
- **sequence of booleans**—select all rows where True

In summary, *just the brackets* primarily selects columns, but, if you pass it a sequence of booleans, it will select all rows that are True.

10.3 Practical boolean selection

We almost never create boolean lists manually like we did above and instead use the actual data to create boolean Series.

Creating boolean Series from column data

By far the most common way to create a boolean Series is from the values of one particular column. We test a condition using one of the six comparison operators:

- <
- <=
- >
- >=
- ==
- !=

Let's begin completing practical boolean selection examples by reading in the bikes dataset.

```
[4]: bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Create a boolean Series

Let's create a boolean Series by determining which rows have a trip duration greater than 1,000 seconds. To make the comparison, we select the `tripduration` column as a Series and compare it against the integer

1,000.

```
[5]: filt = bikes['tripduration'] > 1000
filt.head(3)
```

```
[5]: 0    False
      1    False
      2    True
Name: tripduration, dtype: bool
```

When we write `bikes['tripduration'] > 1000`, pandas compares each value in the `tripduration` column against 1,000. It returns a new Series the same length as `tripduration` with boolean values corresponding to the outcome of the comparison. Let's verify that the `filt` Series is the same length as the DataFrame.

```
[6]: len(filt)
```

```
[6]: 50089
```

```
[7]: len(bikes)
```

```
[7]: 50089
```

Manually verify correctness

Take a look at the `tripduration` column to manually verify that only the third row satisfied the condition. That ride lasted 1,040 seconds which is greater than 1,000 resulting in a value of `True`. The first two rides lasted less than 1,000 seconds and resulting with `False`.

Complete the boolean selection

We can now place the `filt` boolean Series into *just the brackets* to filter the entire DataFrame. This returns all the rows in the bikes dataset that have a trip duration greater than 1,000. Manually verify that the `tripduration` values on the screen are greater than 1,000.

```
[8]: bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
10	24383	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...	79.0	10.0	9.2	-9999.0	mostlycloudy

How many rows have a trip duration greater than 1000?

To answer this question, let's assign the result of the boolean selection to a variable, and then compare the number of rows between it and the original DataFrame.

```
[9]: bikes_duration_1000 = bikes[filt]
```

Let's find the number of rows in each DataFrame.

```
[10]: len(bikes)
```

```
[10]: 50089
```

```
[11]: len(bikes_duration_1000)
```

```
[11]: 10178
```

We compute that 20% of the rides are longer than 1,000 seconds.

```
[12]: len(bikes_duration_1000) / len(bikes)
```

```
[12]: 0.20319830701351593
```

10.4 Boolean selection in one line

Often, you will see boolean selection completed in a single line of code instead of the two lines we used above. The expression for the filter is placed directly within *just the brackets*. Although this method will save a line of code, I recommend assigning the filter as a separate variable to help with readability.

```
[13]: bikes[bikes['tripduration'] > 1000].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
10	24383	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...	79.0	10.0	9.2	-9999.0	mostlycloudy

10.5 Single condition expression

Our first example tested a single condition (whether the trip duration was 1,000 or more). Let's test a different single condition and find all the rides that happened when the weather was cloudy. We use the `==` operator to test for equality and again pass this variable to *just the brackets* which completes our selection.

```
[14]: filt = bikes['events'] == 'cloudy'
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
6	18880	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565	...	66.0	10.0	15.0	-9999.0	cloudy
7	19689	Male	2013-07-03 09:07:00	2013-07-03 09:16:00	505	...	64.0	7.0	5.8	-9999.0	cloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy

10.6 Summary of single condition boolean selection

Boolean selection refers to the act of filtering your data based on the values, and not on the labels or integer location. There are two main steps to do boolean selection:

1. Create a boolean Series - commonly done by comparing one column of data to another value
2. Place the boolean Series inside *just the brackets* to filter the data

10.7 Exercises

Continue to use the bikes dataset for the first few exercises.

Exercise 1

Find all the rides with temperature below 0.

[]:

Exercise 2

Find all the rides with wind speed greater than 30.

[]:

Exercise 3

Find all the rides that began from station ‘Millennium Park’.

[]:

Exercise 4

Find all the rides with wind speed less than 0. How is this possible?

[]:

Exercise 5

Find all the rides where the starting number of bikes at the station (dpcapacity_start) was more than 50.

[]:

Exercise 6

Did any rides happen in temperature over 100 degrees?

[]:

Read in new data

Read in the movie dataset by executing the cell below and use it for the following exercises.

```
[15]: import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 7

Select all movies that have ‘Tom Hanks’ as `actor1`. How many of these movies has he starred in?

[]:

Exercise 8

Select movies with an IMDB score greater than 9.

[]:

Exercise 9

Write a function that accepts a single parameter to find the number of movies for a given content rating. Use the function to find the number of movies for ratings ‘R’, ‘PG-13’, and ‘PG’.

[]:

Chapter 11

Boolean Selection Multiple Conditions

Thus far, our boolean selections involved just a single condition. It is possible to have as many conditions as you would like. To do so, you will need to combine your boolean expressions using the three logical operators, and, or, and not.

11.1 Logical operators

Core Python provides the logical operators `and`, `or`, and `not` to combine multiple conditions together. These operators always return a boolean value. Let's take a look at a few simple examples to review.

We begin by testing whether five is greater than three and that 10 is greater than 20. There are two conditions here, with the first evaluating as `True` and the second as `False`. The `and` operator only returns `True` if both conditions are `True`, so in this case it returns `False`.

```
[1]: 5 > 3 and 10 > 20
```

```
[1]: False
```

Let's keep the same conditions and change the logical operator to `or` which returns `True` if one or more of the conditions evaluate as `True`.

```
[2]: 5 > 3 or 10 > 20
```

```
[2]: True
```

The `not` operator inverts a condition. Below, we invert the last expression. Because `not` has higher precedence than `or`, we use parentheses to ensure the `or` condition is evaluated first.

```
[3]: not (5 > 3 or 10 > 20)
```

```
[3]: False
```

Different logical operators for boolean Series

These built-in logical operators do not work for creating multiple conditions with a boolean Series. Instead, you must use the following operators.

- `&` for and (ampersand character)
- `|` for or (pipe character)
- `~` for not (tilde character)

Let's use the bikes dataset to make our multiple condition queries.

```
[4]: import pandas as pd
bikes = pd.read_csv('../data/bikes.csv')
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Our first multiple condition expression

Let's find all the rides longer than 1,000 seconds by males. This query has two conditions - trip durations greater than 1,000 and a gender of 'Male'. The way we approach the problem is to assign each condition to a separate variable. Since we desire both of the conditions to be true, we must use the and (`&`) operator. Each single condition is placed on its own line before using the `&` operator to create the final filter that completes the boolean selection.

```
[5]: filt1 = bikes['tripduration'] > 1000
filt2 = bikes['gender'] == 'Male'
filt = filt1 & filt2
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
10	24383	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...	79.0	10.0	9.2	-9999.0	mostlycloudy

11.2 Multiple conditions in one line

It is possible to combine the entire expression into a single line. Many pandas users like doing this, so it is a good idea to know how it's done as you will definitely encounter it.

Use parentheses to separate conditions

You must encapsulate each condition within a set of parentheses in order to make this work. Each condition is separated like this:

```
(bikes['tripduration'] > 1000) & (bikes['events'] == 'cloudy')
```

Same results

The above expression is placed inside of *just the brackets* to get the same results. Again, I prefer assigning each condition to its own variable name for better readability.

```
[6]: bikes[(bikes['tripduration'] > 1000) & (bikes['gender'] == 'Male')].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
10	24383	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...	79.0	10.0	9.2	-9999.0	mostlycloudy

11.3 Using an or condition

Let's find all the rides that were done by females **or** had trip durations longer than 1,000 seconds. In this example, we need at least one of the conditions to be true, which necessitates the use of the **or** (`|`) operator.

```
[7]: filt1 = bikes['tripduration'] > 1000
filt2 = bikes['gender'] == 'Female'
filt = filt1 | filt2
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
9	23558	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	...	81.0	10.0	12.7	-9999.0	mostlycloudy

11.4 Inverting a condition with the not operator

The tilde character, `~`, represents the not operator and inverts a condition. For instance, if we wanted all the rides with trip duration less than or equal to 1,000, we could do it like this:

```
[8]: filt = bikes['tripduration'] > 1000
bikes[~filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
3	12907	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...	72.0	10.0	16.1	-9999.0	mostlycloudy

Of course, inverting a single condition like this isn't too useful as we can use the less than or equal to operator instead.

```
[9]: filt = bikes['tripduration'] <= 1000
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
3	12907	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...	72.0	10.0	16.1	-9999.0	mostlycloudy

Invert a more complex condition

Typically, we save the not operator for reversing more complex conditions. Let's invert the condition for selecting rides by females or those with duration over 1,000 seconds. Logically, this should return only male riders with duration 1,000 or less. The ~ operator has precedence over the | operator, so we use parentheses to ensure that the or operation is completed first. That result is then inverted.

```
[10]: filt1 = bikes['tripduration'] > 1000
filt2 = bikes['gender'] == 'Female'
filt = ~(filt1 | filt2)
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
3	12907	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	...	72.0	10.0	16.1	-9999.0	mostlycloudy

Even more complex conditions

It is possible to build extremely complex conditions to select rows of your DataFrame that meet a very specific query. For instance, we can select males riders with trip duration between 5,000 and 10,000 seconds along with female riders with trip duration between 2,000 and 3,000 seconds. With multiple conditions, it's probably best to break out the logic into multiple steps:

```
[11]: filt1 = ((bikes['gender'] == 'Male') &
             (bikes['tripduration'] >= 5000) &
             (bikes['tripduration'] <= 10000))

filt2 = ((bikes['gender'] == 'Female') &
             (bikes['tripduration'] >= 2000) &
             (bikes['tripduration'] <= 3000))

filt = filt1 | filt2
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy
173	178550	Female	2013-08-08 08:49:00	2013-08-08 09:31:00	2502	...	71.1	10.0	10.4	-9999.0	mostlycloudy
258	253476	Female	2013-08-17 22:10:00	2013-08-17 22:53:00	2566	...	69.1	10.0	5.8	-9999.0	clear

11.5 Many equality conditions in a single column

Occasionally, we want to test equality in a single column with multiple values. This is most common in string columns. For instance, let's say we wanted to find all the rides where the events were either 'rain', 'snow', 'tstorms', or 'sleet'. One way to do this would be with four or conditions.

```
[12]: filt = ((bikes['events'] == 'rain') |
             (bikes['events'] == 'snow') |
             (bikes['events'] == 'tstorms') |
             (bikes['events'] == 'sleet'))
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
45	66336	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...	82.9	10.0	5.8	0.0	rain
78	89180	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	...	82.4	10.0	11.5	0.0	tstorms
79	89228	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	...	82.4	10.0	11.5	0.0	tstorms

Use the `isin` method instead

Instead of using an operator, we use the `isin` method. Pass it a list (or a set) of all the values you want to test equality with. The `isin` method returns a boolean Series and in this example, the same exact boolean Series as the previous one.

```
[13]: filt = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
45	66336	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...	82.9	10.0	5.8	0.0	rain
78	89180	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	...	82.4	10.0	11.5	0.0	tstorms
79	89228	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	...	82.4	10.0	11.5	0.0	tstorms

Combining `isin` with other filters

You can use the resulting boolean Series from the `isin` method in the same way as you would from the logical operators. For instance, If we wanted to find all the rides that had the same events as above and had a duration greater than 2,000 we would do the following:

```
[14]: filt1 = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
filt2 = bikes['tripduration'] > 2000
filt = filt1 & filt2
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2344	1266453	Female	2014-03-19 07:23:00	2014-03-19 08:00:00	2181	...	43.0	3.0	6.9	0.07	rain
7697	3557596	Male	2014-09-12 14:20:00	2014-09-12 14:57:00	2213	...	52.0	2.0	12.7	0.00	rain
8357	3801419	Male	2014-09-30 08:21:00	2014-09-30 08:58:00	2246	...	46.9	3.0	11.5	0.00	rain

```
[15]: bikes.dtypes
```

trip_id	int64
gender	object
starttime	object
stoptime	object
tripduration	int64
from_station_name	object
dpcapacity_start	float64
to_station_name	object

```
dpcapacity_end      float64  
temperature         float64  
visibility          float64  
wind_speed          float64  
precipitation       float64  
events              object  
dtype: object
```

[]:

11.6 Exercises

Continue to use the bikes dataset for the first few exercises.

Exercise 1

Find all the rides where visibility was between 0 and 2.

[]:

Exercise 2

Find all the rides with trip duration less than 100 done by females.

[]:

Exercise 3

Find all the rides from ‘Daley Center Plaza’ to ‘Michigan Ave & Washington St’.

[]:

Exercise 4

Find all the rides with temperature greater than 90 or visibility equal to 1 or wind speed greater than 20.

[]:

Exercise 5

Invert the condition from exercise 4.

[]:

Exercise 6

Are there any rides where the weather event was snow and the temperature was greater than 40?

[]:

Read in the movie dataset by executing the cell below and use it for the following exercises.

```
[16]: import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 7

Select all movies with an IMDB score between 8 and 9.

[]:

Exercise 8

Select all movies rated ‘PG-13’ that had IMDB scores between 8 and 9.

[]:

Exercise 9

Select movies that were rated either R, PG-13, or PG.

[]:

Exercise 10

Select movies that are either rated PG-13 or had an IMDB score greater than 7.

[]:

Exercise 11

Find all the movies that have at least one of the three actors with more than 10,000 Facebook likes.

[]:

Exercise 12

Invert the condition from exercise 10. In words, what have you selected?

[]:

Exercise 13

Select all movies from the 1970’s.

[]:

Chapter 12

Boolean Selection More

In this chapter, we explore several more possible ways to use boolean selection to filter data.

12.1 Boolean selection on a Series

All of the examples thus far have taken place on DataFrames. Boolean selection on a Series happens almost identically. Since there is only one dimension of data, the queries you ask are usually going to be simpler. First, let's select a single column of data as a Series such as the temperature column from the bikes dataset.

```
[1]: import pandas as pd  
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])  
temp = bikes['temperature']  
temp.head(3)
```

```
[1]: 0    73.9  
1    69.1  
2    73.0  
Name: temperature, dtype: float64
```

Let's select temperatures greater than 90. The procedure is the same as with DataFrames. Create a boolean Series and pass that Series to *just the brackets*.

```
[2]: filt = temp > 90  
temp[filt].head(3)
```

```
[2]: 54    91.0  
55    91.0  
56    91.0  
Name: temperature, dtype: float64
```

Select temperatures less than 0 or greater than 95. Multiple condition boolean Series also work the same.

```
[3]: filt1 = temp < 0  
filt2 = temp > 95  
filt = filt1 | filt2  
temp[filt].head()
```

```
[3]: 395    96.1
      396    96.1
      397    96.1
      1871   -2.0
      2049   -2.0
Name: temperature, dtype: float64
```

Set the index as starttime

The default index is not very helpful. Let's use the `set_index` method to make the `starttime` column the new index. While, this column may not be unique it does provide us with useful labels for each row.

```
[4]: bikes2 = bikes.set_index('starttime')
bikes2.head(3)
```

	trip_id	gender	stoptime	tripduration	from_station_name	...	temperature	visibility	wind_speed	precipitation	events
starttime											
2013-06-28 19:01:00	7147	Male	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	73.9	10.0	12.7	-9999.0	mostlycloudy
2013-06-28 22:53:00	7524	Male	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	69.1	10.0	6.9	-9999.0	partlycloudy
2013-06-30 14:43:00	10927	Male	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	73.0	10.0	16.1	-9999.0	mostlycloudy

Let's get back our temperature Series with its updated index.

```
[5]: temp2 = bikes2['temperature']
temp2.head()
```

```
[5]: starttime
2013-06-28 19:01:00    73.9
2013-06-28 22:53:00    69.1
2013-06-30 14:43:00    73.0
2013-07-01 10:05:00    72.0
2013-07-01 11:16:00    73.0
Name: temperature, dtype: float64
```

Let's select temperatures greater than 90. We expect to get a summer month and we do.

```
[6]: filt = temp2 > 90
temp2[filt].head(5)
```

```
[6]: starttime
2013-07-16 15:13:00    91.0
2013-07-16 15:31:00    91.0
2013-07-16 16:35:00    91.0
2013-07-17 17:08:00    93.0
2013-07-17 17:25:00    93.0
Name: temperature, dtype: float64
```

Select temperature less than 0 or greater than 95. We expect to get some winter months in the result and we do.

```
[7]: filt1 = temp2 < 0
filt2 = temp2 > 95
filt = filt1 | filt2
temp2[filt].head()
```

```
[7]: starttime
2013-08-30 15:33:00    96.1
2013-08-30 15:37:00    96.1
2013-08-30 15:49:00    96.1
2013-12-12 05:13:00   -2.0
2014-01-23 06:15:00   -2.0
Name: temperature, dtype: float64
```

12.2 The between method

The `between` method returns a boolean Series by testing whether the current value is between two given values. For instance, if want to select the temperatures between 50 and 60 degrees we do the following:

```
[8]: filt = temp2.between(50, 60)
filt.head(3)
```

```
[8]: starttime
2013-06-28 19:01:00    False
2013-06-28 22:53:00    False
2013-06-30 14:43:00    False
Name: temperature, dtype: bool
```

By default, the `between` method is inclusive of the given values, so temperatures of exactly 50 or 60 would be found in the result. We pass this boolean Series to *just the brackets* to complete the selection.

```
[9]: temp2[filt].head(3)
```

```
[9]: starttime
2013-09-13 07:55:00    54.0
2013-09-13 08:04:00    57.9
2013-09-13 08:04:00    57.9
Name: temperature, dtype: float64
```

12.3 Simultaneous boolean selection of rows and column labels with loc

The `loc` indexer was thoroughly covered in an earlier chapter and will now be brought up again to show how it can simultaneously select rows with boolean selection and columns by labels.

Remember that `loc` takes both a row selection and a column selection separated by a comma. Since the row selection comes first, you can pass it the same exact inputs that you do for *just the brackets* and get the same results. Let's run some of the previous examples of boolean selection with `loc`. Here, we select all rides with trip duration greater than 1,000.

```
[10]: filt = bikes['tripduration'] > 1000
bikes.loc[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
10	24383	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...	79.0	10.0	9.2	-9999.0	mostlycloudy

Here, we select all weather events that are either rain, snow, tstorms, or sleet.

```
[11]: filt = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
bikes.loc[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
45	66336	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...	82.9	10.0	5.8	0.0	rain
78	89180	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	...	82.4	10.0	11.5	0.0	tstorms
79	89228	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	...	82.4	10.0	11.5	0.0	tstorms

Separate row and column selection with a comma for loc

The nice benefit of `loc` is that it allows us to simultaneously select rows with boolean selection and select columns by label. Let's select rides during rain or snow and the columns `events` and `tripduration`.

```
[12]: filt = bikes['events'].isin(['rain', 'snow'])
cols = ['events', 'tripduration']
bikes.loc[filt, cols].head()
```

	events	tripduration
45	rain	727
112	rain	1395
124	rain	442
161	rain	890
498	rain	978

Now let's find all female riders with trip duration greater than 5,000 when it was cloudy. We'll only return the columns used during the boolean selection.

```
[13]: filt1 = bikes['gender'] == 'Female'
filt2 = bikes['tripduration'] > 5000
filt3 = bikes['events'] == 'cloudy'
filt = filt1 & filt2 & filt3
cols = ['gender', 'tripduration', 'events']
bikes.loc[filt, cols]
```

	gender	tripduration	events
2712	Female	79988	cloudy
14455	Female	7197	cloudy
22868	Female	13205	cloudy
36441	Female	19922	cloudy

12.4 Column to column comparisons

So far, we created filters by comparing each of our column values to a single scalar value. It is possible to do element-by-element comparisons by comparing two columns to one another. For instance, the total bike capacity at each station at the start and end of the ride is stored in the `dpcapacity_start` and `dpcapacity_end` columns. If we wanted to test whether there were more capacity at the start of the ride vs the end, we would do the following:

```
[14]: filt = bikes['dpcapacity_start'] > bikes['dpcapacity_end']
```

Let's use this filter with `loc` to return all the rows where the start capacity is greater than the end.

```
[15]: cols = ['dpcapacity_start', 'dpcapacity_end']
bikes.loc[filt, cols].head(3)
```

	dpcapacity_start	dpcapacity_end
1	31.0	19.0
6	31.0	19.0
8	31.0	15.0

Boolean selection with `iloc` does not work

The pandas developers decided not to allow boolean selection with `iloc`. The following raises an error.

```
[16]: bikes.iloc[filt]
```

```
NotImplementedError: iLocation based boolean indexing on an integer type is not available
```

12.5 Finding Missing Values with `isna`

The `isna` method called from either a DataFrame or a Series returns `True` for every value that is missing and `False` for any other value. Let's see this in action by calling `isna` on the start capacity column.

```
[17]: bikes['dpcapacity_start'].isna().head(3)
```

```
[17]: 0    False
      1    False
      2    False
Name: dpcapacity_start, dtype: bool
```

Filtering for missing values

We can now use this boolean Series to select all the rows where the capacity start column is missing. Verify that those values are indeed missing.

```
[18]: filt = bikes['dpcapacity_start'].isna()
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
17566	7319012	Male	2015-09-06 07:52:00	2015-09-06 07:55:00	207	...	75.0	10.0	4.6	-9999.0	mostlycloudy
17605	7341764	Female	2015-09-07 09:52:00	2015-09-07 09:57:00	293	...	81.0	10.0	8.1	-9999.0	mostlycloudy
17990	7468970	Male	2015-09-15 08:25:00	2015-09-15 08:33:00	473	...	68.0	10.0	9.2	-9999.0	mostlycloudy

`isnull` is an alias for `isna`

There is an identical method named `isnull` that you will see in other tutorials. It is an **alias** of `isna` meaning it does the exact same thing but has a different name. Either one is suitable to use, but I prefer `isna` because of the similarity to `NAN`, the representation of missing values. There are also other methods such as `dropna` and `fillna` that have 'na' in their names.

12.6 Exercises

Continue to use the bikes dataset for the first few exercises.

Exercise 1

Select the wind speed column as a Series and assign it to a variable. Are there any negative wind speeds?

```
[ ]:
```

Exercise 2

Select all wind speed values between 12 and 16.

```
[ ]:
```

Exercise 3

Select the events and gender columns for all trip durations longer than 1,000 seconds.

[]:

Read in the movie dataset by executing the cell below and use it for the following exercises.

[19]:

```
import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 4

Select all the movies such that the Facebook likes for actor 2 are greater than those for actor 1.

[]:

Exercise 5

Select the year, content rating, and IMDB score columns for movies from the year 2016 with IMDB score less than 4.

[]:

Exercise 6

Select all the movies that are missing values for content rating.

[]:

Exercise 7

Select all the movies that are missing values for both the gross and budget columns. Return just those columns to verify that those values are indeed missing.

[]:

Exercise 8

Write a function `find_missing` that has three parameters, `df`, `col1` and `col2` where `df` is a DataFrame and `col1` and `col2` are column names. This function should return all the rows of the DataFrame where `col1` and `col2` are missing. Only return the two columns as well. Answer problem 7 with this function.

[]:

Chapter 13

Filtering with the `query` Method

The previous chapters on boolean selection showed us how to filter our DataFrames and Series based on their values. We created conditions, usually involving the comparison operators, resulting in boolean Series and passed them to *just the brackets* to filter the data.

In this chapter we cover the `query` method, which enables us to also make selections based on the values of the DataFrame or Series. The `query` method is easier and more intuitive to use than boolean selection, but doesn't provide as much functionality to filter the data. Still, it is a good method to know about to make your subset selections more readable.

13.1 The `query` method

The `query` method allows you to filter the data by writing the condition as a string. For instance, you would use the string '`tripduration > 1000`' to select all rows of the `bikes` dataset that have a `tripduration` greater than 1,000. Let's read in the bikes dataset and run this command now.

```
[1]: import pandas as pd  
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])  
bikes.query('tripduration > 1000').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy
10	24383	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	...	79.0	10.0	9.2	-9999.0	mostlycloudy

Less syntax and more readable

The `query` method generally uses less syntax than boolean selection and is usually more readable. For instance, the following reproduces the last result with boolean selection:

```
bikes[bikes['tripduration'] > 1000]
```

This looks a bit clumsy with the name `bikes` written twice right next to one another. The `query` method has its own set of rules for what constitutes a correctly written condition within the string you pass it. The

rest of this chapter covers all of the available functionality of the `query` method. This syntax only works within the `query` method and is not allowed anywhere else in pandas.

Use strings and, or, not

Unlike boolean selection, you can use the strings `and`, `or`, and `not` instead of the operators `&`, `|`, and `~` which further aides readability with `query`. Let's select all rides with `tripduration` greater than 1,000 and `temperature` greater than 85.

```
[2]: bikes.query('tripduration > 1000 and temperature > 85').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
53	68864	Male	2013-07-16 13:04:00	2013-07-16 13:28:00	1435	...	90.0	10.0	8.1	-9999.0	mostlycloudy
60	71812	Male	2013-07-17 10:23:00	2013-07-17 10:40:00	1024	...	88.0	10.0	5.8	-9999.0	partlycloudy

Chained comparisons

Let's say we want to find all rides where the temperature was between 50 and 60 degrees. You can do this with `query` by using the `and` operator.

```
[3]: bikes.query('temperature >= 50 and temperature <= 60').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
590	483022	Female	2013-09-13 07:55:00	2013-09-13 08:01:00	319	...	54.0	10.0	15.0	-9999.0	partlycloudy
591	483128	Male	2013-09-13 08:04:00	2013-09-13 08:16:00	738	...	57.9	10.0	13.8	-9999.0	partlycloudy
592	483127	Female	2013-09-13 08:04:00	2013-09-13 08:14:00	599	...	57.9	10.0	13.8	-9999.0	partlycloudy

While this syntax is valid, there is a better way. You can use a **chained comparison** to make the string even more readable and concise. A chained comparison places the column name between two comparison operators. The following implies that 50 is less than or equal to the temperature and the temperature is less than or equal to 60 which is equivalent to our previous selection.

```
[4]: bikes.query('50 <= temperature <= 60').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
590	483022	Female	2013-09-13 07:55:00	2013-09-13 08:01:00	319	...	54.0	10.0	15.0	-9999.0	partlycloudy
591	483128	Male	2013-09-13 08:04:00	2013-09-13 08:16:00	738	...	57.9	10.0	13.8	-9999.0	partlycloudy
592	483127	Female	2013-09-13 08:04:00	2013-09-13 08:14:00	599	...	57.9	10.0	13.8	-9999.0	partlycloudy

Reference strings with quotes

If you would like to reference a literal string within `query`, you need to surround it in quotes, or else pandas will attempt to use it as a column name. Let's select all rides done by a 'Female' with a trip duration greater than 2,000.

```
[5]: bikes.query('gender == "Female" and tripduration > 2000').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy
173	178550	Female	2013-08-08 08:49:00	2013-08-08 09:31:00	2502	...	71.1	10.0	10.4	-9999.0	mostlycloudy

Forgetting quotes

If you do not use quotes around your literal string, then pandas assumes that value is a column name. The following raises an error. It believes you are accessing a column name `Female`, but that doesn't exist.

```
[6]: bikes.query('gender == Female and tripduration > 2000')
```

```
UndefinedVariableError: name 'Female' is not defined
```

Column to column comparisons

It is possible to compare each value in one column with each value in another column. Here, we filter for all the rides where there were more bikes at the start than at the end.

```
[7]: bikes.query('dpcapacity_start > dpcapacity_end').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
6	18880	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565	...	66.0	10.0	15.0	-9999.0	cloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy

Use ‘in’ for multiple equalities

You can check whether each value in a column is equal to one or more other values by using the word ‘in’ within your query. Use the syntax for creating a list within the query string to contain all the values you’d like to check. The following tests whether the ride weather event was snow or rain.

```
[8]: bikes.query('events in ["snow", "rain"]').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
45	66336	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	...	82.9	10.0	5.8	0.0	rain
112	111568	Male	2013-07-26 19:10:00	2013-07-26 19:33:00	1395	...	66.9	8.0	12.7	0.0	rain
124	130156	Male	2013-07-30 18:53:00	2013-07-30 19:00:00	442	...	69.1	10.0	3.5	0.0	rain

There are multiple syntaxes for the above that all work the same, but I prefer using the above as it is most similar to the `isin` method used during boolean selection.

- `bikes.query('["snow", "rain"] in events')`
- `bikes.query('["snow", "rain"] == events')`
- `bikes.query('events == ["snow", "rain"]')`

Use ‘not in’ to invert the condition

You can invert the result of an ‘in’ clause by placing the word ‘not’ before it. Here, we find all the rides that did not have the weather events cloudy, partly cloudy or mostly cloudy.

```
[9]: bikes.query('events not in ["cloudy", "partlycloudy", "mostlycloudy"]').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
25	47798	Female	2013-07-11 08:17:00	2013-07-11 08:31:00	830	...	73.9	10.0	8.1	-9999.0	clear
26	51130	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043	...	64.9	10.0	0.0	-9999.0	clear
33	53963	Male	2013-07-12 17:22:00	2013-07-12 17:34:00	730	...	79.0	10.0	10.4	-9999.0	clear

Arithmetic operations within query

It is possible to write arithmetic operations within `query` just as you would outside of it. For instance, if we wanted to find all the rides such that there were 20 or more bikes at the start station than at the end, we do the following.

```
[10]: bikes.query('dpcapacity_start - dpcapacity_end >= 20').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
54	69250	Male	2013-07-16 15:13:00	2013-07-16 15:18:00	347	...	91.0	10.0	8.1	-9999.0	mostlycloudy
66	73852	Male	2013-07-17 20:56:00	2013-07-17 21:14:00	1073	...	86.0	10.0	9.2	-9999.0	partlycloudy
116	112863	Male	2013-07-27 09:54:00	2013-07-27 09:56:00	121	...	60.8	10.0	11.5	-9999.0	cloudy

Filtering for right triangles

Let's read in the triangles dataset which contains the lengths of each side of a triangle as the columns `a`, `b`, and `c`.

```
[11]: triangles = pd.read_csv('../data/triangles.csv')
triangles.head()
```

	a	b	c
0	2	3	4
1	3	2	4
2	3	4	5
3	3	5	6
4	3	6	7

We can use the `query` method to find all the right triangles, those that satisfy the Pythagorean Theorem. We write the condition using the arithmetic and comparison operators.

```
[12]: triangles.query('a ** 2 + b ** 2 == c ** 2').head()
```

	a	b	c
	2	3	4 5
	5	4	3 5
	14	5	12 13
	21	6	8 10
	33	7	24 25

The syntax is quite a bit nicer than the boolean selection alternative.

```
[13]: filt = triangles['a'] ** 2 + triangles['b'] ** 2 == triangles['c'] ** 2
triangles[filt].head()
```

	a	b	c
	2	3	4 5
	5	4	3 5
	14	5	12 13
	21	6	8 10
	33	7	24 25

Use the @ symbol to reference a variable name

By default, all words within the query string attempt to reference a column name. You can, however, reference a variable name by preceding it with the @ symbol. Let's assign the variable name `min_length` to 5,000 and reference it in a query to find all the rides where trip duration was greater than it.

```
[14]: min_length = 5000
bikes.query('tripduration > @min_length').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy

Using the index with query

You can even use the word `index` to make comparisons against the index as if it were a normal column. In the bikes DataFrame, the index is just the integers beginning at 0. Here, we select only the `events` that were 'cloudy' for an index value greater than 4,000.

```
[15]: bikes.query('index > 4000 and events == "cloudy" ').head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
4007	2003400	Male	2014-06-07 14:07:00	2014-06-07 14:31:00	1434	...	82.0	10.0	13.8	-9999.0	cloudy
4008	2004978	Male	2014-06-07 14:58:00	2014-06-07 15:19:00	1258	...	82.0	10.0	13.8	-9999.0	cloudy
4009	2005778	Male	2014-06-07 15:23:00	2014-06-07 15:28:00	297	...	80.1	10.0	13.8	-9999.0	cloudy

Using column names with spaces

pandas allows DataFrames to have column names with spaces in them. In order to use a column name containing spaces within `query`, you'll need to surround it with backticks. If you don't use the backticks you'll get an error. Let's read in the San Francisco employee compensation dataset which contains multiple column names that have spaces.

```
[16]: sf_emp = pd.read_csv('../data/sf_employee_compensation.csv')
sf_emp.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

Let's find all the employees that are in the organization group of 'Public Protection'.

```
[17]: sf_emp.query(`organization group` == "Public Protection").head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20
7	2013	Public Protection	Police Officer	78591.02	2050.18	832.9	15383.49	11004.42	4471.61

Selecting columns with query

Unfortunately the `query` method does not give us the ability to select a subset of the columns when filtering the data. You would have to do normal column selection after calling the method. Here, we use *just the brackets* to select three columns after finding all the rides where the weather was snow or rain.

```
[18]: cols = ['starttime', 'temperature', 'events']
bikes.query('events in ["snow", "rain"]')[cols].head()
```

	starttime	temperature	events
45	2013-07-15 16:43:00	82.9	rain
112	2013-07-26 19:10:00	66.9	rain
124	2013-07-30 18:53:00	69.1	rain
161	2013-08-05 17:09:00	68.0	rain
498	2013-09-07 16:09:00	81.0	rain

13.2 Summary

The `query` method provides an alternative to boolean selection to filter the data based on the values. Here are the rules for the string you provide.

- The expression in the string must evaluate as True or False for every row
- Column names may be accessed directly with their name
- Often you will use one of the comparison operators to create a condition
- Use `and`, `or`, and `not` to create more complex conditions
- To use a literal string, surround it with quotes
- Use chained comparison operators to shorten syntax
- Use `in` to test multiple equalities. Provide the test values in a list
- All arithmetic operators work just as they do outside of the string
- Use the `@` character to reference a variable name
- Use backticks to reference a column name with spaces in it

13.3 Exercises

Use the `bikes` dataset for the first few exercises.

Exercise 1

Use the `query` method to select trip durations between 5,000 and 10,000.

```
[ ]:
```

Exercise 2

Use the `query` method to select trip durations between 5,000 and 10,000 when the weather was snow or rain. Retrieve the same data with boolean selection.

```
[ ]:
```

Exercise 3

Use the `query` method to select trip durations between 5,000 and 10,000 when it was snow or rain. Create a list outside of the `query` method to hold the weather and reference that variable with `@` within `query`.

[]:

Read in the movie dataset by executing the cell below and use it for the following exercises.

```
[19]: import pandas as pd
pd.set_option('display.max_columns', 50)
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

Exercise 4

Use the `query` method to find all movies where the total number of Facebook likes for all three actors is greater than 50,000.

[]:

Exercise 5

Select all the movies where the number of user voters is less than 10 times the number of reviews.

[]:

Exercise 6

Select all the movies made in the 1990's that were rated R with an IMDB score greater than 8.

[]:

Chapter 14

Miscellaneous Subset Selection

In this chapter, a few more methods for subset selection are described. The methods used in this chapter do not add any additional functionality to pandas, but are covered for completeness.

I personally do not use the methods described in this chapter and suggest that you also avoid them. They are all valid syntax and some pandas users do actually use them, so you may find them valuable.

14.1 Selecting a column with dot notation

The best way to select a single column from a DataFrame as a Series is by placing the name of the column within *just the brackets*. There's actually an alternative way to select a single column of data and that is with dot notation. Let's read in the the `sample_data2.csv` dataset.

```
[1]: import pandas as pd  
df = pd.read_csv('../data/sample_data2.csv')  
df
```

	name	average score	max
0	Niko	99	100
1	Penelope	100	102
2	Aria	88	93

Place the name of the column directly after the name dot as if it were an attribute.

```
[2]: df.name.head()
```

```
[2]: 0      Niko  
1      Penelope  
2      Aria  
Name: name, dtype: object
```

This produces an identical result as using *just the brackets*.

```
[3]: df['name'].head()
```

```
[3]: 0      Niko
     1    Penelope
     2       Aria
Name: name, dtype: object
```

Avoid dot notation - use just the brackets

Although this method for column selection requires less syntax and is used by many pandas users, it has many downsides. The following is a partial list of the functionality that is impossible using dot notation.

- Select column names with spaces
- Select column names that have the same name as methods
- Select columns with variables

Examples of each scenario will now be covered. Using dot notation does not allow you to select columns with spaces. Selecting the column `average score` raises a syntax error.

```
[4]: df.average score
```

SyntaxError: invalid syntax

The only way to select this column is with *just the brackets*.

```
[5]: df['average score']
```

```
[5]: 0    99
     1   100
     2    88
Name: average score, dtype: int64
```

Dot notation is unable to select columns that are the same name as methods. For instance, `max` is a method that all DataFrames have. In this particular DataFrame, it also the name of the column. Attempting to select it via dot notation will access the method.

```
[6]: df.max
```

```
[6]: <bound method DataFrame.max of          name  average score  max
  0      Niko           99    100
  1  Penelope         100    102
  2      Aria           88    93>
```

Again, the only way to select this column is with *just the brackets*.

```
[7]: df['max']
```

```
[7]: 0    100
     1   102
     2    93
Name: max, dtype: int64
```

Dot notation is unable to select a column using a variable name. Let's say we assign the variable `col` to the string 'name' which is the name of the first column. Attempting to select it via dot notation raises an error.

```
[8]: col = 'name'  
df.col
```

```
AttributeError: 'DataFrame' object has no attribute 'col'
```

Once again, use *just the brackets*.

```
[9]: df[col]
```

```
[9]: 0      Niko  
1      Penelope  
2      Aria  
Name: name, dtype: object
```

Video with 10 reasons why using the brackets are superior

There are actually many more reasons to use the brackets over dot notation. If you are interested in hearing all of my reasons, [watch this video](#).

14.2 Slicing rows with just the brackets

So far, we have covered three ways to select subsets of data with *just the brackets*. You can use a single string, a list of strings, or a boolean Series. Let's quickly review those ways right now using the bikes dataset.

```
[10]: bikes = pd.read_csv('../data/bikes.csv')
```

A single string

```
[11]: bikes['tripduration'].head(3)
```

```
[11]: 0      993  
1      623  
2     1040  
Name: tripduration, dtype: int64
```

A list of strings

```
[12]: cols = ['trip_id', 'tripduration']  
bikes[cols].head(3)
```

	trip_id	tripduration
0	7147	993
1	7524	623
2	10927	1040

A boolean Series

The previous two examples selected columns. Boolean Series select rows.

```
[13]: filt = bikes['tripduration'] > 5000
bikes[filt].head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
18	40924	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	...	79.0	10.0	13.8	0.0	cloudy
40	61401	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	...	87.1	10.0	8.1	-9999.0	partlycloudy
77	87005	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	...	82.9	10.0	5.8	-9999.0	mostlycloudy

Using a slice

It is possible to use slice notation within just the brackets. For example, the following selects the rows beginning at location 2 up to location 10 with a step size of 3. You can even use slice notation when the index is strings.

```
[14]: bikes[2:10:3]
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy
5	13595	Male	2013-07-01 12:37:00	2013-07-01 12:48:00	660	...	73.0	10.0	17.3	-9999.0	mostlycloudy
8	21028	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	...	71.1	8.0	0.0	-9999.0	cloudy

I do not recommend using slicing with *just the brackets*

Although slicing with *just the brackets* seems simple, I do not recommend using it. This is because it is ambiguous and can make selections either by integer location or by label. I always prefer explicit, unambiguous methods. Both `loc` and `iloc` are unambiguous and explicit. Meaning that even without knowing anything about the DataFrame, you would be able to explain exactly how the selection will take place. If you do want to slice the rows, then use `loc` if you are using labels or `iloc` if you are using integer location, but do not use *just the brackets*.

14.3 Selecting a single cell with `at` and `iat`

pandas provides two more rarely seen indexers, `at`, and `iat`. These indexers are analogous to `loc` and `iloc` respectively, but only select a single cell of a DataFrame. Since they only select a single cell, you must pass both a row and column selection as either a label (`loc`) or an integer location (`iloc`). Let's see an example of each.

```
[15]: bikes.at[40, 'temperature']
```

```
[15]: 87.1
```

```
[16]: bikes.iat[-30, 5]
```

```
[16]: 'Marshfield Ave & Cortland St'
```

The current index labels for `bikes` is integers which is why the number 40 was used above. It is the label for a row, but also happens to be an integer.

What's the purpose of these indexers?

All usages of `at` and `iat` may be replaced with `loc` and `iloc` and would produce the exact same results. These `at` and `iat` indexers are optimized to select a single cell of data and therefore provide slightly better performance than `loc` or `iloc`. Let's verify this below.

```
[17]: bikes.loc[40, 'temperature']
```

```
[17]: 87.1
```

```
[18]: bikes.iloc[-30, 5]
```

```
[18]: 'Marshfield Ave & Cortland St'
```

I never use these indexers

Personally, I never use these specialty indexers as the performance advantage for a single selection is minor. It would require a case where single element selections were happening in great numbers to see any significant improvement and doing so is rare in data analysis.

Much bigger performance improvement using numpy directly

If you truly wanted a large performance improvement for single-cell selection, you would select directly from numpy arrays and not a pandas DataFrame. Below, the data is extracted into the underlying numpy array with the `values` attribute. We then time the performance for selecting with numpy and also with `iat` and `iloc` on a DataFrame. On my machine, `iat` shows a 30-40% improvement over `iloc`, but selecting with numpy is about 50x as fast. There is no comparison here, if you care about performance for selecting a single cell of data, use numpy.

```
[19]: values = bikes.values
```

```
[20]: %timeit -n 5 values[-30, 5]
```

```
187 ns ± 107 ns per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

```
[21]: %timeit -n 5 bikes.iat[-30, 5]
```

```
6.96 µs ± 2.76 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

```
[22]: %timeit -n 5 bikes.iloc[-30, 5]
```

```
12.3 µs ± 2.58 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```


Part III

Essential Series Commands

Chapter 15

Series Attributes and Statistical Methods

Our main accomplishment up to this point has been selecting subsets of data. We have not changed the data or made many interesting calculations. Our selections have happened in two ways:

- Selection by label and integer location
- Selection by actual values (boolean selection and the `query` method)

Other than `query`, these selections all used the square brackets.

15.1 Calling methods on a Series

In this chapter we will call many methods that perform actions on a Series. We have actually already called some methods such as the `head`, `tail`, `isna`, and `set_index`. There are over 200 methods available to both the Series and DataFrame.

Use a subset of methods

It can be quite overwhelming to think about having to learn and memorize this staggering amount of functionality. The good news is that many of these methods are unnecessary and don't add any extra functionality. Furthermore, many methods are remnants from the early days of pandas and have few/no use cases or have been **deprecated**. When a method is deprecated, then it is both discouraged from being used and will likely be removed from the library in the future. For example, the `ix` indexer was useful in the earlier days of pandas to simultaneously select rows and columns of data. It was deprecated in favor of the `loc` and `iloc` indexers and removed in pandas version 1.0.

Minimally sufficient pandas

I suggest using a subset of the pandas library that allows you to do as many tasks as possible. I focus on the subset of pandas that maximizes both performance and readability. Since there is so much functionality, power users of pandas can think of very creative and complex code to accomplish different tasks. This is not necessarily a positive thing, and when working with a group of other data analysts can lead to confusion for those that are not familiar with the syntax. One of my most popular blog posts is titled [Minimally Sufficient Pandas](#) and goes into greater detail on this.

15.2 Series Attributes and Methods

We begin our exploration of attributes and methods with Series objects. It is far simpler to focus on a single column of data than multiple columns in a DataFrame.

View the API for a complete list of functionality

Modern programming languages use the term **Application Programming Interface** or **API** to list and describe all the possible functionality therein. The pandas API reference can be found [here](#). This is a huge list, but as mentioned above, only a subset of this page is needed for the vast majority of tasks.

The best of the pandas Series API

The pandas Series object is a single dimension of data and easier to work with than an entire DataFrame. We start with it and cover the most basic and important methods below. You may find it useful to navigate to the [Series API](#) section of the documentation so that you can have a full list of the functionality.

City of Houston Employee Data

We will use a public dataset containing City of Houston employee information on their position, race, sex, and salary. This dataset was last updated in July of 2019 and contains nearly all of the employees for the City of Houston. Notice that the column `hire_date` can be read in as a datetime.

```
[1]: import pandas as pd
emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Select a single column as a Series

Let's select the `salary` column as a Series and use it to explore the Series API.

```
[2]: salary = emp['salary']
salary.head()
```

```
[2]: 0    87545.38
1    82182.00
2    49275.00
3    75942.10
4    69355.26
Name: salary, dtype: float64
```

Let's verify that we have a Series object.

```
[3]: type(salary)
```

```
[3]: pandas.core.series.Series
```

15.3 Core Series attributes

pandas Series have [many attributes](#), but only a few are important to know. The attributes to be aware of are:

- `index`
- `values`
- `shape`
- `size`
- `dtype`

The `index` and `values` were covered in a previous chapter. Only `shape`, `size` and `dtype` are new. The `shape` and `size` attributes are nearly identical to one another. They both return the number of values in the Series. The `shape` returns it as a one-item tuple, while `size` returns an integer. The `dtype` attribute returns the data type of the values. Remember that all values in the Series share the same data type. Let's display these now.

```
[4]: salary.shape
```

```
[4]: (24308,)
```

```
[5]: salary.size
```

```
[5]: 24308
```

```
[6]: salary.dtype
```

```
[6]: dtype('float64')
```

len function also returns the number of values

The built-in `len` function returns the same number as the `size` attribute.

```
[7]: len(salary)
```

```
[7]: 24308
```

Even though they both report the same number, I typically use the `len` function, as it returns the number of rows when used on a DataFrame. The DataFrame `size` attribute returns the total number of values in the DataFrame.

15.4 Arithmetic operators

Series have the ability to work with all of the following common arithmetic operators:

- `+` - Addition
- `-` - Subtraction
- `*` - Multiplication
- `/` - Division
- `//` - Floor division
- `**` - Exponentiation
- `%` - Modular division (returns the remainder)

All of the arithmetic operators operate on every value in the Series. Let's see some examples beginning by adding 5 to every value in the Series.

```
[8]: result = salary + 5
result.head(3)
```

```
[8]: 0    87550.38
      1    82187.00
      2    49280.00
Name: salary, dtype: float64
```

Let's raise each value in the Series to the .2 power with the exponentiation operator, **.

```
[9]: result = salary ** .2
result.head(3)
```

```
[9]: 0    9.737482
      1    9.615134
      2    8.680112
Name: salary, dtype: float64
```

Let's divide each value in the Series by 173. This single forward slash is referred to as **true division** and returns all decimal values.

```
[10]: result = salary / 173
result.head(3)
```

```
[10]: 0    506.042659
      1    475.040462
      2    284.826590
Name: salary, dtype: float64
```

Two forward slashes are used for **floor division**. The decimals are truncated (and not rounded) from the result.

```
[11]: result = salary // 173
result.head(3)
```

```
[11]: 0    506.0
      1    475.0
      2    284.0
Name: salary, dtype: float64
```

Isn't this chapter about calling methods?

Although the above operations are not actual methods and do not use dot notation, they work similarly as methods. You can think of them as methods that take exactly one parameter, the other object that is being operated on.

Arithmetic operations are vectorized

All the above arithmetic operations are **vectorized**. This means that the operation was applied to each value in the Series without an explicit writing of a for-loop. Python lists do not work like this and require an explicit for-loop to operate on each value.

15.5 Comparison operations

The following six comparison operators work similarly as their arithmetic analogs from above:

- < - Less than

- `<=` - Less than or equal to
- `>` - Greater than
- `>=` - Greater than or equal to
- `==` - Equals to
- `!=` - Not equal to

In the boolean selection chapters, we used these vectorized comparison operations (without the terminology) to produce a Series of booleans. Let's see a few examples below beginning by testing whether each salary is greater than 50,000.

```
[12]: result = salary > 50000
result.head(3)
```

```
[12]: 0    True
      1    True
      2   False
Name: salary, dtype: bool
```

Here, we test whether each salary is not equal to 82,182

```
[13]: result = salary != 82182
result.head(3)
```

```
[13]: 0    True
      1   False
      2    True
Name: salary, dtype: bool
```

15.6 Boolean and bitwise operators

Python has three boolean operators, the keywords `and`, `or`, and `not`. These operators are syntactically unable to do vectorized boolean operations. Instead, pandas and numpy rely on the bitwise and, or, and not operators, respectively `&`, `|`, and `~` to perform vectorized boolean operations. They were thoroughly covered in the preceding chapters. Let's do one example as a review and determine whether or not a salary is less than 50,000 or greater than 100,000.

```
[14]: result = (salary < 50000) | (salary > 100000)
result.head(3)
```

```
[14]: 0   False
      1   False
      2    True
Name: salary, dtype: bool
```

15.7 Statistical methods

We now call *actual* methods that compute [basic descriptive statistics](#) on a numerical Series. You might want to click the previous link to have the list of all the possible statistical methods. We call the methods explicitly with dot notation. It is useful to place these methods into two categories - those that **aggregate** and those that do not.

Aggregation methods

A method that performs an aggregation returns a **single** number to summarize the Series. Examples of methods that aggregate are:

- `sum`
- `min`
- `max`
- `mean`
- `median`
- `std` - standard deviation
- `var` - variance
- `count` - returns number of non-missing values
- `describe` - returns most of the above aggregations in one Series
- `quantile` - returns the given percentile of the distribution

Non-aggregation methods

Any other method that does not return a single value is not an aggregation. Some examples of these methods are:

- `abs` - takes absolute value
- `round` - round to the nearest given decimal place
- `cummin` - cumulative minimum
- `cummax` - cumulative maximum
- `cumsum` - cumulative sum

15.8 Aggregation methods

Let's see a few examples of common aggregation methods. We begin by summing every value in the Series with the `sum` method.

```
[15]: salary.sum()
```

```
[15]: 1359826363.82
```

Get the minimum value of the Series with the `min` method.

```
[16]: salary.min()
```

```
[16]: 9912.0
```

Get the maximum value of a Series with the `max` method.

```
[17]: salary.max()
```

```
[17]: 342784.0
```

Use the `quantile` method to return the given percentile of the Series. It accepts values between 0 and 1. By default, it returns the 50th percentile. Below, we pass it .95 to return the 95th percentile of salary. This means that 95 percent of the employees for the City of Houston have this salary or below.

```
[18]: salary.quantile(.95)
```

```
[18]: 96063.9030000001
```

The `count` method

The `count` method returns the number of non-missing values. It does NOT return the total number of values in the Series. Since this number is less than `len(salary)`, we know missing values exist.

```
[19]: salary.count()
```

```
[19]: 23362
```

```
[20]: len(salary)
```

```
[20]: 24308
```

pandas ignores missing values by default

One big difference between pandas and numpy is that pandas ignores missing values by default. When calling aggregation methods such as `sum` or `mean`, pandas ignores any missing value as if that piece of data did not exist. On the other hand, numpy returns `nan` for its aggregation methods when one or more values are missing. Let's verify this by extracting the values of `salary` as a numpy array and then calling the array `sum` method.

```
[21]: salary.values.sum()
```

```
[21]: nan
```

We can make pandas Series behave like numpy by setting the `skipna` parameter to `False`. All of the statistical methods have the `skipna` parameter available.

```
[22]: salary.sum(skipna=False)
```

```
[22]: nan
```

The `describe` method

The `describe` method returns several aggregations at once as a Series. The name of the aggregation is placed in the index. By default, it returns the count (number of non-missing values), min, median, mean, max, standard deviation, and 25th and 75 percentiles.

```
[23]: salary.describe()
```

```
[23]: count      23362.000000
mean       58206.761571
std        23322.315285
min        9912.000000
25%       41122.000000
50%       56956.640000
75%       69355.260000
max       342784.000000
Name: salary, dtype: float64
```

Use the `percentiles` parameter to control which percentiles get returned. Pass it a list of all the percentiles (numbers between 0 and 1) you would like returned.

```
[24]: salary.describe(percentiles=[.1, .2, .5, .8, .9, .99])
```

```
[24]: count      23362.000000
mean       58206.761571
std        23322.315285
min        9912.000000
10%       33030.000000
20%       38314.000000
50%       56956.640000
80%       75942.100000
90%       86993.900000
99%      130036.000000
max       342784.000000
Name: salary, dtype: float64
```

15.9 Non-Aggregation methods

Many of these computational methods aggregate and return a single value, but others do not. For instance, the `abs` method takes the absolute value of each individual value in the Series. It returns a Series with the same number of values as the original. In this example, none of the values in the Series are negative, so the values remain the same.

```
[25]: salary.abs().head(3)
```

```
[25]: 0    87545.38
1    82182.00
2    49275.00
Name: salary, dtype: float64
```

The `round` method rounds each value to the nearest given decimal place. Use the `decimals` parameter to determine the place of the rounding. Negative numbers may be used to round places to the left of the decimal. In the following example, we round to the nearest thousand.

```
[26]: salary.round(decimals=-3).head(3)
```

```
[26]: 0    88000.0
1    82000.0
2    49000.0
Name: salary, dtype: float64
```

The `decimals` parameter is the first and only parameter to the `round` method, so it is rare that its name is used when calling it. We can get the same result as the above with `salary.round(-3)`.

Accumulation methods

There are a few accumulation methods that work by keeping track of previous data. For instance, the `cummin` method keeps track of the current minimum value in the Series. It begins at the top with the first value. Since it's the first, it will be the minimum. It then continues down the Series to the second value. If the second value is less than the first, then it will be the new minimum. If not, the first value will remain as the minimum. It returns a Series with the same length as the original of all the current minimums. With our Series, the first salary remains the lowest until the fifth value which remains the lowest until the 10th value.

```
[27]: salary.cummin().head(10)
```

```
[27]: 0    87545.38
      1    82182.00
      2    49275.00
      3    49275.00
      4    49275.00
      5    44616.00
      6    39998.00
      7    39998.00
      8    39998.00
      9    39998.00
Name: salary, dtype: float64
```

We can accumulate the sum beginning from the first value with the `cumsum` method.

```
[28]: salary.cumsum().head()
```

```
[28]: 0    87545.38
      1    169727.38
      2    219002.38
      3    294944.48
      4    364299.74
Name: salary, dtype: float64
```

Non-aggregation methods return an entirely new Series

The non-aggregation methods return an entirely new Series and do not modify the calling Series. This is a crucial concept to understand. pandas has only a few operations and methods that modify objects in-place. Nearly all of the time, a new object is returned. Here, we begin by assigning the result of the `round` method to a variable name.

```
[29]: salary_round = salary.round(decimals=-3)
salary_round.head(3)
```

```
[29]: 0    88000.0
      1    82000.0
      2    49000.0
Name: salary, dtype: float64
```

Let's verify that the calling object has not changed. The `salary` Series is the calling object, i.e., the one that is calling the method and remains unchanged.

```
[30]: salary.head(3)
```

```
[30]: 0    87545.38
      1    82182.00
      2    49275.00
Name: salary, dtype: float64
```

15.10 Series with a non-default index

Let's use a different Series that does not use the default `RangeIndex` to run some of the same methods as above. We'll read in the movie dataset with the title as the index and select the `imdb_score` column as a Series.

```
[31]: movie = pd.read_csv('../data/movie.csv', index_col='title')
score = movie['imdb_score']
score.head()
```

```
[31]: title
Avatar                      7.9
Pirates of the Caribbean: At World's End    7.1
Spectre                      6.8
The Dark Knight Rises        8.5
Star Wars: Episode VII - The Force Awakens   7.1
Name: imdb_score, dtype: float64
```

All of the methods in this chapter work the exact same way as they do with the default index. They all operate on the **values** of the Series and NOT on the index. The index is merely a label for the values. The methods do calculations on the values. Let's show this by taking the mean of the scores. Notice how a single value is returned. The index has nothing to do with these calculations.

```
[32]: score.mean()
```

```
[32]: 6.437428803905615
```

Let's show this is the case with another method and calculate the statistical variance with `var`.

```
[33]: score.var()
```

```
[33]: 1.2719375585109596
```

Calling the non-aggregation methods is where some confusion might arise. Below, we round each score to the nearest whole number. Since we are not aggregating, a Series is returned, and the original index remains with it. Again, no calculation is done on the index. The calculation is only applied to the values.

```
[34]: score.round().head()
```

```
[34]: title
Avatar                      8.0
Pirates of the Caribbean: At World's End    7.0
Spectre                      7.0
The Dark Knight Rises        8.0
Star Wars: Episode VII - The Force Awakens   7.0
Name: imdb_score, dtype: float64
```

Here, we find the current maximum value with the `cummax` method. The index helps out here by informing us which movie is attached to the score. `Avatar` retains the highest score until it is surpassed by '`The Dark Knight Rises`'.

```
[35]: score.cummax().head()
```

```
[35]: title
Avatar                      7.9
Pirates of the Caribbean: At World's End    7.9
Spectre                      7.9
The Dark Knight Rises        8.5
Star Wars: Episode VII - The Force Awakens   8.5
Name: imdb_score, dtype: float64
```

15.11 Operations on a boolean Series

All of the above methods were called on a Series with numeric values. In this section, we will complete several of the same aggregation and non-aggregation methods on a Series of booleans. Let's create a boolean Series by determining which movies had a score greater than eight.

```
[36]: score_8 = score > 8
score_8.head()
```

```
[36]: title
Avatar                      False
Pirates of the Caribbean: At World's End    False
Spectre                      False
The Dark Knight Rises        True
Star Wars: Episode VII - The Force Awakens   False
Name: imdb_score, dtype: bool
```

We can use this Series to filter the data just like we did in the chapters on boolean selection.

```
[37]: only_8 = score[score_8]
only_8.head()
```

```
[37]: title
The Dark Knight Rises      8.5
The Avengers                8.1
Captain America: Civil War  8.2
Toy Story 3                 8.3
WALL·E                     8.4
Name: imdb_score, dtype: float64
```

We can determine the number of movies that have a score greater than eight by finding the length of this result.

```
[38]: len(only_8)
```

```
[38]: 249
```

Sum a boolean Series

We can find the number of movies with a score greater than 8 without doing boolean selection. Instead, we can call the `sum` method.

```
[39]: score_8.sum()
```

[39]: 249

Boolean values are treated as numeric

When performing arithmetic calculations, pandas treats boolean values as numeric. `False` evaluates as 0 and `True` evaluates as 1. With the `score_8` boolean Series, there are 249 `True` values with the rest being `False`. Calling the `sum` method on any boolean Series returns the number of `True` values in that Series.

It is possible to compute this sum without first assigning the boolean Series to a new variable name. We can surround the condition in parentheses and then call the `sum` method.

[40]: `(score > 8).sum()`

[40]: 249

Explanation of this one line of code

Let's examine the line `(score > 8).sum()`. Python first evaluates the expression in parentheses - `score > 8`. This results in a Series, which has all the available methods as any other Series. We then call the `sum` method on this Series to get the desired result.

15.12 Exercises

Continue to use the `score` Series for the first several exercises.

Exercise 1

What is the data type of `score` and how many values does it contain?

[]:

Exercise 2

What is the maximum and minimum score?

[]:

Exercise 3

How many movies have scores greater than 6?

[]:

Exercise 4

How many movies have scores greater than 4 and less than 7?

[]:

Exercise 5

Find the difference between the median and mean of the scores.

[]:

Exercise 6

Add 1 to every value of `score` and then calculate the median.

[]:

Exercise 7

Calculate the median of `score` and add 1 to this. Why is this value the same as Exercise 7?

[]:

Exercise 8

Return a Series that has only scores above the 99.9th percentile.

[]:

Exercise 9

Assign the gross column of the movie dataset to its own variable name as a Series. Round it to the nearest million.

[]:

Exercise 10

Calculate the cumulative sum of the gross Series and then select the 99th integer location.

[]:

Exercise 11

Select the first 100 values of the gross Series and then calculate the sum. Does the result match exercise 10?

[]:

Chapter 16

Series Missing Value Methods

In the previous chapter, we covered the most common attributes and statistical methods for pandas Series objects. In this chapter, we cover several methods that handle missing values within a Series. The official documentation has a [separate section in its Series API](#) just for these methods that handle missing values. Let's begin by reading in the movie dataset and selecting the `duration` series, which contains the length of each movie in minutes.

```
[1]: import pandas as pd  
movie = pd.read_csv('../data/movie.csv', index_col='title')  
duration = movie['duration']  
duration.head()
```

```
[1]: title  
Avatar           178.0  
Pirates of the Caribbean: At World's End    169.0  
Spectre          148.0  
The Dark Knight Rises      164.0  
Star Wars: Episode VII - The Force Awakens    NaN  
Name: duration, dtype: float64
```

16.1 Methods for handling missing values

pandas provides the following methods to handle missing values:

- `isna` - Returns a Series of booleans based on whether each value is missing or not
- `notna` - Exact opposite of `isna`
- `dropna` - Drops the missing values from the Series
- `fillna` - Fills missing values in a variety of ways
- `interpolate` - Fills missing values with statistical interpolation

As a reminder, boolean and integer columns do not allow missing values, so these methods will not be useful for Series with those data types.

16.2 The `isna` method

The `isna` method tests whether each value in the Series is missing or not. It returns a new boolean Series where `True` corresponds with missing values. Non-missing values return as `False`. Let's call the `isna` method on our `duration` Series. One of the first five values is missing.

[2]: `duration.isna().head()`

```
[2]: title
Avatar                         False
Pirates of the Caribbean: At World's End  False
Spectre                          False
The Dark Knight Rises            False
Star Wars: Episode VII - The Force Awakens  True
Name: duration, dtype: bool
```

Counting the number of missing values

pandas does not have a single method that counts the number of missing values in a Series. We are forced to do a little work to find the result. Calling the `sum` method after the `isna` method will count the number of missing values as `True/False` evaluate a 1/0.

[3]: `duration.isna().sum()`

[3]: 15

Alternatively, you can use the `count` method to count the number of non-missing values and subtract it from the length of the Series.

[4]: `len(duration) - duration.count()`

[4]: 15

Finding the percentage of missing values

To find the percentage of missing values in a Series, chain the `mean` method to the `isna` method.

[5]: `duration.isna().mean()`

[5]: 0.0030512611879576893

Alternate calculation

The last calculation might be confusing. We could have been more explicit and calculated the percentage of missing values by dividing the number missing by the total size of the Series as done below.

```
[6]: total = len(duration)
num_missing = total - duration.count()
num_missing / total
```

[6]: 0.0030512611879576893

Why does taking the mean of the boolean Series work?

The mean is defined as the sum of all values divided by the total number of values. In the case of a boolean Series, its sum is just the number of `True` values and in this specific example is equal to the number of missing values.

The `notna` method

The `notna` method works analogously to the `isna` method but returns `True` for non-missing values. Let's verify that it returns the exact opposite result as `isna`.

```
[7]: duration.notna().head()
```

```
[7]: title
Avatar                         True
Pirates of the Caribbean: At World's End  True
Spectre                          True
The Dark Knight Rises            True
Star Wars: Episode VII - The Force Awakens False
Name: duration, dtype: bool
```

Let's count the number of non-missing values by summing up the previous result.

```
[8]: duration.notna().sum()
```

```
[8]: 4901
```

This is the exact calculation that the `count` method performs, so the above is unnecessary.

```
[9]: duration.count()
```

```
[9]: 4901
```

16.3 Dropping missing values with `dropna`

The `dropna` method returns a new Series without any missing values. In the `duration` Series, the fifth value was missing. After calling `dropna`, it's no longer there.

```
[10]: duration2 = duration.dropna()
duration2.head()
```

```
[10]: title
Avatar                         178.0
Pirates of the Caribbean: At World's End  169.0
Spectre                          148.0
The Dark Knight Rises            164.0
John Carter                      132.0
Name: duration, dtype: float64
```

Above, we calculated that there were 15 missing values in the Series. Let's verify that the length of the new Series has decreased by this amount.

```
[11]: len(duration2)
```

```
[11]: 4901
```

```
[12]: len(duration)
```

```
[12]: 4916
```

16.4 Filling missing values with the `fillna` method

There are a number of ways that have been developed to fill missing values. Some of these are quite complex and involve machine learning. pandas only provides a couple simple choices with the `fillna` method. Let's read a dataset with data on a person's weight over time. There are several missing values here.

```
[13]: weight_loss = pd.read_csv('../data/weight_loss.csv')
weight_loss
```

	date	weight
0	2019-10-01	NaN
1	2019-10-02	205.0
2	2019-10-03	NaN
3	2019-10-04	NaN
4	2019-10-05	201.0
5	2019-10-06	NaN
6	2019-10-07	NaN
7	2019-10-08	NaN
8	2019-10-09	199.0
9	2019-10-10	NaN

This part of the book works with Series, so let's select just the weight column.

```
[14]: weight = weight_loss['weight']
weight
```

```
[14]: 0      NaN
      1    205.0
      2      NaN
      3      NaN
      4    201.0
      5      NaN
      6      NaN
      7      NaN
      8    199.0
      9      NaN
Name: weight, dtype: float64
```

There are a few options with the `fillna` method. Passing it a single number replaces each missing value with that number. Let's fill in all missing values with 200.

```
[15]: weight.fillna(200)
```

```
[15]: 0    200.0
      1    205.0
      2    200.0
      3    200.0
```

```
4    201.0
5    200.0
6    200.0
7    200.0
8    199.0
9    200.0
Name: weight, dtype: float64
```

A common strategy involves filling missing values with either the mean or the median. Let's calculate the median, assign it to a variable name, and then call the `fillna` method. Here, the median is 201.

```
[16]: median = weight.median()
weight.fillna(median)
```

```
[16]: 0    201.0
1    205.0
2    201.0
3    201.0
4    201.0
5    201.0
6    201.0
7    201.0
8    199.0
9    201.0
Name: weight, dtype: float64
```

Filling with the previous or next non-missing value

The `fillna` method provides an option to use either the previous or next non-missing value. To fill the missing value with the most recent previous non-missing value, set the `method` parameter to ‘`ffill`’, which stands for ‘forward fill’. Let's forward fill our `weight` Series. Notice, that the first value remains missing as there is no non-missing value that comes before it.

```
[17]: weight.fillna(method='ffill')
```

```
[17]: 0      NaN
1    205.0
2    205.0
3    205.0
4    201.0
5    201.0
6    201.0
7    201.0
8    199.0
9    199.0
Name: weight, dtype: float64
```

Similarly, we can fill the missing values with the next non-missing value by setting the `method` parameter to ‘`bfill`’, which stands for ‘backward fill’. This time the last value remains missing as no non-missing value follows it.

```
[18]: weight.fillna(method='bfill')
```

```
[18]: 0    205.0
      1    205.0
      2    201.0
      3    201.0
      4    201.0
      5    199.0
      6    199.0
      7    199.0
      8    199.0
      9    NaN
Name: weight, dtype: float64
```

Limit the number of missing values filled

By default, all missing values will be filled when the `method` parameter is set. You can limit the number of consecutive missing values filled with the `limit` parameter by setting it equal to an integer. Here, we only allow the very next missing value to be filled.

```
[19]: weight.fillna(method='ffill', limit=1)
```

```
[19]: 0    NaN
      1    205.0
      2    205.0
      3    NaN
      4    201.0
      5    201.0
      6    NaN
      7    NaN
      8    199.0
      9    199.0
Name: weight, dtype: float64
```

16.5 Filling missing values with `interpolate`

Interpolation is a more complex method for filling in missing values that gives you more control as to how the values are filled. There are many different methods for interpolation that all involve statistical calculations. In pandas, we use the `interpolate` method and choose the type of interpolation with the `method` parameter. By default, pandas does linear interpolation.

```
[20]: weight.interpolate(method='linear')
```

```
[20]: 0    NaN
      1    205.000000
      2    203.666667
      3    202.333333
      4    201.000000
      5    200.500000
      6    200.000000
      7    199.500000
      8    199.000000
      9    199.000000
```

```
Name: weight, dtype: float64
```

Linear interpolation is the simplest method and computes missing values such that they are modeled using a straight line. With the `weight` Series, 205 is the 2nd value and 201 the fifth. There are two missing values between them. We compute the slope of the line between these two points as $\frac{201-205}{5-2} = -\frac{4}{3}$.

The missing values are filled in with the equation $205 - \frac{4}{3}x$ where x is the number of values from the last non-missing value. This process repeats for the next interval of missing values with a new slope calculated. Interestingly, pandas fills any missing values that appear after the last non-missing value with that last non-missing value. You can prevent this by setting the `limit_area` parameter to ‘inside’.

```
[21]: weight.interpolate(method='linear', limit_area='inside')
```

```
[21]: 0      NaN
1    205.000000
2    203.666667
3    202.333333
4    201.000000
5    200.500000
6    200.000000
7    199.500000
8    199.000000
9      NaN
Name: weight, dtype: float64
```

Many more interpolations methods

There are many more interpolation methods such as quadratic, cubic, spline, and others. All of these methods are processed by the `scipy` library’s `interpolate` module. You’ll need to [navigate to the scipy documentation](#) to read the exact details of how the methods work. Here, we interpolate using the quadratic method.

```
[22]: weight.interpolate('quadratic')
```

```
[22]: 0      NaN
1    205.000000
2    203.428571
3    202.095238
4    201.000000
5    200.142857
6    199.523810
7    199.142857
8    199.000000
9      NaN
Name: weight, dtype: float64
```

16.6 Graphing interpolation methods

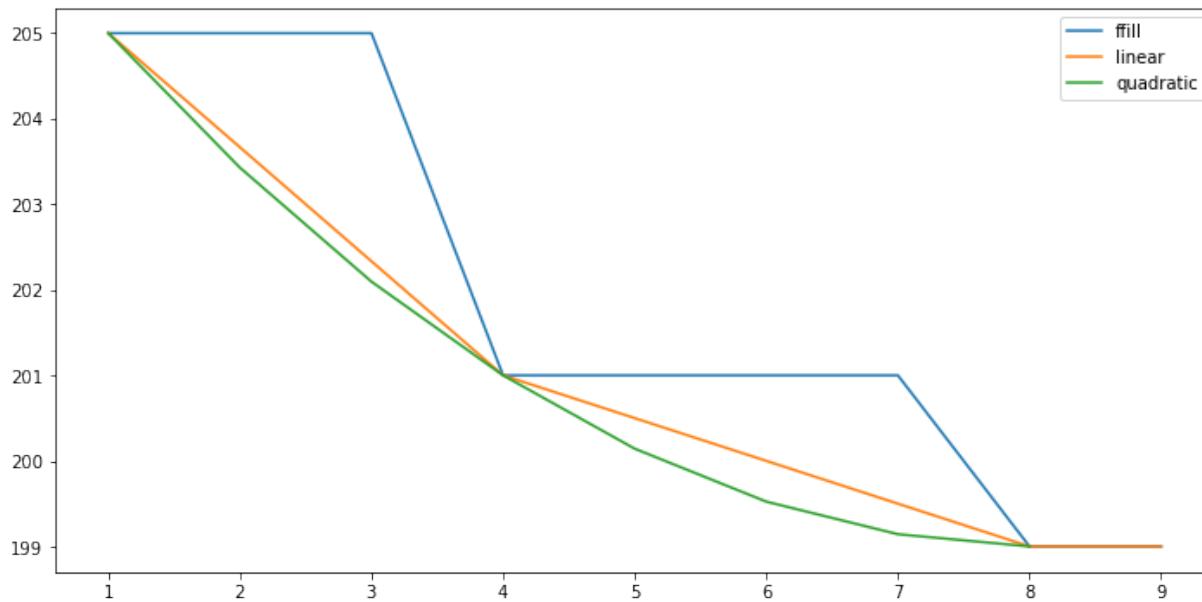
Comparing different interpolation methods can be aided with a graph. Here, we compare three different methods:

- `fillna` with forward filling
- linear interpolation

- quadratic interpolation

The plotting commands will be explained in the **Visualization** part of the course.

```
[23]: %matplotlib inline
weight.fillna(method='ffill').plot(kind='line', label='ffill', legend=True, ▾
    ↵ figsize=(12, 6))
weight.interpolate('linear').plot(kind='line', label='linear', legend=True)
weight.interpolate('quadratic').plot(kind='line', label='quadratic', legend=True);
```



16.7 Interpolation methods use the index

All of the interpolation methods (except linear) use the index values during their calculation. Below, we create a copy of our data and change the index to be numbers other than the sequence of integers beginning at 0. Notice, how the interpolated values are different from above.

```
[24]: w2 = weight.copy()
w2.index = [0, 1, 5, 6, 8, 12, 22, 27, 30, 35]
w2.interpolate('quadratic')
```

```
[24]: 0      NaN
1      205.000000
5      202.515450
6      201.977161
8      201.000000
12     199.443350
22     197.871473
27     198.328258
30     199.000000
35     NaN
Name: weight, dtype: float64
```

16.8 Exercises

Execute the following cell to use the actor 1 and actor 2 Facebook like Series for the first few exercises.

```
[25]: actor1_fb = movie['actor1_fb']
actor2_fb = movie['actor2_fb']
actor1_fb.head()
```

```
[25]: title
Avatar           1000.0
Pirates of the Caribbean: At World's End 40000.0
Spectre          11000.0
The Dark Knight Rises   27000.0
Star Wars: Episode VII - The Force Awakens 131.0
Name: actor1_fb, dtype: float64
```

```
[26]: actor2_fb.head()
```

```
[26]: title
Avatar           936.0
Pirates of the Caribbean: At World's End 5000.0
Spectre          393.0
The Dark Knight Rises   23000.0
Star Wars: Episode VII - The Force Awakens 12.0
Name: actor2_fb, dtype: float64
```

Exercise 1

What percentage of actor 1 Facebook likes are missing?

```
[ ]:
```

Exercise 2

Use the `notna` method to find the number of non-missing values in the actor 1 Facebook like column. Verify this number is the same as the `count` method.

```
[ ]:
```

Exercise 3

Fill the missing values of `actor1_fb` with the maximum of `actor2_fb`. Save this result to variable `actor1_fb_full`.

```
[ ]:
```

Exercise 4

Verify the results of Exercise 3 by selecting just the values of `actor1_fb_full` that were filled by `actor2_fb`.

```
[ ]:
```

Exercise 5

Use the `duration` Series and test whether each movie is greater than 100. Assign the resulting Series to `filt`. Then test whether the `duration` Series is less than or equal to 100 and assign it to `filt2`. Call the `sum` method on both of these new Series and add their results together. Why doesn't this result equal the total length of the Series? Shouldn't a value be either greater than 100 or less than or equal to 100?

[]:

Exercise 6

How many missing values are there in the `year` column?

[]:

Exercise 7

Select the language column as a Series and assign it to a variable with the same name. Create a variable `filt` that determines whether a language is missing. Create a new Series that fills in all missing languages with 'English' and assign it to the variable `language2`. Output both `language` and `language2` for the movies that originally had missing values.

[]:

Exercise 8

Repeat exercise 7 without first assigning the language column to a variable. Reference it by using *just the brackets*. Still make a variable `language2` to verify the results.

[]:

Exercise 9

Select the `gross` column, and drop all missing values from it. Confirm that the new length of the resulting Series is correct.

[]:

Exercise 10

Read in `girl_height.csv` as a DataFrame and output all of the data. The average height for each age is found in the `height` column. Assign the `height_na` Series to a variable. Notice that all ages from 2 through 12 are missing, but all other ages have the same value as the `height` column. Use the `interpolate` method to fill in the missing values with method 'linear', 'quadratic', and 'cubic'. Save each interpolated Series to a variable with the same name as its method.

[]:

Exercise 11

Uncomment and run the commands below to plot each interpolated Series. Which one provides the best estimate for height?

```
[27]: # %matplotlib inline
# girl_height['height'].plot(figsize=(12, 6), label='actual', legend=True)
# linear.plot(label='linear', legend=True)
# quadratic.plot(label='quadratic', legend=True)
# cubic.plot(label='cubic', legend=True)
```


Chapter 17

Series Sorting, Ranking, and Uniqueness

In this chapter, we cover a few more important methods on sorting and ranking the values in our Series, along with finding unique values and removing duplicates. We read in the movie dataset, set the title as the index, and select the `imdb_score` column as a Series.

```
[1]: import pandas as pd  
movie = pd.read_csv('../data/movie.csv', index_col='title')  
score = movie['imdb_score']  
score.head()
```

```
[1]: title  
Avatar 7.9  
Pirates of the Caribbean: At World's End 7.1  
Spectre 6.8  
The Dark Knight Rises 8.5  
Star Wars: Episode VII - The Force Awakens 7.1  
Name: imdb_score, dtype: float64
```

17.1 Sorting

The `sort_values` method sorts the Series from least to greatest by default. It places missing values at the end. You may call it without any arguments.

```
[2]: score.sort_values().head(3)
```

```
[2]: title  
Justin Bieber: Never Say Never 1.6  
Foodfight! 1.7  
Disaster Movie 1.9  
Name: imdb_score, dtype: float64
```

To sort from greatest to least, set the `ascending` parameter to `False`.

```
[3]: score.sort_values(ascending=False).head(3)
```

```
[3]: title  
Towering Inferno 9.5  
The Shawshank Redemption 9.3
```

```
The Godfather           9.2
Name: imdb_score, dtype: float64
```

Making missing value appear first

By default, all missing values are placed at the bottom of the resulting Series. You can change this so that they appear first by setting the `na_position` method to ‘first’. This is a good way to quickly view all the missing values in your Series. Here, we sort the `duration` column so that it’s missing values come first.

```
[4]: movie['duration'].sort_values(na_position='first').head()
```

```
[4]: title
Star Wars: Episode VII - The Force Awakens      NaN
Harry Potter and the Deathly Hallows: Part II   NaN
Harry Potter and the Deathly Hallows: Part I    NaN
Black Water Transit                            NaN
War & Peace                                 NaN
Name: duration, dtype: float64
```

Sorting the index

Since Series also have an index, pandas allows you to sort by it as well with the `sort_index` method.

```
[5]: score.sort_index().head(3)
```

```
[5]: title
#Horror          3.3
10 Cloverfield Lane 7.3
10 Days in a Madhouse 7.5
Name: imdb_score, dtype: float64
```

As with `sort_values`, the same `ascending` parameter exists to change the sort from greatest to least.

```
[6]: score.sort_index(ascending=False).head(3)
```

```
[6]: title
Æon Flux          5.5
xXx: State of the Union 4.3
xXx                5.8
Name: imdb_score, dtype: float64
```

Python uses the Unicode code point (an integer) of each character to compare strings. We can use the built-in `ord` function to find each characters code point. For instance, the character ‘#’ evaluates as 35, which is less than the value for ‘1’ and ‘A’. The movie ‘#Horror’ has the smallest starting character code point and appears first when sorted from least to greatest.

```
[7]: ord('#')
```

```
[7]: 35
```

```
[8]: ord('1')
```

```
[8]: 49
```

```
[9]: ord('A')
```

```
[9]: 65
```

When sorting the opposite direction, the movie ‘Æon Flux’ begins with the ‘Æ’ character which has code point 198 and is the largest starting character. All lowercase letters have higher code points than all uppercase letters, so movies that begin with lowercase letters will be at the top as most movie begin with a capital letter.

```
[10]: ord('Æ')
```

```
[10]: 198
```

```
[11]: ord('x')
```

```
[11]: 120
```

```
[12]: ord('a')
```

```
[12]: 97
```

```
[13]: ord('Z')
```

```
[13]: 90
```

17.2 Ranking

The `rank` method provides a numerical ranking for each value in the Series. By default, it ranks the values in ascending order beginning at 1. This method is easier to understand when working on a smaller Series. Let’s assign the first 10 scores to the variable name `score10`.

```
[14]: score10 = score.head(10)  
score10
```

```
[14]: title  
Avatar                7.9  
Pirates of the Caribbean: At World's End    7.1  
Spectre               6.8  
The Dark Knight Rises            8.5  
Star Wars: Episode VII - The Force Awakens   7.1  
John Carter             6.6  
Spider-Man 3              6.2  
Tangled                 7.8  
Avengers: Age of Ultron           7.5  
Harry Potter and the Half-Blood Prince        7.5  
Name: imdb_score, dtype: float64
```

Every value in this Series will now be ranked from least to greatest. The lowest scoring value gets a ranking of 1, while the greatest gets a ranking of 10.

```
[15]: score10.rank()
```

```
[15]: title
      Avatar                         9.0
      Pirates of the Caribbean: At World's End   4.5
      Spectre                         3.0
      The Dark Knight Rises           10.0
      Star Wars: Episode VII - The Force Awakens 4.5
      John Carter                      2.0
      Spider-Man 3                      1.0
      Tangled                           8.0
      Avengers: Age of Ultron          6.5
      Harry Potter and the Half-Blood Prince    6.5
      Name: imdb_score, dtype: float64
```

This method can be confusing the first time it is used. First of all, it does NOT sort the data. Notice that the titles in the index are in the same order as the original.

It provides the rank, just like you would rank runners in a race. If you look at the original data, the movie Spider-Man 3 has the lowest `imdb_score` at 6.2. In the Series resulting from the `rank` method, it gets the value 1. The next lowest score is 6.6 from the movie John Carter, which results in a ranking of 2, followed by Spectre with a ranking of 3.

Ranking ties

After Spectre, Pirates of the Caribbean and Star Wars: Episode VII tied with a score of 7.1. There are several methods available to choose how ties are ranked. By default, pandas uses the ‘average’ method which works by averaging the total rank number for those tied values as if they were not tied.

For example, there are two movies tied for the fourth rank. If they were not tied, they would be ranked 4 and 5. The average of this is 4.5 and each movie is given this rank. The ranking would continue here at 6.

Let's say there were five movies tied for the fourth rank (instead of two), then their non-tied ranks would be 4, 5, 6, 7, and 8 for an average rank of 6. Each of the five movies would be given this rank. The ranking would then continue with 9.

There are actually two sets of ties in the above dataset. *Avengers: Age of Ultron* and *Harry Potter and the Half-Blood Prince* both have an `imdb_score` of 7.5 and are given the average rank of 6.5 as their non-tied ranks would be 6 and 7.

Change tie method

Use the `method` parameter to change how pandas handles ties. Using a ‘dense’ rank, will give each movie tied the same rank and not skip any number when moving to the next. Here, the first set of ties is given rank 4, which is immediately followed by the second set and given rank 5. The highest numerical rank is only 8 using dense ranking and not 10 as before.

```
[16]: score10.rank(method='dense')
```

```
[16]: title
      Avatar                         7.0
      Pirates of the Caribbean: At World's End 4.0
      Spectre                         3.0
      The Dark Knight Rises           8.0
```

```
Star Wars: Episode VII - The Force Awakens      4.0
John Carter                                    2.0
Spider-Man 3                                     1.0
Tangled                                         6.0
Avengers: Age of Ultron                        5.0
Harry Potter and the Half-Blood Prince        5.0
Name: imdb_score, dtype: float64
```

There are three other methods to handle ties: * ‘min’ - give each tie the minimum rank number * ‘max’ - give each tie the maximum rank number * ‘first’ - arbitrarily give the tie that comes first in the dataset the lower/higher number.

Rank for greatest to least

For movies, it makes more sense to rank the movie with the highest score as 1, which is done by setting the `ascending` parameter to `False`. The ‘min’ method of ranking is used to handle ties.

```
[17]: score10.rank(ascending=False, method='min')
```

```
[17]: title
Avatar                                         2.0
Pirates of the Caribbean: At World's End       6.0
Spectre                                         8.0
The Dark Knight Rises                         1.0
Star Wars: Episode VII - The Force Awakens     6.0
John Carter                                    9.0
Spider-Man 3                                    10.0
Tangled                                         3.0
Avengers: Age of Ultron                        4.0
Harry Potter and the Half-Blood Prince        4.0
Name: imdb_score, dtype: float64
```

17.3 Uniqueness

There are a few methods that deal with unique values in a Series:

- `unique` - Returns a numpy array of all the unique values in order of their appearance
- `nunique` - Returns the number of unique values in the Series
- `drop_duplicates` - Returns a pandas Series of just the unique values

The `unique` method

The `unique` method returns each unique value in the Series preserving the order of its appearance. Let’s select the `content_rating` column as a Series and use the `unique` method to get all the unique ratings. Interestingly, it returns a numpy array and NOT a pandas Series.

```
[18]: unique_ratings = movie['content_rating'].unique()
unique_ratings
```

```
[18]: array(['PG-13', nan, 'PG', 'G', 'R', 'TV-14', 'TV-PG', 'TV-MA', 'TV-G',
           'Not Rated', 'Unrated', 'Approved', 'TV-Y', 'NC-17', 'X', 'TV-Y7',
           'GP', 'Passed', 'M'], dtype=object)
```

The `nunique` method

The `nunique` method returns the number of unique values in the Series.

```
[19]: movie['content_rating'].nunique()
```

```
[19]: 18
```

You might expect that the number of unique values to be same as the length of the array returned from the `unique` method. This might not be the case as the `nunique` does not count missing values if they are present. Since there are missing values in this Series, `nunique` returns one less.

```
[20]: len(unique_ratings)
```

```
[20]: 19
```

You can choose to count a unique missing value with `nunique` by setting the `dropna` parameter to `False`. This will add one to the count if any missing values are present.

```
[21]: movie['content_rating'].nunique(dropna=False)
```

```
[21]: 19
```

The `drop_duplicates` method

The `drop_duplicates` method is similar to `unique` but returns a pandas Series. By default, it keeps the first unique value it encounters.

```
[22]: duration_unique_series = movie['content_rating'].drop_duplicates()
duration_unique_series.head()
```

```
[22]: title
      Avatar           PG-13
      Star Wars: Episode VII - The Force Awakens    NaN
      Tangled            PG
      Monsters University        G
      The Lovers          R
      Name: content_rating, dtype: object
```

It will contain the same number of values as the Series returned from the `unique` method.

```
[23]: len(duration_unique_series)
```

```
[23]: 19
```

Why does it matter that `drop_duplicates` keeps the first value?

A Series is composed of both an index and the values. Both `unique` and `drop_duplicates` only consider the values of a Series. But, the index will likely be different for values that are the same, so order does matter with `drop_duplicates`. Set the `keep` parameter to `last` to keep the very last occurrence or to `False` to drop all values that are duplicates. Notice how the index for the movie rated 'G' is different.

```
[24]: movie['content_rating'].drop_duplicates(keep='last').head(7)
```

[24]: title

```
Arthur           TV-Y
The Powerpuff Girls    TV-Y7
Hang 'Em High        M
Home Movies          TV-PG
The Broadway Melody   Passed
Happy Valley          TV-MA
Sunday School Musical   G
Name: content_rating, dtype: object
```

Preference for drop_duplicates

Both the `unique` and `drop_duplicates` methods accomplish very similar tasks. The `unique` method returns its results as a numpy array, which isn't ideal. In my opinion, it's better to keep all your results as pandas objects (Series or DataFrames). Working with fewer types of objects makes your data analysis easier. Additionally, the `drop_duplicates` method is more flexible with the availability of the `keep` parameter.

The `unique` method is only available for Series, unlike `drop_duplicates`, which is available for both Series and DataFrames. Because the `unique` method's functionality is a subset of `drop_duplicates`, I recommend only using `drop_duplicates`.

17.4 Exercises

Exercise 1

Select the column holding the number of reviews as a Series and sort it from greatest to least.

[]:

Exercise 2

Find the number of unique actors in each of the actor columns. Do not count missing values. Use three separate calls to `nunique`.

[]:

Exercise 3

Select the `year` column, sort it, and drop any duplicates.

[]:

Exercise 4

Get the same result as Exercise 3 by dropping duplicates first and then sort. Which method is faster?

[]:

Exercise 5

Rank each movie by duration from greatest to least and then sort this ranking from least to greatest. Output the top 10 values. Do you get the same result by sorting the duration from greatest to least?

[]:

Exercise 6

Select actor1 as a Series and sort it from least to greatest, but have missing values appear first. Output the first 10 values.

[]:

Chapter 18

Series Methods More

In this chapter, we cover several more less common, but still useful and important Series methods that you need to know in order to be fully capable at analyzing data with pandas.

- `agg` - Compute multiple aggregations at once
- `idxmax` and `idxmin` - Return the index of the max/min
- `diff` and `pct_change` - Find the difference/percent change from one value to the next
- `sample` - Randomly sample values in a Series
- `nsmallest/nlargest` - Return the top/bottom n values in a Series
- `replace` - Replace one or more values in a variety of ways

Let's begin by reading in the movie dataset and selecting the `imdb_score` Series.

```
[1]: import pandas as pd  
movie = pd.read_csv('../data/movie.csv', index_col='title')  
score = movie['imdb_score']  
score.head()
```

```
[1]: title  
Avatar                7.9  
Pirates of the Caribbean: At World's End    7.1  
Spectre               6.8  
The Dark Knight Rises            8.5  
Star Wars: Episode VII - The Force Awakens   7.1  
Name: imdb_score, dtype: float64
```

18.1 The `agg` method

The `agg` method allows you to compute several aggregations simultaneously. Provide it a list with the Series aggregation methods as strings. For instance, the following computes the minimum and maximum of a Series and returns the result as a Series as well.

```
[2]: score.agg(['min', 'max'])
```

```
[2]: min    1.6  
max    9.5  
Name: imdb_score, dtype: float64
```

You may provide any number of aggregation methods to the `agg` method. The `agg` method is similar to `describe`, but calculates just the aggregations you desire.

```
[3]: score.agg(['min', 'max', 'count', 'nunique'])
```

```
[3]: min           1.6
max            9.5
count        4916.0
nunique       78.0
Name: imdb_score, dtype: float64
```

18.2 Index of maximum and minimum

The `max` and `min` methods may be used to return the minimum and maximum values of a Series. Occasionally, we will want to know the index label for these maximum or minimum values and can do so with the `idxmax` and `idxmin` methods. Let's first find the maximum and minimum values.

```
[4]: score.max()
```

```
[4]: 9.5
```

```
[5]: score.min()
```

```
[5]: 1.6
```

To see the movies associated with these max and min values, call the `idxmin` and `idxmax` methods.

```
[6]: score.idxmax()
```

```
[6]: 'Towering Inferno'
```

```
[7]: score.idxmin()
```

```
[7]: 'Justin Bieber: Never Say Never'
```

Let's verify these results by dropping any missing values and sorting the Series.

```
[8]: score_sorted = score.dropna().sort_values(ascending=False)
```

We can now output the first and last few values to verify.

```
[9]: score_sorted.head(3)
```

```
[9]: title
Towering Inferno          9.5
The Shawshank Redemption   9.3
The Godfather              9.2
Name: imdb_score, dtype: float64
```

```
[10]: score_sorted.tail(3)
```

[10]: title

```
Disaster Movie           1.9
Foodfight!              1.7
Justin Bieber: Never Say Never  1.6
Name: imdb_score, dtype: float64
```

Both `idxmax` and `idxmin` always return a single index label. If two or more values share the maximum/minimum then pandas returns the index label that appears first in the Series. Since, one value is returned, `idxmax` and `idxmin` are considered aggregation methods.

18.3 Differencing methods `diff` and `pct_change`

The `diff` method takes the difference between the current value and some other value. By default, the other value is the immediate preceding one. The first value in the Series has no previous value, so its difference will be missing in the result. Let's read a small sample of Microsoft's stock dataset found in the stocks folder containing 10 trading days worth of information.

[11]: `msft = pd.read_csv('..../data/stocks/msft_sample.csv')`
`msft`

	date	open	high	low	close	adjusted_close	volume	dividend_amount
0	2019-10-08	137.08	137.76	135.6200	135.67	135.67	25550500	0.0
1	2019-10-09	137.46	138.70	136.9700	138.24	138.24	19749900	0.0
2	2019-10-10	138.49	139.67	138.2500	139.10	139.10	17654600	0.0
3	2019-10-11	140.12	141.03	139.5000	139.68	139.68	25446000	0.0
4	2019-10-14	139.69	140.29	139.5200	139.55	139.55	13304300	0.0
5	2019-10-15	140.06	141.79	139.8100	141.57	141.57	19695700	0.0
6	2019-10-16	140.79	140.99	139.5300	140.41	140.41	20751600	0.0
7	2019-10-17	140.95	141.42	139.0200	139.69	139.69	21460600	0.0
8	2019-10-18	139.76	140.00	136.5638	137.41	137.41	27654449	0.0
9	2019-10-21	138.45	138.50	137.0100	138.39	138.39	20668059	0.0

Let's select the `adjusted_close` column as a Series and call the `diff` method on it. The difference between the second and first values is 2.57 and is now the new second value in the returned Series.

[12]: `ac = msft['adjusted_close']`
`ac.diff()`

[12]: 0 NaN
1 2.57
2 0.86
3 0.58
4 -0.13
5 2.02
6 -1.16
7 -0.72

```
8    -2.28
9     0.98
Name: adjusted_close, dtype: float64
```

It's possible to control which two values are subtracted. By default, the `periods` parameter is set to 1. Here, we change it to 3. The first possible difference happens between the fourth (139.68) and first (135.67) values, resulting in 4.01. The first three values have no value three positions ahead of them, so they are now missing.

```
[13]: ac.diff(periods=3)
```

```
[13]: 0      NaN
1      NaN
2      NaN
3     4.01
4     1.31
5     2.47
6     0.73
7     0.14
8    -4.16
9    -2.02
Name: adjusted_close, dtype: float64
```

We can take the difference between the current value and a value further ahead by using negative integers. Here, we take the current value and subtract the second value following it. The last two values are missing as they do not have two values ahead.

```
[14]: ac.diff(-2)
```

```
[14]: 0    -3.43
1    -1.44
2    -0.45
3    -1.89
4    -0.86
5     1.88
6     3.00
7     1.30
8      NaN
9      NaN
Name: adjusted_close, dtype: float64
```

The `pct_change` method works analogously but returns the percentage difference instead.

```
[15]: ac.pct_change()
```

```
[15]: 0        NaN
1    0.018943
2    0.006221
3    0.004170
4   -0.000931
5    0.014475
6   -0.008194
```

```
7   -0.005128
8   -0.016322
9    0.007132
Name: adjusted_close, dtype: float64
```

```
[16]: ac.pct_change(-2)
```

```
[16]: 0   -0.024659
1   -0.010309
2   -0.003225
3   -0.013350
4   -0.006125
5    0.013458
6    0.021832
7    0.009394
8      NaN
9      NaN
Name: adjusted_close, dtype: float64
```

18.4 The nlargest and nsmallest methods

The `nlargest` and `nsmallest` methods are convenience methods to quickly return the top `n` values in the Series in order. By default, they return the top 5 values. Use the parameter `n` to choose how many values to return. Here, we select the top 4 movies by score.

```
[17]: score.nlargest(n=4)
```

```
[17]: title
Towering Inferno      9.5
The Shawshank Redemption 9.3
The Godfather        9.2
Dekalog              9.1
Name: imdb_score, dtype: float64
```

By default, `nlargest` and `nsmallest` return exactly `n` values even if there are ties. Let's produce a similar result by calling `sort_values` and returning the first five values. You'll notice that two movies are tied for the fourth highest score. By default, `nlargest` returns the first one.

```
[18]: score.sort_values(ascending=False).head()
```

```
[18]: title
Towering Inferno      9.5
The Shawshank Redemption 9.3
The Godfather        9.2
Dekalog              9.1
Kickboxer: Vengeance 9.1
Name: imdb_score, dtype: float64
```

If you'd like to keep the top `n` values and ties, set the `keep` parameter to the string 'all'. There is only one other movie with a value of 9.1, but if there were more, all of them would be returned here.

```
[19]: score.nlargest(n=4, keep='all')
```

```
[19]: title
Towering Inferno      9.5
The Shawshank Redemption 9.3
The Godfather        9.2
Dekalog              9.1
Kickboxer: Vengeance 9.1
Name: imdb_score, dtype: float64
```

The `nsmallest` method works analogously and returns the smallest `n` values.

```
[20]: score.nsmallest(n=3)
```

```
[20]: title
Justin Bieber: Never Say Never    1.6
Foodfight!                      1.7
Disaster Movie                   1.9
Name: imdb_score, dtype: float64
```

By default, the first tie is kept, but setting `keep` to 'last' return the last occurrence of the nth ranked value.

```
[21]: score.nsmallest(n=3, keep='last')
```

```
[21]: title
Justin Bieber: Never Say Never    1.6
Foodfight!                      1.7
The Helix... Loaded             1.9
Name: imdb_score, dtype: float64
```

18.5 Randomly sample a Series

The `sample` method is great for randomly sampling the values in your Series. Set the `n` parameter of the `sample` method to an integer to return that many randomly selected values.

```
[22]: score.sample(n=5)
```

```
[22]: title
The Majestic          6.9
The Man in the Iron Mask 6.4
Stuart Little         5.9
Juwanna Mann          4.5
The To Do List         5.8
Name: imdb_score, dtype: float64
```

By default, the sampling is done without replacement, so there is no possibility of selecting the same piece of data. If you attempt to choose a sample larger than the number of values in the Series, you'll get an error.

```
[23]: score.sample(n=5000)
```

`ValueError`: Cannot take a larger sample than population when 'replace=False'

However, you can sample with replacement, meaning that you can get duplicate pieces of data by setting the `replace` parameter to `False`.

```
[24]: score.sample(n=5000, replace=True).head()
```

```
[24]: title
Snakes on a Plane      5.6
Fifty Shades of Black  3.5
Red Lights            6.2
Good Night, and Good Luck. 7.5
About Last Night      6.1
Name: imdb_score, dtype: float64
```

You can also sample a fraction of the dataset by using the `frac` parameter. Here we take a random sample of 15% of the data.

```
[25]: score_sample = score.sample(frac=.15)
score_sample.head()
```

```
[25]: title
Anchorman 2: The Legend Continues 6.3
Happy Valley                      8.5
Grease                            7.2
Seven Psychopaths                 7.2
Halloween 5                       5.2
Name: imdb_score, dtype: float64
```

Let's verify that the sample is indeed 15% of the total length of the original.

```
[26]: len(score_sample)
```

```
[26]: 737
```

```
[27]: len(score) * .15
```

```
[27]: 737.4
```

18.6 The `replace` method

The `replace` method helps you replace particular values in your Series with other values. There are a lot of options with the `replace` method to handle many different kinds of replacement. This section will only discuss basic replacements. Let's select the color column from the movie dataset as a Series.

```
[28]: color = movie['color']
color.head()
```

```
[28]: title
Avatar                  Color
Pirates of the Caribbean: At World's End  Color
Spectre                 Color
The Dark Knight Rises   Color
```

```
Star Wars: Episode VII - The Force Awakens      NaN
Name: color, dtype: object
```

The simplest way to replace a value in the Series is to pass the `replace` method two arguments. The first is the value you'd like to replace and the second is the replacement value. Here, we replace the exact string 'Color' with 'Colour'.

```
[29]: color.replace('Color', 'Colour').head()
```

```
[29]: title
Avatar                      Colour
Pirates of the Caribbean: At World's End    Colour
Spectre                       Colour
The Dark Knight Rises           Colour
Star Wars: Episode VII - The Force Awakens   NaN
Name: color, dtype: object
```

The `replace` method works with columns of all data types. Here we use the `score` Series to replace the value 7.1 with 999.

```
[30]: score.head()
```

```
[30]: title
Avatar                      7.9
Pirates of the Caribbean: At World's End    7.1
Spectre                       6.8
The Dark Knight Rises           8.5
Star Wars: Episode VII - The Force Awakens   7.1
Name: imdb_score, dtype: float64
```

```
[31]: score.replace(7.1, 999).head()
```

```
[31]: title
Avatar                      7.9
Pirates of the Caribbean: At World's End    999.0
Spectre                       6.8
The Dark Knight Rises           8.5
Star Wars: Episode VII - The Force Awakens   999.0
Name: imdb_score, dtype: float64
```

You might think you can replace specific words within strings, and you would be correct, but the way you do so takes more effort. Let's take a look at the `genres` column as a Series.

```
[32]: genres = movie['genres']
genres.head()
```

```
[32]: title
Avatar                      Action|Adventure|Fantasy|Sci-Fi
Pirates of the Caribbean: At World's End    Action|Adventure|Fantasy
Spectre                       Action|Adventure|Thriller
The Dark Knight Rises           Action|Thriller
Star Wars: Episode VII - The Force Awakens   Documentary
Name: genres, dtype: object
```

Let's say we are interested in replacing the string 'Adventure' with 'Adv' to shorten the length of each string in this column. The following won't work.

```
[33]: genres.replace('Adventure', 'Adv').head()
```

```
[33]: title
Avatar                               Action|Adventure|Fantasy|Sci-Fi
Pirates of the Caribbean: At World's End      Action|Adventure|Fantasy
Spectre                                Action|Adventure|Thriller
The Dark Knight Rises                  Action|Thriller
Star Wars: Episode VII - The Force Awakens    Documentary
Name: genres, dtype: object
```

By default, the `replace` method works by matching the entire value in the Series. The genre must be exactly 'Adventure' for it to be replaced without any other text surrounding it. It is possible to do this within-string replacement, but you'll need a lesson on regular expressions first. Setting the `regex` parameter to `True` will do the trick. The following is presented with some precaution. You should not use the `regex` parameter until you understand the fundamentals of regular expressions, which are thoroughly covered in its own part of the course.

```
[34]: genres.replace('Adventure', 'Adv', regex=True).head()
```

```
[34]: title
Avatar                               Action|Adv|Fantasy|Sci-Fi
Pirates of the Caribbean: At World's End      Action|Adv|Fantasy
Spectre                                Action|Adv|Thriller
The Dark Knight Rises                  Action|Thriller
Star Wars: Episode VII - The Force Awakens    Documentary
Name: genres, dtype: object
```

18.7 Exercises

Read in the employee dataset by executing the cell below and use it for the following exercises.

```
[35]: emp = pd.read_csv('../data/employee.csv')
emp.head()
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black
3	Police	SENIOR POLICE OFFICER	1997-05-27	75942.10	Male	Hispanic
4	Police	SENIOR POLICE OFFICER	2006-01-23	69355.26	Male	White

Exercise 1

Find the minimum, maximum, mean, median, and standard deviation of the salary column. Return the result as a Series.

[]:

Exercise 2

Use the `idxmax` and `idxmin` methods to find the index where the maximum and minimum salaries are located in the DataFrame. Then use the `loc` indexer to select both of those rows as a DataFrame.

[]:

Exercise 3

Repeat exercise 3, but do so on the `imdb_score` column from the movie dataset.

[]:

Exercise 4

The `idxmax` and `idxmin` methods are aggregations as they return a single value. Use the `agg` method to return the min/max `imdb_score` and the label for each score.

[]:

Exercise 5

Read in 20 years of Microsoft stock data, setting the ‘timestamp’ column as the index. Find the top 5 largest one-day percentage gains in the `adjusted_close` column.

[]:

Exercise 6

Randomly sample the `actor1` column as a Series with replacement to select three values. Use random state 12345. Setting a random state ensures that the same random sample is chosen regardless of which machine or version of numpy is being used.

[]:

Exercise 7

Select the title column from the employee dataset as a Series. Replace all occurrences of ‘POLICE OFFICER’ and ‘SENIOR POLICE OFFICER’ with ‘POLICE’. You can use a list as the first argument passed to the `replace` method.

[]:

Chapter 19

Series String Methods

The previous chapters in this part focused mainly on Series that contained numeric values. In this chapter, we focus on methods that work for Series containing string data. Columns of strings are processed quite differently than columns of numeric values. Remember, that strings are read in as the **object** data type which technically may contain any Python object. The majority of the time, object columns are entirely composed of strings.

With the release of pandas 1.0, a dedicated string data type was made available. All of the commands in this chapter will use the old object data type but would also work for this new string data type. The reason the string data type is not used is that it is experimental and its functionality could change in the future. Let's begin by reading in the employee dataset and selecting the `dept` column as a Series.

```
[1]: import pandas as pd  
emp = pd.read_csv('../data/employee.csv')  
dept = emp['dept']  
dept.head(3)
```

```
[1]: 0          Police  
     1          Other  
     2  Houston Public Works  
Name: dept, dtype: object
```

Attempt to take the mean

Several methods that worked on numeric columns will either not work with strings or provide little value. For instance, the `mean` method raises an exception when attempted on a string column.

```
[2]: dept.mean()
```

```
TypeError: Could not convert ... to numeric
```

Other methods do work

Many of the other methods we covered from the previous chapters in this part work with string columns, such as finding the maximum department. The `max` of a string is based on its alphabetical ordering.

```
[3]: dept.max()
```

[3]: 'Solid Waste Management'

Missing values

Many other methods work with string columns identically as they do with numeric columns. Below, we calculate the number of missing values. Object data type Series can contain multiple missing value representations. The numpy NaN and NaT and Python None are all counted as missing.

[4]: dept.isna().sum()

[4]: 0

19.1 The value_counts method

The `value_counts` method is one of the most valuable methods for string columns. It returns the count of each unique value in the Series and sorts it from most to least common.

[5]: dept.value_counts()

```
[5]: Police          7573
      Fire           4376
      Houston Public Works 4190
      Other          3373
      Health & Human Services 1353
      Houston Airport System 1216
      Parks & Recreation    1152
      Library          563
      Solid Waste Management 512
      Name: dept, dtype: int64
```

Notice what object is returned

The `value_counts` method returns a Series with the unique values as the index and the count as the new values.

Use `normalize=True` for relative frequency

We can use `value_counts` to return the relative frequency (proportion) of each occurrence instead of the raw count by setting the parameter `normalize` to `True`. For instance, this tells us that 39% of the employees are members of the police department. To reduce the amount of noise and simplify the output, the frequencies are rounded to the nearest hundredth place.

[6]: dept.value_counts(normalize=True).round(2)

```
[6]: Police          0.31
      Fire           0.18
      Houston Public Works 0.17
      Other          0.14
      Health & Human Services 0.06
      Houston Airport System 0.05
      Parks & Recreation    0.05
      Library          0.02
```

```
Solid Waste Management      0.02
Name: dept, dtype: float64
```

value_counts works for columns of all data types

The `value_counts` method works for columns of all data types, and not just strings. It's just usually more informative for string columns. Let's use it on the salary column to see if we have common salaries.

```
[7]: emp['salary'].value_counts().head()
```

```
[7]: 75942.10    1291
87545.38     651
68116.62     639
62540.14     521
48189.70     469
Name: salary, dtype: int64
```

19.2 Special methods just for object columns

pandas provides a collection of methods only available to object (and string) columns. These methods are not directly available using dot notation from the DataFrame variable name and you will not be able to find them as you normally do.

To access these special string-only methods, first append the Series variable name with `.str` followed by another dot and then the specific string method. pandas refers to this as the `str` accessor. Think of the term ‘accessor’ as giving the Series *access* to specialized string methods. [Visit the official documentation](#) to take a look at the several dozen string-only methods available with the `str` accessor. Let's use the title column for these string-only methods.

```
[8]: title = emp['title']
title.head()
```

```
[8]: 0          POLICE SERGEANT
1    ASSISTANT CITY ATTORNEY II
2        SENIOR SLUDGE PROCESSOR
3        SENIOR POLICE OFFICER
4        SENIOR POLICE OFFICER
Name: title, dtype: object
```

Make each value lowercase

Let's begin by calling a simple string method to make each value in the `title` Series uppercase. We will use the `lower` method of the `str` accessor.

```
[9]: title.str.lower().head()
```

```
[9]: 0          police sergeant
1    assistant city attorney ii
2        senior sludge processor
3        senior police officer
4        senior police officer
Name: title, dtype: object
```

Lot's of methods, but mostly easy to use

There is quite a lot of functionality to manipulate and probe strings in almost any way you can imagine. We will not cover every single method possible, but instead, walk through examples of some of the more common ones such as the ones that follow here:

- `count` - Returns the number of non-overlapping occurrences of the passed string.
- `contains` - Checks to see whether each string contains the given string. Returns a boolean Series
- `len` - Returns the number of characters in each string
- `split` - Splits the string into multiple strings by a given separator
- `replace` - Replaces parts of a string with other characters

19.3 The `count` string method

The `count` method returns the number of non-overlapping occurrences of the passed string. Here, we count the number of uppercase ‘O’ characters appear in each string.

```
[10]: title.str.count('O').head()
```

```
[10]: 0    1
      1    1
      2    3
      3    3
      4    3
Name: title, dtype: int64
```

You are not limited to single characters. Here, we count the number of times ‘ER’ appears in each string.

```
[11]: title.str.count('ER').head()
```

```
[11]: 0    1
      1    0
      2    0
      3    1
      4    1
Name: title, dtype: int64
```

19.4 The `contains` str method

The `contains` method returns a boolean whether or not the passed string is contained somewhere within the string. Let’s determine if any titles contain the letter ‘Z’?

```
[12]: title.str.contains('Z').head()
```

```
[12]: 0    False
      1    False
      2    False
      3    False
      4    False
Name: title, dtype: bool
```

We can then sum this boolean Series to find the number of employees that have a title containing ‘Z’.

```
[13]: title.str.contains('Z').sum()
```

```
[13]: 0
```

Let's find out which employees have the word 'POLICE' somewhere in their title.

```
[14]: title.str.contains('POLICE').head()
```

```
[14]: 0    True
      1    False
      2    False
      3    True
      4    True
Name: title, dtype: bool
```

Summing this Series reveals the number of employees that have the word 'POLICE' somewhere in their title.

```
[15]: title.str.contains('POLICE').sum()
```

```
[15]: 6690
```

19.5 The len str method

The `len` string method returns the length of every string. Take note that this is completely different and unrelated to the `len` built-in function which returns the number of elements in a Series.

```
[16]: title.str.len().head()
```

```
[16]: 0    15
      1    26
      2    23
      3    21
      4    21
Name: title, dtype: int64
```

19.6 The split str method

The `split` method splits each string into multiple separate strings based on a given separator. The default separator is a single space. The following splits on each space and returns a Series of lists.

```
[17]: title.str.split().head(3)
```

```
[17]: 0          [POLICE, SERGEANT]
      1          [ASSISTANT, CITY, ATTORNEY, II]
      2          [SENIOR, SLUDGE, PROCESSOR]
Name: title, dtype: object
```

Set the `expand` parameter to `True` to return a DataFrame:

```
[18]: title.str.split(expand=True).head(3)
```

	0	1	2	3	4	5	6
0	POLICE	SERGEANT		None	None	None	None
1	ASSISTANT		CITY	ATTORNEY	II	None	None
2	SENIOR	SLUDGE	PROCESSOR		None	None	None

Here, we split on the string ‘AN’. Note that the string used for splitting is removed and not contained in the result.

```
[19]: title.str.split('AN', expand=True).head(3)
```

	0	1	2	3
0	POLICE SERGE	T	None	None
1	ASSIST T CITY ATTORNEY II	None	None	
2	SENIOR SLUDGE PROCESSOR	None	None	None

19.7 The replace str method

The `replace` string method allows you to replace one section of the string (a substring) with some other string. You must pass two string arguments to `replace` - the string you want to replace and its replacement value. Here, we replace ‘SENIOR’ with ‘SR.’

```
[20]: title.head()
```

```
[20]: 0      POLICE SERGEANT
 1    ASSISTANT CITY ATTORNEY II
 2      SENIOR SLUDGE PROCESSOR
 3      SENIOR POLICE OFFICER
 4      SENIOR POLICE OFFICER
Name: title, dtype: object
```

```
[21]: title.str.replace('SENIOR', 'SR.').head()
```

```
[21]: 0      POLICE SERGEANT
 1    ASSISTANT CITY ATTORNEY II
 2      SR. SLUDGE PROCESSOR
 3      SR. POLICE OFFICER
 4      SR. POLICE OFFICER
Name: title, dtype: object
```

19.8 Selecting substrings with the brackets

Selecting one or more characters of a regular Python string is simple and accomplished by using either an integer or slice notation within the brackets. Let’s review this concept now.

```
[22]: some_string = 'The Astros will win the world series in 2020 without cheating'
```

Select the character at integer location 5.

```
[23]: some_string[5]
```

```
[23]: 's'
```

Select the 24th to 36th characters with slice notation.

```
[24]: some_string[24:36]
```

```
[24]: 'world series'
```

You can use the same square brackets appended directly to the `str` accessor to select one or more characters from every string in a Series. Let's begin by selecting the character at integer location 10.

```
[25]: title.str[10].head(3)
```

```
[25]: 0    G
      1    C
      2    D
Name: title, dtype: object
```

In the following example, we use slice notation to select the last five characters of each string.

```
[26]: title.str[-5:].head(3)
```

```
[26]: 0    GEANT
      1    EY II
      2    ESSOR
Name: title, dtype: object
```

Slice notation is used again to select from the 5th to 15th character.

```
[27]: title.str[5:15].head()
```

```
[27]: 0    E SERGEANT
      1    TANT CITY
      2    R SLUDGE P
      3    R POLICE O
      4    R POLICE O
Name: title, dtype: object
```

Many more string-only methods

There are many more string-only methods that were not covered in this chapter. I would encourage you to explore them on your own. Many of them overlap with the built-in Python string methods.

19.9 Regular expressions

Regular expressions help match patterns within text. Many of the string methods presented above accept regular expressions as inputs for more advanced string maneuvering. They are an important part of doing data analysis and are covered thoroughly in their own part of this book.

19.10 Exercises

Read in the movie dataset assigning the actor1 column to a variable name as a Series by executing the cell below. All missing values have been dropped from this Series. Use this Series for the exercises below.

```
[28]: movie = pd.read_csv('../data/movie.csv', index_col='title')
actor1 = movie['actor1'].dropna()
actor1.head(3)
```

```
[28]: title
Avatar           CCH Pounder
Pirates of the Caribbean: At World's End    Johnny Depp
Spectre          Christoph Waltz
Name: actor1, dtype: object
```

Exercise 1

Which actor 1 has appeared in the most movies? Can you write an expression that returns this actors name as a string?

```
[ ]:
```

Exercise 2

What percent of movies have the top 100 most frequent actor 1's appeared in?

```
[ ]:
```

Exercise 3

How many actor 1's have appeared in exactly one movie?

```
[ ]:
```

Exercise 4

How many actor 1's have more than 3 e's in their name? Output a unique array of just these actor names so we can manually verify them.

```
[ ]:
```

Exercise 5

Get a unique list of all actors that have the name 'Johnson' as part of their name.

```
[ ]:
```

Exercise 6

How many unique actor 1 names end in 'x'?

```
[ ]:
```

Exercise 7

The pandas string methods overlap with the built-in Python string methods. Find all the public method names that are in-common to both. Then find the public methods that are unique to each.

[]:

Chapter 20

Series Datetime Methods

In this chapter, we focus on methods that work for Series containing datetime data. Just like pandas has the `str` accessor to give us access to string-only methods, it also has the `dt` accessor to give us access to datetime-only methods. Let's read in the bikes dataset which has two datetime columns, `starttime` and `stoptime`.

```
[1]: import pandas as pd  
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])  
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

20.1 The `dt` accessor

This chapter focuses on the attributes and methods that are available with the `dt` accessor. [Visit the API](#) to view all of them.

Only available for Series

The `dt` and `str` accessors are only available to Series and not DataFrames. You will have to select a single Series first in order to use them. Let's begin by selecting the `starttime` column as a Series.

```
[2]: start = bikes['starttime']
```

Datetime attributes and methods are simpler than strings

Almost all the attributes and methods available to datetime Series are simple and straightforward. Let's begin by outputting the head of the Series so that we can visually verify the results of the attributes and methods.

```
[3]: start.head(3)
```

```
[3]: 0    2013-06-28 19:01:00  
1    2013-06-28 22:53:00  
2    2013-06-30 14:43:00  
Name: starttime, dtype: datetime64[ns]
```

20.2 Datetime Attributes

Unlike the `str` accessor, many of the available objects to the `dt` accessor are attributes and not methods. These are not called, but simply accessed to return a new Series.

Retrieving a part of the datetime

There are many attributes that return a particular part of the datetime such as `year`, `month`, `day`, `hour`, `minute`, `second`, etc... as integers. Let's see retrieve these components of the datetime as their own Series.

```
[4]: start.dt.year.head(3)
```

```
[4]: 0    2013  
1    2013  
2    2013  
Name: starttime, dtype: int64
```

```
[5]: start.dt.month.head(3)
```

```
[5]: 0    6  
1    6  
2    6  
Name: starttime, dtype: int64
```

```
[6]: start.dt.day.head(3)
```

```
[6]: 0    28  
1    28  
2    30  
Name: starttime, dtype: int64
```

```
[7]: start.dt.hour.head(3)
```

```
[7]: 0    19  
1    22  
2    14  
Name: starttime, dtype: int64
```

```
[8]: start.dt.minute.head(3)
```

```
[8]: 0    1  
1    53  
2    43  
Name: starttime, dtype: int64
```

```
[9]: start.dt.second.head(3)
```

```
[9]: 0    0  
1    0  
2    0  
Name: starttime, dtype: int64
```

We can also return the day and week of the year as integers. For day of the week, 0 corresponds to Monday and 6 to Sunday.

```
[10]: start.dt.dayofweek.head(3)
```

```
[10]: 0    4  
1    4  
2    6  
Name: starttime, dtype: int64
```

```
[11]: start.dt.weekofyear.head()
```

```
[11]: 0    26  
1    26  
2    26  
3    27  
4    27  
Name: starttime, dtype: int64
```

Start or End?

There are several attributes that return boolean Series based on whether the datetime is the start or end of the month, quarter, or year.

```
[12]: start.dt.is_month_start.head()
```

```
[12]: 0    False  
1    False  
2    False  
3    True  
4    True  
Name: starttime, dtype: bool
```

```
[13]: start.dt.is_quarter_end.head()
```

```
[13]: 0    False  
1    False  
2    True  
3    False  
4    False  
Name: starttime, dtype: bool
```

```
[14]: start.dt.is_year_start.head()
```

```
[14]: 0    False
      1    False
      2    False
      3    False
      4    False
Name: starttime, dtype: bool
```

20.3 Datetime methods

There are only a few methods that are available to the `dt` accessor with the most useful being `ceil`, `round`, `floor`, `strftime`, and `to_period`. To use these methods, you need to be familiar with the [offset aliases](#), which are short strings, usually one character in length, that represent a unit of time. Below are a few of the offset aliases.

- H - hour
- T or min - minute
- S - second
- D - day
- W - week

Use offset aliases with datetime methods

Let's output our datetime Series again, and then call some of these methods that require offset aliases.

```
[15]: start.head(3)
```

```
[15]: 0    2013-06-28 19:01:00
      1    2013-06-28 22:53:00
      2    2013-06-30 14:43:00
Name: starttime, dtype: datetime64[ns]
```

`ceil` rounds up to the nearest unit

Use the `ceil` method to round up to the nearest hour by using the offset alias '`H`'.

```
[16]: start.dt.ceil('H').head(3)
```

```
[16]: 0    2013-06-28 20:00:00
      1    2013-06-28 23:00:00
      2    2013-06-30 15:00:00
Name: starttime, dtype: datetime64[ns]
```

Round up to the nearest day.

```
[17]: start.dt.ceil('D').head(3)
```

```
[17]: 0    2013-06-29
      1    2013-06-29
      2    2013-07-01
Name: starttime, dtype: datetime64[ns]
```

floor rounds down to the nearest unit

Use the `floor` method to round down to the nearest minute.

```
[18]: start.dt.floor('min').head(3)
```

```
[18]: 0    2013-06-28 19:01:00  
1    2013-06-28 22:53:00  
2    2013-06-30 14:43:00  
Name: starttime, dtype: datetime64[ns]
```

round rounds to nearest whole unit

The `round` method uses typical rounding logic. Here, we round to the nearest hour.

```
[19]: start.dt.round('H').head(3)
```

```
[19]: 0    2013-06-28 19:00:00  
1    2013-06-28 23:00:00  
2    2013-06-30 15:00:00  
Name: starttime, dtype: datetime64[ns]
```

20.4 Format time as a string with strftime

The `strftime` method stands for **string format time**. It converts each datetime value into a string object. You will use something called **string directives** to convert a part of a datetime to a string. For instance, the string directive ‘%A’ converts to the weekday. Consult [Python’s documentation](#) to view all of the string directives.

Below is an example using multiple string directives to form a complex string from a datetime. You can write any other string intertwined with the directives.

By default, the maximum column width of a pandas DataFrame is 60 characters. The `set_option` function is used to increase this width so that the entire new string value is viewable in the output.

```
[20]: pd.set_option('display.max_colwidth', 100)  
start.dt.strftime('On %A, %B %d, %Y at %X something great happened').head(3)
```

```
[20]: 0    On Friday, June 28, 2013 at 19:01:00 something great happened  
1    On Friday, June 28, 2013 at 22:53:00 something great happened  
2    On Sunday, June 30, 2013 at 14:43:00 something great happened  
Name: starttime, dtype: object
```

20.5 Convert to period

A period is a special data type unique to pandas (it don’t exist in numpy) and represents an entire period of time such as the entire month of June, 2012, the entire year 1998, or the entire minute of June 11, 2011 12:34 p.m. This contrasts with datetimes, which represent a single moment in time with nanosecond precision. In pandas, datetimes always have precision down to nanoseconds. A period refers to some period of time.

Use offset aliases to convert to a period

To convert to a datetime column to a period column, use the same offset aliases from above. Let's convert the start datetime column to a period column representing an entire month.

```
[21]: per = start.dt.to_period('M').head()
per
```

```
[21]: 0    2013-06
      1    2013-06
      2    2013-06
      3    2013-07
      4    2013-07
Name: starttime, dtype: period[M]
```

Let's verify that the data type of this Series is indeed a period.

```
[22]: per.dtype
```

```
[22]: period[M]
```

Let's see another example with a different offset alias converting the datetime to a time period of an hour.

```
[23]: start.dt.to_period('h').head(3)
```

```
[23]: 0    2013-06-28 19:00
      1    2013-06-28 22:00
      2    2013-06-30 14:00
Name: starttime, dtype: period[H]
```

Period Series also have a dt accessor

A Series with data type of period has its own special attributes and methods accessible with the `dt` accessor. They overlap substantially with the datetime `dt` attributes and methods. Currently the [official documentation only shows the period properties](#). You can discover all of the attributes and methods and how to use them by placing a dot after the `dt` and pressing tab. Below, we get the start and end of the period. Note that pandas returns these values as datetimes and not periods.

```
[24]: per.dt.start_time
```

```
[24]: 0    2013-06-01
      1    2013-06-01
      2    2013-06-01
      3    2013-07-01
      4    2013-07-01
Name: starttime, dtype: datetime64[ns]
```

```
[25]: per.dt.end_time
```

```
[25]: 0    2013-06-30 23:59:59.999999999
      1    2013-06-30 23:59:59.999999999
      2    2013-06-30 23:59:59.999999999
      3    2013-07-31 23:59:59.999999999
```

```
4 2013-07-31 23:59:59.999999999
Name: starttime, dtype: datetime64[ns]
```

20.6 Timedeltas

Timedeltas are a separate data type that represent an amount of time such as 5 minutes and 34 seconds. The highest unit of a timedelta is days and they always have nanosecond precision. Timedeltas are also available in numpy. Timedelta Series have special attributes and methods accessible with the `dt` accessor as you can [find in the documentation](#).

Creating a timedelta

One way to create a timedelta Series is to subtract two datetime Series from each other. Here, we select `stoptime` as a Series and subtract the `start` Series from it.

```
[26]: stop = bikes['stoptime']
ride_length = stop - start
ride_length.head(3)
```

```
[26]: 0    00:16:00
1    00:10:00
2    00:18:00
dtype: timedelta64[ns]
```

Again, a good way to discover and learn about the attributes and methods is by pressing tab after placing a dot after `dt`. Let's begin by converting each of the timedeltas into seconds.

```
[27]: ride_length.dt.seconds.head(3)
```

```
[27]: 0      960
1      600
2     1080
dtype: int64
```

There are a few timedelta methods that take offset aliases. Numbers may be placed next to offset aliases to designate a more specific amount of time. Below, we round to the nearest 10 minutes.

```
[28]: ride_length.dt.round('10min').head(3)
```

```
[28]: 0    00:20:00
1    00:10:00
2    00:20:00
dtype: timedelta64[ns]
```

20.7 Exercises

Use the `start` Series for the following exercises.

Exercise 1

What percentage of bike rides happen in January?

[]:

Exercise 2

What percentage of bike rides happen on the weekend?

[]:

Exercise 3

What percentage of bike rides happen on the last day of the month?

[]:

Exercise 4

We would expect that the value of the minutes recorded for each starting ride is approximately random. Can you show some data that confirms or rejects this?

[]:

Exercise 5

Assign the length of the ride to `ride_length`. Then find the percentage of rides that lasted longer than 30 minutes.

[]:

Chapter 21

Project - Testing Normality of Stock Market Returns

In this chapter, we examine the daily return of Microsoft's stock to determine if it follows a normal distribution.

Getting Microsoft stock price data

Twenty years of Microsoft stock price data is stored in the msft20.csv file in the stocks directory. Let's read this in setting the date as the index.

```
[1]: import pandas as pd  
msft = pd.read_csv('../data/stocks/msft20.csv', parse_dates=['date'], index_col='date')  
msft.head()
```

	open	high	low	close	adjusted_close	volume	dividend_amount
date							
1999-10-19	88.250	89.250	85.250	86.313	27.8594	69945600	0.0
1999-10-20	91.563	92.375	90.250	92.250	29.7758	88090600	0.0
1999-10-21	90.563	93.125	90.500	93.063	30.0381	60801200	0.0
1999-10-22	93.563	93.875	91.750	92.688	29.9171	43650600	0.0
1999-10-25	92.000	93.563	91.125	92.438	29.8364	30492200	0.0

Select the closing price

For this problem, we are only interested in the closing price. Select the adjusted_close as this is adjusted for any stock splits that may have occurred.

```
[2]: close = msft['adjusted_close']  
close.head(3)
```

```
[2]: date  
1999-10-19    27.8594  
1999-10-20    29.7758
```

```
1999-10-21    30.0381
Name: adjusted_close, dtype: float64
```

Daily percent change

pandas Series have a method called `pct_change`, which returns the percentage difference between the current and previous elements. Let's use it to calculate the daily return.

```
[3]: close_change = close.pct_change()
close_change.head(3)
```

```
[3]: date
1999-10-19      NaN
1999-10-20    0.068788
1999-10-21    0.008809
Name: adjusted_close, dtype: float64
```

Handling Missing Value

The first date has a missing value since there was no previous date. The `dropna` method can be used to remove any `NaN` elements.

```
[4]: close_change = close_change.dropna()
close_change.head(3)
```

```
[4]: date
1999-10-20    0.068788
1999-10-21    0.008809
1999-10-22   -0.004028
Name: adjusted_close, dtype: float64
```

Checking for Normality

There are formal statistical tests for normality that can be used. Instead we will focus on simple data exploration to give us insight.

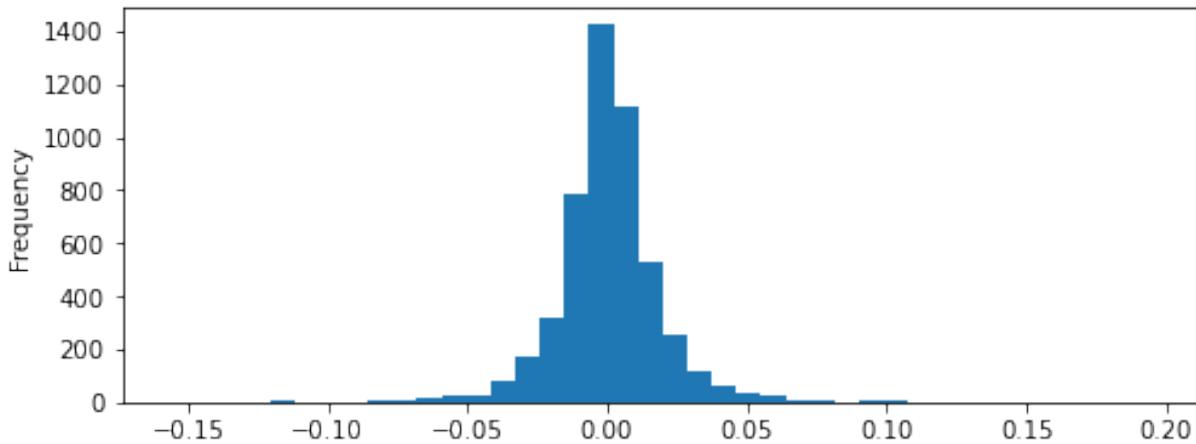
Plotting the returns

The main plotting library in Python is matplotlib which will be covered in greater detail in the **Visualization** part. To output plots directly into the notebook, the magic command `%matplotlib inline` must be executed first.

```
[5]: %matplotlib inline
```

pandas objects have hooks into matplotlib so it's not necessary to import matplotlib directly. The `plot` method can be used to create a number of different kinds of plots with the `kind` parameter. We pass it the string `hist` to create a histogram, along with a histogram-specific argument `bins` to control the number of bins which is also the number of bars plotted. We also change the size of the figure (in inches) by passing a tuple to the `figsize` parameter.

```
[6]: close_change.plot(kind='hist', bins=40, figsize=(8, 3));
```



Use boolean selection to check for normality

The plot above is symmetrical and somewhat bell-shaped. It could possibly represent a normal distribution. To more formally check for normality, we can count the number of observations that are within 1, 2, and 3 standard deviations. The [68-95-99.7 rule](#) can be used to determine if the data is approximately normal. We first need to calculate the mean and standard deviation.

```
[7]: mean = close_change.mean()
      std = close_change.std()
      mean, std
```

```
[7]: (0.0005010304494320834, 0.019119644436205206)
```

Absolute number of standard deviations from the mean

To standardize our results, we can find the number of standard deviations away from the mean each daily return is. To do this, we subtract the mean from the entire Series and then divide by the standard deviation. This quantity is referred to as the **z-score**.

```
[8]: z_score = (close_change - mean) / std
```

To help make calculations easier, we will use the absolute value of the z-score.

```
[9]: z_score_abs = z_score.abs()
```

Find the percentage by taking the mean

Let's find the percentage of returns within 1, 2, and 3 standard deviations by taking the mean of a boolean Series.

```
[10]: pct_within1 = round((z_score_abs < 1).mean(), 3)
      pct_within2 = round((z_score_abs < 2).mean(), 3)
      pct_within3 = round((z_score_abs < 3).mean(), 3)
      pct_within1, pct_within2, pct_within3
```

```
[10]: (0.793, 0.951, 0.983)
```

21.1 Results discussion

The percentages of returns within 1, 2 and 3 standard deviations are fairly different than the 68-95-99.7 rule. Much more of the data was concentrated within 1 standard deviation. A much greater percentage of the returns were greater than 3 standard deviations from the mean compared to just .3% for the rule. This strongly suggests that a normal distribution would not be a good fit for this type of data.

Using the percentile to check for normality

Alternatively, we can work backwards and find the z-score that represents the 68th, 95th, and 99.7th percentiles of the distribution. For normally distributed data, we would expect these to be 1, 2, and 3 respectively. The `quantile` method completes this operation for us. Note how far off the 68th and 99.7th percentiles are.

```
[11]: z_score_abs.quantile([.68, .95, .997]).round(2)
```

```
[11]: 0.680    0.71
      0.950    1.98
      0.997    5.16
Name: adjusted_close, dtype: float64
```

Check that all Series values are True

Let's say we wanted to check if all the stock price returns were within 4 standard deviations from the mean. For boolean Series, the `all` method returns `True` if all values are `True`, and `False` otherwise.

```
[12]: criteria = z_score_abs < 4
criteria.head(3)
```

```
[12]: date
1999-10-20    True
1999-10-21    True
1999-10-22    True
Name: adjusted_close, dtype: bool
```

```
[13]: criteria.all()
```

```
[13]: False
```

We can duplicate the above logic with the `any` method. The `any` method returns `True` if one or more values in the Series are `True`. Here, we check if any of the returns are greater than or equal to 4.

```
[14]: criteria = z_score_abs >= 4
criteria.any()
```

```
[14]: True
```

21.2 Exercises

Execute the cells below to read in 20 years of Apple (AAPL) data as a Series and answer the exercises below with it.

```
[15]: stocks = pd.read_csv('../data/stocks/stocks10.csv', index_col='date',
    parse_dates=['date'])
stocks.head(3)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN

```
[16]: aapl = stocks['AAPL']
aapl.head()
```

```
[16]: date
1999-10-25    2.32
1999-10-26    2.34
1999-10-27    2.38
1999-10-28    2.43
1999-10-29    2.50
Name: AAPL, dtype: float64
```

Exercise 1

Use one line of code to find the daily percentage returns of AAPL and drop any missing values. Save the result to `aapl_change`.

[]:

Exercise 2

Find the mean daily return for Apple, the first and last closing prices, and the number of trading days. Store all four of these values into separate variables.

[]:

Exercise 3

If Apple returned its mean percentage return every single day since the first day you have data, what would its last closing price be? Is it the same as the actual last closing price? You need to use all the variables calculated from Exercise 2.

[]:

Exercise 4

Find the z-score for the Apple daily returns. Save this to a variable `z_score_raw`. What is the max and minimum score?

[]:

Exercise 5

What percentage did Apple's stock increase when it had its highest maximum raw z-score?

[]:

Exercise 6

Create a function that accepts a Series of stock closing prices. Have it return the percentage of prices within 1, 2, and 3 standard deviations from the mean. Use your function to return results for different stocks found in the `stocks` DataFrame.

[]:

Exercise 7

How many days did Apple close above 100 and below 120?

[]:

Exercise 8

How many days did Apple close below 50 or above 150?

[]:

Exercise 9

Look up the definition for interquartile range and select all Apple closing prices that are within this range. There are multiple ways to do this. Check the `quantile` method.

[]:

Exercise 10

Find the date of the highest closing price. Find out how many trading days it has been since Apple recorded its highest closing price.

[]:

Part IV

Essential DataFrame Commands

Chapter 22

Introduction to DataFrame

In the previous part, we covered the most common and fundamental attributes and methods for a Series. This chapter begins our coverage of similar and analogous operations for DataFrames.

22.1 DataFrames vs Series

DataFrames and Series are extremely similar objects. A Series is just a single dimension of data and is usually formed from a single column of a DataFrame. Series have an index and values, but no columns. A DataFrame can be thought of as a collection of Series objects each directly accessible by placing the column name within *just the brackets*.

View the API for a complete list of functionality

As we did for Series, it can be helpful to see the entire list of attributes and methods available to DataFrames. Visit the [DataFrame section](#) of the API to see all of its functionality.

Best of DataFrame API

DataFrames have an abundance of attributes and methods that do not give any additional functionality to the library. We focus on the core attributes and methods that give you the most power to complete a data analysis.

Minimally Sufficient Pandas

As a gentle reminder, it is my opinion that you stick with a minimal subset of pandas when doing an analysis. Using more obscure methods does not make you a better analyst. The point of a data analysis is to clearly expose the information held within the data. Just about everything that you want to do can be clearly expressed with minimal pandas syntax.

Bikes dataset

We will use the bikes dataset to introduce the core attributes and methods of DataFrames.

```
[1]: import pandas as pd  
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])  
bikes.head(3)
```

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy
2	10927	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	...	73.0	10.0	16.1	-9999.0	mostlycloudy

22.2 Core DataFrame attributes

The core DataFrame attributes that you need to know are listed below.

- `index`
- `columns`
- `values`
- `dtypes`
- `shape`
- `size`

Review the index and columns

We've discussed the index and columns extensively before this chapter. As a review, the purpose of the index is to label each row, just as the the purpose for the columns is to label each column. The index and columns are not data. They are labels for the data. If no index is set during read, pandas uses the default `RangeIndex` which defines a sequence of consecutive integers beginning at 0.

```
[2]: bikes.index
```

```
[2]: RangeIndex(start=0, stop=50089, step=1)
```

Let's access the column names with the `columns` attribute.

```
[3]: bikes.columns
```

```
[3]: Index(['trip_id', 'gender', 'starttime', 'stoptime', 'tripduration',
       'from_station_name', 'dpcapacity_start', 'to_station_name',
       'dpcapacity_end', 'temperature', 'visibility', 'wind_speed',
       'precipitation', 'events'],
       dtype='object')
```

values returns a 2-D numpy array

The `values` attribute returns a two-dimensional numpy array of all the column values. It is like a DataFrame with no index or columns.

```
[4]: bikes.values
```

```
[4]: array([[7147, 'Male', Timestamp('2013-06-28 19:01:00'), ..., 12.7,
       -9999.0, 'mostlycloudy'],
          [7524, 'Male', Timestamp('2013-06-28 22:53:00'), ..., 6.9,
```

```
-9999.0, 'partlycloudy'],
[10927, 'Male', Timestamp('2013-06-30 14:43:00'), ..., 16.1,
-9999.0, 'mostlycloudy'],
...,
[17534972, 'Male', Timestamp('2017-12-30 13:34:00'), ..., 16.1,
-9999.0, 'partlycloudy'],
[17535645, 'Female', Timestamp('2017-12-31 09:30:00'), ..., 11.5,
-9999.0, 'partlycloudy'],
[17536246, 'Male', Timestamp('2017-12-31 15:22:00'), ..., 15.0,
-9999.0, 'partlycloudy']], dtype=object)
```

Advanced discussion on the `values` attribute

The `values` attribute always returns a single numpy array, which may lead you to believe that pandas stores its data as a single numpy array. This isn't the case. pandas has separate two-dimensional numpy arrays for each data type in the DataFrame. For instance, if a DataFrame contains integers, floats, strings, and datetimes, then pandas will have four separate numpy arrays to contain data for each of those data types. The reason for this, is that numpy arrays can only be of one specific data type. In other words, numpy arrays contain **homogeneous** data. The exception to this is the 'object' numpy data type which can contain any Python object.

Whenever you access the `values` attribute of a DataFrame, pandas concatenates together all of the numpy arrays together to return a single array. Notice that the data type of the resulting `bikes.values` array is object. This is because the `bikes` DataFrame contains string columns and the only valid numpy data type that contains both numeric and string values is object.

If you had a DataFrame consisting only of integers and floats, then the `values` attribute would return an array with a data type of float. pandas always chooses the data type that loses no information when you access the `values` attribute. Below, an integer column (`tripduration`) and a float column (`dpcapacity_start`) are selected and then the `values` attribute is accessed to return a numpy array with a float data type.

```
[5]: bikes[['tripduration', 'dpcapacity_start']].values.dtype
```

```
[5]: dtype('float64')
```

shape returns a tuple of the number of rows and columns

The `shape` attribute returns a Python tuple of length 2 containing the number of rows and columns.

```
[6]: shape = bikes.shape
shape
```

```
[6]: (50089, 14)
```

You can get the number of rows or columns as an integer by selecting them from the tuple.

```
[7]: shape[0]
```

```
[7]: 50089
```

It isn't necessary to assign the tuple to a variable beforehand.

```
[8]: bikes.shape[1]
```

[8]: 14

size returns the total number of elements in the DataFrame

The `size` attribute is a bit tricky and returns the total number of elements in the DataFrame. This is simply the number of rows multiplied by the number of columns.

[9]: `bikes.size`

[9]: 701246

We can verify this by multiplying the number of rows and columns together.

[10]: `shape[0] * shape[1]`

[10]: 701246

The `len` function returns the number of rows

Passing in the DataFrame to the built-in Python `len` function returns the number of rows.

[11]: `len(bikes)`

[11]: 50089

22.3 Arithmetic DataFrame operations

We now cover the arithmetic operators `+`, `-`, `*`, `/`, `**`, `//`, `%` on a DataFrame. Let's say we have DataFrame, `df`, and execute `df + 5`. pandas attempts to add 5 to each value in the DataFrame. This operation only completes if all the columns are numeric (or boolean).

Attempt to add 5 to bikes

If we attempt to add 5 to `bikes` we get an error, as we have a mix of numeric, object, and datetime columns. Adding the integer 5 to a string or datetime columns is impossible and a `TypeError` is raised.

[12]: `bikes + 5`

```
TypeError: Addition/subtraction of integers and integer-arrays with
         DatetimeArray is no longer supported. Instead of adding/subtracting `n`, use `n *
         obj.freq`
```

Select only numeric data with `select_dtypes`

DataFrames have a method unique to them called `select_dtypes` which selects a subset of columns with the passed type. Pass it the data type you want to select as a string. For example, pass it the string '`int64`' to select all the 64-bit integer columns.

[13]: `bikes.select_dtypes('int64').head(3)`

	trip_id	tripduration
0	7147	993
1	7524	623
2	10927	1040

Select all 64-bit float columns by passing it the string ‘float64’.

```
[14]: bikes.select_dtypes('float64').head(3)
```

	dpcapacity_start	dpcapacity_end	temperature	visibility	wind_speed	precipitation
0	11.0	15.0	73.9	10.0	12.7	-9999.0
1	31.0	19.0	69.1	10.0	6.9	-9999.0
2	15.0	23.0	73.0	10.0	16.1	-9999.0

All columns from each data type may be selected in this manner. You can also select columns of multiple data types by using a list. Here, we select both 64-bit integers and floats.

```
[15]: bikes.select_dtypes(['int64', 'float64']).head(3)
```

	trip_id	tripduration	dpcapacity_start	dpcapacity_end	temperature	visibility	wind_speed	precipitation
0	7147	993	11.0	15.0	73.9	10.0	12.7	-9999.0
1	7524	623	31.0	19.0	69.1	10.0	6.9	-9999.0
2	10927	1040	15.0	23.0	73.0	10.0	16.1	-9999.0

Integers and floats of different bit sizes are considered different data types and you will need to use their exact name to select them with this method. For instance, ‘int16’ selects all 16-bit integer columns. A deep dive into all of pandas data types is found in the 05. Data Types part.

Use the string ‘number’ to select all numeric data

pandas offers a shortcut to select all numeric data types (integers and floats) with the string ‘number’. We assign this result to `bikes_number`.

```
[16]: bikes_number = bikes.select_dtypes('number')
bikes_number.head(3)
```

	trip_id	tripduration	dpcapacity_start	dpcapacity_end	temperature	visibility	wind_speed	precipitation
0	7147	993	11.0	15.0	73.9	10.0	12.7	-9999.0
1	7524	623	31.0	19.0	69.1	10.0	6.9	-9999.0
2	10927	1040	15.0	23.0	73.0	10.0	16.1	-9999.0

Add 5 to `bikes_number`

Now that we have a DataFrame consisting entirely of numeric data, we can successfully add 5 to it.

```
[17]: (bikes_number + 5).head(3)
```

	trip_id	tripduration	dpcapacity_start	dpcapacity_end	temperature	visibility	wind_speed	precipitation
0	7152	998	16.0	20.0	78.9	15.0	17.7	-9994.0
1	7529	628	36.0	24.0	74.1	15.0	11.9	-9994.0
2	10932	1045	20.0	28.0	78.0	15.0	21.1	-9994.0

Addition and multiplication with string columns

Addition actually works with strings by appending the word being added to each value. You can also use multiplication to concatenate each string value to itself. We first select all of the string columns and assign the result to a new variable name.

```
[18]: bikes_strings = bikes.select_dtypes('object')
bikes_strings.head(3)
```

	gender	from_station_name	to_station_name	events
0	Male	Lake Shore Dr & Monroe St	Michigan Ave & Oak St	mostlycloudy
1	Male	Clinton St & Washington Blvd	Wells St & Walton St	partlycloudy
2	Male	Sheffield Ave & Kingsbury St	Dearborn St & Monroe St	mostlycloudy

The addition operator can now be used to append a string to each value in the DataFrame.

```
[19]: (bikes_strings + ' SOMESTRING').head(3)
```

	gender	from_station_name	to_station_name	events
0	Male SOMESTRING	Lake Shore Dr & Monroe St SOMESTRING	Michigan Ave & Oak St SOMESTRING	mostlycloudy SOMESTRING
1	Male SOMESTRING	Clinton St & Washington Blvd SOMESTRING	Wells St & Walton St SOMESTRING	partlycloudy SOMESTRING
2	Male SOMESTRING	Sheffield Ave & Kingsbury St SOMESTRING	Dearborn St & Monroe St SOMESTRING	mostlycloudy SOMESTRING

Similarly, the multiplication operator can be used with an integer to concatenate each string value to itself.

```
[20]: (bikes_strings * 3).head(3)
```

	gender	from_station_name	to_station_name	events
0	MaleMaleMale	Lake Shore Dr & Monroe St Lake Shore Dr & Monroe...	Michigan Ave & Oak St Michigan Ave & Oak St Michigan...	mostlycloudy mostlycloudy mostlycloudy
1	MaleMaleMale	Clinton St & Washington Blvd Clinton St & Washi...	Wells St & Walton St Wells St & Walton St Wells ...	partlycloudy partlycloudy partlycloudy
2	MaleMaleMale	Sheffield Ave & Kingsbury St Sheffield Ave & Ki...	Dearborn St & Monroe St Dearborn St & Monroe St...	mostlycloudy mostlycloudy mostlycloudy

22.4 DataFrames comparison operators

The comparison operators (`<`, `<=`, `>`, `>=`, `==`, `!=`) work similarly to the arithmetic ones and return a DataFrame of all boolean columns. Here, we test whether every value in the DataFrame is greater than 5.

[21]: `(bikes_number > 5).head(3)`

	trip_id	tripduration	dpcapacity_start	dpcapacity_end	temperature	visibility	wind_speed	precipitation
0	True	True	True	True	True	True	True	False
1	True	True	True	True	True	True	True	False
2	True	True	True	True	True	True	True	False

Take care when working with the entire DataFrame

When you perform one of the above operations, you are applying that operation to every value in the DataFrame. Make sure this is what you want because all values will be affected.

22.5 Overlap of DataFrame and Series methods

Most of the methods that exist for Series also exist for DataFrames and vice-versa. This is good news as it would be a huge hassle to have a different set of methods for such similar objects. In this section, we find all the attributes and methods that are either in-common or unique to Series and DataFrames. We begin by using a set comprehension to get all of the public (those that don't begin with an underscore) attributes and methods for each type.

[22]: `df_public = {method for method in dir(pd.DataFrame) if not method.startswith('_')}`
`series_public = {method for method in dir(pd.Series) if not method.startswith('_')}`

Let's output the total number of methods for each type. Notice how they have nearly the same amount.

[23]: `len(df_public), len(series_public)`

[23]: (205, 207)

Let's find the number of methods in common. The ampersand computes the intersection between sets. About 90% of the attributes and methods are the same.

```
[24]: len(df_public & series_public)
```

[24]: 176

Attributes and methods unique to DataFrames

The minus sign computes the difference between one set and another. It returns all of the elements unique to the first set. Below we return the attributes and methods unique to DataFrames.

```
[25]: print(df_public - series_public)
```

```
{'merge', 'join', 'eval', 'info', 'to_parquet', 'itertuples', 'select_dtypes', 'insert',
'melt', 'pivot_table', 'columns', 'to_html', 'pivot', 'from_dict', 'lookup',
'to_feather', 'applymap', 'to_gbq', 'style', 'to_stata', 'boxplot', 'from_records',
'set_index', 'stack', 'to_records', 'iterrows', 'corrwith', 'query', 'assign'}
```

Output the methods unique to Series

Reversing the operation returns the attributes and methods unique to Series.

```
[26]: print(series_public - df_public)
```

```
{'is_monotonic_decreasing', 'autocorr', 'ravel', 'searchsorted',
'is_monotonic_increasing', 'tolist', 'argmin', 'is_monotonic', 'item', 'dt', 'cat',
'array', 'repeat', 'name', 'value_counts', 'unique', 'hasnans', 'to_frame', ' nbytes',
'factorize', 'argmax', 'map', 'rdivmod', 'is_unique', 'divmod', 'to_list', 'str',
'dtype', 'argsort', 'view', 'between'}
```

22.6 Data Dictionaries

A data dictionary is an important element of a data analysis and at a minimum gives us the column name and description of each column. Other information on each column can be kept in it such as the data type of each column or number of missing values.

The college dataset has a data dictionary available that can be read in as a DataFrame. It contains the descriptions of each column, which is important with this dataset as the column names are not easily decipherable.

```
[27]: # There are more than 20 (the default) columns
pd.set_option('display.max_columns', 40)
college = pd.read_csv('../data/college.csv', index_col='instnm')
college.head(3)
```

instnm	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
Alabama A & M University	Normal	AL	1.0	0.0	0.0	...	0.7356	0.8284	0.1049	30300	33888
University of Alabama at Birmingham	Birmingham	AL	0.0	0.0	0.0	...	0.3460	0.5214	0.2422	39700	21941.5
Amridge University	Montgomery	AL	0.0	0.0	0.0	...	0.6801	0.7795	0.8540	40100	23370

The data dictionary is a CSV, which can be read in as a DataFrame to help understand the information in the college DataFrame.

```
[28]: pd.read_csv('../data/dictionaries/college_data_dictionary.csv')
```

	column_name	description
0	instnm	Institution Name
1	city	City Location
2	stabbr	State Abbreviation
3	hbcu	Historically Black College or University
4	menonly	0/1 Men Only
5	womenonly	0/1 Women only
6	relaffil	0/1 Religious Affiliation
7	satvrmid	SAT Verbal Median
8	satmtmid	SAT Math Median
9	distanceonly	Distance Education Only
10	ugds	Undergraduate Enrollment
11	ugds_white	Percent Undergrad White
12	ugds_black	Percent Undergrad Black
13	ugds_hisp	Percent Undergrad Hispanic
14	ugds_asian	Percent Undergrad Asian
15	ugds_aian	Percent Undergrad American Indian/Alaskan Native
16	ugds_nhpi	Percent Undergrad Native Hawaiian/Pacific Islander
17	ugds_2mor	Percent Undergrad 2 or more races
18	ugds_nra	Percent Undergrad non-resident aliens
19	ugds_unkn	Percent Undergrad race unknown
20	pptug_ef	Percent Students part-time
21	curroper	0/1 Currently Operating
22	pctpell	Percent Students with Pell grant
23	pctfloan	Percent Students with federal loan
24	ug25abv	Percent Students Older than 25
25	md_earn_wne_p10	Median Earnings 10 years after enrollment
26	grad_debt_mdn_supp	Median debt of completers

22.7 Exercises

Exercise 1

Select only the 64-bit float columns from the `college` DataFrame. How many are there?

[]:

Exercise 2

When you call the `info` method on a DataFrame, one of the very last items that gets outputted is the count of columns for each data type. Can you think of a different combination of pandas operations that would return this as a Series.

[]:

Chapter 23

DataFrame Statistical Methods

In this chapter, we cover [statistical methods](#) for the DataFrame, which are identical to those available to a Series. Again, we distinguish between methods that aggregate and those that do not. A method that performs an aggregation returns a **single** number to summarize the values. Any method that does not return a single value is not an aggregation. We begin by reading in the San Francisco employee compensation dataset.

```
[1]: import pandas as pd  
sf_emp = pd.read_csv('../data/sf_employee_compensation.csv')  
sf_emp.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

23.1 Aggregation methods

The following are some common aggregation methods available for DataFrames:

- `sum`
- `min`
- `max`
- `mean`
- `median`
- `std` - standard deviation
- `var` - variance
- `count` - returns number of non-missing values
- `describe` - returns most of the above aggregations in one Series
- `quantile` - returns the given percentile of the distribution

Differences between DataFrame and Series methods

When calling an aggregation method on a DataFrame, it is applied to each individual column by default. For instance, calling the `sum` method sums each column individually. A single value is returned for each column. Calling the `sum` method on a Series produces a single scalar value.

Select numeric columns

Some of these statistical methods above work only with numeric columns. In order to successfully call these methods, we'll select only the columns with compensation information.

```
[2]: comp = sf_emp.iloc[:, 3:]
comp.head()
```

	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	71414.01	0.00	0.00	14038.58	12918.24	5872.04
1	67941.06	0.00	0.00	13030.23	10047.52	5608.37
2	116956.72	59975.43	19037.30	24796.44	15788.97	3222.20
3	31856.00	0.00	0.00	6791.73	5262.99	2574.91
4	29590.58	0.00	5898.73	960.81	0.00	9230.03

Take the mean of each column

Let's demonstrate taking the mean of each column by calling the `mean` aggregation method.

```
[3]: comp.mean()
```

```
[3]: salaries      53715.441133
overtime       4201.272687
other salaries  2816.296542
retirement     10484.755614
health and dental  9382.390735
other benefits   4053.381941
dtype: float64
```

Did you notice what type of object was returned?

pandas takes the mean of each column and returns a Series. The new Series uses the old column names as the index and the calculated mean as the values. Let's call a couple of different aggregation methods.

```
[4]: comp.max()
```

```
[4]: salaries      645739.46
overtime       258124.17
other salaries  239294.57
retirement     120791.40
health and dental  36369.96
other benefits   37563.46
dtype: float64
```

```
[5]: comp.std()
```

```
[5]: salaries      47686.502923
overtime       11601.573498
other salaries  6637.820066
retirement     9922.598455
health and dental 7379.199008
other benefits  4171.974274
dtype: float64
```

Potentially confusing orientation

The above results should be fairly easy to understand. If someone asked you what the standard deviation of the `overtime` column, you would easily be able to respond with the correct number. What is potentially confusing is the orientation of the result. We began with a DataFrame, and were returned a Series which is visually displayed in the notebook as a vertical sequence of values. The orientation of the columns changed. It might have been easier to understand the operation if the columns remained horizontal as in the following image.

	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	53715.441133	4201.272687	2816.296542	10484.755614	9382.390735	4053.381941

DataFrames are collections of columns

It's good to think of DataFrames as a collection of columns as opposed to a collection of rows. It is the column that is the fundamental component of the DataFrame. Each column has a data type and all values in that column are the same data type. It is the column that is acted on by default by most of the methods as demonstrated with the aggregations above.

23.2 Changing the direction of the operation

Since DataFrames are two-dimensional, we might want to complete an operation horizontally across the rows instead of vertically down the columns.

The `axis` parameter controls the direction of the operation

Most DataFrame methods have an `axis` parameter. This is a crucial parameter to understand as it controls the direction of the operation. By default, operations happen vertically down each column.

Each axis may be referenced by number or string label

DataFrames are two-dimensional and therefore have two axes. Both the rows and the columns may be referenced with a number or a string label. The rows are referenced by the number 0 and also by the label 'index'. The columns are referenced by the number 1 and also by the label 'columns'.

Default value of `axis` is 0

For most DataFrame methods, the default value of the `axis` parameter is 0. Technically, you will see `None` in the method signature, but if you don't explicitly set it, pandas will use 0. You can also refer to it as the string 'index'. Let's take the mean of each column again, but use the string 'index' for the value of the `axis` parameter. This produces the exact same result as calling it with the defaults.

```
[6]: comp.mean(axis='index')
```

```
[6]: salaries           53715.441133
overtime            4201.272687
other salaries      2816.296542
retirement          10484.755614
health and dental   9382.390735
other benefits       4053.381941
dtype: float64
```

We could have set `axis` to 0, which also returns the same result.

```
[7]: comp.mean(axis=0)
```

```
[7]: salaries           53715.441133
overtime            4201.272687
other salaries      2816.296542
retirement          10484.755614
health and dental   9382.390735
other benefits       4053.381941
dtype: float64
```

Since the default behavior is to act vertically, it's not necessary to specify the `axis` parameter as such, and most people do not do so when calculating aggregations on each column. I recommend calling aggregation methods that act vertically without using the `axis` parameter.

Change the direction of the operation with `axis='columns'`

Let's change the direction of the operation and sum each row by setting the `axis` parameter to the string 'columns'. This gives us the total compensation for each employee.

```
[8]: total_emp_com = comp.sum(axis='columns')
total_emp_com.head(10)
```

```
[8]: 0    104242.87
 1    96627.18
 2    239777.06
 3    46485.63
 4    45680.15
 5    14095.03
 6    86078.82
 7    112333.62
 8    206498.36
 9    66604.32
dtype: float64
```

A Series is returned with the same length as the DataFrame. Let's verify this is the case.

```
[9]: len(comp), len(total_emp_com)
```

```
[9]: (50000, 50000)
```

Instead of using the string 'columns', you can set `axis` to 1 to achieve the same result.

```
[10]: comp.sum(axis=1).head(10)
```

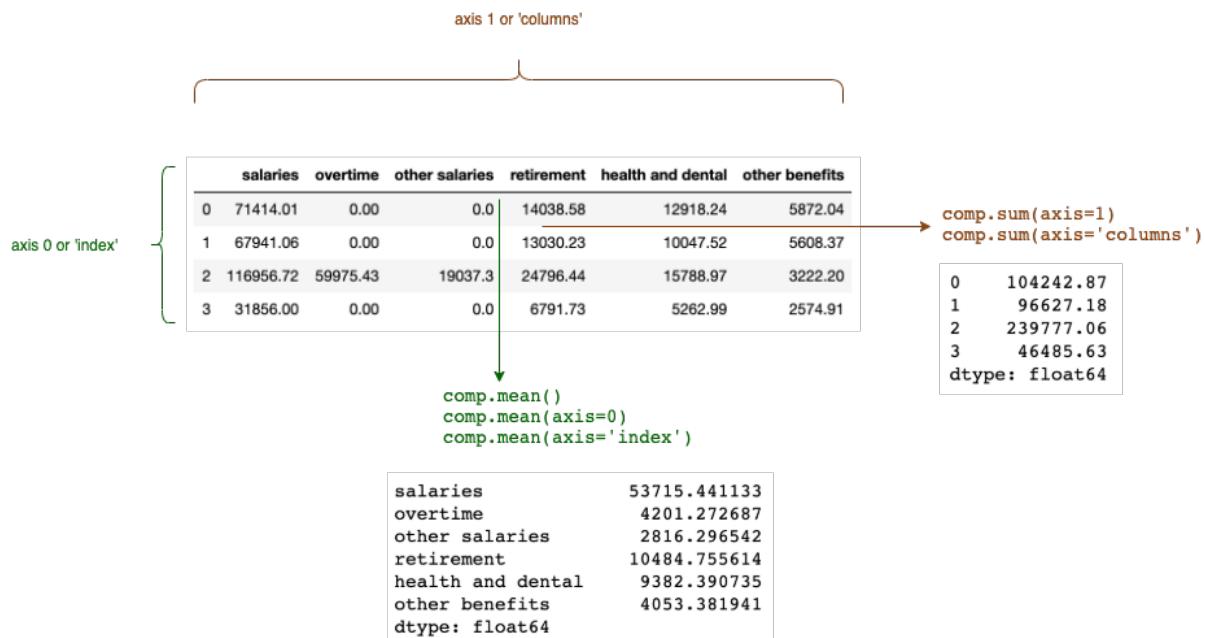
```
[10]: 0    104242.87
      1    96627.18
      2    239777.06
      3    46485.63
      4    45680.15
      5    14095.03
      6    86078.82
      7    112333.62
      8    206498.36
      9    66604.32
      dtype: float64
```

Use either `axis='columns'` or `axis=1`

You are free to use either `axis='columns'` or `axis=1` as they both accomplish the same exact task.

Difficult to remember

It's definitely confusing and difficult to remember which direction the operation is going to happen. As with the examples above, using 'index' or 0 sums up each column while using 'columns' or 1 sums up each row.



A little trick that helps me remember is that when using `axis='columns'` the result is going to be the same length as a column in the DataFrame. Let's verify this below.

Summary of the `axis` parameter

- **axis 0**
 - Default axis for most DataFrame methods
 - Also referenced by the string 'index'
 - Operations happen vertically, up and down the columns
 - Example - `df.sum()` computes the sum of each column individually
- **axis 1**

- Also referenced by the string ‘columns’
- Operations happen horizontally, left to right across each row
- Example - `df.sum(axis='columns')` computes the sum of each row individually

23.3 Non-Aggregation methods

The non-aggregation DataFrame methods do not return a single value for each column, and instead return a DataFrame that usually has the same shape as the original. Here are some common non-aggregation methods.

- `abs` - takes absolute value
- `round` - round to the nearest given decimal place
- `cummin` - cumulative minimum
- `cummax` - cumulative maximum
- `cumsum` - cumulative sum

Let’s use the `round` method to round each column to the nearest thousand. Remember that negative numbers round to the left of the decimal place.

```
[11]: comp.round(-3).head(3)
```

	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	71000.0	0.0	0.0	14000.0	13000.0	6000.0
1	68000.0	0.0	0.0	13000.0	10000.0	6000.0
2	117000.0	60000.0	19000.0	25000.0	16000.0	3000.0

You can use the `round` method on DataFrames that contain non-numeric data. pandas will intelligently ignore the columns where rounding is not possible.

```
[12]: sf_emp.round(-3).head(3)
```

year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits	
0	2000	Public Protection	Personnel Technician	71000.0	0.0	0.0	14000.0	13000.0	6000.0
1	2000	General Administration & Finance	Planner 2	68000.0	0.0	0.0	13000.0	10000.0	6000.0
2	2000	Public Protection	Firefighter	117000.0	60000.0	19000.0	25000.0	16000.0	3000.0

All numeric columns from above were rounded to the nearest thousand including the year. In many cases, you’ll want to round different columns to different decimal places. You can do so by providing the `round` method a dictionary mapping the column name to the decimal place. Below, we round only the salaries and retirement columns.

```
[13]: sf_emp.round({'salaries': -3, 'retirement': -1}).head(3)
```

year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0 2013	Public Protection	Personnel Technician	71000.0	0.00	0.0	14040.0	12918.24	5872.04
1 2013	General Administration & Finance	Planner 2	68000.0	0.00	0.0	13030.0	10047.52	5608.37
2 2013	Public Protection	Firefighter	117000.0	59975.43	19037.3	24800.0	15788.97	3222.20

Some methods don't have an axis parameter

Methods such as `round` work independently of the axis and therefore do not have an `axis` parameter. Other non-aggregation methods such as `cumsum` do have an `axis` parameter. Called with the defaults (`axis=0`), the `cumsum` method computes the cumulative sum of each column individually.

```
[14]: comp.cumsum().head(3)
```

	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	139355.07	0.00	0.0	27068.81	22965.76	11480.41
2	256311.79	59975.43	19037.3	51865.25	38754.73	14702.61

Changing the direction of the operation, the `cumsum` method calculates the cumulative sum of each row individually.

```
[15]: comp.cumsum(axis='columns').head(3)
```

	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	71414.01	71414.01	71414.01	85452.59	98370.83	104242.87
1	67941.06	67941.06	67941.06	80971.29	91018.81	96627.18
2	116956.72	176932.15	195969.45	220765.89	236554.86	239777.06

The values in the last column of the above DataFrame are equal to the sum of the entire row.

```
[16]: comp.sum(axis=1).head(3)
```

```
[16]: 0      104242.87
1      96627.18
2     239777.06
dtype: float64
```

Summary statistics for all columns with the `describe` method

The `describe` method calculates several summary statistics for each column and is a nice way to inspect all of your data at once. Notice that a DataFrame is returned with the name of each summary statistic in the

index. By default, it returns the 25th, 50th, and 75th percentiles. You can customize these by passing in a list of numbers between 0 and 1 to the `percentiles` parameter.

```
[17]: comp.describe(percentiles=[.1, .2, .4, .5, .9, .99])
```

	salaries	overtime	other salaries	retirement	health and dental	other benefits
count	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000
mean	53715.441133	4201.272687	2816.296542	10484.755614	9382.390735	4053.381941
std	47686.502923	11601.573498	6637.820066	9922.598455	7379.199008	4171.974274
min	-2984.520000	-18458.150000	-604.850000	-13692.120000	-287.370000	-8584.140000
10%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
20%	1073.878000	0.000000	0.000000	0.000000	514.270000	96.512000
40%	32216.216000	0.000000	0.000000	5630.182000	7566.940000	1994.890000
50%	52181.955000	0.000000	164.770000	10427.540000	11416.360000	3107.040000
90%	118387.701000	13441.805000	8956.738000	23660.124000	14486.020000	9286.771000
99%	186236.458000	57417.594000	25787.002700	37884.390000	29507.920000	18707.402500
max	645739.460000	258124.170000	239294.570000	120791.400000	36369.960000	37563.460000

The `describe` method with non-numeric columns

The `comp` DataFrame from above contains only numeric columns. If `describe` is called on a DataFrame containing a mix of numeric and non-numeric columns, then summary statistics for just the numeric columns will be returned. The others will be ignored. The original `sf_emp` DataFrame contains a mix of data types. Let's call `describe` on it. Notice how the number of columns after calling `describe` decreased from 9 to 7.

```
[18]: sf_emp.shape
```

```
[18]: (50000, 9)
```

```
[19]: sf_emp.describe()
```

	year	salaries	overtime	other salaries	retirement	health and dental	other benefits
count	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000	50000.000000
mean	2016.531760	53715.441133	4201.272687	2816.296542	10484.755614	9382.390735	4053.381941
std	1.877153	47686.502923	11601.573498	6637.820066	9922.598455	7379.199008	4171.974274
min	2013.000000	-2984.520000	-18458.150000	-604.850000	-13692.120000	-287.370000	-8584.140000
25%	2015.000000	5281.285000	0.000000	0.000000	0.000000	2106.925000	398.465000
50%	2017.000000	52181.955000	0.000000	164.770000	10427.540000	11416.360000	3107.040000
75%	2018.000000	85455.002500	1907.260000	2727.590000	17227.312500	13371.030000	6433.782500
max	2019.000000	645739.460000	258124.170000	239294.570000	120791.400000	36369.960000	37563.460000

```
[20]: sf_emp.describe().shape
```

[20]: (8, 7)

Calling `describe` on non-numeric columns

The `describe` method can work with non-numeric columns, but you'll need to set the `include` parameter to a string of the data type you would like to use. Below, a summary of the object (string) columns is produced. Notice that pandas returns a completely different set of summary statistics that make more sense with strings.

```
[21]: sf_emp.describe(include='object')
```

	organization group	job
count	50000	50000
unique	7	1140
top	Public Works, Transportation & Commerce	Transit Operator
freq	12751	3105

Transposing a DataFrame with the `T` attribute

Transposing a DataFrame ‘rotates’ the data 90 degrees. The columns and the rows switch places. The first column is now the first row. The `.T` attribute transposes the DataFrame. I find this useful after running the `describe` method when there are many columns as it’s easier to read many rows of data as opposed to many columns of data.

```
[22]: sf_emp.describe().T
```

	count	mean	std	min	25%	50%	75%	max
year	50000.0	2016.531760	1.877153	2013.00	2015.000	2017.000	2018.0000	2019.00
salaries	50000.0	53715.441133	47686.502923	-2984.52	5281.285	52181.955	85455.0025	645739.46
overtime	50000.0	4201.272687	11601.573498	18458.15	0.000	0.000	1907.2600	258124.17
other salaries	50000.0	2816.296542	6637.820066	-604.85	0.000	164.770	2727.5900	239294.57
retirement	50000.0	10484.755614	9922.598455	13692.12	0.000	10427.540	17227.3125	120791.40
health and dental	50000.0	9382.390735	7379.199008	-287.37	2106.925	11416.360	13371.0300	36369.96
other benefits	50000.0	4053.381941	4171.974274	-8584.14	398.465	3107.040	6433.7825	37563.46

23.4 Nuisance Columns

Above, we called common statistical methods from the `comp` DataFrame, which was composed of only numeric columns. It is possible to call these same methods from DataFrames composed of any combination of data types.

Dropping columns that don't work with the method

pandas allows you to call these statistical methods on DataFrames containing columns with data types that don't work for that particular method. The entire `sf_emp` DataFrame contains string and numeric columns. Taking the mean of a string column does not work. Instead of raising an error, pandas **silently** drops these column. These DataFrame columns that don't compute with certain methods are sometimes referred to as **nuisance columns**.

Let's show this by calling the `mean` method on the San Francisco employee compensation dataset with all of the original columns. We will work with only 100 rows of the data, which will be explained shortly.

```
[23]: sf_emp_100 = sf_emp.head(100)
sf_emp_100.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

Calling the `mean` method drops the two columns containing strings from the result. No error is raised.

```
[24]: sf_emp_100.mean()
```

```
[24]: year           2013.0000
salaries        66812.8721
overtime        4680.5567
other salaries  3718.1512
retirement      12658.8683
health and dental  9044.6313
other benefits   4585.1201
dtype: float64
```

Many methods do work with non-numeric data types

Many of the aggregation methods do work with string and datetime columns. Let's find the max of all the `sf_emp` columns.

```
[25]: sf_emp_100.max()
```

```
[25]: year                  2013
organization group  Public Works, Transportation & Commerce
```

```
job                           X-Ray Laboratory Aide
salaries                      285446
overtime                     59975.4
other salaries                 24897
retirement                    54710.5
health and dental              15789
other benefits                  17780.9
dtype: object
```

The `sum` method is valid for string (but not datetime) columns and concatenates all the values together to produce one long string. This usually isn't something you'd like to do. It's also a computationally expensive operation. The following call to `sum` took about 4 seconds on the full dataset (50k rows) on my machine.

[26]: `sf_emp_100.sum()`

```
[26]: year                         201300
organization group   Public ProtectionGeneral Administration & Fin...
job                  Personnel TechnicianPlanner 2FirefighterIT Ope...
salaries                      6.68129e+06
overtime                     468056
other salaries                 371815
retirement                    1.26589e+06
health and dental              904463
other benefits                  458512
dtype: object
```

Use `numeric_only=True`

The `sum` method, as well as all the other aggregation methods, provides the boolean parameter `numeric_only` that is defaulted to `False`. By setting it to `True`, pandas will only apply the method to boolean, integer, and float columns. The following operation only took 7 ms on the full dataset on my machine or more than 1,000 times faster than the previous one.

[27]: `sf_emp.sum(numeric_only=True)`

```
[27]: year           1.008266e+08
salaries          2.685772e+09
overtime          2.100636e+08
other salaries    1.408148e+08
retirement        5.242378e+08
health and dental 4.691195e+08
other benefits     2.026691e+08
dtype: float64
```

The slow `mean` method

The `mean` method is also extremely slow, even though it only works on numeric columns. This is because pandas takes the `sum` of all the columns first and then divides by the length. The reason pandas doesn't just skip over string columns is that they are technically object columns and an object column can hold any data type. The only way for pandas to decide whether or not the `mean` will work on an object column is to actually sum up every value first and then attempt to divide by the length. If that fails, then it will skip it. The issue with this, is that it is extremely slow for string columns since strings can be summed. pandas only

fails after the string column has been concatenated together when it attempts to divide by the length. If you want to take the `mean` on a DataFrame with string columns, make sure you set `numeric_only` to `True`.

Even on this small dataset of 100 rows, there is a substantial performance difference.

```
[28]: %timeit -n 5 sf_emp_100.mean()
```

5.8 ms ± 939 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

```
[29]: %timeit -n 5 sf_emp_100.mean(numeric_only=True)
```

1.83 ms ± 268 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

23.5 Exercises

Execute the following cell to read in the movie dataset with the title in the index selecting all three actor Facebook like columns.

```
[30]: movie = pd.read_csv('../data/movie.csv', index_col='title')
cols = ['actor1_fb', 'actor2_fb', 'actor3_fb']
actor_fb = movie[cols]
actor_fb.head(3)
```

	actor1_fb	actor2_fb	actor3_fb
title			
Avatar	1000.0	936.0	855.0
Pirates of the Caribbean: At World's End	40000.0	5000.0	1000.0
Spectre	11000.0	393.0	161.0

Exercise 1

Calculate the mean of each actor Facebook like column. Which actor (1, 2, or 3) has the highest mean?

```
[ ]:
```

Exercise 2

The result of exercise 1 is a Series of three values. Can you call a method on this Series to choose the column name with the highest mean Facebook likes.

```
[ ]:
```

Exercise 3

Calculate the total Facebook likes of all three actors for each movie

```
[ ]:
```

Exercise 4

What percentage of movies have more than 10,000 total actor FB likes?

[]:

Exercise 5

Find the median gross revenue in millions of dollars for the movies that have more than 10,000 total actor FB likes. Do the same for movies with 10,000 or less total actor FB likes.

[]:

Exercise 6

From exercise 5, it appears that movies with more than 10,000 total actor FB likes gross 2.5 times as much. This may be due to the fact that newer movies have more actors that are recognized by FB users. Find the median year produced for both groups.

[]:

Exercise 7

For each movie made in the year 2016, what is the median of the total actor FB likes?

[]:

Exercise 8

Write a function that has a single parameter, `year`. Have it return the median of the total actor FB likes for the given year. Test your function with the year 2016 and verify the result with Exercise 6.

[]:

Exercise 9

Write a loop to print out the year and median total actor FB likes for that year from 1990 to 2016

[]:

Use the college dataset with the institution name as the index for the remaining exercises.

```
[31]: college = pd.read_csv('../data/college.csv', index_col='instnm')
college.head(3)
```

instnm	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
Alabama A & M University	Normal	AL	1.0	0.0	0.0	...	0.7356	0.8284	0.1049	30300	33888
University of Alabama at Birmingham	Birmingham	AL	0.0	0.0	0.0	...	0.3460	0.5214	0.2422	39700	21941.5
Amridge University	Montgomery	AL	0.0	0.0	0.0	...	0.6801	0.7795	0.8540	40100	23370

Exercise 10

Find the number of non-missing values in each column and again for each row.

[]:

Exercise 11

What is the average number of non-missing values for each row?

[]:

Exercise 12

The ugds column of the college dataset contains the total undergraduate population. What is the least number of colleges it would take to have have a total of more than 5 million students?

[]:

Exercise 13

Call the `describe` method, but make it work only for the string columns.

[]:

Exercise 14

Call the `max` method, but only return columns that are numeric.

[]:

Chapter 24

DataFrame Missing Value Methods

This chapter covers the methods available to a DataFrame to handle missing values. They are the exact same methods that were presented for the Series in the last part. These methods behave analogously as they do with Series, but are more complex as they deal with missing values in all of the columns in the DataFrame and have more options for their usage. We begin by reading in the movie dataset setting the title as the index.

```
[1]: import pandas as pd
pd.set_option('display.max_columns', 40)
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	2009.0	Color	PG-13	178.0	James Cameron	...	avatar future marine native paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	...	goddess marriage ceremony marriage proposal pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	...	bomb espionage sequel spy terrorist	English	UK	245000000.0	6.8

24.1 Methods for handling missing values

pandas provides the following DataFrame methods to handle missing values:

- `isna` - Returns a DataFrame of booleans based on whether each value is missing or not
- `notna` - Exact opposite of `isna`
- `fillna` - Fills missing values in a variety of ways
- `dropna` - Drops the missing values from the Series
- `interpolate` - Fills missing values with statistical interpolation

24.2 The `isna` method

The `isna` method returns a DataFrame of all boolean values. All missing values evaluate as `True` with everything else evaluating as `False`. Let's call this method on our `movie` DataFrame.

```
[2]: movie.isna().head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title						...					
Avatar	False	False		False	False	False	...	False	False	False	False
Pirates of the Caribbean: At World's End	False	False		False	False	False	...	False	False	False	False
Spectre	False	False		False	False	False	...	False	False	False	False

Finding the number of missing values in each column

There is no single direct method for finding the number of missing values in each column of a DataFrame. However, the `count` method returns the number of non-missing values of each column. Let's see an example of that here.

```
[3]: movie.count().head()
```

```
[3]: year      4810
color     4897
content_rating 4616
duration    4901
director_name 4814
dtype: int64
```

To find the number of missing values, we will need to first call the `isna` method to turn each value into a boolean, and then chain the `sum` method to count the number of missing values in each column.

```
[4]: movie.isna().sum().head()
```

```
[4]: year      106
color     19
content_rating 300
duration    15
director_name 102
dtype: int64
```

Find the percentage of missing values by calling the `mean` method

As we have seen before, taking the mean of a boolean Series returns the percentage of values that are `True`, and in this case, returns the percentage of missing values for each column.

```
[5]: movie.isna().mean().head()
```

```
[5]: year      0.021562
color     0.003865
content_rating 0.061025
duration    0.003051
director_name 0.020749
dtype: float64
```

Nicer visuals with rounding

Although the above result is accurate, it isn't that easy to read. The last few decimal places provide little information and are not necessary to report. The result isn't technically a percentage either, but a fraction. Let's round the values and multiply by 100 to get a nicer visual report that is a percentage.

```
[6]: movie.isna().mean().round(3).head() * 100
```

```
[6]: year           2.2
      color          0.4
      content_rating 6.1
      duration        0.3
      director_name   2.1
      dtype: float64
```

The `notna` method

The `notna` method is the exact opposite of `isna` and evaluates as `True` for all non-missing values.

```
[7]: movie.notna().head(3)
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Avatar	True	True		True	True		True	...	True	True	True
Pirates of the Caribbean: At World's End	True	True		True	True		True	...	True	True	True
Spectre	True	True		True	True		True	...	True	True	True

24.3 Dropping rows and columns with the `dropna` method

The `dropna` method has more options than its Series counterpart. It allows you to drop either rows or columns of your DataFrame in a variety of ways. Called with the defaults, it drops all `rows` that have at least 1 missing value. Over 1,000 rows get dropped after calling the `dropna` method on our `movie` DataFrame.

```
[8]: movie.dropna().shape
```

```
[8]: (3707, 21)
```

Drop rows where only a particular column is missing

By default, the `dropna` method drops any rows where there are one or missing values for that entire row. pandas gives us the option of only dropping rows where a particular column or group of columns have missing values. Instead of looking at all of the columns for missing values, we can restrict the columns with the `subset` parameter. Set this parameter to a list of the column names to inspect for missing values. Below, we only drop rows that are missing either the `year` or `content_rating`.

```
[9]: movie.dropna(subset=['year', 'content_rating']).shape
```

```
[9]: (4552, 21)
```

Another option for `dropna` is setting the minimum threshold for non-missing values in each row. Our current DataFrame has 21 columns. Below, we use the `thresh` parameter to ensure that each row has at least 18 non-missing values.

```
[10]: movie.dropna(thresh=18).shape
```

```
[10]: (4767, 21)
```

Drop columns with missing values

Just like we did with the statistical methods, we can change the direction of the operation for `dropna`, and drop columns instead of rows by setting the `axis` parameter to the string ‘columns’ or the integer 1. Let’s drop all columns that have 1 or more missing values.

```
[11]: movie.dropna(axis=1).head()
```

		genres	num_voted_users	imdb_score
	title			
	Avatar	Action Adventure Fantasy Sci-Fi	886204	7.9
	Pirates of the Caribbean: At World's End	Action Adventure Fantasy	471220	7.1
	Spectre	Action Adventure Thriller	275868	6.8
	The Dark Knight Rises	Action Thriller	1144337	8.5
	Star Wars: Episode VII - The Force Awakens	Documentary	8	7.1

The `subset` parameter is available when dropping columns and can be given a list of index labels to limit the rows to look for missing values. The `thresh` parameter can be set as an integer to represent the minimum number of non-missing values in a particular column in order to be kept in the result. Here we drop all of the columns that don’t have at least 4,800 non-missing values.

```
[12]: movie.dropna(axis=1, thresh=4800).shape
```

```
[12]: (4916, 17)
```

24.4 Filling missing values with the `fillna` method

The `fillna` method fills the missing values in your DataFrame in a couple of different ways.

Filling the missing values with a given constant

The most basic way to use the `fillna` method is to pass it a single value which replaces every missing value with this constant. The following replaces all missing values with the string ‘FILLED’. Note, that this replaces all missing value representations - `NaN`, `NaT`, `NA`, and `None`. We only output the fifth row as it was the only one of the first five to contain missing values.

```
[13]: movie.fillna('FILLED').iloc[4:5, :]
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Star Wars: Episode VII - The Force Awakens	FILLED	FILLED	FILLED	FILLED	Doug Walker	...	FILLED	FILLED	FILLED	FILLED	7.1

Filling a DataFrame's missing values with a single constant value may not be what you'd need in a real situation. The `movie` DataFrame has columns of different data types and by using the string '`FILLED`' we will have changed the data type of any numeric column to object. For instance, the column `duration` was originally a float and is now an object.

Use a dictionary to fill specific columns

A more practical application would be to fill each column with a different constant value. We can use `fillna` to do this by passing it a dictionary that maps the column name to the missing value replacement. The following fills the `content_rating` column with 'PG' and the `duration` column with 199. Missing values in other columns are not filled.

```
[14]: movie.fillna({'content_rating': 'PG', 'duration': 199}).iloc[4:5]
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Star Wars: Episode VII - The Force Awakens	NAN	NAN	PG	199.0	Doug Walker	...	NAN	NAN	NAN	NAN	7.1

Fill all columns with the mean or median

A somewhat common approach to filling missing values is to use the mean or median of the column as the replacement. Taking the `median` of a DataFrame returns a Series that has each column name labeling its median. A Series is also a valid object that can be passed to the `fillna` method. Let's begin by finding the median of each column.

```
[15]: mm = movie.median(numeric_only=True)
mm
```

```
[15]: year          2005.0
duration      103.0
director_fb    48.0
actor1_fb      982.0
actor2_fb      593.0
actor3_fb      366.0
gross         25043962.0
num_reviews    108.0
num_voted_users 33132.5
budget        19850000.0
```

```
imdb_score          6.6
dtype: float64
```

Pass the above Series to `fillna` to use a different missing value for each column. The string columns will not be filled as they do not have a median.

```
[16]: movie.fillna(mm).iloc[4:5]
```

	year	color	content_rating	duration	director_name	...	plot_keywords	language	country	budget	imdb_score
title											
Star Wars: Episode VII - The Force Awakens	2005.0	NaN	NaN	103.0	Doug Walker	...	NaN	NaN	NaN	19850000.0	7.1

Filling missing values with the mean isn't necessarily a good strategy when doing data analysis. The example above is merely used to demonstrate how the `fillna` method works.

Filling missing values with the preceding or following values

Instead of filling missing values with a constant, you can fill in missing values with the immediately preceding or following known value. Let's use a simple csv to clearly see how this done.

```
[17]: df = pd.read_csv('~/data/missing_example.csv')
df
```

	day	orders	sales
0	Monday	NaN	21.43
1	NaN	5.0	NaN
2	NaN	10.0	NaN
3	Tuesday	NaN	NaN
4	NaN	3.0	NaN

The `method` parameter controls whether you are going to fill in the missing values with the immediate preceding or following value. Set it equal to the string '`ffill`' to fill missing values going 'forward', i.e. using the immediate preceding value.

```
[18]: df.fillna(method='ffill')
```

	day	orders	sales
0	Monday	NaN	21.43
1	Monday	5.0	21.43
2	Monday	10.0	21.43
3	Tuesday	10.0	21.43
4	Tuesday	3.0	21.43

Notice that the first value for the `orders` column is still missing as there is no immediate preceding value to fill it with. Use the string ‘`bfill`’ to fill missing values going backwards.

```
[19]: df.fillna(method='bfill')
```

	day	orders	sales
0	Monday	5.0	21.43
1	Tuesday	5.0	NaN
2	Tuesday	10.0	NaN
3	Tuesday	3.0	NaN
4	NaN	3.0	NaN

You can limit the number of consecutive values filled with the `limit` parameter. Here we limit the number filled to only 1.

```
[20]: df.fillna(method='ffill', limit=1)
```

	day	orders	sales
0	Monday	NaN	21.43
1	Monday	5.0	21.43
2	NaN	10.0	NaN
3	Tuesday	10.0	NaN
4	Tuesday	3.0	NaN

When you use `fillna`, you must choose between filling with a constant, demonstrated by the first example, or by forward or backfilling with the `method` parameter, shown in the second example.

24.5 The `interpolate` method

The `interpolate` method was covered extensively in the Series part. It works the same with DataFrames and fills in missing values using a variety of statistical methods. By default, it uses linear interpolation on every column in the DataFrame. Let’s read a small sample of stock price data for Microsoft, Apple, and Schlumberger with several missing values.

```
[21]: df_stocks = pd.read_csv('../data/stocks/sample_missing.csv')
df_stocks
```

	date	MSFT	AAPL	SLB
0	2019-10-08	135.67	224.40	30.71
1	2019-10-09	NaN	NaN	NaN
2	2019-10-10	139.10	230.09	31.07
3	2019-10-11	NaN	NaN	NaN
4	2019-10-14	NaN	NaN	NaN
5	2019-10-15	141.57	235.32	32.81
6	2019-10-16	NaN	NaN	NaN
7	2019-10-17	NaN	NaN	NaN
8	2019-10-18	137.41	236.41	32.31
9	2019-10-21	NaN	NaN	NaN

Let's use linear interpolation to fill in the missing values of each column.

```
[22]: df_stocks.interpolate('linear')
```

	date	MSFT	AAPL	SLB
0	2019-10-08	135.670000	224.400000	30.710000
1	2019-10-09	137.385000	227.245000	30.890000
2	2019-10-10	139.100000	230.090000	31.070000
3	2019-10-11	139.923333	231.833333	31.650000
4	2019-10-14	140.746667	233.576667	32.230000
5	2019-10-15	141.570000	235.320000	32.810000
6	2019-10-16	140.183333	235.683333	32.643333
7	2019-10-17	138.796667	236.046667	32.476667
8	2019-10-18	137.410000	236.410000	32.310000
9	2019-10-21	137.410000	236.410000	32.310000

Here, we use quadratic interpolation and limit

```
[23]: df_stocks.interpolate('quadratic', limit=1)
```

	date	MSFT	AAPL	SLB
0	2019-10-08	135.670000	224.400000	30.710000
1	2019-10-09	137.524023	227.463333	30.767701
2	2019-10-10	139.100000	230.090000	31.070000
3	2019-10-11	140.397931	232.280000	31.616897
4	2019-10-14	NaN	NaN	NaN
5	2019-10-15	141.570000	235.320000	32.810000
6	2019-10-16	140.985517	236.146667	32.962720
7	2019-10-17	NaN	NaN	NaN
8	2019-10-18	137.410000	236.410000	32.310000
9	2019-10-21	NaN	NaN	NaN

24.6 Exercises

Execute the next cell to read in the college dataset with the institution name as the index and use it for the first few exercises.

```
[24]: college = pd.read_csv('../data/college.csv', index_col='instnm')
college.head(3)
```

instnm	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
Alabama A & M University	Normal	AL	1.0	0.0	0.0	...	0.7356	0.8284	0.1049	30300	33888
University of Alabama at Birmingham	Birmingham	AL	0.0	0.0	0.0	...	0.3460	0.5214	0.2422	39700	21941.5
Amridge University	Montgomery	AL	0.0	0.0	0.0	...	0.6801	0.7795	0.8540	40100	23370

The data dictionary is helpful to decipher the meaning of the columns. Uncomment the next line to read it in.

```
[25]: # pd.read_csv('../data/college_data_dictionary.csv')
```

Exercise 1

Find the number of missing values for each row.

```
[ ]:
```

Exercise 2

What percentage of rows have more than 5 missing values?

[]:

Exercise 3

How many total missing values are there in the entire DataFrame?

[]:

Exercise 4

How many total non-missing values are there in the entire DataFrame?

[]:

Exercise 5

How many rows will be dropped when the `dropna` method is called with its defaults. Calculate this number without calling the `dropna` method.

[]:

Exercise 6

Verify the result from exercise 5 by calling the `dropna` method.

[]:

Exercise 7

Drop all the rows that are missing the `ugds` column.

[]:

Exercise 8

Drop all columns that have more than 5% of their values missing.

[]:

Exercise 9

Fill in the missing values with the maximum value of each column.

[]:

Chapter 25

DataFrame Sorting, Ranking, and Uniqueness

In this chapter, we cover methods that help with sorting, ranking, and uniqueness on the entire DataFrame. All the methods presented in this chapter appeared in the same chapter for Series with the exception of `unique` which is, ironically, unique to just the Series.

25.1 Sorting

pandas allows you to sort the rows of a DataFrame either by the values or by the index with the `sort_values` and `sort_index` methods.

The `sort_values` method

The `sort_values` DataFrame method sorts the DataFrame by the values of one or more columns. Pass the `by` parameter a column name or list of column names to sort. By default, the sorting takes place in ascending manner (from least to greatest). The college dataset is used for the remainder of the examples in this chapter.

```
[1]: import pandas as pd
college = pd.read_csv('../data/college.csv', index_col='instnm')
college.head(3)
```

instnm	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
Alabama A & M University		Normal	AL	1.0	0.0	0.0	...	0.7356	0.8284	0.1049	30300
University of Alabama at Birmingham	Birmingham	AL	0.0	0.0	0.0	...	0.3460	0.5214	0.2422	39700	21941.5
Amridge University	Montgomery	AL	0.0	0.0	0.0	...	0.6801	0.7795	0.8540	40100	23370

Let's sort the rows by city.

```
[2]: college.sort_values(by='city').head(3)
```

	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
instnm											
Northern State University	Aberdeen	SD	0.0	0.0	0.0	...	0.2272	0.4303	0.1766	33600	24847
Grays Harbor College	Aberdeen	WA	0.0	0.0	0.0	...	0.4530	0.1502	0.5087	27000	11490
Presentation College	Aberdeen	SD	0.0	0.0	0.0	...	0.4829	0.7560	0.3097	35900	25000

Sort from greatest to least

Set the `ascending` parameter to `False` to sort in the opposite direction. Since the `by` parameter is first, it is usually omitted.

```
[3]: college.sort_values('city', ascending=False).head(3)
```

	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
instnm											
Ohio University-Zanesville Campus	Zanesville	OH	0.0	0.0	0.0	...	0.3947	0.5245	0.3560	39600	21250
Mid-East CTC-Adult Education	Zanesville	OH	0.0	0.0	0.0	...	0.3712	0.4991	0.4961	29800	6943
Zane State College	Zanesville	OH	0.0	0.0	0.0	...	0.3645	0.3434	0.3185	23800	13960.5

Simultaneously sort two or more columns

Sort by any number of columns by passing a list of their names to the `by` parameter. The sort begins with the first column. For instance, the following sorts all the colleges by state, and then within each state, sorts by undergraduate population.

```
[4]: cols = ['stabbr', 'ugds']
state_ugds_sort = college.sort_values(cols)
state_ugds_sort[cols].head(3)
```

	stabbr	ugds
instnm		
Alaska Bible College	AK	27.0
Alaska Christian College	AK	68.0
Ilisagvik College	AK	109.0

Let's select just the state of Oklahoma to verify that it too is sorted by undergraduate population.

```
[5]: state_ugds_sort.query('stabbr == "OK"')[cols].head(3)
```

	stabbr	ugds
instnm		
Hollywood Cosmetology Center	OK	6.0
Claremore Beauty College	OK	15.0
Northwest Technology Center-Fairview	OK	15.0

Sort multiple columns in different directions

The `ascending` parameter may be passed a list of booleans corresponding to the list of column names in the `by` parameter. The following sorts by state from least to greatest, and then by undergraduate population from greatest to least.

```
[6]: college[cols].sort_values(cols, ascending=[True, False]).head(3)
```

	stabbr	ugds
instnm		
University of Alaska Anchorage	AK	12865.0
University of Alaska Fairbanks	AK	5536.0
Charter College-Anchorage	AK	3256.0

Sort by the index

The DataFrame may be sorted by its index with the `sort_index` method.

```
[7]: college.sort_index(ascending=False).head(3)
```

	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
instnm											
eClips School of Cosmetology and Barbering	Cape Girardeau	MO	0.0	0.0	0.0	...	0.9496	0.9832	0.3772	PrivacySuppressed	9500
duCret School of Arts	Plainfield	NJ	0.0	0.0	0.0	...	0.4375	0.5000	0.1250	PrivacySuppressed	PrivacySuppressed
Zane State College	Zanesville	OH	0.0	0.0	0.0	...	0.3645	0.3434	0.3185	23800	13960.5

Sort the columns

Interestingly, you can use the same `sort_index` method to sort the columns of the DataFrame. You must remember that pandas uses an `Index` object to contain the columns, which is the same object that contains the index. To sort the columns, set the `axis` parameter to ‘columns’ or 1. This is identical to how we changed the direction of the operation of the statistical methods in previous chapters. Perhaps this method would have been more appropriately named `sort_axis` instead since it sorts either axis.

```
[8]: college.sort_index(axis=1).head(2)
```

	city	curroper	distanceonly	grad_debt_mdn_supp	hbcu	...	ugds_nhpi	ugds_nra	ugds_unkn	ugds_white	womenonly
instnm											
Alabama A & M University	Normal	1	0.0	33888	1.0	...	0.0019	0.0059	0.0138	0.0333	0.0
University of Alabama at Birmingham	Birmingham	1	0.0	21941.5	0.0	...	0.0007	0.0179	0.0100	0.5922	0.0

25.2 Ranking

The `rank` method ranks the values in each column, independently, the same way it does for a Series. Let's explore ranking on a subset of the columns from the movie dataset. To simplify matters, we will work with just the first 5 rows of data.

```
[9]: movie = pd.read_csv('../data/movie.csv', index_col='title')
cols = ['year', 'director_name', 'duration', 'director_fb', 'actor1_fb',
        'actor2_fb', 'actor3_fb', 'num_reviews', 'imdb_score']
movie_5 = movie[cols].head()
movie_5
```

	year	director_name	duration	director_fb	actor1_fb	actor2_fb	actor3_fb	num_reviews	imdb_score
title									
Avatar	2009.0	James Cameron	178.0	0.0	1000.0	936.0	855.0	723.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Gore Verbinski	169.0	563.0	40000.0	5000.0	1000.0	302.0	7.1
Spectre	2015.0	Sam Mendes	148.0	0.0	11000.0	393.0	161.0	602.0	6.8
The Dark Knight Rises	2012.0	Christopher Nolan	164.0	22000.0	27000.0	23000.0	23000.0	813.0	8.5
Star Wars: Episode VII - The Force Awakens	NaN	Doug Walker	NaN	131.0	131.0	12.0	NaN	NaN	7.1

Calling the `rank` method with the defaults ranks each value in ascending order beginning with 1. For instance, take a look at the `actor1_fb` column. The smallest value is the fifth value and is given rank 1 below. The `rank` method also ranks string columns.

```
[10]: movie_5.rank()
```

	year	director_name	duration	director_fb	actor1_fb	actor2_fb	actor3_fb	num_reviews	imdb_score
title									
Avatar	2.0		4.0	4.0	1.5	2.0	3.0	2.0	3.0
Pirates of the Caribbean: At World's End	1.0		3.0	3.0	4.0	5.0	4.0	3.0	2.5
Spectre	4.0		5.0	1.0	1.5	3.0	2.0	1.0	2.0
The Dark Knight Rises	3.0		1.0	2.0	5.0	4.0	5.0	4.0	5.0
Star Wars: Episode VII - The Force Awakens	Nan		2.0	Nan	3.0	1.0	1.0	Nan	2.5

Here, we rank from greatest to least using the minimum rank for ties which take place in the `director_fb` and `imdb_score` columns.

```
[11]: movie_5.rank(ascending=False, method='min')
```

	year	director_name	duration	director_fb	actor1_fb	actor2_fb	actor3_fb	num_reviews	imdb_score
title									
Avatar	3.0		2.0	1.0	4.0	4.0	3.0	3.0	2.0
Pirates of the Caribbean: At World's End	4.0		3.0	2.0	2.0	1.0	2.0	2.0	3.0
Spectre	1.0		1.0	4.0	4.0	3.0	4.0	4.0	5.0
The Dark Knight Rises	2.0		5.0	3.0	1.0	2.0	1.0	1.0	1.0
Star Wars: Episode VII - The Force Awakens	Nan		4.0	Nan	3.0	5.0	5.0	Nan	3.0

25.3 Uniqueness

In this section we cover the `nunique` and `drop_duplicates` DataFrame methods which exist for Series. Interestingly, there is no `unique` method for DataFrames. We'll use the City of Houston employee dataset for the next set of examples.

```
[12]: emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

The `nunique` method

The `nunique` method returns the number of unique values in each column as a Series. The old column names are now the new index values.

```
[13]: emp.nunique()
```

```
[13]: dept      9
       title     693
      hire_date  3955
      salary     4102
        sex      2
      race      5
dtype: int64
```

To count any missing values as exactly one more unique set the `dropna` parameter to `False`.

```
[14]: emp.nunique(dropna=False)
```

```
[14]: dept      9
       title     693
      hire_date  3955
      salary     4103
        sex      2
      race      6
dtype: int64
```

The `drop_duplicates` method

The default call to the `drop_duplicates` method returns only unique rows of the DataFrame. It does not use the index value in its search for duplicates. If two or more rows are duplicated, the first row is kept. Let's see if there are any duplicate rows in the employee dataset.

```
[15]: emp.shape
```

```
[15]: (24308, 6)
```

```
[16]: emp.drop_duplicates().shape
```

```
[16]: (17034, 6)
```

Interestingly, there are some rows with the exact same information for all six columns.

Drop duplicates based on a subset of columns

Instead of dropping rows where the entire row is duplicated, you can restrict the search for duplication to a subset of the columns. Pass the `subset` parameter a single column name or a list of column names. The following example returns a single row for each unique department.

```
[17]: emp.drop_duplicates(subset='dept')
```

	dept		title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White	
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic	
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black	
8	Fire	FIRE FIGHTER	2015-08-01	48189.70	Male	White	
10	Health & Human Services	ADMINISTRATIVE SPECIALIST	2016-04-25	52451.00	Female	Black	
12	Library	CUSTOMER SERVICE CLERK	2017-11-13	28787.00	Female	Hispanic	
20	Houston Airport System	SEMI-SKILLED LABORER	2009-07-06	35859.00	Female	Hispanic	
23	Solid Waste Management	SENIOR REFUSE TRUCK DRIVER	2016-04-25	35422.00	Male	Black	
35	Parks & Recreation	RECREATION ASSISTANT	2015-06-17	29058.00	Male	Black	

The following returns the first row for each unique combination of race and sex.

```
[18]: emp.drop_duplicates(subset=['race', 'sex']).head(4)
```

	dept		title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White	
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic	
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black	
5	Other	SENIOR ACCOUNT CLERK	2017-10-09	44616.00	Female	Black	

25.4 Finding the maximum/minimum of a group

In this section, we will explore a practical example of the `drop_duplicates` method. Let's say we are interested in finding each employee with the maximum salary per department. This results in a DataFrame with a single row for each department. Let's begin by sorting by department first and then salary, making sure salary is sorted from greatest to least.

```
[19]: emp_sorted = emp.sort_values(['dept', 'salary'], ascending=[True, False])
emp_sorted.head(3)
```

	dept	title	hire_date	salary	sex	race
1732	Fire	PHYSICIAN,MD	2014-09-27	342784.0	Male	White
1975	Fire	PHYSICIAN,MD	2014-09-27	342784.0	Male	Asian
4680	Fire	PHYSICIAN,MD	2015-11-23	342784.0	Male	White

The data is correctly sorted, but the information we want is not easily accessible. We desire a single row for each department. We can now turn to the `drop_duplicates` method and use the `subset` parameter to keep the first row of every department.

```
[20]: emp_sorted.drop_duplicates(subset='dept')
```

	dept	title	hire_date	salary	sex	race
1732	Fire	PHYSICIAN,MD	2014-09-27	342784.0	Male	White
8405	Health & Human Services	CHIEF PHYSICIAN,MD	2017-07-31	186685.0	Female	White
3897	Houston Airport System	AVIATION DIRECTOR	2010-06-01	275000.0	Male	Hispanic
10704	Houston Public Works	PUBLIC WORKS DIRECTOR	2005-08-10	275000.0	Female	White
7564	Library	LIBRARY DIRECTOR	2005-11-07	170000.0	Female	Black
13338	Other	CITY ATTORNEY	2016-05-02	275000.0	Male	Black
11679	Parks & Recreation	PARKS & RECREATION DIRECTOR	2017-07-05	150000.0	Male	White
4413	Police	POLICE CHIEF	2016-11-30	280000.0	Male	Hispanic
20244	Solid Waste Management	SOLID WASTE DIRECTOR	2001-05-14	195000.0	Male	Black

We can rewrite our solution without assigning the result of `sort_values` to a variable by chaining the `drop_duplicates` method directly after it. Here, we find the employee with the lowest salary per department.

```
[21]: emp.sort_values(['dept', 'salary']) \
    .drop_duplicates(subset='dept')
```

	dept	title	hire_date	salary	sex	race
7000	Fire	ASSISTANT EMS PHYSICIAN DIRECTOR	2017-07-10	16411.0	Female	Black
16485	Health & Human Services	CUSTOMER SERVICE CLERK	2017-11-20	25064.0	Male	Black
668	Houston Airport System	LABORER	2017-09-11	26125.0	Female	Hispanic
10547	Houston Public Works	LABORER	2014-03-17	28205.0	Male	Black
1183	Library	CUSTOMER SERVICE CLERK	2016-01-19	9912.0	Female	Hispanic
4040	Other	COUNCIL INTERN (EXECUTIVE LEVEL)	2018-08-14	24960.0	Female	Hispanic
64	Parks & Recreation	LABORER	2018-04-23	24960.0	Male	Hispanic
2935	Police	CUSTOMER SERVICE CLERK	2016-10-02	26915.0	Female	Hispanic
3135	Solid Waste Management	LABORER	2018-08-27	27040.0	Female	Black

It's actually sufficient to sort just by salary as the first value encountered for each department will be the employee with the highest salary. Note, how the final data will be sorted by salary and not by department.

```
[22]: emp.sort_values('salary', ascending=False) \
    .drop_duplicates(subset='dept')
```

	dept	title	hire_date	salary	sex	race
6909	Fire	PHYSICIAN,MD	2014-09-27	342784.0	Male	Black
4413	Police	POLICE CHIEF	2016-11-30	280000.0	Male	Hispanic
10704	Houston Public Works	PUBLIC WORKS DIRECTOR	2005-08-10	275000.0	Female	White
13338	Other	CITY ATTORNEY	2016-05-02	275000.0	Male	Black
3897	Houston Airport System	AVIATION DIRECTOR	2010-06-01	275000.0	Male	Hispanic
20244	Solid Waste Management	SOLID WASTE DIRECTOR	2001-05-14	195000.0	Male	Black
8405	Health & Human Services	CHIEF PHYSICIAN,MD	2017-07-31	186685.0	Female	White
7564	Library	LIBRARY DIRECTOR	2005-11-07	170000.0	Female	Black
11679	Parks & Recreation	PARKS & RECREATION DIRECTOR	2017-07-05	150000.0	Male	White

This short chain of steps combining `sort_values` with `drop_duplicates` is a generic and common pattern for finding the maximum or minimum of some column within groups formed by other columns. Below, we find the minimum salary for every unique combination of department, race, and sex.

```
[23]: emp.sort_values(['salary']) \
    .drop_duplicates(subset=['dept', 'race', 'sex']).head()
```

	dept		title	hire_date	salary	sex	race
13705	Library	CUSTOMER SERVICE CLERK		2016-01-19	9912.0	Female	White
13305	Library	CUSTOMER SERVICE CLERK		2016-02-08	9912.0	Female	Hispanic
9838	Library	CUSTOMER SERVICE CLERK		2016-02-29	9912.0	Female	Black
23952	Library	SENIOR CUSTOMER SERVICE CLERK		2017-08-28	10661.0	Male	White
4164	Library	INVENTORY MANAGEMENT CLERK		2017-02-27	10952.0	Male	Black

25.5 Exercises

Use the employee dataset for the first few exercises.

Exercise 1

Sort department, race, sex ascending along with salary descending.

[]:

Exercise 2

How many unique combinations of department and title exist?

[]:

Exercise 3

Since only Series methods have a `unique` method, can you think of a creative way of getting the same result as exercise 2 with the `unique` method?

[]:

Exercise 4

Find the frequency of occurrence of all race and sex combinations using the trick from exercise 3. For instance, you would return an object that contains the number of ‘Hispanic Males’, ‘Black Females’, etc...

[]:

Use the college dataset for the remaining exercises

Execute the following cell to read in the college dataset which sets the institution name (`instnm`) as the index.

```
[24]: college = pd.read_csv('../data/college.csv', index_col='instnm')
college.head(3)
```

instnm	city	stabbr	hbcu	menonly	womenonly	...	pctpell	pctfloan	ug25abv	md_earn_wne_p10	grad_debt_mdn_supp
Alabama A & M University	Normal	AL	1.0	0.0	0.0	...	0.7356	0.8284	0.1049	30300	33888
University of Alabama at Birmingham	Birmingham	AL	0.0	0.0	0.0	...	0.3460	0.5214	0.2422	39700	21941.5
Amridge University	Montgomery	AL	0.0	0.0	0.0	...	0.6801	0.7795	0.8540	40100	23370

Exercise 5

Select the columns `stabbr`, `satvrmid`, `satmtmid` and `ugds` columns for the state of Texas ('TX') that have an undergraduate student population of more than 20,000. Drop any rows with missing values and assign the result to the variable name `college_tx`.

```
[ ]:
```

Exercise 6

Rank each column from the `college_tx` DataFrame from greatest to least.

```
[ ]:
```

Exercise 7

Using the full college dataset, find the largest school by population for each state. Return only the `stabbr` and `ugds` columns sorting by `ugds`.

```
[ ]:
```

Exercise 8

Several of the columns from the college dataset contain binary data (are either 0 or 1). Can you identify the names of these columns?

```
[ ]:
```


Chapter 26

DataFrame Structure Methods

In this chapter, we cover several different methods that change the structure of the DataFrame. We will be adding and dropping rows and columns from our DataFrame and renaming the labels for both the rows and columns.

26.1 Adding a new column to the DataFrame

A new column may be added to a DataFrame using similar syntax as selecting a single column with *just the brackets*. This is done without the use of a method. The general syntax will look like the following:

```
>>> df['new_column'] = <some expression>
```

Let's begin by reading in the college dataset with the institution name set as the index. We'll use a small subset of this DataFrame consisting of the first three rows and six of the columns.

```
[1]: import pandas as pd
college = pd.read_csv('../data/college.csv', index_col='instnm')
cols = ['city', 'stabbr', 'relaffil', 'satvrmid', 'satmtmid', 'ugds']
cs = college[cols].head(3)
cs
```

		city	stabbr	relaffil	satvrmid	satmtmid	ugds
	instnm						
	Alabama A & M University	Normal	AL	0	424.0	420.0	4206.0
	University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	11383.0
	Amridge University	Montgomery	AL	1	NaN	NaN	291.0

Reading in a subset of columns with the `usecols` parameter

The above set of commands is suboptimal. Instead of reading in all of the columns in the college dataset with the `read_csv` function, we can choose a subset to read with the `usecols` parameter. Pass it a list of the columns we want to read in. We can also use the `nrows` parameter to only read in exactly `n` rows.

```
[2]: cols = ['instnm', 'city', 'stabbr', 'relaffil', 'satvrmid', 'satmtmid', 'ugds']
cs = pd.read_csv('../data/college.csv', index_col='instnm', usecols=cols, nrows=3)
cs
```

		city	stabbr	relaffil	satvrmid	satmtmid	ugds
		instnm					
	Alabama A & M University	Normal	AL	0	424.0	420.0	4206.0
	University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	11383.0
	Amridge University	Montgomery	AL	1	NaN	NaN	291.0

Let's add the two SAT columns together and assign the result as a new column. The new column will always be appended to the end of the DataFrame.

```
[3]: cs['sat_total'] = cs['satmtmid'] + cs['satvrmid']
cs
```

		city	stabbr	relaffil	satvrmid	satmtmid	ugds	sat_total
		instnm						
	Alabama A & M University	Normal	AL	0	424.0	420.0	4206.0	844.0
	University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	11383.0	1135.0
	Amridge University	Montgomery	AL	1	NaN	NaN	291.0	NaN

Setting a column equal to a scalar value

You can create a new column by assigning it to be a single scalar value. For instance, the following assignment creates a new column of values equal to the number -99.

```
[4]: cs['some_num'] = -99
cs
```

		city	stabbr	relaffil	satvrmid	satmtmid	ugds	sat_total	some_num
		instnm							
	Alabama A & M University	Normal	AL	0	424.0	420.0	4206.0	844.0	-99
	University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	11383.0	1135.0	-99
	Amridge University	Montgomery	AL	1	NaN	NaN	291.0	NaN	-99

Overwriting an existing column

You can replace the contents of an existing column by assigning it to some other value. Below, we increase the undergraduate population of each college by 10%.

```
[5]: cs['ugds'] = cs['ugds'] * 1.1
cs
```

	city	stabbr	reaffil	satvrmid	satmtmid	ugds	sat_total	some_num
instnm								
Alabama A & M University	Normal	AL	0	424.0	420.0	4626.6	844.0	-99
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	12521.3	1135.0	-99
Amridge University	Montgomery	AL	1	NaN	NaN	320.1	NaN	-99

Create a new column from a numpy array

You can create a new column by assigning it to a numpy array (or another Python sequence) that is the same length as the DataFrame. Below, we create a column of random normal variables.

```
[6]: import numpy as np
cs['random_normal'] = np.random.randn(len(cs))
cs
```

	city	stabbr	reaffil	satvrmid	satmtmid	ugds	sat_total	some_num	random_normal
instnm									
Alabama A & M University	Normal	AL	0	424.0	420.0	4626.6	844.0	-99	1.304677
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	12521.3	1135.0	-99	-0.189258
Amridge University	Montgomery	AL	1	NaN	NaN	320.1	NaN	-99	-0.192813

26.2 Copying a DataFrame

The `copy` method is available to make a completely new copy of a DataFrame that is not associated with the original. This may be necessary in some situations because assigning a DataFrame to a new variable does not copy it. Let's read in a sample DataFrame and assign it to the variable name `df`.

```
[7]: df = pd.read_csv('../data/sample_data.csv', index_col=0)
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

We can create a new variable name by assigning it to `df`. This does not make a new copy of the data.

```
[8]: df1 = df
df1
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

If you are unfamiliar with Python, you might make the mistake and assume that `df` and `df1` reference different DataFrames. What we have, is a single DataFrame object that is referenced by two different variable names. We can prove this with the `is` operator.

```
[9]: df is df1
```

[9]: True

Let's further demonstrate this by modifying `df` by adding a new column to it.

```
[10]: df['new_col'] = 5
df
```

	state	color	food	age	height	score	new_col
name							
Jane	NY	blue	Steak	30	165	4.6	5
Niko	TX	green	Lamb	2	70	8.3	5
Aaron	FL	red	Mango	12	120	9.0	5
Penelope	AL	white	Apple	4	80	3.3	5
Dean	AK	gray	Cheese	32	180	1.8	5
Christina	TX	black	Melon	33	172	9.5	5
Cornelia	TX	red	Beans	69	150	2.2	5

Let's now output df1 to show that it too has changed.

[11]: df1

	state	color	food	age	height	score	new_col
name							
Jane	NY	blue	Steak	30	165	4.6	5
Niko	TX	green	Lamb	2	70	8.3	5
Aaron	FL	red	Mango	12	120	9.0	5
Penelope	AL	white	Apple	4	80	3.3	5
Dean	AK	gray	Cheese	32	180	1.8	5
Christina	TX	black	Melon	33	172	9.5	5
Cornelia	TX	red	Beans	69	150	2.2	5

The variables df and df1 are just two different names that reference the same underlying DataFrame. If you'd like to create a completely new DataFrame with the same data, you need to use the copy method. Let's reread in the same dataset again, but this time assign df1 to a copy of df.

[12]: df = pd.read_csv('../data/sample_data.csv', index_col=0)
df1 = df.copy()

Testing whether df and df1 reference the same DataFrame will result now yield False.

[13]: df is df1

[13]: False

If we add a column to df it will have no effect on df1.

[14]: df['new_col'] = 5
df

	state	color	food	age	height	score	new_col
name							
Jane	NY	blue	Steak	30	165	4.6	5
Niko	TX	green	Lamb	2	70	8.3	5
Aaron	FL	red	Mango	12	120	9.0	5
Penelope	AL	white	Apple	4	80	3.3	5
Dean	AK	gray	Cheese	32	180	1.8	5
Christina	TX	black	Melon	33	172	9.5	5
Cornelia	TX	red	Beans	69	150	2.2	5

Outputting `df1` shows that it is unchanged.

[15]: `df1`

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

26.3 Column and Row Dropping and Renaming

pandas provides the method `drop` to remove rows or columns from the DataFrame. The `rename` method is provided to rename the row or column labels.

Dropping Columns

The `drop` method drops columns passed to the `columns` parameter by setting it to either a string or a list of strings. Let's see examples of dropping a single column and then multiple columns. Remember, that methods return completely new objects, so the original DataFrame is not affected. You'll need to assign the result of the operation to a new variable name if you'd like to proceed with the slimmer DataFrame.

[16]: `cs.drop(columns='city')`

	stabbr	relaffil	satvrmid	satmtmid	ugds	sat_total	some_num	random_normal
instnm								
Alabama A & M University	AL	0	424.0	420.0	4626.6	844.0	-99	1.304677
University of Alabama at Birmingham	AL	0	570.0	565.0	12521.3	1135.0	-99	-0.189258
Amridge University	AL	1	NaN	NaN	320.1	NaN	-99	-0.192813

Use a list to drop multiple columns.

```
[17]: cols = ['city', 'stabbr', 'satvrmid']
          cs.drop(columns=cols)
```

	relaffil	satmtmid	ugds	sat_total	some_num	random_normal
	instnm					
Alabama A & M University	0	420.0	4626.6	844.0	-99	1.304677
University of Alabama at Birmingham	0	565.0	12521.3	1135.0	-99	-0.189258
Amridge University	1	NaN	320.1	NaN	-99	-0.192813

You can also drop rows by `label` and not integer location with the `drop` method using a single label or a list of labels.

```
[18]: rows = ['Alabama A & M University', 'University of Alabama at Birmingham']
cs.drop(index=rows)
```

	city	stabbr	relaffil	satvrmid	satmtmid	ugds	sat_total	some_num	random_normal
instnm									
Amridge University	Montgomery	AL	1	NaN	NaN	320.1	NaN	-99	-0.192813

Renaming Columns

The `rename` method is used to rename columns. Pass a dictionary to the `columns` parameter with keys equal to the old column name and values equal to the new column name. The college dataset has lots of columns with abbreviations that are not immediately recognized. Below, we replace a couple of these columns with more explicit names.

instnm	city	state_abbreviation	religious_affiliation	satvrmid	satmtmid	ugds	sat_total	some_num	random_normal
Alabama A & M University	Normal	AL		0	424.0	420.0	4626.6	844.0	-99
University of Alabama at Birmingham	Birmingham	AL		0	570.0	565.0	12521.3	1135.0	-99
Amridge University	Montgomery	AL		1	NaN	NaN	320.1	NaN	-99

Renaming all columns at once

Instead of using the `rename` method to rename individual columns, you can assign the `columns` attribute a list of the new column names. The length of the list must be the same as the number of columns. Let's first save the original column names to their own variable name so that we can use them in the future.

```
[20]: orig_cols = cs.columns
orig_cols
```

```
[20]: Index(['city', 'stabbr', 'relaffil', 'satvrmid', 'satmtmid', 'ugds',
       'sat_total', 'some_num', 'random_normal'],
       dtype='object')
```

Let's overwrite all of the old columns by assigning them to a list of new column names. Overwriting columns like this does so in-place without making a new copy of the DataFrame.

```
[21]: cs.columns = ['CITY', 'STATE', 'RELAFFIL', 'SATVERBAL', 'SATMATH', 'UGDS', ▾
                   ↵'SAT_TOTAL', 'SOME_NUM', 'RN']
cs
```

instnm	CITY	STATE	RELAFFIL	SATVERBAL	SATMATH	UGDS	SAT_TOTAL	SOME_NUM	RN
Alabama A & M University	Normal	AL	0	424.0	420.0	4626.6	844.0	-99	1.304677
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	12521.3	1135.0	-99	0.189258
Amridge University	Montgomery	AL	1	NaN	NaN	320.1	NaN	-99	0.192813

Let's overwrite these column names again so that they are back to the original names.

```
[22]: cs.columns = orig_cols
cs
```

	city	stabbr	reaffil	satvrmid	satmtmid	ugds	sat_total	some_num	random_normal
instnm									
Alabama A & M University	Normal	AL	0	424.0	420.0	4626.6	844.0	-99	1.304677
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	12521.3	1135.0	-99	-0.189258
Amridge University	Montgomery	AL	1	NaN	NaN	320.1	NaN	-99	-0.192813

26.4 Inserting columns in the middle of a DataFrame

We previously learned about adding a new column to a DataFrame using just the brackets. New columns are always appended to the end of the DataFrame. You can instead use the `insert` method to place the new column in a specific location other than the end. This method has the following three required parameters:

- `loc` - the integer location of the new column
- `column` - the name of the new column
- `value` - the values of the new column

This method works **in-place** and is one of the only ones that does so by default. This means that the calling DataFrame gets modified and nothing is returned. There is no assignment statement when using `insert`. Let's insert the same SAT total right after the `satmtmid` column. We will call it `sat_total_insert` to differentiate it from the column on the end.

```
[23]: new_vals = cs['satvrmid'] + cs['satmtmid']
cs.insert(5, 'sat_total_insert', new_vals)
cs
```

	city	stabbr	reaffil	satvrmid	satmtmid	sat_total_insert	ugds	sat_total	some_num	random_normal
instnm										
Alabama A & M University	Normal	AL	0	424.0	420.0	844.0	4626.6	844.0	-99	1.304677
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	1135.0	12521.3	1135.0	-99	-0.189258
Amridge University	Montgomery	AL	1	NaN	NaN	NaN	320.1	NaN	-99	-0.192813

One minor annoyance is that you must know the integer location of where you'd like to insert the new column. In the above example, it's easy-enough to just count, but a more automated solution would be nice. The pandas Index object has a method called `get_loc` which returns the integer location of a column name.

This is a rare instance in this book where an Index method is used. I advise not digging into Index objects unless there is some very specialized need. So, with some hesitation, I present the `get_loc` Index method here. First, access the `columns` attribute (which is an Index object) and pass the `get_loc` method the name of the column.

```
[24]: cs.columns.get_loc('satmtmid')
```

[24]: 4

Make note that the `get_loc` method does not exist for Series or DataFrame objects. It is strictly an Index method available to either the index or the columns.

Comparison to Python lists

The DataFrame `insert` method is analogous to a Python list method with the same name. It too inserts a value into the list in-place given an integer location. Let's complete an example to compare how it works.

```
[25]: a = ['some', 'list', 'of', 'strings']
a
```

[25]: ['some', 'list', 'of', 'strings']

Call the list `insert` method which mutates the list in-place.

```
[26]: a.insert(1, 'short')
a
```

[26]: ['some', 'short', 'list', 'of', 'strings']

There's also an `index` method that returns the integer location of a particular item in the list which is analogous to the `get_loc` method.

```
[27]: a.index('of')
```

[27]: 3

26.5 The pop method

The DataFrame `pop` method removes a single column from a DataFrame and returns it as a Series. This is different than the `drop` method which removes a column or columns and returns a new DataFrame of the remaining columns. The `pop` method modifies the calling DataFrame in-place. Below, we remove the `ugds` column and assign it to a variable with the same name.

```
[28]: ugds = cs.pop('ugds')
ugds
```

```
[28]: instnm
Alabama A & M University          4626.6
University of Alabama at Birmingham 12521.3
Amridge University                  320.1
Name: ugds, dtype: float64
```

The `cs` DataFrame no longer contains the `ugds` column.

```
[29]: cs
```

	city	stabbr	relaffil	satvrmid	satmtmid	sat_total_insert	sat_total	some_num	random_normal
instnm									
Alabama A & M University	Normal	AL	0	424.0	420.0	844.0	844.0	-99	1.304677
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	1135.0	1135.0	-99	-0.189258
Amridge University	Montgomery	AL	1	NaN	NaN	NaN	NaN	-99	-0.192813

26.6 Exercises

Run the cell below to create a variable name `college_all` that contains all of the rows of the college dataset along with six of the columns. We use the ‘

```
[30]: cols = ['instnm', 'city', 'stabbr', 'relaffil', 'satvrmid', 'satmtmid', 'ugds']
college_all = pd.read_csv('../data/college.csv', index_col='instnm', usecols=cols)
college_all.head()
```

	city	stabbr	relaffil	satvrmid	satmtmid	ugds
instnm						
Alabama A & M University	Normal	AL	0	424.0	420.0	4206.0
University of Alabama at Birmingham	Birmingham	AL	0	570.0	565.0	11383.0
Amridge University	Montgomery	AL	1	NaN	NaN	291.0
University of Alabama in Huntsville	Huntsville	AL	0	595.0	590.0	5451.0
Alabama State University	Montgomery	AL	0	425.0	430.0	4811.0

Exercise 1

Create a new boolean column in the `college_all` DataFrame named ‘Verbal Higher’ that is True for every college that has a higher verbal than math SAT score. Find the mean of this new column. Why does this number look suspiciously low?

```
[ ]:
```

Exercise 2

Find the real percentage of schools with higher verbal than math SAT scores.

```
[ ]:
```

Exercise 3

Create a new column called ‘median all’ that has every value set to the median population of all the schools.

[]:

Exercise 4

Rename the row label ‘Texas A & M University-College Station’ to ‘TAMU’. Reassign the result back to `college_all` and then select this row as a Series.

[]:

Use the City of Houston employee dataset

Execute the following cell to read in the City of Houston employee dataset and use it for the remaining problems.

```
[31]: emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Exercise 5

Create a new column named `bonus` and insert it right after the salary column equal. Have its value equal to 10% of the salary. Round the bonus to the nearest thousand.

[]:

Chapter 27

DataFrame Methods More

In this chapter, we cover several more less common, but still useful and important `DataFrames` methods that you need to know in order to be fully capable at analyzing data with pandas.

- `agg` - Compute multiple aggregations at once
- `idxmax` and `idxmin` - Return the index of the max/min
- `diff` and `pct_change` - Find the difference/percent change from one value to the next
- `sample` - Randomly sample values in a Series
- `nsmallest/nlargest` - Retun the top/bottom `n` values
- `replace` - Replace one or more values in a variety of ways
- `corr` - Computes the correlation between each pair of numeric columns

Let's read in the movie dataset with the title in the index and select just the numeric columns.

```
[1]: import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title').select_dtypes('number')
movie.head()
```

	year	duration	director_fb	actor1_fb	actor2_fb	...	gross	num_reviews	num_voted_users	budget	imdb_score
title											
Avatar	2009.0	178.0	0.0	1000.0	936.0	...	760505847.0	723.0	886204	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	169.0	563.0	40000.0	5000.0	...	309404152.0	302.0	471220	300000000.0	7.1
Spectre	2015.0	148.0	0.0	11000.0	393.0	...	200074175.0	602.0	275868	245000000.0	6.8
The Dark Knight Rises	2012.0	164.0	22000.0	27000.0	23000.0	...	448130642.0	813.0	1144337	250000000.0	8.5
Star Wars: Episode VII - The Force Awakens	Nan	Nan	131.0	131.0	12.0	...	Nan	Nan	8	Nan	7.1

27.1 The `agg` method

The `agg` method allows us to calculate several aggregations at once by providing it a list of the aggregation methods as strings. Here, we find the min, max, and number of unique values for each column.

```
[2]: aggs = movie.agg(['min', 'max', 'nunique'])
aggs
```

	year	duration	director_fb	actor1_fb	actor2_fb	...	gross	num_reviews	num_voted_users	budget	imdb_score
min	1916.0	7.0	0.0	0.0	0.0	...	162.0	1.0	5	2.180000e+02	1.6
max	2016.0	511.0	23000.0	640000.0	137000.0	...	760505847.0	813.0	1689764	4.200000e+09	9.5
nunique	91.0	191.0	435.0	877.0	917.0	...	4033.0	528.0	4750	4.380000e+02	78.0

This returned data might be easier to read when transposed. Let's transpose the results with the T attribute.

```
[3]: aggs.T
```

	min	max	nunique
year	1916.0	2.016000e+03	91.0
duration	7.0	5.110000e+02	191.0
director_fb	0.0	2.300000e+04	435.0
actor1_fb	0.0	6.400000e+05	877.0
actor2_fb	0.0	1.370000e+05	917.0
actor3_fb	0.0	2.300000e+04	906.0
gross	162.0	7.605058e+08	4033.0
num_reviews	1.0	8.130000e+02	528.0
num_voted_users	5.0	1.689764e+06	4750.0
budget	218.0	4.200000e+09	438.0
imdb_score	1.6	9.500000e+00	78.0

27.2 The idxmax and idxmin methods

The `idxmax` and `idxmin` methods return the index where the maximum value occurs for each column. When we call the `idxmax` method on our DataFrame, we learn that the movie with longest duration is ‘Trapped’, the movie with the highest gross is ‘Avatar’, the one highest IMDB score is ‘Towering Inferno’, etc... These methods do NOT work for string columns and will error if used with them.

```
[4]: movie.idxmax()
```

```
[4]: year           Batman v Superman: Dawn of Justice
      duration        Trapped
      director_fb      Don Jon
      actor1_fb        Anchorman: The Legend of Ron Burgundy
      actor2_fb        The Final Destination
      actor3_fb        The Dark Knight Rises
      gross            Avatar
      num_reviews       The Dark Knight Rises
      num_voted_users  The Shawshank Redemption
      budget            Lady Vengeance
```

```
imdb_score                                Towering Inferno
dtype: object
```

27.3 The diff and pct_change methods

The `diff` and `pct_change` methods work just as they do on a Series. Let's read in the `stocks10` dataset which contains the closing stock price for ten stocks beginning from 2010.

```
[5]: stocks = pd.read_csv('../data/stocks/stocks10.csv', index_col='date',
                       parse_dates=['date'])
stocks.head()
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN
1999-10-28	29.01	2.43	16.59	71.00	NaN	21.19	38.85	19.79	NaN	NaN
1999-10-29	29.88	2.50	17.21	70.62	NaN	21.47	39.25	20.00	NaN	NaN

The `diff` method takes the difference between the current value and the nth value preceding it. Below, we get the change in price from two trading days prior.

```
[6]: stocks.head().diff(2)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-26	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-27	-0.51	0.06	-0.50	-6.81	NaN	-0.65	-2.05	1.49	NaN	NaN
1999-10-28	-0.81	0.09	-0.06	-10.25	NaN	0.30	1.74	2.51	NaN	NaN
1999-10-29	0.55	0.12	0.69	-5.32	NaN	0.67	2.31	1.73	NaN	NaN

The `pct_change` return the percentage change as a fraction. Here, we round the number and multiply by 100 so the results are easier to interpret.

```
[7]: stocks.head().pct_change(2).round(3) * 100
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-26	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-27	-1.7	2.6	-2.9	-8.2	NaN	-3.0	-5.3	8.9	NaN	NaN
1999-10-28	-2.7	3.8	-0.4	-12.6	NaN	1.4	4.7	14.5	NaN	NaN
1999-10-29	1.9	5.0	4.2	-7.0	NaN	3.2	6.3	9.5	NaN	NaN

27.4 The sample method

The `sample` method randomly samples rows or columns from the DataFrame. Here, we select three random rows. By default, sampling is done without replacement, so these will be three unique rows.

```
[8]: movie.sample(3)
```

	year	duration	director_fb	actor1_fb	actor2_fb	...	gross	num_reviews	num_voted_users	budget	imdb_score
title											
WALL·E	2008.0	98.0	475.0	1000.0	729.0	...	223806889.0	421.0	718837	180000000.0	8.4
Dance Flick	2009.0	88.0	82.0	756.0	713.0	...	25615792.0	83.0	10161	25000000.0	3.5
Turbulence	1997.0	100.0	0.0	995.0	879.0	...	11466088.0	44.0	8983	71000000.0	4.7

It's possible to randomly sample columns by setting the `axis` parameter to 'columns' or 1.

```
[9]: movie.sample(5, axis='columns').head()
```

	gross	duration	num_voted_users	imdb_score	budget
title					
Avatar	760505847.0	178.0	886204	7.9	237000000.0
Pirates of the Caribbean: At World's End	309404152.0	169.0	471220	7.1	300000000.0
Spectre	200074175.0	148.0	275868	6.8	245000000.0
The Dark Knight Rises	448130642.0	164.0	1144337	8.5	250000000.0
Star Wars: Episode VII - The Force Awakens	NaN	NaN	8	7.1	NaN

Use the `frac` parameter to select a random fraction of the rows and set `replace` equal to `True` to sample with replacement. Here, we select a random 25% of the rows with replacement.

```
[10]: movie.sample(frac=.25, replace=True).shape
```

```
[10]: (1229, 11)
```

27.5 The nlargest/nsmallest methods

The `nlargest` and `nsmallest` methods provide a similar solution that `sort_values` does. Pass them `n`, the number of rows you want to return and `columns`, a string of a column name you would like to use to determine the ordering. The following returns all the rows for movies with the three highest gross values.

```
[11]: movie.nlargest(3, 'gross')
```

	year	duration	director_fb	actor1_fb	actor2_fb	...	gross	num_reviews	num_voted_users	budget	imdb_score
title											
Avatar	2009.0	178.0	0.0	1000.0	936.0	...	760505847.0	723.0	886204	237000000.0	7.9
Titanic	1997.0	194.0	0.0	29000.0	14000.0	...	658672302.0	315.0	793059	200000000.0	7.7
Jurassic World	2015.0	124.0	365.0	3000.0	2000.0	...	652177271.0	644.0	418214	150000000.0	7.0

It is possible to duplicate this with `sort_values` together with the `head` method.

```
[12]: movie.sort_values('gross', ascending=False).head(3)
```

	year	duration	director_fb	actor1_fb	actor2_fb	...	gross	num_reviews	num_voted_users	budget	imdb_score
title											
Avatar	2009.0	178.0	0.0	1000.0	936.0	...	760505847.0	723.0	886204	237000000.0	7.9
Titanic	1997.0	194.0	0.0	29000.0	14000.0	...	658672302.0	315.0	793059	200000000.0	7.7
Jurassic World	2015.0	124.0	365.0	3000.0	2000.0	...	652177271.0	644.0	418214	150000000.0	7.0

Why use `nlargest/nsmallest`?

While `nlargest/nsmallest` can be duplicated with `sort_values`, in theory, `nlargest/nsmallest` should perform better as it uses a [selection algorithm](#) and not a sorting one. The `nlargest/nsmallest` methods also have the ability to keep the top `n` rows with ties by setting the `keep` parameter to `True`.

27.6 The corr method

The `corr` method computes the correlation between every pair of numeric columns in the DataFrame. By default, it computes Pearson's correlation coefficient which is a metric that determines how well the two variables are linearly related returning a score ranging between -1 and 1. When an increase in one variable always corresponds with the same relative increase in the other variable, a perfect positive linear relationship exists and yields a score of 1.

For example, the relationship between Celsius and Fahrenheit is a perfect positive relationship. An increase in one degree Celsius always corresponds with an increase in a 1.8 degree change in Fahrenheit. A perfect negative linear relationship does the opposite and yields the score -1. An increase in one variable always corresponds with the same relative decrease in the other.

The result of the `corr` method is a square DataFrame (has the same number of rows as columns) where the new row labels are the same as the original columns. The number of rows will equal the number of columns. Let's call the method now to compute the correlation between each pair of stocks.

```
[13]: stocks.corr().round(2)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
MSFT	1.00	0.92	0.24	0.98	0.72	0.57	0.91	0.76	0.90	0.99
AAPL	0.92	1.00	0.50	0.94	0.80	0.78	0.95	0.88	0.91	0.97
SLB	0.24	0.50	1.00	0.29	0.06	0.88	0.42	0.65	-0.43	-0.12
AMZN	0.98	0.94	0.29	1.00	0.72	0.62	0.91	0.77	0.90	0.97
TSLA	0.72	0.80	0.06	0.72	1.00	0.64	0.73	0.80	0.82	0.79
XOM	0.57	0.78	0.88	0.62	0.64	1.00	0.73	0.83	0.19	0.63
WMT	0.91	0.95	0.42	0.91	0.73	0.73	1.00	0.84	0.74	0.94
T	0.76	0.88	0.65	0.77	0.80	0.83	0.84	1.00	0.78	0.82
FB	0.90	0.91	-0.43	0.90	0.82	0.19	0.74	0.78	1.00	0.92
V	0.99	0.97	-0.12	0.97	0.79	0.63	0.94	0.82	0.92	1.00

Take a look at the first column of data. This is the pairwise correlation between MSFT and all other stocks. For example, the correlation between MSFT and TSLA is .72. This means that there is a tendency for the stocks MSFT and TSLA to move in the same direction. One should not read too much into correlation. By itself, correlation does not imply a causal relationship between the variables. It is simply a single metric to provide some information about the linear relationship.

The above DataFrame is also **symmetric**. All values along the diagonal are 1, as each stock has a perfect correlation with itself. All values to the left of the diagonal are the same as they are to the right, as the correlation is the same regardless of the order.

Notice that the technology stocks, MSFT, AAPL, AMZN, and FB are all highly correlated with one another. The energy stocks, XOM and SLB, are also highly correlated with one another, but less correlated with the technology stocks.

Series correlation method

Series also have a `corr` method. You must pass it a Series to find its correlation. Below, we get the correlation between MSFT and AAPL, which is the same value found in the DataFrame above.

```
[14]: stocks['MSFT'].corr(stocks['AAPL'])
```

```
[14]: 0.922168731540195
```

27.7 The `replace` method

The `replace` method can be used to replace values in your DataFrame. It is very powerful and flexible. It is also quite complex as there are many different combinations of parameters to handle a variety of different kinds of replacement. Let's read in the first 5 rows of the San Francisco employee compensation dataset dropping the year column. Each numeric column is rounded to the nearest ten-thousand.

```
[15]: sf_emp_head = pd.read_csv('../data/sf_employee_compensation.csv', nrows=5)
sf_emp_head = sf_emp_head.drop(columns='year').round(-4)
sf_emp_head
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Public Protection	Personnel Technician	70000.0	0.0	0.0	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	0.0	0.0	10000.0	10000.0	10000.0
2	Public Protection	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	0.0
3	Community Health	IT Operations Support Admin III	30000.0	0.0	0.0	10000.0	10000.0	0.0
4	Community Health	Special Nurse	30000.0	0.0	10000.0	0.0	0.0	10000.0

The `replace` method has two main parameters, `to_replace` and `value`. The simplest application is to set each one to a single value. Below, we replace all of the values equal to 10,000 with 9,999. All values in the entire DataFrame are searched to be replaced.

```
[16]: sf_emp_head.replace(to_replace=10000, value=9999)
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Public Protection	Personnel Technician	70000.0	0.0	0.0	9999.0	9999.0	9999.0
1	General Administration & Finance	Planner 2	70000.0	0.0	0.0	9999.0	9999.0	9999.0
2	Public Protection	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	0.0
3	Community Health	IT Operations Support Admin III	30000.0	0.0	0.0	9999.0	9999.0	0.0
4	Community Health	Special Nurse	30000.0	0.0	9999.0	0.0	0.0	9999.0

The `replace` method can also replace exact strings. We replace ‘Public Protection’ with ‘PP’.

```
[17]: sf_emp_head.replace(to_replace='Public Protection', value='PP')
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	PP	Personnel Technician	70000.0	0.0	0.0	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	0.0	0.0	10000.0	10000.0	10000.0
2	PP	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	0.0
3	Community Health	IT Operations Support Admn III	30000.0	0.0	0.0	10000.0	10000.0	0.0
4	Community Health	Special Nurse	30000.0	0.0	10000.0	0.0	0.0	10000.0

Instead of using two parameters, you can set `to_replace` to a dictionary to map the old values to the new values. When using a dictionary, you do not use the parameter `value`. Below, we replace ‘Community Health’ with ‘Health’.

```
[18]: sf_emp_head.replace(to_replace={'Community Health': 'Health'})
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Public Protection	Personnel Technician	70000.0	0.0	0.0	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	0.0	0.0	10000.0	10000.0	10000.0
2	Public Protection	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	0.0
3	Health	IT Operations Support Admn III	30000.0	0.0	0.0	10000.0	10000.0	0.0
4	Health	Special Nurse	30000.0	0.0	10000.0	0.0	0.0	10000.0

You can replace as many values as you’d like with a dictionary. The first parameter is `to_replace`, so we can call this method without explicitly providing the parameter name. We import `numpy` to help replace all zeros with missing values.

```
[19]: import numpy as np
sf_emp_head.replace({'Community Health':'Health', 0: np.nan})
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Public Protection	Personnel Technician	70000.0	NaN	NaN	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	NaN	NaN	10000.0	10000.0	10000.0
2	Public Protection	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	NaN
3	Health	IT Operations Support Admin III	30000.0	NaN	NaN	10000.0	10000.0	NaN
4	Health	Special Nurse	30000.0	NaN	10000.0	NaN	NaN	10000.0

Specifying which columns to search for replacement

Calling `replace` as we did above replaces all values in all columns that match the value to replace. Instead, we might be interested in only replacing values in a particular column, or replacing the same value with different values depending on the column.

We can specify which columns to replace which values by using in a dictionary of dictionaries, where the keys of the dictionary specify the column names and the values are the dictionary of original values mapped to their replacement. Take a look at the following dictionary. When passed to the `replace` method, it instructs it to replace 0 with nan and 60,000 with 99,999 for just the overtime column. The retirement column will have 0 replaced with -999.

```
{'overtime':{0: np.nan,
             60000: 99999},
 'retirement': {0: -999}}
```

Let's use this dictionary to make the specific replacement.

```
[20]: sf_emp_head.replace({'overtime':{0: np.nan, 60000:99999},
                           'retirement': {0:-999}})
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Public Protection	Personnel Technician	70000.0	NaN	0.0	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	NaN	0.0	10000.0	10000.0	10000.0
2	Public Protection	Firefighter	120000.0	99999.0	20000.0	20000.0	20000.0	0.0
3	Community Health	IT Operations Support Admin III	30000.0	NaN	0.0	10000.0	10000.0	0.0
4	Community Health	Special Nurse	30000.0	NaN	10000.0	-999.0	0.0	10000.0

Replacing Substrings

As we've seen before, the `replace` method searches for exact strings. Attempting to replace 'Public' with 'Pub.' will do nothing in our DataFrame as there is no exact value 'Public'.

```
[21]: sf_emp_head.replace({'Public': 'Pub.'})
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Public Protection	Personnel Technician	70000.0	0.0	0.0	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	0.0	0.0	10000.0	10000.0	10000.0
2	Public Protection	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	0.0
3	Community Health	IT Operations Support Admin III	30000.0	0.0	0.0	10000.0	10000.0	0.0
4	Community Health	Special Nurse	30000.0	0.0	10000.0	0.0	0.0	10000.0

Let's run the same command but set the `regex` parameter to `True`.

```
[22]: sf_emp_head.replace({'Public': 'Pub.'}, regex=True)
```

	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	Pub. Protection	Personnel Technician	70000.0	0.0	0.0	10000.0	10000.0	10000.0
1	General Administration & Finance	Planner 2	70000.0	0.0	0.0	10000.0	10000.0	10000.0
2	Pub. Protection	Firefighter	120000.0	60000.0	20000.0	20000.0	20000.0	0.0
3	Community Health	IT Operations Support Admin III	30000.0	0.0	0.0	10000.0	10000.0	0.0
4	Community Health	Special Nurse	30000.0	0.0	10000.0	0.0	0.0	10000.0

27.8 Methods available only to Series and not DataFrames

There are more than a few methods that are available only to Series objects, but the following are the most important.

No str or dt accessor

DataFrames have no special methods just for strings or datetimes. There is no `str` or `dt` accessor. They can only be used on Series objects.

No unique or value_counts

Both `unique` and `value_counts` are only available to Series as well.

27.9 Exercises

Execute the following cell to read in the City of Houston dataset and use it to answer the next exercise.

```
[23]: emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Exercise 1

Find the relative frequency of departments for all employees and then find the relative frequency of departments for the top 100 salaries. Compare the differences.

```
[ ]:
```

Exercise 2

Find the day that each stock had its largest percentage one-day drop in price.

```
[ ]:
```


Chapter 28

Assigning Subsets of Data

In previous chapters, we learned how to select subsets of data and create new columns with the assignment statement. In this chapter, we assign subsets of data with new data. This will overwrite the old data in-place. Let's begin by reading in our sample DataFrame.

```
[1]: import pandas as pd  
df = pd.read_csv('../data/sample_data.csv', index_col=0)  
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	2	70	8.3
Aaron	FL	red	Mango	12	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	32	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

28.1 Setting new data with loc

The `loc` indexer simultaneously selects rows and columns from a DataFrame using labels. We covered this in great detail in previous chapters. Let's review this by selecting the age of Niko, Aaron, and Dean as a Series.

```
[2]: rows = ['Niko', 'Aaron', 'Dean']  
df.loc[rows, 'age']
```

```
[2]: name  
Niko      2  
Aaron    12  
Dean     32  
Name: age, dtype: int64
```

We can assign these new values with a list or an array of the same length, or a single scalar value. Let's use the assignment statement to assign new values.

```
[3]: df.loc[rows, 'age'] = [4, 13, 34]
```

Let's verify that the assignment happened correctly.

```
[4]: df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	4	70	8.3
Aaron	FL	red	Mango	13	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	34	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

You can even use one of the augmented assignment operators (`+=`, `-=`, etc...) to operate on the selection itself. Here, we increase the age of these three values by 2.

```
[5]: df.loc[rows, 'age'] += 2
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	Lamb	6	70	8.3
Aaron	FL	red	Mango	15	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	36	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

Set new row values

It's possible to modify values from a single row with `loc`. Here, we change the food and height column values for the row labeled with 'Niko'.

```
[6]: cols = ['food', 'height']
df.loc['Niko', cols] = ['PIZZA', 82]
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	4.6
Niko	TX	green	PIZZA	6	82	8.3
Aaron	FL	red	Mango	15	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	36	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

28.2 Setting new data with iloc

Setting new data with the `iloc` indexer works analogously. We begin by setting a single cell of data. This changes the first row and last column of data.

```
[7]: df.iloc[0, -1] = 99.9
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	99.9
Niko	TX	green	PIZZA	6	82	8.3
Aaron	FL	red	Mango	15	120	9.0
Penelope	AL	white	Apple	4	80	3.3
Dean	AK	gray	Cheese	36	180	1.8
Christina	TX	black	Melon	33	172	9.5
Cornelia	TX	red	Beans	69	150	2.2

The `iloc` indexer can take a single integer, a list of integers, or a slice. Below, we slice the rows and use a single integer for the columns.

```
[8]: df.iloc[3:, 4] = [155, 205, 195, 165]
df
```

	state	color	food	age	height	score
name						
Jane	NY	blue	Steak	30	165	99.9
Niko	TX	green	PIZZA	6	82	8.3
Aaron	FL	red	Mango	15	120	9.0
Penelope	AL	white	Apple	4	155	3.3
Dean	AK	gray	Cheese	36	205	1.8
Christina	TX	black	Melon	33	195	9.5
Cornelia	TX	red	Beans	69	165	2.2

28.3 Boolean selection assignment

Typically, you will not be manually setting rows and columns as shown above. A more common task is to select a portion of the DataFrame with boolean selection and assign that selection new values. Let's see some examples with the employee dataset.

```
[9]: emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept		title	hire_date	salary	sex	race
0	Police		POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II		2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR		2006-01-09	49275.00	Male	Black

Let's say we wanted to raise the minimum salary for all police department employees to 60,000. Before making the assignment let's find the number of police department employees currently making less than this.

```
[10]: filt1 = emp['salary'] < 60000
filt2 = emp['dept'] == 'Police'
filt = filt1 & filt2
filt.sum()
```

[10]: 2190

Use the `loc` indexer to select just the employees that meet the conditions for the above filter and reassign their salary.

Let's use our same filter to select those employees and verify that their salary has now changed.

```
[11]: emp[filt].head(3)
```

	dept	title	hire_date	salary	sex	race
38	Police	POLICE OFFICER	2017-07-17	56956.64	Male	White
54	Police	POLICE TRAINEE	2018-09-04	42000.00	Male	Asian
59	Police	ADMINISTRATIVE SPECIALIST	1998-12-11	51407.00	Female	Hispanic

28.4 Improper Assignment

The above assignment is often done improperly, and in a way that has no effect. Let's reread in the dataset as a new variable name `emp2` and recreate our filters.

```
[12]: emp2 = pd.read_csv('../data/employee.csv')
filt1 = emp2['salary'] < 60000
filt2 = emp2['dept'] == 'Police'
filt = filt1 & filt2
emp2[filt].head(3)
```

	dept	title	hire_date	salary	sex	race
38	Police	POLICE OFFICER	2017-07-17	56956.64	Male	White
54	Police	POLICE TRAINEE	2018-09-04	42000.00	Male	Asian
59	Police	ADMINISTRATIVE SPECIALIST	1998-12-11	51407.00	Female	Hispanic

The last expression from above returns a DataFrame object which we can use *just the brackets* again to select the `salary` column.

```
[13]: emp2[filt]['salary'].head(3)
```

```
[13]: 38    56956.64
      54    42000.00
      59    51407.00
Name: salary, dtype: float64
```

If we try to use an assignment statement with the above syntax, no change will take place and a `SettingWithCopyWarning` will be emitted. Let's attempt the assignment and trigger the warning. Note, that this is a warning and not an error. The statement completed successfully.

```
[14]: emp2[filt]['salary'] = 60000
```

```
/Users/Ted/miniconda3/envs/minimal_ds/lib/python3.7/site-
packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

"""Entry point for launching an IPython kernel.

Selecting those employees that we were hoping to change salary exposes the improper assignment.

[15]: `emp2[filt].head(3)`

	dept		title	hire_date	salary	sex	race
38	Police		POLICE OFFICER	2017-07-17	56956.64	Male	White
54	Police		POLICE TRAINEE	2018-09-04	42000.00	Male	Asian
59	Police	ADMINISTRATIVE SPECIALIST		1998-12-11	51407.00	Female	Hispanic

What went wrong?

Executing `emp2[filt]['salary']` is called **chained indexing** in the pandas documentation or with the terminology in this book **chained selections**. There were two consecutive selections. The first was boolean selection with `[filt]` followed immediately by single-column selection with `['salary']`.

The issue is that the first selection, `emp2[filt]`, creates a completely new DataFrame with its own copy of data in memory that has nothing to do with the original DataFrame. From this new DataFrame, we select the `salary` column and attempt to reassign each value. What we have done is set the salary for this copy of the data. pandas is nice-enough to give us a warning that we might not have accomplished what we thought we did. In this example, the warning proved to be correct and our original DataFrame was not modified. In order to properly assign a subset of data using boolean selection along a column, you need to use `loc`, which is a single selection (one set of brackets) that doesn't involve making a copy of the data.

The `SettingWithCopyWarning` requires a deeper discussion to fully understand which will be presented in a later chapter.

28.5 Exercises

Use the bikes dataset for all of the following exercises.

[16]: `bikes = pd.read_csv('../data/bikes.csv')`

Exercise 1

Change the values of `events` to ‘HEAT WAVE’ for all rides where `temperature` is above 95. Verify this by outputting just the `events` and `temperature` columns that meet the condition.

[]:

Exercise 2

Increase the trip duration by 50% for all the rides that took place with a wind speed above 40. Output just the trip duration and wind speed columns both before and after the assignment.

[]:

Exercise 3

Change the trip duration for the first two rows to 0.

[]:

Part V

Data Types

Chapter 29

Integer, Float, and Boolean Data types

This chapter will go deeper into the different integer, float, and boolean data types that are available to pandas Series and DataFrames. As a reminder, all values in a Series are exactly one data type. Similarly, all values in each column of a DataFrame are exactly one data type.

While the words ‘boolean’, ‘integer’, and ‘float’ are useful descriptions, they are not the technical names of the actual data types. This chapter will give a complete picture of all the exact integer, float, and boolean data types that are available and what they mean. We will also be making heavy use of the `astype` method to change data types.

29.1 Constructing a Series

Before getting started with our data type discussion, we will learn how to create a Series using its constructor. All of our previous Series have been created by selecting a single column from a DataFrame. It is also possible to create a Series manually using `pd.Series` by passing it a list of values. Let’s create a simple integer Series with a few values.

```
[1]: import pandas as pd  
s_int = pd.Series([50, 99, 130])  
s_int
```

```
[1]: 0    50  
1    99  
2   130  
dtype: int64
```

Creating a Series in this manner is often referred to as using the **constructor**, which is a generic programming term used to describe the creation and instantiation of a new object of a particular type. We used the Series constructor to create a new Series of integers.

29.2 Integer data type

The visual output of a Series displays the data type of the values below it. In this case, it is ‘int64’, which formally represents a 64-bit integer. This data type comes directly from numpy, which allows integers to be either 8, 16, 32, or 64 bits in size. A 64-bit integer can contain up to 2 raised to the 64th power number of integers. Let’s see how many numbers this is.

```
[2]: 2 ** 64
```

[2]: 18446744073709551616

The ‘int64’ data type has both positive and negative integers. Let’s divide the above number by 2 to get the maximum integer allowed.

[3]: 2 ** 64 // 2

[3]: 9223372036854775808

numpy has a function called `iinfo` that returns the exact integer information for each integer data type. Pass it the data type as a string to get the information. Note that the range of integers is exactly 2 raised to the 64th power when accounting for 0.

[4]: `import numpy as np
np.iinfo('int64')`

[4]: `iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)`

We can find the range of 8-bit integers with the following command.

[5]: `np.iinfo('int8')`

[5]: `iinfo(min=-128, max=127, dtype=int8)`

The span of numbers for ‘int8’ goes from -128 to 127 or 256 total numbers. This is equivalent to 2 raised to the 8th power.

29.3 Changing Data Types with `astype`

You can change a Series data type with the `astype` method by passing it the string name of the data type. The string name is going to be the base data type, which is ‘int’ in this case, appended directly to the number of bits (8, 16, 32, or 64).

Below, we change the data type to ‘int8’. Notice that the third value now displays as -126 and not its original value of 130. The maximum 8-bit integer is 127, making 130 greater than it by 3. numpy assumes you know what you are doing and does not check that this number goes beyond its maximum. Instead, the number is represented by the third integer greater than the minimum -128 which is -126.

[6]: `s_int.astype('int8')`

[6]:

0	50
1	99
2	-126
dtype: int8	

Default integer types

Unfortunately, the default data type is dependent on the platform that you are using. numpy is built with the C programming language, and uses the size of the C `long` type as its default. For 32-bit Linux, macOS, and Windows machines, this will be 32 bits. For 64-bit Linux and macOS machines, it will be 64 bits. For 64-bit Windows machines this will be 32-bits. The first two rows of this table show the size of C long types for each platform.

Windows machines

If you have a Windows machine, you may have noticed that your data type still says ‘int64’ even though the previous section just said otherwise. This is because we constructed our Series with a list and not a numpy array. Let’s construct a numpy array of integers using its constructor, `array`.

```
[7]: a = np.array([1, 5])
a
```

```
[7]: array([1, 5])
```

numpy arrays use the exact same `dtype` attribute as Series to access the data type. Windows users should see ‘int32’ as the data type, while 64-bit Linux and macOS users should see ‘int64’.

```
[8]: a.dtype
```

```
[8]: dtype('int64')
```

Constructing the Series with a numpy array will show the same data type as above.

```
[9]: pd.Series(a)
```

```
[9]: 0    1
      1    5
      dtype: int64
```

29.4 Unsigned Integers

The default integer data types split half of their range between negative and positive integers. It is possible to limit your integers to just the non-negative integers by using the unsigned integer data type abbreviated with the string ‘uint’. The same sizes 8, 16, 32, and 64 bits are available. Let’s convert the original `s_int` Series to ‘uint8’.

Calling the `astype` method returns a copy of the data, and does not work in-place. We first verify that the `s_int` Series remained still has data type of ‘int64’.

```
[10]: s_int.dtype
```

```
[10]: dtype('int64')
```

We now convert the data type to an 8-bit unsigned integer.

```
[11]: s_int.astype('uint8')
```

```
[11]: 0    50
      1    99
      2   130
      dtype: uint8
```

The last value remains correctly as 130 as the range of this new data type is from 0 to 255. Let’s verify this with the `iinfo` method.

```
[12]: np.iinfo('uint8')
```

[12]: `iinfo(min=0, max=255, dtype=uint8)`

It is rare to use unsigned integers, but they are available, and can be useful in specific situations where you would like to save memory. Typically, it won't be necessary to use them, so sticking with the default integer data type should work.

29.5 Nullable integer data type

With the release of pandas version 0.24 in late 2019, a new nullable integer data type became available to pandas users. This new data type allows for missing values to be present in a column of integers. This is a completely separate data type than the normal integer data types. The original integer data types still exist and are unable to contain missing values. Let's verify this by attempting to construct an integer Series with missing values. Notice, that we use the `dtype` parameter to attempt to force the data type to be 'int64'.

[13]: `pd.Series([50, 99, 130, np.nan], dtype='int64')`

```
ValueError: cannot convert float NaN to integer
```

If you do not use the `dtype` parameter, then the Series will be constructed, but use the more flexible float64 data type which does allow for missing values.

[14]: `pd.Series([50, 99, 130, np.nan])`

```
0    50.0
1    99.0
2   130.0
3    NaN
dtype: float64
```

Constructing a Series of nullable integers

The nullable integer data type is represented by the string 'Int' (as opposed to 'int'). The important distinction is the capitalized first letter I. The same four bit sizes, 8, 16, 32, and 64 are available. Let's construct a Series of nullable integers using the string 'Int64'.

[15]: `s_nullable_int = pd.Series([50, 99, 130, np.nan], dtype='Int64')`
`s_nullable_int`

```
0    50
1    99
2   130
3    <NA>
dtype: Int64
```

The missing value is visually represented by <NA> which is different than `NaN` when the Series was constructed with `float64`. pandas introduced its own missing value object `NA` that is distinct from numpy's `NaN`. pandas will convert any missing value in nullable integer Series to its own 'NA'. Let's use pandas `NA` object directly in the Series construction to show that the same Series is created.

[16]: `pd.Series([50, 99, 130, pd.NA], dtype='Int64')`

```
[16]: 0      50
       1      99
       2     130
       3    <NA>
dtype: Int64
```

Nullable integers are experimental

The nullable integer data type is labeled as “experimental” indicating that its behavior might change in the future. There are also many bugs associated with this data type. It is my suggestion that you avoid using this data type for serious work until it is no longer experimental

Nullable integers don't exist in numpy

The nullable integer data type is only available for pandas objects. It does not exist in numpy.

Different behavior of nullable integer

The nullable integer data type exhibits different behavior than the normal integer data type. If attempting to construct a Series with values that are not within its range, an exception will be raised and not wrap around like it did above. This is probably better behavior to prevent mistakes.

```
[17]: pd.Series([50, 99, 130, pd.NA], dtype='Int8')
```

```
TypeError: cannot safely cast non-equivalent object to int8
```

One major difference is that boolean selection (and the DataFrame `query` method) will not work if a missing value exists. Let's show this by selecting all values in both the normal integer and nullable integer Series that are greater than 100.

```
[18]: filt = s_int > 100
s_int[filt]
```

```
[18]: 2     130
dtype: int64
```

Boolean selection will not work for nullable integers.

```
[19]: filt = s_nullable_int > 100
s_nullable_int[filt]
```

```
[19]: 2     130
dtype: Int64
```

Missing value evaluation with comparison operators

The reason the nullable integer data type fails to properly do missing value is because the pandas `NA` object evaluates as `NA` when used with one of the comparison operators. Let's inspect the values formed by the filter from above.

```
[20]: filt = s_nullable_int > 100
filt
```

```
[20]: 0    False
      1    False
      2     True
      3    <NA>
dtype: boolean
```

This returns a Series with the nullable boolean data type, new to pandas in 1.0 and covered in detail below. It is impossible (at the moment) to do boolean selection when one of the values in the filter is missing.

When using the original integer data type, ‘int64’, you won’t have this problem. The numpy `nan` object evaluates as `False` which allows you to do boolean selection. Let’s construct the same Series, but this time allow pandas to use its default data type, ‘float64’, for numbers mixed with missing values.

```
[21]: s = pd.Series([50, 99, 130, np.nan])
s
```

```
[21]: 0    50.0
      1    99.0
      2   130.0
      3    NaN
dtype: float64
```

Using the same operation for the filter will return a Series of all boolean values (using the original non-nullable boolean data type). This Series can be used for boolean selection.

```
[22]: filt = s > 100
filt
```

```
[22]: 0    False
      1    False
      2     True
      3    False
dtype: bool
```

All comparisons with numpy `nan` result in `False`, while all comparisons with the new pandas `NA` evaluate as `NA`. A few examples involving different comparison operators are presented below for each of the missing value objects.

```
[23]: np.nan > 5
```

```
[23]: False
```

```
[24]: np.nan < 5
```

```
[24]: False
```

```
[25]: np.nan == 5
```

```
[25]: False
```

```
[26]: pd.NA > 5
```

```
[26]: <NA>
```

```
[27]: pd.NA < 5
```

[27]: <NA>

```
[28]: pd.NA == 5
```

[28]: <NA>

Filtering with nullable integers

In my opinion, this is a major limitation. Boolean selection will work if no missing values are present in the data. Below, we fill in the missing values in the filter with `False` using the `fillna` method. This workaround now matches the previous behavior.

```
[29]: filt = s_nullable_int > 100
s_nullable_int[filt.fillna(False)]
```

```
[29]: 2    130
      dtype: Int64
```

Unsigned nullable integers

Unsigned nullable integers are also available with the string ‘UIInt’. Our Series construction that failed with the signed 8-bit nullables from above now works with the unsigned 8-bit nullable integer as 130 is part of the range.

```
[30]: pd.Series([50, 99, 130, pd.NA], dtype='UIInt8')
```

```
[30]: 0    50
      1    99
      2    130
      3    <NA>
      dtype: UInt8
```

29.6 Summary of integer data type

In summary, there are four major types of integers, each of them available in 8, 16, 32, and 64 bits.

- Integer - `int`
- Nullable Integer - `Int`
- Unsigned Integer - `uint`
- Unsigned Nullable Integer - `UIInt`

29.7 Float data types

Float columns contain numbers with decimal places. The default ‘float’ size for all platforms is 64 bits (the same size as a C double) and also referred to as ‘double-precision’. numpy has additional float sizes of 16 and 32 bits. All float data types can contain missing values. Let’s create a Series with floats containing a single missing value and verify the data type.

```
[31]: s_float = pd.Series([4.247, 1234.56789, np.nan])
s_float
```

```
[31]: 0      4.24700
1     1234.56789
2          NaN
dtype: float64
```

```
[32]: s_float.dtype
```

```
[32]: dtype('float64')
```

Below, we change the data type to a 32-bit float, also known as ‘single-precision’ float. Notice how the second value has changed, as a 32-bit float does not have enough precision to map its value exactly.

```
[33]: s_float.astype('float32')
```

```
[33]: 0      4.247000
1     1234.567871
2          NaN
dtype: float32
```

We can use the numpy `finfo` function to get information on each float type. For instance, the ‘float32’ data type guarantees us 6 significant digits of precision as seen with the ‘resolution’ attribute below.

```
[34]: np.finfo('float32')
```

```
[34]: finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

A 16-bit ‘half-precision’ float only guarantees 3 digits of precision.

```
[35]: np.finfo('float16')
```

```
[35]: finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04, dtype=float16)
```

```
[36]: s_float.astype('float16')
```

```
[36]: 0      4.246094
1     1235.000000
2          NaN
dtype: float16
```

29.8 Changing from float to int

So far, we have only changed the size of integer or float data types. We can change types from float to integer and vice-versa. Below, we attempt to go from a ‘float64’ to an ‘int64’. This will fail as the normal integer data type does not allow for missing values.

```
[37]: s_float.astype('int64')
```

```
ValueError: Cannot convert non-finite values (NA or inf) to integer
```

If you drop the missing values, then conversion is possible. The decimals are truncated and NOT rounded.

```
[38]: s_float.dropna().astype('int64')
```

```
[38]: 0      4  
1    1234  
dtype: int64
```

The behavior for the nullable integer data type differs, it does not allow conversion if there are any decimal places have values.

```
[39]: s_float.astype('Int64')
```

```
TypeError: cannot safely cast non-equivalent float64 to int64
```

If you remove the decimals (through rounding as one example), then conversion to the nullable integer is possible.

```
[40]: s_float.round(0).astype('Int64')
```

```
[40]: 0      4  
1    1235  
2    <NA>  
dtype: Int64
```

Going from integer to float will not have as dramatic of an effect and there should be no loss of data as long as the float has enough bits to represent all the digits.

```
[41]: s_int.astype('float64')
```

```
[41]: 0      50.0  
1      99.0  
2     130.0  
dtype: float64
```

Conversion from nullable integer to float is also possible. Since floats are numpy data types, they use NaN as their missing value representation and not pd.NA.

```
[42]: s_nullable_int.astype('float64')
```

```
[42]: 0      50.0  
1      99.0  
2     130.0  
3      NaN  
dtype: float64
```

Visual display of integers and floats

pandas always display floats with decimals even if there are no significant digits after the decimal. At a minimum, the .0 will be present. On the other hand, integers are always displayed without a decimal. You can use this rule of thumb to determine the data type without actually accessing the `dtypes` attribute.

No pandas float data type

pandas relies completely on numpy for its float data type. There is no special pandas-only float data type. Integers and booleans have both numpy and pandas data types, with the pandas variety able to include missing values.

29.9 Boolean data type

Let's now cover the simpler boolean data types. Booleans have a single 8-bit data type in numpy. It makes sense that bit sizes of 16, 32, or 64 don't exist for boolean data types as there are only two possible values, `True` and `False`. You may be curious as to why booleans are not represented with a single bit. This is because a byte (8 bits) is the smallest addressable unit of memory available to modern computers. Let's create a boolean Series using the constructor.

```
[43]: s_bool = pd.Series([True, False])
s_bool
```

```
[43]: 0      True
      1      False
      dtype: bool
```

We verify the data type below.

```
[44]: s_bool.dtype
```

```
[44]: dtype('bool')
```

Converting to and from boolean

It is possible to convert integer and float columns to boolean and vice-versa. The only value that will be converted to `False` is 0. All other values are converted to `True`. Use the string 'bool' to convert to boolean. We begin by creating a Series with the integer data type.

```
[45]: s = pd.Series([0, 1, 99, -14])
```

Call the `astype` method to make the conversion.

```
[46]: s.astype('bool')
```

```
[46]: 0      False
      1      True
      2      True
      3      True
      dtype: bool
```

Series with float data types work the same when converting to boolean. Only the value 0 is converted to `False`. Any other value becomes `True`.

```
[47]: s = pd.Series([0, 0.0001, -3.99])  
s
```

```
[47]: 0    0.0000  
1    0.0001  
2   -3.9900  
dtype: float64
```

```
[48]: s.astype('bool')
```

```
[48]: 0    False  
1     True  
2     True  
dtype: bool
```

Python itself uses the same rules to convert integers and floats to boolean. Let's see a few examples with the built-in `bool` constructor.

```
[49]: bool(5)
```

```
[49]: True
```

```
[50]: bool(0)
```

```
[50]: False
```

```
[51]: bool(-3.4)
```

```
[51]: True
```

```
[52]: bool(0.00000)
```

```
[52]: False
```

Converting a boolean Series to integer or float will convert all `True` values to 1 and `False` to 0.

```
[53]: s_bool.astype('int64')
```

```
[53]: 0    1  
1    0  
dtype: int64
```

Using a 64-bit integer to store a boolean is overkill. The smallest integer type, `int8` (or `uint8`), can be used to save memory.

```
[54]: s_bool.astype('int8')
```

```
[54]: 0    1  
1    0  
dtype: int8
```

29.10 Nullable boolean data type

With the release of pandas 1.0, a new nullable boolean type was made available to support missing values. Until this release, pandas relied on numpy's boolean data type. The nullable boolean is a pandas-only data type, just like the nullable integer. The original boolean data type still exists, but does not support missing values. Let's verify that the original boolean data type cannot contain missing values.

```
[55]: s = pd.Series([True, False, np.nan], dtype='bool')
s
```

```
[55]: 0      True
      1      False
      2      True
dtype: bool
```

Surprisingly, this does not fail, but instead converts the numpy `nan` object to `True`. This follows the rule that every non-zero converts to `True` for booleans, even missing values. Confusingly, assigning `nan` to one value in a boolean Series converts the entire Series to a float64.

```
[56]: s.loc[0] = np.nan
s
```

```
[56]: 0      NaN
      1      0.0
      2      1.0
dtype: float64
```

Adding to the confusion, constructing a Series of booleans mixed with missing values without explicitly setting the data type preserves the individual type of each value by using the flexible object data type. While this is possible, I advise against mixing data of different types in a single column.

```
[57]: s = pd.Series([True, False, np.nan])
s
```

```
[57]: 0      True
      1      False
      2      NaN
dtype: object
```

Using the new nullable boolean data type

The new nullable boolean data type uses the string ‘boolean’ to reference it as opposed to ‘bool’. Like the nullable integer, it is a pandas-only data type and is not able to be used during boolean selection. Here, we construct a Series with the nullable boolean data type.

```
[58]: s = pd.Series([True, False, np.nan], dtype='boolean')
s
```

```
[58]: 0      True
      1      False
      2      <NA>
dtype: boolean
```

29.11 Changing data types with an arithmetic operation

Computing an arithmetic operation on a Series can change the resulting Series data type. Division always converts an integer Series to float, even if the result is whole numbers.

```
[59]: s = pd.Series([-15, 45])  
s
```

```
[59]: 0    -15  
1     45  
dtype: int64
```

The result of using the division operator on an integer Series is always a float.

```
[60]: s / 15
```

```
[60]: 0    -1.0  
1     3.0  
dtype: float64
```

This is consistent with core Python which makes the same type conversion.

```
[61]: type(15 / 3)
```

```
[61]: float
```

Using floor division, on the other hand, keeps the result as an integer as long as the divisor is an integer.

```
[62]: s // 77
```

```
[62]: 0    -1  
1     0  
dtype: int64
```

Multiplying an integer Series by a float always converts it to another float, even if all the results are integers.

```
[63]: s * 4.4
```

```
[63]: 0    -66.0  
1    198.0  
dtype: float64
```

Many other combinations of data types paired with operations are possible. They will not be presented here. Instead, it is encouraged for you to test them on your own to view the result.

Changing data types by assigning new values

As we saw with boolean Series, it is possible to change the data type by assigning one of the values to a different type. Setting one value in an integer Series to a float changes the entire data type of the Series to a float as well.

```
[64]: s.loc[0] = 1.99  
s
```

```
[64]: 0      1.99
      1     45.00
      dtype: float64
```

Setting a non-nullable boolean Series value to the new pandas NA object converts the Series to object and not to the nullable boolean.

```
[65]: s = pd.Series([True, False])
       s.loc[0] = pd.NA
       s
```

```
[65]: 0      <NA>
      1     False
      dtype: object
```

From here, you can convert to the nullable boolean with the `astype` method.

```
[66]: s.astype('boolean')
```

```
[66]: 0      <NA>
      1     False
      dtype: boolean
```

Converting the data type of an entire Series can be an expensive operation, so it's important to be aware of the possibilities when it happens. There's no need to remember these conversion rules, but to proceed when caution when the possibility arises.

29.12 Setting data types in numpy arrays

We've discussed setting the data type for a pandas Series with the `dtype` parameter. In this section, we will learn how to set the data type for a numpy array, which uses the same parameter `dtype`. You can set it to the string name of the data type you desire. Let's construct an 8-bit integer array. Notice that 150 exceeds the max value of 127.

```
[67]: np.array([1, 5, 150], dtype='int8')
```

```
[67]: array([ 1, 5, -106], dtype=int8)
```

Here we force numpy to use a 32-bit float. Normally, it would default this array as a 32 or 64 bit integer (depending on your machine).

```
[68]: np.array([1, 5, 150], dtype='float32')
```

```
[68]: array([ 1., 5., 150.], dtype=float32)
```

Creating Series from numpy arrays

The Series constructor accepts one-dimensional numpy arrays as input. The resulting Series will have the same data type as the numpy array.

```
[69]: a = np.array([1, 5, 150], dtype='float32')
       pd.Series(a)
```

```
[69]: 0      1.0
       1      5.0
       2    150.0
dtype: float32
```

29.13 Different syntax for data types

All of the data type conversions in this chapter were accomplished by using a string such as ‘int8’. There is an alternative approach that can be used. Instead of a string, you can use the actual object itself available directly from numpy or pandas. This section showcases a different method to setting the data type of a Series. For instance, we can use `np.int8` instead of the string ‘int8’ to specify a data type.

```
[70]: pd.Series([10, 50]).astype(np.int8)
```

```
[70]: 0      10
       1      50
dtype: int8
```

The following is equivalent to the above.

```
[71]: pd.Series([10, 50]).astype('int8')
```

```
[71]: 0      10
       1      50
dtype: int8
```

All of the numpy data type objects have the same name as their string counterparts. This isn’t true for the pandas-only data types. These objects end with the word ‘Dtype’. For instance, to convert to a 32-bit nullable integer, you can use `pd.Int32Dtype()`. The parentheses instantiate this data type.

```
[72]: pd.Series([10, 50]).astype(pd.Int32Dtype())
```

```
[72]: 0      10
       1      50
dtype: Int32
```

For the nullable boolean data type, use `pd.BooleanDtype()`.

```
[73]: pd.Series([True, False, np.nan], dtype=pd.BooleanDtype())
```

```
[73]: 0      True
       1     False
       2    <NA>
dtype: boolean
```

Does it matter which one you use?

You are free to use either one, but I typically use strings when specifying a data type, as numpy must be imported to use the object directly.

29.14 Boolean, integer, and float data type summary

Numpy data types available to pandas

Generic Name	String Name of Default	All Bit Sizes
Boolean	bool	8
Integer	int64 or int32	8, 16, 32, 64
Unsigned Integer	uint64 or uint32	8, 16, 32, 64
Float	float64	16, 32, 64

Note - Missing values only available to floats

Data types specific to pandas

Generic Name	String Name of Default	Pandas Object name of Default	All Bit Sizes
Nullable Boolean	Boolean	pd.BooleanDtype()	8
Nullable Integer	Int64	pd.Int64Dtype()	8, 16, 32, 64
Nullable Unsigned Integer	UInt64	pd.UInt64Dtype()	8, 16, 32, 64

Note - Missing values available as pd.NA

29.15 Exercises

Exercise 1

Find the maximum number of a 16-bit integer using arithmetic operations. Then verify it with numpy's `iinfo` function.

[]:

Exercise 2

Construct a Series that has the nullable integer data type. Make sure it has a mix of integers and missing values.

[]:

Exercise 3

Take a look at the Series below. Change its data type such that it uses the least amount of memory and preserves the numbers as they are.

[]:

Exercise 4

Find the precision of a 32-bit float and then create a numpy array with values that have decimal places past that precision.

[]:

Exercise 5

Create a Series of numbers that have decimal places. Use the `astype` method to convert it to an integer and then back to a float. Are the decimals from the original Series preserved?

[]:

Exercise 6

Create a numpy array with 8-bit unsigned integers. Use negative numbers in the construction along with numbers greater than 255. Does the output make sense?

[]:

Exercise 7

Create a numpy array that has the values 50 and 100 in it, but do so without actually using those two values (or any operations that create them).

[]:

Exercise 8

Create a numpy array that contains two integers and the numpy nan missing value. Assign it to a variable name and output it to the screen. What data type is it?

[]:

Exercise 9

Construct a Series from the array created in exercise 8. What data type is it? Construct a new Series with the same array forcing it to be a nullable integer.

[]:

Exercise 10

Construct a Series of 32-bit nullable integers using the data type object itself (and not the string).

[]:

Chapter 30

Object, String, and Categorical Data Types

In this chapter, our focus will be on the data types that primarily contain strings. Currently, pandas uses the three data types object, string, and categorical. The string data type is new to pandas 1.0 and experimental. It is suggested that you do not use it for serious work as its functionality may change. The categorical data type has yet to be discussed, but is a fantastic way to save memory and significantly increase performance for operations done on columns containing strings..

30.1 Object data types

Until the release of pandas 1.0, pandas did not have a string-only data type. Instead, it used the ‘object’ data type to hold strings. As mentioned previously, the object data type has no limitation as to what Python object can be within it. It is essentially a catch-all for any item that you desire to be in a DataFrame that doesn’t belong to the other specific data types.

Series with the object data type do not have analogous size representation like integer and float data types do. There is no ‘object64’, only a single ‘object’ data type. Each item can be of a different type and therefore of a different size.

Though the object data type may contain any Python object, it is primarily used to hold strings. Let’s begin by constructing a Series with a couple of strings as values.

```
[1]: import pandas as pd
import numpy as np
s_object = pd.Series(['some', 'strings'])
s_object
```

```
[1]: 0      some
1    strings
dtype: object
```

As you can see from the bottom of the Series output, the data type is ‘object’. When we verify this, you’ll see the output `dtype('O')`. The object data type also comes directly from numpy and uses ‘O’ to represent it instead of the full name.

```
[2]: s_object.dtype
```

```
[2]: dtype('O')
```

Because object is the most flexible type, any Series may be converted to it. Here we convert a Series of integers to object.

```
[3]: s = pd.Series([5, 10])
s.astype('object')
```

```
[3]: 0      5
     1     10
dtype: object
```

The underlying values are still integers. We verify this by finding the type of the first value.

```
[4]: type(s.loc[0])
```

```
[4]: numpy.int64
```

This is something you would never want to do as numpy integer arrays are optimized for fast computation. By converting to an object array, you would lose this excellent benefit. In the upcoming example, a numpy array is created with 100,000 random integers between 0 and 100. A second array is created by converting the data type to object. We then time how long it takes to sum each array. On my machine, the integer array is about 50x as fast as object array even though they both hold the exact same data.

```
[5]: a_fast = np.random.randint(low=0, high=100, size=100_000)
a_slow = a_fast.astype('object')
```

```
[6]: %timeit -n 5 a_fast.sum()
```

58.8 μ s \pm 21.2 μ s per loop (mean \pm std. dev. of 7 runs, 5 loops each)

```
[7]: %timeit -n 5 a_slow.sum()
```

2.8 ms \pm 101 μ s per loop (mean \pm std. dev. of 7 runs, 5 loops each)

Many different types in a Series

There is no restriction on what can be placed in a Series with the object data type. The following Series contains a list, a boolean, a string, a float, and a dictionary.

```
[8]: pd.Series([[1,2], True, 'some string', 4.5, {'key': 'value'}])
```

```
[8]: 0      [1, 2]
     1      True
     2    some string
     3      4.5
     4  {'key': 'value'}
dtype: object
```

Object Series usually contain strings

As we have mentioned in previous chapters, when you encounter a Series or column of a DataFrame that has object as its data type, it usually contains nothing but strings.

Poor practice to store complex data types within Series

Even though you are allowed to place any Python object within a Series, it's generally considered poor practice to do so. Series with object data types are designed to be filled with strings as the `str` accessor is available for this data type. This shouldn't be seen as an absolute statement since it might be necessary to use the flexibility of these object columns for some advanced operations.

30.2 Categorical data type

We now introduce the categorical data type, which is unique to pandas and does not exist within numpy. The categorical data type is often used whenever a column of data has known, limited, and discrete values. This is often the case for string columns and is easiest to understand with an example. Let's read in the City of Houston employee dataset.

```
[9]: import pandas as pd
emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Changing data types to categorical

Any column may be converted to the categorical data type, but most often it is string (and occasionally an integer) column that are chosen. In this dataset, each string column is a good candidate to be categorical. Let's select the `dept` column and count the occurrences of each unique value.

```
[10]: dept = emp['dept']
dept.value_counts()
```

```
[10]: Police           7573
      Fire            4376
      Houston Public Works 4190
      Other            3373
      Health & Human Services 1353
      Houston Airport System 1216
      Parks & Recreation    1152
      Library            563
      Solid Waste Management 512
      Name: dept, dtype: int64
```

Values are known, limited, and discrete

In this section, we discuss the properties that make a particular column a good candidate to convert to categorical. The total possible number of values in the column should be **known**. There shouldn't be any mysterious values that will appear when future data of the same kind is collected. With our department column, it's likely that we will be aware of each individual department and that new departments will not be created often.

The unique values in the column should be **limited** and much less than the total number of values. There are only 9 unique department values, which is substantially less than the total number of values (24,000+).

The values should be **discrete**, meaning that there are no partial values. Each value in the column must be one of the known categories. With our data, each employee works in exactly one department. There is no partial department. You either work in one department or another, but not both. You must be one of those 9 categories.

If the values are limited and discrete then they will repeat, often with high counts. Due to all these properties, the `dept` can be and should be converted to the categorical data type.

Converting to categorical with the `astype` method

The simplest way to convert a Series to categorical is to pass the string ‘category’ to the `astype` method. We assign this new Series to the `dept_cat` variable name.

```
[11]: dept_cat = dept.astype('category')
dept_cat.head()
```

```
[11]: 0          Police
      1          Other
      2    Houston Public Works
      3          Police
      4          Police
Name: dept, dtype: category
Categories (9, object): [Fire, Health & Human Services, Houston Airport System, Houston
Public Works, ..., Other, Parks & Recreation, Police, Solid Waste Management]
```

Visual display of a categorical Series

The output looks a bit different than a normal Series. Let’s verify that we do indeed have a Series.

```
[12]: type(dept_cat)
```

```
[12]: pandas.core.series.Series
```

The index and the values of our new categorical Series appear identical to the output of a Series that has the object data type. The difference is the appearance of the unique categories below the Series.

The formal data type

Let’s verify that the data type is categorical by accessing the `dtype` attribute.

```
[13]: dept_cat.dtype
```

```
[13]: CategoricalDtype(categories=['Fire', 'Health & Human Services', 'Houston Airport
System',
      'Houston Public Works', 'Library', 'Other',
      'Parks & Recreation', 'Police', 'Solid Waste Management'],
ordered=False)
```

What is CategoricalDtype?

The formal pandas object for categorical data is `CategoricalDtype`. This can be confusing since pandas also uses the string ‘category’ in the Series output. But this is no different than what we saw with the other pandas-only data types. For instance, nullable integers can use either the string ‘Int64’ or the pandas object `pd.Int64Dtype()`.

30.3 Why the categorical data type is useful

On the surface, our categorical Series looks very similar to its object Series counterpart, but there are some significant differences.

Internal storage of categorical data

Categorical data is stored much more efficiently than object data. Each unique value in a column of categorical data is stored **once** regardless of how many times it repeats in the Series and each of the unique values has an integer code that refers to it. It is these integers that are stored in memory to represent the data.

Object columns store each value in a unique location in memory. For instance, the string ‘Police’ appears over 7,000 times in the `dept` Series. Each one of these strings is stored in a unique location in memory. Using integers to represent categories can save a tremendous amount of memory.

Example of categorical storage

Let’s create a simplified example to show how pandas stores categorical data internally using Python lists. In this example, we’ll have three unique departments. They are stored exactly once in the list `cats` below. The actual data is stored in the `vals` list containing the values 0, 1, and 2.

```
[14]: cats = ['Police', 'Fire', 'Library']
       vals = [1, 1, 0, 2, 0, 1, 2, 2, 1, 2, 1]
```

The `cats` list acts as a mapping from integer location to string value. The integer 0 corresponds with ‘Police’, 1 with ‘Fire’, and 2 with ‘Library’. We can convert each value in the `vals` list to its corresponding category using the list comprehension below.

```
[15]: [cats[val] for val in vals]
```

```
[15]: ['Fire',
       'Fire',
       'Police',
       'Library',
       'Police',
       'Fire',
       'Library',
       'Library',
       'Fire',
       'Library',
       'Fire']
```

30.4 The cat accessor

We previously covered the `str` and `dt` accessors which provide us special access to string-only and datetime-only attributes and methods. The `cat` accessor provides us with special attributes and methods for categorical Series. Let's take a look at some of the important attributes and methods it provides.

Get the categories

The unique sequence of categories can be retrieved with the `categories` attribute.

```
[16]: dept_cat.cat.categories
```

```
[16]: Index(['Fire', 'Health & Human Services', 'Houston Airport System',
           'Houston Public Works', 'Library', 'Other', 'Parks & Recreation',
           'Police', 'Solid Waste Management'],
           dtype='object')
```

Get the integer codes

The underlying integer codes may be retrieved with the `codes` attribute which returns a Series the same length as the original. Notice that it uses 8-bit integers to store the data.

```
[17]: dept_cat.cat.codes.head()
```

```
[17]: 0    7
      1    5
      2    3
      3    7
      4    7
dtype: int8
```

Verify codes correspond with categories

The first three values of the `dept_cat` Series are ‘Police’, ‘Other’, and ‘Houston Public Works’. Let's verify that the first three codes correspond with the categories.

```
[18]: dept_cat.cat.categories[7]
```

```
[18]: 'Police'
```

```
[19]: dept_cat.cat.categories[5]
```

```
[19]: 'Other'
```

```
[20]: dept_cat.cat.categories[3]
```

```
[20]: 'Houston Public Works'
```

30.5 Modifying categories

When you create a column of categorical data, the categories are semi-permanent and take some additional work to modify. In this section, you'll learn how to modify, add, and remove categories.

All categories remain after subset selection

The number of categories remain the same after a subset selection, even when some of the categorical values do not appear in the result. Let's select the first five values of the `dept_cat` Series, assign it to the variable `dept_cat_5` and output the result.

```
[21]: dept_cat_5 = dept_cat.head()
dept_cat_5
```

```
[21]: 0          Police
      1          Other
      2  Houston Public Works
      3          Police
      4          Police
Name: dept, dtype: category
Categories (9, object): [Fire, Health & Human Services, Houston Airport System, Houston
Public Works, ..., Other, Parks & Recreation, Police, Solid Waste Management]
```

Notice that there are still 9 categories, even though there are only three unique values in the result. The categories will not change unless you explicitly run a command to do so. If you'd like to clean up your new column of data, you can run the `remove_unused_categories` method from the `cat` accessor. This does not work in-place, so you'll need to assign the result to variable to keep the changes.

```
[22]: dept_cat_5.cat.remove_unused_categories()
```

```
[22]: 0          Police
      1          Other
      2  Houston Public Works
      3          Police
      4          Police
Name: dept, dtype: category
Categories (3, object): [Houston Public Works, Other, Police]
```

Assigning a new category

Attempting to assign a new value that isn't one of the current categories will result in an error. Here, we attempt to assign the first value to 'Information Technology', which is not a current category.

```
[23]: dept_cat.loc[0] = 'Information Technology'
```

```
ValueError: Cannot setitem on a Categorical with a new category, set the
→categories first
```

The error instructs us to add a new category. This is done by passing a single category or list of categories to the `add_categories` method from the `cat` accessor. Let's complete the operation and output the number of categories.

```
[24]: dept_cat = dept_cat.cat.add_categories('Information Technology')
len(dept_cat.cat.categories)
```

```
[24]: 10
```

Let's select the last category to verify it is the one we added.

```
[25]: dept_cat.cat.categories[-1]
```

```
[25]: 'Information Technology'
```

We can now successfully make the assignment and output the first few rows to verify it.

```
[26]: dept_cat.loc[0] = 'Information Technology'
dept_cat.head(3)
```

```
[26]: 0    Information Technology
      1          Other
      2    Houston Public Works
Name: dept, dtype: category
Categories (10, object): [Fire, Health & Human Services, Houston Airport System, Houston
Public Works, ..., Parks & Recreation, Police, Solid Waste Management, Information
Technology]
```

Removing categories that exist

In the rare event that you'd like to remove categories that exist as values in your column, do so with the `cat` accessor's `remove_categories` method. They will be replaced with missing values.

```
[27]: dept_cat.cat.remove_categories('Police').head()
```

```
[27]: 0    Information Technology
      1          Other
      2    Houston Public Works
      3          NaN
      4          NaN
Name: dept, dtype: category
Categories (9, object): [Fire, Health & Human Services, Houston Airport System, Houston
Public Works, ..., Other, Parks & Recreation, Solid Waste Management, Information
Technology]
```

Missing values are not categories

There is no separate category for missing values. The categorical data type uses the numpy `NaN` as its missing value representation. All missing value operations will work as normal. Here we calculate the total number of missing values after removing the 'Police' category.

```
[28]: dept_cat.cat.remove_categories('Police').isna().sum()
```

```
[28]: 7572
```

30.6 Massive reduction in memory used

One of the biggest benefits of using categorical columns is the amount of memory saved. Instead of using a string for every value, an integer code is used. Integers take up significantly less space than strings. pandas also uses the smallest integer size to store the codes. For instance, if there are less than 128 categories,

an int8 is used. pandas has chosen not to use unsigned integers for code storage, so only half the normal capacity is available.

The `memory_usage` method

pandas provides the `memory_usage` method to return the number of bytes used by the Series. To get the exact memory for string columns, you need to set the parameter `deep` to `True`. Let's get the original amount of memory used.

```
[29]: orig_mem = dept.memory_usage(deep=True)  
orig_mem
```

```
[29]: 1643103
```

Use the method again to get the memory used on our categorical Series.

```
[30]: cat_mem = dept_cat.memory_usage(deep=True)  
cat_mem
```

```
[30]: 25475
```

Let's find the percentage reduction in memory used by converting to categorical.

```
[31]: 1 - cat_mem / orig_mem
```

```
[31]: 0.984495798498329
```

An astounding 98.4% reduction in memory takes place. Using 8-bit integers instead of the entire string made a huge difference.

30.7 Speeding up operations

Another nice benefit of using the categorical data type is the increase in performance for most operations. Let's cover a few examples that show this performance improvement.

Comparison operators

The comparison operators should complete much faster. In the example below, we are testing equality of each value to the string 'Police'. Using the categorical Series results in around 10 times better performance.

```
[32]: %timeit -n 5 dept == 'Police'
```

```
2.36 ms ± 454 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

```
[33]: %timeit -n 5 dept_cat == 'Police'
```

```
134 µs ± 30.1 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

This is equivalent to checking whether the `codes` integer Series equals 7, the category number for police.

```
[34]: %timeit -n 5 dept_cat.cat.codes == 7
```

```
276 µs ± 78.6 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

Sorting

Sorting also executes much faster as it's only necessary to sort the unique values.

```
[35]: %timeit -n 5 dept.sort_values()
```

13.3 ms ± 803 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

```
[36]: %timeit -n 5 dept_cat.sort_values()
```

1.6 ms ± 92.7 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

Most other operations

Most other operations happen faster. Here, we see the difference when using the `value_counts` method.

```
[37]: %timeit -n 5 dept.value_counts()
```

2.46 ms ± 12.2 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

```
[38]: %timeit -n 5 dept_cat.value_counts()
```

979 µs ± 91.8 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

30.8 The str accessor is still available

Even though we've converted to the categorical data type, the `str` accessor is still available to use as long as the original data contained strings. Here, we make all the values uppercase.

```
[39]: dept_cat.str.upper().head()
```

```
[39]: 0    INFORMATION TECHNOLOGY
      1          OTHER
      2    HOUSTON PUBLIC WORKS
      3          POLICE
      4          POLICE
Name: dept, dtype: object
```

Unfortunately, an object Series is returned for all the `str` accessor methods that return strings, so you'll have to convert it again to categorical after the operation completes if you want to keep it as a categorical.

30.9 Ordered categories

Generally speaking, there are two types of categorical data, **nominal** and **ordinal**. For nominal categorical data, the values have no natural ordering. With ordinal data, the values do have a natural ordering. To help remember, both 'ordinal' and 'order' begin with 'ord'.

The departments from above are a good example of nominal data, as no department has any natural precedence over the other. There is no good example in our employee dataset of ordinal data. Let's read in the diamonds dataset, which has information on the size, cut, color, clarity, and price for many diamonds.

```
[40]: diamonds = pd.read_csv('../data/diamonds.csv')
diamonds.head(3)
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31

```
[41]: diamonds.shape
```

[41]: (53940, 10)

If you are familiar with diamonds, then you know that cut, color, and clarity have a specific ordering corresponding to the quality of that property. Let's take a look at the clarity column's unique values.

```
[42]: clarity = diamonds['clarity']
clarity.value_counts()
```

```
[42]: SI1      13065
VS2      12258
SI2      9194
VS1      8171
VVS2     5066
VVS1     3655
IF       1790
I1       741
Name: clarity, dtype: int64
```

Before creating an ordered categorical, let's convert clarity to an unordered categorical like we did above. Passing the `astype` method the string 'category' always creates an unordered categorical data type.

```
[43]: clarity_cat = diamonds['clarity'].astype('category')
clarity_cat.head()
```

```
[43]: 0    SI2
1    SI1
2    VS1
3    VS2
4    SI2
Name: clarity, dtype: category
Categories (8, object): [I1, IF, SI1, SI2, VS1, VS2, VVS1, VVS2]
```

We can use the `ordered` attribute available from the `cat` accessor to verify that it is unordered.

```
[44]: clarity_cat.cat.ordered
```

[44]: False

The data dictionary contains information on the ordering of each categorical variable. Let's read it in, changing the column width option `display` so that we can read the entirety of the description.

```
[45]: pd.set_option('display.max_colwidth', 100)
pd.read_csv('../data/dictionaries/diamonds_dictionary.csv')
```

Column Name	Description
0 carat	weight of the diamond (0.2--5.01)
1 clarity	a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))
2 color	diamond colour, from J (worst) to D (best)
3 cut	quality of the cut (Fair, Good, Very Good, Premium, Ideal)
4 depth	total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43--79)
5 price	price in US dollars (\$326--\$18,823)
6 table	width of top of diamond relative to widest point (43--95)
7 x	length in mm (0--10.74)
8 y	width in mm (0--58.9)
9 z	depth in mm (0--31.8)

Creating an ordered categorical

Creating an ordered categorical column takes a bit of work. Here are the three steps that you need to take:

1. Create a list of all of the unique categories in the order that you desire.
2. Use the `CategoricalDtype` constructor available directly from pd. Pass it the list of categories and set the `ordered` parameter to `True`. Assign this result to a variable name.
3. Pass the variable name in step 2 to the `astype` Series method.

Here, we complete the three steps and assign the new ordered categorical Series to `clarity_ordered_cat`.

```
[46]: cats = ['I1', 'SI2', 'SI1', 'VS2', 'VS1', 'VVS2', 'VVS1', 'IF']
clarity_dtype = pd.CategoricalDtype(cats, ordered=True)
clarity_ordered_cat = clarity.astype(clarity_dtype)
clarity_ordered_cat.head()
```

```
[46]: 0    SI2
      1    SI1
      2    VS1
      3    VS2
      4    SI2
Name: clarity, dtype: category
Categories (8, object): [I1 < SI2 < SI1 < VS2 < VS1 < VVS2 < VVS1 < IF]
```

Notice that the categories appear at the bottom with less than signs separating them indicating the order. Let's verify the data type of our new Series.

```
[47]: clarity_ordered_cat.dtype
```

```
[47]: CategoricalDtype(categories=['I1', 'SI2', 'SI1', 'VS2', 'VS1', 'VVS2', 'VVS1', 'IF'],
ordered=True)
```

From the end of the output, you can see that it is ordered. Let's also verify this with the `ordered` attribute.

```
[48]: clarity_ordered_cat.cat.ordered
```

```
[48]: True
```

Special properties of ordered categorical Series

Ordered categorical Series have some behavior that differs from their unordered counterparts. For instance, taking the maximum or minimum value returns the category ranked as the best or worst and is not based on alphabetical ordering.

```
[49]: clarity_ordered_cat.max()
```

```
[49]: 'IF'
```

Attempting to call the `max` method on an unordered categorical raises an error.

```
[50]: clarity_cat.max()
```

```
TypeError: Categorical is not ordered for operation max  
you can use .as_ordered() to change the Categorical to an ordered one
```

The original object Series does work with the `max` method and returns the value with greatest alphabetical value, which is different than the best measurement for clarity.

```
[51]: clarity.max()
```

```
[51]: 'VVS2'
```

Sorting ordered categoricals is done by their category order and not alphabetically. Here we call the `value_counts` method on both the ordered and unordered clarity Series and then sort the index.

```
[52]: clarity_ordered_cat.value_counts().sort_index()
```

```
[52]: I1      741  
SI2     9194  
SI1    13065  
VS2    12258  
VS1     8171  
VVS2    5066  
VVS1    3655  
IF      1790  
Name: clarity, dtype: int64
```

Sorting the unordered categorical index sorts alphabetically.

```
[53]: clarity_cat.value_counts().sort_index()
```

```
[53]: I1      741  
IF      1790  
SI1    13065
```

```

SI2      9194
VS1      8171
VS2      12258
VVS1     3655
VVS2     5066
Name: clarity, dtype: int64

```

30.10 Integers can be categories

Any column, regardless of its data type, may be converted to categorical and not just strings. If the values are known, limited, and discrete, then they are good candidates for categorical data. Integers are the primary non-string data type that represent categorical data. Here are some examples of integer categorical data:

- Rating of a movie/hotel/restaurant given that the range is known such as the integers (1-5)
- Zip codes for a particular city
- Hurricane strength category (1-5)

As with our string columns, integer categorical columns may be unordered (like zip codes) or ordered (like movie ratings). Let's read in just two columns of the housing dataset.

```
[54]: housing = pd.read_csv('../data/housing.csv', usecols=['MSSubClass', 'SalePrice'])
housing.head()
```

	MSSubClass	SalePrice
0	60	208500
1	20	181500
2	60	223500
3	70	140000
4	60	250000

By default, these columns are read in as integers. Looking at the data dictionary, 'MSSubClass' is a candidate for categorical as it identifies the type of dwelling. Here are a few of the integer codes and their corresponding description. There is no inherent order for these values.

- 20 - 1-STORY 1946 & NEWER ALL STYLES
- 30 - 1-STORY 1945 & OLDER
- 40 - 1-STORY W/FINISHED ATTIC ALL AGES
- 45 - 1-1/2 STORY - UNFINISHED ALL AGES
- 50 - 1-1/2 STORY FINISHED ALL AGES

30.11 The new string data type

With the release of pandas 1.0, a new pandas-only data type called 'string' was made available. It can only contain strings and missing values. Again, avoid using this data type for serious work until it is no longer experimental.

Use the string 'string' to create this data type in the Series constructor or with the `astype` method. You can also use the pandas object `pd.StringDtype` directly. Both Series below are identical.

```
[55]: s_string = pd.Series(['Police', 'Fire', 'Police', pd.NA], dtype='string')
s_string = pd.Series(['Police', 'Fire', 'Police', pd.NA], dtype=pd.StringDtype())
s_string
```

```
[55]: 0    Police
      1     Fire
      2    Police
      3    <NA>
dtype: string
```

Similarities and differences between string and object data types

The intended purpose of the string data type is to finally provide pandas users with a data type that is guaranteed to only contain strings (and missing values). This should reduce errors as the object data type is capable of containing anything. That said, the functionality between the two data types is going to be similar. Let's create an object Series from our string Series using the `astype` method.

```
[56]: s_object = s_string.astype('object')
s_object
```

```
[56]: 0    Police
      1     Fire
      2    Police
      3    <NA>
dtype: object
```

One of the major differences between the two data types is how they handle boolean selection (filtering). When containing missing values, this is impossible with the string data type.

```
[57]: s_string[s_string == 'Police']
```

```
[57]: 0    Police
      2    Police
dtype: string
```

However, it is possible with the old object data type.

```
[58]: s_object[s_object == 'Police']
```

```
[58]: 0    Police
      2    Police
dtype: object
```

The `str` accessor is available for the string data type and contains all the same methods.

```
[59]: s_string.str.upper()
```

```
[59]: 0    POLICE
      1     FIRE
      2    POLICE
      3    <NA>
dtype: string
```

30.12 Converting strings to numeric

It is possible to convert strings consisting entirely of numerical characters to either integer or float. Let's construct a Series of strings that look just like floats. It will default as the object data type.

```
[60]: s = pd.Series(['4.5', '3.19'])
s
```

```
[60]: 0    4.5
      1    3.19
      dtype: object
```

Notice that the quotation marks are not present for strings in the visual display of the DataFrame in the notebook, so they appear to be floats. But, you'll also notice that the decimals are not aligned one on top of the other and each value has a different number of places. This isn't the normal display for actual float columns. Let's create an actual float column (with the same values) so you can see the difference in the visual display. Let's make the conversion to an actual float data type with the `astype` method. Notice that the decimals will always align.

```
[61]: s.astype('float64')
```

```
[61]: 0    4.50
      1    3.19
      dtype: float64
```

30.13 Force conversion with `pd.to_numeric`

You may have a Series of string values where some can be converted to a numeric and others that cannot. In this situation, it is not possible to use the `astype` method to make the conversion, as you can see with the following error.

```
[62]: s = pd.Series(['4.5', '3.19', 'NOT AVAILABLE'])
s
```

```
[62]: 0        4.5
      1        3.19
      2    NOT AVAILABLE
      dtype: object
```

```
[63]: s.astype('float64')
```

```
ValueError: could not convert string to float: 'NOT AVAILABLE'
```

Instead, you must turn to the `to_numeric` function, which works similarly to `astype`, but has an option to force the conversion to happen. You do this by setting the `errors` parameter to the string 'coerce'. Any value that cannot be converted will be set as missing.

```
[64]: pd.to_numeric(s, errors='coerce')
```

```
[64]: 0    4.50
      1    3.19
```

```
2      NaN
dtype: float64
```

Notice that `to_numeric` is a function and not a method. You must access it directly from `pd`. The `astype` method does have an `errors` parameter, but it does not have the option for ‘coerce’. It would be quite nice if the developers implemented this option for `astype`, then we wouldn’t need to use `to_numeric`.

Converting to strings

You can convert all the values to a string with either the string ‘str’ or the built-in `str` class. Let’s create a Series of integers and then convert it to strings. The technical data type will be object.

```
[65]: s = pd.Series([10, 20, 99])
s.astype('str')
```

```
[65]: 0    10
1    20
2    99
dtype: object
```

Let’s verify that the underlying values are actually strings by accessing the `values` attribute. A numpy array is returned, and uses quote marks for its visual display of string data.

```
[66]: s.astype('str').values
```

```
[66]: array(['10', '20', '99'], dtype=object)
```

30.14 Object, String, and Categorical data type summary

Generic Name	String Name of Default	All Bit Sizes	Notes
Object	object	any	May contain any Python object
String	string	any	Only contains strings
Category	category	Smallest integer capable of holding all categories	Use <code>pd.CategoricalDtype()</code> to create ordinal categories

30.15 Exercises

30.16 2. Object, String, and Categorical Data Types

Exercise 1

Using its constructor, create a Series containing three two-item lists of integers. Then call the `sum` method on the Series. What is returned?

```
[ ]:
```

Exercise 2

Use the constructor to create a Series of integers, floats, and booleans. Do not set the `dtype` parameter. What data type is your Series?

[]:

Exercise 3

Construct a Series with the same values but force the data type to be a float. Does it work? What happens to the non-float values?

[]:

Exercise 4

Construct a Series containing three strings and the four missing values `None`, `np.nan`, `pd.NA`, and `pd.NaT` assigning the result to a variable.

[]:

Exercise 5

Using pandas, count the number of missing values in exercise 4.

[]:

Exercise 6

Convert the Series from exercise 4 to the new string data type. Notice what happens to the missing values.

[]:

Read in the movie dataset

Execute the cell below to read in the first 10 columns of the movie dataset setting the index to be the title.

```
[67]: pd.set_option('display.max_columns', 100)
movie = pd.read_csv('../data/movie.csv', index_col='title', usecols=range(10))
movie.head(3)
```

	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb	actor2
title									
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0	Joel David Moore
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	Johnny Depp	40000.0	Orlando Bloom
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	Christoph Waltz	11000.0	Rory Kinnear

Exercise 8

Which of the columns above are good candidates for the categorical data type?

[]:

Exercise 9

Select the `content_rating` column as a Series and convert it to categorical. Assign the result to the variable `rating`.

[]:

Exercise 10

Write an expression that returns the number of categories.

[]:

Exercise 11

Prove that the `str` accessor still works with categorical columns by making the ratings lowercase.

[]:

Exercise 12

Assign the rating ‘GGG’ as the first value.

[]:

Exercise 13

Convert the following Series to integer.

[68]:

```
s = pd.Series(['1', '2'])
```

Exercise 14

Convert the following Series to integer.

[69]:

```
s = pd.Series(['1', '2', 'BAD DATA'])
```

Read in the diamonds dataset

Execute the next cell to read in the diamonds dataset.

[70]:

```
diamonds = pd.read_csv('../data/diamonds.csv')
diamonds.head(3)
```

	carat	cut	color	clarity	depth	table	price	x	y	z
0	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31

Exercise 15

Select the `cut` column as a Series and convert it to an ordered categorical. Use the data dictionary from above.

[]:

Exercise 16

By only knowing that `cut_cat` is an ordered categorical, write an expression to get the percentage of diamonds that have the lowest category.

[]:

Chapter 31

Datetime, Timedelta, and Period Data Types

In this chapter, we will cover the datetime, timedelta, and period data types, which all relate to time, but are completely independent from one another. This chapter will not go into usage of these data types within the context of a data analysis - that is left to the chapters in part 7, Time Series. We will focus on learning what these data types are and how they are constructed in an array or a pandas Series.

- **Datetime** - One single moment in time with nanosecond precision. It always contains both a date (year, month, day) and a time (hour, minute, second, part of second) component. Example - June 8, 1989 10:45 AM
- **Timedelta** - An amount of time with nanosecond precision. It has units of days, hours, minutes, seconds, and parts of a second. It is not attached to any date. Example - 5 hours, 34 minutes, 2.89 seconds
- **Period** - A span of time. There is a start and end of a period. The units depend on the length of the time period. Example - June 8, 1989 - The start time would be June 8, 1989 at midnight and end time would be one nanosecond before midnight of June 9, 1989.

31.1 The numpy datetime64 data type

Both numpy and pandas have a ‘datetime64’ data type. pandas uses numpy’s datetime64 data type as a base and builds quite a bit more functionality on top of it. As the name implies, a datetime64 value always uses 64 bits of memory. There is no other size for datetimes other than 64 bits. However, a datetime64 object must have a date or time **unit**. The units can be years, months, weeks, days, hours, minutes, seconds, and parts of a second up to an attosecond (10-18 of a second).

The unit determines the precision of a datetime value. For instance, if the unit is months then each datetime will have a year and month component. If the unit is hours then each datetime will have year, month, day, and hour components.

The official string representation of datetime64 data types must contain the units and is placed within brackets following the word ‘datetime’. For instance, `datetime64[s]` is the official representation for a datetime64 object with second precision and `datetime64[ns]` has nanosecond precision. Visit the [numpy documentation to view all of the possible units](#).

There are a few ways to create a datetime array in numpy. Just as we did above, we will pass the `dtype` parameter the precise data type we desire. The values passed to `np.array` must be integers. numpy converts these integers to a datetime with the specified unit. It does this by treating 0 as the **unix epoch** which is January 1, 1970 at midnight. In the following example, we create an array from the three integers 10, -120, and 410. The data type is a datetime with month precision (referenced by the string ‘M’) . The integer 10

corresponds to 10 months after the epoch or November, 1970. The integer -120 corresponds to 120 months (10 years) before the epoch or January 1960 and the last represents 410 months after the epoch.

```
[1]: import numpy as np
import pandas as pd
np.array([10, -120, 410], dtype='datetime64[M]')
```

```
[1]: array(['1970-11', '1960-01', '2004-03'], dtype='datetime64[M]')
```

Let's use second precision with the same integers. Now, the first value corresponds to 10 seconds after the epoch. Notice how the last unit in the output is seconds. A capital 'T' separates the date and time components.

```
[2]: np.array([10, -120, 410], dtype='datetime64[s]')
```

```
[2]: array(['1970-01-01T00:00:10', '1969-12-31T23:58:00',
       '1970-01-01T00:06:50'], dtype='datetime64[s]')
```

In this final example, hour precision is used with the same data. Notice again in the output how the precision shows only up to hours.

```
[3]: np.array([10, -120, 410], dtype='datetime64[h]')
```

```
[3]: array(['1970-01-01T10', '1969-12-27T00', '1970-01-18T02'],
       dtype='datetime64[h]')
```

Available integers

You can use all integers that are available to 64-bit integers. Let's print this info out using the `iinfo` function.

```
[4]: np.iinfo('int64')
```

```
[4]: iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

Available time span

The precision of the datetime will limit its available time span. For instance, the very highest datetime possible will be with the maximum 64-bit integer which is 9223372036854775807 ($2^{63} - 1$). Let's convert the min and max 64-bit integers to datetimes with millisecond precision.

```
[5]: np.array([-9223372036854775808, 9223372036854775807], dtype='datetime64[ms]')
```

```
[5]: array([
       'NaT', '292278994-08-17T07:12:55.807'],
       dtype='datetime64[ms]')
```

```
[6]: np.array([-9223372036854775807, 9223372036854775807], dtype='datetime64[as]')
```

```
[6]: array(['1969-12-31T23:59:50.776627963145224193',
       '1970-01-01T00:00:09.223372036854775807'],
       dtype='datetime64[as]')
```

NaT

Notice that the first value returned as ‘NaT’ which stands for ‘Not a Time’. Instead of using the minimum integer as a datetime, numpy uses it to signal a missing value. A value such as this is usually referred to as a **sentinel value**, a special reserved value for a specific situation. The minimum 64-bit integer is not available to be used as a normal datetime.

Finding the real available time span

Let’s use the second lowest integer instead to find the actual available timespan.

```
[7]: np.array([-9223372036854775807, 9223372036854775807], dtype='datetime64[ms]')
```

```
[7]: array(['-292275055-05-16T16:47:04.193', '292278994-08-17T07:12:55.807'],
      dtype='datetime64[ms]')
```

When allowing for millisecond precision, we are able to use dates between 292 million years ago to 292 million years in the future.. If we require nanosecond precision, then our available timespan reduces dramatically such that it begins on September 21, 1677 and ends on April 11, 2262. Any datetime outside of that range will not be able to be represented with nanosecond precision.

```
[8]: np.array([-9223372036854775807, 9223372036854775807], dtype='datetime64[ns]')
```

```
[8]: array(['1677-09-21T00:12:43.145224193', '2262-04-11T23:47:16.854775807'],
      dtype='datetime64[ns]')
```

31.2 The pandas datetime64 data type

pandas datetime64 data type is very similar to numpy’s, but not quite the same. One major difference is with its precision. pandas datetime64 is only available with **nanosecond** precision. Let’s create a pandas Series from a numpy datetime array that has month precision.

```
[9]: a = np.array([10, -120, 410], dtype='datetime64[M]')
a
```

```
[9]: array(['1970-11', '1960-01', '2004-03'], dtype='datetime64[M]')
```

We take this array and pass it to the Series constructor.

```
[10]: s = pd.Series(a)
s
```

```
[10]: 0    1970-11-01
1    1960-01-01
2    2004-03-01
dtype: datetime64[ns]
```

Notice that the Series data type has nanosecond precision even though it was created from a numpy array with month precision. You might be wondering why the hour, minute, second, and nanoseconds are not viewable in the above Series output. These components do exist, but pandas intelligently does not output them as showing lots of zeros would dilute the information. You can view the underlying numpy array to verify that the nanosecond precision exists. A nanosecond is one-billionth of a second, which is in the 9th decimal place.

```
[11]: s.values
```

```
[11]: array(['1970-11-01T00:00:00.000000000', '1960-01-01T00:00:00.000000000',
           '2004-03-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

Converting integer Series to datetime

You can convert a Series of integers to a datetime with the `astype` method. Let's first create a Series of integers.

```
[12]: s = pd.Series([0, 4, 30])
s
```

```
[12]: 0      0
      1      4
      2     30
dtype: int64
```

We convert the Series to a datetime with year precision. As always, the resulting precision will always be in nanoseconds.

```
[13]: s.astype('datetime64[Y]')
```

```
[13]: 0    1970-01-01
      1    1974-01-01
      2    2000-01-01
dtype: datetime64[ns]
```

Use months as the units which again eventually converts to nanoseconds.

```
[14]: s.astype('datetime64[M]')
```

```
[14]: 0    1970-01-01
      1    1970-05-01
      2    1972-07-01
dtype: datetime64[ns]
```

Using strings to construct datetime Series

You can pass strings of the format 'YYYY-MM-DD hh-mm-ss' (and add parts of second as decimals following seconds) to the Series constructor setting the `dtype` parameter to 'datetime64[ns]'.

```
[15]: pd.Series(['2001-10-01', '2022-01-09 14:29:33.51'], dtype='datetime64[ns]')
```

```
[15]: 0    2001-10-01 00:00:00.000
      1    2022-01-09 14:29:33.510
dtype: datetime64[ns]
```

It's possible to create a Series with missing values by using either the string 'NaT' or the actual pandas object `pd.NaT`.

```
[16]: pd.Series(['2001-10-01', '2022-01-09 14:29', 'NaT', pd.NaT], dtype='datetime64[ns]')
```

```
[16]: 0    2001-10-01 00:00:00
      1    2022-01-09 14:29:00
      2                NaT
      3                NaT
dtype: datetime64[ns]
```

More ways to create datetimes

There are more ways to create datetimes, such as with `pd.Timestamp`, `pd.to_datetime`, and `pd.date_range`. These methods will be discussed in the Time Series part.

31.3 The numpy timedelta64 data type

A timedelta refers to an amount of time like 4 days and 24 minutes or 123 milliseconds. In numpy, a timedelta is expressed as an integer along with a unit ranging from years to attoseconds with the same character abbreviation as datetimes. Let's create a numpy array of `timedelta64[D]` values. The D represents day precision.

```
[17]: np.array([1, 2, 100], dtype='timedelta64[D]')
```

```
[17]: array([ 1,  2, 100], dtype='timedelta64[D]')
```

The output for numpy timedeltas will only ever be integers. If you try and use a float to represent a portion of time, then your values will be truncated.

```
[18]: np.array([1.2, 2.7]).astype('timedelta64[Y]')
```

```
[18]: array([1, 2], dtype='timedelta64[Y]')
```

Because of this, you'll need to reduce your amount of time to the lowest available unit if you want to use numpy's timedelta. For instance, if you want to create a timedelta of 5 hours, 34 minutes, and 17 seconds, you'd need to convert to seconds ($5 * 3600 + 34 * 60 + 17$).

```
[19]: 5 * 3600 + 34 * 60 + 17
```

```
[19]: 20057
```

```
[20]: np.array([20057], dtype='timedelta64[s]')
```

```
[20]: array([20057], dtype='timedelta64[s]')
```

31.4 The pandas timedelta64 data type

The pandas timedelta64 data type is more intuitive to use than numpy's and has more features. In pandas, all timedeltas have nanosecond precision. You are not given a choice. Below, we convert a Series of integers to timedeltas. We specify the unit as minutes ('m'), but pandas will eventually convert this to nanosecond precision.

```
[21]: pd.Series([10, 50, 423]).astype('timedelta64[m]')
```

```
[21]: 0    00:10:00
      1    00:50:00
      2    07:03:00
dtype: timedelta64[ns]
```

Take a look at the value 423. We are telling pandas to treat this as 423 minutes, which is 7 hours, 3 minutes, 0 seconds, and 0 nanoseconds. This is the value that is returned. Let's use the same Series of integers and use hours as the units.

```
[22]: s = pd.Series([10, 50, 423]).astype('timedelta64[h]')
s
```

```
[22]: 0    0 days 10:00:00
      1    2 days 02:00:00
      2   17 days 15:00:00
dtype: timedelta64[ns]
```

To prove that pandas uses nanosecond precision, view the underlying array.

```
[23]: s.values
```

```
[23]: array([ 360000000000000, 180000000000000, 1522800000000000000],
          dtype='timedelta64[ns]')
```

Even though numpy allows timedeltas to have year precision, the largest unit used in the representation of timedeltas within pandas is days. The following Series still has nanosecond precision, but the visual representation is shown in days. pandas does not use years in its representation as a year is not a consistent measure of time. One year from now could mean 365 or 366 days. Similarly, months are also not consistent measures of time. Days are the largest unit of time that have a consistent measure (always 24 hours), so this is what pandas has chosen to represent timedeltas that span more than 24 hours.

```
[24]: pd.Series([1, 10, 50]).astype('timedelta64[Y]')
```

```
[24]: 0    365 days 05:49:12
      1   3652 days 10:12:00
      2  18262 days 03:00:00
dtype: timedelta64[ns]
```

31.5 The pandas Period data type

The period data type is unique to pandas and does not exist in numpy. To construct a Series with data type period, use a list of strings with precision up to the unit you desire. Below, we construct a Series with three strings that have a year and month component, but no further precision. In the constructor, we set the `dtype` parameter to `period[M]` where the 'M' represents month precision.

```
[25]: s = pd.Series(['2000-10', '2002-06', '2010-08'], dtype='period[M]')
s
```

```
[25]: 0    2000-10
      1    2002-06
      2    2010-08
dtype: period[M]
```

The first value, 2000-10 represents the entire month of October, 2000. Technically, this is from October 1, 2000 at midnight until one nanosecond before midnight of November 1, 2000. The other values each represent an entire month of time.

It is also possible to convert Series of strings to the period data type with the `astype` method in a similar fashion.

```
[26]: pd.Series(['2000-10', '2002-06', '2010-08']).astype('period[M]')
```

```
[26]: 0    2000-10
      1    2002-06
      2    2010-08
dtype: period[M]
```

However, you cannot convert a datetime Series to a period with the `astype` method. Below, we create a two-item Series of datetimes.

```
[27]: s = pd.Series(['2001-10-15', '2002-01-09 14:29:33.51'], dtype='datetime64[ns]')
s
```

```
[27]: 0    2001-10-15 00:00:00.000
      1    2002-01-09 14:29:33.510
dtype: datetime64[ns]
```

Attempting to convert to a month period fails with the `astype` method.

```
[28]: s.astype('period[M]')
```

```
DateParseError: signed integer is greater than maximum
```

However, you can make the conversion with the `to_period` method found under the `dt` accessor.

```
[29]: s.dt.to_period('M')
```

```
[29]: 0    2001-10
      1    2002-01
dtype: period[M]
```

31.6 Datetime, Timedelta, and Period data type summary

Generic Name	String Name of Default	All Bit Sizes	Notes
Datetime	datetime64[u]	64	A single moment in time
Timedelta	timedelta64[u]	64	An amount of time
Period	period[u]	64	A span of time

Note - Must specify `u`, the unit of date or time above. Datetime and timedelta columns will be in nanosecond precision.

31.7 Exercises

Exercise 1

Create a numpy array of datetimes with year precision for the years 2000, 2010, and 2020. Assign the result to a variable.

[]:

Exercise 2

Staying in numpy, convert the array created in exercise 1 to a data type with second precision and assign the result to a new variable.

[]:

Exercise 3

Staying in numpy, use the `astype` method to return the number of seconds after the epoch for each value from the array created in exercise 2.

[]:

Exercise 4

Use the integers from exercise 3 to in the numpy array constructor to get the same result as exercise 2.

[]:

Exercise 5

Construct a Series of integers for the years 2000, 2010, and 2020. Then convert it to datetime with the `astype` method.

[]:

Exercise 6

What month is it 1 million minutes after the unix epoch?

[]:

Exercise 7

Construct a datetime Series using strings with precision down to nanoseconds (9 digits after the decimal)

[]:

Exercise 8

Using only arithmetic operations, find the amount of time 1 million seconds is. Report your answer as ‘W days, X hours, Y minutes, Z seconds’.

[]:

Exercise 9

Verify the results of exercise 8 by creating a pandas timedelta Series.

[]:

Exercise 10

Construct a Series with the data type period that has the hour 10 a.m. through 12 a.m. as the time period on January 1st for the years 2019, 2020, and 2021.

[]:

Chapter 32

DataFrame Data Type Conversion

The previous chapters in this part focused on the understanding and construction of data types outside of the context of real data. We mainly used numpy arrays and pandas Series for exploring these data types. In this chapter, we will work with entire DataFrames of real data.

Let's see some examples by reading in a few of the columns from the college dataset. The `usecols` parameter is set to just six columns, one of which, the institution name, is placed in the index.

```
[1]: import pandas as pd
cols = ['instnm', 'hbcu', 'relaffil', 'ugds', 'md_earn_wne_p10', 'grad_debt_mdn_supp']
college = pd.read_csv('../data/college.csv', index_col='instnm', usecols=cols)
college.head(3)
```

		hbcu	relaffil	ugds	md_earn_wne_p10	grad_debt_mdn_supp
	instnm					
	Alabama A & M University	1.0	0	4206.0	30300	33888
	University of Alabama at Birmingham	0.0	0	11383.0	39700	21941.5
	Amridge University	0.0	1	291.0	40100	23370

From the above visual display, it appears that all of the columns are numeric. But, unlike Series, we cannot tell with DataFrames, as each column's specific data type is not presented in the output. You must access them with the `dtypes` attribute.

```
[2]: college.dtypes
```

```
[2]: hbcu           float64
      relaffil        int64
      ugds           float64
      md_earn_wne_p10    object
      grad_debt_mdn_supp    object
      dtype: object
```

You might be surprised to see that the last two columns are read in as objects. They certainly appear to be numeric. If you look closely at the values in the 'grad_debt_mdn_supp' column, you'll notice that the second value has a single decimal value while the others do not. Columns that have the float data type will never have a visual display like this. The decimal places will always align.

When a seemingly numeric column is read in as an object, it is a clue that there are strings in this column. One of the first things we can do to investigate this issue is output some of the values from the underlying numpy array with the `values` attribute. Here, we take a look at the first five values to see that they strings.

```
[3]: college['grad_debt_mdn_supp'].head().values
```

```
[3]: array(['33888', '21941.5', '23370', '24097', '33118.5'], dtype=object)
```

If the `read_csv` function initially reads in a column with the data type object, then you can guarantee that all non-missing values will be strings. We can get the exact type of an individual value by extracting it from the DataFrame and passing it to the built-in `type` function.

```
[4]: val = college.loc['Alabama A & M University', 'grad_debt_mdn_supp']
val
```

```
[4]: '33888'
```

```
[5]: type(val)
```

```
[5]: str
```

The `read_csv` function won't read in a column of data as strings unless it contains non-numeric characters. One way to find non-numeric characters is to sort the string column in descending order. Numeric characters have a lower unicode code point than alphabetic characters, so this should put the alphabetic strings to the top.

Instead of sorting the column itself, we first call the `value_counts` method and then sort the index in descending order to place all the non-numeric values at the very top. This also reveals the frequency of occurrence.

```
[6]: college['grad_debt_mdn_supp'].value_counts().sort_index(ascending=False).head(3)
```

```
[6]: PrivacySuppressed    1510
9999                 1
9998                 1
Name: grad_debt_mdn_supp, dtype: int64
```

This method isn't perfect for uncovering non-numeric strings since it's possible a string can begin with a digit only to be followed by alphabetic characters. Regular expressions (which are covered in an upcoming part) are needed to search for more specific patterns.

Converting non-numeric values to missing

We've uncovered one string, 'PrivacySuppressed', in the 'grad_debt_mdn_supp' column which forced pandas to use object instead of float for its data type. If we want to use this column as a float, we'll need to convert the strings to some numeric value or a missing value.

Let's go ahead and convert the string to a missing value. As we saw previously, we cannot use the `astype` method to make this conversion as the `errors` parameter cannot be set to 'coerce'. We need to use the `to_numeric` function to convert this column to a float. We set the `errors` parameter to 'coerce' to force any value that isn't able to be converted to missing. One minor annoyance is that `to_numeric` converts only a single column at a time. Below, we overwrite both of the object columns (the other column has the same exact issue) with two separate calls of the `to_numeric` function.

```
[7]: college['grad_debt_mdn_supp'] = pd.to_numeric(college['grad_debt_mdn_supp'],  
    ↪errors='coerce')  
college['md_earn_wne_p10'] = pd.to_numeric(college['md_earn_wne_p10'], errors='coerce')  
college.head(3)
```

	hbcu	relaffil	ugds	md_earn_wne_p10	grad_debt_mdn_supp
instnm					
Alabama A & M University	1.0	0	4206.0	30300.0	33888.0
University of Alabama at Birmingham	0.0	0	11383.0	39700.0	21941.5
Amridge University	0.0	1	291.0	40100.0	23370.0

Notice how the decimals align. Let's verify that the new data types are float.

```
[8]: college.dtypes
```

```
[8]: hbcu           float64  
reلافیل        int64  
ugds            float64  
md_earn_wne_p10 float64  
grad_debt_mdn_supp float64  
dtype: object
```

32.1 The astype method for DataFrames

The `astype` method is still useful for DataFrames. We can convert all columns at once to a different type. Below, we convert each column to a 32-bit float and immediately verify the data types.

```
[9]: college.astype('float32').dtypes
```

```
[9]: hbcu           float32  
reلافیل        float32  
ugds            float32  
md_earn_wne_p10 float32  
grad_debt_mdn_supp float32  
dtype: object
```

You can change the data type of specific columns by using a dictionary to map the column name to the desired type. Here, we change `reلافیل` to an 8-bit integer and `ugds` to a 32-bit float.

```
[10]: college.astype({'reلافیل': 'int8', 'ugds': 'float32'}).dtypes
```

```
[10]: hbcu           float64  
reلافیل        int8  
ugds            float32  
md_earn_wne_p10 float64  
grad_debt_mdn_supp float64  
dtype: object
```

32.2 Reading in data with known missing values

You can avoid having to use `to_numeric` if you know the missing value representation in your dataset before you read in your data. Set the `na_values` parameter of the `read_csv` function to the string that represents missing values. You can use a list to specify more values and a dictionary to specify different missing values for each column. Here, we read in our college dataset again and convert every occurrence of ‘PrivacySuppressed’ to missing on read.

```
[11]: college = pd.read_csv('../data/college.csv', index_col='instnm',
                           usecols=cols, na_values='PrivacySuppressed')
college.head(3)
```

	hbcu	relaffil	ugds	md_earn_wne_p10	grad_debt_mdn_supp
instnm					
Alabama A & M University	1.0	0	4206.0	30300.0	33888.0
University of Alabama at Birmingham	0.0	0	11383.0	39700.0	21941.5
Amridge University	0.0	1	291.0	40100.0	23370.0

It appears as though the last two columns were read in as floats. Let’s verify that pandas has correctly read in the last two columns as floats.

```
[12]: college.dtypes
```

```
[12]: hbcu                  float64
      relaffil             int64
      ugds                  float64
      md_earn_wne_p10       float64
      grad_debt_mdn_supp   float64
      dtype: object
```

32.3 More data type conversion with the housing dataset

Let’s see another example of converting data types on DataFrames with the housing dataset. We begin by reading in just seven of the columns by setting the `usecols` parameter to a list of those column names.

```
[13]: cols = ['MSSubClass', 'LotConfig', 'Condition1', 'YearBuilt',
            'ExterQual', 'OverallQual', 'SalePrice']
housing = pd.read_csv('../data/housing.csv', usecols=cols)
housing.head()
```

	MSSubClass	LotConfig	Condition1	OverallQual	YearBuilt	ExterQual	SalePrice
0	60	Inside	Norm	7	2003	Gd	208500
1	20	FR2	Feedr	6	1976	TA	181500
2	60	Inside	Norm	7	2001	Gd	223500
3	70	Corner	Norm	7	1915	TA	140000
4	60	FR2	Norm	8	2000	Gd	250000

Let's see how pandas has chosen to read in this data.

[14]: `housing.dtypes`

```
[14]: MSSubClass      int64
LotConfig        object
Condition1       object
OverallQual     int64
YearBuilt        int64
ExterQual       object
SalePrice        int64
dtype: object
```

By examining the data dictionary, 'MSSubClass', 'LotConfig', and 'Condition1' are nominal categorical data. Both 'OverallQual' and 'ExterQual' are ordinal, and 'YearBuilt' and 'SalePrice' are integers. We use the data dictionary to create a python dictionary mapping each column to a new data type.

Ordered categorical data types must be constructed with the `CategoricalDtype` constructor. We also choose unsigned integers with less bit sizes for the 'YearBuilt' and 'SalePrice' columns. Finally, we pass this dictionary to the `astype` method to make the conversion. It is not necessary to specify every column like we did here. Those not specified retain their data type.

[15]:

```
oq_dtype = pd.CategoricalDtype(range(1, 11), ordered=True)
eq_dtype = pd.CategoricalDtype(['Po', 'Fa', 'TA', 'Gd', 'Ex'], ordered=True)
dtype_dict = {'MSSubClass': 'category',
              'LotConfig': 'category',
              'Condition1': 'category',
              'YearBuilt': 'uint16',
              'SalePrice': 'uint32',
              'OverallQual': oq_dtype,
              'ExterQual': eq_dtype}
housing2 = housing.astype(dtype_dict)
housing2.dtypes
```

```
[15]: MSSubClass      category
LotConfig        category
Condition1       category
OverallQual     category
YearBuilt        uint16
ExterQual       category
SalePrice        uint32
dtype: object
```

This data type conversion may be done on read by using the same dictionary passed to the `dtype` parameter.

```
[16]: housing2 = pd.read_csv('../data/housing.csv', usecols=cols, dtype=dtype_dict)
housing2.dtypes
```

```
[16]: MSSubClass      category
LotConfig        category
Condition1       category
OverallQual      category
YearBuilt         uint16
ExterQual        category
SalePrice         uint32
dtype: object
```

The amount of memory saved by converting to these new data types is substantial with most of it coming from the change to categorical. Let's get the memory usage for both the original and converted datasets.

```
[17]: orig_mem = housing.memory_usage(index=False, deep=True)
orig_mem
```

```
[17]: MSSubClass      11680
LotConfig        91921
Condition1       89237
OverallQual      11680
YearBuilt         11680
ExterQual        86140
SalePrice         11680
dtype: int64
```

```
[18]: new_mem = housing2.memory_usage(index=False, deep=True)
new_mem
```

```
[18]: MSSubClass      2989
LotConfig        1930
Condition1       2332
OverallQual      1592
YearBuilt         2920
ExterQual        1915
SalePrice         5840
dtype: int64
```

Summing these two Series gets the total memory used for each. Dividing these totals reveals a reduction in memory of 94 percent.

```
[19]: new_mem.sum() / orig_mem.sum()
```

```
[19]: 0.062155672604755144
```

32.4 Exercises

Exercise 1

Read in the bikes data and select the `tripduration` column. Find its data type and then use the

`memory_usage` method to find how much memory (in bytes) it is using. Change its data type to the smallest possible type so that no information is lost. What percentage of memory has been saved?

[]:

Exercise 2

Read in the diamonds dataset and convert the data types of each column so they use the least amount of memory without losing any information.

[]:

Part VI

Grouping Data

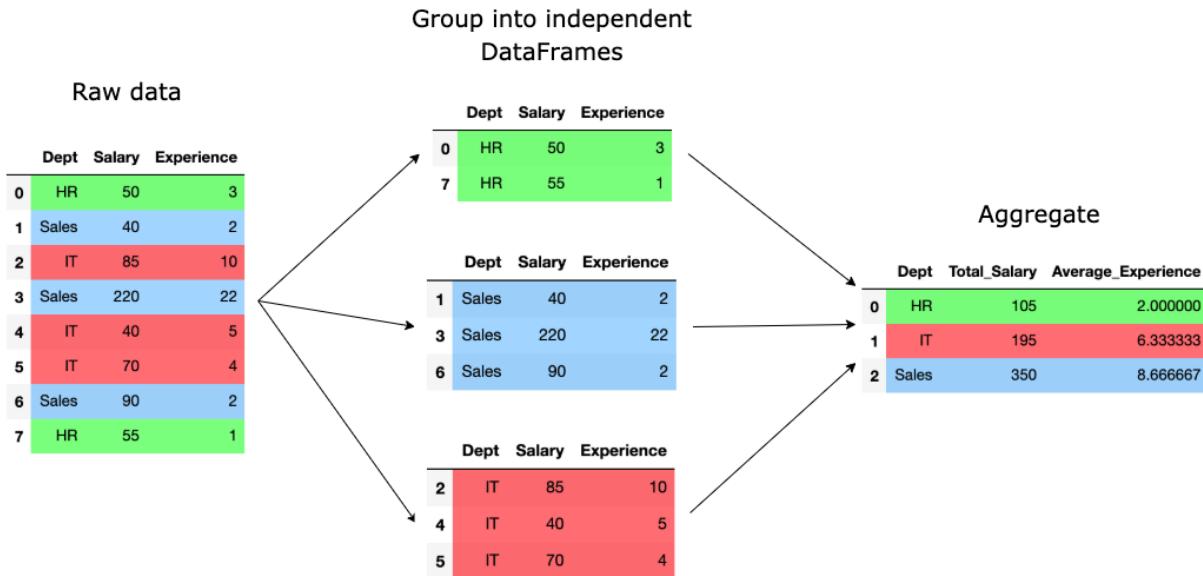
Chapter 33

Grouping Aggregation Basics

In previous chapters, when we called methods such as `sum` on our DataFrames, the action was performed to every single value in each column as a whole. In this chapter, we will perform actions to distinct groups within our data, and not to the whole.

33.1 Group into independent DataFrames, then aggregate

Take a look at the image below. We will group our original data into independent DataFrames based on the unique values of one or more columns. Below, our groups are based on the value of the ‘Dept’ column. Once the data is split into these independent DataFrames, an aggregation is performed on each. Below, the Salary column is aggregated by the sum function and the Experience column is aggregated by the mean function.



Examples of questions we can answer

Grouping data is an extremely common technique used in data analysis and can help us answer a variety of questions. Some examples follow:

- What is the maximum salary for every department at a company?
- What is the average temperature and precipitation for every month for different cities?
- What are the top 5 best selling shirts at each store?

33.2 Grouping with the groupby method

The `groupby` method handles most of the tasks in pandas involving grouping data. This one method is responsible for grouping the data into independent DataFrames as well as performing the aggregation, and usually does so in a single line of code.

Aggregation

By far, the most common type of function to call on each group is some kind of aggregation, though it's possible to manipulate the data in each group any way you want. This chapter only covers how to perform aggregations on each group.

Grouping Column, Aggregating Column, Aggregating Function

Every `groupby` aggregation has three separate components - the grouping column, the aggregating column, and the aggregating function.

- **Grouping column** - Every distinct value in this column forms its own group
- **Aggregating column** - The column we are applying the function to such that it aggregates (returns a single value). This column is usually numeric.
- **Aggregating function** - The function that is applied to the aggregating column.

33.3 Syntax for using the groupby method

The `groupby` method is not as straightforward to use as most other methods, and will take more effort to learn. Making it even more difficult, is the different valid types of syntax that do the same thing. Only one version of the syntax will be covered at first with the others delegated to a separate chapter.

Must use method chaining with groupby

Nearly all of the calls to `groupby` must have another method chained to it in order to return a result. The `groupby` method is a **two-step process**. First, we inform pandas how we would like to group, and then we chain the `agg` method to inform pandas how to aggregate. The general syntax takes the following form.

```
df.groupby('grouping column').agg(new_column=('aggregating column', 'aggregating function'))
```

The grouping column, aggregating column and aggregating function may be provided as strings. The new column name, `new_column`, is provided as a parameter name in the `agg` method and is assigned to the two-item tuple of the aggregating column and aggregating function.

First groupby aggregation

Let's begin our usage of the `groupby` method by finding the average salary of every employee by department from the City of Houston dataset.

```
[1]: import pandas as pd
emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Using the syntax from above, we produce the following solution:

```
[2]: emp.groupby('dept').agg(avg_salary=('salary', 'mean')).round(-3)
```

dept	avg_salary
Fire	61000.0
Health & Human Services	55000.0
Houston Airport System	55000.0
Houston Public Works	51000.0
Library	42000.0
Other	61000.0
Parks & Recreation	37000.0
Police	67000.0
Solid Waste Management	44000.0

Identify each piece

When completing a groupby aggregation, it is important to identify each of the pieces. This will help you insert them in the right place of the syntax above. In the above example:

- **Grouping column** - dept
- **Aggregating column** - salary
- **Aggregating function** - mean

New column name

You can use any valid variable name for the new column, but since Python does not allow spaces in variables names (as well as some other limitations), you won't have the full flexibility to name them however you want.

Tuple of aggregating column and aggregating function

The parameter used as the new column name must be set equal to a two-item tuple of the aggregating column and the aggregating function.

Use string names for aggregation functions

Notice that a string was used to identify the aggregation in the syntax above. pandas understands many string aggregation functions. Below are most of the available string names you can use. Later on we will see where these names came from and how to discover them on your own.

- `sum`
- `min`
- `max`
- `mean`
- `median`
- `std`
- `var`
- `count` - count of non-missing values
- `size` - count of all elements
- `first` - first value in group
- `last` - last value in group
- `idxmax` - index of maximum value in group
- `idxmin` - index of minimum value in group
- `nunique` - number of unique values in group

More examples

Let's complete a couple more examples with different grouping columns and different aggregating functions. The salary column is the only numeric column, so we will continue to use it as the aggregating column. Let's find the maximum salary by each title. The grouping column is `title`, the aggregating column is `salary`, and the aggregating function is `max`. We use `max_salary` as the new column name.

```
[3]: emp.groupby('title').agg(max_salary=('salary', 'max')).head().round(-3)
```

	max_salary
title	
3-1-1 TELECOMMUNICATOR	44000.0
3-1-1 TELECOMMUNICATOR SUPERVISOR	54000.0
9-1-1 CUSTODIAN OF RECORDS	58000.0
9-1-1 PSAP SUPERVISOR	69000.0
9-1-1 PSAP SUPERVISOR-FIRE/EMS	73000.0

Let's find the sum of all salaries by sex. The grouping column is `sex`, the aggregating column is `salary`, the aggregating function is `sum` and the new column is `sum_salary`.

```
[4]: emp.groupby('sex').agg(sum_salary=('salary', 'sum'))
```

	sum_salary
sex	
Female	3.980109e+08
Male	9.618155e+08

33.4 The index when grouping

If you were paying close attention, you noticed that the grouping column gets placed in the index after a call to the `groupby` method. In the example below, notice that `sex` is the new index and is not a column. Also note that the returned object is a **one-column DataFrame** and NOT a Series.

```
[5]: sex_avg_salary = emp.groupby('sex').agg(avg_salary=('salary', 'mean')).round(-3)
sex_avg_salary
```

avg_salary	
sex	
Female	55000.0
Male	60000.0

The index name

You might be confused as to why there is the word ‘sex’ directly above the index. It looks like it is a column name, but technically it is not. It is the `name` of the index and can be accessed as an `Index` attribute.

```
[6]: sex_avg_salary.index.name
```

```
[6]: 'sex'
```

The `reset_index` method

All `DataFrames` come equipped with a `reset_index` method which turns the index into the first column of a `DataFrame`. The new index will become a simple `RangeIndex`, the sequence of integers beginning at 0.

```
[7]: emp.groupby('sex').agg(avg_salary=('salary', 'mean')).round(-3).reset_index()
```

	sex	avg_salary
0	Female	55000.0
1	Male	60000.0

Not sorting the groups

You may have also noticed that the returned `DataFrame` is sorted by the grouping column by default. This is a nice feature, but you can turn it off by setting the `sort` parameter to `False`. This will increase its performance.

```
[8]: emp.groupby('title', sort=False).agg(max_salary=('salary', 'max')).head().round(-3)
```

	max_salary
	title
POLICE SERGEANT	88000.0
ASSISTANT CITY ATTORNEY II	90000.0
SENIOR SLUDGE PROCESSOR	49000.0
SENIOR POLICE OFFICER	76000.0
SENIOR ACCOUNT CLERK	48000.0

33.5 More on method chaining with groupby

The `groupby` syntax is a bit strange in that it requires method chaining to deliver results. Let's examine the results of making a call just to the `groupby` method.

```
[9]: emp.groupby('sex')
```

```
[9]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x10e81f190>
```

What is that?

The result of any call to a method in Python always returns something, even if that object is `None`. Calling the `groupby` method is no different. It has formally returned a `DataFrameGroupBy` object. Just like all pandas objects, you can see a list of all its [attributes and methods in the API](#). This type of object is not crucial to dive into at this point. Calling `groupby` by itself does not do much. You are simply alerting pandas that you would like to create distinct groups using the unique values in a particular column.

Assign the `groupby` object to a variable

Let's assign the result of the call to `groupby` as a variable and verify its type.

```
[10]: g = emp.groupby('sex')
       type(g)
```

```
[10]: pandas.core.groupby.generic.DataFrameGroupBy
```

33.6 GroupBy objects

The documentation refers to the object returned from a call to the `groupby` method as a **GroupBy** object. Technically, there are two specific objects - `DataFrameGroupBy` (as we saw above) and `SeriesGroupBy`. It's not necessary to know much about these objects. Just be aware that a call to `groupby` returns some other object that is not a `DataFrame` or a `Series`. It is a `GroupBy` object with its own attributes and methods.

Some exploration of the `GroupBy` object

This `GroupBy` object can be explored just like any other object in Python. It has attributes and methods that can be accessed through dot notation. Below, we take a look at the `groups` and `ngrps` attributes. The `groups` attribute is a dictionary that maps the group value to the index location of each row of that group, while `ngrps` returns the number of distinct groups.

[11]: `g.groups`

```
[11]: {'Female': Int64Index([      5,       6,      10,      12,      14,      19,      20,      24,      26,
                                28,
                                ...
                               24279, 24283, 24289, 24295, 24297, 24299, 24300, 24302, 24304,
                               24306],
                               dtype='int64', length=7358),
 'Male': Int64Index([      0,       1,       2,       3,       4,       7,       8,       9,      11,
                                13,
                                ...
                               24291, 24292, 24293, 24294, 24296, 24298, 24301, 24303, 24305,
                               24307],
                               dtype='int64', length=16950)}
```

[12]: `g.ngroups`

[12]: 2

Calling the `agg` method from the `GroupBy` object

We can call the `agg` method from this `GroupBy` object to complete another aggregation similar to how we did before.

[13]: `g.agg(std_salary=('salary', 'std'))`

std_salary	
sex	
Female	25087.145161
Male	22306.107822

Atypical usage

Even though it is syntactically correct to assign the result of a call to the `groupby` method to a variable name and then call the `agg` method, it is rarely written like this and should just be completed in a single line. The primary message of this last section was to show you that an intermediate object was created (either a `DataFrameGroupBy` or a `SeriesGroupBy`) and that this object is what is then being used to do the aggregation.

33.7 Exercises

Exercise 1

Find the maximum salary for each sex.

[]:

Exercise 2

Find the median salary for each department.

[]:

Exercise 3

Find the average salary for each race. Return a DataFrame with the race as a column.

[]:

Exercise 4

Find the number of employees in each department.

[]:

Exercise 5

Find the number of unique titles there are for each department.

[]:

Exercise 6

Find the index of the employee with the maximum salary for each department and then use those index values to select their entire rows from the original DataFrame.

[]:

Use the NYC deaths dataset for the remaining exercises

Execute the cell below to read in the NYC deaths dataset and use it to answer the following exercises.

```
[14]: deaths = pd.read_csv('../data/nyc_deaths.csv')
deaths.head(3)
```

	year	cause	sex	race	deaths
0	2007	Accidents	F	Asian	32
1	2007	Accidents	F	Black	87
2	2007	Accidents	F	Hispanic	71

Exercise 7

What year had the most deaths?

[]:

Exercise 8

Find the total number of deaths by race and sort by most to least.

[]:

Exercise 9

Find the total number of deaths by cause and then select the five highest causes.

[]:

Chapter 34

Grouping and Aggregating with Multiple Columns

In this chapter, we learn how to form groups using more than one column. We will also aggregate more than one column and learn how to apply more than one aggregation function to each group. Let's read in the San Francisco employee compensation dataset.

```
[1]: import pandas as pd  
sf_emp = pd.read_csv('../data/sf_employee_compensation.csv')  
sf_emp.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

34.1 Review grouping and aggregating with a single column

In the previous chapter, we had a single grouping column, aggregating column, and aggregating function. The following syntax was used as a guide:

```
df.groupby('grouping column').agg(new_column=('aggregating column', 'aggregating function'))
```

Let's see this again by calculating the average salary for each organization group.

```
[2]: sf_emp.groupby('organization group').agg(avg_salary=('salaries', 'mean')).round(-3)
```

organization group	avg_salary
Community Health	59000.0
Culture & Recreation	29000.0
General Administration & Finance	59000.0
General City Responsibilities	34000.0
Human Welfare & Neighborhood Development	44000.0
Public Protection	77000.0
Public Works, Transportation & Commerce	58000.0

34.2 Grouping with multiple columns

To create groups based on distinct values from multiple columns, we need to pass a list of these columns to the `groupby` method. Let's find the average salary for every unique combination of year and organization group.

```
[3]: sf_emp.groupby(['year', 'organization group']) \
    .agg(avg_salary=('salaries', 'mean')).round(-3).head(10)
```

year	organization group	avg_salary
2013	Community Health	59000.0
	Culture & Recreation	31000.0
	General Administration & Finance	65000.0
	General City Responsibilities	13000.0
	Human Welfare & Neighborhood Development	50000.0
	Public Protection	87000.0
2014	Public Works, Transportation & Commerce	64000.0
	Community Health	61000.0
	Culture & Recreation	28000.0
2015	General Administration & Finance	57000.0
	General City Responsibilities	30000.0
	Human Welfare & Neighborhood Development	40000.0

What happened to our index?

Both year and organization group are no longer columns and have been pushed into the index. This is called a **multi-level index**. The year and organization group are considered **levels** of the index and are NOT columns. You'll notice that duplicated values in the outer level are not visible in an index when they immediately follow one another such as with the year level above.

The MultiIndex is confusing and not necessary for beginners

In my opinion, the multi-level index does not add much value to pandas and can interfere with learning. I advise those new to pandas to avoid using it until they have mastered the basics. Personally, I rarely use it myself and prefer the values of the index to be DataFrame columns.

By default, all grouping columns will be added to the index. From this point on, we will chain the `reset_index` method to return these levels to columns.

```
[4]: sf_emp.groupby(['year', 'organization group']) \
    .agg(avg_salary=('salaries', 'mean')).round(-3).reset_index().head(10)
```

	year	organization group	avg_salary
0	2013	Community Health	59000.0
1	2013	Culture & Recreation	31000.0
2	2013	General Administration & Finance	65000.0
3	2013	General City Responsibilities	13000.0
4	2013	Human Welfare & Neighborhood Development	50000.0
5	2013	Public Protection	87000.0
6	2013	Public Works, Transportation & Commerce	64000.0
7	2014	Community Health	61000.0
8	2014	Culture & Recreation	28000.0
9	2014	General Administration & Finance	57000.0

Isn't it easier to read with a MultiIndex?

The MultiIndex can make the results easier to read, but it makes further data analysis more difficult as you need to become familiar with special syntax just for the MultiIndex. This added complexity for beginners is not worth the benefit.

34.3 Aggregating multiple columns

To aggregate multiple columns, set a new parameter in the `agg` method that is equal to another two-item tuple containing the aggregating column and aggregating function. Here, we find the average salary and overtime for each organization group.

```
[5]: sf_emp.groupby('organization group').agg(avg_salary=('salaries', 'mean'),
                                             avg_overtime=('overtime', 'mean')) \
    .round(-3).reset_index()
```

	organization group	avg_salary	avg_overtime
0	Community Health	59000.0	2000.0
1	Culture & Recreation	29000.0	1000.0
2	General Administration & Finance	59000.0	1000.0
3	General City Responsibilities	34000.0	3000.0
4	Human Welfare & Neighborhood Development	44000.0	1000.0
5	Public Protection	77000.0	12000.0
6	Public Works, Transportation & Commerce	58000.0	5000.0

34.4 Multiple grouping columns, aggregating columns, and aggregating functions

We can combine the last two approaches to simultaneously have multiple grouping and aggregating columns along with multiple aggregating functions. The following finds the mean, min, and max salaries along with the average overtime for every unique combination of year and organization group.

```
[6]: sf_emp.groupby(['year', 'organization group']) \
    .agg(avg_salary=('salaries', 'mean'),
        min_salary=('salaries', 'min'),
        max_salary=('salaries', 'max'),
        avg_overtime=('overtime', 'mean')).round(-3).reset_index().head(10)
```

year	organization group	avg_salary	min_salary	max_salary	avg_overtime
0 2013	Community Health	59000.0	0.0	231000.0	2000.0
1 2013	Culture & Recreation	31000.0	0.0	144000.0	1000.0
2 2013	General Administration & Finance	65000.0	0.0	285000.0	1000.0
3 2013	General City Responsibilities	13000.0	0.0	37000.0	2000.0
4 2013	Human Welfare & Neighborhood Development	50000.0	0.0	180000.0	0.0
5 2013	Public Protection	87000.0	-3000.0	238000.0	11000.0
6 2013	Public Works, Transportation & Commerce	64000.0	-1000.0	217000.0	5000.0
7 2014	Community Health	61000.0	0.0	228000.0	1000.0
8 2014	Culture & Recreation	28000.0	0.0	169000.0	1000.0
9 2014	General Administration & Finance	57000.0	0.0	233000.0	1000.0

34.5 Getting the size of each group

Let's say we are interested in the number of occurrences of each unique value in the grouping column. We can use the `size` aggregating function like this.

```
[7]: sf_emp.groupby('organization group').agg(size_salaries=('salaries', 'size'))
```

organization group	size_salaries
Community Health	9044
Culture & Recreation	3697
General Administration & Finance	3707
General City Responsibilities	9176
Human Welfare & Neighborhood Development	3758
Public Protection	7867
Public Works, Transportation & Commerce	12751

The `size` aggregating function returns the number of values in each group. It is independent of the aggregating column, so regardless of which one you use, the same value is returned. Here we use three different aggregating columns to prove that the size of the group is the same.

```
[8]: sf_emp.groupby('organization group') \
    .agg(size_salary=('salaries', 'size'),
        size_overtime=('overtime', 'size'),
        size_retirement=('retirement', 'size')).reset_index().head(10)
```

	organization group	size_salary	size_overtime	size_retirement
0	Community Health	9044	9044	9044
1	Culture & Recreation	3697	3697	3697
2	General Administration & Finance	3707	3707	3707
3	General City Responsibilities	9176	9176	9176
4	Human Welfare & Neighborhood Development	3758	3758	3758
5	Public Protection	7867	7867	7867
6	Public Works, Transportation & Commerce	12751	12751	12751

Just use `value_counts`

There isn't a need to call the `groupby` method with the `size` aggregating function when grouping by a single column. This is exactly what the Series method `value_counts` was designed for. It has the added benefit of sorting the values as well.

```
[9]: sf_emp['organization group'].value_counts()
```

Public Works, Transportation & Commerce	12751
General City Responsibilities	9176
Community Health	9044
Public Protection	7867
Human Welfare & Neighborhood Development	3758
General Administration & Finance	3707

```
Culture & Recreation           3697
Name: organization group, dtype: int64
```

Multiple group size

When using multiple grouping columns, you'll need to use the `groupby` method to find the size of each group as there is no `value_counts` method for DataFrames. The choice for aggregating column does not matter as they all return the same result.

```
[10]: sf_emp.groupby(['year', 'organization group']) \
    .agg(size_salary=('salaries', 'size')).head(10)
```

		size_salary
	year	organization group
2013		Community Health
		Culture & Recreation
		General Administration & Finance
		General City Responsibilities
		Human Welfare & Neighborhood Development
		Public Protection
2014		Public Works, Transportation & Commerce
		Community Health
		Culture & Recreation
		General Administration & Finance
		1092
		408
		437
		22
		385
		940
		1402
		1110
		451
		433
		433

Alternative Syntax for size

You can call the `size` method directly after grouping without the need to use `agg`. This returns the same data as a Series.

```
[11]: sf_emp.groupby(['year', 'organization group']).size().head(10)
```

```
[11]: year  organization group
2013  Community Health           1092
      Culture & Recreation        408
      General Administration & Finance 437
      General City Responsibilities     22
      Human Welfare & Neighborhood Development 385
      Public Protection             940
      Public Works, Transportation & Commerce 1402
2014  Community Health           1110
      Culture & Recreation        451
      General Administration & Finance 433
dtype: int64
```

Rename the column when using `reset_index`

When calling `reset_index` on a Series, like we did above, the new column name for the Series values was the `name` attribute of the Series. If it doesn't exist (like in the example above) then you can supply the column name with the `name` parameter with `reset_index`.

```
[12]: sf_emp.groupby(['year', 'organization group']).size().reset_index(name='size').head(10)
```

	year	organization group	size
0	2013	Community Health	1092
1	2013	Culture & Recreation	408
2	2013	General Administration & Finance	437
3	2013	General City Responsibilities	22
4	2013	Human Welfare & Neighborhood Development	385
5	2013	Public Protection	940
6	2013	Public Works, Transportation & Commerce	1402
7	2014	Community Health	1110
8	2014	Culture & Recreation	451
9	2014	General Administration & Finance	433

34.6 Exercises

Execute the following cell to read in the City of Houston employee data and use it for the first few exercises.

Exercise 1

For each department and sex, find the number of unique position titles, the total number of employees, and the average salary. Make sure there is no multi-level index.

```
[ ]:
```

Exercise 2

For each department, race, and sex find the min and max and salaries.

```
[ ]:
```

Exercise 3

Which city name appears the most frequently. Do this in two different ways. Do it once with and once without the `groupby` method?

```
[ ]:
```

Exercise 4

Does the city 'Houston' only appear in the state of Texas (abbreviated 'TX')?

[]:

Exercise 5

Find the maximum undergraduate population for each state?

[]:

Exercise 6

Find the largest college from each state. From those colleges, find the difference between the largest and smallest.

[]:

Exercise 7

Find the name and population of the largest college per state.

[]:

Exercise 8

Do distance only schools tend to have more or less student population than non-distance-only schools?

[]:

Exercise 9

Do distance only schools tend to be more or less religiously affiliated than non-distance-only schools?

[]:

Exercise 10

What state has the lowest percentage of currently operating schools of those that have religious affiliation?

[]:

Exercise 11

Find the top 5 historically black colleges that have the highest undergraduate white percentage (ugds_white)?

[]:

Chapter 35

Grouping with Pivot Tables

A pivot table aggregates data between the intersection of the unique values of two (or more) columns of your data. In the pivot table below, the two columns are the `race` and the `sex`. All pivot tables must aggregate some other column of data. Here, the salary is averaged. There are 5 unique races and 2 unique values for sex. The pivot table shows the mean of salary for each possible combination. Having the data in this structure, can make it easier to read and make comparisons.

sex	Female	Male
race		
Asian	66000	65000
Black	52000	52000
Hispanic	49000	58000
Native American	49000	62000
White	66000	67000

Creating a simple pivot table in pandas - four components

There are four components to a basic pivot table in pandas.

- Two grouping columns
- One aggregating column
- One aggregating function

In the example above, the two grouping columns are `race` and `sex`. The aggregating column is `salary` and the aggregating function is `mean`.

35.1 Creating the pivot table above with pandas

Let's read in the employee dataset and use it to recreate the pivot table above in pandas.

```
[1]: import pandas as pd  
emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])  
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

The `pivot_table` method creates pivot tables for us in pandas. To use a pivot table, we set the `index`, `columns`, `values`, and `aggfunc` parameters, which take on the following component of the pivot table.

- `index` - grouping column
- `columns` - grouping column
- `values` - aggregating column
- `aggfunc` - aggregating function (defaulted to the mean)

```
[2]: emp.pivot_table(index='race', columns='sex', values='salary', aggfunc='mean')
```

sex	Female	Male
race		
Asian	65846.027190	65071.505725
Black	52416.790860	52154.353471
Hispanic	48835.416765	57637.428134
Native American	48767.227179	61672.815192
White	66411.819615	66653.901622

Round results and convert to integer to reduce noise

The above DataFrame has lots of excess decimal values that aren't important in this result. Rounding to the nearest thousand and changing the data type to be integers reduces this noise.

```
[3]: emp.pivot_table(index='race', columns='sex', values='salary', aggfunc='mean')\
    .round(-3).astype('int').astype('int64')
```

sex	Female	Male
race		
Asian	66000	65000
Black	52000	52000
Hispanic	49000	58000
Native American	49000	62000
White	66000	67000

Easily compare female vs male salary

Pivot tables make comparisons between groups easy. In this instance, the difference between female and male average salary is easily seen.

Column and index labels are sorted

Notice that the labels for each of the index and columns of a pivot table come from the unique values of the grouping columns. They are also sorted in alphabetical order. The intersection of each label is where the aggregated data appears.

35.2 Comparison to groupby

The `pivot_table` method is very similar to a `groupby` aggregation. Both are capable of producing the exact same results. Below, we replicate the results of our pivot table with `groupby`.

```
[4]: emp.groupby(['race', 'sex']).agg(mean_salary=('salary', 'mean')) \
    .round(-3).astype('int64')
```

mean_salary		
race	sex	
Asian	Female	66000
	Male	65000
Black	Female	52000
	Male	52000
Hispanic	Female	49000
	Male	58000
Native American	Female	49000
	Male	62000
White	Female	66000
	Male	67000

Data is more difficult to make comparisons

This call to `groupby` produced the exact same result as the pivot table but in a different shape. Having all of our data in a vertical column makes it difficult to make comparisons.

Wide vs long data

Pivot tables produce **wide** data, with new columns for each unique value of one of the grouping columns. Wide data is typically easier to read and make decisions with. The `groupby` method returns **long** data with the results of each group in a single column making it more difficult to make comparisons. This type of data is generally easier to continue analyzing with more commands.

The default aggregation is mean

By default, the `aggfunc` parameter is set to ‘mean’. But, even if you are using the mean as your aggregation function, I advise that you explicitly state it in your call to `pivot_table` so that it’s clear what you are doing.

All aggregation strings are available for `pivot_table`

All the aggregation strings (‘min’, ‘max’, ‘mean’, etc...) are available to a `pivot_table` just as they were with `groupby`. Here we find the max salary for the same groups.

```
[5]: emp.pivot_table(index='race', columns='sex', values='salary', aggfunc='max')
```

sex	Female	Male
race		
Asian	342784.0	342784.0
Black	342784.0	342784.0
Hispanic	180000.0	342784.0
Native American	99784.0	121548.0
White	342784.0	342784.0

Where is the ‘pivoting’?

Microsoft Excel is well-known for its pivot tables that are easily set up by dragging and dropping different columns into different boxes, ‘pivoting’ the data around. With pandas, you’ll have to change the parameter values and call the `pivot_table` method again in order to get the same effect. Let’s pivot the table by putting sex along the index and race along the columns.

```
[6]: emp.pivot_table(index='sex', columns='race', values='salary', aggfunc='max')
```

race	Asian	Black	Hispanic	Native American	White
sex					
Female	342784.0	342784.0	180000.0	99784.0	342784.0
Male	342784.0	342784.0	342784.0	121548.0	342784.0

35.3 Styling pivot tables

You can style your DataFrame by changing the text color, background color, font, and several other items through the `style` accessor. It works similarly to `str` and `dt` accessors in that it gives you access to style-only methods through the dot notation. [Visit the documentation](#) for descriptions on all the methods. Let’s assign a pivot table computing the mean salary by department and race to a variable.

```
[7]: dept_race_mean = emp.pivot_table(index='dept', columns='race',
                                         values='salary', aggfunc='mean') \
                                         .round(-3).astype('int64')
dept_race_mean
```

	race	Asian	Black	Hispanic	Native American	White
dept						
Fire	69000	60000	57000		62000	62000
Health & Human Services	64000	57000	47000		45000	66000
Houston Airport System	53000	51000	46000		60000	69000
Houston Public Works	68000	45000	49000		52000	66000
Library	40000	40000	40000		30000	48000
Other	72000	59000	53000		57000	75000
Parks & Recreation	40000	35000	36000		39000	45000
Police	67000	62000	65000		69000	71000
Solid Waste Management	62000	43000	43000		31000	48000

Highlighting the maximum value

The `highlight_max` method highlights the maximum value in each column or row. By default, it highlights the maximum value of each column, just like most other pandas methods.

```
[8]: dept_race_mean.style.highlight_max()
```

	race	Asian	Black	Hispanic	Native American	White
dept						
Fire	69000	60000	57000		62000	62000
Health & Human Services	64000	57000	47000		45000	66000
Houston Airport System	53000	51000	46000		60000	69000
Houston Public Works	68000	45000	49000		52000	66000
Library	40000	40000	40000		30000	48000
Other	72000	59000	53000		57000	75000
Parks & Recreation	40000	35000	36000		39000	45000
Police	67000	62000	65000		69000	71000
Solid Waste Management	62000	43000	43000		31000	48000

Change direction with axis

You can highlight the max of each row by setting the `axis` parameter to ‘columns’ or 1.

```
[9]: dept_race_mean.style.highlight_max(axis='columns')
```

	race	Asian	Black	Hispanic	Native American	White
dept						
	Fire	69000	60000	57000	62000	62000
Health & Human Services	64000	57000	47000	45000	66000	
Houston Airport System	53000	51000	46000	60000	69000	
Houston Public Works	68000	45000	49000	52000	66000	
Library	40000	40000	40000	30000	48000	
Other	72000	59000	53000	57000	75000	
Parks & Recreation	40000	35000	36000	39000	45000	
Police	67000	62000	65000	69000	71000	
Solid Waste Management	62000	43000	43000	31000	48000	

Background color gradients

Use `background_gradient` to color the background based on the value of the cell. You can change the colors by choosing a [Matplotlib colormap](#).

```
[10]: dept_race_mean.style.background_gradient(cmap='coolwarm')
```

	race	Asian	Black	Hispanic	Native American	White
dept						
	Fire	69000	60000	57000	62000	62000
Health & Human Services	64000	57000	47000	45000	66000	
Houston Airport System	53000	51000	46000	60000	69000	
Houston Public Works	68000	45000	49000	52000	66000	
Library	40000	40000	40000	30000	48000	
Other	72000	59000	53000	57000	75000	
Parks & Recreation	40000	35000	36000	39000	45000	
Police	67000	62000	65000	69000	71000	
Solid Waste Management	62000	43000	43000	31000	48000	

Adding commas to numbers

Make your data easier to read by inserting commas into large numbers with the `format` method. To do so, you must be aware of how to use the [string format specification](#) from core Python. To add commas use the string '`{:, .0f}`'. If you are unfamiliar with format specification, use the link provided. But, as a quick overview, the actual specification is what comes after the colon and does not include the curly braces. Here, it is '`,d.0f`'. The comma is the digit separator. The `.0` stands for the numbers after the decimal, which is 0 in this case. The character '`f`' is the 'type' and stands for fixed-point notation so that you can include decimals if you would like. If you wanted to include two decimal places, you would use the format

specification '{:, .2f}'.

```
[11]: dept_race_mean.style.format('{:, .0f}')
```

	race	Asian	Black	Hispanic	Native American	White
dept						
	Fire	69,000	60,000	57,000	62,000	62,000
Health & Human Services	64,000	57,000	47,000	45,000	66,000	
Houston Airport System	53,000	51,000	46,000	60,000	69,000	
Houston Public Works	68,000	45,000	49,000	52,000	66,000	
Library	40,000	40,000	40,000	30,000	48,000	
Other	72,000	59,000	53,000	57,000	75,000	
Parks & Recreation	40,000	35,000	36,000	39,000	45,000	
Police	67,000	62,000	65,000	69,000	71,000	
Solid Waste Management	62,000	43,000	43,000	31,000	48,000	

Multiple style formats

You can add multiple different styles to your DataFrame by chaining together style methods. Before doing so, it's important to know that the returned object from a call to one of the style methods is NOT a DataFrame. It's a special kind of styled object.

```
[12]: df_styled = dept_race_mean.style.format('{:, .0f}')
       type(df_styled)
```

```
[12]: pandas.io.formats.style.Styler
```

You cannot use this object as a normal DataFrame. If you try and call a normal DataFrame method such as `mean`, you'll get an error.

```
[13]: df_styled.mean()
```

```
AttributeError: 'Styler' object has no attribute 'mean'
```

You can retrieve the original (unstylyed data) with the `data` attribute. Here, we verify it is the same object as the pivot table we calculated above.

```
[14]: df_styled.data is dept_race_mean
```

```
[14]: True
```

The benefit of having this new styled object is that you can call one style method after another without referencing the `style` attribute again. Here, we add commas and highlight the maximum and minimum value of each column.

```
[15]: dept_race_mean.style.format('{:.0f}') \
    .highlight_max(color='yellow') \
    .highlight_min(color='lightblue')
```

	race	Asian	Black	Hispanic	Native American	White
dept						
Fire	69,000	60,000	57,000		62,000	62,000
Health & Human Services	64,000	57,000	47,000		45,000	66,000
Houston Airport System	53,000	51,000	46,000		60,000	69,000
Houston Public Works	68,000	45,000	49,000		52,000	66,000
Library	40,000	40,000	40,000		30,000	48,000
Other	72,000	59,000	53,000		57,000	75,000
Parks & Recreation	40,000	35,000	36,000		39,000	45,000
Police	67,000	62,000	65,000		69,000	71,000
Solid Waste Management	62,000	43,000	43,000		31,000	48,000

35.4 Getting the size of each group

You can use the `pivot_table` method to get the total number of occurrences of each of the grouping columns. When doing so, it is not necessary to have an aggregating column (the `values` parameter). pandas knows that the size of the group is independent of what is aggregating, so it does not enforce it that you use it. Here, we get the size of each unique combination of department and sex.

```
[16]: emp.pivot_table(index='dept', columns='race', aggfunc='size')
```

	race	Asian	Black	Hispanic	Native American	White
dept						
Fire	80	699	1159		36	2399
Health & Human Services	140	668	399		11	132
Houston Airport System	123	427	317		9	337
Houston Public Works	377	2229	820		24	735
Library	47	223	139		4	150
Other	256	1515	898		18	680
Parks & Recreation	36	641	329		4	139
Police	479	1852	2010		39	3189
Solid Waste Management	9	407	77		1	18

35.5 Add margins to get row and column totals

The optional parameter `margins` can be set to `True` to add one additional row and column to the pivot table which calculates the same aggregate function to the entire row or column. In the following pivot table, the average salary for all fire department employees is 61,000, and the average salary for all hispanic employees is 55,000.

```
[17]: emp.pivot_table(index='dept', columns='race', values='salary',
                     aggfunc='mean', margins=True) \
    .round(-3) \
    .style.format('{:, .0f}'')
```

	race	Asian	Black	Hispanic	Native American	White	All
dept							
Fire	69,000	60,000	57,000		62,000	62,000	61,000
Health & Human Services	64,000	57,000	47,000		45,000	66,000	55,000
Houston Airport System	53,000	51,000	46,000		60,000	69,000	55,000
Houston Public Works	68,000	45,000	49,000		52,000	66,000	51,000
Library	40,000	40,000	40,000		30,000	48,000	42,000
Other	72,000	59,000	53,000		57,000	75,000	61,000
Parks & Recreation	40,000	35,000	36,000		39,000	45,000	37,000
Police	67,000	62,000	65,000		69,000	71,000	67,000
Solid Waste Management	62,000	43,000	43,000		31,000	48,000	44,000
All	65,000	52,000	55,000		58,000	67,000	58,000

The average of all the employees is given by the bottom right value. Let's verify this is correct by computing the average manually.

```
[18]: emp['salary'].mean()
```

```
[18]: 58206.76157092714
```

35.6 Non-standard pivot tables

There are many different kinds of pivot tables that we can create besides one with exactly two grouping columns, one aggregating column and one aggregating function. I am calling such pivot tables ‘non-standard’, though this is just to help differentiate them from the previous ones created above.

A single grouping column

It is possible to have just a single grouping column in a pivot table. Here, we find the average salary for each department.

```
[19]: emp.pivot_table(index='dept', values='salary', aggfunc='mean').round(-3)
```

		salary
		dept
	Fire	61000.0
	Health & Human Services	55000.0
	Houston Airport System	55000.0
	Houston Public Works	51000.0
	Library	42000.0
	Other	61000.0
	Parks & Recreation	37000.0
	Police	67000.0
	Solid Waste Management	44000.0

When using a single grouping column, the result will be the exact same as a groupby aggregation. The groupby has the advantage of being able to rename the resulting aggregate column.

```
[20]: emp.groupby('dept').agg(average_salary=('salary', 'mean')).round(-3)
```

		average_salary
		dept
	Fire	61000.0
	Health & Human Services	55000.0
	Houston Airport System	55000.0
	Houston Public Works	51000.0
	Library	42000.0
	Other	61000.0
	Parks & Recreation	37000.0
	Police	67000.0
	Solid Waste Management	44000.0

You can pivot this one column table by choosing use the `columns` parameter.

```
[21]: emp.pivot_table(columns='dept', values='salary', aggfunc='mean').round(-3)
```

dept	Fire	Health & Human Services	Houston Airport System	Houston Public Works	Library	Other	Parks & Recreation	Police	Solid Waste Management
salary	61000.0	55000.0	55000.0	51000.0	42000.0	61000.0	37000.0	67000.0	44000.0

More than two grouping columns

You can use any number of grouping columns when creating pivot tables. Use a list to contain the columns you want for the rows or columns. Here we find the average salary by department, sex, and race. The unique combinations of department and sex are placed in the index and become a multi-level index.

```
[22]: emp.pivot_table(index=['dept', 'sex'], columns='race',
                     values='salary', aggfunc='max') \
    .round(-3).head(10).style.format('{:, .0f}'')
```

		dept	sex	race	Asian	Black	Hispanic	Native American	White
				Female	343,000	343,000	90,000	70,000	343,000
Fire				Male	343,000	343,000	343,000	116,000	343,000
				Female	150,000	187,000	130,000	62,000	187,000
Health & Human Services				Male	150,000	187,000	130,000	55,000	139,000
				Female	119,000	156,000	165,000	91,000	180,000
Houston Airport System				Male	134,000	195,000	275,000	106,000	197,000
				Female	126,000	181,000	123,000	57,000	275,000
Houston Public Works				Male	180,000	216,000	165,000	109,000	180,000
				Female	80,000	170,000	108,000	31,000	116,000
Library				Male	62,000	115,000	115,000	29,000	104,000

We can pivot this so that the result so that there is a multi-level column index.

```
[23]: emp.pivot_table(index='dept', columns=['race', 'sex'],
                     values='salary', aggfunc='max') \
    .round(-3).style.format('{:, .0f}'')
```

	race		Asian		Black		Hispanic		Native American		White	
	sex	Female	Male	Female	Male	Female	Male	Female	Male	Female	Male	
dept												
Fire	343,000	343,000	343,000	343,000	90,000	343,000	70,000	116,000	343,000	343,000	343,000	
Health & Human Services	150,000	150,000	187,000	187,000	130,000	130,000	62,000	55,000	187,000	139,000		
Houston Airport System	119,000	134,000	156,000	195,000	165,000	275,000	91,000	106,000	180,000	197,000		
Houston Public Works	126,000	180,000	181,000	216,000	123,000	165,000	57,000	109,000	275,000	180,000		
Library	80,000	62,000	170,000	115,000	108,000	115,000	31,000	29,000	116,000	104,000		
Other	205,000	169,000	191,000	275,000	180,000	191,000	100,000	122,000	205,000	202,000		
Parks & Recreation	77,000	71,000	132,000	102,000	105,000	131,000	30,000	53,000	81,000	150,000		
Police	133,000	174,000	187,000	199,000	116,000	280,000	88,000	98,000	188,000	199,000		
Solid Waste Management	102,000	85,000	149,000	195,000	100,000	61,000	31,000	nan	103,000	83,000		

Keep the multi-level index with pivot tables

I advise that you keep the multiple levels when using a pivot table (if you happen to produce them). This is the opposite advice that I gave when grouping. The reasoning is that pivot tables are much more likely to be a final product - something that you use in a presentation or a report and won't be doing further analysis on. Therefore, you won't have to handle the multi-level index or columns. After doing a groupby, it is more likely that you'll be running other pandas commands, and doing so is much easier with a normal single level index.

Multiple aggregating columns

It is possible to aggregate more than one column as well. Because the employee dataset only has one main aggregating column, we will add a column for years of experience. The data was pulled in 2019, so we will subtract the year hired from it to get the approximate years of experience.

```
[24]: emp['experience'] = 2019 - emp['hire_date'].dt.year
emp.head(3)
```

	dept	title	hire_date	salary	sex	race	experience
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White	18
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic	9
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black	13

Let's find the average salary and experience for every department and race. Notice that the columns are now multiple levels. Because of this, it becomes harder to use methods like `round` which you might need to specify different decimals for different columns. We compromise here and format our data to a single decimal place. The margins for each aggregating column are also there.

```
[25]: emp.pivot_table(index='dept', columns='race', values=['salary', 'experience'],  
                     aggfunc='mean', margins=True).style.format('{:.1f}')
```

	experience										salary		
dept	race	Asian	Black	Hispanic	Native American	White	All	Asian	Black	Hispanic	Native American	White	All
Fire	9.1	15.1	12.5	14.2	15.1	14.3	69,444.4	59,760.0	57,172.6	62,300.8	62,159.7	60,588.8	
Health & Human Services	9.6	10.0	10.3	13.5	9.2	10.0	63,803.4	56,765.3	46,622.1	44,537.5	65,618.5	55,263.3	
Houston Airport System	13.9	12.7	13.3	14.9	12.8	13.0	53,321.1	50,829.9	46,064.7	60,132.1	69,139.2	54,993.0	
Houston Public Works	12.7	11.1	11.6	10.2	11.8	11.5	68,128.4	44,829.6	48,829.6	51,950.8	65,667.4	51,412.7	
Library	7.1	9.5	9.1	12.8	10.9	9.6	39,699.0	40,302.2	39,752.9	29,780.5	47,845.7	42,051.3	
Other	11.4	11.9	10.6	12.9	10.5	11.2	72,340.0	58,655.7	52,989.1	56,699.2	75,005.2	61,476.3	
Parks & Recreation	9.6	7.1	11.0	7.2	9.2	8.5	39,840.1	35,464.7	36,403.3	39,020.8	45,170.1	37,057.0	
Police	12.5	16.2	14.8	18.1	19.7	14.5	66,725.4	61,925.0	64,903.4	68,556.4	70,932.1	66,621.2	
Solid Waste Management	11.8	8.2	11.6	1.0	6.6	8.7	61,529.3	43,258.8	43,409.4	31,325.0	48,422.2	43,760.8	
All	11.6	11.8	12.0	13.6	14.1	12.6	65,316.2	52,264.2	54,811.3	58,153.1	66,611.7	58,209.9	

Multiple aggregating functions

All components of a pivot table are capable of taking multiple values, including the aggregating functions. Here, we find the minimum, maximum, and average salary for each department and sex.

```
[26]: emp.pivot_table(index='dept', columns='sex', values='salary',  
                     aggfunc=['min', 'max', 'mean'], margins=True) \  
       .round(-3).style.format('{:, .0f}')
```

dept	sex	min			max			mean		
		Female	Male	All	Female	Male	All	Female	Male	All
Fire	Female	16,000	17,000	16,000	343,000	343,000	343,000	62,000	60,000	61,000
Health & Human Services	Female	25,000	25,000	25,000	187,000	187,000	187,000	54,000	59,000	55,000
Houston Airport System	Female	26,000	26,000	26,000	180,000	275,000	275,000	51,000	57,000	55,000
Houston Public Works	Female	29,000	28,000	28,000	275,000	216,000	275,000	51,000	51,000	51,000
Library	Female	10,000	11,000	10,000	170,000	115,000	170,000	41,000	44,000	42,000
Other	Female	25,000	26,000	25,000	205,000	275,000	275,000	61,000	62,000	61,000
Parks & Recreation	Female	25,000	25,000	25,000	132,000	150,000	150,000	37,000	37,000	37,000
Police	Female	27,000	28,000	27,000	188,000	280,000	280,000	58,000	69,000	67,000
Solid Waste Management	Female	27,000	27,000	27,000	149,000	195,000	195,000	47,000	43,000	44,000
All	Female	10,000	11,000	10,000	343,000	343,000	343,000	55,000	60,000	58,000

Reducing readability of a pivot table

Pivot tables with more than two grouping columns, or multiple aggregating columns and aggregating functions become less readable as the amount of data displayed can become huge. Here, we find the minimum, maximum, and average salary and experience for all combinations of department, sex, and race. Only the first few rows and columns are output.

```
[27]: df_messy_pivot = emp.pivot_table(index=['dept', 'sex'], columns='race',
                                     values=['salary', 'experience'],
                                     aggfunc=['min', 'max', 'mean'], margins=True)
df_messy_pivot.iloc[:6, :10]
```

dept	sex	min											
		experience						salary					
		race	Asian	Black	Hispanic	Native American	White	All	Asian	Black	Hispanic	Native American	
Fire	Female	1.0	1.0	1.0	4.0	1.0	1	39104.0	16411.0	28024.0	48189.7		
	Male	1.0	1.0	1.0	1.0	1.0	1	28024.0	28024.0	26000.0	28024.0		
Health & Human Services	Female	1.0	1.0	1.0	4.0	1.0	1	30888.0	25064.0	27560.0	33010.0		
	Male	1.0	1.0	1.0	2.0	1.0	1	35714.0	25064.0	25813.0	42848.0		
Houston Airport System	Female	1.0	1.0	1.0	2.0	1.0	1	26894.0	26125.0	26125.0	29058.0		
	Male	1.0	1.0	1.0	2.0	1.0	1	26915.0	26125.0	26894.0	39790.0		

These large tables can be useful, but the results are often better presented as multiple different pivot tables

than a single one. This pivot table has 19 rows and 36 columns with two levels in the index and three levels in the columns. It won't easily fit on a single screen.

[28]: `df_messy_pivot.shape`

[28]: (19, 36)

35.7 Exercises

Execute the following cell to read in the flights dataset and inserts columns for the day and month name. Use it for the following exercises.

[29]: `import pandas as pd
flights = pd.read_csv('../data/flights.csv', parse_dates=['date'])
flights.insert(1, 'day_of_week', flights['date'].dt.day_name())
flights.insert(2, 'month', flights['date'].dt.month_name())
flights.head(3)`

	date	day_of_week	month	airline	origin	...	carrier_delay	weather_delay	nas_delay	security_delay	late_aircraft_delay
0	2018-01-01	Monday	January	UA	LAS	...	0	0	0	0	0
1	2018-01-01	Monday	January	WN	DEN	...	0	0	0	0	0
2	2018-01-01	Monday	January	B6	JFK	...	0	83	8	0	0

Exercise 1

What is the average carrier delay for each day of the week for each airline? Highlight the worst day of the week for each airline.

[]:

Exercise 2

Use a pivot table to find the total number of canceled flights for each origin airport and airline. Place the airlines in the columns. Use the result to find the origin airport with the most cancelled flights for each airline. Also return this maximum number of cancelled flights.

[]:

Exercise 3

Find the total distance flown for each airline for each month. Highlight the month with the most number of miles flown and use the style `format` method to put commas in the numbers so that they are easier to read.

[]:

Exercise 4

Create a pivot table that shows the number of flights flown for every day of the week for every month.

[]:

Exercise 5

In exercise 4, the months and days of week are ordered alphabetically. It would be better if these values were ordered chronologically. Can you return a result that has both groups in the correct order. Use Monday as the first day of the week.

[]:

Exercise 6

Create a new column in the flights dataset called 'dep_time_hour' and set it equal to the hour (this will be an integer 0 through 23) of the flight. Find the average carrier delay for every month and dep_time_hour. Place the month in the columns.

[]:

Exercise 7

Use both `groupby` and `pivot_table` to compute the average and median distance flown by day of the week.

[]:

Chapter 36

Counting with Crosstabs

In this chapter, we explore the pandas `crosstab` function, which produces very similar results as the `pivot_table` method, but allows us to count occurrences in a cleaner way with more functionality.

Exploring mental health survey data

We will be using mental health survey data found on [Kaggle datasets](#). This dataset is from a 2014-2015 survey that measured the attitude towards mental health and frequency of mental health disorders in the tech workplace. Let's read in the dataset and output the number of rows and columns.

```
[1]: import pandas as pd  
mh = pd.read_csv('../data/mental_health.csv')  
mh.head(3)
```

	year	age	gender	country	family_history	...	leave	mental_health_consequence	phys_health_consequence	coworkers	supervisor
0	2014	37	Female	United States		No ...	Somewhat easy		No	No	Some of them
1	2014	44	Male	United States		No ...	Don't know		Maybe	No	No
2	2014	32	Male	Canada		No ...	Somewhat difficult		No	No	Yes

```
[2]: mh.shape
```

```
[2]: (1091, 19)
```

Data Dictionary

The data dictionary will help you understand the questions asked behind the data collected on each column. Some of the column descriptions are larger than the default 50 character setting. We change the option `'display.max_colwidth'` before outputting the data dictionary.

```
[3]: pd.set_option('display.max_colwidth', 100)  
mh_dd = pd.read_csv('../data/dictionaries/mental_health_dd.csv')  
mh_dd
```

Column Name		Description
0	year	Year the survey was submitted
1	age	Respondent age
2	gender	Respondent gender
3	country	Respondent country
4	family_history	Do you have a family history of mental illness?
5	treatment	Have you sought treatment for a mental health condition?
6	work_interfere	Does your mental health condition interfere with your work (if you have one)?
7	no_employees	How many employees does your company or organization have?
8	tech_company	Is your employer primarily a tech company/organization?
9	benefits	Does your employer provide mental health benefits?
10	care_options	Do you know the options for mental health care your employer provides?
11	wellness_program	Has your employer ever discussed mental health?
12	seek_help	Does your employer provide resources to learn and seek help about mental health issues?
13	anonymity	Is your anonymity protected if you use mental health resources?
14	leave	How easy is it for you to take medical leave for a mental health condition?
15	mental_health_consequence	Would discussing a mental health issue with your employer have negative consequences?
16	phys_health_consequence	Would discussing a physical health issue with your employer have negative consequences?
17	coworkers	Would you be willing to discuss a mental health issue with your coworkers?
18	supervisor	Would you be willing to discuss a mental health issue with your supervisor?

Converting columns to categorical

All of the questions have a limited number of discrete answer choices with each column from gender to the end being a string. Let's verify that the number of unique values in each column is limited, with only age being the exception.

```
[4]: mh.nunique()
```

```
[4]:
```

year	2
age	49
gender	2
country	8
family_history	2
treatment	2
work_interfere	4
no_employees	6
tech_company	2
benefits	3

```

care_options           3
wellness_program       3
seek_help              3
anonymity              3
leave                  5
mental_health_consequence 3
phys_health_consequence 3
coworkers              3
supervisor              3
dtype: int64

```

Let's convert all of the columns beginning from gender to the end to categorical. Instead of converting each column individually, we can convert an entire selection of the DataFrame and overwrite the old columns at the same time. This will save a tremendous amount of space and help performance when grouping.

```
[5]: mh.loc[:, 'gender':] = mh.loc[:, 'gender':].astype('category')
mh.dtypes
```

```

[5]: year                  int64
age                   int64
gender                category
country               category
family_history        category
treatment             category
work_interfere        category
no_employees          category
tech_company          category
benefits              category
care_options           category
wellness_program       category
seek_help              category
anonymity              category
leave                  category
mental_health_consequence  category
phys_health_consequence  category
coworkers              category
supervisor              category
dtype: object

```

36.1 Frequency counting with a Series

Previously, we learned how to count the frequency of values in a Series, single column of data, with the `value_counts` method. Let's review this by finding the number of survey respondents by country.

```
[6]: mh['country'].value_counts()
```

```

[6]: United States      717
United Kingdom        177
Canada                 68
Germany                43
Netherlands            27

```

```
Ireland      27
Australia   21
France      11
Name: country, dtype: int64
```

The relative frequencies are returned by setting `normalize` to `True`.

```
[7]: mh['country'].value_counts(normalize=True).round(3)
```

```
[7]: United States    0.657
United Kingdom    0.162
Canada           0.062
Germany          0.039
Netherlands      0.025
Ireland          0.025
Australia         0.019
France           0.010
Name: country, dtype: float64
```

36.2 Counting the mental health occurrences by country

If we are interested in counting the co-occurrence of values appearing in two or more columns, then we can no longer use `value_counts` as it is a Series-only method. Instead we can use either the `groupby` or `pivot_table` methods. Let's see examples of counting the occurrences of seeking mental health treatment (the '`treatment`' column) by country.

Counting frequency with `groupby`

Use both as the grouping columns and then call the `size` method directly. You can also use the `agg` method and choose any column to aggregate as the size is the same for all groups.

```
[8]: mh.groupby(['country', 'treatment']).size().reset_index(name='count')
```

	country	treatment	count
0	Australia	No	8
1	Australia	Yes	13
2	Canada	No	33
3	Canada	Yes	35
4	France	No	11
5	France	Yes	0
6	Germany	No	24
7	Germany	Yes	19
8	Ireland	No	14
9	Ireland	Yes	13
10	Netherlands	No	18
11	Netherlands	Yes	9
12	United Kingdom	No	90
13	United Kingdom	Yes	87
14	United States	No	327
15	United States	Yes	390

The `groupby` method creates long data making it more difficult to make comparisons. Also, if one combination of country and treatment do not exist, such as France and Yes, then it won't appear in the result.

Counting frequency with `pivot_table`

We use `pivot_table` just how we did in the previous chapter for counting occurrences, but make use of the `fill_value` parameter to set any missing combinations to 0. Otherwise, a missing value would be used for the France/Yes combination.

```
[9]: mh.pivot_table(index='country', columns='treatment', aggfunc='size', fill_value=0)
```

	treatment	No	Yes
country			
Australia	8	13	
Canada	33	35	
France	11	0	
Germany	24	19	
Ireland	14	13	
Netherlands	18	9	
United Kingdom	90	87	
United States	327	390	

36.3 Counting frequency with the `crosstab` function

The `crosstab` function is built specifically for the situation of counting co-occurrences of values between two or more columns. The name comes from **cross tabulation** which is the more generic term used in data analysis outside of pandas.

Unfortunately, `crosstab` is a function and NOT a method. This means it is not bound to any DataFrame object, but must be accessed directly from `pd`. It has many of the same parameter names as the `pivot_table` method and is used similarly. Since it is not bound to any DataFrame object, you must set its parameters to Series and not strings. By default, it will compute the size of each group so there is no need to set the `aggfunc` parameter.

```
[10]: pd.crosstab(index=mh['country'], columns=mh['treatment'])
```

	treatment	No	Yes
country			
Australia	8	13	
Canada	33	35	
France	11	0	
Germany	24	19	
Ireland	14	13	
Netherlands	18	9	
United Kingdom	90	87	
United States	327	390	

Relative frequencies - only available with `crosstab`

The result is identical to what the `pivot_table` method produced. You might be wondering why there is a need to even know about this function. There is one big benefit of using the `crosstab` function is its ability to return relative frequencies with the `normalize` parameter. This isn't easily doable with `groupby`

or `pivot_table`. The `crosstab` function allows you to normalize over the rows, columns, or all the data. For instance, to find the relative frequency of people who have sought treatment in each country, you can normalize across each row like this.

```
[11]: pd.crosstab(index=mh['country'], columns=mh['treatment'], normalize='index').round(2)
```

	treatment	No	Yes
country			
Australia	0.38	0.62	
Canada	0.49	0.51	
France	1.00	0.00	
Germany	0.56	0.44	
Ireland	0.52	0.48	
Netherlands	0.67	0.33	
United Kingdom	0.51	0.49	
United States	0.46	0.54	

Set the `normalize` parameter to the '`columns`' to return the relative frequency going the other direction. The returned DataFrame informs us that of all the respondents seeking treatment, 15.4% were from the United Kingdom.

```
[12]: pd.crosstab(index=mh['country'], columns=mh['treatment'], normalize='columns').round(3)
```

	treatment	No	Yes
country			
Australia	0.015	0.023	
Canada	0.063	0.062	
France	0.021	0.000	
Germany	0.046	0.034	
Ireland	0.027	0.023	
Netherlands	0.034	0.016	
United Kingdom	0.171	0.154	
United States	0.623	0.689	

It's possible to find the relative frequency against all of the data by setting the `normalize` parameter '`all`'. From the returned DataFrame, 2.1% of all respondents are Germans who have not received mental health treatment.

```
[13]: pd.crosstab(index=mh['country'], columns=mh['treatment'], normalize='all').round(3)
```

	treatment	No	Yes
country			
Australia	0.007	0.012	
Canada	0.030	0.032	
France	0.010	0.000	
Germany	0.022	0.017	
Ireland	0.013	0.012	
Netherlands	0.016	0.008	
United Kingdom	0.082	0.080	
United States	0.300	0.357	

Adding margins

You can add margins as well by setting the `margins` parameter to `True`. Here, we go back to raw counts and add margins for all rows and columns.

```
[14]: pd.crosstab(index=mh['country'], columns=mh['treatment'], margins=True).round(3)
```

	treatment	No	Yes	All
country				
Australia	8	13	21	
Canada	33	35	68	
France	11	0	11	
Germany	24	19	43	
Ireland	14	13	27	
Netherlands	18	9	27	
United Kingdom	90	87	177	
United States	327	390	717	
All	525	566	1091	

When normalizing the data, the margins calculated depend on the direction of the normalization. Here, we add margins when normalizing down the columns. We can use this margin to determine the degree to which each country is overrepresented (or underrepresented) in each treatment category. For instance, 65.7% of the respondents were from the United States. Of those respondents seeking treatment, 68.9% were from the United States informing us that respondents from the United States were overrepresented in that category.

```
[15]: pd.crosstab(index=mh['country'], columns=mh['treatment'],
                 normalize='columns', margins=True).round(3)
```

treatment	No	Yes	All
country			
Australia	0.015	0.023	0.019
Canada	0.063	0.062	0.062
France	0.021	0.000	0.010
Germany	0.046	0.034	0.039
Ireland	0.027	0.023	0.025
Netherlands	0.034	0.016	0.025
United Kingdom	0.171	0.154	0.162
United States	0.623	0.689	0.657

Commonly called contingency tables

Cross tabulations are also known as [contingency tables](#). We can formally test whether one group has significant under or over representation with chi-square tests.

crosstab is almost unnecessary in pandas

It's important to know that `crosstab` and `pivot_table` are extremely similar and `crosstab` would be basically unnecessary if `pivot_table` had an easy way to normalize counts. The only case that necessitates `crosstab` is when creating a contingency table that normalizes the counts. Otherwise, you can use `pivot_table`.

36.4 Exercises

Exercise 1

Do people with a family history of mental illness seek treatment more often than those who do not?

[]:

Exercise 2

Find the total number and ratio of employees that seek treatment for companies that provide health benefits vs those that do not.

[]:

Exercise 3

You can provide a list of multiple columns to both the `index` and `columns` parameters of the `crosstab` function. Put country and number of employees in the index and benefits and treatment in the columns. It's probably easier to make separate list variables first.

[]:

Chapter 37

Alternate Groupby Syntax

This chapter covers a few other alternative syntaxes available to do aggregations with the `groupby` method. This chapter has great potential to confuse beginning pandas users since these methods do not give you any extra power to do data analysis. However, many other people who use pandas will use these syntaxes so it's important to be aware that they exist. Let's begin by reading in the San Francisco employee compensation dataset.

```
[1]: import pandas as pd
import numpy as np
sf_emp = pd.read_csv('../data/sf_employee_compensation.csv')
sf_emp.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

37.1 Aggregating a single column

Originally, we set our new column name equal to a two-item tuple of the aggregating column and the aggregating function within the `agg` method.

```
[2]: sf_emp.groupby('organization group').agg(mean_salary=('salaries', 'mean'))
```

organization group	mean_salary
Community Health	59002.519797
Culture & Recreation	29496.244217
General Administration & Finance	59452.791117
General City Responsibilities	33787.559971
Human Welfare & Neighborhood Development	44410.937355
Public Protection	77298.638409
Public Works, Transportation & Commerce	57852.310785

Alternative - use a dictionary

Instead of a tuple, you can use a dictionary to map the aggregating column to the aggregating function. The generic syntax takes on the following form:

```
df.groupby('grouping column').agg({'aggregating column': 'aggregating function'})
```

Although this syntax uses less code, it does not allow you to rename columns during the aggregation.

[3] : sf_emp.groupby('organization group').agg({'salaries': 'mean'})

organization group	salaries
Community Health	59002.519797
Culture & Recreation	29496.244217
General Administration & Finance	59452.791117
General City Responsibilities	33787.559971
Human Welfare & Neighborhood Development	44410.937355
Public Protection	77298.638409
Public Works, Transportation & Commerce	57852.310785

Alternative - select the column with the brackets

Instead of using a dictionary, place the aggregating columns in brackets following the groupby method and then pass the aggregating function as a string to the agg method. The generic syntax takes on the following form:

```
df.groupby('grouping column')[['aggregating column']].agg('aggregating function')
```

[4] : sf_emp.groupby('organization group')[['salaries']].agg('mean')

[4] : organization group
Community Health 59002.519797


```
max_salary=('salaries', 'max'))
```

organization group		mean_salary	min_salary	max_salary
Community Health	59002.519797	-2716.50	284024.31	
Culture & Recreation	29496.244217	0.00	237551.67	
General Administration & Finance	59452.791117	-1683.60	301104.04	
General City Responsibilities	33787.559971	0.00	365749.75	
Human Welfare & Neighborhood Development	44410.937355	-1119.86	221076.01	
Public Protection	77298.638409	-2984.52	645739.46	
Public Works, Transportation & Commerce	57852.310785	-1461.50	253688.17	

37.2 No Aggregating Columns

You actually do not need to specify the aggregating columns when grouping. Pandas will silently drop the columns that don't work for the particular aggregation method. For instance, only numeric columns have a mean. All other columns will be dropped. Here, we take the min, max, and mean of all the numeric columns for each combination of year and organization group.

```
[8]: sf_emp.groupby(['year', 'organization group']).agg(['min', 'max', 'mean']).head().round(-3)
```

year	organization group	salaries		overtime		...		health and dental		other benefits		
		min	max	mean	min	max	...	max	mean	min	max	mean
2013	Community Health	0.0	231000.0	59000.0	0.0	50000.0	...	13000.0	8000.0	2000.0	18000.0	5000.0
	Culture & Recreation	0.0	144000.0	31000.0	0.0	26000.0	...	13000.0	6000.0	-0.0	17000.0	3000.0
	General Administration & Finance	0.0	285000.0	65000.0	0.0	39000.0	...	13000.0	9000.0	0.0	18000.0	5000.0
	General City Responsibilities	0.0	37000.0	13000.0	0.0	10000.0	...	8000.0	3000.0	0.0	4000.0	1000.0
	Human Welfare & Neighborhood Development	0.0	180000.0	50000.0	0.0	20000.0	...	13000.0	9000.0	0.0	18000.0	4000.0

You can even call a method directly after grouping to apply it to all columns.

```
[9]: sf_emp.groupby(['year', 'organization group']).mean().head().round(-3)
```

year	organization group	salaries	overtime	other salaries	retirement	health and dental	other benefits
2013	Community Health	59000.0	2000.0	4000.0	11000.0	8000.0	5000.0
	Culture & Recreation	31000.0	1000.0	1000.0	5000.0	6000.0	3000.0
	General Administration & Finance	65000.0	1000.0	1000.0	12000.0	9000.0	5000.0
	General City Responsibilities	13000.0	2000.0	2000.0	3000.0	3000.0	1000.0
	Human Welfare & Neighborhood Development	50000.0	0.0	1000.0	9000.0	9000.0	4000.0

Chapter 38

Custom Aggregation

Pandas groupby objects come with many built-in aggregate functions. These are all available as strings within the `agg` method. There are, of course, many other possible aggregations that are not directly available. It is possible to define your own customized aggregate function. These customized functions must return a single value. We begin by reading in the City of Houston dataset.

```
[1]: import pandas as pd
import numpy as np
emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Writing your own custom aggregation function

Let's suppose you would like to know the difference between the max and min value of a column for each group. Pandas does not have an aggregate function built to do this. You will have to define this one yourself.

Each customized aggregate function is defined just as a regular Python function with the `def` keyword. Each function is **implicitly** passed the aggregating column, of the group, as a **Series**. This means that all Series methods will work on the passed argument. The `min_max` function below takes one argument, `s`, which is a Series object. It returns the difference between the max and min values of that Series.

```
[2]: def min_max(s):
       return s.max() - s.min()
```

38.1 Using a customized aggregation function

Customized aggregation functions are used similarly to the built-in aggregation functions. When using them within the `agg` method, use the actual function object and not the string name. The following finds the difference between the maximum and minimum salaries for each department.

```
[3]: emp.groupby('dept').agg(min_max_salary_diff=('salary', min_max))
```

dept	min_max_salary_diff
Fire	326373.0
Health & Human Services	161621.0
Houston Airport System	248875.0
Houston Public Works	246795.0
Library	160088.0
Other	250040.0
Parks & Recreation	125040.0
Police	253085.0
Solid Waste Management	167960.0

Implicit passing of aggregation Series

The above `agg` method passed the `salary` column as a Series to our customized aggregation function, `min_max`, for each group. The parameter `s` takes on this Series. We say this is implicit, because we don't actually see the function executed. An **explicit** call to `min_max` would look like this:

```
[4]: min_max(emp['salary'])
```

```
[4]: 332872.0
```

Our custom aggregation function is not passed the entire salary column as done above, but just the salary column for the current group. Let's manually recreate the grouping that pandas did by first getting the unique values of departments.

```
[5]: unique_depts = emp['dept'].drop_duplicates()
unique_depts.values
```

```
[5]: array(['Police', 'Other', 'Houston Public Works', 'Fire',
       'Health & Human Services', 'Library', 'Houston Airport System',
       'Solid Waste Management', 'Parks & Recreation'], dtype=object)
```

We can loop through these departments and filter for just the salary of that department and pass that Series into the `min_max` function. The results are then printed to the screen matching the above work done by pandas.

```
[6]: for dept in unique_depts:
    salary_group = emp.query('dept == @dept')['salary']
    min_max_diff = min_max(salary_group)
    print(f'{dept:30} {min_max_diff}')
```

Police	253085.0
Other	250040.0
Houston Public Works	246795.0
Fire	326373.0
Health & Human Services	161621.0

Library	160088.0
Houston Airport System	248875.0
Solid Waste Management	167960.0
Parks & Recreation	125040.0

Using an anonymous function

For custom aggregation functions that have just a single line of code with little logic, an anonymous function can be used. But, if there is complex logic and it is a function that you will reuse, then define a normal function. Here, we use an anonymous function to recreate the result of `min_max`.

```
[7]: emp.groupby('dept').agg(min_max_salary_diff=('salary', lambda s: s.max() - s.min()))
```

dept	min_max_salary_diff
Fire	326373.0
Health & Human Services	161621.0
Houston Airport System	248875.0
Houston Public Works	246795.0
Library	160088.0
Other	250040.0
Parks & Recreation	125040.0
Police	253085.0
Solid Waste Management	167960.0

Custom aggregation function must return a single value

The custom aggregation function you define must return a single value, or else an exception will be raised. Let's create a custom aggregation that adds 5 to each value. This will return a Series the same size as the group and not a single value.

```
[8]: def add5(s):
    return s + 5
```

Attempting to use this custom aggregation produces an error informing us that we did not produce an aggregated value.

```
[9]: emp.groupby('dept').agg(salary_plus_5=('salary', add5))
```

`ValueError: Must produce aggregated value`

Combine custom aggregation function with built-ins

The custom aggregation function can be used in conjunction with any number of other built-in aggregation functions that we have previously seen. You will have to rename the columns to remove the MultiIndex as usual.

```
[10]: emp.groupby(['dept', 'sex']) \
    .agg(min_salary=('salary', 'min'),
         max_salary=('salary', 'max'),
         min_max_salary_diff=('salary', min_max)) \
    .head(10).round(-3).style.format('{:.0f}')
```

			min_salary	max_salary	min_max_salary_diff
	dept	sex			
	Fire	Female	16,000	343,000	326,000
		Male	17,000	343,000	326,000
	Health & Human Services	Female	25,000	187,000	162,000
		Male	25,000	187,000	162,000
	Houston Airport System	Female	26,000	180,000	154,000
		Male	26,000	275,000	249,000
	Houston Public Works	Female	29,000	275,000	246,000
		Male	28,000	216,000	188,000
	Library	Female	10,000	170,000	160,000
		Male	11,000	115,000	105,000

38.2 Find the mean salary for the five highest paid employees per department

In this section, we will find the mean salary of the five highest paid employees per department. This again requires a custom aggregation function. The same Series of salaries will be implicitly passed to it. We use the `nlargest` method to select the top five salaries and then take the mean of them to return a single value.

```
[11]: def top5_mean(s):
        return s.nlargest(5).mean()
```

Let's use this function to generate the desired result.

```
[12]: emp.groupby('dept').agg(top5_average_salary=('salary', top5_mean)).round(-3)
```

top5_average_salary	
dept	
Fire	343000.0
Health & Human Services	182000.0
Houston Airport System	212000.0
Houston Public Works	206000.0
Library	132000.0
Other	225000.0
Parks & Recreation	129000.0
Police	211000.0
Solid Waste Management	148000.0

What percent of total salary do these five employees represent?

Let's take our question a step further and find out the percentage of the total salary from each department that these five employees represent. Instead of finding the mean, we take the sum of the top five salaries and divide it by the sum of all salaries for that department.

```
[13]: def top5_percent(s):
    return s.nlargest(5).sum() / s.sum() * 100
```

Again, we pass our function to the `agg` method to get the result.

```
[14]: emp.groupby('dept').agg(top5_average_salary=('salary', top5_percent)).round(2)
```

top5_average_salary	
dept	
Fire	0.65
Health & Human Services	1.22
Houston Airport System	1.58
Houston Public Works	0.48
Library	2.78
Other	0.54
Parks & Recreation	1.51
Police	0.24
Solid Waste Management	3.30

The above results tell us that the top five highest paid employees of the fire department represent 0.65% of the total combined fire department salary. Let's verify this by calculating it without using `groupby`.

```
[15]: fire_sal_sorted = emp.query('dept == "Fire"')['salary'] \
        .sort_values(ascending=False)
fire_sal_sorted.iloc[:5].sum() / fire_sal_sorted.sum() * 100
```

```
[15]: 0.6465830603130651
```

These numbers would be more meaningful if we compared them against the percentage of the number of employees that these five represent. For instance, if there are 200 fire department employees, then five would represent 2.5% ($5 / 200$). Another custom aggregation function is used below. Notice that the `count` method is used as several employees are missing salaries. If there are less than five employees in the group, 100% will be returned.

```
[16]: def count5_percent(s):
    return min(5 / s.count(), 1) * 100
```

Let's use both custom aggregation functions.

```
[17]: emp.groupby('dept').agg(top5_average_salary=('salary', top5_percent),
                           count5_percent=('salary', count5_percent)).round(2)
```

	top5_average_salary	count5_percent
dept		
Fire	0.65	0.11
Health & Human Services	1.22	0.37
Houston Airport System	1.58	0.41
Houston Public Works	0.48	0.12
Library	2.78	0.89
Other	0.54	0.15
Parks & Recreation	1.51	0.43
Police	0.24	0.08
Solid Waste Management	3.30	0.98

This gives us more meaning behind the results. The five fire department employees account for .11% of all employees, but .65% of the total salary. Since not all employees receive the same salary, we expect the top five to have a disproportional percentage of the total salary.

Using a custom aggregation function in a pivot table

As we learned, pivot tables are nearly identical to groupby aggregations. pandas allows you to set the `aggfunc` of the `pivot_table` method to a custom aggregation function. It will implicitly pass the Series of the `values` column to the aggregation function. Here we call the `top5_percent` on the `salary` column for each unique combination of department and sex.

```
[18]: emp.pivot_table(index='dept', columns='sex',
                     values='salary', aggfunc=top5_percent).round(2)
```

	sex	Female	Male
dept			
Fire	10.13	0.69	
Health & Human Services	1.64	3.82	
Houston Airport System	3.54	2.39	
Houston Public Works	1.47	0.61	
Library	3.94	7.46	
Other	0.86	1.15	
Parks & Recreation	3.98	2.10	
Police	0.92	0.30	
Solid Waste Management	9.27	4.33	

38.3 Percentage of employees by department with salaries greater than 100,000

In this section, we'll find the percentage of employees by each department with salaries greater than 100,000 using a custom aggregation function. Let's define our function below.

```
[19]: def perc_gt_100k(s):
    return (s > 100_000).mean() * 100
```

We use this custom function just like the others in this chapter to get our result.

```
[20]: emp.groupby('dept').agg(percent_greater_100k=('salary', perc_gt_100k)).round(2)
```

dept	percent_greater_100k
Fire	0.87
Health & Human Services	5.76
Houston Airport System	8.06
Houston Public Works	4.01
Library	1.60
Other	10.44
Parks & Recreation	0.78
Police	1.25
Solid Waste Management	2.34

Not accounting for missing values

Some employees do not have salaries and when we make the comparison `s > 100_000`, those missing values will evaluate as `False`. Let's find the number of employees per department that have missing values with an anonymous custom function.

```
[21]: emp.groupby('dept').agg(num_missing=('salary', lambda x: x.isna().sum()))
```

dept	num_missing
Fire	0.0
Health & Human Services	0.0
Houston Airport System	0.0
Houston Public Works	0.0
Library	0.0
Other	0.0
Parks & Recreation	0.0
Police	946.0
Solid Waste Management	0.0

If we want to find the percentage of employees with known salary that have a salary greater than 100,000, then we'll have to either drop these missing values or use a denominator that is equivalent to the number of employees with known values. Below, we use the second option and create a new custom function that uses the `count` method to count the number of employees with known salary. Only the police department's returned statistic will change, as it was the only one that had missing values.

```
[22]: def perc_gt_100k_real(s):
    return (s > 100_000).sum() / s.count() * 100

emp.groupby('dept').agg(percent_greater_100k=('salary', perc_gt_100k_real)).round(2)
```

percent_greater_100k	
dept	
Fire	0.87
Health & Human Services	5.76
Houston Airport System	8.06
Houston Public Works	4.01
Library	1.60
Other	10.44
Parks & Recreation	0.78
Police	1.43
Solid Waste Management	2.34

38.4 Optimizing a Custom Aggregation function

Custom aggregation functions have potential to be some of the slowest operations you'll perform. pandas optimizes its own built-in aggregations but cannot do the same for the ones you define. This section will show you how to optimize the performance of your custom aggregation function.

Create a larger DataFrame to test performance

Our current DataFrame contains only 24,000 rows and performance metrics matter more with larger datasets. To simulate a larger dataset, we'll use the `sample` method to select 100,000 random rows with replacement.

```
[23]: emp_large = emp.sample(100_000, replace=True).reset_index(drop=True)
emp_large.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Other	ASSISTANT DIRECTOR (EXECUTIVE LEVEL)	1992-07-28	127205.0	Female	White
1	Houston Public Works	WATER SERVICE INSPECTOR I	2009-10-19	37170.0	Male	Black
2	Houston Public Works	SEMI-SKILLED LABORER	2017-09-26	29432.0	Male	Black

Let's verify that we have a DataFrame with 100,000 rows.

```
[24]: emp_large.shape
```

```
[24]: (100000, 6)
```

Let's measure the performance of one of custom aggregation functions, `perc_gt_100k`, with this larger dataset.

```
[25]: %timeit -r 1 -n 5 emp_large.groupby('dept').agg({'salary': perc_gt_100k})
```

12.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

The number of groups can greatly impact the performance of an operation. Let's take a look at the number of unique values in each column.

```
[26]: emp_large.nunique()
```

```
[26]: dept      9
       title    690
      hire_date 3929
     salary    4063
      sex       2
     race      5
dtype: int64
```

There are 690 unique titles in our dataset. Let's use this column as the grouping column instead of department to show the difference in performance.

```
[27]: %timeit -r 1 -n 5 emp_large.groupby('title').agg({'salary': perc_gt_100k})
```

166 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

On my machine, grouping by title was about 15 times slower, showcasing the vast differences in performance when grouping by columns with different numbers of groups.

Convert grouping columns to categorical

The first step to help making your custom aggregation (and any aggregation) faster is to convert the grouping column to categorical if it is an object column containing strings. Grouping columns tend to be strings so it is a change that you'll do often. We create two new category columns for department and title and append them to the end of our DataFrame. When performing this on your own data, overwrite the original columns. We keep the original columns in this instance to show the difference in performance.

```
[28]: emp_large[['dept_cat', 'title_cat']] = emp_large[['dept', 'title']].astype('category')
```

Before testing our custom function, let's compare the performance difference of built-in `mean` function between the two versions of the department column. Grouping with the categorical column reduced the run time by two-thirds.

```
[29]: %timeit -r 1 -n 5 emp_large.groupby('dept').agg({'salary': 'mean'})
```

7.24 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

```
[30]: %timeit -r 1 -n 5 emp_large.groupby('dept_cat').agg({'salary': 'mean'})
```

2.25 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

Let's run this comparison using title as the grouping column. A similar performance improvement is seen.

```
[31]: %timeit -r 1 -n 5 emp_large.groupby('title').agg({'salary': 'mean'})
```

8.88 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

```
[32]: %timeit -r 1 -n 5 emp_large.groupby('title_cat').agg({'salary': 'mean'})
```

2.34 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

Above, we timed our performance with the `perc_gt_100k` custom aggregation function. Let's redo this operation using the new categorical columns. Run time decreased around 50% with department, but stayed the same for title.

```
[33]: %timeit -r 1 -n 5 emp_large.groupby('dept_cat').agg({'salary': perc_gt_100k})
```

6.22 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

```
[34]: %timeit -r 1 -n 5 emp_large.groupby('title_cat').agg({'salary': perc_gt_100k})
```

175 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

Complete operations that are independent of the group outside of the custom function

The next step to optimizing your custom function is to look for operations that are independent of the group. The `perc_gt_100k` function is simple and has only one line of code in its body. Let's examine the part of this line that tests whether each value is greater than 100,000.

```
s > 100_000
```

This operation is **independent** of the group. Specifically, this operation does not change whether the group is ‘Fire’, ‘Police’, ‘Library’, or any of the other groups. Since there is no dependency on the group, we can compute it on the entire DataFrame first, outside of the custom aggregation function. We store the results as a new column in our DataFrame.

```
[35]: emp_large['gt_100k'] = emp_large['salary'] > 100_000
emp_large.head(3)
```

	dept	title	hire_date	salary	sex	race	dept_cat	title_cat	gt_100k
0	Other	ASSISTANT DIRECTOR (EXECUTIVE LEVEL)	1992-07-28	127205.0	Female	White	Other	ASSISTANT DIRECTOR (EXECUTIVE LEVEL)	True
1	Houston Public Works	WATER SERVICE INSPECTOR I	2009-10-19	37170.0	Male	Black	Houston Public Works	WATER SERVICE INSPECTOR I	False
2	Houston Public Works	SEMI-SKILLED LABORER	2017-09-26	29432.0	Male	Black	Houston Public Works	SEMI-SKILLED LABORER	False

Use built-in aggregations if possible

We can now use the built-in `mean` aggregation function on the ‘gt_100k’ column to get our desired result. There is no need for the custom function anymore. Note that the results are likely to be different compared to above because `emp_large` was built from random rows.

```
[36]: emp_large.groupby('dept_cat').agg({'gt_100k': 'mean'}).round(3) * 100
```

gt_100k	
dept_cat	
Fire	0.8
Health & Human Services	5.5
Houston Airport System	7.9
Houston Public Works	3.6
Library	2.1
Other	10.7
Parks & Recreation	0.8
Police	1.2
Solid Waste Management	2.0

Built-in aggregation functions are optimized and should perform faster than custom aggregation functions doing similar tasks. Let's calculate the run time for this updated procedure taking into account the addition of the new column.

```
[37]: %%timeit -r 1 -n 5
emp_large['gt_100k'] = emp_large['salary'] > 100_000
emp_large.groupby('dept_cat').agg({'gt_100k': 'mean'}).round(3) * 100
```

5.17 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

A modest (approximately 20%) performance gain is seen when grouping by department. Let's measure performance when grouping by title.

```
[38]: %%timeit -r 1 -n 5
emp_large['gt_100k'] = emp_large['salary'] > 100_000
emp_large.groupby('title_cat').agg({'gt_100k': 'mean'}).round(3) * 100
```

5.12 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

Remarkably, the run time is nearly the same when grouping by title as it is when grouping by department and an astounding 20-30 times faster than the custom aggregation function.

Use alternative syntax

As we've learned, there are a variety of groupby aggregation syntaxes and for whatever reason pandas has implemented them differently. The following syntax where the aggregating column appears to be selected as a Series after the call to `groupby` has the best performance. Using this alternative syntax improves our performance 60x over the original.

```
df.groupby('grouping column')[‘aggregating column’].agg_func()
```

```
[39]: %%timeit -r 1 -n 5
emp_large['gt_100k'] = emp_large['salary'] > 100_000
emp_large.groupby('title_cat')['gt_100k'].mean()
```

2.18 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

Using multiple built-in groupby aggregations

To optimize performance, you should avoid custom aggregation functions altogether and opt for those built-in to pandas. In some cases, it may be necessary to use multiple groupby aggregations that use built-in pandas functions to avoid a single custom one. Take a look at the main portion of the `perc_gt_100k_real` function, which takes into account missing values.

```
(s > 100_000).sum() / s.count()
```

There are two aggregations, `sum` and `count`, which makes it appear that a custom aggregation will be necessary. Instead, we can use two groupby operations with built-in aggregation functions and then divide the results. Let's first calculate the total number of employees for each department with salaries greater than 100,000. You'll need to use the syntax that returns the result as a Series.

```
[40]: num_gt_100k = emp_large.groupby('dept_cat')['gt_100k'].sum()
num_gt_100k.head(3)
```

```
[40]: dept_cat
Fire           145.0
Health & Human Services   300.0
Houston Airport System    394.0
Name: gt_100k, dtype: float64
```

Similarly, we find the number of employees that have non-missing values for salary.

```
[41]: num_has_salary = emp_large.groupby('dept_cat')['salary'].count()
num_has_salary.head(3)
```

```
[41]: dept_cat
Fire           18081
Health & Human Services   5422
Houston Airport System    5019
Name: salary, dtype: int64
```

We can then divide these two Series together to get the desired result.

```
[42]: (num_gt_100k / num_has_salary).round(3) * 100
```

```
[42]: dept_cat
Fire           0.8
Health & Human Services   5.5
Houston Airport System    7.9
Houston Public Works     3.6
Library          2.1
Other            10.7
Parks & Recreation    0.8
Police           1.4
Solid Waste Management 2.0
dtype: float64
```

Let's measure the performance of this strategy using `title` as the grouping column and include the time it takes to create the boolean '`gt_100k`' column. An extra optimization is done, by assigning the initial grouping to variable `g`, as it is redundant to calculate it twice.

```
[43]: %%timeit -r 1 -n 5
emp_large['gt_100k'] = emp_large['salary'] > 100_000
g = emp_large.groupby('title_cat')
num_gt_100k = g['gt_100k'].sum()
num_has_salary = g['salary'].count()
(num_gt_100k / num_has_salary).round(3) * 100
```

3.35 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

We can compare that result to the performance of our custom aggregation function. The custom aggregation function performs a massive 50x worse.

```
[44]: %timeit -r 1 -n 5 emp_large.groupby('title_cat')['salary'].agg(perc_gt_100k_real)
```

```
/Users/Ted/miniconda3/envs/minimal_ds/lib/python3.7/site-
packages/ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in
long_scalars
```

189 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 5 loops each)

Instead of using multiple calls to groupby, we can instead fill in missing values in ‘gt_100k’ where there are missing values for salary. This allows us to use just the `mean` aggregation. The performance is slightly worse than with the multiple groupby calls.

```
[45]: %%timeit -n 5
emp_large['gt_100k'] = emp_large['salary'] > 100_000
filt = emp_large['salary'].isna()
emp_large.loc[filt, 'gt_100k'] = np.nan
emp_large.groupby('title_cat')['gt_100k'].mean()
```

5.2 ms ± 242 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)

38.5 Summary of Custom Aggregation Functions

- Convert grouping columns to categorical
- Complete operations that are independent of the group outside of the custom function
- Use built-in aggregations if possible
- Use alternative groupby syntax
- Use multiple built-in groupby aggregations if necessary

Complexity vs Performance

This is usually a topic of debate when deciding on which Pandas methods to use. I typically like to avoid custom aggregation functions at all cost as they can drastically reduce performance for larger datasets.

Readability (low complexity) is very valuable when sharing your code or looking back at it at a later date. Custom aggregation function might provide more readability, but usually it isn’t that much, so I tend to rarely use them as the performance difference is much more significant.

38.6 Exercises

Execute the cell below to read in the flights dataset and then use it for the following exercises.

```
[46]: import pandas as pd
flights = pd.read_csv('../data/flights.csv', parse_dates=['date'])
flights.head(3)
```

	date	airline	origin	dest	dep_time	...	carrier_delay	weather_delay	nas_delay	security_delay	late_aircraft_delay
0	2018-01-01	UA	LAS	IAH	100	...	0	0	0	0	0
1	2018-01-01	WN	DEN	PHX	515	...	0	0	0	0	0
2	2018-01-01	B6	JFK	BOS	550	...	0	83	8	0	0

Exercise 1

What are the three airlines with the least number of flights?

[]:

Exercise 2

For each airline, find the 75th percentile of flight distance. Use a custom aggregation function.

[]:

Exercise 3

For each airline, find out what percentage of its flights leave on a Tuesday. Use a custom aggregation function.

[]:

Exercise 4

Optimize exercise 2 without using a custom aggregation. What is the performance difference?

[]:

Exercise 5

The range of salaries per department was calculated using the `min_max` custom function from the beginning of this chapter. Use this same function to calculate the range of distance for each airline. Then calculate this range again without a custom function.

[]:

Exercise 6

Which origin airport has the highest percentage of its flights cancelled?

[]:

Use the college dataset

Execute the following cell which reads in a few columns from the college dataset, sets the institution name as the index and converts ‘stabbr’ and ‘relaffil’ to categorical.

```
[47]: cols = ['instnm', 'stabbr', 'relaffil', 'satvrmid', 'satmtmid', 'ugds']
college = pd.read_csv('../data/college.csv', usecols=cols,
                      index_col='instnm', dtype={'stabbr': 'category',
                                                 'relaffil': 'category'})
college.head(3)
```

		stabbr	relaffil	satvrmid	satmtmid	ugds
	instnm					
	Alabama A & M University	AL	0	424.0	420.0	4206.0
	University of Alabama at Birmingham	AL	0	570.0	565.0	11383.0
	Amridge University	AL	1	NaN	NaN	291.0

Exercise 7

How many states have more schools with a higher ‘satvrmid’ than ‘satmtmid’? Make sure to not count schools that have missing values for either one.

```
[ ]:
```

Exercise 8

Create a pivot table that shows the percentage of schools with less than 1,000 students in each state by religious affiliation. Also return the count of schools.

```
[ ]:
```

Chapter 39

Transform and Filter with Groupby

All of the groupby chapters have focused on aggregation, which is the most common operation to perform. However, there are many more things we can do to our groups besides return a single value. In this chapter, we cover the groupby `filter` method, which filters entire groups as a whole from your DataFrame and is very similar to boolean selection. We'll also cover the groupby `transform` method, which performs an operation to the entire group and returns a Series or DataFrame the same length as the original.

39.1 The groupby `filter` method

The groupby `filter` method does boolean selection for entire groups. The entire group is kept or rejected as a whole. A DataFrame with the same number of columns is returned. An example with a small fake dataset can help us learn how it works.

```
[1]: import pandas as pd
item = ['A', 'A', 'B', 'B', 'B', 'C', 'C', 'D', 'D']
quantity = [2, 10, 3, 7, 6, 5, 2, 10, 12]
df = pd.DataFrame({'item': item, 'quantity': quantity})
df
```

	item	quantity
0	A	2
1	A	10
2	B	3
3	B	7
4	B	6
5	C	5
6	C	2
7	D	10
8	D	12

Review boolean selection

Before we filter by group, let's review boolean selection from earlier in the book. With boolean selection, we create a boolean Series (usually by using one of the comparison operators) and then pass this filter to *just the brackets*. For instance, let's select all the rows with quantity greater than 5.

```
[2]: filt = df['quantity'] > 5
df[filt]
```

	item	quantity
1	A	10
3	B	7
4	B	6
7	D	10
8	D	12

Filter by group total

Instead of filtering by each individual row, we can filter entire groups. Let's say we want to keep only the items with a total quantity greater than 15. We could start by finding the total quantity using a basic groupby aggregation.

```
[3]: total = df.groupby('item').agg(total_quantity=('quantity', 'sum')).reset_index()
total
```

	item	total_quantity
0	A	12
1	B	16
2	C	7
3	D	22

We can use normal boolean selection to filter this aggregated DataFrame down to just the items that meet our criteria.

```
[4]: filt = total['total_quantity'] > 15
total[filt]
```

	item	total_quantity
1	B	16
3	D	22

Let's get just the items that meet this criteria as a Series.

```
[5]: items = total.loc[filt, 'item']
items
```

```
[5]: 1    B
      3    D
Name: item, dtype: object
```

From here we can use the `isin` method on our original DataFrame to get what we desire.

```
[6]: filt2 = df['item'].isin(items)
df[filt2]
```

	item	quantity
2	B	3
3	B	7
4	B	6
7	D	10
8	D	12

Shortcut with the groupby filter

The groupby `filter` method handles this procedure in a more direct manner. It is a somewhat complicated method so it will take some time to understand. You first must create a function that returns a single boolean value. pandas will implicitly pass this function a DataFrame consisting of just the rows of the current group.

Take a look at the `find_total` function below. It gets called once per group. It receives the current group as a DataFrame and assigns it to the variable `sub_df`. You can call any normal DataFrame methods on `sub_df`. Here, we select the quantity column and sum it. We then compare this sum against 15 and return a boolean.

```
[7]: def find_total(sub_df):
       return sub_df['quantity'].sum() > 15
```

We pass this function to the groupby `filter` method to complete the selection.

```
[8]: df.groupby('item').filter(find_total)
```

	item	quantity
2	B	3
3	B	7
4	B	6
7	D	10
8	D	12

Viewing each “Sub-DataFrame”

The name `sub_df` was chosen to signify that the object being passed to `find_total` was indeed a DataFrame. We can print out each sub-DataFrame during each call to `find_total` to get a better idea of what is happening. Let's add a print statement to it.

```
[9]: def find_total2(sub_df):
    print(sub_df, end='\n\n')
    return sub_df['quantity'].sum() > 15
```

This function will be called four times, once for each group, print out the current sub-DataFrame and the return a boolean.

```
[10]: df.groupby('item').filter(find_total2)
```

```
    item  quantity
0     A         2
1     A        10
```

```
    item  quantity
2     B         3
3     B         7
4     B         6
```

```
    item  quantity
5     C         5
6     C         2
```

```
    item  quantity
7     D        10
8     D        12
```

	item	quantity
2	B	3
3	B	7
4	B	6
7	D	10
8	D	12

39.2 Getting a nicer display

Instead of printing to the screen, we can use the `display` function from the `IPython.display` module to get the same HTML output that we are accustomed to. This can be quite helpful when debugging. The total quantity is also printed to the screen.

```
[11]: from IPython.display import display

def find_total3(sub_df):
```

```
display(sub_df)
total = sub_df['quantity'].sum()
print(f'total is {total}')
return total > 15
```

```
[12]: df.groupby('item').filter(find_total3)
```

	item	quantity
0	A	2
1	A	10

total is 12

	item	quantity
2	B	3
3	B	7
4	B	6

total is 16

	item	quantity
5	C	5
6	C	2

total is 7

	item	quantity
7	D	10
8	D	12

total is 22

	item	quantity
2	B	3
3	B	7
4	B	6
7	D	10
8	D	12

Using an anonymous function

If the custom function can be written in a single line, you may use an anonymous function. The same sub-DataFrame is passed to it like above.

```
[13]: df.groupby('item').filter(lambda sub_df: sub_df['quantity'].sum() > 15)
```

	item	quantity
2	B	3
3	B	7
4	B	6
7	D	10
8	D	12

Summary of the groupby filter method

- Must write a custom function
- The custom function implicitly gets passed a DataFrame of just that group
- The custom function must return a single boolean value
- Each group is either kept or dropped based on the returned boolean value
- The end result is the original DataFrame (same number of columns) with the rows of groups that met the criteria

39.3 Finding actors that appear in at least 25 movies

Let's complete a more practical example with the movie dataset by filtering for actors that have appeared in at least 25 movies. Only a few of the columns are read.

```
[14]: cols = ['title', 'year', 'content_rating', 'director_name', 'actor1', 'num_reviews',  
           'imdb_score']  
  
movie = pd.read_csv('../data/movie.csv', usecols=cols)  
movie.head(3)
```

	title	year	content_rating	director_name	actor1	num_reviews	imdb_score
0	Avatar	2009.0	PG-13	James Cameron	CCH Pounder	723.0	7.9
1	Pirates of the Caribbean: At World's End	2007.0	PG-13	Gore Verbinski	Johnny Depp	302.0	7.1
2	Spectre	2015.0	PG-13	Sam Mendes	Christoph Waltz	602.0	6.8

Create a custom function

Our custom function is very simple. We merely need to check if the number of rows of the implicitly passed DataFrame is 25 or more. Note, that we are only considering the actor1 column.

```
[15]: movie_top_actor = movie.groupby('actor1').filter(lambda sub_df: len(sub_df) >= 25)
movie_top_actor.head()
```

		title	year	content_rating	director_name	actor1	num_reviews	imdb_score
1	Pirates of the Caribbean: At World's End	2007.0	PG-13	Gore Verbinski	Johnny Depp	302.0	7.1	
6	Spider-Man 3	2007.0	PG-13	Sam Raimi	J.K. Simmons	392.0	6.2	
13	Pirates of the Caribbean: Dead Man's Chest	2006.0	PG-13	Gore Verbinski	Johnny Depp	313.0	7.3	
14	The Lone Ranger	2013.0	PG-13	Gore Verbinski	Johnny Depp	450.0	6.5	
18	Pirates of the Caribbean: On Stranger Tides	2011.0	PG-13	Rob Marshall	Johnny Depp	448.0	6.7	

```
[16]: movie_top_actor.shape
```

[16]: (416, 7)

Let's verify the results by returning the frequency of occurrence for each actor1 of the returned DataFrame.

```
[17]: movie_top_actor['actor1'].value_counts()
```

```
[17]: Robert De Niro      48
Johnny Depp            36
Nicolas Cage          32
Matt Damon             29
J.K. Simmons           29
Denzel Washington     29
Bruce Willis           28
Liam Neeson            27
Harrison Ford          27
Steve Buscemi          27
Robin Williams         27
Robert Downey Jr.      26
Bill Murray             26
Jason Statham          25
Name: actor1, dtype: int64
```

39.4 Multiple conditions

The custom function you create to filter your data can test as many conditions as you desire as long as it returns a single boolean value. Let's return all movies that have an actor1 with 25 or more appearances along with an average IMDB score greater than 7. We define a function that evaluates each condition.

```
[18]: def top_actor_score(sub_df):
    return len(sub_df) >= 25 and sub_df['imdb_score'].mean() > 7
```

Pass this function to the groupby filter method to get the result.

```
[19]: movie_top_actor_score = movie.groupby('actor1').filter(top_actor_score)
movie_top_actor_score.shape
```

[19]: (56, 7)

Only 56 rows remain in this filtered DataFrame than the previous one. Let's verify that each actor1 left meets both criteria.

```
[20]: movie_top_actor_score.groupby('actor1').agg(num_movies=('actor1', 'size'),
                                              mean_imdb_score=('imdb_score', 'mean'))
```

	num_movies	mean_imdb_score
actor1		
Denzel Washington	29	7.055172
Harrison Ford	27	7.159259

39.5 The groupby transform method

There are two separate ways to use the groupby `transform` method. You will perform an operation on the group that either aggregates or returns the same number of values as the group

Aggregation with `transform`

The groupby `transform` method can perform an aggregation just like the `agg` method, but will return the aggregated value for each row in the group. An example showing the difference can clear this up. Let's use our example dataset and use the groupby `agg` method to sum the quantity of each item.

```
[21]: df.groupby('item').agg(total_quantity=('quantity', 'sum'))
```

	total_quantity
item	
A	12
B	16
C	7
D	22

We can perform the same aggregation with `transform`, but it will return the same number of rows as the original. The syntax for `transform` is different than `agg`. The aggregating column (`quantity`) is placed in the brackets following the call to `groupby` and then the `transform` method is called with the string name of the aggregation. A Series is returned.

```
[22]: df.groupby('item')['quantity'].transform('sum')
```

[22]: 0 12
1 12
2 16

```

3    16
4    16
5     7
6     7
7    22
8    22
Name: quantity, dtype: int64

```

Can append this to the original DataFrame

Since `transform` always returns an object the same length as the original DataFrame, it is common to append the result to the original DataFrame.

```
[23]: df2 = df.copy()
df2['group total'] = df.groupby('item')['quantity'].transform('sum')
df2
```

	item	quantity	group total
0	A	2	12
1	A	10	12
2	B	3	16
3	B	7	16
4	B	6	16
5	C	5	7
6	C	2	7
7	D	10	22
8	D	12	22

transform second use case - return a new value for each row in the group

You can also use `transform` to apply a specific transformation to each value in the group. For instance, we can divide each value in the group by the total of that specific group. For this, we need a custom function.

```
[24]: def divide_max(sub_series):
        return sub_series / sub_series.sum()
```

The `transform` method must either return a single value or a sequence of values the same length as each group. In this instance, it returns a Series the same length as the group.

```
[25]: df2['perc_of_total'] = df.groupby('item')['quantity'].transform(divide_max).round(2)
df2
```

	item	quantity	group total	perc_of_total
0	A	2	12	0.17
1	A	10	12	0.83
2	B	3	16	0.19
3	B	7	16	0.44
4	B	6	16	0.38
5	C	5	7	0.71
6	C	2	7	0.29
7	D	10	22	0.45
8	D	12	22	0.55

Implicitly passed a Series

The `transform` method is different than `filter` in that it implicitly passes just a Series of data to the custom function. You only have access to that one Series inside of the custom function and not all of the columns like you do with `filter`. It can be instructive to print out everything that is happening within the custom function. Here, we print out both the implicitly passed original Series and the returned transformed Series.

```
[26]: def divide_max2(sub_series):
    print("Original")
    display(sub_series)
    print("Transformed")
    display(sub_series / sub_series.sum())
    print("")
    return sub_series / sub_series.sum()

df.groupby('item')['quantity'].transform(divide_max2)
```

Original

```
0      2
1     10
Name: A, dtype: int64
```

Transformed

```
0      0.166667
1      0.833333
Name: A, dtype: float64
```

Original

```
2      3
3      7
4      6
```

```
Name: B, dtype: int64
```

Transformed

```
2    0.1875  
3    0.4375  
4    0.3750
```

```
Name: B, dtype: float64
```

Original

```
5    5  
6    2  
Name: C, dtype: int64
```

Transformed

```
5    0.714286  
6    0.285714  
Name: C, dtype: float64
```

Original

```
7    10  
8    12  
Name: D, dtype: int64
```

Transformed

```
7    0.454545  
8    0.545455  
Name: D, dtype: float64
```

```
[26]: 0    0.166667  
1    0.833333  
2    0.187500  
3    0.437500  
4    0.375000  
5    0.714286  
6    0.285714  
7    0.454545  
8    0.545455  
Name: quantity, dtype: float64
```

transform must return either a single value or a Series the same length as the group

The custom function that you use with `transform` must return either a single value or a Series the same exact length as the group. Our first use-case returned an aggregation (a single value), while our second returned the Series divided by the max of each group.

Find difference from the mean

Let's read in the City of Houston employee dataset and transform each salary so that it shows the difference between it and the mean salary of that employee's department.

```
[27]: emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

We define a custom function that subtracts the mean of that group from all the values in the group.

```
[28]: def sub_mean(s):
       return (s - s.mean()).round(-3)
```

We call the `transform` method with this function and create a new column which informs us how much more or less each employee is making relative to the mean of their department.

```
[29]: emp['salary_diff_mean'] = emp.groupby('dept')['salary'].transform(sub_mean)
emp.head()
```

	dept	title	hire_date	salary	sex	race	salary_diff_mean
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White	21000.0
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic	21000.0
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black	-2000.0
3	Police	SENIOR POLICE OFFICER	1997-05-27	75942.10	Male	Hispanic	9000.0
4	Police	SENIOR POLICE OFFICER	2006-01-23	69355.26	Male	White	3000.0

39.6 Transforming multiple columns

It's possible to use the `transform` method on multiple columns instead of just one that we've been using. We begin by reading in a few columns of the college dataset.

```
[30]: cols = ['instnm', 'stabbr', 'reaffil', 'satvrmid', 'satmtmid', 'ugds']
college = pd.read_csv('../data/college.csv', usecols=cols, index_col='instnm')
college.head(3)
```

	stabbr	reaffil	satvrmid	satmtmid	ugds
instnm					
Alabama A & M University	AL	0	424.0	420.0	4206.0
University of Alabama at Birmingham	AL	0	570.0	565.0	11383.0
Amridge University	AL	1	NaN	NaN	291.0

Place all of the columns you desire to pass through the `transform` column in a list with brackets following the call to `groupby`. The following takes the mean SAT verbal and SAT math scores for each state.

```
[31]: mean_sat = college.groupby('stabbr')[['satvrmid', 'satmtmid']].transform('mean') \
        .round(0)
mean_sat.head(3)
```

	satvrmid	satmtmid
instnm		
Alabama A & M University	508.0	504.0
University of Alabama at Birmingham	508.0	504.0
Amridge University	508.0	504.0

These columns can then be appended to the original DataFrame.

```
[32]: college[['sat_verbal_mean', 'sat_math_mean']] = mean_sat
college.head(3)
```

	stabbr	reaffil	satvrmid	satmtmid	ugds	sat_verbal_mean	sat_math_mean
instnm							
Alabama A & M University	AL	0	424.0	420.0	4206.0	508.0	504.0
University of Alabama at Birmingham	AL	0	570.0	565.0	11383.0	508.0	504.0
Amridge University	AL	1	NaN	NaN	291.0	508.0	504.0

Let's filter for a different state (Texas) so verify that the mean scores are different.

```
[33]: college.query('stabbr == "TX"').head(3)
```

	stabbr	relaffil	satvrmid	satmtmid	ugds	sat_verbal_mean	sat_math_mean
instnm							
Abilene Christian University	TX	1	530.0	545.0	3572.0	511.0	523.0
Alvin Community College	TX	0	NaN	NaN	4682.0	511.0	523.0
Amarillo College	TX	0	NaN	NaN	9346.0	511.0	523.0

Standardization

A common transformation for numeric columns is to subtract the mean and divide by the standard deviation. This is called **standardization** and is usually completed before doing machine learning. It provides a relative metric of how many standard deviations away from the mean each value is. This metric is also known as the **z-score**. Let's define a custom function to produce the calculation.

```
[34]: def standardize(s):
        return (s - s.mean()) / s.std()
```

Let's standardize the SAT score and undergraduate population columns by state.

```
[35]: college.groupby('stabbr')[['satvrmid', 'satmtmid', 'ugds']].transform(standardize) \
    .round(2).head(3)
```

		satvrmid	satmtmid	ugds
instnm				
Alabama A & M University		-1.55	-1.43	0.30
University of Alabama at Birmingham		1.13	1.03	1.84
Amridge University		NaN	NaN	-0.54

Transforming all columns

If no columns are provided after the call to the `groupby` method then all columns will be transformed. If a column cannot be transformed (such as string column when taking the mean), then it will be silently dropped. Here we transform all of the numeric columns by immediately calling the `transform` method after grouping.

```
[36]: college.groupby('stabbr').transform('mean').head(3)
```

	relaffil	satvrmid	satmtmid	ugds	sat_verbal_mean	sat_math_mean
instnm						
Alabama A & M University	0.25	508.47619	504.285714	2789.865169	508.0	504.0
University of Alabama at Birmingham	0.25	508.47619	504.285714	2789.865169	508.0	504.0
Amridge University	0.25	508.47619	504.285714	2789.865169	508.0	504.0

39.7 Summary of the groupby transform method

- Syntax - df.groupby('grouping col')['transformed col'].transform(func)
- The function accepts a pandas Series of all the values in the group
- The function must return either a single value or a Series the same length as the group
- Define either a custom function or use a string name of a pandas aggregation function
- If a single value is returned from the custom function, then that value is repeated for the length of the group
- The final pandas object returned always has the same number of values as the original

39.8 Exercises

Execute the cell below to reread the college dataset and use it for the exercises below.

```
[37]: cols = ['instnm', 'stabbr', 'relaffil', 'satvrmid', 'satmtmid', 'ugds']
college = pd.read_csv('../data/college.csv', usecols=cols, index_col='instnm')
college.head(3)
```

	stabbr	relaffil	satvrmid	satmtmid	ugds
instnm					
Alabama A & M University	AL	0	424.0	420.0	4206.0
University of Alabama at Birmingham	AL	0	570.0	565.0	11383.0
Amridge University	AL	1	NaN	NaN	291.0

Exercise 1

Filter the college DataFrame for states that have more than 500,000 total undergraduate students. Can you verify your results?

```
[ ]:
```

Exercise 2

Filter the college DataFrame for states that have a an average undergraduate student population greater than 2,500 and have more than 30 religiously affiliated schools. Can you verify your results?

```
[ ]:
```

Exercise 3

The maximum SAT score for each test is 800. Create a new column in the college dataset that shows each school's percentage of maximum for each SAT score.

[]:

Use the City of Houston dataset

Execute the following cell to read in the City of Houston employee dataset and then use it for the following exercises.

```
[38]: emp = pd.read_csv('../data/employee.csv')
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Exercise 4

Filter it so that only position titles with an average salary of 100,000 remain. Can you verify your results?

[]:

Exercise 5

Filter the employee dataset so that only position titles with at least 5 employees and an average salary of 80,000 remain. Can you verify the results?

[]:

Exercise 6

Add a column to the DataFrame that contains the median salary based on department, sex, and race.

[]:

Exercise 7

Add a new column, `pct_max_dept_sex`, to the employee DataFrame that holds the employees percentage of the maximum salary for each department and sex. For instance, if a male HPD employee makes 80,000 and the maximum male HPD salary is 120,000 then the value for this employee would be 80,000/120,000 or .666. Verify this value for the first employee.

[]:

Chapter 40

Other Groupby Methods

There are many other groupby methods than `agg`, `filter`, and `transform`. In this chapter, you'll learn how to discover and use them.

40.1 Kinds of groupby attributes and methods

All groupby methods act on either a Series or a DataFrame. If there is a single column name within the brackets following the call to the `groupby` method, then it acts on a Series. If there are no brackets or multiple column names in the brackets, then it acts on a DataFrame. Let's see some examples of the two kinds of `groupby` methods available. Let's begin by reading in the San Francisco employee compensation dataset.

```
[1]: import pandas as pd
import numpy as np
sf_emp = pd.read_csv('../data/sf_employee_compensation.csv')
sf_emp.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

All grouping begins with a call to the `groupby` method, providing it the grouping column(s). Let's assign the object returned when grouping by organization group and output its type.

```
[2]: g_df = sf_emp.groupby('organization group')
type(g_df)
```

```
[2]: pandas.core.groupby.generic.DataFrameGroupBy
```

The technical name for this object is `DataFrameGroupBy` and its methods can act on the entire DataFrame. Let's call the same `groupby` method, but this time put the `salaries` column in brackets following it. A

`SeriesGroupBy` is produced and its methods can only act on the salaries Series.

```
[3]: g_series = sf_emp.groupby('organization group')['salaries']
      type(g_series)
```

```
[3]: pandas.core.groupby.generic.SeriesGroupBy
```

Explore the differences between the objects with the `agg` method

Let's use the `agg` method to explore the differences between the `g_df` and `g_series` objects. With `g_df`, you can aggregate any of the other columns, not just salaries. Here, we find the mean salary and max retirement for each organization group.

```
[4]: g_df.agg(mean_salary=('salaries', 'mean'), max_retirement=('retirement', 'max')) \
      .round(-3).style.format('{:, .0f}')
```

organization group	mean_salary	max_retirement
Community Health	59,000	65,000
Culture & Recreation	29,000	43,000
General Administration & Finance	59,000	60,000
General City Responsibilities	34,000	68,000
Human Welfare & Neighborhood Development	44,000	41,000
Public Protection	77,000	121,000
Public Works, Transportation & Commerce	58,000	48,000

With `g_series`, you only have access to the salaries column and not any other column.

```
[5]: g_series.agg(mean_salary=('salaries', 'mean')).round(-3).style.format('{:, .0f}')
```

organization group	mean_salary
Community Health	59,000
Culture & Recreation	29,000
General Administration & Finance	59,000
General City Responsibilities	34,000
Human Welfare & Neighborhood Development	44,000
Public Protection	77,000
Public Works, Transportation & Commerce	58,000

More aggregations with each object

Both `g_df` and `g_series` allow you to aggregate with multiple different syntaxes. For instance, you can pass in a list of aggregation strings to the `agg` method. For `g_df`, this will perform the aggregation on all of the non-grouping columns (and silently drop the columns where the operation fails). A multi-level column index will be returned.

```
[6]: g_df.agg(['mean', 'median']).round(-3).style.format('{:, .0f}')
```

organization group	year		salaries		overtime		other salaries		retirement		health and dental		other benefits	
	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median	mean	median
	Community Health	2,000	2,000	59,000	55,000	2,000	0	4,000	1,000	11,000	11,000	9,000	10,000	5,000
Culture & Recreation	2,000	2,000	29,000	14,000	1,000	0	1,000	0	5,000	0	6,000	5,000	3,000	1,000
General Administration & Finance	2,000	2,000	59,000	55,000	1,000	0	1,000	0	11,000	11,000	9,000	11,000	5,000	5,000
General City Responsibilities	2,000	2,000	34,000	0	3,000	0	1,000	0	7,000	0	8,000	0	3,000	0
Human Welfare & Neighborhood Development	2,000	2,000	44,000	40,000	1,000	0	1,000	0	8,000	8,000	9,000	9,000	4,000	3,000
Public Protection	2,000	2,000	77,000	82,000	12,000	2,000	7,000	4,000	16,000	16,000	12,000	13,000	3,000	2,000
Public Works, Transportation & Commerce	2,000	2,000	58,000	60,000	5,000	0	2,000	0	12,000	12,000	11,000	13,000	5,000	5,000

With `g_series`, only the salaries column will be aggregated.

```
[7]: g_series.agg(['mean', 'median']).round(-3).style.format('{:, .0f}')
```

organization group	mean	median
	59,000	55,000
Community Health	29,000	14,000
Culture & Recreation	59,000	55,000
General Administration & Finance	34,000	0
General City Responsibilities	44,000	40,000
Human Welfare & Neighborhood Development	77,000	82,000
Public Protection	58,000	60,000
Public Works, Transportation & Commerce		

Hopefully, the difference between `DataFrameGroupBy` and `SeriesGroupBy` objects are clear. The former has the entire `DataFrame` available to be acted on, while the latter only acts on the single `Series` selected after the call to `groupby`.

40.2 Finding all available attributes and methods

The vast majority of DataFrameGroupBy and SeriesGroupBy attributes and methods overlap. Let's retrieve all the public attributes and methods for each and print out the SeriesGroupBy ones to the screen.

```
[8]: public_gbs_methods = [m for m in dir(g_series) if not m.startswith('_')]
public_gbdf_methods = [m for m in dir(g_df) if not m.startswith('_')
                       and m not in sf_emp.columns]
for i, method in enumerate(public_gbs_methods):
    end = '\n' if i % 4 == 3 else ''
    print(f'{method:25}', end=end)
```

agg	aggregate	all	any
apply	backfill	bfill	corr
count	cov	cumcount	cummax
cummin	cumprod	cumsum	describe
diff	dtype	expanding	ffill
fillna	filter	first	get_group
groups	head	hist	idxmax
idxmin	indices	is_monotonic_decreasing	
is_monotonic_increasing			
last	mad	max	mean
median	min	ndim	ngroup
nunique	nlargest	nsmallest	nth
pipe	ohlc	pad	pct_change
rank	plot	prod	quantile
shift	resample	rolling	sem
sum	size	skew	std
tshift	tail	take	transform
	unique	value_counts	var

You should be familiar with many of these attributes and methods as they overlap with the ones available directly from a normal Series. We can take the set difference to determine the attributes and methods unique to each one. These are unique to SeriesGroupBy.

```
[9]: set(public_gbs_methods) - set(public_gbdf_methods)
```

```
[9]: {'dtype',
      'is_monotonic_decreasing',
      'is_monotonic_increasing',
      'nlargest',
      'nsmallest',
      'unique',
      'value_counts'}
```

These are unique to DataFrameGroupBy objects.

```
[10]: set(public_gbdf_methods) - set(public_gbs_methods)
```

```
[10]: {'boxplot', 'corrwith', 'dtypes'}
```

40.3 Calling single aggregation methods

You can bypass the `agg` method by calling the aggregation method directly from one of the `groupby` objects. The disadvantage is that you won't be able to rename the resulting column. Here, we take the maximum of the salaries column for each organization group.

```
[11]: g_series.max()
```

```
[11]: organization group
Community Health           284024.31
Culture & Recreation      237551.67
General Administration & Finance 301104.04
General City Responsibilities 365749.75
Human Welfare & Neighborhood Development 221076.01
Public Protection          645739.46
Public Works, Transportation & Commerce 253688.17
Name: salaries, dtype: float64
```

With `g_df`, the maximum of all non-grouping columns is returned. Notice that the job column shows up in the result. This is because the `max` method works for strings in an object data type column.

```
[12]: g_df.max()
```

organization group	year	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
Community Health	2019	X-Ray Laboratory Aide	284024.31	78206.57	184400.35	64849.97	29536.06	31810.05
Culture & Recreation	2019	Volunteer/Outreach Coord	237551.67	100443.69	25774.91	42640.90	29620.66	30915.83
General Administration & Finance	2019	Worker's Compensation Adjuster	301104.04	42005.20	123838.62	60099.27	29521.98	34138.52
General City Responsibilities	2019	Youth Comm Advisor	365749.75	214556.62	106461.30	68260.85	36369.96	37243.28
Human Welfare & Neighborhood Development	2019	Welfare Fraud Investigator	221076.01	41581.89	26899.60	40959.96	29522.02	33314.41
Public Protection	2019	Victim/Witness Investigator 3	645739.46	258124.17	239294.57	120791.40	36369.92	37563.46
Public Works, Transportation & Commerce	2019	Wireropecable Maint Mech Train	253688.17	152088.80	48361.88	47617.30	33495.46	33602.30

The entire syntax

It's rare that the intermediate call to the `groupby` method will be assigned to a variable as we've done in this chapter. We are only doing this to avoid the repetitive nature of calling the same method over again. When

completing these operations in practice, you'll likely begin with the original DataFrame, call the `groupby` method, and then chain the grouping method you desire. Below, the full syntax is given for the last two operations.

```
sf_emp.groupby('organization group')['salaries'].max()
sf_emp.groupby('organization group').max()
```

More aggregating methods

Most of the aggregating methods available to normal Series and DataFrames are available to their groupby counterparts. Nearly all of them return a single value for each group. However, the `describe` method returns many aggregations. You can provide it a list of percentiles to return as well. Here, we get many summary statistics for the salaries column on all the groups.

```
[13]: g_series.describe(percentiles=[.01, .2, .5, .8, .99]).round(0).style.format('{:, .0f}')
```

	count	mean	std	min	1%	20%	50%	80%	99%	max
organization group										
Community Health	9,044	59,003	47,983	2,716	0	10,551	54,653	97,752	203,195	284,024
Culture & Recreation	3,697	29,496	33,063	0	0	1,535	14,336	61,125	129,433	237,552
General Administration & Finance	3,707	59,453	51,372	1,684	0	5,906	55,100	100,346	201,274	301,104
General City Responsibilities	9,176	33,788	47,922	0	0	0	411	77,756	175,333	365,750
Human Welfare & Neighborhood Development	3,758	44,411	39,557	1,120	0	3,799	40,310	81,490	154,825	221,076
Public Protection	7,867	77,299	49,381	2,985	0	22,251	82,421	119,000	193,350	645,739
Public Works, Transportation & Commerce	12,751	57,852	41,369	1,462	0	12,168	60,108	90,459	169,203	253,688

Calling the `describe` method on `g_df` would return a very wide DataFrame with all of these statistics calculated on each numeric column.

40.4 head, tail, and nth groupby methods

The `head` and `tail` methods return the first and last five rows, respectively, of each group. Set the parameter `n` to an integer to control the number of rows returned per group. Here we return the first two rows of the entire DataFrame for each organization group. Notice that the order of the rows are preserved and they are not sorted by the grouping column.

```
[14]: g_df.head(2)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.00	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.00	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.30	24796.44	15788.97	3222.20
3	2013	Community Health	IT Operations Support Admn III	31856.00	0.00	0.00	6791.73	5262.99	2574.91
4	2013	Community Health	Special Nurse	29590.58	0.00	5898.73	960.81	0.00	9230.03
5	2013	Public Works, Transportation & Commerce	School Crossing Guard	13063.05	0.00	0.00	0.00	0.00	1031.98
6	2013	Public Works, Transportation & Commerce	Custodian	51293.81	0.00	5086.76	12227.17	12918.24	4552.84
9	2013	Culture & Recreation	Library Page	42175.61	0.00	463.19	9629.21	11124.05	3212.26
14	2013	General Administration & Finance	Prnpl Admin Analyst II	94416.74	0.00	0.00	16721.26	11063.09	7597.74
17	2013	Culture & Recreation	Recreation Leader	1821.16	0.00	27.00	0.00	664.72	143.09
36	2013	Human Welfare & Neighborhood Development	Public Svc Aide-Public Works	10172.48	0.00	0.00	0.00	4820.67	787.56
98	2013	Human Welfare & Neighborhood Development	Emp & Training Spec 2	76596.01	0.00	624.00	14061.84	12801.79	6029.32
276	2013	General City Responsibilities	Transit Operator	37274.01	5465.13	812.42	9747.52	7771.37	3421.54
723	2013	General City Responsibilities	EEO Senior Specialist	23270.40	0.00	7195.04	4230.56	0.00	2388.79

Using the same operation on a `g_series` isn't as clear as only the values of the salaries are returned without the context of the grouping column.

[15]: `g_series.head(2)`

[15]:

0	71414.01
1	67941.06
2	116956.72
3	31856.00

```
4      29590.58
5      13063.05
6      51293.81
9      42175.61
14     94416.74
17     1821.16
36     10172.48
98     76596.01
276    37274.01
723    23270.40
Name: salaries, dtype: float64
```

The `nth` groupby method allows you to control exactly which rows from the group are returned using integer location. Pass it a single integer or a list of integers. For instance, the following returns rows with integer location 5 and 10 from each group.

```
[16]: g_df.nth([5, 10])
```

organization group	year	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
Community Health	2013	Manager V	145865.95	0.00	0.00	25832.82	12801.79	17356.29
Community Health	2013	Painter	77948.03	1056.84	3323.00	15988.76	12918.24	6785.64
Culture & Recreation	2013	Executive Secretary 1	64781.63	0.00	581.87	11902.67	11937.49	5394.02
Culture & Recreation	2013	Museum Guard	16950.09	0.00	0.00	0.00	4460.62	1312.27
General Administration & Finance	2013	Sr Payroll & Personnel Clerk	73899.03	1454.65	1336.47	14637.50	12918.24	5710.47
General Administration & Finance	2013	Attorney (Civil/Criminal)	117287.47	0.00	9064.08	22402.14	8430.06	9359.44
General City Responsibilities	2013	Transit Operator	23585.59	4104.34	987.61	6288.09	4917.43	2254.10
General City Responsibilities	2013	Transit Operator	17631.69	1725.48	212.77	4567.05	3572.13	1513.98
Human Welfare & Neighborhood Development	2013	Eligibility Worker	60834.44	2511.03	1420.00	12257.78	12891.58	5238.55
Human Welfare & Neighborhood Development	2013	Senior Eligibility Worker	72393.14	0.00	1785.20	13507.87	12826.41	6206.84
Public Protection	2013	Police Officer 2	0.00	0.00	167.03	0.00	3.04	0.00
Public Protection	2013	Community Police Services Aide	66123.02	1234.17	9522.49	14443.75	12918.24	6310.89
Public Works, Transportation & Commerce	2013	Transit Supervisor	80538.20	3026.08	6682.25	15215.01	11708.04	7517.02
Public Works, Transportation & Commerce	2013	Transit Operator	11252.03	302.71	86.37	3097.66	3476.43	899.79

40.5 Non-aggregating methods

Most of the other methods do not aggregate and instead return a Series or DataFrame with the same length as the group. For the most part, they work exactly the same as they do on regular Series or DataFrames. To help teach these methods, a small fake DataFrame will be created.

```
[17]: df = pd.DataFrame({'item': ['A', 'B', 'A', 'A', 'B', 'A', 'B', 'B'],
                      'quantity': [5, 3, 8, np.nan, 2, 15, np.nan, 6]})
```

```
df
```

	item	quantity
0	A	5.0
1	B	3.0
2	A	8.0
3	A	NaN
4	B	2.0
5	A	15.0
6	B	NaN
7	B	6.0

We'll use a SeriesGroupBy object to showcase these methods.

```
[18]: g_series = df.groupby('item')['quantity']
```

All of the methods in this section preserve the order of the original values. They do NOT sort by the group. Take for instance, the `cumsum` method which accumulates the sum beginning from the top by group.

```
[19]: g_series.cumsum()
```

```
[19]: 0      5.0
1      3.0
2     13.0
3      NaN
4      5.0
5     28.0
6      NaN
7     11.0
Name: quantity, dtype: float64
```

A Series is returned, but is difficult to decipher without it being attached to the original DataFrame. Let's add it as a column and then re-examine the output.

```
[20]: df['quantity_cumsum'] = g_series.cumsum()
df
```

	item	quantity	quantity_cumsum
0	A	5.0	5.0
1	B	3.0	3.0
2	A	8.0	13.0
3	A	NaN	NaN
4	B	2.0	5.0
5	A	15.0	28.0
6	B	NaN	NaN
7	B	6.0	11.0

Each group has the quantity column accumulated independently for each group. The method `cumcount` is unique to groupby objects and provides the integer location of each row by group beginning with 0.

```
[21]: df['group_iloc'] = g_series.cumcount()
df
```

	item	quantity	quantity_cumsum	group_iloc
0	A	5.0	5.0	0
1	B	3.0	3.0	0
2	A	8.0	13.0	1
3	A	NaN	NaN	2
4	B	2.0	5.0	1
5	A	15.0	28.0	3
6	B	NaN	NaN	2
7	B	6.0	11.0	3

Each quantity can be ranked using the `rank` method. Below, the largest quantity of each group gets ranked 1.

```
[22]: df['group_rank'] = g_series.rank(ascending=False)
df
```

	item	quantity	quantity_cumsum	group_iloc	group_rank
0	A	5.0	5.0	0	3.0
1	B	3.0	3.0	0	2.0
2	A	8.0	13.0	1	2.0
3	A	NaN	NaN	2	NaN
4	B	2.0	5.0	1	3.0
5	A	15.0	28.0	3	1.0
6	B	NaN	NaN	2	NaN
7	B	6.0	11.0	3	1.0

We fill in missing values with the previous known missing value of that group with the `fillna` method.

```
[23]: df['group_ffill'] = g_series.fillna(method='ffill')
df
```

	item	quantity	quantity_cumsum	group_iloc	group_rank	group_ffill
0	A	5.0	5.0	0	3.0	5.0
1	B	3.0	3.0	0	2.0	3.0
2	A	8.0	13.0	1	2.0	8.0
3	A	NaN	NaN	2	NaN	8.0
4	B	2.0	5.0	1	3.0	2.0
5	A	15.0	28.0	3	1.0	15.0
6	B	NaN	NaN	2	NaN	2.0
7	B	6.0	11.0	3	1.0	6.0

40.6 Exercises

Execute the next cell to read in some of the columns from the flights dataset and use it to answer the following exercises.

```
[24]: import pandas as pd
cols = ['date', 'airline', 'origin', 'dest', 'dep_time', 'arr_time',
        'cancelled', 'air_time', 'distance', 'carrier_delay']
flights = pd.read_csv('../data/flights.csv', parse_dates=['date'], usecols=cols)
flights.head(3)
```

	date	airline	origin	dest	dep_time	arr_time	cancelled	air_time	distance	carrier_delay
0	2018-01-01	UA	LAS	IAH	100	547	0	134.0	1222.0	0
1	2018-01-01	WN	DEN	PHX	515	720	0	91.0	602.0	0
2	2018-01-01	B6	JFK	BOS	550	657	0	39.0	187.0	0

Exercise 1

For each airline, return the first and last row of each group. Use the `nth` group by method.

[]:

Exercise 2

For every origin and destination combination, select the 5000th flight.

[]:

Exercise 3

Find the date of the 10th cancelled flight for each airline.

[]:

Exercise 4

Find the average carrier delay for each origin and destination combination with more than 300 flights.

[]:

Chapter 41

Create Your Own Data Analysis

41.1 Overview

In this notebook, you will create your own data analysis by asking and answering your own questions on a dataset of your choice. It is an important skill to be able to explore data completely on your own without a specific direction.

Places to find data

- [Kaggle datasets](#)
- [data.world](#)
- Most large US cities, [NYC](#), [Denver](#), [US Gov't](#), simply do a web search for “open data [US city]”

Suggestions for choosing a dataset

- Find one that interests you
- For Kaggle, filter the size to those that are small (<10MB) and to those that are csv files
- Choose a dataset that contains both string and numeric columns. String columns are usually used for grouping, so its important to have a dataset with a few of them
- If you pick a dataset that has many columns (more than 20), you might want to select a small subset of them to analyze
- If you pick a dataset that has many rows (more than 100k), you might want to take a random sample of it with the `sample` method to help speed up processing

The Assignment

- Think of an interesting question that you would like answered
- Edit the markdown of the first problem below with your question
- Answer your question directly below it
- Copy and paste the markdown of the first question into a new cell to start the second question
- Continue asking and answering questions
- It may help to begin with easier questions and gradually ask more difficult ones
- Ask and answer at least 10 questions
- It is not necessary to know the answer to a question

Explore the limitations of your current knowledge

I encourage you to ask questions for which you may not know the answer. We have not quite covered all the possible tools in Pandas so you will not be able to answer everything.

Compiling Results

Once you have completed the exploration, email Ted a single notebook (an .ipynb file) along with the data. He will consolidate all the notebooks into a single notebook. He will then provide answers to the questions and send out the compiled results back to you.

MAKE YOUR NOTEBOOKS CLEAN!!!

It's important that the notebook you submit is clean without random code or extra work not related to the problems. They should resemble my solution notebooks. Make sure you append the `head` method to your DataFrame and Series output.

Use this notebook as a rough draft and use a second for the final copy

Use this notebook as a rough draft. When you are finished asking and answering your questions, make a copy of this notebook using the **File** menu above and leave just questions and answers. This will be the final copy you submit to me. To check that your notebook runs all cells successfully, select **Run All** from the **Cell** menu above. Make sure to send me the data file as well.

41.2 Begin Asking and Answering Questions

Exercise 1

Double-click this cell and replace this sentence with a question of your own

[]:

Part VII

Time Series

Chapter 42

Datetime and Timedelta

This chapter covers two distinct concepts, datetimes and timedeltas, and how they are created and used in pandas. A datetime represents a specific **moment** in time. A timedelta represents an **amount** of time.

42.1 Date vs Time vs Datetime

There is a distinction that needs to be made between the terms **date**, **time**, and **datetime**. They all three mean different things.

- **date** - Only the month, day, and year. So 2016-01-01 would represent January 1, 2016 and be considered a **date**.
- **time** - Only the hours, minutes, seconds, and parts of a second (milli, micro, nano, etc...). 5 hours, 45 minutes and 6.74234 seconds for example would be considered a **time**.
- **datetime** - A combination of a date and time. It has both date (year, month, day) and time (hour, minute, second, part of second) components. January 1, 2016 at 5:45 p.m. would be an example of a **datetime**.

The Python standard library contains the [datetime module](#). It is a popular and important module, but will not be covered here since pandas builds its own datetime and timedelta objects that are more powerful.

Datetimes in numpy

In part 5 of this book, we covered the numpy datetime data type. It is more powerful and flexible than core Python's datetime module, but does not have the features of the pandas datetime object. This chapter only covers datetimes in pandas.

42.2 Creating single datetime objects in pandas

In part 5, we used the Series constructor to create Series of datetimes. It's actually possible to create single datetime objects with the `to_datetime` function and the `Timestamp` constructor.

Creating a single datetime with the `to_datetime` function

The `to_datetime` function converts it's inputs to a single scalar datetime with nanosecond precision. These are analogous to single integers, floats, or strings. They are not part of an array, Series, or DataFrame. It can take a variety of different of different inputs. We begin by converting a string with the format 'YYYY-MM-DD' to a datetime.

```
[1]: import pandas as pd
d = pd.to_datetime('2020-01-05')
d
```

```
[1]: Timestamp('2020-01-05 00:00:00')
```

This is a new type of object. Let's retrieve its type.

```
[2]: type(d)
```

```
[2]: pandas._libs.tslibs.timestamps.Timestamp
```

Why is a `Timestamp` object returned?

The type that pandas uses for individual datetimes is `Timestamp`. In general, the word ‘timestamp’ has the same meaning as `datetime`. If you look at the docstring for `to_datetime` it states the following:

Convert argument to `datetime`.

It would have been nice if pandas had chosen the name `Datetime` for the type so that it could match the name of the data type and function, but it did not, and thus there is potential for confusion. Let's create a Series of datetimes to show that the data type is '`datetime64[ns]`'.

```
[3]: s = pd.Series(['2020-01-05', '2020-01-06'], dtype='datetime64[ns]')
s
```

```
[3]: 0    2020-01-05
 1    2020-01-06
dtype: datetime64[ns]
```

When selecting a single value from this Series, a `Timestamp` object is returned. In the official documentation, both words ‘timestamp’ and ‘`datetime`’ are used interchangeably to refer to the same concept - an object with year, month, day, hour, minute, second, and part of second components.

```
[4]: s.loc[0]
```

```
[4]: Timestamp('2020-01-05 00:00:00')
```

More string formats

Let's see more examples of strings with different formats that can be converted to datetimes. Here, we use a hyphen to separate the components but do not place the leading zero in front of the month and day. It's important to remember that `to_datetime` is a function and not a Series or DataFrame method. It must be accessed directly from `pd`.

```
[5]: pd.to_datetime('2016-1-5')
```

```
[5]: Timestamp('2016-01-05 00:00:00')
```

The hour, minute, second, and part of second components were not explicitly given, so pandas sets them to 0. Let's slowly create more datetimes by adding one more component each time. Here, we add the hour.

```
[6]: pd.to_datetime('2020-1-5 15')
```

```
[6]: Timestamp('2020-01-05 15:00:00')
```

The hour and minute are separated by a colon.

```
[7]: pd.to_datetime('2020-1-5 15:39')
```

```
[7]: Timestamp('2020-01-05 15:39:00')
```

The minute and second are also separated by a colon.

```
[8]: pd.to_datetime('2020-1-5 15:39:55')
```

```
[8]: Timestamp('2020-01-05 15:39:55')
```

The part of second needs to be separated from the second by a decimal. There is only enough precision to contain nanoseconds, which are nine places after the decimal. The last two decimal places are truncated below.

```
[9]: pd.to_datetime('2020-1-5 15:39:55.12345678912')
```

```
[9]: Timestamp('2020-01-05 15:39:55.123456789')
```

Forward slashes can be used instead of hyphens to separate the date components. The hour, minute, and second components do not require any separator.

```
[10]: pd.to_datetime('2020/01/05 15:39:55.123456789')
```

```
[10]: Timestamp('2020-01-05 15:39:55.123456789')
```

The date components also don't need a separator.

```
[11]: pd.to_datetime('20200105 15:39:55.123456789')
```

```
[11]: Timestamp('2020-01-05 15:39:55.123456789')
```

You can also use the month name spelled out as a string, have an ending for the day, and use AM/PM to denote part of day.

```
[12]: pd.to_datetime('January 5th, 2020 03:39:55 PM')
```

```
[12]: Timestamp('2020-01-05 15:39:55')
```

Epoch

The term epoch refers to the origin of a particular era. Like many other programming languages, Python uses January 1, 1970 (also known as the Unix epoch) as its epoch for keeping track of datetime. In pandas, integers are used to represent the number of nanoseconds that have elapsed since the epoch.

Converting numbers to Timestamps

The `to_datetime` function also accepts numbers and converts them to Timestamps. By default, it uses nanoseconds as the units for the passed number. The following creates a datetime 100 nanoseconds after January 1, 1970.

```
[13]: pd.to_datetime(100)
```

```
[13]: Timestamp('1970-01-01 00:00:00.000000100')
```

Specify unit

The default unit is nanoseconds, but you can specify a different one with the `unit` parameter. Here, we create a datetime 100 seconds after the epoch.

```
[14]: pd.to_datetime(100, unit='s')
```

```
[14]: Timestamp('1970-01-01 00:01:40')
```

Here a datetime 20,000 days after the epoch is created.

```
[15]: pd.to_datetime(20_000, unit='d')
```

```
[15]: Timestamp('2024-10-04 00:00:00')
```

Datetimes in DataFrames

It's more common to encounter datetimes in a DataFrame. Let's read in the City of Houston employee dataset converting the `hire_date` column to a datetime.

```
[16]: emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
emp.dtypes
```

```
[16]: dept          object
      title         object
      hire_date    datetime64[ns]
      salary        float64
      sex           object
      race          object
      dtype: object
```

Each individual value in the datetime columns is a Timestamp

If we extract the `hire_date` column as a Series and print out the first few rows, you will see that data type (at the bottom of the output) is still written with the word `datetime64[ns]`.

```
[17]: hire_date = emp['hire_date']
      hire_date.head()
```

```
[17]: 0    2001-12-03
      1    2010-11-15
      2    2006-01-09
      3    1997-05-27
      4    2006-01-23
      Name: hire_date, dtype: datetime64[ns]
```

If we select the first value in the Series, we get a Timestamp.

```
[18]: hire_date.loc[0]
```

```
[18]: Timestamp('2001-12-03 00:00:00')
```

42.3 Timestamp attributes and methods

Timestamp objects have similar attributes and methods as the `dt` Series accessor. Let's create a Timestamp and retrieve see some of these attributes.

```
[19]: ts = pd.to_datetime('February 14, 2019 20:45.56')
```

```
[20]: ts.year
```

```
[20]: 2019
```

```
[21]: ts.month
```

```
[21]: 2
```

```
[22]: ts.day
```

```
[22]: 14
```

```
[23]: ts.second
```

```
[23]: 33
```

```
[24]: ts.weekofyear
```

```
[24]: 7
```

```
[25]: ts.dayofyear
```

```
[25]: 45
```

Several methods also exist, which we use below.

```
[26]: ts.day_name()
```

```
[26]: 'Thursday'
```

```
[27]: ts.month_name()
```

```
[27]: 'February'
```

The offset aliases are used for the `round`, `ceil`, and `floor` methods. Here, we round to the nearest hour.

```
[28]: ts.round('H')
```

```
[28]: Timestamp('2019-02-14 21:00:00')
```

42.4 Timedelta - an amount of time

A timedelta is a specific amount of time such as 20 seconds, or 13 days 5 minutes and 10 seconds. Use the `to_timedelta` function to create a Timedelta object. It works analogously to the `to_datetime` function. A wide variety of strings are able to be converted to Timedeltas.

```
[29]: pd.to_timedelta('5 days 03:12:45.123')
```

```
[29]: Timedelta('5 days 03:12:45.123000')
```

```
[30]: pd.to_timedelta('10h 13ms')
```

```
[30]: Timedelta('0 days 10:00:00.013000')
```

Converting numbers to Timedeltas with `to_timedelta`

As with `to_datetime`, passing a number to `to_timedelta` will be by default treated as the number of nanoseconds. Use the `unit` parameter to change the time unit. We start by converting 123,000 nanoseconds to a timedelta.

```
[31]: pd.to_timedelta(123_000)
```

```
[31]: Timedelta('0 days 00:00:00.000123')
```

Here, we create a timedelta of exactly 500 days.

```
[32]: pd.to_timedelta(500, unit='d')
```

```
[32]: Timedelta('500 days 00:00:00')
```

Over 700 hours converted to a timedelta.

```
[33]: pd.to_timedelta(705.87, unit='h')
```

```
[33]: Timedelta('29 days 09:52:12')
```

Since years is not a standard amount, you'll get an error if you use it's unit abbreviation, 'y'. Month is also not a standard unit so you won't be able to use it either.

```
[34]: pd.to_timedelta(23, unit='y')
```

```
ValueError: Units 'M' and 'Y' are no longer supported, as they do not represent unambiguous timedelta values durations.
```

No name confusion with Timedelta

Pandas Timedelta is built upon numpy's timedelta64 data type which is superior to the standard library's datetime module's timedelta. Fortunately, the pandas developers used the name timedelta for the data type which is the same as numpy's. There is no name confusion here, unlike there is with timestamp/datetime.

42.5 Timedelta attributes and methods

There are many attributes and methods available to Timedelta objects. Let's see some below:

```
[35]: td = pd.to_timedelta(705.87, unit='h')
td
```

```
[35]: Timedelta('29 days 09:52:12')
```

```
[36]: td.days
```

```
[36]: 29
```

```
[37]: td.seconds
```

```
[37]: 35532
```

```
[38]: td.components
```

```
[38]: Components(days=29, hours=9, minutes=52, seconds=12, milliseconds=0, microseconds=0,
nanoseconds=0)
```

42.6 Creating timedeltas by subtracting datetimes

It is possible to create timedeltas by subtracting two datetimes.

```
[39]: dt1 = pd.to_datetime('2012-12-21 5:30')
dt2 = pd.to_datetime('2016-1-1 12:45:12')
dt2 - dt1
```

```
[39]: Timedelta('1106 days 07:15:12')
```

Negative Timedeltas

A negative timedelta is possible just like any negative number is.

```
[40]: dt1 - dt2
```

```
[40]: Timedelta('-1107 days +16:44:48')
```

Math with Timedeltas

You can do many different math operations with two timedeltas together.

```
[41]: td1 = pd.to_timedelta('05:23:10')
td2 = pd.to_timedelta('00:02:20')
td1 - td2
```

```
[41]: Timedelta('0 days 05:20:50')
```

```
[42]: td2 + 5 * td2
```

[42]: `Timedelta('0 days 00:14:00')`

Dividing two timedeltas will remove the units and return a number.

[43]: `td1 / td2`

[43]: 138.5

Creating Timedeltas in a DataFrame by subtracting two Datetime columns

The bikes dataset has two datetime columns, `starttime` and `stoptime`.

[44]: `bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])
bikes.head(2)`

	trip_id	gender	starttime	stoptime	tripduration	...	temperature	visibility	wind_speed	precipitation	events
0	7147	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	...	73.9	10.0	12.7	-9999.0	mostlycloudy
1	7524	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	...	69.1	10.0	6.9	-9999.0	partlycloudy

Let's find the amount of time that elapsed between the start and stop times.

[45]: `time_elapsed = bikes['stoptime'] - bikes['starttime']
time_elapsed.head()`

[45]: 0 00:16:00
1 00:10:00
2 00:18:00
3 00:11:00
4 00:02:00
dtype: timedelta64[ns]

Since both start and stop time are datetime columns, subtracting them resulted in a timedelta column. The maximum unit of time for timedelta is days.

42.7 Exercises

Exercise 1

What day of the week was Jan 15, 1997?

[]:

Exercise 2

Was 1925 a leap year?

[]:

Exercise 3

What year will it be 1 million hours after the UNIX epoch?

[]:

Exercise 4

Create the datetime July 20, 1969 at 2:56 a.m. and 15 seconds.

[]:

Exercise 5

Neil Armstrong stepped on the moon at the time in the last exercise. How many days have passed since that happened? Use the string ‘today’ when creating your datetime.

[]:

Exercise 6

Which is larger - 35 days or 700 hours?

[]:

Exercise 7

The City of Houston employee data was retrieved on June 1, 2019. Can you calculate the exact amount of years of experience and assign as a new column named **experience**?

[]:

Chapter 43

Introduction to Time Series

Broadly speaking, time series data are points of data gathered over time. The time order is meaningful and typically there is only one observation per unit of time. The time will uniquely identify each record. Often, the time is evenly spaced between each data point.

Examples of time series data include stock market closing prices, levels of CO₂ in the atmosphere, unemployment rates, etc... pandas has good functionality with regards to manipulating dates, aggregating over different time periods, sampling different periods of time, and more. Let's begin by reading in 20 years of stock market data, putting the 'date' column in the index.

```
[1]: import pandas as pd
stocks = pd.read_csv('../data/stocks/stocks10.csv', parse_dates=['date'],
                     index_col='date')
stocks.head(3)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN

43.1 Set the datetime column as the index

If you do have time series data where the values of one datetime column uniquely identify each row, then it's best to use this column as the index. pandas provides extra functionality to DataFrames that have a datetime index.

DateTimeIndex

Setting a datetime column as the index technically creates a DateTimeIndex. You can directly call specific datetime methods on it like you can with the `dt` accessor. Let's extract it and examine the first five values.

```
[2]: idx = stocks.index
idx[:5]
```

```
[2]: DatetimeIndex(['1999-10-25', '1999-10-26', '1999-10-27', '1999-10-28',
                   '1999-10-29'],
                   dtype='datetime64[ns]', name='date', freq=None)
```

Let's verify the type of index we have.

```
[3]: type(idx)
```

```
[3]: pandas.core.indexes.datetimes.DatetimeIndex
```

Now, let's get the year, month and weekday name directly from this index object. The first five values for each attribute are returned.

```
[4]: idx.year[:5]
```

```
[4]: Int64Index([1999, 1999, 1999, 1999, 1999], dtype='int64', name='date')
```

```
[5]: idx.month[:5]
```

```
[5]: Int64Index([10, 10, 10, 10, 10], dtype='int64', name='date')
```

```
[6]: idx.day_name()[:5]
```

```
[6]: Index(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'],
          dtype='object', name='date')
```

43.2 Easy subset selection with a DateTimeIndex

One big advantage of a DateTimeIndex is the ability to select subsets of data without using boolean indexing. We can use strings to represent specific datetimes and pass those strings to the `loc` indexer. Here, we select the data for January 5th, 2017.

```
[7]: stocks.loc['2017-1-5']
```

```
[7]: MSFT      59.23
AAPL     111.73
SLB      76.93
AMZN    780.45
TSLA    226.75
XOM      79.11
WMT      64.88
T       36.08
FB      120.67
V       79.61
Name: 2017-01-05 00:00:00, dtype: float64
```

Partial string matching to select entire months or years

You can select entire years or months (or other spans of time) by using a string with less precision. In the following example, we select the entire month of February, 2017.

```
[8]: stocks.loc['2017-2'].head(3)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
2017-02-01	60.45	123.36	75.01	832.35	249.24	74.10	62.09	35.99	133.23	80.94
2017-02-02	60.06	123.15	74.35	839.95	251.55	74.55	62.53	35.24	130.84	80.80
2017-02-03	60.54	123.68	74.40	810.20	251.33	74.63	62.34	35.30	130.98	84.51

Below, we select the entire year 2016.

```
[9]: stocks.loc['2016'].head(3)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
2016-01-04	50.71	98.74	60.79	636.99	223.41	66.84	55.99	27.65	102.22	73.61
2016-01-05	50.94	96.27	61.07	633.79	223.43	67.41	57.32	27.85	102.73	74.16
2016-01-06	50.01	94.38	59.49	632.65	219.04	66.85	57.90	27.81	102.97	73.19

Slicing with partial string matching

Use slice notation to select a specific date range. Below, we select from March 28, 2017 through April 3, 2017. Note that the stop value is inclusive.

```
[10]: stocks['2017-3-28':'2017-4-3']
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
2017-03-28	62.45	138.38	71.06	856.00	277.45	73.78	66.41	35.56	141.76	87.66
2017-03-29	62.62	138.68	71.39	874.32	277.38	73.94	66.81	35.47	142.65	87.72
2017-03-30	62.85	138.50	70.63	876.34	277.92	75.46	67.61	35.74	142.41	87.55
2017-03-31	62.99	138.24	70.87	886.54	278.30	73.93	68.07	35.56	142.05	87.41
2017-04-03	62.70	138.28	70.50	891.51	298.52	73.99	67.83	35.57	142.28	87.90

43.3 Sampling specific times

Let's say you are interested in selecting the closing prices for the last day of every year in the dataset. pandas provides the `asfreq` method to do so. You must pass it an **offset alias** as a string. An offset alias determines the frequency of the time series data you would like to sample. The table below shows the most common offset aliases. To reference all of the [offset aliases](#), visit in the official documentation.

Alias	Description	Alias	Description
Y	year end	D	day
YS	year start	H	hourly

Alias	Description	Alias	Description
Q	quarter end	T or min	minutes
QS	quarter start	S	seconds
M	month end	L or ms	milliseconds
MS	month start	U or us	microseconds
W	weekly	N	nanoseconds

In our example, we need the offset alias 'Y' for the year end frequency. We pass this as a string to the `asfreq` method to return the very last day of the each year. Note that `asfreq` only works for DataFrames with a `DateTimeIndex`.

```
[11]: stocks.asfreq('Y').head(8)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-12-31	37.68	3.20	18.11	76.12	NaN	23.49	48.21	18.49	NaN	NaN
2000-12-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2001-12-31	21.38	1.37	18.58	10.82	NaN	24.18	40.56	15.52	NaN	NaN
2002-12-31	16.69	0.89	14.68	18.89	NaN	22.03	35.79	11.13	NaN	NaN
2003-12-31	17.82	1.33	19.72	52.62	NaN	26.58	37.84	11.20	NaN	NaN
2004-12-31	19.45	4.01	24.72	44.29	NaN	34.04	38.03	11.62	NaN	NaN
2005-12-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2006-12-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Business offset aliases

This isn't quite what we want because the stock market is open only during the week and December 31st falls on a weekend some years. The `asfreq` method returns one row for each frequency regardless if there is data for that date. All values for frequencies that do not appear in the DataFrame will be filled with missing values.

Most of the offset aliases above can be prepended by the character 'B' to signify a business offset alias. Business offset aliases only consider the weekdays Monday through Friday. Let's change the offset alias to 'BY' to signify business year end frequency. Using this, we correctly select the last trading day of each year.

```
[12]: stocks.asfreq('BY').head(8)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-12-31	37.68	3.20	18.11	76.12	NaN	23.49	48.21	18.49	NaN	NaN
2000-12-29	14.00	0.93	26.32	15.56	NaN	25.89	37.23	18.50	NaN	NaN
2001-12-31	21.38	1.37	18.58	10.82	NaN	24.18	40.56	15.52	NaN	NaN
2002-12-31	16.69	0.89	14.68	18.89	NaN	22.03	35.79	11.13	NaN	NaN
2003-12-31	17.82	1.33	19.72	52.62	NaN	26.58	37.84	11.20	NaN	NaN
2004-12-31	19.45	4.01	24.72	44.29	NaN	34.04	38.03	11.62	NaN	NaN
2005-12-30	19.27	8.96	36.62	47.15	NaN	38.05	34.12	11.64	NaN	NaN
2006-12-29	22.32	10.58	48.10	39.46	NaN	52.92	34.16	17.83	NaN	NaN

Anchored offset aliases

Let's say we would like to select every Thursday. We'll need to use a slightly different string called an **anchored offset alias**. You can anchor years and quarters to months and weeks to days by placing a dash and the abbreviation of the anchor after the offset alias. For example, BY-APR signifies business year frequency ending in April. Below, we anchor weeks to Thursday. The default anchor for weeks is Sunday.

```
[13]: stocks.asfreq('W-THU').head()
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-28	29.01	2.43	16.59	71.00	NaN	21.19	38.85	19.79	NaN	NaN
1999-11-04	29.61	2.61	17.09	63.06	NaN	21.16	39.03	19.39	NaN	NaN
1999-11-11	28.93	2.88	17.57	73.00	NaN	22.40	40.03	19.39	NaN	NaN
1999-11-18	27.42	2.79	18.95	77.94	NaN	23.63	41.38	19.47	NaN	NaN
1999-11-25	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

43.4 Upsampling - Increasing the number of rows

The above selections choose a specific subset of rows. This is called **downsampling** when we select a subset of the original data. Instead, we may choose to **upsample** and increase the number of rows. This will lead to rows of all missing values. Both upsampling and downsampling ensure that the rows are evenly spaced units of time. Let's return a DataFrame with a single row for each day of the year. Currently, only the trading days are in the dataset.

```
[14]: stocks.asfreq('D').head(7)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN
1999-10-28	29.01	2.43	16.59	71.00	NaN	21.19	38.85	19.79	NaN	NaN
1999-10-29	29.88	2.50	17.21	70.62	NaN	21.47	39.25	20.00	NaN	NaN
1999-10-30	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-31	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Use integers in the offset alias

You can provide more precise offsets by placing an integer in front of the offset alias. These represent a multiple the of offset alias. For example, ‘3M’ stands for 3 months and ‘15s’ for 15 seconds. To select every 6th Wednesday, we do the following:

```
[15]: stocks.asfreq('6W-WED').head()
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN
1999-12-08	29.61	3.43	16.88	88.56	NaN	24.34	40.88	20.50	NaN	NaN
2000-01-19	34.54	3.32	21.25	66.81	NaN	24.95	44.68	15.46	NaN	NaN
2000-03-01	29.31	4.06	25.00	65.88	NaN	22.30	34.18	16.01	NaN	NaN
2000-04-12	25.62	3.41	23.76	56.38	NaN	23.44	43.60	18.02	NaN	NaN

You can also upsample by smaller units than what is present in the index. For instance, ‘4H’ will make a new row for every 4 hours.

```
[16]: stocks.asfreq('4H').head(8)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25 00:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-25 04:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-25 08:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-25 12:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-25 16:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-25 20:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1999-10-26 00:00:00	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-26 04:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

You can fill in the missing values with the previous or next known values using the `method` parameter which can be set to either ‘ffill’ or ‘bfill’. Here we fill the missing values using the previously known value in the column.

```
[17]: stocks.asfreq('4H', method='ffill').head(8)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25 00:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-25 04:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-25 08:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-25 12:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-25 16:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-25 20:00:00	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26 00:00:00	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-26 04:00:00	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN

No duplicates are allowed and dates must be ordered

Upsampling and downsampling work properly when there are no duplicate dates and when the data is ordered. Let’s take the employee dataset which has a datetime column, but is definitely not time series data.

```
[18]: emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
emp = emp.set_index('hire_date')
emp.head(3)
```

	dept		title	salary	sex	race
hire_date						
2001-12-03	Police	POLICE SERGEANT	87545.38	Male	White	
2010-11-15	Other	ASSISTANT CITY ATTORNEY II	82182.00	Male	Hispanic	
2006-01-09	Houston Public Works	SENIOR SLUDGE PROCESSOR	49275.00	Male	Black	

If we try and sample it by Year (which is meaningless in this dataset) we get an empty DataFrame.

```
[19]: emp.asfreq('Y')
```

	dept	title	salary	sex	race
hire_date					

Even if we try and make it more like a time series by sorting the index, the operation will only be successful if there are no duplicate dates. The error tells us that at least one hire date is not unique.

```
[20]: emp = emp.sort_index()
emp.head(3)
```

	dept		title	salary	sex	race
hire_date						
1968-12-13	Police	SENIOR POLICE OFFICER	NaN	Male	Black	
1969-03-21	Police	POLICE SERGEANT	NaN	Male	Hispanic	
1969-10-06	Other	SENIOR PUBLIC LOSS INVESTIGATOR	75067.0	Female	White	

```
[21]: emp.asfreq('W')
```

`ValueError: cannot reindex from a duplicate axis`

Selection with partial string still works.

```
[22]: emp.loc['2012-1':'2012-2'].head()
```

	dept		title	salary	sex	race
hire_date						
2012-01-03	Other		COUNCIL MEMBER	62983.0	Female	White
2012-01-03	Other		COUNCIL MEMBER	62983.0	Male	White
2012-01-03	Other	DEPUTY DIRECTOR (EXECUTIVE LEVEL)	142170.0	Female	Black	
2012-01-03	Police	CRIMINAL INTELLIGENCE ANALYST	59322.0	Male	White	
2012-01-03	Health & Human Services	SURVEILLANCE INVESTIGATOR-EPIDEMILOGY	46654.0	Female	Black	

43.5 Exercises

Exercise 1

Read in the weather time series dataset and place the date column in the index.

[]:

Exercise 2

What was the temperature on June 11, 2011?

[]:

Exercise 3

How many days did it rain during the last three months of 2011?

[]:

Exercise 4

Which year had more snow days, 2007 or 2012?

[]:

Exercise 5

Select every other thursday

[]:

Exercise 6

Select the first day of each month.

[]:

Chapter 44

Grouping by Time

In previous notebooks, we learned how to downsample/upsample time series data. In this notebook, we will group spans of time together to get a result. For instance, we can find out the number of up or down days for a stock within each trading month, or calculate the number of flights per day for an airline. pandas gives you the ability to group by a period of time. Let's begin by reading in our stock dataset.

```
[1]: import pandas as pd  
stocks = pd.read_csv('../data/stocks/stocks10.csv', parse_dates=['date'],  
                     index_col='date')  
stocks.head(3)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN

Find the average closing price of Amazon for every month

If we are interested in finding the average closing price of Amazon for every month, then we need to group by month and aggregate the closing price with the mean function.

Grouping column, aggregating column, and aggregating method

This procedure is very similar to how we grouped and aggregated columns in previous notebooks. The only difference is that our grouping column will now be a datetime column with an additional specification for the amount of time.

Use the `resample` method

Instead of the `groupby` method, we use a special method for grouping time together called `resample`. We must pass the `resample` method an offset alias string. The rest of the process is the exact same as the `groupby` method. We call the `agg` method and pass it a dictionary mapping the aggregating columns to the aggregating functions.

resample syntax

The first parameter we pass to `resample` is the `offset alias`. Here, we choose to group by month. We then chain the `agg` method and must use one of the alternative syntaxes as the pandas developers have not yet implemented column renaming for the `resample` method.

```
[2]: stocks.resample('M').agg({'AMZN': 'mean'}).head(3)
```

AMZN	
	date
1999-10-31	76.312000
1999-11-30	76.240952
1999-12-31	91.070000

Use any number of aggregation functions

Map the aggregating column to a list of aggregating functions.

```
[3]: stocks.resample('M').agg({'AMZN': ['size', 'min', 'mean', 'max']}).head(3)
```

AMZN				
	size	min	mean	max
	date			
1999-10-31	5	70.62	76.312000	82.75
1999-11-30	21	63.06	76.240952	93.12
1999-12-31	22	76.12	91.070000	106.69

Group by Quarter

```
[4]: stocks.resample('Q').agg({'AMZN': ['size', 'min', 'mean', 'max']}).head(4)
```

AMZN				
	size	min	mean	max
	date			
1999-12-31	48	63.06	83.045000	106.69
2000-03-31	63	61.69	68.957302	89.38
2000-06-30	63	33.88	51.630794	67.56
2000-09-30	63	30.00	38.105079	45.88

Label as the entire Period

Notice how the end date of both the month and day are used as the returned index labels for the time periods. We can change the index labels so that they show just the time period we are aggregating over by setting the `kind` parameter to ‘period’.

```
[5]: amzn_period = stocks.resample('Q', kind='period').agg({'AMZN': ['size', 'min', 'mean', 'max']})
amzn_period.head(4)
```

AMZN				
	size	min	mean	max
date				
1999Q4	48	63.06	83.045000	106.69
2000Q1	63	61.69	68.957302	89.38
2000Q2	63	33.88	51.630794	67.56
2000Q3	63	30.00	38.105079	45.88

44.1 The PeriodIndex

We no longer have a DatetimeIndex. Pandas has a completely separate type of object for this called the **PeriodIndex**. The index label ‘2016Q1’ refers to the entire period of the first quarter of 2016. Let’s inspect the index to see the new type.

```
[6]: amzn_period.index[:10]
```

```
[6]: PeriodIndex(['1999Q4', '2000Q1', '2000Q2', '2000Q3', '2000Q4', '2001Q1',
                 '2001Q2', '2001Q3', '2001Q4', '2002Q1'],
                dtype='period[Q-DEC]', name='date', freq='Q-DEC')
```

44.2 The Period data type

Pandas also has a completely separate data type called a **Period** to represent **columns** of data in a DataFrame that are specific **periods of time**. This is directly analogous to the PeriodIndex, but for DataFrame columns. Examples of a Period are the entire month of June 2014, or the entire 15 minute period from June 12, 2014 5:15 to June 12, 2014 5:30.

Convert a datetime column to a Period

We can use the `to_period` available with the `dt` accessor to convert datetimes to Period data types. You must pass it an offset alias to denote the length of the time period. Let’s convert the `date` column in the weather dataset to a monthly Period column .

```
[7]: weather = pd.read_csv('../data/weather.csv', parse_dates=['date'])
weather.head(3)
```

		date	rain	snow	temperature
	0	2007-01-01	Yes	No	68.0
	1	2007-01-02	No	No	55.9
	2	2007-01-03	No	No	62.1

Let's make the conversion from datetime to period and assign the result as a new column in the DataFrame.

```
[8]: date = weather['date']
weather['date_period'] = weather['date'].dt.to_period('M')
weather.head(3)
```

	date	rain	snow	temperature	date_period
0	2007-01-01	Yes	No	68.0	2007-01
1	2007-01-02	No	No	55.9	2007-01
2	2007-01-03	No	No	62.1	2007-01

Let's verify the data type by accessing the `dtypes` attribute.

```
[9]: weather.dtypes
```

```
[9]: date      datetime64[ns]
rain       object
snow       object
temperature float64
date_period period[M]
dtype: object
```

Let's view an individual value from a period data type column.

```
[10]: weather.loc[0, 'date_period']
```

```
[10]: Period('2007-01', 'M')
```

The `dt` accessor works for Period columns

Even though it is technically labeled as object, pandas still has attributes and methods specific to periods.

```
[11]: weather['date_period'].dt.month.head(3)
```

```
[11]: 0    1
1    1
2    1
Name: date_period, dtype: int64
```

```
[12]: weather['date_period'].dt.month.head(3)
```

```
[12]: 0    1
      1    1
      2    1
Name: date_period, dtype: int64
```

Return the span of time with the `freq` attribute.

```
[13]: weather['date_period'].dt.freq
```

```
[13]: <MonthEnd>
```

44.3 Anchored offsets

By default, when grouping by week, pandas chooses to end the week on Sunday. Let's verify this by grouping by week and taking the resulting index label and determining its weekday name.

```
[14]: week_mean = stocks.resample('W').agg({'AMZN': ['size', 'min', 'mean', 'max']})
week_mean.head(3)
```

AMZN				
	size	min	mean	max
date				
1999-10-31	5	70.62	76.312	82.75
1999-11-07	5	63.06	65.874	69.12
1999-11-14	5	70.81	73.750	78.00

```
[15]: week_mean.index[0].day_name()
```

```
[15]: 'Sunday'
```

Anchor by a different day

You can anchor the week to any day you choose by appending a dash and then the first three letters of the day of the week. Let's anchor the week to Wednesday.

```
[16]: stocks.resample('W-WED').agg({'AMZN': ['size', 'min', 'mean', 'max']}).head(3)
```

AMZN				
	size	min	mean	max
date				
1999-10-27	3	75.94	79.980	82.75
1999-11-03	5	65.81	68.598	71.00
1999-11-10	5	63.06	69.762	78.00

Longer intervals of time with numbers appended to offset aliases

We can actually add more details to our offset aliases by using a number to specify an amount of that particular offset alias. For instance, `5M` will group in 5 month intervals.

```
[17]: stocks.resample('5M').agg({'AMZN': ['size', 'min', 'mean', 'max']}).head(3)
```

AMZN					
		size	min	mean	max
	date				
	1999-10-31	5	70.62	76.312000	82.75
	2000-03-31	106	61.69	74.989717	106.69
	2000-08-31	106	30.00	45.460660	67.56

Group by every 22 weeks anchored to Thursday.

```
[18]: stocks.resample('22W-THU').agg({'AMZN': ['size', 'min', 'mean', 'max']}).head(3)
```

AMZN					
		size	min	mean	max
	date				
	1999-10-28	4	71.00	77.735000	82.75
	2000-03-30	106	61.69	75.023868	106.69
	2000-08-31	107	30.00	45.661963	67.56

44.4 Calling `resample` on a datetime column

The `resample` method can still work without a DatetimeIndex. If there is a column that is of the datetime data type, you can use the `on` parameter to specify that column. Let's reset the index and then call `resample` on that DataFrame.

```
[19]: amzn_reset = stocks.reset_index()
amzn_reset.head(3)
```

	date	MSFT	AAPL	SLB	AMZN	...	XOM	WMT	T	FB	V
0	1999-10-25	29.84	2.32	17.02	82.75	...	21.45	38.99	16.78	NaN	NaN
1	1999-10-26	29.82	2.34	16.65	81.25	...	20.89	37.11	17.28	NaN	NaN
2	1999-10-27	29.33	2.38	16.52	75.94	...	20.80	36.94	18.27	NaN	NaN

The only difference is that we specify the grouping column with the `on` parameter. The result is the exact same.

```
[20]: amzn_reset.resample('W-WED', on='date').agg({'AMZN': ['size', 'min', 'mean', 'max']}).head(3)
```

AMZN					
	date	size	min	mean	max
	1999-10-27	3	75.94	79.980	82.75
	1999-11-03	5	65.81	68.598	71.00
	1999-11-10	5	63.06	69.762	78.00

44.5 Calling resample on a Series

Above, we called `resample` on a DataFrame. We can also use it for Series. Let's select Amazon's closing price as a Series.

```
[21]: amzn_close = stocks['AMZN']
amzn_close.head(3)
```

```
[21]: date
1999-10-25    82.75
1999-10-26    81.25
1999-10-27    75.94
Name: AMZN, dtype: float64
```

For a Series, the aggregating column is just the values. It's not necessary to use the `agg` method in order to aggregate. Instead, we can call aggregation methods directly. Here, we find the mean closing price by month.

```
[22]: amzn_close.resample('M').mean().head()
```

```
[22]: date
1999-10-31    76.312000
1999-11-30    76.240952
1999-12-31    91.070000
2000-01-31    68.049500
2000-02-29    72.463000
Freq: M, Name: AMZN, dtype: float64
```

To compute multiple aggregations, use the `agg` method and pass it a list of the aggregating functions as strings. Here we find the total number of trading days ('size'), the min, max, and mean of the closing price for every three year period.

```
[23]: amzn_close.resample('3Y', kind='period').agg(['size', 'min', 'max', 'mean'])
```

	size	min	max	mean
date				
1999	548	5.97	106.69	34.194252
2002	756	9.13	59.91	32.596918
2005	754	26.07	100.82	47.668316
2008	757	35.03	184.76	98.729260
2011	754	160.97	404.39	238.383581
2014	756	286.95	844.36	503.404153
2017	708	753.67	2039.51	1446.483912

44.6 Exercises

Execute the following cell that reads in 20 years of Microsoft stock data and use it for the first few exercises.

```
[24]: msft = pd.read_csv('../data/stocks/msft20.csv', parse_dates=['date'], index_col='date')
msft.head(3)
```

	open	high	low	close	adjusted_close	volume	dividend_amount
date							
1999-10-19	88.250	89.250	85.25	86.313	27.8594	69945600	0.0
1999-10-20	91.563	92.375	90.25	92.250	29.7758	88090600	0.0
1999-10-21	90.563	93.125	90.50	93.063	30.0381	60801200	0.0

Exercise 1

In which week did MSFT have the greatest number of its shares (volume) traded?

```
[ ]:
```

Exercise 2

With help from the `diff` method, find the quarter containing the most number of up days.

```
[ ]:
```

Exercise 3

Find the mean price per year along with the minimum and maximum volume.

```
[ ]:
```

Exercise 4

Use the `to_datetime` function to convert the hire date column into datetimes. Reassign this column in the `emp` DataFrame.

[]:

Exercise 5

Without putting `hire_date` into the index, find the mean salary based on `hire_date` over 5 year periods. Also return the number of salaries used in the mean calculation for each period.

[]:

Chapter 45

Rolling Windows

Often during time series analysis, we would like to calculate a statistic over a rolling window of time. For example, we might want to know the average of the last 3 observations. Here, we have a rolling window size of 3, which includes the current observation plus the preceding two.

Original	Mean	Window Size																						
4	4	1	9	9	4	12	12	1	4	4	1	4	4	1	4	4	1	4	4	1	4	4	1	
1			1			2			2	2.5		2	1	2.5	2	1	2.5	2	1	2.5	2	1	2.5	2
12			12			5.7			3	12	5.7	3	12	5.7	3	12	5.7	3	12	5.7	3	12	5.7	3
8			8				8		3	8	7	3	8	7	3	8	7	3	8	7	3	8	7	3
10			10				10		3	18	10	3	18	10	3	18	10	3	18	10	3	18	10	3
18			18				18		3	10	10	3	10	10	3	10	10	3	10	10	3	10	10	3
2			2				2		2	2		2	2		2	2		2	2		2	2		2
12			12				12		12			12			12			12			12			12

Let's create the same data as a Pandas Series.

```
[1]: import pandas as pd
import numpy as np
s = pd.Series([4, 1, 12, 8, 10, 18, 2, 12])
s
```

```
[1]: 0      4
1      1
2     12
3      8
4     10
5     18
6      2
7     12
dtype: int64
```

We now use the `rolling` method to return the same result as seen in the above visualization. The rest of this notebook explains this method.

```
[2]: s.rolling(3, min_periods=1).agg(['mean', np.size]).round(1)
```

	mean	size
0	4.0	1.0
1	2.5	2.0
2	5.7	3.0
3	7.0	3.0
4	10.0	3.0
5	12.0	3.0
6	10.0	3.0
7	10.7	3.0

Get Stock Market Data Again

With our stock data, we might want to know for each day, the average closing price for the last 5 trading days. The `rolling` method helps accomplish this task. First, let's read in 20 years of stock market data.

```
[3]: stocks = pd.read_csv('../data/stocks/stocks10.csv', parse_dates=['date'],
                       index_col='date')
stocks.head(3)
```

	MSFT	AAPL	SLB	AMZN	TSLA	XOM	WMT	T	FB	V
date										
1999-10-25	29.84	2.32	17.02	82.75	NaN	21.45	38.99	16.78	NaN	NaN
1999-10-26	29.82	2.34	16.65	81.25	NaN	20.89	37.11	17.28	NaN	NaN
1999-10-27	29.33	2.38	16.52	75.94	NaN	20.80	36.94	18.27	NaN	NaN

The `rolling` method works very similarly to `resample`. We pass it the offset alias of the length of our window and then aggregate as usual. It works best when you have a DatetimeIndex, otherwise you will need to specify the datetime column with the `on` parameter. It also works better for Series and not DataFrames. Let's select Tesla closing price for the dates where it existed as a stock by dropping missing values.

```
[4]: tsla = stocks['TSLA'].dropna()
tsla.head()
```

```
[4]: date
2010-06-29    23.89
2010-06-30    23.83
2010-07-01    21.96
2010-07-02    19.20
2010-07-06    16.11
Name: TSLA, dtype: float64
```

Here, we get the average closing price for the last five days.

```
[5]: tsla.rolling('5D').mean().head()
```

[5]: date

```
2010-06-29    23.890000
2010-06-30    23.860000
2010-07-01    23.226667
2010-07-02    22.220000
2010-07-06    17.655000
Name: TSLA, dtype: float64
```

Explanation

At each data point, the average of the **last** 5 days worth of data, which **includes the current day** are found. For instance, let's say the current day is Nov 10, 2017. Pandas will get all data back until Nov 6, 2017. It will aggregate all values found within this range. In this dataset where we have only one value per day, the maximum number of values to be aggregated in any window is 5.

This does not mean the window size is always going to contain 5 values. Most will contain less as there are no trading days on the weekend.

We can include an additional aggregation function, `np.size`, to find the number of values in each window. We should be able to use the string '`size`' but there appears to be a bug in pandas and it's giving us an error.

[6]: `tsla.rolling('5D').agg(['mean', np.size]).head()`

	mean	size
date		
2010-06-29	23.890000	1.0
2010-06-30	23.860000	2.0
2010-07-01	23.226667	3.0
2010-07-02	22.220000	4.0
2010-07-06	17.655000	2.0

45.1 Keep window size the same with an integer

Instead of using an offset alias, you can specify a specific window size with an integer. The following will always use the last 5 values (trading days in this case), regardless of how many actual days pass, to determine an average. When using an integer for the window, the `rolling` method enforces that there must be that number of values present or else a missing value will be the result. This is what you are seeing below.

[7]: `tsla.rolling(5).mean().head(10)`

[7]: date

```
2010-06-29      NaN
2010-06-30      NaN
2010-07-01      NaN
2010-07-02      NaN
2010-07-06    20.998
2010-07-07    19.380
```

```
2010-07-08    18.106
2010-07-09    17.194
2010-07-12    16.764
2010-07-13    17.170
Name: TSLA, dtype: float64
```

Set the minimum window size

If you would like a non-missing value produced regardless of the window size, use the `min_periods` parameter to control it.

```
[8]: tsla.rolling(5, min_periods=3).agg(['mean', np.size]).head(8)
```

	mean	size
date		
2010-06-29	NaN	NaN
2010-06-30	NaN	NaN
2010-07-01	23.226667	3.0
2010-07-02	22.220000	4.0
2010-07-06	20.998000	5.0
2010-07-07	19.380000	5.0
2010-07-08	18.106000	5.0
2010-07-09	17.194000	5.0

You can center the window around the current row with the `center` parameter. It will use an equal number of values before and after the current row.

```
[9]: tsla.rolling(5, min_periods=3, center=True).agg(['mean', np.size]).head(8)
```

	mean	size
date		
2010-06-29	23.226667	3.0
2010-06-30	22.220000	4.0
2010-07-01	20.998000	5.0
2010-07-02	19.380000	5.0
2010-07-06	18.106000	5.0
2010-07-07	17.194000	5.0
2010-07-08	16.764000	5.0
2010-07-09	17.170000	5.0

45.2 Plotting

Let's find the trailing 50-day min, mean, and max of the closing price. Here, we will require at least 50 trading days worth of data.

```
[10]: rolling_stats = tsla.rolling(50).agg(['min', 'mean', 'max'])
rolling_stats.head(3)
```

		min	mean	max
	date			
	2010-06-29	NaN	NaN	NaN
	2010-06-30	NaN	NaN	NaN
	2010-07-01	NaN	NaN	NaN

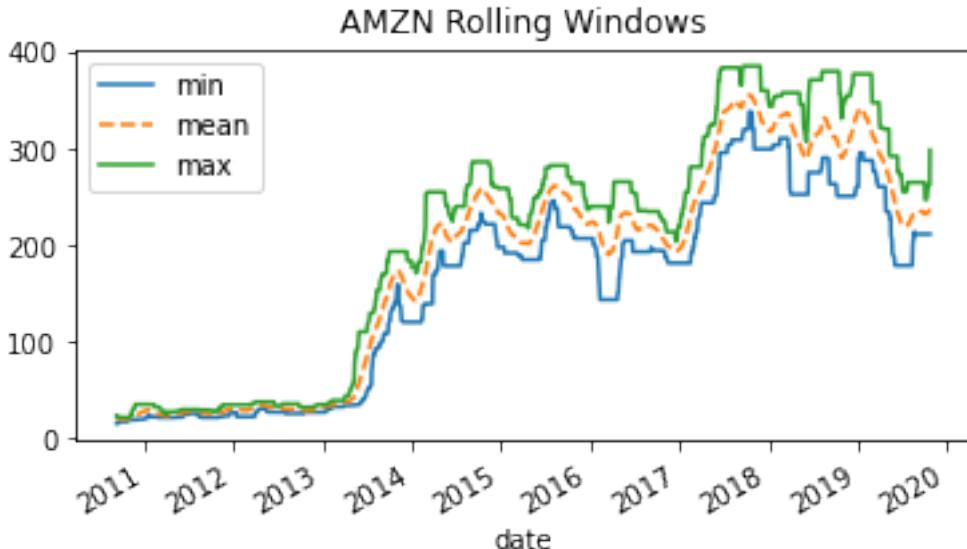
Remove all rows that did not have 50 preceding days worth of data.

```
[11]: rolling_stats = rolling_stats.dropna()
rolling_stats.head()
```

		min	mean	max
	date			
	2010-09-08	15.8	19.8336	23.89
	2010-09-09	15.8	19.7700	23.83
	2010-09-10	15.8	19.6968	21.96
	2010-09-13	15.8	19.6720	21.95
	2010-09-14	15.8	19.7104	21.95

Plot each column with a different color and line style.

```
[12]: %matplotlib inline
rolling_stats.plot(figsize=(6, 3), style=['-', '--', '-'],
                    title='AMZN Rolling Windows');
```



45.3 Exercises

Exercise 1

Use the employee dataset for this exercise. Attempt to take a rolling average on salary using a 30 day time span on hire date. Does the error message make sense?

```
[ ]:
```

Exercise 2

Set hire date as the index and then select the salary column as a Series. Sort the Series by date and drop the missing values. Now select a subset that only has hire dates from 1990 onwards. Then find a 1,000 day rolling average. Finally make a call to the `plot` method. Make sure you inline matplotlib if you did not do it earlier.

```
[ ]:
```

Exercise 3

Read in the energy consumption dataset. Select just the residential source and plot a 12 month trailing rolling mean of the energy.

```
[ ]:
```

Chapter 46

Grouping by Time and another Column

In the previous chapters, we learned how to group by an amount of time with `resample`. Let's do that again here finding the average salary of every employee based on a span of 5 years.

```
[1]: import pandas as pd  
emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'],  
                  index_col='hire_date')  
emp.head(3)
```

	dept	title	salary	sex	race
hire_date					
2001-12-03	Police	POLICE SERGEANT	87545.38	Male	White
2010-11-15	Other	ASSISTANT CITY ATTORNEY II	82182.00	Male	Hispanic
2006-01-09	Houston Public Works	SENIOR SLUDGE PROCESSOR	49275.00	Male	Black

```
[2]: emp.resample('5Y').agg({'salary': 'mean'}).round(-3)
```

	salary
	hire_date
1968-12-31	NaN
1973-12-31	65000.0
1978-12-31	78000.0
1983-12-31	74000.0
1988-12-31	71000.0
1993-12-31	69000.0
1998-12-31	68000.0
2003-12-31	63000.0
2008-12-31	60000.0
2013-12-31	57000.0
2018-12-31	48000.0

Replicating `resample` with `groupby + Grouper`

The following syntax is a bit strange, so it might take reading it a few times to understand what is going on. You can group by time within the `groupby` method but you must use the `pd.Grouper` type to specify the frequency (the offset alias).

Specify the frequency

The main parameter for `Grouper` is `freq`. Set it to the offset alias. I like using the variable name `tg` which stands for ‘time grouper’.

```
[3]: tg = pd.Grouper(freq='5Y')
```

Think of `pd.Grouper` like a dictionary that holds information

The only use case for this object is to pass it into `groupby`. It might be easier to just think of it as a dictionary that holds the frequency. Once we pass it to `groupby`, we can aggregate like we normally do and get the same result as we did with `resample`.

```
[4]: emp.groupby(tg).agg({'salary':'mean'}).round(-3)
```

hire_date	salary
1968-12-31	NaN
1973-12-31	65000.0
1978-12-31	78000.0
1983-12-31	74000.0
1988-12-31	71000.0
1993-12-31	69000.0
1998-12-31	68000.0
2003-12-31	63000.0
2008-12-31	60000.0
2013-12-31	57000.0
2018-12-31	48000.0

46.1 Grouping by an amount of time and another column

There are two different ways to group by time and another column. The difference is subtle but important and can make a difference in the result. The datetime column and the other column can either be grouped **together** or grouped **independently**. Let's say we wanted to find the average salary over 5-year time periods for each sex.

Group together

To group sex and a 5-year time span together, we must use `groupby`. We will simply pass a list of both the `Grouper` object and the column name to `groupby`.

```
[5]: tg = pd.Grouper(freq='5Y')
groups = ['sex', tg]
emp.groupby(groups).agg({'salary':'mean'}).round(-3)
```

		salary
	sex	hire_date
Female	1973-12-31	59000.0
	1978-12-31	76000.0
	1983-12-31	63000.0
	1988-12-31	63000.0
	1993-12-31	62000.0
	1998-12-31	60000.0
	2003-12-31	58000.0
	2008-12-31	57000.0
	2013-12-31	56000.0
	2018-12-31	48000.0
Male	1968-12-31	NaN
	1973-12-31	70000.0
	1978-12-31	79000.0
	1983-12-31	76000.0
	1988-12-31	74000.0
	1993-12-31	73000.0
	1998-12-31	71000.0
	2003-12-31	65000.0
	2008-12-31	62000.0
	2013-12-31	57000.0
	2018-12-31	48000.0

Datetimes are the same

Notice, how the datetimes for both female and male groups are the same. This is not going to be the case below.

```
[6]: emp.query('sex == "Male"').index.min()
```

```
[6]: Timestamp('1968-12-13 00:00:00')
```

46.2 Group independently

To group independently, we first group the non-datetime column with the `groupby` method. The `Groupby` object has a `resample` method which allows you to then group by an amount of time **within** the groups you just created. You use it just like it was being called from a DataFrame. Notice how the hire dates for males and females are different.

```
[7]: emp.groupby('sex').resample('5Y').agg({'salary':'mean'}).round(-3)
```

salary		
sex	hire_date	
Female	1969-12-31	75000.0
	1974-12-31	43000.0
	1979-12-31	67000.0
	1984-12-31	61000.0
	1989-12-31	62000.0
	1994-12-31	63000.0
	1999-12-31	58000.0
	2004-12-31	58000.0
	2009-12-31	57000.0
	2014-12-31	55000.0
	2019-12-31	48000.0
Male	1968-12-31	NaN
	1973-12-31	70000.0
	1978-12-31	79000.0
	1983-12-31	76000.0
	1988-12-31	74000.0
	1993-12-31	73000.0
	1998-12-31	71000.0
	2003-12-31	65000.0
	2008-12-31	62000.0
	2013-12-31	57000.0
	2018-12-31	48000.0

Different results

It's important to see that you will get different results depending on whether you group together or group independently. The reason the results are different is because the earliest male and female employees don't have a hire date that is an exact 5 year multiple difference. The earliest hire date for female employees was 1969 while it is 1968 for males. If the first male and female employees were both hired in 1968 (or 1969), then the returned datetime index would have been the same.

46.3 Using a pivot table with Grouper for easier comparisons

You can pass a Grouper object to a pivot table to get a nice final product. This groups sex together with time.

```
[8]: emp.pivot_table(index=tg, columns='sex', values='salary').round(-3)
```

	sex	Female	Male
hire_date			
1973-12-31	59000.0	70000.0	
1978-12-31	76000.0	79000.0	
1983-12-31	63000.0	76000.0	
1988-12-31	63000.0	74000.0	
1993-12-31	62000.0	73000.0	
1998-12-31	60000.0	71000.0	
2003-12-31	58000.0	65000.0	
2008-12-31	57000.0	62000.0	
2013-12-31	56000.0	57000.0	
2018-12-31	48000.0	48000.0	

Using Grouper on a datetime column

If your datetime column is not in the index, you can still use Grouper. Just specify the column name with the `key` parameter. See the example below with `hire_date` not in the index.

```
[9]: emp2 = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
emp2.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

```
[10]: tg2 = pd.Grouper(freq='10Y', key='hire_date')
emp2.groupby(['sex', tg2]).agg({'salary':'mean'})
```

		salary
sex	hire_date	
Female	1978-12-31	72408.300000
	1988-12-31	62609.175556
	1998-12-31	60755.045322
	2008-12-31	57086.636143
	2018-12-31	50754.220978
Male	1968-12-31	NaN
	1978-12-31	78184.825833
	1988-12-31	74938.171306
	1998-12-31	71673.462248
	2008-12-31	63441.740493
	2018-12-31	50878.627740

46.4 Exercises

Exercise 1

Read in the energy consumption dataset. Find the average energy consumption per sector per 10 year time span beginning from the first year of data. Return the results as both a groupby and a pivot table. Experiment with adding 'S' to the end of your offset alias. How does this change the results?

```
[ ]:
```


Part VIII

Regular Expressions

Chapter 47

Introduction to Regular Expressions

Regular Expressions give us a way to do much more powerful string manipulation. A regular expression, or simply **regex**, is a special string that describes a specific pattern that you would like to match in another string.

Examples of questions that regexes can answer

It might be helpful to see a list of questions that a regex pattern can match:

- Match all words that begin with ‘S’ and end in ‘y’
- Match the word ‘friend’ or ‘freind’
- Match a word with at least 3 digits in it
- Match all Gmail email addresses
- Capture the word immediately following the word ‘Author’
- Capture the word immediately following the third occurrence of the word ‘coffee’

47.1 The `contains` and `extract` methods

We will be primarily concerned with finding matching patterns within string values of a Pandas Series. We will then select all values within the Series that match the pattern via boolean indexing. The `contains` string Series method will be used for this.

Eventually, we will use the `extract` string Series method to extract particular substrings from the strings within the Series.

A simple example without regular expressions

Let’s match all movie titles that contain either an ‘x’, ‘y’, or ‘z’. Without using a regex, we would use multiple `contains` string methods separating them with the logical `or` symbol:

```
[1]: import pandas as pd
movie = pd.read_csv('../data/movie.csv')
title = movie['title']
title.head(3)
```

```
[1]: 0                               Avatar
1   Pirates of the Caribbean: At World's End
2                               Spectre
Name: title, dtype: object
```

```
[2]: has_xyz = title.str.contains('x') | title.str.contains('y') | title.str.contains('z')
title[has_xyz].head()
```

```
[2]: 9    Harry Potter and the Half-Blood Prince
21          The Amazing Spider-Man
30          Skyfall
35          Monsters University
37          Transformers: Age of Extinction
Name: title, dtype: object
```

We can sum up this boolean Series to determine the number of values that have either an ‘x’, ‘y’, or ‘z’ in them.

```
[3]: has_xyz.sum()
```

```
[3]: 1193
```

Use a regex instead

Instead, we can use the regex '[xyz]', which matches the pattern for any string that contains an ‘x’, ‘y’, or ‘z’. We can verify that we get the same total. This regex plus many more will be covered in detail below.

```
[4]: title.str.contains('[xyz]').sum()
```

```
[4]: 1193
```

47.2 Mini-Programming Language

Regular expressions are a miniature programming language that have their own strict set of rules just like any other language. The syntax is written as a string mixing both **literal** and **special** characters.

Literal vs Special Characters

There are two distinct categories of characters within a regex string - **Literal** and **Special**

- **Literal** - these characters don't have any special meaning. They simply represent themselves. They are also referred to as **regular** characters.
- **Special** - these characters do have a special meaning. Each special character represents something very specific. They are also referred to as **metacharacters**.

Matching with only Literal Characters

The simplest regex patterns you can write contain only literal characters. These strings will look like any ordinary string. Let's search for movies that have the word 'Star' in them.

'Star' is a valid regular expression. We will use the `contains` Series string method which accepts a regular expression as its first argument. It returns a boolean Series.

```
[5]: pattern = 'Star'
title.str.contains(pattern).head()
```

```
[5]: 0    False
1    False
```

```
2    False
3    False
4    True
Name: title, dtype: bool
```

Filter for only movies containing ‘Star’

Let’s take this resulting Series and use it for boolean indexing. The result should be the movie titles that have ‘Star’ in them.

```
[6]: pattern = 'Star'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[6]: 4      Star Wars: Episode VII - The Force Awakens
48      Star Trek Into Darkness
57      Star Trek Beyond
159      Star Trek
233     Star Wars: Episode III - Revenge of the Sith
Name: title, dtype: object
```

Regular Expressions are case sensitive

Regexes are case sensitive by default. ‘Star’ only matches movie titles with an uppercase ‘S’ followed immediately by lowercase ‘tar’. Let’s search for lowercase ‘star’:

```
[7]: pattern = 'star'
filt = title.str.contains(pattern)
title[filt]
```

```
[7]: 2641    Firestarter
2737    Superstar
Name: title, dtype: object
```

Find all movies containing exact string ‘Star Wars’

```
[8]: pattern = 'Star Wars'
filt = title.str.contains(pattern)
title[filt]
```

```
[8]: 4      Star Wars: Episode VII - The Force Awakens
233     Star Wars: Episode III - Revenge of the Sith
234     Star Wars: Episode II - Attack of the Clones
237     Star Wars: Episode I - The Phantom Menace
1521    Star Wars: Episode VI - Return of the Jedi
2031    Star Wars: Episode V - The Empire Strikes Back
2973      Star Wars: Episode IV - A New Hope
3271      Star Wars: The Clone Wars
Name: title, dtype: object
```

Find all movies containing exact string ‘hine’:

```
[9]: pattern = 'hine'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[9]: 94      Terminator 3: Rise of the Machines
    475          The Time Machine
   1302          Sunshine
   1372        Hot Tub Time Machine
   1710       Machine Gun Preacher
Name: title, dtype: object
```

47.3 Special Characters

The following characters are the **special** or **metacharacters**

. ^ \$ * + ? { } [] \ | ()

The rest of this chapter is devoted to examples that explain each of the special characters above. This will not be an exhaustive coverage of regular expressions as they can get quite complex. There are even entire books written on the subject.

47.4 The dot metacharacter .

The **dot** or **period** is a special character that matches any character. For example the regex '**m.le**' will match any string that has an **m** followed by any character followed by **le**. It will match 'male', 'mile', 'mole', 'thimble', 'tumble', etc... Let's see how many movie titles have this pattern:

```
[10]: pattern = 'm.le'
filt = title.str.contains(pattern)
title[filt]
```

```
[10]: 661           Mona Lisa Smile
    1465          Wimbledon
   1733  Indiana Jones and the Temple of Doom
   1770          A Simple Wish
   1923          The Gambler
   2019          Ready to Rumble
   2326  The Baader Meinhof Complex
   2476          A Simple Plan
   3374        Rumble in the Bronx
   4711          Tumbleweeds
Name: title, dtype: object
```

47.5 The caret metacharacter ^

The caret, **^**, is a special character that forces the pattern to match from the beginning of the string. Let's take a look at the difference between the regexes **War** and **^War**. The first matches the word 'War' anywhere in the string. The second matches the word 'War' only at the beginning. Let's output the differences:

```
[11]: pattern = 'War'
filt = title.str.contains(pattern)
```

```
title[filt].head()
```

```
[11]: 4           Star Wars: Episode VII - The Force Awakens
      27          Captain America: Civil War
      46          World War Z
      64  The Chronicles of Narnia: The Lion, the Witch ...
      108         Warcraft
Name: title, dtype: object
```

```
[12]: pattern = '^War'
filt = title.str.contains(pattern)
title[filt]
```

```
[12]: 108          Warcraft
      187  War of the Worlds
      597          War Horse
     1483  Warriors of Virtue
     1598          Warm Bodies
     1934          War
     1961          Warrior
     2878          WarGames
     3160          War, Inc.
     3431          Warlock
     3536          War & Peace
    4012  Warlock: The Armageddon
Name: title, dtype: object
```

47.6 The dollar sign metacharacter \$

The dollar sign metacharacter, \$, works analogously to the caret but instead forces a match to the **end** of the string. Let's find all the movies that end in 'War':

```
[13]: pattern = 'War$'
filt = title.str.contains(pattern)
title[filt]
```

```
[13]: 27          Captain America: Civil War
      241  The Huntsman: Winter's War
      323          The Flowers of War
      534  Charlie Wilson's War
      611          Hart's War
      666          This Means War
     1160          Lord of War
     1261          The Art of War
     1549  Dragon Wars: D-War
     1934          War
    2867  Tae Guk Gi: The Brotherhood of War
    2962          5 Days of War
    3577          Men of War
    3742          Born of War
Name: title, dtype: object
```

Start and End Anchor tags

The caret and dollar metacharacters are also known as **anchor** tags since they anchor the pattern to either the beginning or end.

47.7 Combining special characters

A regex can have any number of literal and meta special characters. The following regex matches movies that begin with S, followed by any character followed n.

```
[14]: pattern = r'^S.n'  
filt = title.str.contains(pattern)  
title[filt].head(3)
```

```
[14]: 248           San Andreas  
       315           Son of the Mask  
      683 Sin City: A Dame to Kill For  
Name: title, dtype: object
```

47.8 Exercises

Exercise 1

Find all movies that have 2 consecutive z's in them.

```
[ ]:
```

Exercise 2

Find all movies that begin with 9.

```
[ ]:
```

Exercise 3

Find all movies that have a b as their third character.

```
[ ]:
```

Exercise 4

Find all movies with a fourth-to-last character of M and a last character of e.

```
[ ]:
```

Exercise 5

Could you use a regular expression to find a movie that was exactly 6 characters in length?

```
[ ]:
```

Exercise 6

What is a more natural way to complete exercise 5 without a regex?

[]:

Chapter 48

Quantifiers

This notebook continues coverage of the special characters by focusing on those that are used to match repeated characters. These group of metacharacters are called **quantifiers**, because they quantify the type of repetition you desire.

48.1 The asterisk metacharacter *

The **asterisk** or **star** metacharacter matches the previous character 0 or more times. For instance, the regex, 'Ah* No' will look for strings that have an uppercase 'A' followed by 0 or more lowercase 'h' followed by 'No'. Let's see how this works on a Series of sample data:

```
[1]: import pandas as pd
s = pd.Series(['Ouch', 'Ah No', 'Ahh', 'Nooo', 'Ahhhhh No', 'A No', 'A'])
s
```

```
[1]: 0      Ouch
     1      Ah No
     2      Ahh
     3      Nooo
     4      Ahhhhh No
     5      A No
     6      A
dtype: object
```

```
[2]: pattern = 'Ah* No'
filt = s.str.contains(pattern)
s[filt]
```

```
[2]: 1      Ah No
     4      Ahhhhh No
     5      A No
dtype: object
```

Without the 'No' at the end, it would match two more values:

```
[3]: pattern = 'Ah*'
filt = s.str.contains(pattern)
s[filt]
```

```
[3]: 1      Ah No
      2      Ahh
      4      Ahhhhhh No
      5      A No
      6      A
dtype: object
```

48.2 The plus metacharacter +

The **plus** metacharacter is very similar to the asterisk, except that it matches 1 or more of the previous character. Thus for the regex 'Ah+ No', the 'h' must appear at least once.

```
[4]: pattern = 'Ah+ No'
filt = s.str.contains(pattern)
s[filt]
```

```
[4]: 1      Ah No
      4      Ahhhhhh No
dtype: object
```

48.3 The question mark metacharacter ?

The question mark is similar to both the asterisk and the star, except that it matches the previous character 0 or 1 times exactly.

```
[5]: pattern = 'Ah? No'
filt = s.str.contains(pattern)
s[filt]
```

```
[5]: 1      Ah No
      5      A No
dtype: object
```

Using another example, the regex 'Sec?r' will match both 'Secret' and 'Serving'. Basically, the character before the question mark is **optional**.

```
[6]: movie = pd.read_csv('../data/movie.csv')
title = movie['title']
pattern = 'Sec?r'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[6]: 198    Night at the Museum: Secret of the Tomb
282    Harry Potter and the Chamber of Secrets
355        The Secret Life of Walter Mitty
513            The Secret Life of Pets
1229                Secret Window
Name: title, dtype: object
```

48.4 The curly braces metacharacter {m,n}

The curly braces metacharacter matches the previous character a given number of times. There are four different ways to use the curly braces:

- a single integer `a{3}` - matches exactly three ‘a’ characters in a row
- a single integer followed by a comma `a{3,}` - matches three or more ‘a’ characters in a row
- a comma followed by a single integer `a{,3}` - matches zero to three ‘a’ characters in a row
- two integers separated by a comma `a{3,5}` - matches between 3 and 5 ‘a’ characters in a row

Let’s create another Series by hand and match all the strings that begin with ‘A’, have the letter ‘h’ repeat between 2 and 5 times and then followed by ’ No’.

```
[7]: s = pd.Series(['Ouch', 'Ahhh No', 'Ahh No', 'Nooo', 'Ahhhhhh No',
                  'A No', 'A', 'Ahhh'])
```

s

```
[7]: 0      Ouch
      1      Ahhh No
      2      Ahh No
      3      Nooo
      4      Ahhhhhh No
      5      A No
      6      A
      7      Ahhh
dtype: object
```

```
[8]: pattern = 'Ah{2,5} No'
filt = s.str.contains(pattern)
s[filt]
```

```
[8]: 1      Ahhh No
      2      Ahh No
dtype: object
```

48.5 Exercises

Use the title column of the movie Series for these exercises.

```
[9]: movie = pd.read_csv('../data/movie.csv')
title = movie['title']
```

Exercise 1

Find all movies that have a ‘z’ as their 15th character.

[]:

Exercise 2

Find all movies that have the word ‘Friend’ or ‘Friends’ in them.

[]:

Exercise 3

Find all movies that have between 40 and 43 characters in them. Can you verify the results with another `str` accessor method?

[]:

Exercise 4

Find all movies that begin with ‘The’ and end in ‘Movie’

[]:

Exercise 5

Create your own Series and make a regular expression that uses the `+` metacharacter. Is this character necessary?

[]:

Chapter 49

Or Conditions

49.1 The pipe metacharacter |

The pipe metacharacter is equivalent to an **or** condition. It matches the entire word before or after the pipe. The regex 'Friend|Enemy' matches any string with 'Friend' or 'Enemy' in it.

```
[1]: import pandas as pd  
movie = pd.read_csv('../data/movie.csv')  
title = movie['title']  
title.head(3)
```

```
[1]: 0          Avatar  
1    Pirates of the Caribbean: At World's End  
2          Spectre  
Name: title, dtype: object
```

```
[2]: pattern = 'Friend|Enemy'  
filt = title.str.contains(pattern)  
title[filt]
```

```
[2]: 403          Enemy of the State  
408          Enemy at the Gates  
1055         My Best Friend's Wedding  
1214         Behind Enemy Lines  
1413         Friends with Benefits  
1775    How to Lose Friends & Alienate People  
2216         My Best Friend's Girl  
3116  Seeking a Friend for the End of the World  
3495         Friends with Money  
4184         We Are Your Friends  
4279         Dysfunctional Friends  
4670         Mutual Friends  
Name: title, dtype: object
```

You can add as many pipes as you please:

```
[3]: pattern = 'Friend|Enemy|Good|Evil'  
filt = title.str.contains(pattern)
```

```
title[filt].head(10)
```

```
[3]: 55           The Good Dinosaur
      343          A Good Day to Die Hard
      403          Enemy of the State
      408          Enemy at the Gates
      670          Resident Evil: Retribution
      672          The Long Kiss Goodnight
      815          Resident Evil: Afterlife
      923          As Good as It Gets
      976          Resident Evil: Apocalypse
     1055         My Best Friend's Wedding
Name: title, dtype: object
```

49.2 The brackets metacharacter []

The brackets metacharacter allows you to match one of several characters at a single particular position. As we saw regex '[xyz]' matches any single character 'x', 'y', or 'z'.

Another example, 'T[aeiou]d' matches any words that begin with 'T', followed by exactly one vowel and then 'd'. The brackets contain all the possible matches for a single character.

Specifically, it matches the following: 'Tad', 'Ted', 'Tid', 'Tod', and 'Tud'. The brackets are a single character **or** condition, where as the pipe character is an entire phrase **or** condition.

```
[4]: pattern = 'T[aeiou]d'
filt = title.str.contains(pattern)
title[filt]
```

```
[4]: 18       Pirates of the Caribbean: On Stranger Tides
      628          Ted 2
      841          Crimson Tide
      922          Ted
      1594         The Prince of Tides
      2016         Win a Date with Tad Hamilton!
      2171         Bill & Ted's Bogus Journey
      2576         Tidal Wave
      3053         Bill & Ted's Excellent Adventure
      4283         Living Dark: The Story of Ted the Caver
      4809         Tadpole
Name: title, dtype: object
```

Entire character classes within the brackets

Let's say you want to match all the lowercase letters 'a' through 'z'. You could write each letter within the brackets. Thankfully, there is a much easier way with **character classes**.

Character classes are special notation within the brackets that can be used to denote entire subsets of characters. Take the following:

- '[0-9]' represents all digits 0 through 9
- '[a-z]' represents all lowercase letters
- '[A-Z]' represents all uppercase letters

- '[a-zA-Z]' represents all lowercase and uppercase letters

This notation only works within the brackets.

Digits in movies

Let's match all movies with a digit in them.

```
[5]: pattern = '[0-9]'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[5]: 6           Spider-Man 3
      19          Men in Black 3
      31          Spider-Man 2
      32          Iron Man 3
      39  The Amazing Spider-Man 2
Name: title, dtype: object
```

Matching movies with 2 digits in a row

We can match movies with two digits in a row by using the digits character class twice.

```
[6]: pattern = '[0-9][0-9]'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[6]: 60           2012
      85          47 Ronin
      212         The 13th Warrior
      258        300: Rise of an Empire
      268  Around the World in 80 Days
Name: title, dtype: object
```

49.3 Combining Special Characters

You are allowed to combine any number of literal and special characters together with your regex. For instance, matching movies with two or more digits in a row could have been done by using the curly braces for repeats like this:

```
[7]: pattern = '[0-9]{2,}'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[7]: 60           2012
      85          47 Ronin
      212         The 13th Warrior
      258        300: Rise of an Empire
      268  Around the World in 80 Days
Name: title, dtype: object
```

Find all movies that begin with exactly 4 digits in a row

We can use the caret to anchor the digits to the start and the curly braces to match exactly 4 digits.

```
[8]: pattern = '^[0-9]{4}'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[8]: 60                  2012
697     3000 Miles to Graceland
1541                1941
1707                1911
2056                1408
Name: title, dtype: object
```

Find all movies that begin with ‘The’ and end with ‘Movie’

We anchor ‘The’ to the beginning with the caret and ‘Movie’ to the end with the dollar symbol. We use `.*` in the middle to represent any character repeated 0 or more times.

```
[9]: pattern = '^The .* Movie$'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[9]: 319      The Peanuts Movie
561      The Angry Birds Movie
569      The Simpsons Movie
759      The Lego Movie
1586     The SpongeBob SquarePants Movie
Name: title, dtype: object
```

Find all movies that are exactly 10 characters long

`.{10}` matches exactly any 10 characters in a row. We must anchor it to the beginning and end to ensure that the string is exactly 10 characters in length.

```
[10]: pattern = '^.{10}$'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[10]: 22      Robin Hood
28      Battleship
32      Iron Man 3
76      Waterworld
78      Inside Out
Name: title, dtype: object
```

49.4 Exercises

Exercise 1

Find all movies that begin with ‘The’ followed by the next word that begins with digits.

[]:

Exercise 2

Find all movies that have three consecutive capital letters in them.

[]:

Exercise 3

Find all movies that have begin and end with a capital letter.

[]:

Exercise 4

Find all the movies that have a digit followed by a comma followed by a digit.

[]:

Exercise 5

Find all the movies that have either an ampersand or a question mark in them.

[]:

Exercise 6

Which movie has the most ampersands, question marks, and periods in it?

[]:

Chapter 50

Character Sets and Grouping

50.1 Special Characters lose their meaning within the brackets

Special characters lose their special meaning within the brackets. For instance, [.] matches the literal dot and [()*\$] matches any string with the literal parentheses, asterisk, or dollar sign. Let's match movies with an asterisk in them:

```
[1]: import pandas as pd
movie = pd.read_csv('../data/movie.csv')
title = movie['title']

pattern = '[*]'
filt = title.str.contains(pattern)
title[filt]
```

```
[1]: 3828           What the #$*! Do We (K)now!?
      3902           M*A*S*H
      4162   Everything You Always Wanted to Know About Sex...
Name: title, dtype: object
```

Match movies with either an asterisk or dollar sign.

```
[2]: pattern = '[*$]'
filt = title.str.contains(pattern)
title[filt]
```

```
[2]: 3466           Ca$h
      3828           What the #$*! Do We (K)now!?
      3902           M*A*S*H
      4162   Everything You Always Wanted to Know About Sex...
Name: title, dtype: object
```

Excluding character classes with the caret

It is possible to exclude character classes by putting a caret as the first character inside the brackets. For instance, Z[^aeiou] matches strings that begin with 'Z' followed by a non-vowel.

```
[3]: pattern = 'Z[^aeiou]'
filt = title.str.contains(pattern)
title[filt]
```

```
[3]: 2975          Doctor Zhivago
      3469          Z Storm
      4511    ZMD: Zombies of Mass Destruction
      4679        Dogtown and Z-Boys
Name: title, dtype: object
```

The following finds all movies that have an uppercase ‘T’ followed by a non-lowercase letter.

```
[4]: pattern = 'T[^a-z]'
filt = title.str.contains(pattern)
title[filt]
```

```
[4]: 40          TRON: Legacy
      585         S.W.A.T.
      1411        TMNT
      1540    The Young and Prodigious T.S. Spivet
      2473         ATL
      2573        Scream: The TV Series
      2994         F.I.S.T.
      3028        E.T. the Extra-Terrestrial
Name: title, dtype: object
```

50.2 The backslash \ metacharacter

The backslash metacharacter is used in conjunction with the very **next** character to change its meaning. Many of the following refer to character classes as seen above.

- \d - all digits, equivalent to [0-9]
- \D - any non-digit.
- \s - any amount of whitespace including normal spaces and tabs
- \S - any non-whitespace
- \w - any ‘word’ character, which is any upper or lowercase letter, digit or underscore. Equivalent to [A-Za-z0-9_]
- \W - any non-word character
- \b - **word boundary**
- \B - non-word boundary

For instance, ^\W matches all strings that begin with a non-word character.

Prefix the string with r to make it a raw string

The backslash is a special character in normal Python strings. \n represents a newline character, \t represents a tab. To be sure your regex is exactly what you see, its best to use **raw** Python strings. Prepend the string with an r **outside** of the quotation marks to make it a raw string. Python will treat the backslash as a literal backslash without any special meaning.

```
[5]: pattern = r'^\W+'
filt = title.str.contains(pattern)
title[filt]
```

```
[5]: 3597      [Rec] 2
      4152      [Rec]
      4349      #Horror
      Name: title, dtype: object
```

Backslash escapes special characters

As we just saw, the special characters lose their special ability within the brackets. Preceding a special character by a backslash has the same effect. For instance `*` represents a literal asterisk and is the same as `[*]`

```
[6]: pattern = r'\*'
filt = title.str.contains(pattern)
title[filt]
```

```
[6]: 3828          What the #$$! Do We (K)now!?
      3902          M*A*S*H
      4162          Everything You Always Wanted to Know About Sex...
      Name: title, dtype: object
```

50.3 The parentheses metacharacters ()

The parentheses metacharacters are used to **group** together parts of the regular expression. For instance, let's say we want to find all movies that begin with the word 'In' or 'My'. You might think about using '`^In|My`':

```
[7]: pattern = '^In|My'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[7]: 54    Indiana Jones and the Kingdom of the Crystal S...
      78          Inside Out
      92    Independence Day: Resurgence
      96          Interstellar
      97          Inception
      Name: title, dtype: object
```

The meaning of `^In|My`

There are a couple things wrong with this regex. As seen above, we returned movies that begin with 'In', such as 'Indiana' or 'Inside' instead of the just the word 'In'.

Second, the movie, 'Journey 2: The Mysterious Island' has 'My' within the name and not at the beginning. This mistake is happening because of **operator precedence** within the regex.

`^In|My` matches movies that begin with the letters 'In' **OR** have 'My' anywhere inside it. The caret is only anchoring 'In'.

50.4 Using parentheses to change operator precedence

We can use parentheses to change the operator precedence just how we do in mathematical expressions. Let's modify our expression to '`^(In|My)`'. Ignore the warning for now. We will take care of it below.

```
[8]: pattern = '^^(In|My)'
filt = title.str.contains(pattern)
title[filt].head(15)
```

```
/Users/Ted/miniconda3/envs/minimal_ds/lib/python3.7/site-
packages/pandas/core/strings.py:1954: UserWarning: This pattern has match groups. To
actually get the groups, use str.extract.
    return func(self, *args, **kwargs)
```

```
[8]: 54      Indiana Jones and the Kingdom of the Crystal S...
          Inside Out
  78
  92      Independence Day: Resurgence
  96          Interstellar
  97          Inception
 253          Insurgent
316      In the Heart of the Sea
516          Independence Day
527          Inspector Gadget
582          Inglourious Basterds
630          Mystery Men
764          Invictus
767          My Favorite Martian
771          Intolerable Cruelty
 796          Inkheart
Name: title, dtype: object
```

Getting closer

We grouped In|My together so the movie must begin with them. We'd like to make sure that In and My are not part of a larger word. To do that we can use the word boundary, \b, which ensures that the word is ended. * '^(In|My)\b'

```
[9]: pattern = r'^^(In|My)\b'
filt = title.str.contains(pattern)
title[filt].head(15)
```

```
[9]: 316          In the Heart of the Sea
  767          My Favorite Martian
 799      In the Name of the King: A Dungeon Siege Tale
1055          My Best Friend's Wedding
1405          In & Out
1428          In Time
1660          My Super Ex-Girlfriend
1685          In Dreams
1781          My Sister's Keeper
1832          In Good Company
1945          My Soul to Take
2049      In the Valley of Elah
2104          My Fellow Americans
2216          My Best Friend's Girl
2375          My Big Fat Greek Wedding 2
Name: title, dtype: object
```

Why are we getting UserWarning: This pattern has match groups?

Besides operator precedence, parentheses have an alternative function and that is to extract specific text from a string. In regex terminology, we call this a **capturing group**. This warning is alerting us that we have used the syntax for a capture group but are not using a method to do extraction. It tells us to use the `extract` method if in fact we were interested in extracting this group.

Specifying a non-capturing group

Our regular expression is valid in its current state. We can signal that this is a **non-capturing group** by placing ?: as the first two characters inside of the parentheses. This eliminates the warning.

```
[10]: pattern = r'^(?:In|My)\b'
filt = title.str.contains(pattern)
title[filt].head()
```

```
[10]: 316           In the Heart of the Sea
      767           My Favorite Martian
     799   In the Name of the King: A Dungeon Siege Tale
    1055           My Best Friend's Wedding
    1405           In & Out
Name: title, dtype: object
```

50.5 Using capture groups with the `extract` string method

We can use the exact same pattern with the `extract` string method to extract the group.

```
[11]: pattern = r'^(?:In|My)\b'
title.str.extract(pattern).head()
```

0
0 NaN
1 NaN
2 NaN
3 NaN
4 NaN

Why are all the values missing?

Only a small fraction of the movie titles begin with ‘In’ or ‘My’. Let’s drop the missing values and see the extracted text:

```
[12]: pattern = r'^(?:In|My)\b'
title.str.extract(pattern).dropna().head()
```

0	
316	In
767	My
799	In
1055	My
1405	In

Extracting the fourth word of movie titles that begin with ‘In’ or ‘My’

Let’s try something a bit more complex and extract the fourth word of all movies that begin with the words ‘In’ or ‘My’. For instance, the movie, ‘In the Heart of the Sea’ meets our criteria. The word ‘of’ would be extracted from it. We will make the assumption that words are separated by spaces.

To accomplish this, we need to match movies that begin with ‘In’ or ‘My’ and then match two words, before capturing the fourth word. We already saw that `^(?:In|My)` completes the first part of this task.

We now need to match a space followed by a word. We can use `\s` to match a space and follow this with `\S+` to match any number of non-space characters. Combining these, `\s\S+` is what we can use to match our definition of a “word”.

We want to match this pattern exactly twice. We can do so with `{2}`, but we must ensure that it applies to the all of `\s\S+`, so we must wrap it in parentheses to control for operator precedence like this - `(\s\S+){2}`. We must signal that this is not a capturing group and arrive at `(?:\s\S+){2}` to match two consecutive words.

We still need to match a space after the third word and then capture the fourth word. Finally, we have a workable regex with the following:

```
[13]: pattern = r'^^(?:In|My)(?:\s\S+){2}\s(\S+)'
title.str.extract(pattern).dropna().head()
```

0	
316	of
799	of
1055	Wedding
1945	Take
2049	of

extract must have capture groups

The regex used with the `extract` string method must have capture groups. If not, an error will be raised.

Multiple capture groups for extract

You can capture more than one group with `extract`. Take a look at the following regex which captures the first word after a movie that begins with ‘The’ and the first word after ‘of’.

```
[14]: pattern = r'^The (\S+) .*of (\S+)'
title.str.extract(pattern).dropna().head()
```

	0	1
16	Chronicles	Narnia:
20	Hobbit:	the
23	Hobbit:	Smaug
63	Legend	Tarzan
64	Chronicles	Narnia:

50.6 Many other string methods take regexes

You can use regular expressions in several other Series string methods such as `count`, `replace` and `split`. For instance, the following counts the times consecutive lowercase vowels appear for each string. We then find the maximum number of times this happens within the movie titles.

```
[15]: pattern = r'[aeiou]{2}'
title.str.count(pattern).max()
```

[15]: 4

50.7 Other Dialects of Regex

Regular expressions are not quite standardized for every single programming language, so you will need to ensure you are implementing the right ‘dialect’ for each language.

50.8 More to Regex

There is more to regular expressions not covered in these notebooks.

- [Official Python Documentation](#)
- [Thorough Online Tutorial](#)
- [Practice with explanations](#) - make sure to choose Python

50.9 Regex Summary

- Literal characters represent themselves
- Special or metacharacters represent something entirely different
- Primarily usage of regex is to either match a particular string or extract a substring
- Many Pandas string methods accept regular expressions
- You will often use `contains` and `extract`
- Use raw Python strings when writing regex. Raw strings have ‘r’ prepended to them.

Metacharacter Summary

- All metacharacters - . ^ \$ * + ? { } [] \ | ()

The dot .

- . - Matches any character except line breaks

Anchors - ^, \$

- ^ - Anchors next characters to beginning
 - ^My matches strings that begin with 'My'
- \$ - Anchors previous characters to end
 - Movie\$ matches strings that end with 'Movie'

Quantifiers - *, +, ?, {}

- * - Matches 0 or more occurrences of previous character
- + - Matches 1 or more occurrences of previous character
- ? - Matches 0 or 1 occurrences of previous character
- {m} - Matches exactly m of the previous character,
- {m,} - Matches m or more of the previous character
- {,n} - Matches up to n of the previous character
- {m,n} - Matches between m and n repeats of the previous character

Character Sets

- [] - A character set to match one out of many characters. [aeiou] matches a single vowel
- [a-z], [A-Z], [0-9] - Character sets for lowercase, uppercase, and digits
- [^abc] - Use caret at beginning of bracket to match anything but these characters
- \ - backslash changes meaning of next character
- \s - whitespace - single space, tab, new-line
- \S - non-whitespace
- \w - word character - lower/uppercase, digits, and underscore
- \W - non-word-character
- \d - digits
- \D - non-digits
- \b - word boundary - matches empty string between words, that is between \w and \W
- \B - non-word boundary
- \. - Escapes all special characters such as literal dot here. * matches the literal asterisk

50.10 Exercises

Exercise 1

For all movies that begin with 'The' and are followed by the next word that begins with a digit, extract just the digits part of this word.

[]:

Exercise 2

Find all movies that have two separate numbers in them. An example would be, '7 days and 7 nights'.

[]:

Exercise 3

Find all the movies that have 6 or more non-vowel and non-space characters in a row.

[]:

Exercise 4

Extract the very next character after ‘t’ or ‘T’ for each movie.

[]:

Exercise 5

What is the most common character after ‘t’ or ‘T’?

[]:

Exercise 6

Extract all the words that begin with ‘T’ or ‘t’ and end in ‘e’ then find their frequency. Research the word boundary special character.

[]:

Chapter 51

Project - Explore Newsgroups with Regexes

The machine learning library Scikit-Learn has several thousand posts from [an old internet newsgroup](#). 400 of these posts are in the stored in the `newsgroups.csv` file in the data directory. This is a great dataset to practice your regular expressions.

Read in data

There are just two columns, one for the category and the other for the text of the post.

```
[1]: import pandas as pd  
news = pd.read_csv('../data/newsgroups.csv')  
news.head()
```

	category	text
0	sci.med	From: nyeda@cnsvax.uwec.edu (David Nye)\nSubject:...
1	talk.politics.guns	From: ndallen@r-node.hub.org (Nigel Allen)\nSubject:...
2	misc.forsale	From: mark@ardsley.business.uwo.ca (Mark Bramw...
3	misc.forsale	From: zmed16@trc.amoco.com (Michael)\nSubject:...
4	talk.politics.guns	From: fcrary@ucsu.Colorado.EDU (Frank Crary)\nSubject:...

The original dataset has 20 newsgroups, labeled categories here.

```
[2]: news['category'].value_counts()
```

```
[2]: misc.forsale      73  
rec.autos           72  
sci.space           65  
talk.politics.guns  64  
rec.sport.baseball  63  
sci.med             63  
Name: category, dtype: int64
```

```
[3]: news['text'].head()
```

```
[3]: 0    From: nyeda@cnsvax.uwec.edu (David Nye)\nSubje...
1    From: ndallen@r-node.hub.org (Nigel Allen)\nSu...
2    From: mark@ardsley.business.uwo.ca (Mark Bramw...
3    From: zmed16@trc.amoco.com (Michael)\nSubject:...
4    From: frary@ucsu.Colorado.EDU (Frank Crary)\n...
Name: text, dtype: object
```

Output an entire post with the `print` function.

```
[4]: print(news['text'].values[0])
```

```
From: nyeda@cnsvax.uwec.edu (David Nye)
Subject: Re: Post Polio Syndrome Information Needed Please !!!
Organization: University of Wisconsin Eau Claire
Lines: 21
```

[reply to keith@actrix.gen.nz (Keith Stewart)]

```
>My wife has become interested through an acquaintance in Post-Polio
>Syndrome This apparently is not recognised in New Zealand and different
>symptoms ( eg chest complaints) are treated separately. Does anyone have
>any information on it
```

It would help if you (and anyone else asking for medical information on some subject) could ask specific questions, as no one is likely to type in a textbook chapter covering all aspects of the subject. If you are looking for a comprehensive review, ask your local hospital librarian. Most are happy to help with a request of this sort.

Briefly, this is a condition in which patients who have significant residual weakness from childhood polio notice progression of the weakness as they get older. One theory is that the remaining motor neurons have to work harder and so die sooner.

David Nye (nyeda@cnsvax.uwec.edu). Midelfort Clinic, Eau Claire WI
This is patently absurd; but whoever wishes to become a philosopher must learn not to be frightened by absurdities. -- Bertrand Russell

51.1 Can you do the following?

- Extract all email addresses
- Distinguish the header from the text body
- Determine if there is a quote in the message (like there is above)
- Find the most frequent words for each category
- Come up with your own questions and answer them

```
[ ]:
```

Chapter 52

Project - Feature Engineering on the Titanic

The titanic dataset has a few columns from which you can use regex to extract information from. Feature engineering involves using existing columns of data to create new columns of data. You will work on doing just that in these exercises. Read it in and then answer the following questions.

```
[1]: import pandas as pd  
titanic = pd.read_csv('../data/titanic.csv')  
titanic.head()
```

	PassengerId	Survived	Pclass	Name	Sex	...	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	...	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	...	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	...	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	...	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	...	0	373450	8.0500	NaN	S

52.1 Exercises

Exercise 1

Extract the first character of the `Ticket` column and save it as a new column `ticket_first`. Find the total number of survivors, the total number of passengers, and the percentage of those who survived **by this column**. Next find the total survival rate for the entire dataset. Does this new column help predict who survived?

[]:

Exercise 2

If you did Exercise 2 correctly, you should see that only 7% of the people with tickets that began with ‘A’ survived. Find the survival rate for all those ‘A’ tickets by `Sex`.

[]:

Exercise 3

Find the survival rate by the last letter of the ticket. Is there any predictive power here?

[]:

Exercise 4

Find the length of each passengers name and assign to the `name_len` column. What is the minimum and maximum name length?

[]:

Exercise 5

Pass the `name_len` column to the `pd.cut` function. Also, pass a list of equal-sized cut points to the `bins` parameter. Assign the resulting Series to the `name_len_cat` column. Find the frequency count of each bin in this column.

[]:

Exercise 6

Is name length a good predictor of survival?

[]:

Exercise 7

Why do you think people with longer names had a better chance at survival?

[]:

Exercise 8

Using the titanic dataset, do your best to extract the title from a person's name. Examples of title are 'Mr.', 'Dr.', 'Miss', etc... Save this to a column called `title`. Find the frequency count of the titles.

[]:

Exercise 9

Does the title have good predictive value of survival?

[]:

Exercise 10

Create a pivot table of survival by title and sex. Use two aggregation functions, mean and size

[]:

Exercise 11

Attempt to extract the first name of each passenger into the column `first_name`. Are there are males and females with the same first name?

[]:

Exercise 12

The exercises have been an exercise in feature engineering. Several new features (columns) have been created from existing columns. Come up with your own feature and test it out on survival.

[]:

Part IX

Tidy Data

Chapter 53

Tidy Data with `melt`

We have analyzed several datasets, but have not done much work to change their structure or do any preprocessing before computation. We immediately began generating results and answering questions. Producing results is typically not the first step of a data analysis. The vast majority of datasets ‘in the wild’ will need some amount of inspection and preprocessing. And in some cases, the entire project will just be about cleaning the data so that it can be further processed by someone else.

This notebook will use many ideas formulated by Hadley Wickham to **tidy** data before introducing a few more steps in order to prepare it for machine learning and visualization.

There’s an infamous data science saying that goes something like this: “data scientists spend 80% of their time cleaning data and the other 20% complaining about cleaning the data.”

53.1 Tidy Data

Tidy data is a term coined by Hadley Wickham, the creator of many popular R packages, to describe a specific **structure** of data that makes for easy analysis. It is recommended that you read [his paper](#) to get a more complete understanding. The basics will be covered below.

Tidy data is a specific structure of data that makes analysis easier. A dataset is tidy when: 1. Each variable forms a column 2. Each observation forms a row 3. Each type of observational unit forms a table

Any dataset that does not meet this definition is considered “messy”.

First example of messy data

Messy data can appear deceptively clean and tidy, especially if you have not been exposed to it before. Let’s read in some data on the average arrival delay for different airlines flying out of different cities.

```
[1]: import pandas as pd  
arrival_delay = pd.read_csv('..../data/tidy/average_arrival_delay.csv')  
arrival_delay
```

	airline	ATL	DEN	DFW	IAH	...	LAX	MSP	ORD	PHX	SFO
0	AA	4.0	9.0	5.0	11.0	...	3.0	1.0	8.0	5.0	3.0
1	AS	6.0	-3.0	-5.0	1.0	...	-3.0	6.0	2.0	-9.0	4.0
2	B6	NaN	12.0	4.0	NaN	...	2.0	NaN	23.0	20.0	5.0
3	DL	0.0	-3.0	10.0	3.0	...	3.0	-1.0	7.0	-4.0	0.0
4	EV	7.0	14.0	10.0	3.0	...	NaN	10.0	8.0	-14.0	NaN
5	F9	20.0	10.0	26.0	1.0	...	8.0	35.0	22.0	16.0	15.0
6	HA	NaN	NaN	NaN	NaN	...	1.0	NaN	NaN	2.0	8.0
7	MQ	21.0	NaN	8.0	7.0	...	28.0	72.0	6.0	NaN	NaN
8	NK	26.0	8.0	15.0	28.0	...	24.0	19.0	23.0	4.0	NaN
9	OO	9.0	7.0	23.0	12.0	...	8.0	2.0	10.0	3.0	9.0
10	UA	5.0	6.0	8.0	9.0	...	4.0	6.0	13.0	0.0	5.0
11	US	1.0	-0.0	2.0	-3.0	...	4.0	-3.0	2.0	3.0	3.0
12	VX	NaN	NaN	NaN	NaN	...	5.0	NaN	19.0	NaN	5.0
13	WN	5.0	5.0	NaN	NaN	...	11.0	-0.0	NaN	6.0	4.0

What's wrong?

Even though the dataset returns perfectly readable and acceptable information, it is not technically a tidy data set. Generally speaking, it is easier to perform further analysis on a tidy dataset than other datasets. The notebook titled “Why Tidy Data?” covers several examples that compare messy to tidy data.

The main issue with the above dataset is that some of the column names are variable values themselves. At this point, you might be confused as to what exactly is meant by a ‘variable’. A simple definition of a variable is **anything that is liable to change**.

What are the variable names?

Only the `airline` column appears to be a variable name that is found directly in the DataFrame above. You must infer the others from the description of the problem. The variables are: + airline + origin airport + average arrival delay

Actual Tidying

To tidy, we need to make sure the three tidy rules are followed. Let’s attempt to restructure our data so that we have a three-column DataFrame with the column names from above.

- The airlines are already in a single column
- The origin airports are column names and need to be transposed into a single column
- The average arrival delay is tiled across the rows

53.2 Melting

Pandas contains a DataFrame method named `melt`, that can take columns and stack them one-by-one on top of each other. It has two important parameter:

- **id_vars** - a list (or single string) of column names that you want to keep as columns.
- **value_vars** - a list (or single string) of column names that you would like to reshape into one column

This ‘reshaping’ into one column is usually referred to as **melting**, **stacking**, or **unpivoting**. The **id_vars** columns will remain in the same column they currently reside, but **repeat** to align with all the newly melted values in the **value_vars** columns.

Let’s keep the `airline` column vertical and melt the origin airports:

```
[2]: airports = ['ATL', 'DEN', 'DFW', 'IAH', 'LAS', 'LAX', 'MSP', 'ORD', 'PHX', 'SFO']
ad_melted = arrival_delay.melt(id_vars='airline', value_vars=airports)
ad_melted.head(20)
```

	airline	variable	value
0	AA	ATL	4.0
1	AS	ATL	6.0
2	B6	ATL	NaN
3	DL	ATL	0.0
4	EV	ATL	7.0
5	F9	ATL	20.0
6	HA	ATL	NaN
7	MQ	ATL	21.0
8	NK	ATL	26.0
9	OO	ATL	9.0
10	UA	ATL	5.0
11	US	ATL	1.0
12	VX	ATL	NaN
13	WN	ATL	5.0
14	AA	DEN	9.0
15	AS	DEN	-3.0
16	B6	DEN	12.0
17	DL	DEN	-3.0
18	EV	DEN	14.0
19	F9	DEN	10.0

Renaming the columns with `var_name` and `value_name`

By default, the `melt` method will name the column containing the old column names as `variable`. The column containing the values of these columns is named `value`.

The `melt` method provides two additional parameters `var_name` and `value_name`. Set these parameters equal to column names of your choice.

value_vars is optional

By default `melt` will melt all the columns that are not `id_vars`. You don't have to explicitly put them in a list.

```
[3]: ad_melted = arrival_delay.melt(id_vars='airline',
                                    var_name='origin_airport',
                                    value_name='avg_arrival_delay')
ad_melted.head(20)
```

	airline	origin_airport	avg_arrival_delay
0	AA	ATL	4.0
1	AS	ATL	6.0
2	B6	ATL	NaN
3	DL	ATL	0.0
4	EV	ATL	7.0
5	F9	ATL	20.0
6	HA	ATL	NaN
7	MQ	ATL	21.0
8	NK	ATL	26.0
9	OO	ATL	9.0
10	UA	ATL	5.0
11	US	ATL	1.0
12	VX	ATL	NaN
13	WN	ATL	5.0
14	AA	DEN	9.0
15	AS	DEN	-3.0
16	B6	DEN	12.0
17	DL	DEN	-3.0
18	EV	DEN	14.0
19	F9	DEN	10.0

Our first tidy dataset

By ensuring that each variable forms its own column, each observation is also in its own row. We now have tidy data.

Key terms - reshaping and restructuring

When you think of tidy data, your brain should think about the terms **reshaping** or **restructuring**. The data is being maneuvered like a jigsaw puzzle. The actual data values are not changing (though some aspects of tidy data will change the values).

53.3 Exercises

Exercise 1

Read in the movie dataset. Select the title column and all of the actor name columns. Restructure the dataset so that there are only three variables - the title of the movie, the actor number (1, 2, or 3), and the actor name. Sort the result by title and output the result.

[]:

Exercise 2

Using the original movie dataset (and keeping its structure), attempt to count the total appearances of each actor in the dataset regardless whether they are 1, 2, or 3. Then repeat this task with your tidy dataset.

[]:

Exercise 3

Tidy the dataset in the `tidy/employee_messy1.csv` file. It contains the count of all employees by race and gender.

[]:

Exercise 4

Tidy the dataset in the `tidy/employee_messy2.csv` file. It contains the count of all employees by department, race and gender.

[]:

Exercise 5

Tidy the dataset in the `tidy/employee_salary_stats.csv` file. Save the tidy dataset to a variable and then select all the median salaries. The select all the median salaries with the original ‘messy’ dataset. Which one is easier to read summary statistics from?

[]:

Chapter 54

Reshaping by Pivoting

Let's recreate our tidy data from the previous notebook.

```
[1]: import pandas as pd  
arrival_delay = pd.read_csv('../data/tidy/average_arrival_delay.csv')  
ad_melted = arrival_delay.melt(id_vars='airline', var_name='origin_airport',  
                                value_name='avg_arrival_delay')  
ad_melted.head()
```

	airline	origin_airport	avg_arrival_delay
0	AA	ATL	4.0
1	AS	ATL	6.0
2	B6	ATL	NaN
3	DL	ATL	0.0
4	EV	ATL	7.0

54.1 Inverting melted data with pivot

Pandas has functionality to invert melted data back to its original messy form. This is sometimes called **pivoting** and is done with the **pivot** method. It has three available parameters.

- **index** - the column that will stay vertical. This column will be set as the index
- **columns** - The column which will be transposed and whose unique values will be made into column names
- **values** - The column which will be tiled across as the new values

```
[2]: df_pivot = ad_melted.pivot(index='airline',  
                               columns='origin_airport',  
                               values='avg_arrival_delay')  
df_pivot.head()
```

origin_airport	ATL	DEN	DFW	IAH	LAS	LAX	MSP	ORD	PHX	SFO
airline										
AA	4.0	9.0	5.0	11.0	8.0	3.0	1.0	8.0	5.0	3.0
AS	6.0	-3.0	-5.0	1.0	2.0	-3.0	6.0	2.0	-9.0	4.0
B6	NaN	12.0	4.0	NaN	11.0	2.0	NaN	23.0	20.0	5.0
DL	0.0	-3.0	10.0	3.0	-3.0	3.0	-1.0	7.0	-4.0	0.0
EV	7.0	14.0	10.0	3.0	NaN	NaN	10.0	8.0	-14.0	NaN

What is that extra data?

You may be disturbed by seeing `origin_airport` and `airline` in the upper left hand corner of the DataFrame. These two pieces of text are **names** for the their respective index. `origin_airport` is the name for the column index and `airline` is the name for the row index.

Remove this noise

We can use the row index name to our advantage. The `reset_index` method uses it as the column name.

```
[3]: df_pivot2 = df_pivot.reset_index()
df_pivot2.head()
```

origin_airport	airline	ATL	DEN	DFW	IAH	...	LAX	MSP	ORD	PHX	SFO
0	AA	4.0	9.0	5.0	11.0	...	3.0	1.0	8.0	5.0	3.0
1	AS	6.0	-3.0	-5.0	1.0	...	-3.0	6.0	2.0	-9.0	4.0
2	B6	NaN	12.0	4.0	NaN	...	2.0	NaN	23.0	20.0	5.0
3	DL	0.0	-3.0	10.0	3.0	...	3.0	-1.0	7.0	-4.0	0.0
4	EV	7.0	14.0	10.0	3.0	...	NaN	10.0	8.0	-14.0	NaN

The name of the columns

That column index name `origin_airport` is still there. Let's remove it with the `rename_axis` method. This method can rename either the row or column index. Here we choose to rename the column index to `None` which effectively removes it.

```
[4]: df_pivot2 = df_pivot2.rename_axis(None, axis='columns')
df_pivot2.head()
```

	airline	ATL	DEN	DFW	IAH	...	LAX	MSP	ORD	PHX	SFO
0	AA	4.0	9.0	5.0	11.0	...	3.0	1.0	8.0	5.0	3.0
1	AS	6.0	-3.0	-5.0	1.0	...	-3.0	6.0	2.0	-9.0	4.0
2	B6	NaN	12.0	4.0	NaN	...	2.0	NaN	23.0	20.0	5.0
3	DL	0.0	-3.0	10.0	3.0	...	3.0	-1.0	7.0	-4.0	0.0
4	EV	7.0	14.0	10.0	3.0	...	NaN	10.0	8.0	-14.0	NaN

All steps in a single cell

```
[5]: df_pivot = ad_melted.pivot(index='airline', columns='origin_airport',
                                values='avg_arrival_delay')
df_pivot = df_pivot.reset_index()
df_pivot = df_pivot.rename_axis(None, axis='columns')
df_pivot.head()
```

	airline	ATL	DEN	DFW	IAH	...	LAX	MSP	ORD	PHX	SFO
0	AA	4.0	9.0	5.0	11.0	...	3.0	1.0	8.0	5.0	3.0
1	AS	6.0	-3.0	-5.0	1.0	...	-3.0	6.0	2.0	-9.0	4.0
2	B6	NaN	12.0	4.0	NaN	...	2.0	NaN	23.0	20.0	5.0
3	DL	0.0	-3.0	10.0	3.0	...	3.0	-1.0	7.0	-4.0	0.0
4	EV	7.0	14.0	10.0	3.0	...	NaN	10.0	8.0	-14.0	NaN

54.2 Pivoting with duplicate values

Let's create a new table very similar to our melted table. We will sample the rows with replacement using the `sample` method to create 1,000 rows of data. This guarantees duplicates in our columns.

```
[6]: df_dupes = ad_melted.sample(n=1000, replace=True)
df_dupes.shape
```

```
[6]: (1000, 3)
```

```
[7]: df_dupes.head()
```

	airline	origin_airport	avg_arrival_delay
114	B6	PHX	20.0
0	AA	ATL	4.0
21	MQ	DEN	NaN
95	US	MSP	-3.0
28	AA	DFW	5.0

Attempting to reshape this won't work

The `df_dupes` DataFrame has the same exact columns and the same values for airline and `origin_airport` as `ad_melted` but will produce an error when calling the `pivot` method. This is because there exists more than one row for each airline and airport combination.

For example, there are many rows that contain the airline as AA and the airport as IAH. Because there are multiple values for this particular intersection, the `pivot` method will not work (even if all of them are the same).

```
[8]: df_dupes.pivot(index='airline',
                   columns='origin_airport',
                   values='avg_arrival_delay')
```

`ValueError: Index contains duplicate entries, cannot reshape`

Verify this with `pivot_table`

The `pivot` method works when there exist exactly one unique combination for each intersection, like it does in the `ad_melted` DataFrame. You can verify the number of occurrences of each airline-airport combination with the `pivot_table` method:

```
[9]: ad_melted.pivot_table(index='airline', columns='origin_airport', aggfunc='size')
```

origin_airport	ATL	DEN	DFW	IAH	LAS	LAX	MSP	ORD	PHX	SFO
airline										
AA	1	1	1	1	1	1	1	1	1	1
AS	1	1	1	1	1	1	1	1	1	1
B6	1	1	1	1	1	1	1	1	1	1
DL	1	1	1	1	1	1	1	1	1	1
EV	1	1	1	1	1	1	1	1	1	1
F9	1	1	1	1	1	1	1	1	1	1
HA	1	1	1	1	1	1	1	1	1	1
MQ	1	1	1	1	1	1	1	1	1	1
NK	1	1	1	1	1	1	1	1	1	1
OO	1	1	1	1	1	1	1	1	1	1
UA	1	1	1	1	1	1	1	1	1	1
US	1	1	1	1	1	1	1	1	1	1
VX	1	1	1	1	1	1	1	1	1	1
WN	1	1	1	1	1	1	1	1	1	1

```
[10]: df_dupes.pivot_table(index='airline', columns='origin_airport', aggfunc='size')
```

origin_airport	ATL	DEN	DFW	IAH	LAS	LAX	MSP	ORD	PHX	SFO
airline										
AA	7	5	6	8	6	7	8	14	7	8
AS	7	7	4	12	9	7	10	12	7	9
B6	3	3	6	8	8	5	6	8	6	10
DL	6	6	7	8	6	5	7	2	5	9
EV	1	5	8	11	7	8	8	4	5	7
F9	8	7	3	4	6	9	8	10	9	10
HA	5	7	11	5	9	10	10	6	7	9
MQ	6	6	7	5	8	18	6	8	13	5
NK	2	9	8	7	11	7	7	4	8	5
OO	7	5	9	7	5	4	3	6	14	7
UA	6	4	2	10	9	10	11	5	7	8
US	7	9	5	3	3	7	9	5	11	5
VX	9	8	3	7	5	9	9	7	2	9
WN	6	5	7	15	12	6	9	6	9	3

54.3 Using pivot_table to aggregate those values

As an alternative, use `pivot_table` which is able to aggregate all the values at the intersection. In this particular instance all the values at each intersection are the same, so `min`, `max`, `mean`, and `median` will all produce the same result.

```
[11]: ad_melted.pivot_table(index='airline', columns='origin_airport',
                           values='avg_arrival_delay', aggfunc='max').head()
```

origin_airport	ATL	DEN	DFW	IAH	LAS	LAX	MSP	ORD	PHX	SFO
airline										
AA	4.0	9.0	5.0	11.0	8.0	3.0	1.0	8.0	5.0	3.0
AS	6.0	-3.0	-5.0	1.0	2.0	-3.0	6.0	2.0	-9.0	4.0
B6	NaN	12.0	4.0	NaN	11.0	2.0	NaN	23.0	20.0	5.0
DL	0.0	-3.0	10.0	3.0	-3.0	3.0	-1.0	7.0	-4.0	0.0
EV	7.0	14.0	10.0	3.0	NaN	NaN	10.0	8.0	-14.0	NaN

54.4 Exercises

Exercise 1

Read the file `tidy/clean_movie1.csv` and then use the `pivot` method to put the country names as the columns. Put the count as the new values for the DataFrame.

[]:

Exercise 2

Read in the NYC deaths dataset and select only males from 2007. Pivot this information so we can more clearly see the breakdown of causes of death by race. Assign the result to a variable.

[]:

Exercise 3

Use the result from Exercise 2 and highlight the leading cause of death for each race. Is it the same for each one?

[]:

Exercise 4

Read in the flights dataset. Find the total number of flights from each airline by their origin airport. Hint: When making a pivot table of just frequency, its not necessary to have a `values` column. Save the results to a variable.

[]:

Exercise 5

Highlight the origin airport with the most flights for each airline. Do a few online searches to determine if those airports are hubs for those airlines.

[]:

Exercise 6

Read in the bikes dataset. For each type of weather event (the `events` column) find the median temperature for males and females.

[]:

Exercise 7

Reshape the movie dataset so that there are two columns, one for all of the actors and one for the content rating of each of their respective movies. Filter this DataFrame so that it contains the top 10 most common actors. Then create a table that displays the number of movies each actor made by content rating. The actor names should be in the index, with the content ratings in the columns, with the counts as the values.

[]:

Chapter 55

Common Messy Datasets

The previous notebooks focused on one particular type of messy dataset. A dataset where the column names are actually variable values and not variable names. This was illustrated with the dataset on arrival delay. The `melt` method will quickly tidy these basic datasets. But, often is the case that datasets take more manipulation to make them tidy. This notebook covers several more common messy datasets.

55.1 Most common messy data problems

1. Column names are variable values, not variable names.
2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables

The first type of messy data was covered in the previous notebook. This notebook will cover the next three examples.

55.2 Multiple variables are stored in one column

A tidy data set requires that values of a single variable are stored in one column.

Column names appear as values in a column

Take a look at the dataset below. Notice how the `Value` column has both numeric and string data types and the `Info` column contains variable names.

```
[1]: import pandas as pd
df = pd.DataFrame(data={'State': ['Texas', 'Arizona', 'Florida'] * 3,
                       'Info': ['Age'] * 3 + ['Salary'] * 3 + ['Hair Color'] * 3,
                       'Value': [10, 15, 20, 3, 4, 5, 'Brown', 'Pink', 'Red']},
                   columns=['State', 'Info', 'Value'])
```

	State	Info	Value
0	Texas	Age	10
1	Arizona	Age	15
2	Florida	Age	20
3	Texas	Salary	3
4	Arizona	Salary	4
5	Florida	Salary	5
6	Texas	Hair Color	Brown
7	Arizona	Hair Color	Pink
8	Florida	Hair Color	Red

The fix

This dataset has three variables in a single column. You can think of it as ‘overly melted’. Pivoting it with the `pivot` method will make it tidy.

```
[2]: df.pivot(index='State', columns='Info', values='Value')
```

	Info	Age	Hair Color	Salary
State				
Arizona	15	Pink	4	
Florida	20	Red	5	
Texas	10	Brown	3	

```
[3]: df_tidy = df.pivot(index='State', columns='Info', values='Value').reset_index()
df_tidy = df_tidy.rename_axis(None, axis='columns')
df_tidy
```

	State	Age	Hair Color	Salary
0	Arizona	15	Pink	4
1	Florida	20	Red	5
2	Texas	10	Brown	3

Checking data types

Whenever we have a mix of variables in a single column, you might also have a mix of data types. It’s important to check the data types after reshaping the data.

```
[4]: df_tidy.dtypes
```

```
[4]: State      object
Age       object
Hair Color  object
Salary     object
dtype: object
```

Changing data types

Both `Age` and `Salary` should be integers but instead are objects. We need to change their data types. We have seen this previously with the function `pd.to_numeric`, but you can also see use the `astype` method. They both do nearly the same thing, except `pd.numeric` gives you more options (which were needed in a previous notebook). We will use each here.

```
[5]: df_tidy['Age'] = df_tidy['Age'].astype('int')
df_tidy['Salary'] = pd.to_numeric(df_tidy['Salary'])
df_tidy.dtypes
```

```
[5]: State      object
Age       int64
Hair Color  object
Salary     int64
dtype: object
```

55.3 Two or more values are stored in the same cell

Two or more values of the same variable or different variable can be stored in the same cell in a DataFrame. You will need to extract the desired quantities which might necessitate regular expressions. Let's take a look at a dataset with multiple variables stored in a single cell.

```
[6]: geo = pd.DataFrame({'City': ['Houston', 'Dallas', 'Austin'],
                       'Geolocation': [ '(29.7604° N, 95.3698° W)', 
                                       '32.7767° N, 96.7970° W',
                                       '30.2672° N, 97.7431° W']})
geo
```

	City	Geolocation
0	Houston	(29.7604° N, 95.3698° W)
1	Dallas	32.7767° N, 96.7970° W
2	Austin	30.2672° N, 97.7431° W

Identify the Variables

The first step in tidying data is identifying the variables. The `Geolocation` column has quite a lot of information packed into it. We will parse it into 4 separate variables.

- latitude
- latitude direction
- longitude
- longitude direction

Extracting information with regular expressions using the str accessor

The `extract` string method takes a regular expression with **capture groups** and returns each captured group as a new column. Our regular expression has 4 capture groups. One for each variable.

```
([0-9.]*) .*?([NS]).*?([0-9.]*) .*?([EW])
```

My explanation

- `([0-9.]*)` - This is a capture group that matches one or more of the digits 0-9 and the literal `.`
- `.*?([NS])` - Matches any number of characters in a non-greedy fashion before capturing N or S.
- The pattern then repeats to capture the longitude and direction

```
[7]: regex = r'([0-9.]*) .*?([NS]).*?([0-9.]*) .*?([EW])'
geo_extract = geo['Geolocation'].str.extract(regex)
geo_extract
```

	0	1	2	3
0	29.7604	N	95.3698	W
1	32.7767	N	96.7970	W
2	30.2672	N	97.7431	W

Column names

pandas defaults the column names of the resulting DataFrame to integers. We would like these new columns appended to our original DataFrame.

Creating multiple new column names

It is possible to create several new columns in our original DataFrame by simply assigning the above resulting DataFrame to a selection of new column names as a list.

```
[8]: geo[['latitude', 'latitude dir', 'longitude', 'longitude dir']] = geo_extract
geo
```

	City	Geolocation	latitude	latitude dir	longitude	longitude dir
0	Houston	(29.7604° N, 95.3698° W)	29.7604		95.3698	W
1	Dallas	32.7767° N, 96.7970° W	32.7767		96.7970	W
2	Austin	30.2672° N, 97.7431° W	30.2672		97.7431	W

Dropping the original column

We can remove the `Geolocation` column as we have finished processing it.

```
[9]: geo = geo.drop(columns='Geolocation')
geo
```

	City	latitude	latitude dir	longitude	longitude dir
0	Houston	29.7604	N	95.3698	W
1	Dallas	32.7767	N	96.7970	W
2	Austin	30.2672	N	97.7431	W

Check and change Data Types

Both latitude and longitude are clearly supposed to be numeric (floats) but since they were extracted from a string, remain as strings. Let's change them to float.

```
[10]: geo.dtypes
```

```
[10]: City          object
latitude      object
latitude dir   object
longitude     object
longitude dir  object
dtype: object
```

```
[11]: geo['latitude'] = geo['latitude'].astype('float')
geo['longitude'] = geo['longitude'].astype('float')
geo.dtypes
```

```
[11]: City          object
latitude      float64
latitude dir   object
longitude     float64
longitude dir  object
dtype: object
```

We now have tidied this dataset which had multiple values stored in a single cell.

```
[12]: geo
```

	City	latitude	latitude dir	longitude	longitude dir
0	Houston	29.7604	N	95.3698	W
1	Dallas	32.7767	N	96.7970	W
2	Austin	30.2672	N	97.7431	W

55.4 Variables are stored in both rows and columns

A more difficult situation occurs when variables are stored down a column and across the column names. Pivoting and melting may have to be used together to make it tidy. Let's take a look at the example below.

```
[13]: tfp = pd.read_csv('../data/tidy/temp_flow_pressure.csv')
tfp
```

Group	Property	2012	2013	2014	2015	2016	
0	A	Pressure	928	873	814	973	870
1	A	Temperature	1026	1038	1009	1036	1042
2	A	Flow	819	806	861	882	856
3	B	Pressure	817	877	914	806	942
4	B	Temperature	1008	1041	1009	1002	1013
5	B	Flow	887	899	837	824	873

Identifying the Variables

Identifying variables in this dataset is not as straightforward as it is in others. There are variable values stored across multiple rows and variable values stored as column names.

We can use the following as variables:

- Group
 - Pressure
 - Temperature
 - Flow
 - Years

The years are column names and the pressure, temperature, and flow are values in the property column. The Group column is the only one in the correct place.

Melt the years

Tidying this particular dataset must happen in multiple stages. We won't be able to tidy each variable at the same time. We will begin by melting the year column names into a single column.

	Group	Property	Year	value
0	A	Pressure	2012	928
1	A	Temperature	2012	1026
2	A	Flow	2012	819
3	B	Pressure	2012	817
4	B	Temperature	2012	1008
5	B	Flow	2012	887
6	A	Pressure	2013	873
7	A	Temperature	2013	1038
8	A	Flow	2013	806
9	B	Pressure	2013	877

Need to pivot Property

We now need to pivot the **Property** column so that the values become column names, and keep Group and Year as columns. The values will come from the **value** column.

Problem! pivot only works with a single column as the index

If we try and pivot by passing a list of values to the **index** parameter, we get an error. Pandas actually thinks we are using the `['Group', 'Year']` not as column names but as values to pivot.

```
[15]: tfp_melt.pivot(index=['Group', 'Year'], columns='Property', values='value')
```

`ValueError: Length of passed values is 30, index implies 2.`

Must use pivot_table

The **pivot_table** method does allow us to keep multiple columns in the index. Multiple aggregation functions produce the same result as there is only one value to aggregate per group.

```
[16]: tfp_tidy = tfp_melt.pivot_table(index=['Group', 'Year'], columns='Property',
                                     values='value', aggfunc='max')
tfp_tidy
```

	Property	Flow	Pressure	Temperature
Group	Year			
A	2012	819	928	1026
	2013	806	873	1038
	2014	861	814	1009
	2015	882	973	1036
	2016	856	870	1042
B	2012	887	817	1008
	2013	899	877	1041
	2014	837	914	1009
	2015	824	806	1002
	2016	873	942	1013

Verify that there is one value per intersection

Let's verify that there is one value per intersection.

```
[17]: tfp_melt.pivot_table(index=['Group', 'Year'], columns='Property',
                           values='value', aggfunc='size')
```

	Property	Flow	Pressure	Temperature
Group	Year			
A	2012	1	1	1
	2013	1	1	1
	2014	1	1	1
	2015	1	1	1
	2016	1	1	1
B	2012	1	1	1
	2013	1	1	1
	2014	1	1	1
	2015	1	1	1
	2016	1	1	1

Clean-up

```
[18]: tfp_tidy = tfp_tidy.reset_index()
tfp_tidy = tfp_tidy.rename_axis(None, axis='columns')
tfp_tidy
```

	Group	Year	Flow	Pressure	Temperature
0	A	2012	819	928	1026
1	A	2013	806	873	1038
2	A	2014	861	814	1009
3	A	2015	882	973	1036
4	A	2016	856	870	1042
5	B	2012	887	817	1008
6	B	2013	899	877	1041
7	B	2014	837	914	1009
8	B	2015	824	806	1002
9	B	2016	873	942	1013

```
[19]: tfp_tidy.dtypes
```

```
[19]: Group      object
      Year       object
      Flow      int64
      Pressure   int64
      Temperature int64
      dtype: object
```

Convert Year to integer

```
[20]: tfp_tidy['Year'] = tfp_tidy['Year'].astype('int')
      tfp_tidy.dtypes
```

```
[20]: Group      object
      Year      int64
      Flow      int64
      Pressure   int64
      Temperature int64
      dtype: object
```

55.5 Steps to produce tidy data

There won't be an exact set of procedures that will always result in a tidy dataset. This guideline may help you turn messy data into tidy data.

1. Identify each variable
2. Look for variable values masquerading as column names
3. Look for column names masquerading as variable values
4. Examine the 5 types of common messy data sets to see which one your dataset most closely resembles
5. You will likely need to use `melt`, `pivot`, and `pivot_table`
6. You might need to separate different variables into their own DataFrame to make for easier tidying
7. Parse string data with the `str` accessor with the help of regular expressions.

55.6 Exercises

Exercise 1

Make the `tidy/country_hour_price.csv` dataset tidy by putting all the hour columns into a single column.

[]:

Exercise 2

If the resulting DataFrame from Exercise 1 has the strings ‘HOUR1’ and ‘HOUR2’ as values in the hour column, then extract just the numerical part of the strings and reassign the result to the hour column.

[]:

Exercise 3

Tidy the `tidy/flights_status.csv` dataset.

[]:

Exercise 4

Tidy the `tidy/metrics.csv` dataset.

[]:

Part X

Joining Data

Chapter 56

Automatic Index Alignment

This chapter discusses **automatic index alignment**, a surprising, occasionally useful, and sometimes frustrating feature built into pandas. Automatic alignment of the index happens when operating on two pandas objects at the same time. Whether operating with two Series, two DataFrames, or one of each, automatic alignment of the index takes place first and then the operation completes.

56.1 Adding two Series - Not as simple as it sounds

Adding two Series together should be a simple, and most of the time it is, but you can be in for quite a surprise if the indexes do not align. Let's create two identical Series. The `copy` method allows us to do this.

```
[1]: import numpy as np
import pandas as pd
s1 = pd.Series(index=['a', 'b', 'c', 'd'], data=[0, 1, 2, 3])
s2 = s1.copy()
```

```
[2]: s1
```

```
[2]: a    0
      b    1
      c    2
      d    3
      dtype: int64
```

```
[3]: s2
```

```
[3]: a    0
      b    1
      c    2
      d    3
      dtype: int64
```

Note, that these are two distinct objects. If we wrote `s2 = s1`, we would not have created a new object, just two variable names that refer to the same object.

```
[4]: s1 is s2
```

```
[4]: False
```

Add the Series together

The Series have the same index and the same values. No surprises here.

```
[5]: s1 + s2
```

```
[5]: a    0  
      b    2  
      c    4  
      d    6  
      dtype: int64
```

Create a new Series with index values in a different order

We create new Series `s3` below with the same index values but in a different position than `s1`

```
[6]: s3 = pd.Series(index=['d', 'c', 'b', 'a'], data=[0, 1, 2, 3])  
s3
```

```
[6]: d    0  
      c    1  
      b    2  
      a    3  
      dtype: int64
```

Add `s1` to `s3`

```
[7]: s1 + s3
```

```
[7]: a    3  
      b    3  
      c    3  
      d    3  
      dtype: int64
```

What happened?

Pandas aligns the data first by the index and then completes the operation. Index ‘a’ aligns for both Series. In `s1` index ‘a’ labels value 3 and in `s3` it labels value 0. Added together they sum to 3. All the indexes align in this manner and all sum to 3.

56.2 Adding a numpy array to a Series

NumPy arrays have no index, just values and integer locations that refer to those values. NumPy arrays align by their integer location (which is what you would expect). Let’s create a simple array with integers 0 to 3 and add it to our Series from above. The index of the Series plays no role in the following operations.

```
[8]: a = np.arange(4)  
a
```

```
[8]: array([0, 1, 2, 3])
```

```
[9]: s1 + a
```

```
[9]: a    0
      b    2
      c    4
      d    6
      dtype: int64
```

```
[10]: s3 + a
```

```
[10]: d    0
      c    2
      b    4
      a    6
      dtype: int64
```

Adding the array to itself also aligns by integer location.

```
[11]: a + a
```

```
[11]: array([0, 2, 4, 6])
```

Adding arrays to Series - Must have same number of elements

For a successful array to Series addition to occur, they both need to have the same number of elements or else an error will occur.

```
[12]: a = np.arange(5)
      a
```

```
[12]: array([0, 1, 2, 3, 4])
```

```
[13]: try:
        s1 + a
    except Exception as e:
        print(type(e), e)
```

```
<class 'ValueError'> operands could not be broadcast together with shapes (4,) (5,)
```

56.3 Adding Series that don't have the same index labels

Adding Series that do not have the same index labels is possible. In fact, adding two Series together will always complete (unless their values are incompatible - such as adding a number to a string).

In the following example, we have two Series of different lengths. `s1` has one more index label, `d`, than `s2` does not have. When we add them together, again the indexes align, except for the `d`. It has no matching index in `s2`. Pandas keeps this label in the returned Series but with a missing value. Any label that does not match in the other Series is always kept and its associated value will always be missing.

```
[14]: s1 = pd.Series(index=['a', 'b', 'c', 'd'], data=[0, 1, 2, 3])
      s2 = pd.Series(index=['a', 'b', 'c'], data=[0, 1, 2])
```

```
[15]: s1 + s2
```

```
[15]: a    0.0
      b    2.0
      c    4.0
      d    NaN
      dtype: float64
```

Missing index labels in each Series

If each of the Series have index labels that do not appear in the other, then they will both be kept in the result with missing values.

```
[16]: s1 = pd.Series(index=['a', 'b', 'c', 'd'], data=[0, 1, 2, 3])
      s2 = pd.Series(index=['a', 'b', 'c', 'e'], data=[0, 1, 2, 3])
```

```
[17]: s1 + s2
```

```
[17]: a    0.0
      b    2.0
      c    4.0
      d    NaN
      e    NaN
      dtype: float64
```

56.4 Adding Series with duplicate values in the index

A big surprise awaits when you add two Series that each share duplicated index labels. Take a look at both Series below. `s1` and `s2` each have 3 ‘a’, index labels. `s1` has 3 ‘b’, 4 ‘c’ and 1 ‘d’ index label while `s2` has 2 ‘b’, 1 ‘c’, 1 ‘e’ labels. Let’s add them together to see what happens.

```
[18]: s1 = pd.Series(index=['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'c', 'd'],
                  data=np.arange(11))
      s2 = pd.Series(index=['a', 'a', 'a', 'b', 'b', 'c', 'e'], data=np.arange(7))
```

```
[19]: s1
```

```
[19]: a    0
      a    1
      a    2
      b    3
      b    4
      b    5
      c    6
      c    7
      c    8
      c    9
      d    10
      dtype: int64
```

```
[20]: s2
```

```
[20]: a    0  
      a    1  
      a    2  
      b    3  
      b    4  
      c    5  
      e    6  
dtype: int64
```

```
[21]: s1 + s2
```

```
[21]: a    0.0  
      a    1.0  
      a    2.0  
      a    1.0  
      a    2.0  
      a    3.0  
      a    2.0  
      a    3.0  
      a    4.0  
      b    6.0  
      b    7.0  
      b    7.0  
      b    8.0  
      b    8.0  
      b    9.0  
      c   11.0  
      c   12.0  
      c   13.0  
      c   14.0  
      d    NaN  
      e    NaN  
dtype: float64
```

```
[22]: len(s1 + s2)
```

```
[22]: 21
```

A Cartesian product has taken place

Each index label ‘a’ from Series `s1` aligns with each index label ‘a’ from `s2`. There are 3 ‘a’ labels in each which creates a total of 9 in the result. This is what is meant by a **Cartesian Product**. All possible combinations of same index labels in each Series will have a result.

Similarly, Series `s1` has 3 ‘b’ labels and `s2` has 2 ‘b’ for a total of 6 in the result. Simply multiply the count of the labels in each Series together to get the total labels in the result.

Label ‘c’ is found 4 times in `s1` and 1 time in `s2` for a total of 4 in the result. Labels ‘d’ and ‘e’ are unique to each Series so only occur once in the result with a missing value.

56.5 An exception to Cartesian Product rule

If both Series share the exact same index labels then no Cartesian product will occur.

```
[23]: s1 = pd.Series(index=['a', 'a', 'a', 'b', 'b'], data=np.arange(5))
s2 = pd.Series(index=['a', 'a', 'a', 'b', 'b'], data=np.arange(5))
```

```
[24]: s1 + s2
```

```
[24]: a    0
      a    2
      a    4
      b    6
      b    8
      dtype: int64
```

But even if one index label is different than a Cartesian product will happen:

```
[25]: s1 = pd.Series(index=['a', 'a', 'a', 'b', 'b'], data=np.arange(5))
s2 = pd.Series(index=['a', 'a', 'a', 'b', 'b', 'c'], data=np.arange(6))
s1 + s2
```

```
[25]: a    0.0
      a    1.0
      a    2.0
      a    1.0
      a    2.0
      a    3.0
      a    2.0
      a    3.0
      a    4.0
      b    6.0
      b    7.0
      b    7.0
      b    8.0
      c    NaN
      dtype: float64
```

Cartesian product still happens if order is not the same

Even if the index labels share the same number of occurrences in the Series, a Cartesian Product will still happen if the order is different. Below, `s1` and `s2` have the same number of 'a' and 'b' labels but have a different order for the 3rd and 4th labels.

```
[26]: s1 = pd.Series(index=['a', 'a', 'b', 'a', 'b'], data=np.arange(5))
s2 = pd.Series(index=['a', 'a', 'a', 'b', 'b'], data=np.arange(5))
s1 + s2
```

```
[26]: a    0
      a    1
      a    2
      a    1
      a    2
```

```
a    3
a    3
a    4
a    5
b    5
b    6
b    7
b    8
dtype: int64
```

56.6 DataFrames align on both their index and columns

```
[27]: df1 = pd.DataFrame(data={'first': np.arange(4), 'second': np.arange(4)},
                       index=['a', 'b', 'c', 'd'])
df2 = df1.copy()
```

Operations happen as expected whenever index and columns match exactly.

```
[28]: df1 + df2
```

	first	second
a	0	0
b	2	2
c	4	4
d	6	6

DataFrame Index alignment

The label needs to be present in both DataFrames for a value to be computed or else it will be missing.

```
[29]: df1 = pd.DataFrame(data={'first': np.arange(4), 'second': np.arange(4)},
                       index=['a', 'b', 'c', 'd'])
df2 = pd.DataFrame(data={'first': np.arange(4), 'second': np.arange(4)},
                   index=['a', 'b', 'c', 'e'])
df1
```

	first	second
a	0	0
b	1	1
c	2	2
d	3	3

```
[30]: df2
```

	first	second
a	0	0
b	1	1
c	2	2
e	3	3

```
[31]: df1 + df2
```

	first	second
a	0.0	0.0
b	2.0	2.0
c	4.0	4.0
d	NaN	NaN
e	NaN	NaN

When Columns do not align

```
[32]: df1 = pd.DataFrame(data={'first': np.arange(4), 'second': np.arange(4)},
                       index=['a', 'b', 'c', 'd'])
df2 = pd.DataFrame(data={'first': np.arange(4), 'third': np.arange(4)},
                   index=['a', 'b', 'c', 'e'])
df1
```

	first	second
a	0	0
b	1	1
c	2	2
d	3	3

```
[33]: df2
```

	first	third
a	0	0
b	1	1
c	2	2
e	3	3

```
[34]: df1 + df2
```

	first	second	third
a	0.0	NaN	NaN
b	2.0	NaN	NaN
c	4.0	NaN	NaN
d	NaN	NaN	NaN
e	NaN	NaN	NaN

Chapter 57

Combining Data

Most data analyses will use multiple different datasets or at least multiple datasets created from the same source. pandas has tools to combine DataFrames in a wide variety of ways.

57.1 Concatenating Data

Concatenating data in pandas refers to stacking DataFrames either one on top of each other or side by side. The `pd.concat` function (NOT a method) is flexible and versatile with many different arguments that give you power to combine two or more datasets at the same time.

Concatenating very similar DataFrames

The `pd.concat` function provides many different and sometimes confusing arguments. We read in two small DataFrames with just three columns and three rows of each. We will use these small datasets to illustrate how the `concat` function works.

```
[1]: import pandas as pd  
amzn = pd.read_csv('../data/stocks/amzn_sample.csv', parse_dates=['date'])  
aapl = pd.read_csv('../data/stocks/aapl_sample.csv', parse_dates=['date'])
```

Stacking data one on top of the other

The first argument for `concat` needs to be a list of DataFrames. As usual in Pandas, the default is to do the action vertically. We stack them with the following command:

```
[2]: pd.concat([amzn, aapl])
```

	date	adjusted_close	volume
0	2018-03-23	1495.56	7843966.0
1	2018-03-26	1555.86	5547618.0
2	2018-03-27	1497.05	6793279.0
0	2018-03-23	164.94	40248954.0
1	2018-03-26	172.77	36272617.0
2	2018-03-27	168.34	38962839.0

Notice that the index was kept the same. Use `ignore_index=True` to make a completely new RangeIndex from 0 to n-1.

```
[3]: pd.concat([amzn, aapl], ignore_index=True)
```

	date	adjusted_close	volume
0	2018-03-23	1495.56	7843966.0
1	2018-03-26	1555.86	5547618.0
2	2018-03-27	1497.05	6793279.0
3	2018-03-23	164.94	40248954.0
4	2018-03-26	172.77	36272617.0
5	2018-03-27	168.34	38962839.0

```
[4]: pd.concat([amzn, aapl], ignore_index=True)
```

	date	adjusted_close	volume
0	2018-03-23	1495.56	7843966.0
1	2018-03-26	1555.86	5547618.0
2	2018-03-27	1497.05	6793279.0
3	2018-03-23	164.94	40248954.0
4	2018-03-26	172.77	36272617.0
5	2018-03-27	168.34	38962839.0

Label each piece of the DataFrame with the `keys` parameter

You can use the `keys` parameter to label each piece of the DataFrame. This creates a MultiLevel index.

```
[5]: pd.concat([amzn, aapl], keys=['amzn', 'aapl'])
```

	date	adjusted_close	volume
amzn	0	2018-03-23	1495.56
	1	2018-03-26	1555.86
	2	2018-03-27	1497.05
aapl	0	164.94	40248954.0
	1	172.77	36272617.0
	2	168.34	38962839.0

Perhaps its better to just make a new column beforehand

```
[6]: amzn['symbol'] = 'amzn'
aapl['symbol'] = 'aapl'
pd.concat([amzn, aapl])
```

	date	adjusted_close	volume	symbol
0	2018-03-23	1495.56	7843966.0	amzn
1	2018-03-26	1555.86	5547618.0	amzn
2	2018-03-27	1497.05	6793279.0	amzn
0	2018-03-23	164.94	40248954.0	aapl
1	2018-03-26	172.77	36272617.0	aapl
2	2018-03-27	168.34	38962839.0	aapl

57.2 Beware! Automatic Alignment of Index

Of extreme importance to `pd.concat` (and all of pandas) is the automatic alignment of indexes that happens behind the scenes. For instance, let's change the second column of `amzn_head` and concatenate once again.

```
[7]: amzn2 = amzn.rename(columns={'Adj. Close': 'close'})
pd.concat([amzn2, aapl])
```

	date	adjusted_close	volume	symbol
0	2018-03-23	1495.56	7843966.0	amzn
1	2018-03-26	1555.86	5547618.0	amzn
2	2018-03-27	1497.05	6793279.0	amzn
0	2018-03-23	164.94	40248954.0	aapl
1	2018-03-26	172.77	36272617.0	aapl
2	2018-03-27	168.34	38962839.0	aapl

57.3 Column names align first

`pd.concat` does automatic alignment on the columns and by default does an outer join. Notice the missing values where the misalignment is. We can force an `inner` join, where only the columns in common are kept.

```
[8]: pd.concat([amzn2, aapl], join='inner')
```

	date	adjusted_close	volume	symbol
0	2018-03-23	1495.56	7843966.0	amzn
1	2018-03-26	1555.86	5547618.0	amzn
2	2018-03-27	1497.05	6793279.0	amzn
0	2018-03-23	164.94	40248954.0	aapl
1	2018-03-26	172.77	36272617.0	aapl
2	2018-03-27	168.34	38962839.0	aapl

57.4 Use axis=1 to change the direction of concatenation

An automatic alignment on the index still happens here.

```
[9]: pd.concat([amzn2, aapl], axis=1)
```

	date	adjusted_close	volume	symbol		date	adjusted_close	volume	symbol
0	2018-03-23	1495.56	7843966.0	amzn	2018-03-23	164.94	40248954.0	aapl	
1	2018-03-26	1555.86	5547618.0	amzn	2018-03-26	172.77	36272617.0	aapl	
2	2018-03-27	1497.05	6793279.0	amzn	2018-03-27	168.34	38962839.0	aapl	

Chapter 58

SQL Databases

58.1 Connecting to a SQL database

You can query SQL database tables with the `read_sql` command.

Setting up your connection with SQLAlchemy

Pandas relies on a third-party library called [SQLAlchemy](#) to establish a connection to a database.

Connection string

To make the connection, we need to pass a connection string to the `create_engine` function. The general form of a connection string is the following:

```
dialect+driver://username:password@host:port/database
```

Read more about [engine configuration here](#).

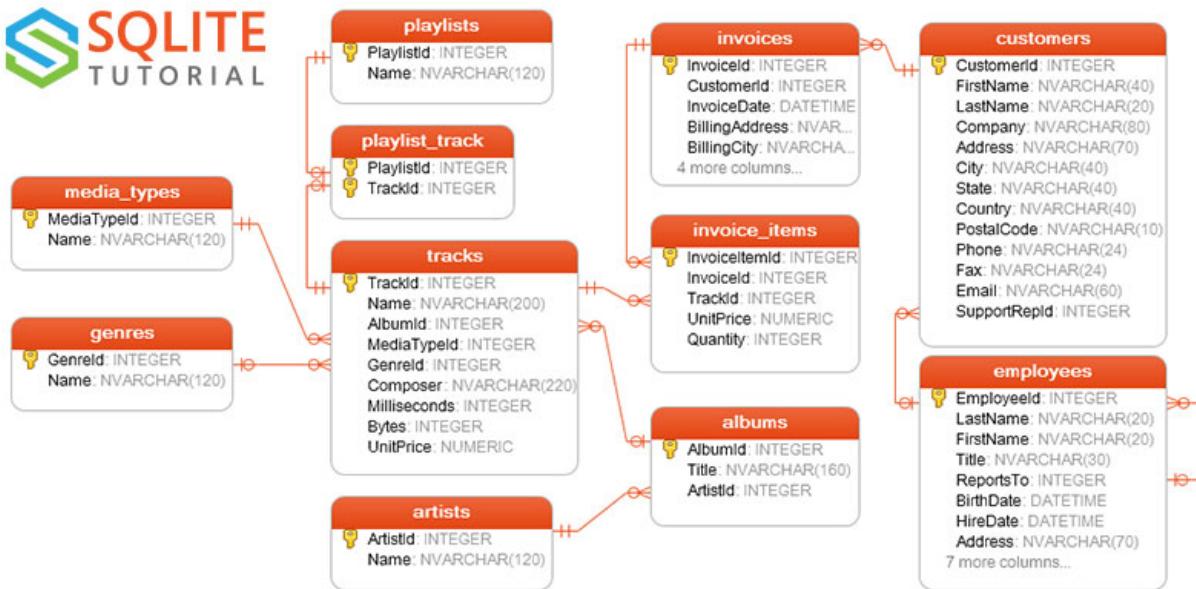
Connection string for sqlite

We will be using sqlite databases in this notebook. Its [connection string](#) is even simpler:

```
sqlite:///<path_to_db>
```

58.2 The Chinook Database

A proper relational database will have a diagram depicting the tables, columns and their data types, and relationships between them.



58.3 Primary and Foreign Keys

A key component of relational databases is the idea of primary and foreign keys. A primary key is a column whose value uniquely identifies each row in the table. A foreign key is a primary key located in a different table than where it is the primary key. A foreign key is not unique and can appear any number of times within its table.

In the above diagram, all the primary keys have a little key symbol next to them. For example, in the **tracks** table, **TrackId** is a primary key and (should) guarantee us that each value in that column is unique.

The **tracks** table has several foreign keys in it as well - **AlbumId**, **MediaTypeId**, and **GenreId**.

Relationships between tables

The relationships between the tables are mapped with lines in the diagram. These lines connect a column of one table to a column in another.

Notice the symbol right before the line connects to each table. The symbols with a single “prong” mean that there is one (or at most one) unique values in that column. The multiple pronged symbol means that there each value can appear more than once.

For example, look at the single-pronged symbol from the **media_types** table connected to the multi-pronged symbol at the **tracks** table. This means that for each **MediaTypeId** in the **media_types**, it might be found multiple times in the **tracks** table.

Looking at the relationship in the opposite direction - each **MediaTypeId** in the **tracks** table is found exactly one time in the **media_types** table.

This is called a one-to-many or a many-to-one relationship. Two single-pronged symbols are a one-to-one relationship. Tables can be set up so there are many-to-many relationships, but this is discouraged.

58.4 Preparing the connection

Let's import the `create_engine` function and pass it the location of the database (relative to our current path).

```
[1]: import pandas as pd
from sqlalchemy import create_engine
engine = create_engine('sqlite:///../data/databases/chinook.db')
```

Back to Pandas

We can import an entire table from the database directly as a Pandas DataFrame with the `read_sql` function. Let's import the `tracks` table.

```
[2]: tracks = pd.read_sql('tracks', con=engine)
tracks.head()
```

	TrackId	Name	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice
0	1	For Those About To Rock (We Salute You)	1	1	1	Angus Young, Malcolm Young, Brian Johnson	343719	11170334	0.99
1	2	Balls to the Wall	2	2	1	None	342562	5510424	0.99
2	3	Fast As a Shark	3	2	1	F. Baltes, S. Kaufman, U. Dirksneider & W. Ho...	230619	3990994	0.99
3	4	Restless and Wild	3	2	1	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. D...	252051	4331779	0.99
4	5	Princess of the Dawn	3	2	1	Deaffy & R.A. Smith-Diesel	375418	6290521	0.99

Use raw SQL

Pass `read_sql` an actual sql query as a string.

```
[3]: tracks.columns
```

```
[3]: Index(['TrackId', 'Name', 'AlbumId', 'MediaTypeId', 'GenreId', 'Composer',
       'Milliseconds', 'Bytes', 'UnitPrice'],
       dtype='object')
```

```
[4]: query = """select name, composer, milliseconds
              from tracks
              where milliseconds > 200000 and composer is not null """
long_tracks = pd.read_sql(query, engine)
```

```
long_tracks.head(10)
```

	Name	Composer	Milliseconds
0	For Those About To Rock (We Salute You)	Angus Young, Malcolm Young, Brian Johnson	343719
1	Fast As a Shark	F. Baltes, S. Kaufman, U. Dirksneider & W. Ho...	230619
2	Restless and Wild	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. D...	252051
3	Princess of the Dawn	Deaffy & R.A. Smith-Diesel	375418
4	Put The Finger On You	Angus Young, Malcolm Young, Brian Johnson	205662
5	Let's Get It Up	Angus Young, Malcolm Young, Brian Johnson	233926
6	Inject The Venom	Angus Young, Malcolm Young, Brian Johnson	210834
7	Snowballed	Angus Young, Malcolm Young, Brian Johnson	203102
8	Evil Walks	Angus Young, Malcolm Young, Brian Johnson	263497
9	Breaking The Rules	Angus Young, Malcolm Young, Brian Johnson	263288

58.5 Joining tables in Pandas with `merge`

The `merge` method allows us to join two Pandas DataFrames together based on the values within one or more columns. It follows sql-style logic and allows for inner, left, right, or outer joins.

Getting the media type name in our tracks table

The `tracks` table has a column called `MediaTypeId` but does not directly store the name of this media type in the table itself.

Let's join the `tracks` table with the `media_types` table to get the name of the media along with the track information in a single table.

```
[5]: media_types = pd.read_sql('media_types', engine)
media_types.head()
```

	MediaTypeId	Name
0	1	MPEG audio file
1	2	Protected AAC audio file
2	3	Protected MPEG-4 video file
3	4	Purchased AAC audio file
4	5	AAC audio file

```
[6]: tracks_media = tracks.merge(media_types, on='MediaTypeId')
tracks_media.head()
```

	TrackId	Name_x	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice	Name_y
0	1	For Those About To Rock (We Salute You)		1	1	Angus Young, Malcolm Young, Brian Johnson	343719	11170334	0.99	MPEG audio file
1	6	Put The Finger On You		1	1	Angus Young, Malcolm Young, Brian Johnson	205662	6713451	0.99	MPEG audio file
2	7	Let's Get It Up		1	1	Angus Young, Malcolm Young, Brian Johnson	233926	7636561	0.99	MPEG audio file
3	8	Inject The Venom		1	1	Angus Young, Malcolm Young, Brian Johnson	210834	6852860	0.99	MPEG audio file
4	9	Snowballed		1	1	Angus Young, Malcolm Young, Brian Johnson	203102	6599424	0.99	MPEG audio file

```
[7]: tracks.shape
```

[7]: (3503, 9)

```
[8]: tracks_media.shape
```

[8]: (3503, 10)

Explanation

The `on` parameter is set to the column name (or names) that is used to join the two tables. The column name must appear in both tables. Notice that the resulting table has a single additional column `Name_y`. Even though the `media_types` table had two columns, Pandas keeps only the non-joining columns in the resulting table.

Pandas will append a suffix to any column names that appear in both tables as to differentiate them. You can control the suffix with the `suffixes` parameter like this:

```
[9]: tracks.merge(media_types, on='MediaTypeId', suffixes=('_left', '_right')).head()
```

	TrackId	Name_left	AlbumId	MediaTypeId	GenreId	Composer	Milliseconds	Bytes	UnitPrice	Name_right
0	1	For Those About To Rock (We Salute You)	1	1	1	Angus Young, Malcolm Young, Brian Johnson	343719	11170334	0.99	MPEG audio file
1	6	Put The Finger On You	1	1	1	Angus Young, Malcolm Young, Brian Johnson	205662	6713451	0.99	MPEG audio file
2	7	Let's Get It Up	1	1	1	Angus Young, Malcolm Young, Brian Johnson	233926	7636561	0.99	MPEG audio file
3	8	Inject The Venom	1	1	1	Angus Young, Malcolm Young, Brian Johnson	210834	6852860	0.99	MPEG audio file
4	9	Snowballed	1	1	1	Angus Young, Malcolm Young, Brian Johnson	203102	6599424	0.99	MPEG audio file

Different column names when joining

If the column names for the joining tables are not the same, use the `left_on` and `right_on` parameters to specify their names explicitly. For instance, let's change the joining column in the `tracks` table.

```
[10]: tracks2 = tracks.rename(columns={'MediaTypeId': 'MTID'})
tracks2.head()
```

	TrackId	Name	AlbumId	MTID	GenreId	Composer	Milliseconds	Bytes	UnitPrice
0	1	For Those About To Rock (We Salute You)	1	1	1	Angus Young, Malcolm Young, Brian Johnson	343719	11170334	0.99
1	2	Balls to the Wall	2	2	1	None	342562	5510424	0.99
2	3	Fast As a Shark	3	2	1	F. Baltes, S. Kaufman, U. Dirksneider & W. Ho...	230619	3990994	0.99
3	4	Restless and Wild	3	2	1	F. Baltes, R.A. Smith-Diesel, S. Kaufman, U. D...	252051	4331779	0.99
4	5	Princess of the Dawn	3	2	1	Deaffy & R.A. Smith-Diesel	375418	6290521	0.99

```
[11]: tracks2.merge(media_types, left_on='MTID', right_on='MediaTypeId').head()
```

	TrackId	Name_x	AlbumId	MTID	GenreId	...	Milliseconds	Bytes	UnitPrice	MediaTypeId	Name_y
0	1	For Those About To Rock (We Salute You)	1	1	1	...	343719	11170334	0.99	1	MPEG audio file
1	6	Put The Finger On You	1	1	1	...	205662	6713451	0.99	1	MPEG audio file
2	7	Let's Get It Up	1	1	1	...	233926	7636561	0.99	1	MPEG audio file
3	8	Inject The Venom	1	1	1	...	210834	6852860	0.99	1	MPEG audio file
4	9	Snowballed	1	1	1	...	203102	6599424	0.99	1	MPEG audio file

58.6 Exercises

Read the tables into Pandas to answer the questions. Do not answer them with raw sql statements

Exercise 1

How many media types does each track have? Answer this by looking at the data diagram and then programmatically.

```
[ ]:
```

Exercise 2

Which track has sold the most copies?

[]:

Exercise 3

Which playlist has the most tracks?

[]:

Exercise 4

Which playlist, that has at least 15 tracks has on average the most expensive tracks?

[]:

Exercise 5

Find the most sold genre per country.

[]:

Exercise 6

Find the name and email of each employee's boss. Make use of the suffix arguments to better label the merged data. Be sure to include employees that don't have bosses. This is called a recursive relationship.

[]:

Exercise 7

Which artists have the longest tracks on average? Return answer in minutes.

[]:

Chapter 59

Data Normalization

Let's take a look at few rows from some visits to a doctor in the `tidy/doctor_visits.csv` file.

```
[1]: import pandas as pd
      import numpy as np
      dv = pd.read_csv('../data/tidy/doctor_visits.csv')
      dv
```

	patient_name	visit_date	patient_address	patient_birthdate	clinic_name	...	cost	doctor_name	doctor_specialty	procedure_name	procedure_code
0	Penelope	5/3/2014	123 Fake Street	4/1/2013	St. Clair	...	100	Smith	General Practitioner	Physical	1
1	Abbas	6/1/2014	105 Real Street	3/14/1947	Younge	...	50	Patel	Radiologist	X-ray	20
2	Eleni	6/3/2014	50 Rosegarden	8/9/1981	Main	...	200	Smith	General Practitioner	Physical	1
3	Abbas	6/10/2015	105 Real Street	3/14/1947	Starsky	...	35	Patel	Radiologist	CT Scan	24
4	Penelope	7/2/2016	123 Fake Street	4/1/2013	Younge	...	44	Brown	Cardiologist	Echocardiogram	31
5	Eleni	3/3/2017	50 Rosegarden	8/9/1981	Younge	...	300	Brown	Cardiologist	Stress Test	36

The information is easy to read

All the information presented in this table is easy to read. Any question that is asked about the visits can be quickly answered.

Yet, something is wrong, what is it?

Although the table of data is sufficient to address our questions, it repeats much of the data and will not scale well as more patients come into the system. It will also be difficult to update historical data, such as if the patient's address changes or a clinic changes names.

A short intro into data normalization

Modern databases attempt to reduce the amount of replication in the data by a process called **normalization**. It involves separating data into tables to minimize replication and increase data accuracy.

From [wikipedia](#): > Database Normalization, or simply normalization, is the process of organizing the columns (attributes) and tables (relations) of a relational database to reduce data redundancy and improve data integrity. Normalization is also the process of simplifying the design of a database so that it achieves the optimum structure. It reduces and eliminates redundant data. In normalization, data integrity is assured. It was first proposed by Dr. Edgar F. Codd, as an integral part of a relational model.

There is much, much more to data normalization. The following example is just a brief overview.

Which values are always the same for each visit?

When looking through the table, you should notice that certain values will repeat for every single visit. For instance, the patient name, address, and birth date are going to be the same. There isn't a need to keep repeating all these values for every visit.

Separating patient values into a distinct table

Let's select all the patient columns in a single DataFrame. The `copy` method is called to ensure that this is new DataFrame is a completely different DataFrame and not referring to the same data as the original.

```
[2]: patient = dv[['patient_name', 'patient_address', 'patient_birthdate']]
      patient
```

	patient_name	patient_address	patient_birthdate
0	Penelope	123 Fake Street	4/1/2013
1	Abbas	105 Real Street	3/14/1947
2	Eleni	50 Rosegarden	8/9/1981
3	Abbas	105 Real Street	3/14/1947
4	Penelope	123 Fake Street	4/1/2013
5	Eleni	50 Rosegarden	8/9/1981

Drop duplicate rows

There is no need to store duplicate values of each row. Let's keep only the unique rows.

```
[3]: patient = patient.drop_duplicates()
      patient
```

	patient_name	patient_address	patient_birthdate
0	Penelope	123 Fake Street	4/1/2013
1	Abbas	105 Real Street	3/14/1947
2	Eleni	50 Rosegarden	8/9/1981

59.1 Create a primary key to uniquely identify each row

When we normalize our data, and separate it into new tables, an additional column is added to the table to uniquely identify each row. The unique value that identifies each row is called a **primary key**. Let's add a primary key to the patient table. Note: If you get the `SettingWithCopyWarning`, ignore it.

```
[4]: patient['patient_id'] = np.arange(len(patient))
      patient
```

```
/Users/Ted/miniconda3/envs/minimal_ds/lib/python3.7/site-
packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
```

Try using `.loc[row_indexer,col_indexer] = value` instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

"""Entry point for launching an IPython kernel.

	patient_name	patient_address	patient_birthdate	patient_id
0	Penelope	123 Fake Street	4/1/2013	0
1	Abbas	105 Real Street	3/14/1947	1
2	Eleni	50 Rosegarden	8/9/1981	2

Rearrange the columns so that `patient_id` is first

It's best to always put the primary key as the first column.

```
[5]: cols = ['patient_id', 'patient_name', 'patient_address', 'patient_birthdate']
patient = patient[cols]
patient
```

	patient_id	patient_name	patient_address	patient_birthdate
0	0	Penelope	123 Fake Street	4/1/2013
1	1	Abbas	105 Real Street	3/14/1947
2	2	Eleni	50 Rosegarden	8/9/1981

59.2 We just created a dimension

In database terminology, the patient table would be considered a **dimension**. Dimensions are things that exist in your data that are independent of any event taking place. They tend to be static and do not change often (such as your name or birth date).

Create tables for all the other dimensions

There are several other dimensions in our original table. Let's create new dimension tables for each one of the following:

- clinic
- doctor
- procedure

We will select each of the columns unique to each dimension, drop the replicated rows and add a primary key as the first column to uniquely identify each row.

Clinic Dimension

```
[6]: cols = ['clinic_name', 'clinic_address']
clinic = dv[cols].drop_duplicates()
clinic
```

	clinic_name	clinic_address
0	St. Clair	501 Downtown Ave
1	Younge	222 Fortnite Dr
2	Main	21 Jayne St
3	Starsky	505 Perry Rd

```
[7]: clinic['clinic_id'] = np.arange(len(clinic))
      clinic
```

	clinic_name	clinic_address	clinic_id
0	St. Clair	501 Downtown Ave	0
1	Younge	222 Fortnite Dr	1
2	Main	21 Jayne St	2
3	Starsky	505 Perry Rd	3

```
[8]: cols = ['clinic_id', 'clinic_name', 'clinic_address']
      clinic = clinic[cols]
      clinic.head()
```

	clinic_id	clinic_name	clinic_address
0	0	St. Clair	501 Downtown Ave
1	1	Younge	222 Fortnite Dr
2	2	Main	21 Jayne St
3	3	Starsky	505 Perry Rd

Doctor Dimension

```
[9]: cols = ['doctor_name', 'doctor_specialty']
      doctor = dv[cols].drop_duplicates()
      doctor
```

	doctor_name	doctor_specialty
0	Smith	General Practitioner
1	Patel	Radiologist
4	Brown	Cardiologist

```
[10]: doctor['doctor_id'] = np.arange(len(doctor))
       doctor
```

	doctor_name	doctor_specialty	doctor_id
0	Smith	General Practitioner	0
1	Patel	Radiologist	1
4	Brown	Cardiologist	2

```
[11]: cols = ['doctor_id', 'doctor_name', 'doctor_specialty']
doctor = doctor[cols]
doctor.head()
```

	doctor_id	doctor_name	doctor_specialty
0	0	Smith	General Practitioner
1	1	Patel	Radiologist
4	2	Brown	Cardiologist

Procedure Dimension

Here, the primary key is already given to us with the procedure code. We will keep it as is.

```
[12]: cols = ['procedure_code', 'procedure_name']
procedure = dv[cols].drop_duplicates()
procedure
```

	procedure_code	procedure_name
0	1	Physical
1	20	X-ray
3	24	CT Scan
4	31	Echocardiogram
5	36	Stress Test

59.3 Replacing original data with primary keys

We can now revisit our original DataFrame and replace all columns in each dimension with a single column, the primary key of that dimension.

Join original table to dimension tables

To make the replacement, we will join our original table to each one of our dimension tables. The `merge` method joins tables together in Pandas. We can specify how the tables will join with the `on` parameter. We will join on all the non-primary key columns. Below, we join the `patient` table. The result is one extra column at the end of the DataFrame, the `patient_id`.

```
[13]: dv_fact = dv.merge(patient, on=['patient_name', 'patient_address',
                                    'patient_birthdate'])
dv_fact
```

	patient_name	visit_date	patient_address	patient_birthdate	clinic_name	...	doctor_name	doctor_specialty	procedure_name	procedure_code	patient_id
0	Penelope	5/3/2014	123 Fake Street	4/1/2013	St. Clair	...	Smith	General Practitioner	Physical	1	0
1	Penelope	7/2/2016	123 Fake Street	4/1/2013	Younge	...	Brown	Cardiologist	Echocardiogram	31	0
2	Abbas	6/1/2014	105 Real Street	3/14/1947	Younge	...	Patel	Radiologist	X-ray	20	1
3	Abbas	6/10/2015	105 Real Street	3/14/1947	Starsky	...	Patel	Radiologist	CT Scan	24	1
4	Eleni	6/3/2014	50 Rosegarden	8/9/1981	Main	...	Smith	General Practitioner	Physical	1	2
5	Eleni	3/3/2017	50 Rosegarden	8/9/1981	Younge	...	Brown	Cardiologist	Stress Test	36	2

Drop the dimension columns

We can now drop all the original patient columns as the `patient_id` now refers to them.

```
[14]: dv_fact = dv_fact.drop(columns=['patient_name', 'patient_address',
                                    'patient_birthdate'])
dv_fact
```

	visit_date	clinic_name	clinic_address	cost	doctor_name	doctor_specialty	procedure_name	procedure_code	patient_id
0	5/3/2014	St. Clair	501 Downtown Ave	100	Smith	General Practitioner	Physical	1	0
1	7/2/2016	Younge	222 Fortnite Dr	44	Brown	Cardiologist	Echocardiogram	31	0
2	6/1/2014	Younge	222 Fortnite Dr	50	Patel	Radiologist	X-ray	20	1
3	6/10/2015	Starsky	505 Perry Rd	35	Patel	Radiologist	CT Scan	24	1
4	6/3/2014	Main	21 Jayne St	200	Smith	General Practitioner	Physical	1	2
5	3/3/2017	Younge	222 Fortnite Dr	300	Brown	Cardiologist	Stress Test	36	2

59.4 Replace all the other dimensions with primary key columns

Doctor Dimension

```
[15]: dv_fact = dv_fact.merge(doctor, on=['doctor_name', 'doctor_specialty'])
dv_fact = dv_fact.drop(columns=['doctor_name', 'doctor_specialty'])
dv_fact
```

	visit_date	clinic_name	clinic_address	cost	procedure_name	procedure_code	patient_id	doctor_id
0	5/3/2014	St. Clair	501 Downtown Ave	100	Physical		1	0
1	6/3/2014	Main	21 Jayne St	200	Physical		1	2
2	7/2/2016	Younge	222 Fortnite Dr	44	Echocardiogram		31	0
3	3/3/2017	Younge	222 Fortnite Dr	300	Stress Test		36	2
4	6/1/2014	Younge	222 Fortnite Dr	50	X-ray		20	1
5	6/10/2015	Starsky	505 Perry Rd	35	CT Scan		24	1

[16]: clinic

	clinic_id	clinic_name	clinic_address
0	0	St. Clair	501 Downtown Ave
1	1	Younge	222 Fortnite Dr
2	2	Main	21 Jayne St
3	3	Starsky	505 Perry Rd

Clinic Dimension

```
[17]: dv_fact = dv_fact.merge(clinic, on=['clinic_name', 'clinic_address'])
dv_fact = dv_fact.drop(columns=['clinic_name', 'clinic_address'])
dv_fact
```

	visit_date	cost	procedure_name	procedure_code	patient_id	doctor_id	clinic_id
0	5/3/2014	100	Physical		1	0	0
1	6/3/2014	200	Physical		1	2	0
2	7/2/2016	44	Echocardiogram		31	0	2
3	3/3/2017	300	Stress Test		36	2	1
4	6/1/2014	50	X-ray		20	1	1
5	6/10/2015	35	CT Scan		24	1	3

Procedure Dimension

Since the primary key is already in the table, we can just drop the `procedure_name` column

```
[18]: dv_fact = dv_fact.drop(columns=['procedure_name'])
dv_fact
```

	visit_date	cost	procedure_code	patient_id	doctor_id	clinic_id
0	5/3/2014	100		1	0	0
1	6/3/2014	200		1	2	0
2	7/2/2016	44		31	0	2
3	3/3/2017	300		36	2	1
4	6/1/2014	50		20	1	1
5	6/10/2015	35		24	1	3

Rearrange columns with foreign keys first

When a primary key from table is found in another table, it is called a **foreign key**. Foreign keys can repeat in the table they are in. Primary keys, however, can never repeat in the tables they are in. This is a very important property. Foreign keys are

```
[19]: cols = ['patient_id', 'clinic_id', 'doctor_id',
            'procedure_code', 'visit_date', 'cost']
dv_fact = dv_fact[cols]
dv_fact
```

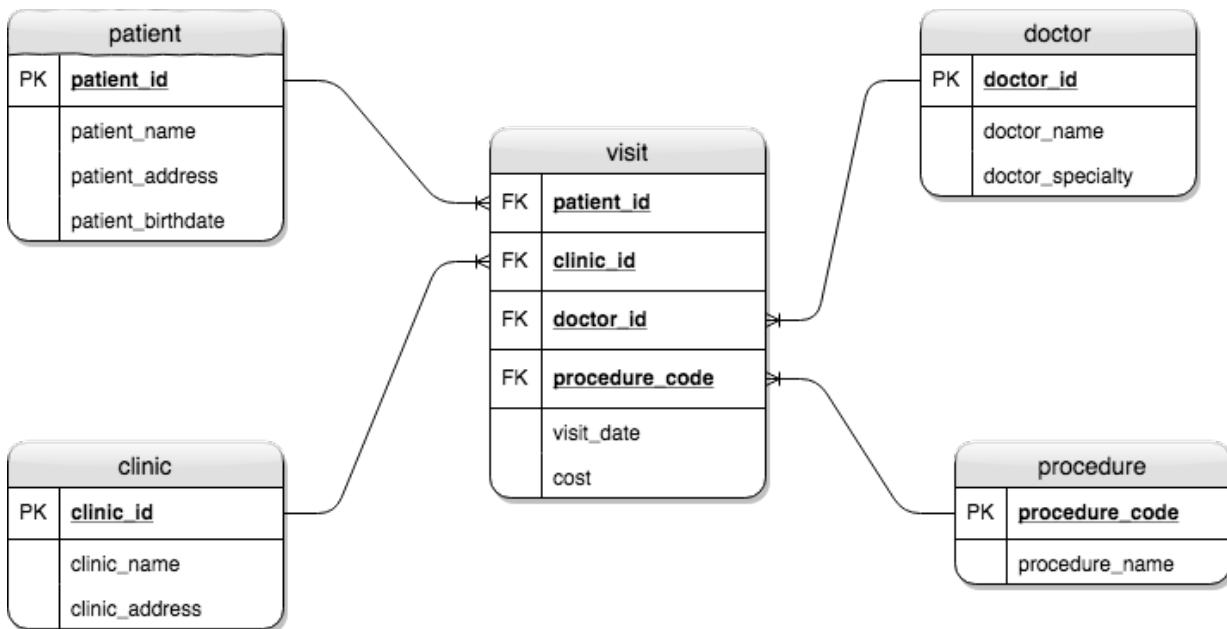
	patient_id	clinic_id	doctor_id	procedure_code	visit_date	cost
0	0	0	0		1	5/3/2014
1	2	2	0		1	6/3/2014
2	0	1	2		31	7/2/2016
3	2	1	2		36	3/3/2017
4	1	1	1		20	6/1/2014
5	1	3	1		24	6/10/2015

59.5 Fact Table

This last DataFrame, `dv_fact` is called a **fact table** using database terminology. Fact tables hold the actual **events** or **transactions** that take place in a business. They hold the columns that are subject to change such as date and cost here. If we had data from a grocery store, our fact table would have columns like the number of items purchased, the cost of each item, and the type of payment used. Fact tables have references to the static dimension tables through foreign keys.

59.6 Data Model Diagram

The diagram of the **data model** or **entity-relationship diagram** is presented below. Data models show the logical relationships between the fact and dimension tables. This type of data model is called a **star schema** and one of the simplest designs.



See [this simple Stack Overflow answer](#) for another description of fact and dimension tables.

59.7 Exercises

Exercise 1

Tidy the dataset `tidy/store_transactions.csv`.

[]:

Part XI

Visualization with Matplotlib

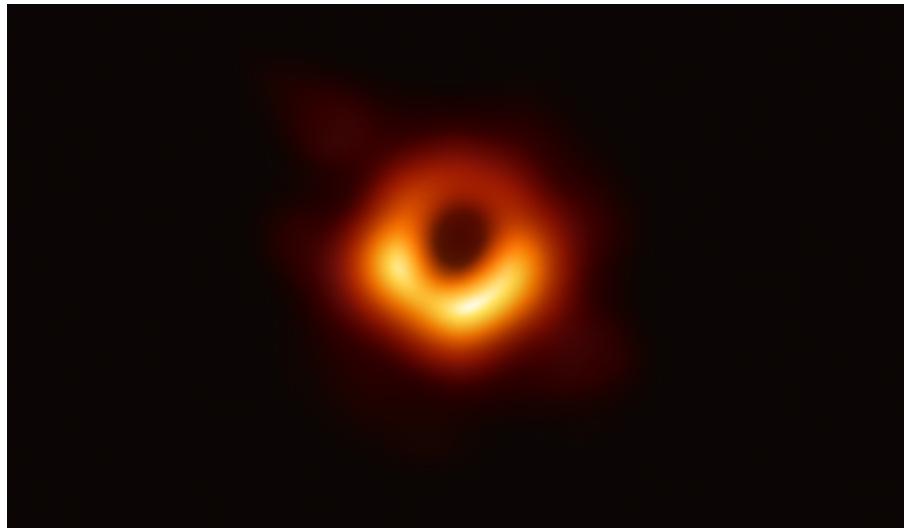
Chapter 60

Introduction to matplotlib

matplotlib is a popular data visualization library in Python that can produce nearly any kind of plot imaginable. It is often used in scientific publications and helped create one of the first ever images of a black hole (below) in April, 2019.

As you have seen a few times in this book, pandas can create plots, so you might be wondering why we need to learn about a completely separate library. All of the plots created with pandas are done internally with matplotlib. Whenever you create a plot with pandas, a matplotlib plotting object is returned. pandas can only create a fraction of the plots available to matplotlib.

The major benefit of using pandas for visualization is that the plots are easier to create. The trade-off is that you won't have as much control as you do with matplotlib. We will eventually cover the pandas (and the seaborn) library for visualization in great detail, but will begin with matplotlib.



60.1 Two interfaces of matplotlib

matplotlib was originally created by the late John Hunter in the early 2000's to mimic the plotting functionality of [Matlab](#), a popular scientific computing software application. In essence, it is a "matlab-like plotting library". One issue with porting a library from one language to another is that the idioms and practices that each language have are not often compatible. Python is usually programmed differently than matlab.

Over the course of matplotlib's development, two separate ways to interface with matplotlib evolved. They are the **state-machine environment** and the **object-oriented** approach. The state-machine environment (known as **pyplot** from here on out) implicitly handles some of the plotting for you.

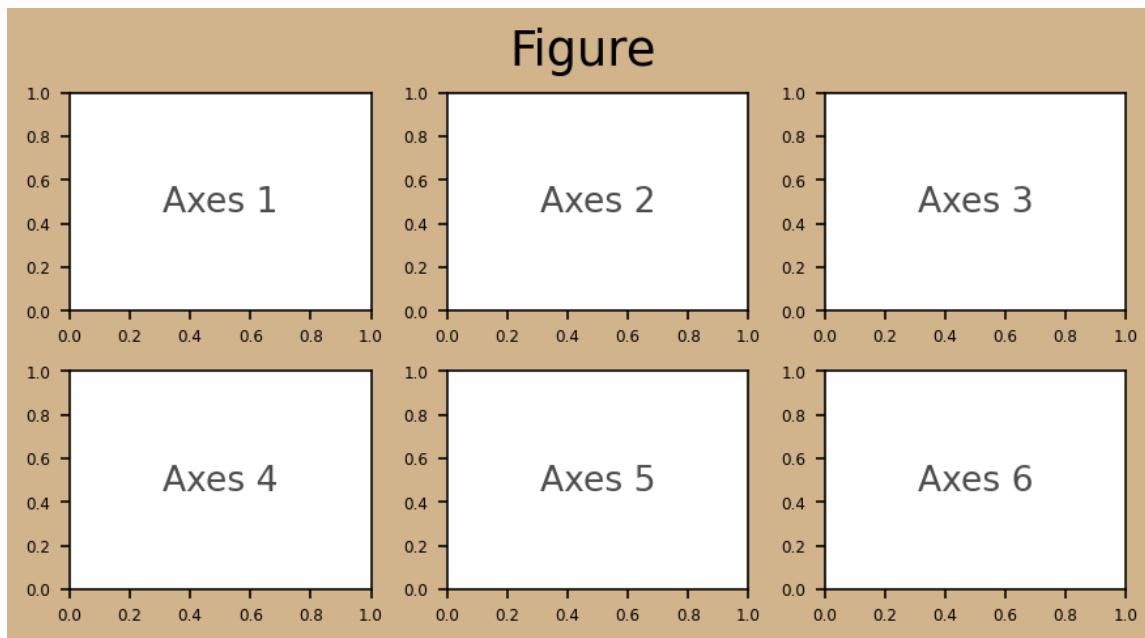
The **object-oriented** approach gives you full control over each element of the plot and fits the style of how Python is usually developed. Most plots can be reproduced with either interface, but the object-oriented approach is explicit, and in my opinion, easier to determine what is happening.

Using only the object-oriented approach

The chapters on matplotlib only use the object-oriented approach, as attempting to learn both at the start is not necessary and confusing. Many tutorials online use pyplot, so it is something that you might need to understand. Thankfully, much of the code between each approach looks quite similar when making a single plot.

60.2 Figure - Axes Hierarchy

There is an important hierarchy that must be understood when working with matplotlib. The highest and outermost part of a plot is the **figure**, which contains all of the other plotting elements. Typically, you do not interact with it much. Inside the figure are the **axes**. This is the actual plotting surface that you normally would refer to as a ‘plot’.



A figure may contain any number of axes. An axes is a container for most of the plotting objects that get drawn onto your screen. This includes the x and y axis, lines, text, points, legends, images, and others.

Axes is a confusing word

The term **axes** is not actually plural and does not mean more than one axis. It literally stands for a single ‘plot’. It’s unfortunate that this fundamental element has a name that is so confusing. I usually pronounce it “axeez” when I am teaching to help differentiate it from the word ‘axis’.

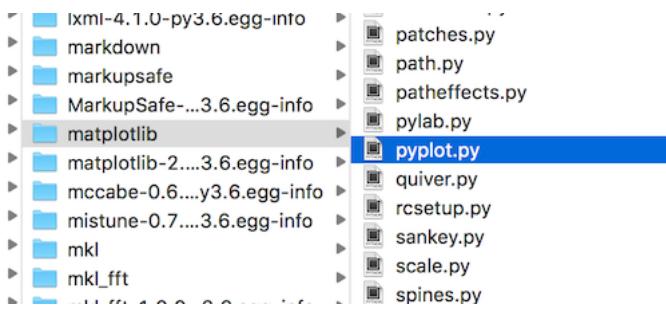
Importing the pyplot module

Importing matplotlib into your workspace is done a little differently than numpy or pandas. You rarely will import matplotlib itself directly like this:

```
import matplotlib
```

The above is perfectly valid code, but the matplotlib developers decided not to put all the main functionality in the top level module. When you `import pandas as pd`, you get access to nearly all of the available

functions and classes of the pandas library. This isn't true with matplotlib. Instead, much of the functionality for quickly plotting is found in the `pyplot` module. If you navigate to the matplotlib source directory found in your site-packages directory, you will see a `pyplot.py` file. This is the module that we want to import into our workspace.



Let's import the `pyplot` module now and alias it as `plt`, which is done by convention.

```
[1]: import matplotlib.pyplot as plt
```

Use `pyplot` to begin

`pyplot` does provide lots of useful functions, one of which creates the figure and any number of axes that you desire. You can do this without `pyplot`, but it involves more syntax. It's also quite standard to begin the object-oriented approach by laying out your figure and axes first with `pyplot` and then proceed by calling methods from these objects.

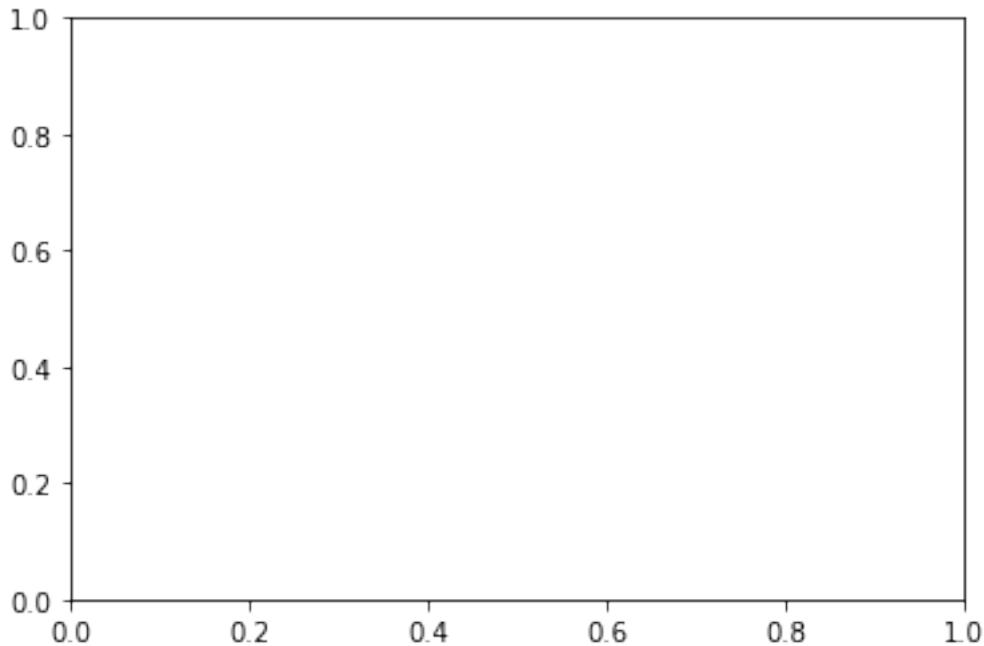
Use the `subplots` function

The `pyplot subplots` function creates a single figure and any number of axes. If you call it with the default arguments, a single axes is created within a figure.

Unpack the `subplots` tuple

The `subplots` function returns a two-item tuple containing the figure and the axes. We unpack each of these objects as their own variable.

```
[2]: fig, ax = plt.subplots()
```



Verify the returned types of the `subplots` function

Let's verify that we indeed have a figure and axes.

```
[3]: type(fig)
```

```
[3]: matplotlib.figure.Figure
```

```
[4]: type(ax)
```

```
[4]: matplotlib.axes._subplots.AxesSubplot
```

Distinguishing the figure from the axes

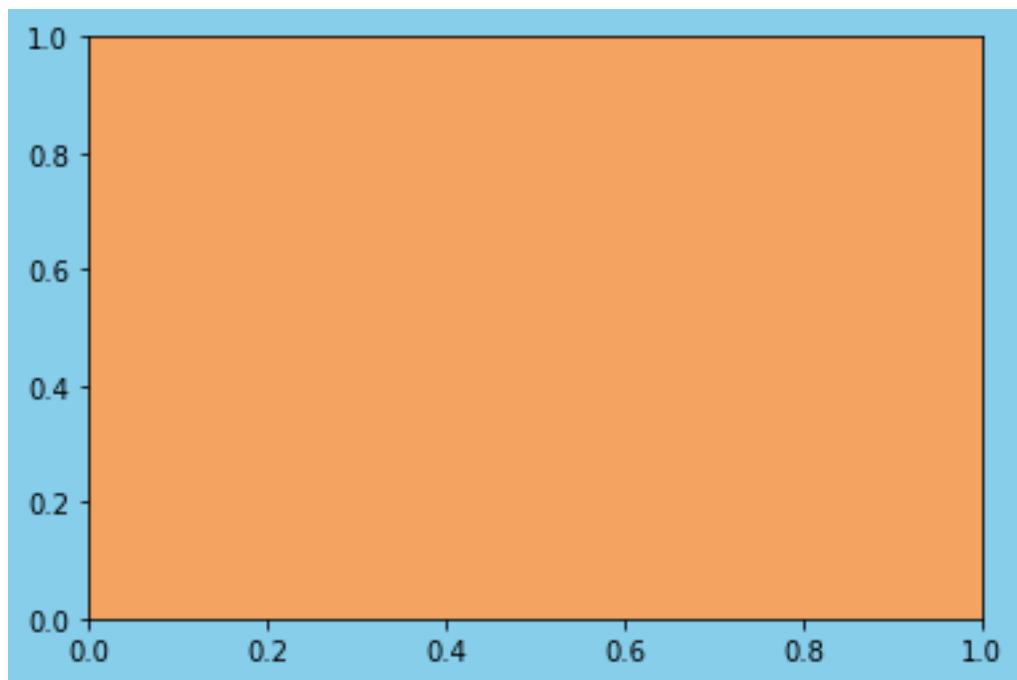
It's not obvious from looking at the image which part is the figure and which is the axes. To help distinguish the figure from the axes, we will set the 'facecolor' (surface) of each to a different color. Both objects have a `set_facecolor` method, which will be passed a named color called in an object-oriented fashion. Colors will be covered in greater detail in an upcoming chapter.

```
[5]: fig.set_facecolor('skyblue')
ax.set_facecolor('sandybrown')
```

Where is the figure?

When using the object-oriented approach, you need to place the figure variable name as the last line in a cell to view it in the notebook. This should now hopefully distinguish the figure from the axes.

```
[6]: fig
```



Why is there no assignment statement?

Notice, that the two calls above to the `set_facecolor` method were made without an assignment statement. Both of these operations happened **in-place**. The calling figure and axes objects were updated without a new one getting created.

Can only view the entire figure not the axes

You cannot view an axes independent of a figure. Running a cell with the `axes` variable name as the last line will simply output a default representation of the object. You can only view a figure in the notebook.

```
[7]: ax
```

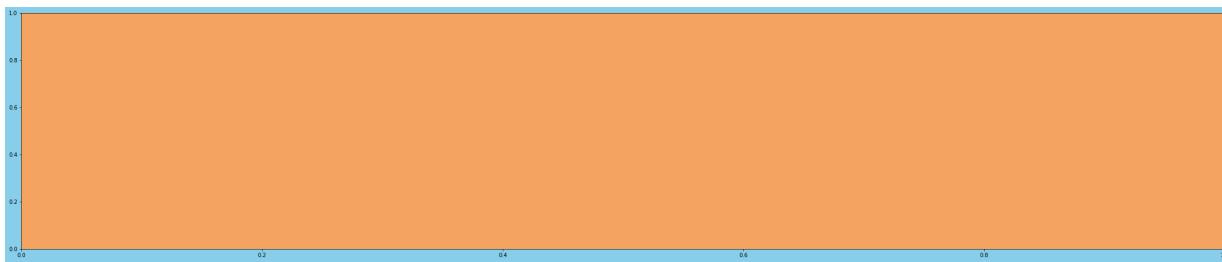
```
[7]: <matplotlib.axes._subplots.AxesSubplot at 0x11b7ab690>
```

60.3 Setting the size of the figure upon creation

By default, all figures have a width of 6 inches and a height of 4 inches. If you are working in a Jupyter Notebook, you'll probably notice that the size of the figure on your screen isn't actually 6 inches by 4 inches. A deeper discussion on what these "inches" really mean can be found in the upcoming chapter, "Matplotlib Resolution". For now, think of these two numbers as the relative width and height of the figure.

We can change the dimensions of the figure when creating it by setting the `figsize` parameter as a two-item tuple of the width and height of the figure. Below, we create a figure with a width of 40 inches and height of 8 inches. Notebooks will always scale down the figure so that it fits in the output area.

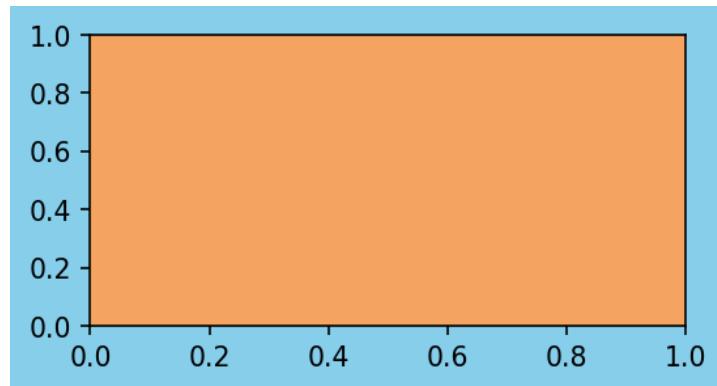
```
[8]: fig, ax = plt.subplots(figsize=(40, 8))
fig.set_facecolor('skyblue')
ax.set_facecolor('sandybrown')
```



Let's create one more figure and axes with dimensions of 4 by 2 and a dots-per-inch (DPI) of 147. The DPI is the number of pixels created per inch and will be discussed in detail in the 'Matplotlib Resolution' chapter. For now, think it of as increasing the sharpness of the image.

You can also set the face color for the figure in the `subplots` function. Notice that the tick labels of the image below are much larger than the image above. Each tick label has a default font size that does not depend on the width of the image. The tick labels above appear very small above because they are relative to a figure that has a width of 40. These same tick labels appear larger in the plot below because the width is 4.

```
[9]: fig, ax = plt.subplots(figsize=(4, 2), facecolor='skyblue', dpi=147)
ax.set_facecolor('sandybrown')
```



Began the Object-Oriented Approach

Both calls to the `set_facecolor` method demonstrated the object-oriented approach to matplotlib. With this approach, every plotting object that is created may be manipulated by calling its methods.

As we will see, everything on our plot is a separate object. Each axis, tick mark, tick label, axis label, plot title, line, and many others are separate objects. Each of these objects may be explicitly referenced and assigned to a variable name. Once we have a reference to a particular object, we can then modify it by calling its methods. Thus far we have two references, `fig` and `ax`.

60.4 Axes methods

Even though we've called both figure and axes methods, it is far more common to call axes methods. It is the axes that contains most of the plotting methods and is the object you will interact with most frequently. The figure is analogous to the frame of a picture. It plays a role, but isn't the main attraction. The bulk of the visualization commands will come from the axes. We'll now begin our exploration of the many axes methods.

Getter and setter methods

Many axes methods begin with either `get_` or `set_` followed by the part of the axes that will get retrieved or modified. These kinds of methods are often referred to as ‘getter’ and ‘setter’ methods. The following list shows several of the most common properties that can be set on our axes. We will see examples of each one below.

- `title`
- `xlabel/ylabel`
- `xlim/ylim`
- `xticks/yticks`
- `xticklabels/yticklabels`

Getting and setting the title of the axes

The `get_title` method returns the title of the axes as a string. There is no title at this moment, so an empty string is returned.

```
[10]: ax.get_title()
```

```
[10]: ''
```

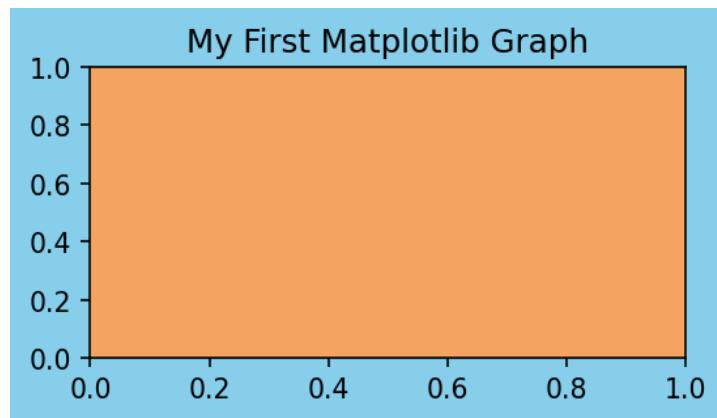
The `set_title` method places a centered title on our Axes when passing it a string. Notice that a matplotlib `Text` object has been returned in the output area. This will be discussed later.

```
[11]: ax.set_title('My First Matplotlib Graph')
```

```
[11]: Text(0.5, 1.0, 'My First Matplotlib Graph')
```

Again, the figure variable name must be placed as the last line in a cell to show in the notebook.

```
[12]: fig
```



Running the `get_title` method again returns the string that was just set as the title.

```
[13]: ax.get_title()
```

```
[13]: 'My First Matplotlib Graph'
```

Getting and setting the labels for the x and y axis

The x and y axis can each be labeled with a single string. By default, there are no x and y axis labels and using their getter methods returns an empty string.

```
[14]: ax.get_xlabel()
```

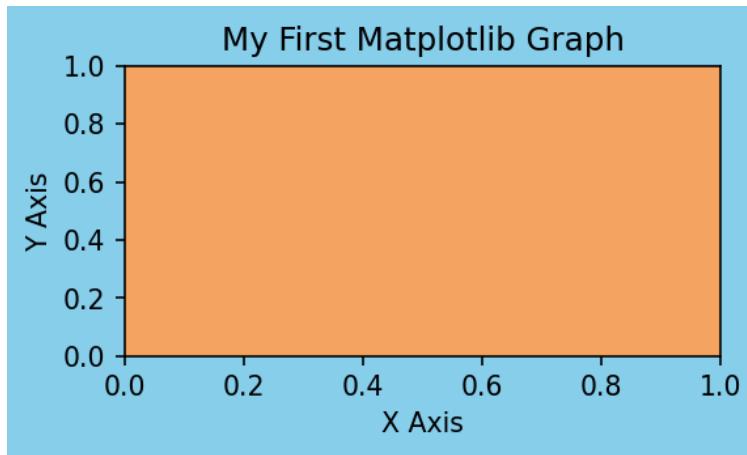
```
[14]: ''
```

```
[15]: ax.get_ylabel()
```

```
[15]: ''
```

We can provide labels for both the x and y axis using the `set_xlabel` and `set_ylabel` commands. We set both labels in the same cell and output the figure.

```
[16]: ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')
fig
```



Let's verify that the getter axis labels work.

```
[17]: ax.get_xlabel()
```

```
[17]: 'X Axis'
```

```
[18]: ax.get_ylabel()
```

```
[18]: 'Y Axis'
```

Getting and setting the x and y limits

By default, the limits of both the x and y axis begin at 0 and end at 1. Let's verify this with the `get_xlim` and `get_ylim` methods, which return a tuple of the limits.

```
[19]: ax.get_xlim()
```

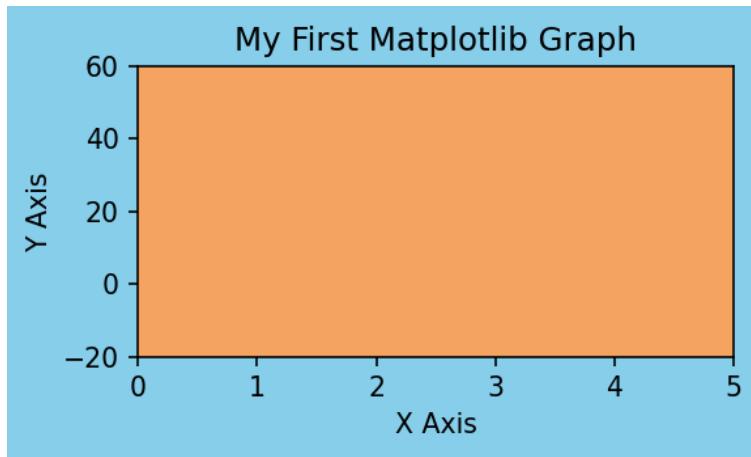
```
[19]: (0.0, 1.0)
```

```
[20]: ax.get_ylim()
```

[20]: (0.0, 1.0)

We can change these limits with the `set_xlim` and `set_ylim` methods by passing them a new lower and upper boundary to change the limits. Notice that the size of the figure remains the same. Only the limits of the x and y axis have changed.

```
[21]: ax.set_xlim(0, 5)
ax.set_ylim(-20, 60)
fig
```



Getting and setting the location of the x and y ticks

In the graph above, ticks are placed every 1 unit along the x-axis and every 20 units along the y-axis. matplotlib chooses reasonable default locations for the ticks. Retrieve the location of these ticks with the `get_xticks` and `get_yticks` methods.

[22]: `ax.get_xticks()`

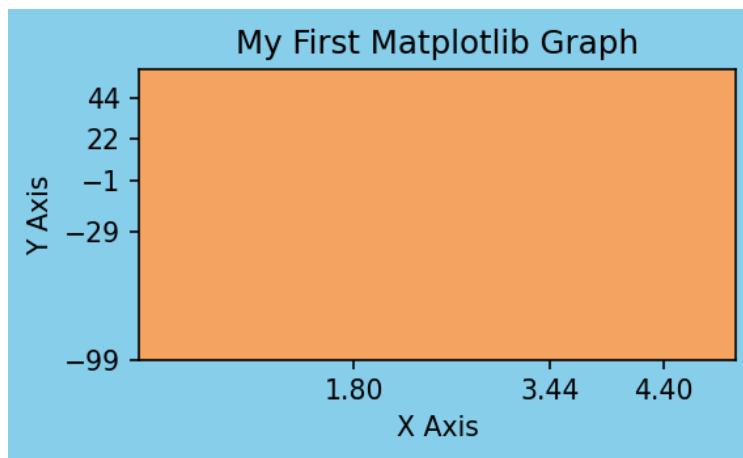
[22]: `array([0., 1., 2., 3., 4., 5.])`

[23]: `ax.get_yticks()`

[23]: `array([-20., 0., 20., 40., 60.])`

The tick locations are returned as numpy arrays. We can specify the exact location of the x and y ticks with the `set_xticks` and `set_yticks` methods. We pass them a list of numbers indicating where we want the ticks. When we set the y-ticks we use a number outside of the current bounds of the axis. This forces matplotlib to change the limits.

```
[24]: ax.set_xticks([1.8, 3.44, 4.4])
ax.set_yticks([-99, -29, -1, 22, 44])
fig
```



Let's verify that the y-axis limits have indeed changed and that the x-axis limits have not.

```
[25]: ax.get_xlim()
```

```
[25]: (0.0, 5.0)
```

```
[26]: ax.get_ylim()
```

```
[26]: (-99.0, 60.0)
```

Getting and setting the x and y tick labels

matplotlib has separate objects for the tick labels, which are the values printed directly below each tick location. The current tick labels for the x-axis and y-axis are the same as the tick locations. Let's attempt to view them with the `get_xticklabels` method.

```
[27]: ax.get_xticklabels()
```

```
[27]: <a list of 3 Text major ticklabel objects>
```

A list-like object is returned, but each item is not visible in the notebook. To convert it to a list, pass it to the `list` constructor which returns three matplotlib text objects.

```
[28]: list(ax.get_xticklabels())
```

```
[28]: [Text(1.8, 0, '1.80'), Text(3.44, 0, '3.44'), Text(4.4, 0, '4.40')]
```

The individual tick labels can now be seen. Let's see the y tick labels.

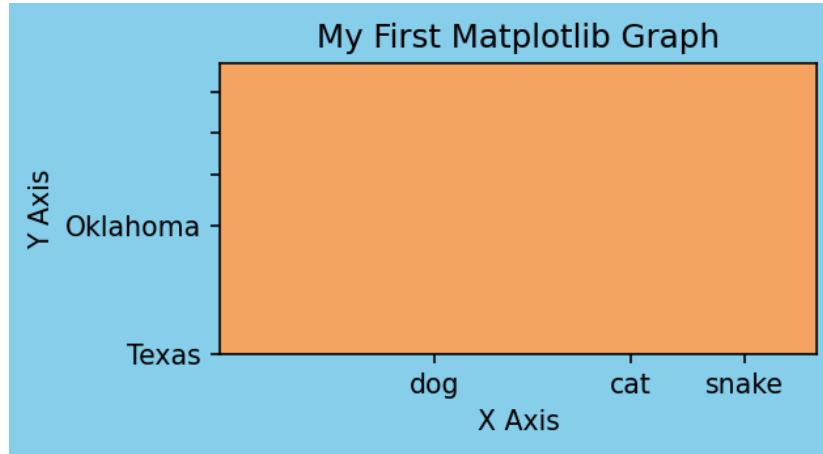
```
[29]: list(ax.get_yticklabels())
```

```
[29]: [Text(0, -99, '-99'),
      Text(0, -29, '-29'),
      Text(0, -1, '-1'),
      Text(0, 22, '22'),
      Text(0, 44, '44')]
```

Pass the `set_xticklabels` and `set_yticklabels` methods a list of values (which can be strings) to use as the new labels. You actually do not need to provide labels for all of the ticks. matplotlib will simply make

those other tick labels equal to the empty string. We provide just two of the five y tick labels. Notice that the tick **lines** are still present, but there are no labels for the remaining three.

```
[30]: ax.set_xticklabels(['dog', 'cat', 'snake'])
ax.set_yticklabels(['Texas', 'Oklahoma'])
fig
```



Get the x and y tick labels again.

```
[31]: list(ax.get_xticklabels())
```

```
[31]: [Text(1.8, 0, 'dog'), Text(3.44, 0, 'cat'), Text(4.4, 0, 'snake')]
```

```
[32]: list(ax.get_yticklabels())
```

```
[32]: [Text(0, -99, 'Texas'),
      Text(0, -29, 'Oklahoma'),
      Text(0, -1, ''),
      Text(0, 22, ''),
      Text(0, 44, '')]
```

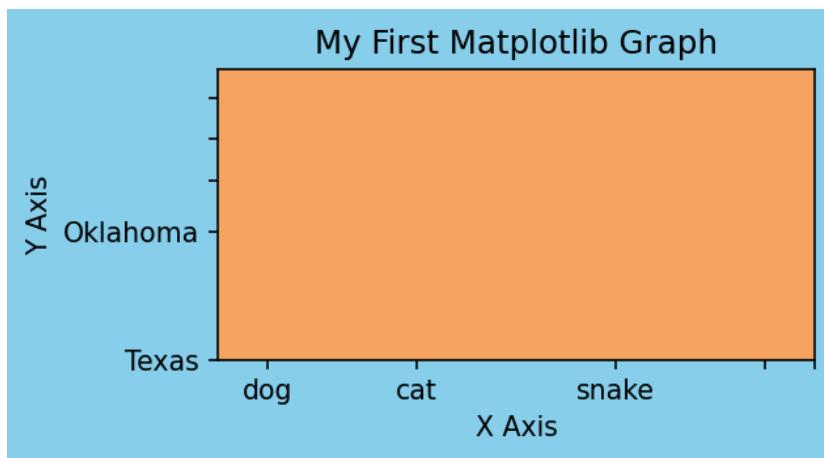
The difference between the tick locations and labels

The tick locations are a completely separate concept than the tick labels. The tick locations are always numeric and determine where on the plot the tick marks appear. The tick labels, on the other hand, are the strings that are used on the graph. By default, the tick labels are a string of the tick location, but you can set them to be any value you want, as we did above.

Set the tick locations again

When we set the tick locations for the x-axis with the `set_xticks` method, the labels were set as the new tick locations. Below, we set five new tick locations for the x-axis. In this instance, the tick labels are kept as the strings above ('dog', 'cat', 'snake') and not set as their numeric position.

```
[33]: ax.set_xticks([5, 20, 40, 55, 60])
fig
```



View the tick labels and notice the last two are empty strings.

```
[34]: list(ax.get_xticklabels())
```

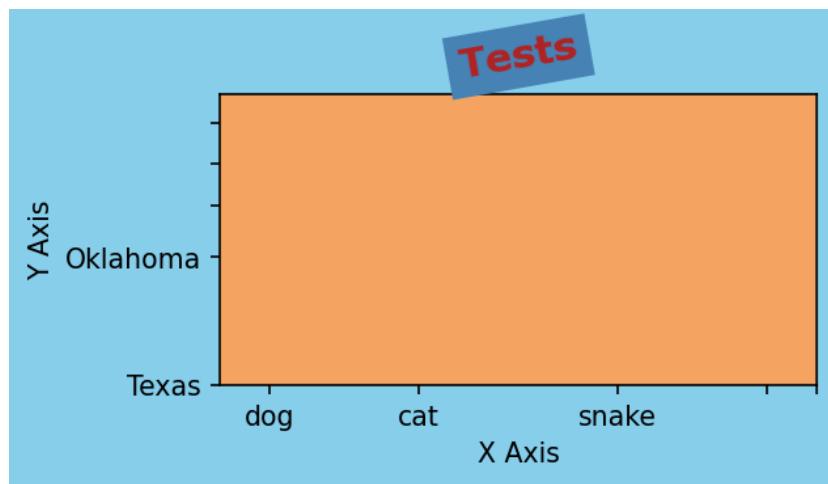
```
[34]: [Text(5, 0, 'dog'),
       Text(20, 0, 'cat'),
       Text(40, 0, 'snake'),
       Text(55, 0, ''),
       Text(60, 0, '')]
```

Styling text

All of the text we placed on our plot used the default matplotlib styling. There are many different parameters that we can set to customize the appearance of our text with some of the most common below. To view all of the options, [visit the text tutorial in the documentation](#). Notice that most of these properties have aliases. Below, we use some of these properties when setting the title.

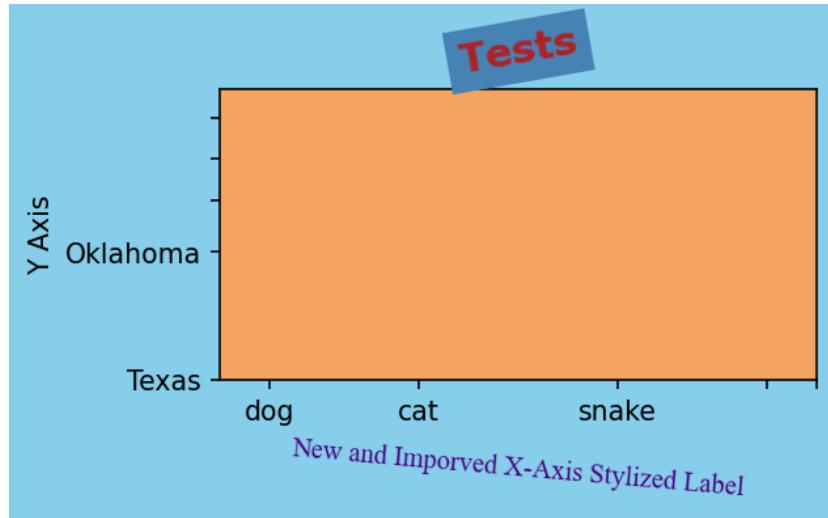
Property	Possible Values
fontsize or size	integer in “points” where 1 point is defaulted to 1/72nd of an inch
fontname or name	name of font as a string
fontweight or weight	'normal', 'bold', 'heavy', 'light', 'ultrabold', 'ultralight'
fontstyle or style	'normal', 'italic', 'oblique'
color or c	text color
backgroundcolor	color of rectangular background of text
horizontalalignment or ha	'left', 'center', 'right'
verticalalignment or va	'center', 'top', 'bottom', 'baseline'
rotation	degree of rotation

```
[35]: ax.set_title('Tests', fontsize=15, fontname='Verdana', fontweight='bold',
                  color='firebrick', backgroundcolor='steelblue', rotation=10)
fig
```



Any other text may be stylized with those same parameters. Below we do so with the x-label.

```
[36]: ax.set_xlabel('New and Improved X-Axis Stylized Label', fontsize=10,
                  color='indigo', fontname='Times New Roman', rotation=-5)
fig
```



60.5 Change tick label and tick line properties with `tick_params`

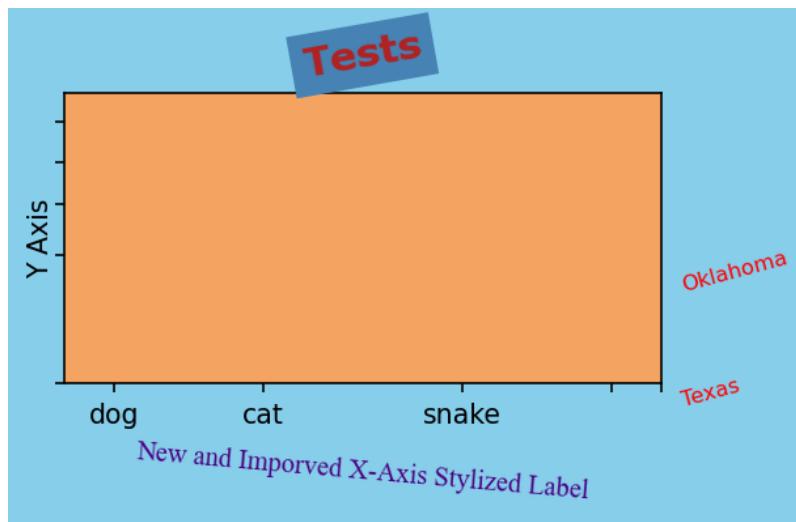
The `tick_params` method allows you to change some of the properties of the tick labels and tick lines.

Changing tick label properties

We'll begin using `tick_params` to change the tick labels. First, set the `axis` parameter to the string '`x`', '`y`', or '`both`' to select which labels to change. The parameters beginning with '`label`' change the tick labels, all of which are listed below. We then use `tick_params` to change properties of the y-axis labels.

- `labelsize`, `labelrotation`, `labelcolor` - change label size, degree of rotation, and color
- `labelleft`, `labelright`, `labelbottom`, `labeltop` - boolean to control whether labels are visible or not

```
[37]: ax.tick_params(axis='y', labelleft=False, labelright=True, labelsize=8,
                  labelrotation=15, labelcolor='red')
fig
```

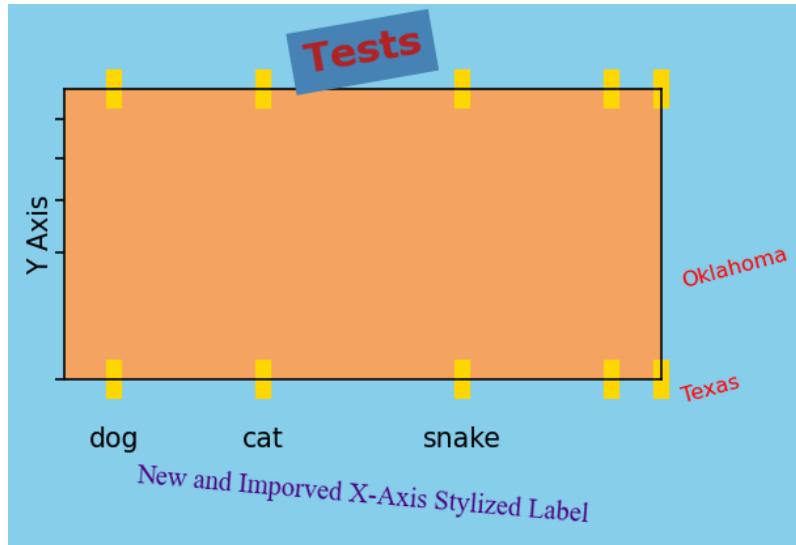


Changing tick line properties

The tick lines are the tiny lines on each axis that denote the tick locations and point to the tick labels. There exists a `get_xticklines` method, but no `set_xticklines` to change them. Instead, use the `tick_params` method to set the properties on the lines themselves. Some of its parameters are:

- `length` - length of tick line in points
- `width` - width of tick line in points
- `pad` - distance in points of tick label from tick line
- `direction` - 'in', 'out', or 'inout'
- `top`, `bottom`, `left`, `right` - boolean corresponding to the top/bottom x-axis and left/right y-axis that determines whether ticks are visible.

```
[38]: ax.tick_params(axis='x', color='gold', length=15, width=6,
                  pad=10, direction='inout', top=True, right=True)
fig
```

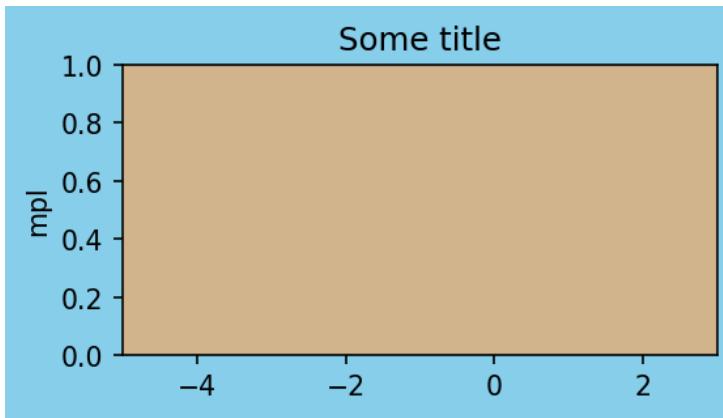


60.6 Setting multiple properties at the same time with `set`

Most matplotlib objects have a `set` method that can be used to set many properties at once in a single line of code. Use the property name as the parameter name and set it equal to the value you would like. Here,

we set the title, face color, x-axis limits, and y-axis label at once.

```
[39]: fig, ax = plt.subplots(figsize=(4, 2), facecolor='skyblue', dpi=147)
ax.set(title='Some title', facecolor='tan', xlim=(-5, 3), ylabel='mpl');
```



60.7 Exercises

Exercise 1

Create a figure with dimensions 5 inches by 3 inches with 147 DPI containing a single axes. Set the facecolor of the figure and the axes. Set a title and labels for the x and y axis. Set three ticks on the y-axis, and two on the x-axis. Give the three ticks on the y-axis a new string label. Change the limits of the x-axis and y-axis so they are larger than the minimum and maximum tick values. Change the size, shape, and color of the y-axis tick lines. Increase the size of the x tick labels and rotate them.

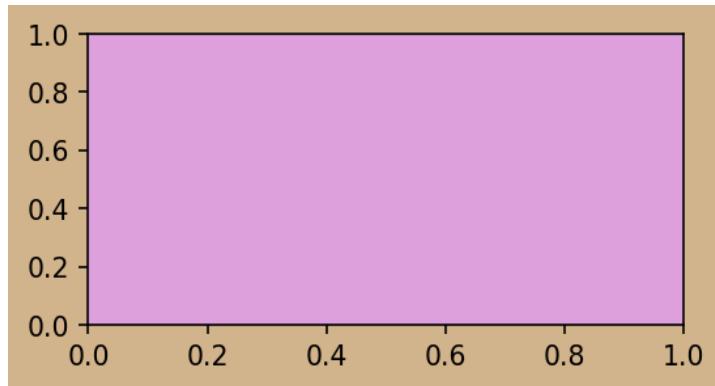
```
[ ]:
```


Chapter 61

Matplotlib Text and Lines

In this chapter, we cover how to add text and lines on our axes. Let's create our figure and axes with the `subplots` function and unpack these objects to `fig` and `ax`. We will also set the background color of each to distinguish one from the other.

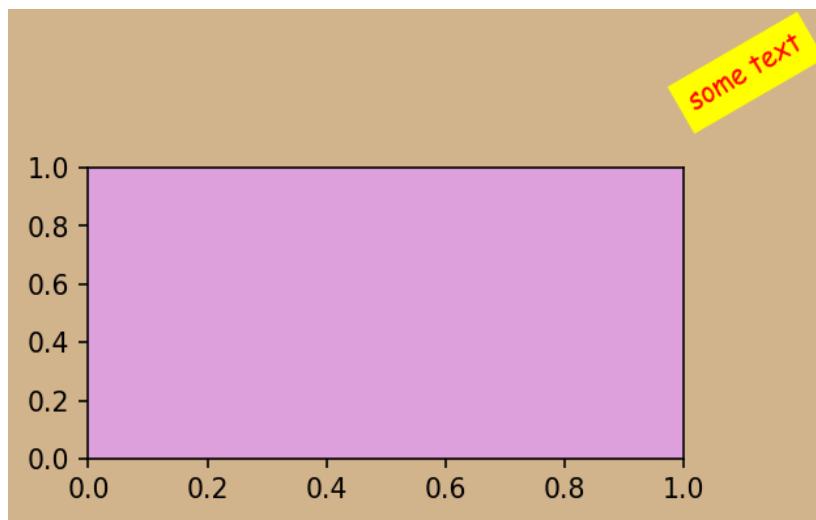
```
[1]: import matplotlib.pyplot as plt
import pandas as pd
fig, ax = plt.subplots(figsize=(4, 2), dpi=147)
fig.set_facecolor('tan')
ax.set_facecolor('plum')
```



61.1 The axes `text` method

Previously, we changed the text of the x and y axis labels, the title of the axes, and the tick labels. We can place text anywhere on our axes with the `text` method. We pass it the x and y coordinates of the text, the text itself, and any of the text properties we previously learned. We choose coordinates outside of the current x and y limits. Although the text appears outside of the bounds of the axes, it is still contained in the axes.

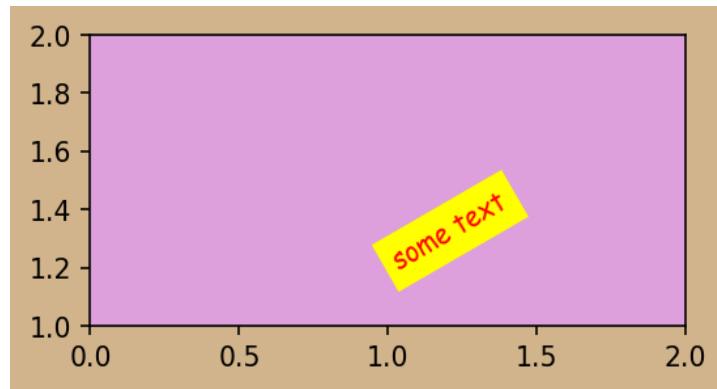
```
[2]: ax.text(x=1, y=1.2, s='some text',
            color='red', backgroundcolor='yellow',
            fontsize=10, rotation=30, fontname='Comic Sans MS')
fig
```



Change the x and y limits

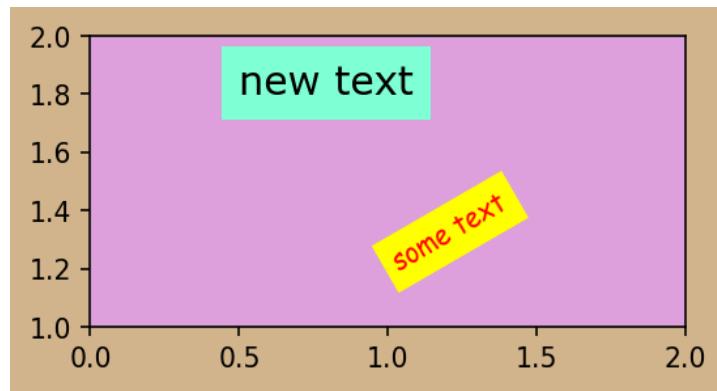
We can change the limits of our x and y axis with the `set_xlim` and `set_ylim` methods so that the text appears within the limits of the axes.

```
[3]: ax.set_xlim(0, 2)
        ax.set_ylim(1, 2)
        fig
```



Assigning the result of a matplotlib text object to a variable

Whenever we call the `text` method, matplotlib returns a reference to that object. Above, we did not assign the result to a variable. Let's add new text and assign the result to a variable name so that we can reference it later.



Let's display the contents of the variable `new_text` and verify its type.

```
[5]: new_text
```

```
[5]: Text(0.5, 1.8, 'new text')
```

```
[6]: type(new_text)
```

```
[6]: matplotlib.text.Text
```

Getter and setter text methods

As with the figure and axes, this text object (and all matplotlib objects) has its own getter and setter methods unique to it. Let's begin by getting of the text properties.

```
[7]: new_text.get_color()
```

```
[7]: 'black'
```

```
[8]: new_text.get_fontsize()
```

```
[8]: 15.0
```

```
[9]: new_text.get_position()
```

```
[9]: (0.5, 1.8)
```

```
[10]: new_text.get_fontname()
```

```
[10]: 'Verdana'
```

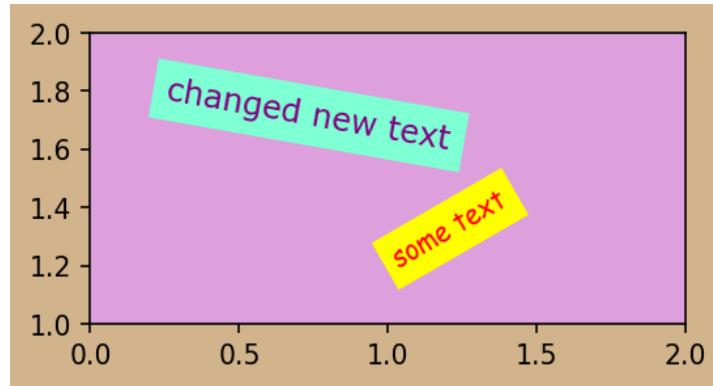
```
[11]: new_text.get_text()
```

```
[11]: 'new text'
```

Let's change a few properties with the setter methods.

```
[12]: new_text.set_color('purple')
new_text.set_fontsize(12)
new_text.set_position((.25, 1.6))
new_text.set_rotation(-10)
```

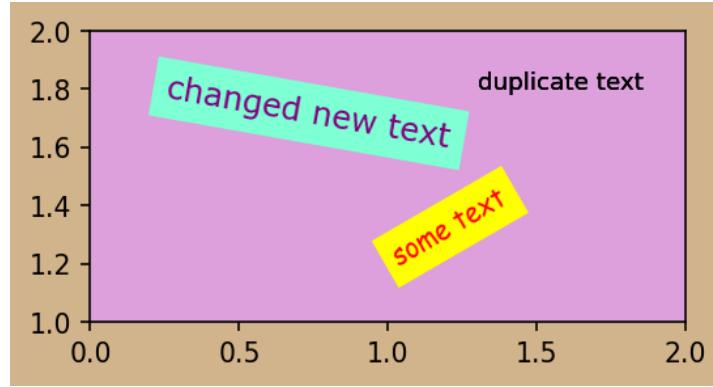
```
new_text.set_text('changed new text')
fig
```



Duplicate text objects

Every time you call the `text` method, a completely new text object is created and added on the axes. Even if the same exact arguments are used or if the same notebook cell is executed, a new text object is placed on the axes. The next cell creates three unique text objects with the same properties at the same location, though it will appear on the screen as a single piece of text.

```
[13]: ax.text(x=1.3, y=1.8, s='duplicate text', size=9)
ax.text(x=1.3, y=1.8, s='duplicate text', size=9)
ax.text(x=1.3, y=1.8, s='duplicate text', size=9)
fig
```



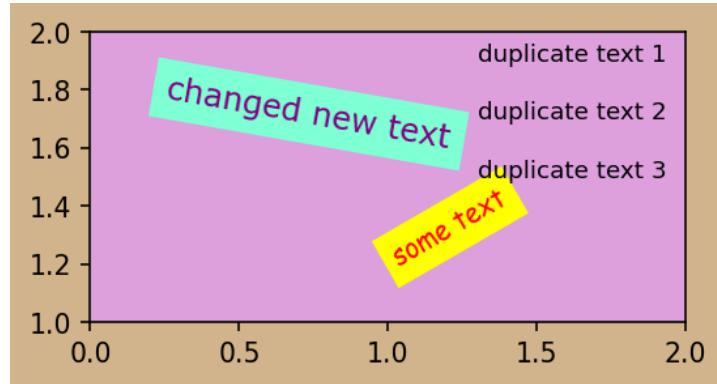
Even though we didn't assign any of the three text objects to a variable, we can access them with the axes `texts` attribute. It returns a list of all the text object in the order they were created.

```
[14]: ax.texts
```

```
[14]: [Text(1, 1.2, 'some text'),
Text(0.25, 1.6, 'changed new text'),
Text(1.3, 1.8, 'duplicate text'),
Text(1.3, 1.8, 'duplicate text'),
Text(1.3, 1.8, 'duplicate text')]
```

Let's access each of the last three text objects and move them to a different location so that we can actually see them. We'll change the text as well.

```
[15]: ax.texts[2].set_position((1.3, 1.9))
ax.texts[2].set_text('duplicate text 1')
ax.texts[3].set_position((1.3, 1.7))
ax.texts[3].set_text('duplicate text 2')
ax.texts[4].set_position((1.3, 1.5))
ax.texts[4].set_text('duplicate text 3')
fig
```



Remove text object

All matplotlib objects can be removed from the axes with the `remove` method. Let's remove the last text object.

```
[16]: ax.texts[4].remove()
fig
```



We verify that now only four text objects remain.

```
[17]: ax.texts
```

```
[17]: [Text(1, 1.2, 'some text'),
Text(0.25, 1.6, 'changed new text'),
Text(1.3, 1.9, 'duplicate text 1'),
Text(1.3, 1.7, 'duplicate text 2')]
```

Get all properties of an object

Most matplotlib objects have a `properties` method that can be called to return a dictionary of each property name with its associated value. Most objects have several dozen properties. Below, all the properties are

assigned to a variable. This dictionary is then iterated through with each property value that is an either a boolean, int, or float printed to the screen.

```
[18]: print(f'{"Property Name":^20} {"Value":^20}')
print('-' * 20, ' ', '-' * 20)
property_dict = ax.texts[0].properties()
for prop, value in property_dict.items():
    if isinstance(value, (str, int, float)):
        print(f'{prop:20} {value:<20}'')
```

Property Name	Value
animated	0
clip_on	0
color	red
fontname	Comic Sans MS
fontsize	10.0
fontstyle	normal
fontvariant	normal
fontweight	normal
horizontalalignment	left
in_layout	1
label	
rotation	30.0
stretch	normal
text	some text
usetex	0
verticalalignment	baseline
visible	1
wrap	0
zorder	3

There are many more properties than the ones listed above. A good way to explore the available properties is to scan through the list of setter methods. Below, all setter methods are printed out.

```
[19]: text_setters = [m for m in dir(ax.texts[0]) if m.startswith('set_')]
for i, setter in enumerate(text_setters):
    end = '\n' if i % 3 == 2 else ''
    print(f'{setter:25}', end=end)
```

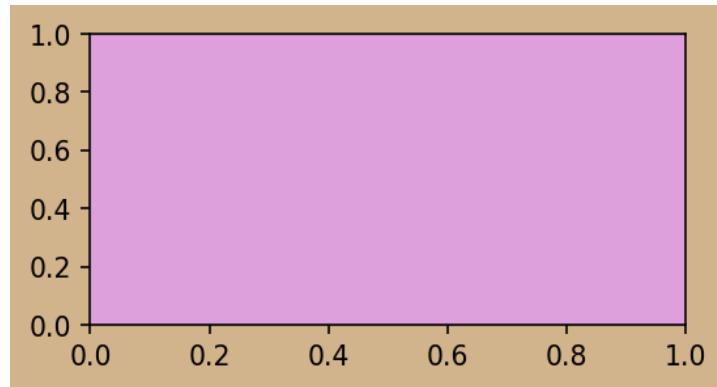
set_agg_filter	set_alpha	set_animated
set_backgroundcolor	set_bbox	set_c
set_clip_box	set_clip_on	set_clip_path
set_color	set_contains	set_family
set_figure	set_font_properties	set_fontfamily
set_fontname	set_fontproperties	set fontsize
set_fontstretch	set_fontstyle	set_fontvariant
set_fontweight	set_gid	set_ha
set_horizontalalignment	set_in_layout	set_label
set_linespacing	set_ma	set_multialignment
set_name	set_path_effects	set_picker
set_position	set_rasterized	set_rotation
set_rotation_mode	set_size	set_sketch_params

```
set_snap          set_stretch        set_style
set_text          set_transform      set_url
set_usetex        set_va           set_variant
set_verticalalignment set_visible    set_weight
set_wrap          set_x            set_y
set_zorder
```

Clear all objects from the axes

You can remove all of the objects from the axes with the `clear` method. All of the labels and limits will reset to their defaults as well. The figure size remains the same along with its and the axes facecolor.

```
[20]: ax.clear()
fig
```

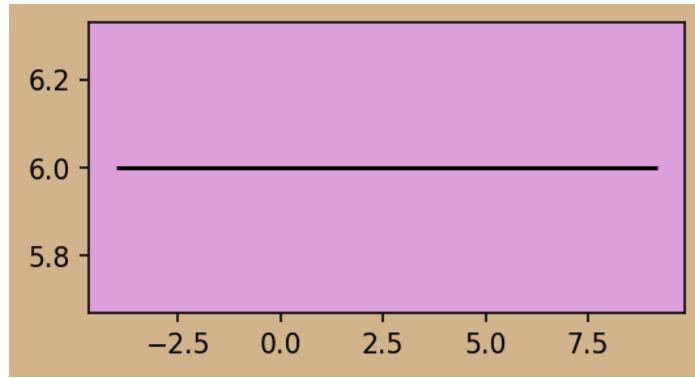


61.2 Creating horizontal lines with `hlines`

Horizontal lines may be added with the `hlines` method. It requires the following three parameters:

- `y` - the y coordinate
- `xmin` - the starting point of the line
- `xmax` - the ending point of the line

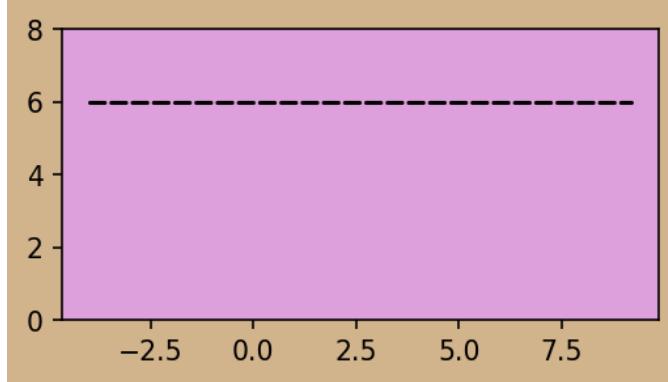
```
[21]: ax.hlines(y=6, xmin=-4, xmax=9.2)
fig
```



If the x and y limits have not been set, then matplotlib will set them for you so that the entire line is visible. Notice that the y-axis limits have shrunk to just a small range around the line's y-coordinate. Most axes plotting functions automatically change the limits of the graph to just the region of where the objects are located (if the limits were not set previously). Let's clear the axes and then set the y-axis limits before

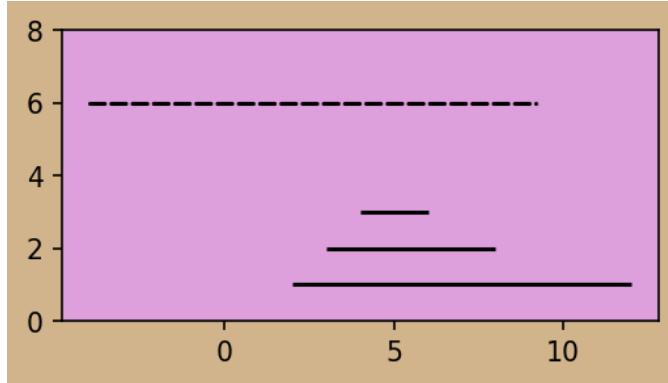
adding the same horizontal line. The x-axis limits will still change so that just enough space is available to fit the line, but the y-axis limits remain set.

```
[22]: ax.clear()
ax.set_ylim(0, 8)
ax.hlines(y=6, xmin=-4, xmax=9.2, capstyle='round', ls='dashed')
fig
```



We can create multiple horizontal lines at the same time using lists. Here, we create three horizontal lines at y-values of 1, 2, and 3 each with a different starting and ending point.

```
[23]: ax.hlines(y=[1, 2, 3], xmin=[2, 3, 4], xmax=[12, 8, 6])
fig
```



Common line properties

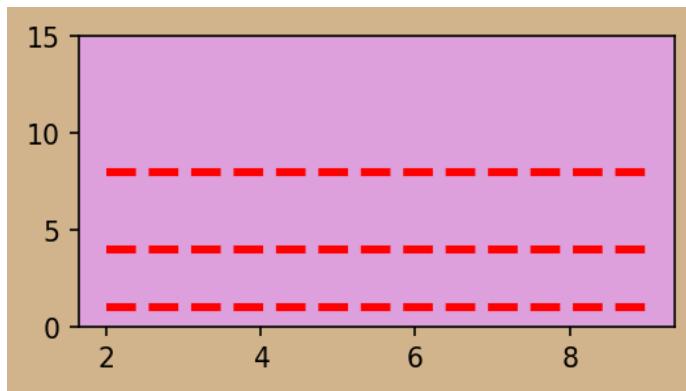
All lines in matplotlib have many different properties that can be set. In addition to `color`, the following are the most common.

Property	Possible Values
<code>linewidth</code> or <code>lw</code>	width of line in points
<code>linestyle</code> or <code>ls</code>	'solid' or '-' (default), 'dashed' or '--', 'dotted' or ':', 'dashdot' or '-.'

We clear our axes and plot three horizontal lines changing the default properties. Notice that we also assign the returned value from the `hlines` method to a variable name.

```
[24]: ax.clear()
ax.set_ylim(0, 15)
```

```
horiz_lines = ax.hlines(y=[1, 4, 8], xmin=2, xmax=9, linewidth=3,
                        linestyle='dashed', color='red')
fig
```



Let's output this object to the screen and identify its type.

```
[25]: horiz_lines
```

```
[25]: <matplotlib.collections.LineCollection at 0x11cf0c590>
```

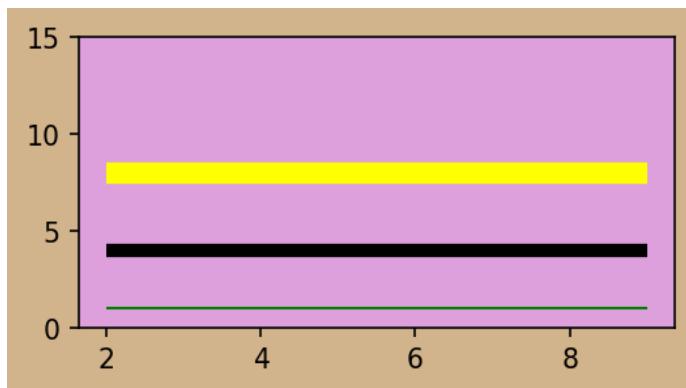
This is a LineCollection object, which, as the name implies, is a collection of lines. The specifics of this object aren't too important to know. But what is important, is your ability to call methods from it. Just like all matplotlib objects it too has getter and setter methods. Let's get the width of the line collection.

```
[26]: horiz_lines.get_linewidth()
```

```
[26]: array([3])
```

The width is returned within a numpy array and not as a scalar because each line can have its own width. Let's set new properties for each of the lines in the collection. Passing a single value will set that property for all lines. Passing in a list sets each line individually.

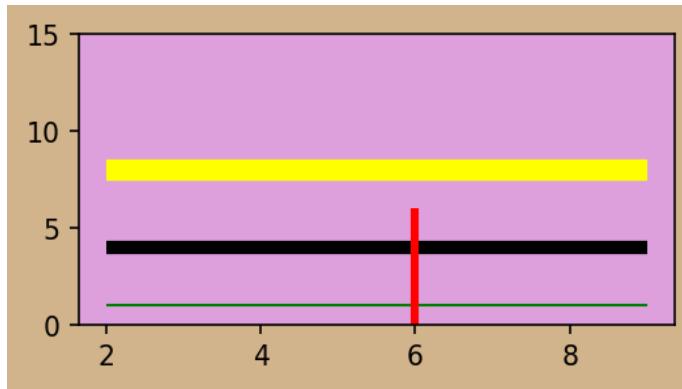
```
[27]: horiz_lines.set_linestyle('solid')
horiz_lines.set_color(['green', 'black', 'yellow'])
horiz_lines.set_linewidth([1, 5, 8])
fig
```



61.3 Create vertical lines with vlines

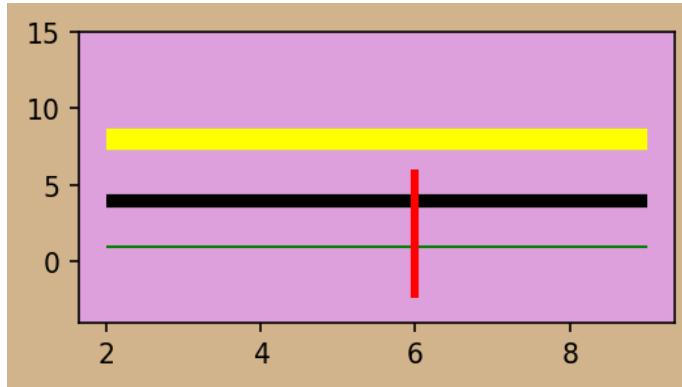
We can use the `vlines` method similarly to create vertical lines. The parameters are the opposite as before. Provide it an x-value of the vertical location of the line along with the starting and ending y-values.

```
[28]: ax.vlines(x=6, ymin=-2.4, ymax=6, lw=3, color='red')
fig
```



Since we've already set the limits of our graph, matplotlib won't expand them to fit the new line. We need to change the limits manually.

```
[29]: ax.set_ylim(-4, 15)
fig
```



Retrieve the lines from the graph

We retrieved all of the text objects as a list with the `texts` attribute. Similarly, we can retrieve all of the LineCollection objects as a list with the `collections` attribute. We have two items in our list. The first collection contains three horizontal lines, while the second contains one vertical line.

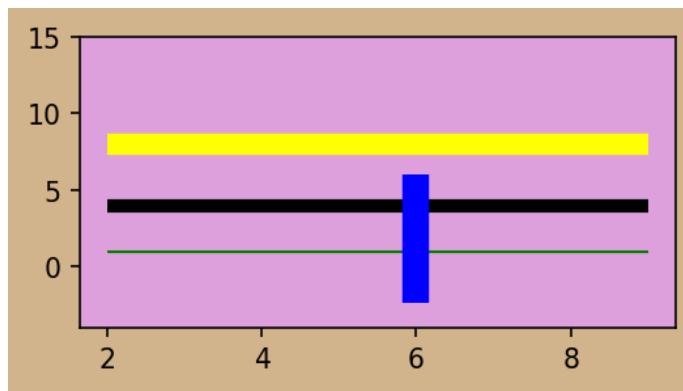
```
[30]: ax.collections
```

```
[30]: [<matplotlib.collections.LineCollection at 0x11cf0c590>,
<matplotlib.collections.LineCollection at 0x11f39b210>]
```

Access the collection and change its properties

Let's access the second line collection we created from the list and then use the setter methods to change its properties.

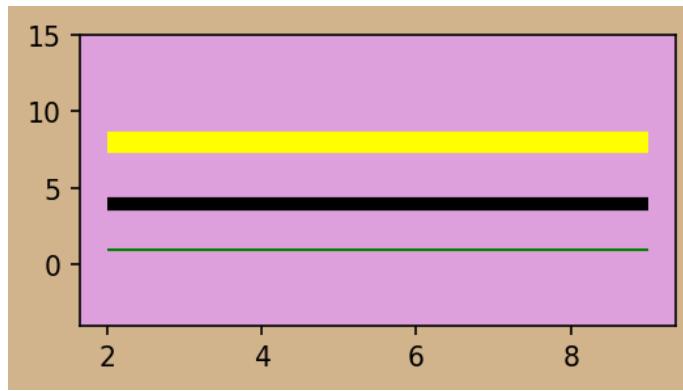
```
[31]: ax.collections[1].set_linewidth(10)
ax.collections[1].set_color('blue')
fig
```



Remove an item from the zxes

As previously mentioned, matplotlib objects have a `remove` method which removes them from the axes. Let's remove our second line collection.

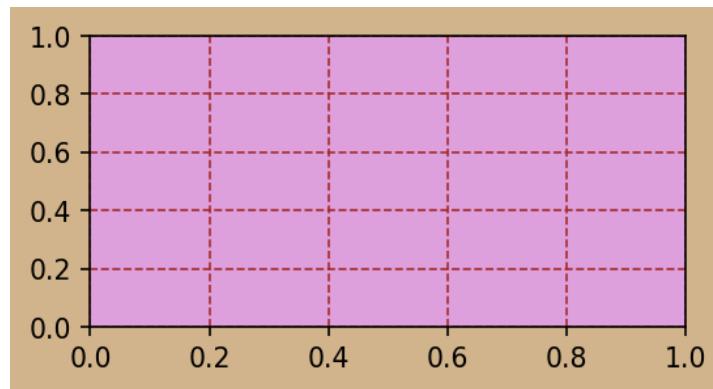
```
[32]: ax.collections[1].remove()
fig
```



61.4 Add grid lines with the grid method

Occasionally, it can be helpful to have horizontal and vertical lines for all of the tick marks in the plot. These are referred to as grid lines and can be added with the axes `grid` method. The first argument is a boolean that controls whether to turn on or off the grid. Here we set it to `True`. Set the `axis` parameter to '`x`', '`y`', or '`both`' to determine which axis will have grid lines. You may also pass it any other line properties.

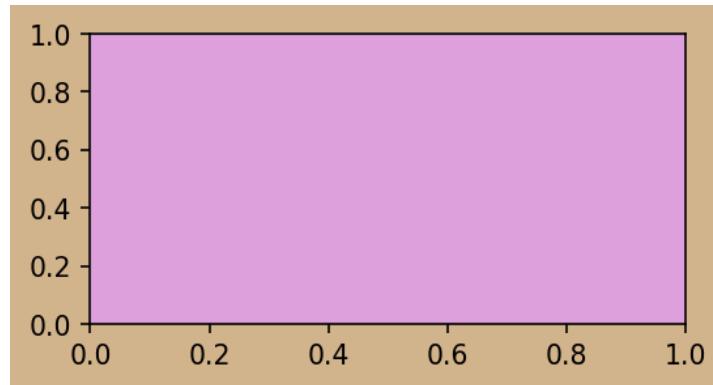
```
[33]: ax.clear()
ax.grid(True, axis='both', linestyle='dashed', color='brown')
fig
```



Toggle the grid on and off

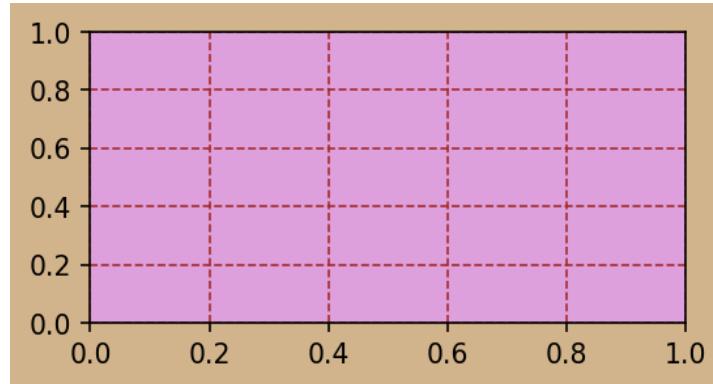
The `grid` method does not return a matplotlib object like the other methods in this chapter. Instead, matplotlib treats the `grid` method like an on/off switch. The first argument is a boolean that is used to toggle the grid. Here, we turn it off.

```
[34]: ax.grid(False)  
fig
```



If we turn it back on, the same properties that we set previously remain.

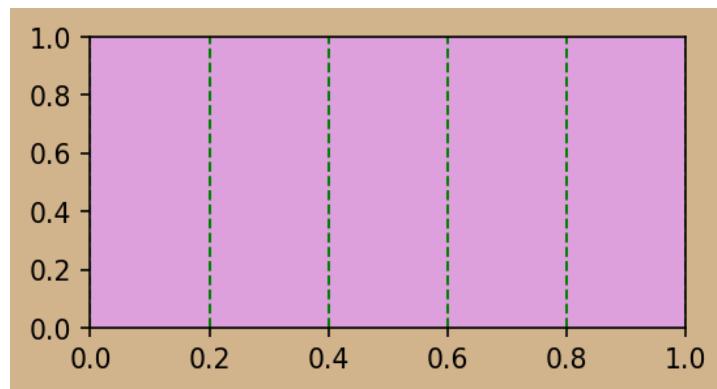
```
[35]: ax.grid(True)  
fig
```



Control which axis has the grid

Use the `axis` method to control which direction the grid appears. By default it is set to ‘both’.

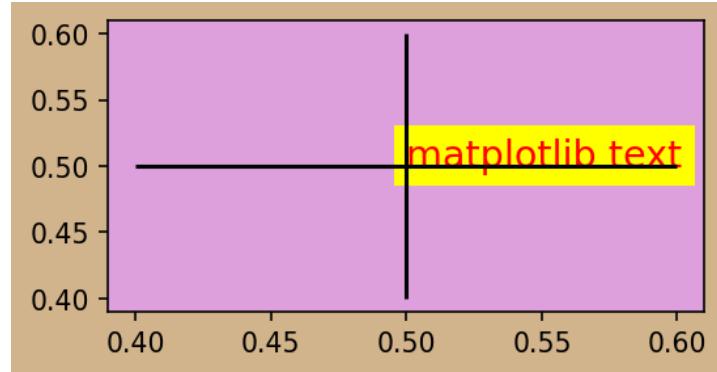
```
[36]: ax.clear()
ax.grid(axis='x', linestyle='dashed', color='green', linewidth=1)
fig
```



61.5 Aligning text horizontally and vertically

By default, text begins from the x and y coordinates given and proceeds to the right. The bottom left of the first letter is placed at this coordinate. Let's add a single text object to our axes at (.5, .5) along with a horizontal and vertical line intersecting at that point. The `zorder` parameter controls the ordering of the plotting objects when they cover the same points. The text is set with a `zorder` of 0, which is the lower than the other objects, ensuring that the lines will be visible on top of it.

```
[37]: ax.clear()
text = ax.text(x=.5, y=.5, s='matplotlib text', size=14,
               backgroundcolor='yellow', color='red', zorder=0)
ax.hlines(y=.5, xmin=.4, xmax=.6)
ax.vlines(x=.5, ymin=.4, ymax=.6)
fig
```



The `horizontalalignment` and `verticalalignment` parameters control where the text is placed relative to its given coordinate. By default, these values are set to 'left' and 'baseline'. matplotlib has created the aliases `ha` and `va` to help shorten the syntax. Let's use the getter methods to verify the default values.

```
[38]: text.get_ha()
```

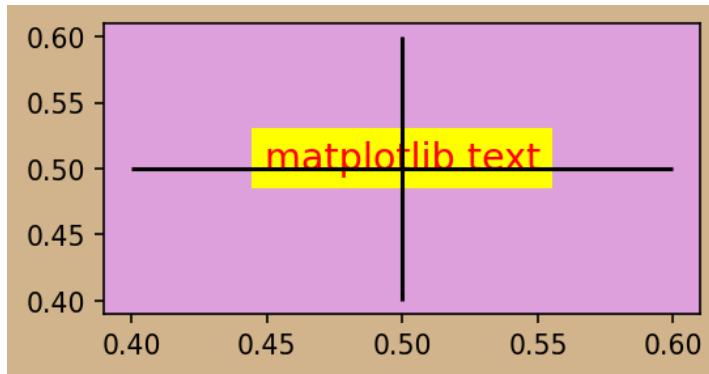
```
[38]: 'left'
```

```
[39]: text.get_va()
```

[39]: 'baseline'

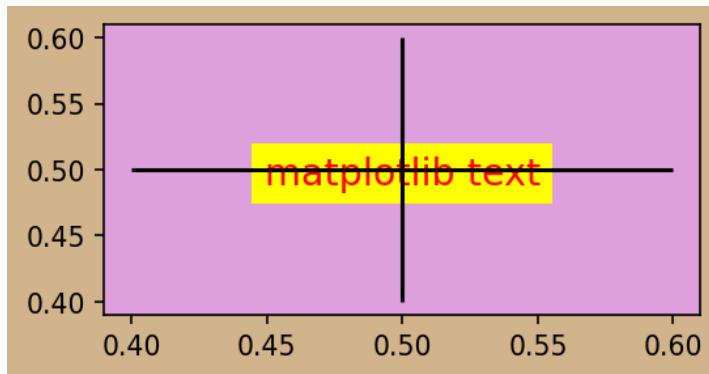
The possible values for `horizontalalignment` are 'left', 'center', and 'right'. Let's change it to 'center'.

```
[40]: text.set_ha('center')
fig
```



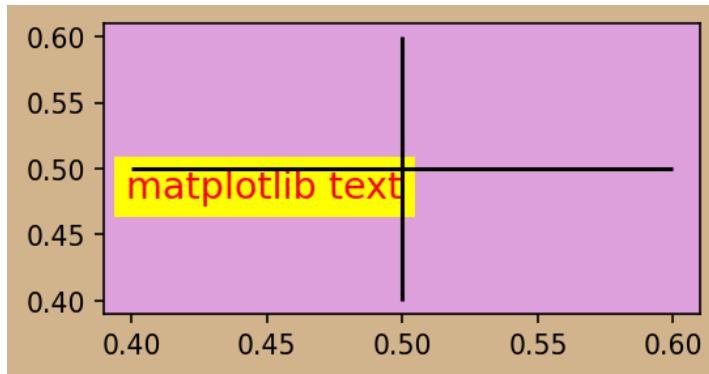
The possible values for `verticalalignment` are 'top', 'bottom', 'center', 'baseline', and 'center_baseline'.

```
[41]: text.set_va('center_baseline')
fig
```



Another combination of text alignment is chosen.

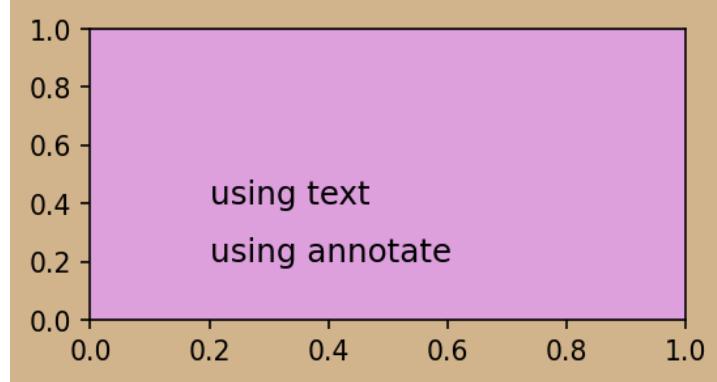
```
[42]: text.set_ha('right')
text.set_va('top')
fig
```



61.6 Add text with arrows using the `annotate` method

The axes `annotate` method is very similar to the `text` method but allows you to draw an arrow from a text label to another point. The method signature for `annotate` is slightly different. Set the `s` parameter to a string of the text you want to display and `xy` to a tuple of the location of the text. Text is added to our axes with both methods below.

```
[43]: ax.clear()
ax.annotate(s='using annotate', xy=(.2, .2), size=12)
ax.text(x=.2, y=.4, s='using text', size=12)
fig
```



Like before, the `texts` attribute returns a list of all the text objects, including those created by the `annotate` method. Let's access this list now and output the official type of each of the two objects.

```
[44]: ax.texts
```

```
[44]: [Text(0.2, 0.2, 'using annotate'), Text(0.2, 0.4, 'using text')]
```

```
[45]: type(ax.texts[0])
```

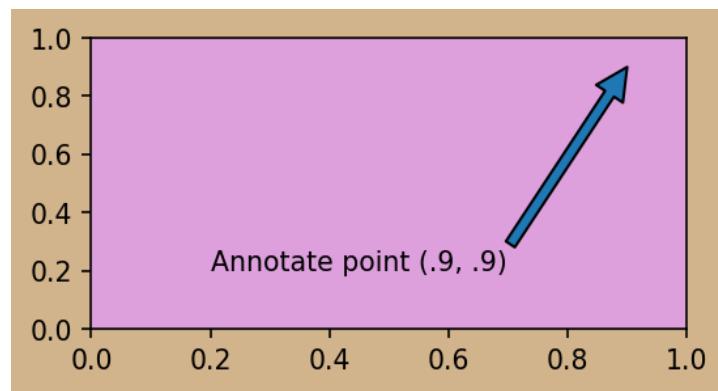
```
[45]: matplotlib.text.Annotation
```

```
[46]: type(ax.texts[1])
```

```
[46]: matplotlib.text.Text
```

The object created from the `annotate` method has a different type. You are free to use either `text` or `annotate` to add text to your axes, but you must use `annotate` if you want to draw an arrow from text to a point. To do so, use the `xy` parameter to denote the location of the point you would like to annotate and `xytext` for the location of where the text will be. For the arrow to appear you must set the `arrowprops` parameter to a dictionary. Using an empty dictionary creates a default arrow.

```
[47]: ax.clear()
ax.annotate(s='Annotate point (.9, .9)', xytext=(.2, .2), xy=(.9, .9), arrowprops={})
fig
```



Styling the arrow

The arrow can be styled in a variety of ways by setting the `arrowprops` parameter to a dictionary. Within this dictionary, you can set the '`arrowstyle`' to a string. This string is composed of the name of the style followed by comma-separated attributes with equal signs denoting their value. All of the possible values for `arrowstyle` are provided in the table below.

Name	Attributes
-	None
->	<code>head_length=0.4, head_width=0.2</code>
-[<code>widthB=1.0, lengthB=0.2, angleB=None</code>
-\ >	<code>head_length=0.4, head_width=0.2</code>
<-	<code>head_length=0.4, head_width=0.2</code>
<->	<code>head_length=0.4, head_width=0.2</code>
<\ -	<code>head_length=0.4, head_width=0.2</code>
<\ -\\ >	<code>head_length=0.4, head_width=0.2</code>
]-	<code>widthA=1.0, lengthA=0.2, angleA=None</code>
]-[<code>widthA=1.0, lengthA=0.2, angleA=None, widthB=1.0, lengthB=0.2, angleB=None</code>
fancy	<code>head_length=0.4, head_width=0.4, tail_width=0.4</code>
simple	<code>head_length=0.5, head_width=0.5, tail_width=0.2</code>
wedge	<code>tail_width=0.3, shrink_factor=0.5</code>
\ -\\	<code>widthA=1.0, angleA=None, widthB=1.0, angleB=None</code>

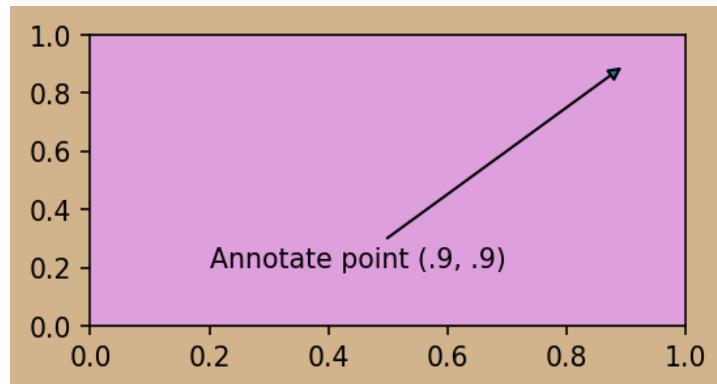
Here are some example strings that are possible.

- `'->, head_length=.4, head_width=1.2'`
- `']-, widthA=1, lengthA=4, angleA=30'`
- `'fancy, head_length=1.4, head_width=2'`
- `'wedge, tail_width=.8, shrink_factor=.3'`
- `'fancy'`

At a minimum, you need to provide one of the names. It is not necessary to provide any of the attributes and if you do not, they will be assigned to their default value shown in the table above. Each arrow style has specific attributes and you can only use the ones made for it. For example, the 'fancy' arrow style has attributes '`head_length`', '`head_width`', and '`tail_width`'. If you provide it '`angleA`', you'll get an error. Let's use the arrowstyle '`-|>`' with the default settings.

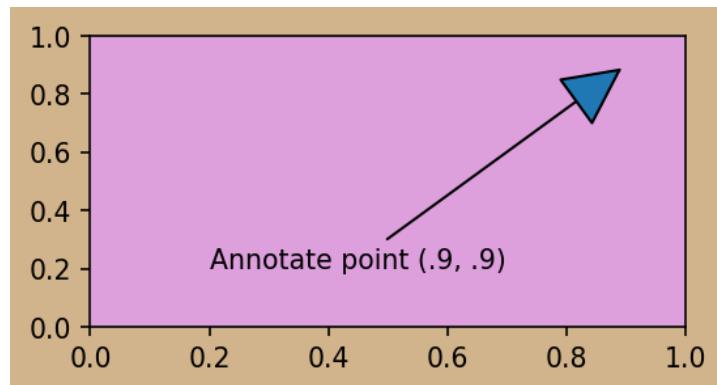
```
[48]: ax.clear()
arrowprop_dict = {'arrowstyle': '-|>'}
ax.annotate(s='Annotate point (.9, .9)', xytext=(.2, .2), xy=(.9, .9),
```

```
    arrowprops=arrowprop_dict)
fig
```



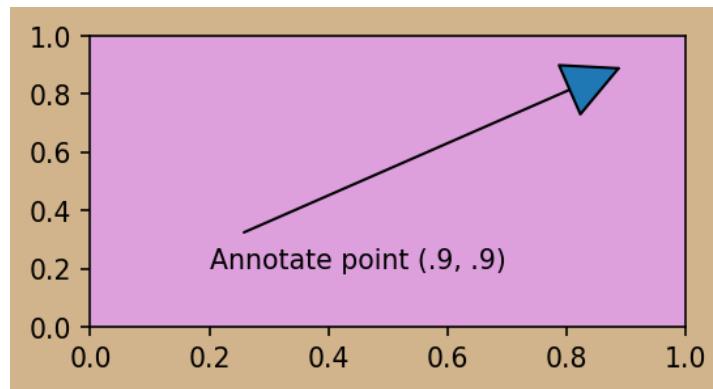
The default settings for this arrow style are .4 and .2 for the 'head_length' and 'head_width' respectively. Let's increase both by modifying our arrowstyle string.

```
[49]: ax.clear()
arrowprop_dict = {'arrowstyle': '-|>, head_length=2, head_width=1'}
ax.annotate(s='Annotate point (.9, .9)', xytext=(.2, .2), xy=(.9, .9),
            arrowprops=arrowprop_dict)
fig
```



You might have noticed that the tail of the arrow is now centered right above the text, whereas before it was anchored at the top-right of the text. Regardless of the arrow style that you choose, the property 'relpos' will be set to (.5, .5). This is the relative position of the start of the arrow to the text. Think of a rectangular box around the text. The bottom left-hand and top right-hand corners have coordinates (0, 0) and (1, 1). Let's add this property to our `arrowprop_dict` changing it so that it begins more to the left and further up.

```
[50]: ax.clear()
arrowprop_dict = {'arrowstyle': '-|>, head_length=2, head_width=1',
                  'relpos': (.1, 1.5)}
ax.annotate(s='Annotate point (.9, .9)', xytext=(.2, .2), xy=(.9, .9),
            arrowprops=arrowprop_dict)
fig
```

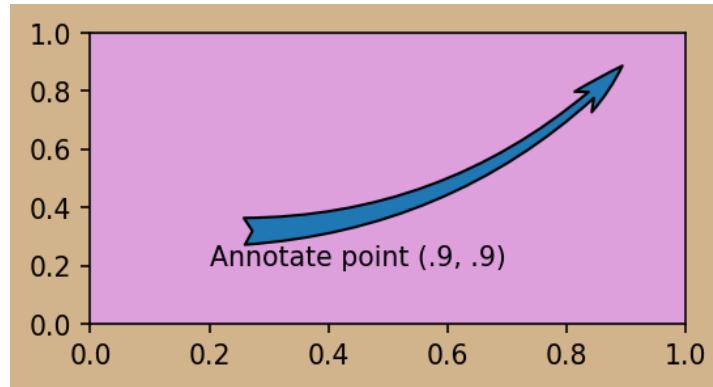


Another property that you might want to change is the 'connectionstyle' which works similarly as arrowstyle. It controls the path that the arrow takes between the two points. Take a look at the table of connection styles below. You'll need to set this property to one of the names, followed by a comma separated list of attributes.

Name	Attributes
angle	angleA=90, angleB=0, rad=0.0
angle3	angleA=90, angleB=0
arc	angleA=0, angleB=0, armA=None, armB=None, rad=0.0
arc3	rad=0.0
bar	armA=0.0, armB=0.0, fraction=0.3, angle=None

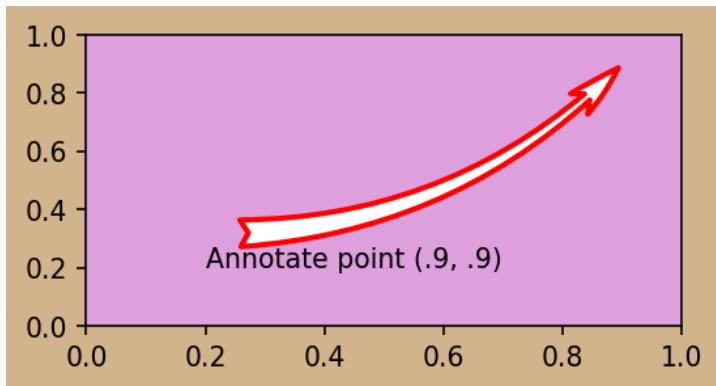
The 'arc3' connection style is one of the easiest to use since it has a single parameter, the radius of the arc. Let's change our arrow style and add this connection style.

```
[51]: ax.clear()
arrowprop_dict = {'arrowstyle': 'fancy, head_length=2, head_width=1, tail_width=1',
                  'relpos': (.1, 1.5),
                  'connectionstyle': 'arc3, rad=.2'}
ax.annotate(s='Annotate point (.9, .9)', xytext=(.2, .2), xy=(.9, .9),
            arrowprops=arrowprop_dict)
fig
```



We can also set generic properties such as `linewidth` or `lw`, `facecolor` or `fc`, and `edgecolor` or `ec` of the arrow. The linewidth corresponds to the width of the edge in points. Its color can be set with edgecolor. The facecolor is the color of the inner surface of the arrow. Below, we change each of these three properties.

```
[52]: ax.clear()
arrowprop_dict = {'arrowstyle': 'fancy, head_length=2, head_width=1, tail_width=1',
                  'relpos': (.1, 1.5),
                  'connectionstyle': 'arc3, rad=.2',
                  'lw': 2, 'ec': 'red', 'fc': 'white'}
ax.annotate(s='Annotate point (.9, .9)', xytext=(.2, .2), xy=(.9, .9),
            arrowprops=arrowprop_dict)
fig
```



61.7 Exercises

Create a new figure and axes for each exercise.

Exercise 1

Add the same text to the same location, but set the horizontal alignment of each so that they don't overlap.

```
[ ]:
```

Exercise 2

Add text so that it is upside down. Use `fontweight` and `fontstyle` to make the text bold and italic.

```
[ ]:
```

Exercise 3

Add a simple piece of text using no other parameters other than `x`, `y`, and `s`. Assign the result to a variable and then use the setter methods to set several properties.

```
[ ]:
```

Exercise 4

Create three “T’s” using `hlines` and `vlines` in different locations, each with a different color and line width.

```
[ ]:
```

Exercise 5

Annotate a point with a double-headed arrow.

[]:

Chapter 62

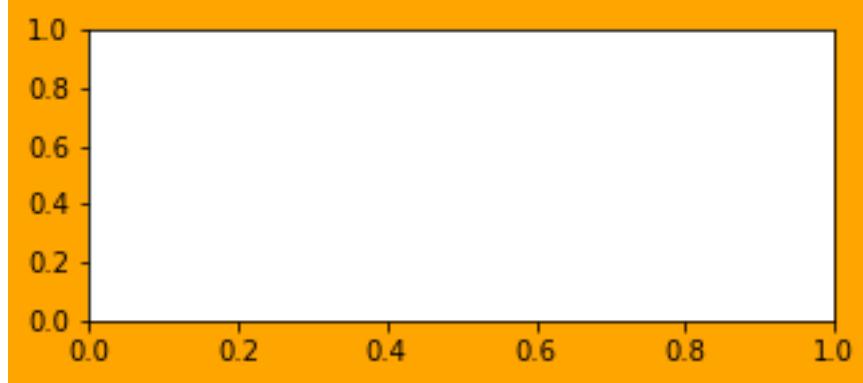
Matplotlib Resolution

In this chapter, we will thoroughly cover the screen resolution of a matplotlib figure. You'll understand what is meant by inches, dots per inch, and typographical points and how to change them to get the exact plot you desire.

62.1 Matplotlib inches

matplotlib uses inches as its unit of measurement for the dimensions of the figure. In Jupyter Notebooks figures are 6 inches in width and 4 inches in height by default. These dimensions can be changed by setting the `figsize` parameter of the `subplots` function. Let's begin by creating a figure with a width of 5 inches and a height of 2 inches.

```
[1]: import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(5, 2), facecolor='orange')
```



If you were to measure the actual width and height of the image produced on your screen, it is likely to be less than the given 5 inches by 2 inches. To help differentiate the inches that matplotlib uses versus the inches on a computer screen, we'll use the terms **figure-inches** and **screen-inches**. In this example, the plot has dimensions of 5 x 2 in figure-inches, and on my screen it's about 2.5 x 1 screen-inches.

To understand why the figure-inches and screen-inches differ, we must understand how computer screen resolution works. Screen resolution is measured in **pixels**, the smallest component of a screen. Dimensions of a screen are reported as the number of pixels along the width and height. For instance, my screen has dimensions of 1,920 by 1,200.

Dots per inch

In order to uncover the screen-inches, we need to understand **dots per inch** or **DPI**. All matplotlib figures have an integer `dpi` property that allows us to find the total number of pixels in the image. By default, the DPI is 72 in a Jupyter Notebook. To find the number of pixels in an image, multiply the figure-inches by the DPI of the matplotlib figure. Our current figure with dimensions of 5 x 2 figure-inches is 360 pixels by 144 pixels. An image with those pixel dimensions is what matplotlib displayed above.

Screen DPI

All computer screens have a resolution in DPI (sometimes referred to as PPI (pixels per inch)) and if this value is known, then the screen-inches can be calculated by dividing the pixels of the image by the screen DPI. For instance, if our screen has a DPI of 120, then the above figure would have screen-inches of 3 ($360 / 120$) by 1.2 ($144 / 120$).

Calculate your screen's DPI

You might need to do some calculations to find your screen's DPI, as this number isn't always found directly in your computer's settings. Often, the diagonal length of your screen will be given. For instance, the diagonal length of my screen is 15.4 inches. If I knew the number of pixels along the diagonal, I could calculate the DPI. Let's use the Pythagorean theorem to calculate the number of pixels along the diagonal.

```
[2]: width = 1920
height = 1200
diag = (width ** 2 + height ** 2) ** .5
diag
```

```
[2]: 2264.155471693585
```

Dividing by the number of inches yields my screen's DPI.

```
[3]: screen_dpi = diag / 15.4
screen_dpi
```

```
[3]: 147.0230825775055
```

My screen's resolution has a DPI of 147. Let's use it to calculate the screen-inches of our above figure.

```
[4]: 360 / screen_dpi
```

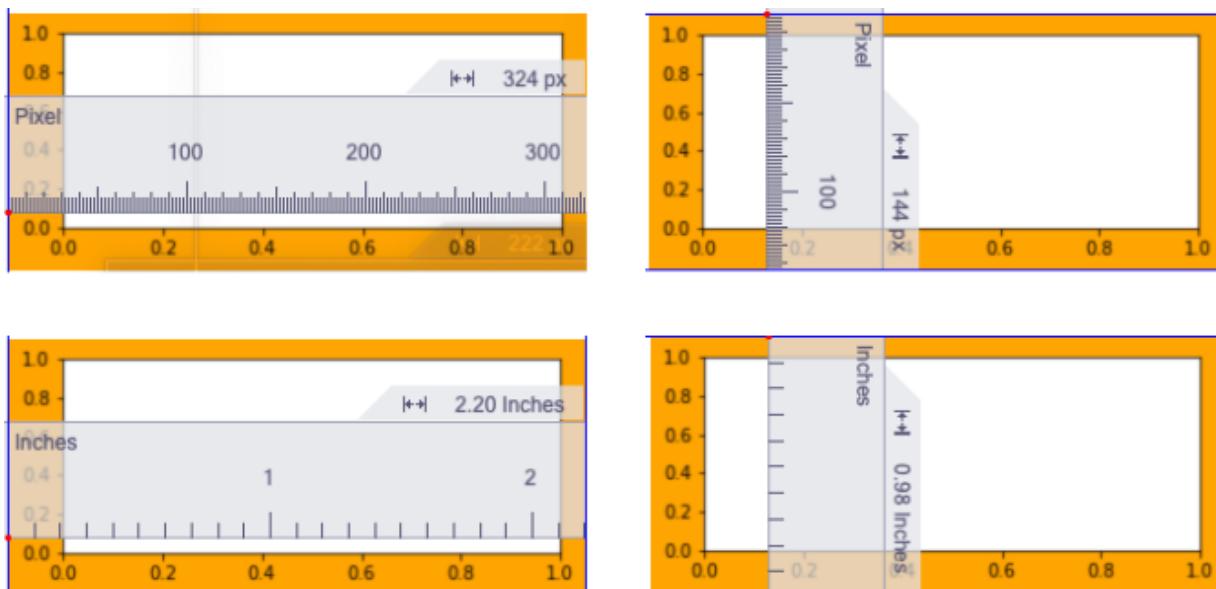
```
[4]: 2.4485951028146915
```

```
[5]: 144 / screen_dpi
```

```
[5]: 0.9794380411258766
```

Verifying calculations with screen measurements

We calculated that the pixel dimensions were 360 x 144 and that the screen-inches were 2.45 x .98. To verify these numbers, I used a program called Onde Rulers, which is an application for macOS that overlays a ruler to help measure items on the screen. This program reported a measurement in pixels of 324 x 144, and a measurement in inches of 2.2 x .98. The width we calculated isn't corresponding to the width of the screen measurements, but the height appears to be exactly correct.



Jupyter Notebooks trim figures

There's a setting in the Jupyter Notebook that automatically trims the edges of a figure. The specific setting `bbox_inches` is set to the string '`tight`' when we first import pyplot. The 'bbox' part of the name stands for 'bounding box' and controls the amount of the figure that is actually shown. When set to '`tight`', it will trim any excess space in the figure where there are no plotting objects in that space. We can view the actual setting by running the following magic command.

```
[6]: %config InlineBackend.print_figure_kwarg
```

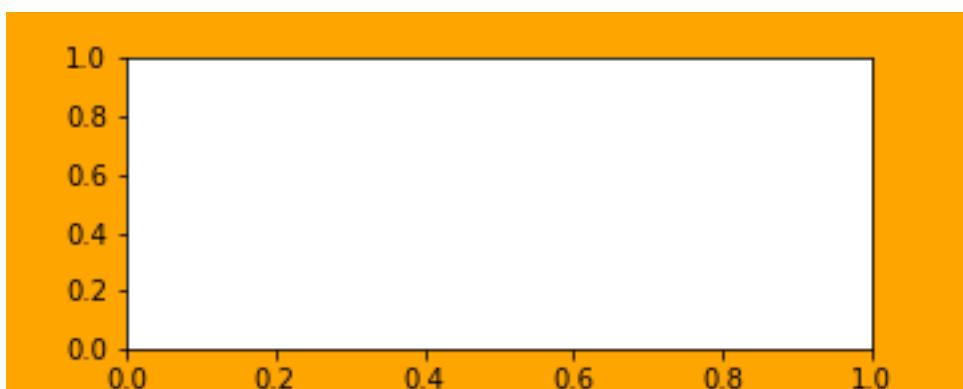
```
[6]: {'bbox_inches': 'tight'}
```

To view the entire figure, with its exact dimensions, we must set the `bbox_inches` to `None`.

```
[7]: %config InlineBackend.print_figure_kwarg = {'bbox_inches': None}
```

Now, when we output the figure, we'll get an image that has the dimensions that we calculated above exactly. Notice that there is a little bit of extra space on the outer left and right edges. This is the part of the figure that was trimmed.

```
[8]: fig
```



Keep the bounding box tight

It's rare that you would need to change this setting, as a 'tight' bounding box is nearly always what you want. In fact, if you have plotting objects that are outside of the dimensions of the figure, then a tight bounding box will include them. The only reason we turned off this setting is to show the exact dimensions of our figure.

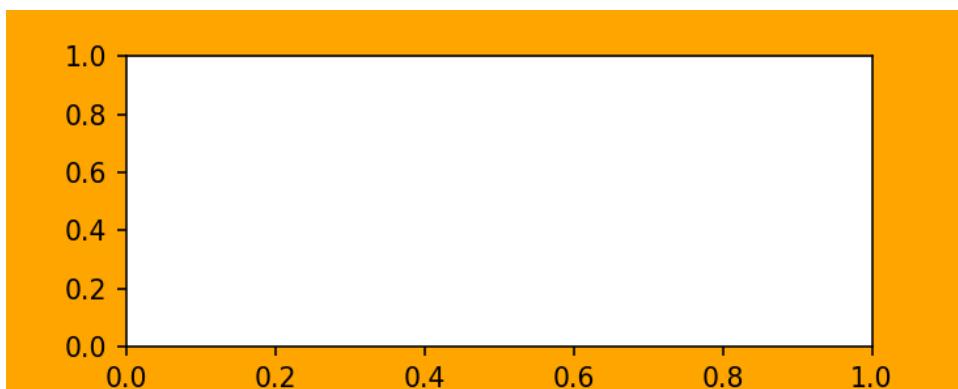
Finding your computer's resolution

When finding your computer's resolution, take care to use the correct dimensions. The actual number of pixels versus the number of pixels that the screen appears to be might differ. For instance, the actual dimensions in pixels for my retina display screen is 2,880 by 1,800 but get scaled down to 1,920 by 1,200.

62.2 Creating figures with custom DPI

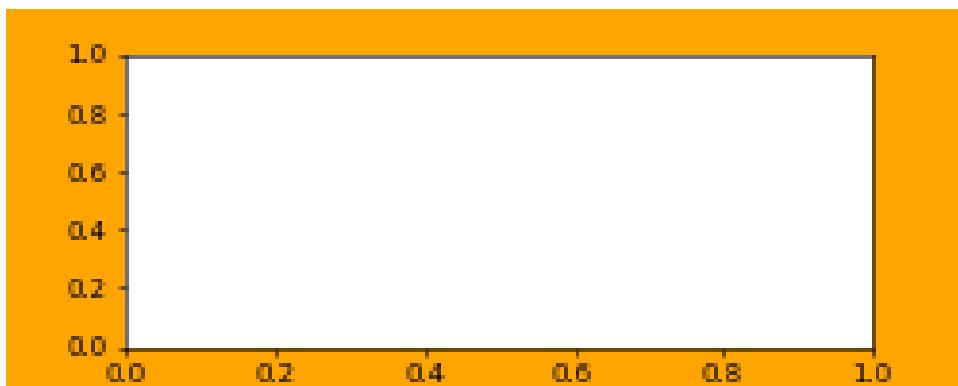
You can create a figure with a specific DPI by setting the `dpi` parameter in the `subplots` function. A new figure with the same dimensions of 5 x 2 figure-inches is created below, but uses my screen's DPI of 147. This should create a figure that has the same figure and screen inches on my screen.

```
[9]: fig, ax = plt.subplots(figsize=(5, 2), facecolor='orange', dpi=147)
```



Using the ruler application (not shown) verifies that the dimensions in screen inches is 5 x 2. Using a smaller DPI than the default of 72 results in figure with even smaller screen-inches. Here we use a DPI of 50, which creates an image with pixel dimensions of 250 by 100 and noticeably blurrier tick labels. Dividing both of these by the screen DPI of 147 gives us the size of 1.7 by .7 screen-inches.

```
[10]: fig, ax = plt.subplots(figsize=(5, 2), facecolor='orange', dpi=50)
```



When the figure is wider than the output area

If you create a figure with a width in screen-inches greater than the width of your output area in a Jupyter Notebook, then it will be scaled down so that all of it is visible. Below, we create a figure that has dimensions 20 x 2 figure-inches. The actual number of horizontal screen-inches of my output area is around 8 inches, so any figure larger than that will get scaled down. Notice that the size of the tick labels are significantly smaller. We’ll find out why in the next section.

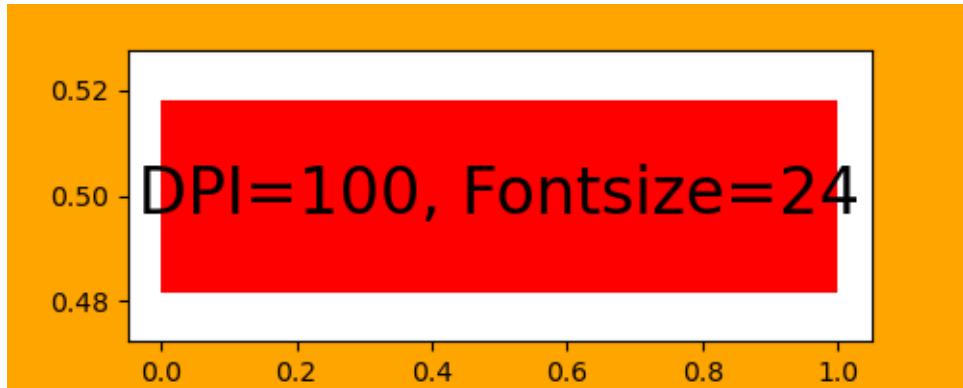
```
[11]: fig, ax = plt.subplots(figsize=(20, 2), facecolor='orange', dpi=147)
```



62.3 Text and line “points”

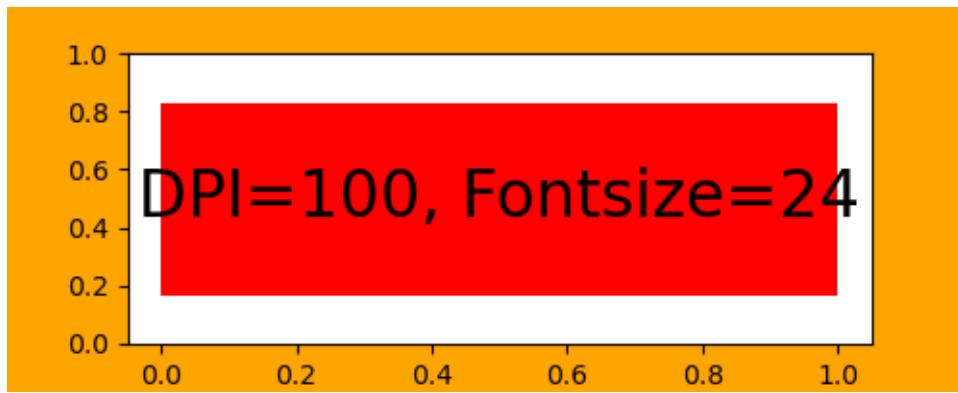
The size of matplotlib text and the width of lines are given in **typographical points**, where one point always equals 1/72nd of a figure-inch. These points have a completely different measurement unit than DPI. They always have the same measure of 1/72nd of a figure-inch. This means that regardless of the DPI, both text and line width will use the same relative amount of space in a figure. Let’s begin by creating a figure with a DPI of 100, then add a horizontal line with width of 72 points, and some text with a font size of 24 points.

```
[12]: fig1, ax1 = plt.subplots(figsize=(5, 2), facecolor='orange', dpi=100)
ax1.hlines(.5, 0, 1, lw=72, color='red')
ax1.text(.5, .5, 'DPI=100, Fontsize=24', ha='center', va='center', fontsize=24);
```



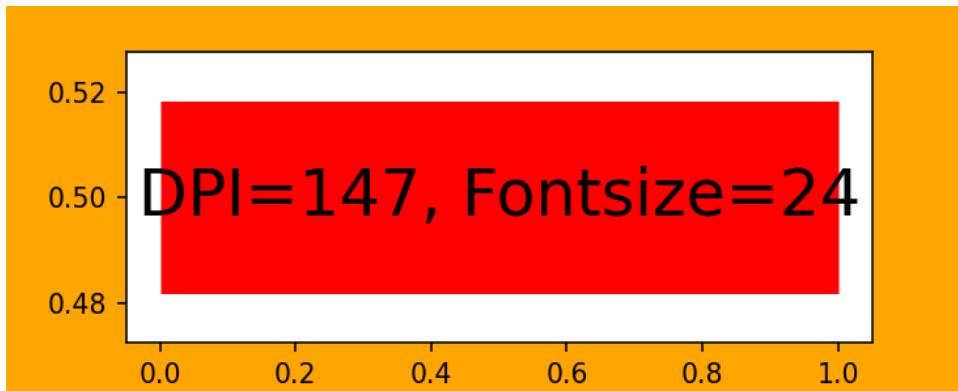
The line has a width of 72 points, which translates to 1 ($72/72$) figure inch. The height of the figure is 2 figure inches, so the line width should be around half the height of the entire figure. From visual inspection, that appears to be the case. Take a look at the limits of the y-axis. They’ve narrowed to just the area where the line was plotted. The range of the x and y axis have no bearing on the text size or line width. They will always take up the same amount of figure-inches. Let’s change the limits of the y-axis to verify this.

```
[13]: ax1.set_ylim(0, 1)
fig1
```



The text has a font size of 24 points or one-third ($24/72$) of a figure inch. As the line is 1 figure inch in width, it does appear that the text height is about $1/3$ of it. Let's create the same figure, but use the DPI for my screen, keeping the width of the line and font size the same.

```
[14]: fig2, ax2 = plt.subplots(figsize=(5, 2), facecolor='orange', dpi=147)
ax2.hlines(.5, 0, 1, lw=72, color='red')
ax2.text(.5, .5, 'DPI=147, Fontsize=24', ha='center', va='center', fontsize=24);
```



Using a different DPI will not change the relative sizes of the line width and text size because they are always 72 points in one figure-inch. The line width of 72 points translates again to 1 figure-inch which is half of the figure height. The text is still 24 points or one-third of a figure-inch.

62.4 Run configuration settings

There are many default run configuration settings that matplotlib has set immediately when you import pyplot in a Jupyter Notebook, some of which are listed below.

- Figure size - 6 inches by 4 inches
- DPI - 72
- font size - 10 points
- line width - 1.5 points
- Axes edge color - black
- Axes face color - white

There are around 300 different run configuration settings which are stored as a dictionary in the `rcParams` variable found in the `pyplot` module. The 'rc' stands for 'run configuration'. Let's retrieve the above values from that dictionary.

```
[15]: plt.rcParams['figure.figsize']
```

```
[15]: [6.0, 4.0]
```

```
[16]: plt.rcParams['figure.dpi']
```

```
[16]: 72.0
```

```
[17]: plt.rcParams['font.size']
```

```
[17]: 10.0
```

```
[18]: plt.rcParams['lines.linewidth']
```

```
[18]: 1.5
```

```
[19]: plt.rcParams['axes.edgecolor']
```

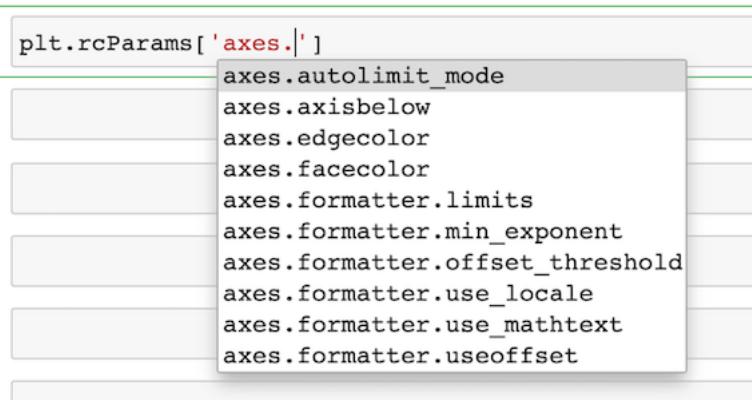
```
[19]: 'black'
```

```
[20]: plt.rcParams['axes.facecolor']
```

```
[20]: 'white'
```

Finding settings

You can output all the possible settings by placing `plt.rcParams` in a cell and executing it. A dictionary-type will be returned mapping the setting name to its value. You'll notice all setting names have a group name followed by a dot and then the specific property. The most likely groups that you will set are axes, figure, legend, xtick, and ytick. If you are running a Jupyter Notebook, you can begin by typing out the group name and then press tab to reveal all the possible settings for that group.



Changing the run configuration settings

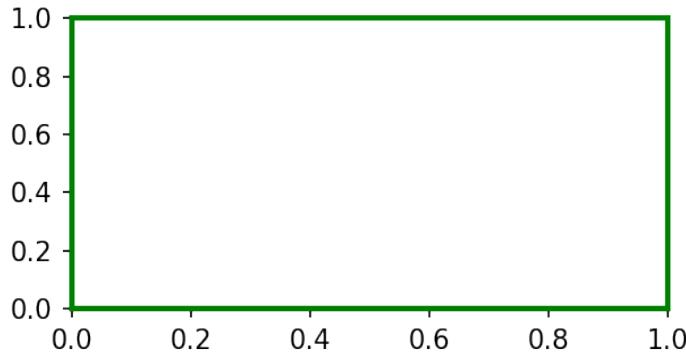
If you find yourself constantly changing a property of a particular matplotlib plotting function to the same value, you might want to set that value as the default by changing the value in the `rcParams` dictionary. To make the change, overwrite the previous value with an assignment statement. Below we change the default values for figure size, dpi, and axes edge color and line width. The axes edge color and line width set the properties for the rectangle that forms the bounds of the axes.

```
[21]: plt.rcParams['figure.figsize'] = (4, 2)
plt.rcParams['figure.dpi'] = 147
```

```
plt.rcParams['axes.edgecolor'] = 'green'  
plt.rcParams['axes.linewidth'] = 2
```

Any figure and axes we create after making these changes will have those new settings. Let's run the `subplots` function to see the changes.

```
[22]: fig, ax = plt.subplots()
```



Relative size of tick labels

The default size of many of the text properties are given as relative values. For instance, both the x and y tick labels have the size '`medium`', while the axes title size is '`large`'. We verify both values below.

```
[23]: plt.rcParams['xtick.labelsize']
```

```
[23]: 'medium'
```

```
[24]: plt.rcParams['axes.titlesize']
```

```
[24]: 'large'
```

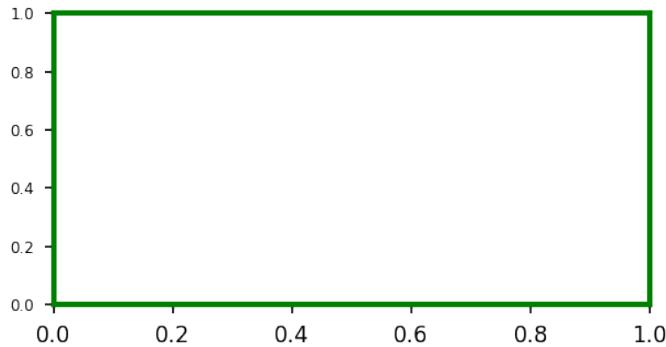
You can change these settings to one of these values: '`xx-small`', '`x-small`', '`small`', '`medium`', '`large`', '`x-large`', or '`xx-large`'. Each of these sizes corresponds to a particular integer font size in points relative to '`medium`', which is set to the default font size of 10 points. Let's verify the font size value.

```
[25]: plt.rcParams['font.size']
```

```
[25]: 10.0
```

Let's change the x-tick and y-tick label sizes to different relative values and create a new figure.

```
[26]: plt.rcParams['xtick.labelsize'] = 'small'  
plt.rcParams['ytick.labelsize'] = 'xx-small'  
fig, ax = plt.subplots()
```



We can get the exact size of the labels by retrieving one from each axis. This returns a text object which has a `get_fontsize` method.

```
[27]: ax.get_xticklabels()[0].get_fontsize()
```

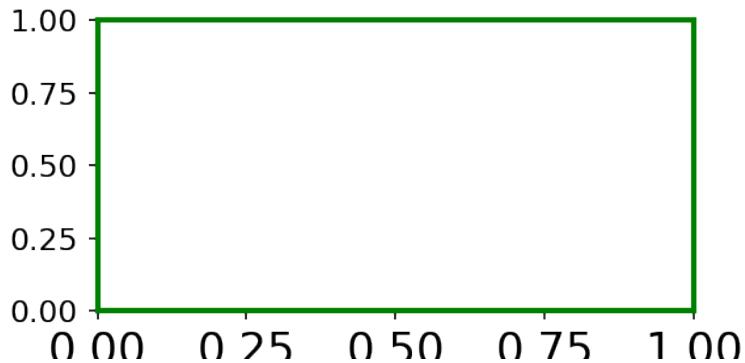
```
[27]: 8.33
```

```
[28]: ax.get_yticklabels()[0].get_fontsize()
```

```
[28]: 5.789999999999999
```

Changing the font size will affect any setting that has a relative size. Both the x-tick and y-tick label size are changed below by changing the font size.

```
[29]: plt.rcParams['font.size'] = 20
fig, ax = plt.subplots()
```



Reset run configuration settings

The changes you make to the run configuration settings only last while the notebook is still running. They will return as their defaults for any new notebook or script you create. To reset all the configuration settings in the current notebook to their default values, call the `rcdefaults` function from `pyplot`.

```
[30]: plt.rcdefaults()
```

There are actually a few differences between these default settings and the ones seen when working in a Jupyter Notebook. Let's retrieve the current default settings for figure size and dpi.

```
[31]: plt.rcParams['figure.figsize']
```

```
[31]: [6.4, 4.8]
```

```
[32]: plt.rcParams['figure.dpi']
```

```
[32]: 100.0
```

When `pyplot` is first imported, a few changes are made to the default configuration settings. We can see exactly what gets changed with the following magic command.

```
[33]: %config InlineBackend.rc
```

```
[33]: {'figure.figsize': (6.0, 4.0),
       'figure.facecolor': (1, 1, 1, 0),
       'figure.edgecolor': (1, 1, 1, 0),
       'font.size': 10,
       'figure.dpi': 72,
       'figure.subplot.bottom': 0.125}
```

If we run the magic command `%matplotlib inline` the settings will return to what they were when we first imported `pyplot`.

```
[34]: %matplotlib inline
plt.rcParams['figure.figsize']
```

```
[34]: [6.0, 4.0]
```

```
[35]: plt.rcParams['figure.dpi']
```

```
[35]: 72.0
```

62.5 Creating style sheets

Instead of loading each of the run configuration settings at the top of the each notebook each time, you can save them to a text file and then load that file. In this file, pair the configuration setting with the desired value, separating them by a colon. The file is NOT a python script, but simply key value pairs. Lines beginning with the hash symbol, `#`, are comments. A simple style sheet is available in the top directory of this book and is titled '`mdap.mplstyle`'. Let's print out all of its contents to the screen.

```
[36]: print(open('..../mdap.mplstyle').read())
```

```
# matplotlib stylesheet for master data analysis with python
figure.dpi: 147
figure.figsize: 4, 2
font.size: 7
```

To load these settings, call the `plt.style.use` function, passing it the location of the style file.

```
[37]: plt.style.use('..../mdap.mplstyle')
```

Let's verify that the settings have once again changed.

```
[38]: plt.rcParams['figure.dpi']
```

```
[38]: 147.0
```

```
[39]: plt.rcParams['font.size']
```

```
[39]: 7.0
```

Using pre-made style sheets

matplotlib comes with several pre-made style sheets that you can use at any time. Retrieve all the available style sheet names as a list with `plt.style.available`. The first five are displayed below.

```
[40]: plt.style.available[:5]
```

```
[40]: ['Solarize_Light2', '_classic_test_patch', 'abc', 'bmh', 'classic']
```

One of the more popular styles is ‘ggplot’, which is based on an R graphing package with the same name. The configuration settings for each style are found in the `plt.style.library` dictionary. Let’s retrieve a few of them for ggplot.

```
[41]: list(plt.style.library['ggplot'].items())[:5]
```

```
[41]: [('axes.axisbelow', True),  
       ('axes.edgecolor', 'white'),  
       ('axes.facecolor', '#E5E5E5'),  
       ('axes.grid', True),  
       ('axes.labelcolor', '#555555')]
```

To choose a style, pass its name to the `plt.style.use` function.

```
[42]: plt.style.use('ggplot')
```

This command overwrites all of the settings present in that style sheet, but keeps any settings not specifically given. For instance, the face color of the axes is changed, but not the figure size.

```
[43]: plt.rcParams['axes.facecolor']
```

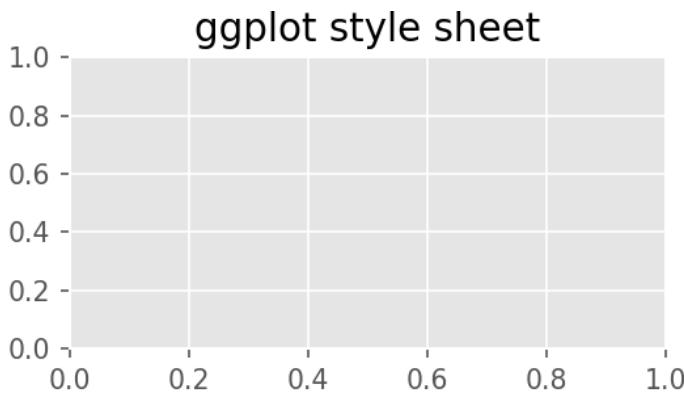
```
[43]: '#E5E5E5'
```

```
[44]: plt.rcParams['figure.figsize']
```

```
[44]: [4.0, 2.0]
```

Let’s create a new figure and axes using our new style based on ggplot.

```
[45]: fig, ax = plt.subplots()  
ax.set_title('ggplot style sheet');
```



Make your style sheet available at all times

Although the file '`mdap.mplstyle`' can be set by calling `plt.style.use`, it isn't widely available for all other projects. matplotlib has a specific directory location where you can save styles to use for any of your work. The helper function `get_configdir` retrieves the directory location.

```
[46]: import matplotlib  
matplotlib.get_configdir()
```



```
[46]: '/Users/Ted/.matplotlib'
```

Within this directory, you must create another directory titled `stylelib` and then place your style sheet in there making sure the name ends in '`.mplstyle`'. After saving the file, you'll be able to call `plt.style.use` from any notebook or script to use that particular style without the file extension. For instance, if you do place the file '`mdap.mplstyle`' in that directory, you can use it by calling `plt.style.use('mdap')`.

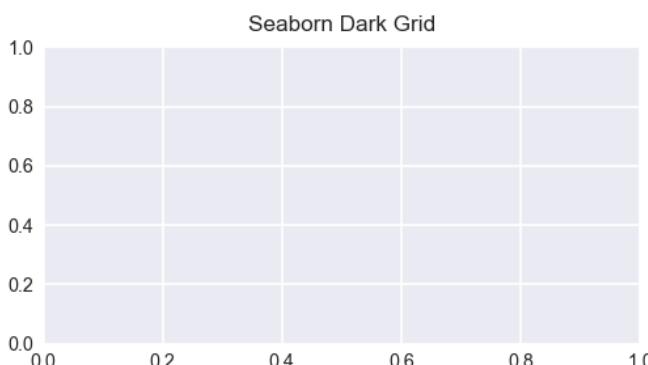
Combining multiple style sheets

It's possible to combine multiple style sheets by passing their names as a list to `plt.style.use`. Each style will change the settings beginning from the left overwriting any previous setting. Perhaps the best use-case is when setting a new style after already changing styles. For instance, our current style is `ggplot`. But if we want to change to `seaborn-darkgrid`, its best to reset to the default, and then use the new style, and then possibly to our book style.

```
[47]: plt.style.use(['default', 'seaborn-darkgrid', '../..../mdap.mplstyle'])
```



```
[48]: fig, ax = plt.subplots()  
ax.set_title('Seaborn Dark Grid');
```



62.6 Exercises

Create a new figure and axes for each exercise.

Exercise 1

Find your screen's DPI and verify it by creating a matplotlib figure with that DPI. Measure the figure with a ruler to verify that the figure-inches match the screen-inches. Set the '`bbox_inches`' notebook setting to `None` and then back to `'tight'` after the exercise.

[]:

Exercise 2

If you create a figure that has a height of 3 inches and a DPI of 120 and add a line that has a width of 144 points, what fraction of the screen height will the line represent?

[]:

Exercise 3

Iterate through all the available styles creating a figure and axes with each one. Put the name of the style in the axes title.

[]:

Exercise 4

Create your own style sheet. You might want to begin by copying one of the pre-made stylesheets.

[]:

Chapter 63

Matplotlib Patches and Colors

In this chapter, we'll add a variety of matplotlib patches to our axes and understand how colors work and how they are chosen. A matplotlib patch is one of several shapes that can be added to an axes. Circles, ellipses, arcs, wedges, rectangles, polygons with any number of sides (triangles, pentagons, hexagons, etc...), and arrows are all examples of matplotlib patches.

63.1 Adding matplotlib patches

All matplotlib patches are located in the `patches` module. We cannot directly create them from our axes object like we did with text, annotation, and horizontal and vertical lines. We'll need to import both the `pyplot` and `patches` module. The style for this book will be used for the remainder of the matplotlib chapters.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib import patches
plt.style.use('.../.../mdap.mplstyle')
```

63.2 Circle patches

All patches must be created using their constructor from the `patches` module. The `Circle` constructor has the following two primary parameters:

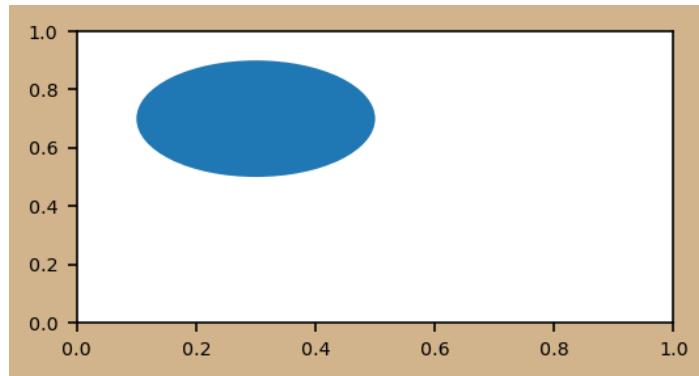
- `xy` - a two-item tuple of the location of the center of the circle
- `radius` - the radius of the circle

Let's instantiate a `Circle` patch and assign the result to a variable.

```
[2]: circle = patches.Circle((.3, .7), radius=.2)
```

Creating a patch does NOT add it to the axes, which we haven't even created. We must pass the patch to the axes `add_patch` method. Let's create our figure and axes, color in the face of the Figure and then add our circle patch, which as a center of (.3, .6). All matplotlib figures have default x and y limits of (0, 1).

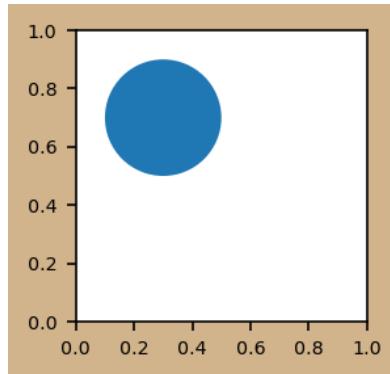
```
[3]: fig, ax = plt.subplots(facecolor='tan')
ax.add_patch(circle);
```



That's not a circle

Our patch looks like an ellipse and not a circle. This is because the limits for both the x and y axis are the same, but the physical dimensions of our figure (4×2) are not. The x-axis is twice as long as the y-axis. matplotlib offers a direct way to change the proportions of our axes so that the x and y dimensions use the same number of pixels for the same distance. Pass the `set_aspect` method the string '`equal`' and it will automatically make the adjustment for you.

```
[4]: ax.set_aspect('equal')
fig
```



Where's the rest of our figure?

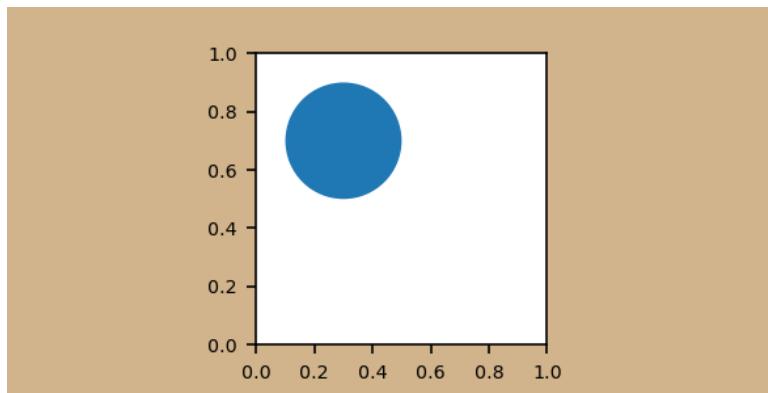
The circle patch now looks like a perfect circle, and the x and y axis appear to be the same length. If you are working through this material in a Jupyter Notebook, you'll notice that the figure does not appear to have the original dimensions (4×2). Let's retrieve these now with one of its getter methods.

```
[5]: fig.get_size_inches()
```

```
[5]: array([4., 2.])
```

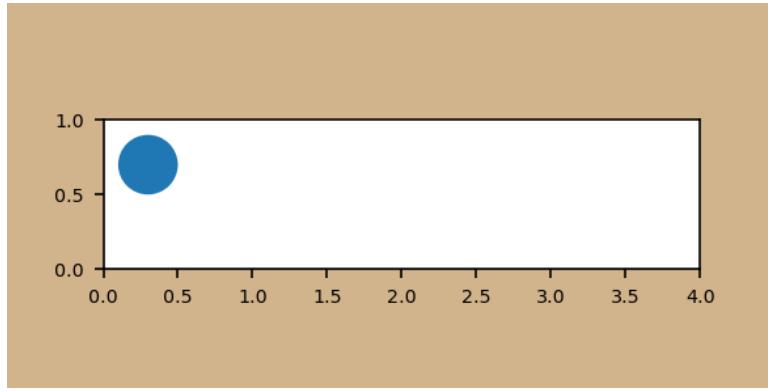
Our figure is still the same size, but Jupyter Notebooks won't display the entire contents if there are no objects in that space. Let's change the inline print settings for `bbox_inches` again to `None` so that the entire figure is displayed.

```
[6]: %config InlineBackend.print_figure_kwarg = {'bbox_inches': None}
fig
```



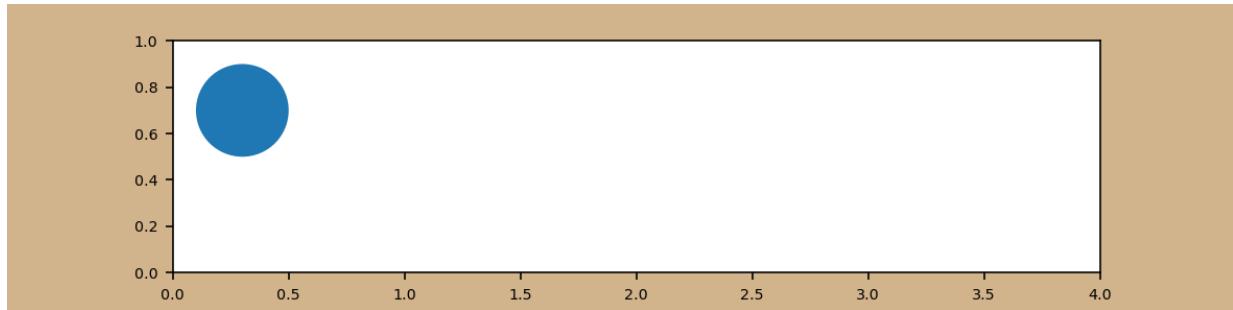
If we change the limits of the x or y axis, the aspect ratio will remain as ‘equal’. Let’s change the x-axis limits to (0, 4).

```
[7]: ax.set_xlim(0, 4)
fig
```



The vertical screen distance of one y unit remains the same as the horizontal screen distance as one x unit. Because our figure size has a width to height ratio of 2 (4 to 2), our current axes (with ratio of 4 to 1) will not fill the entire region of the figure. Let’s change the figure size with the setter method `set_size_inches` so that the ratio is the same as the ratio of x and y axis limits. Any ratio of 4 to 1 will work.

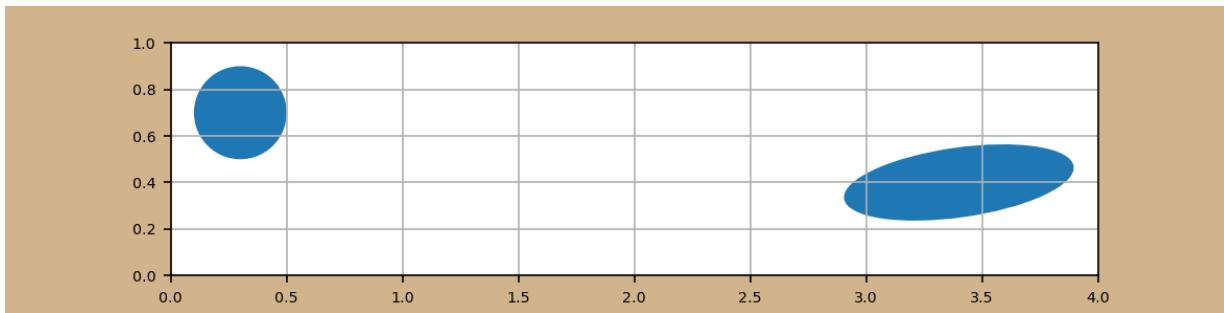
```
[8]: fig.set_size_inches(8, 2)
fig
```



63.3 Ellipse patches

The `Ellipse` constructor creates ellipse patches by providing it a `center` (as a tuple), a `width`, and a `height`. Optionally, you can provide an angle (in degrees) to rotate it counter-clockwise. Grid lines are also added to make it easier to see the locations of each of the patches.

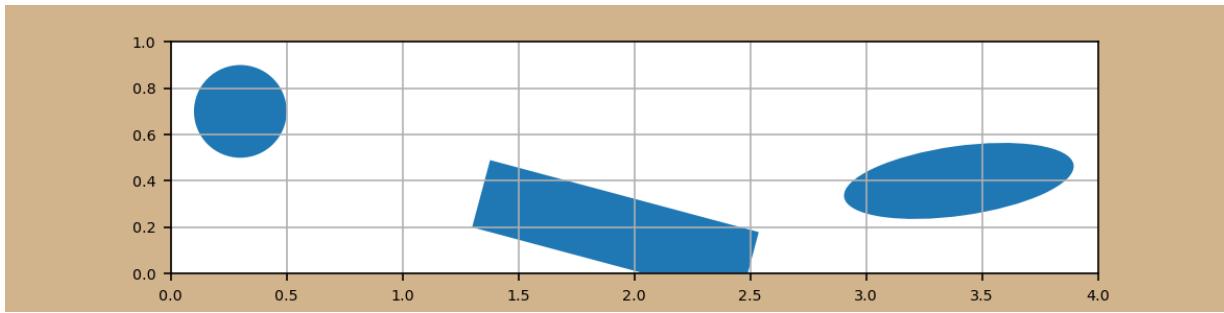
```
[9]: ellipse = patches.Ellipse((3.4, .4), width=1, height=.3, angle=8)
ax.add_patch(ellipse)
ax.grid(True)
fig
```



63.4 Rectangle patches

Rectangle patches are constructed by providing the lower-left coordinate as a tuple, the `width`, and the `height`. Optionally, you can provide an angle (in degrees) to rotate it counter-clockwise. Remember to use the `add_patch` method to actually add the patch to the specified axes.

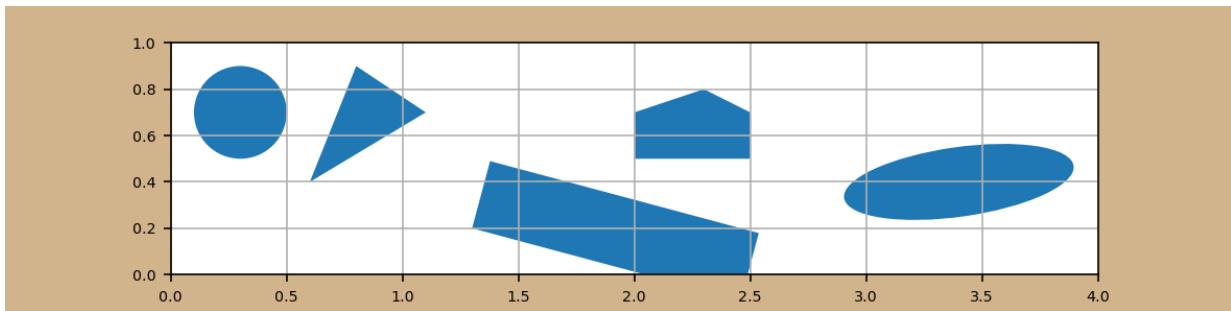
```
[10]: rect = patches.Rectangle((1.3, .2), width=1.2, height=.3, angle=-15)
ax.add_patch(rect)
fig
```



63.5 Polygon patches

The `Polygon` patch constructor allows you to create nearly any shape you want by providing it a list of coordinates of the corners. The points will be connected in the order provided. By default, the first and last points will also be connected. Below, we create a triangle and a pentagon.

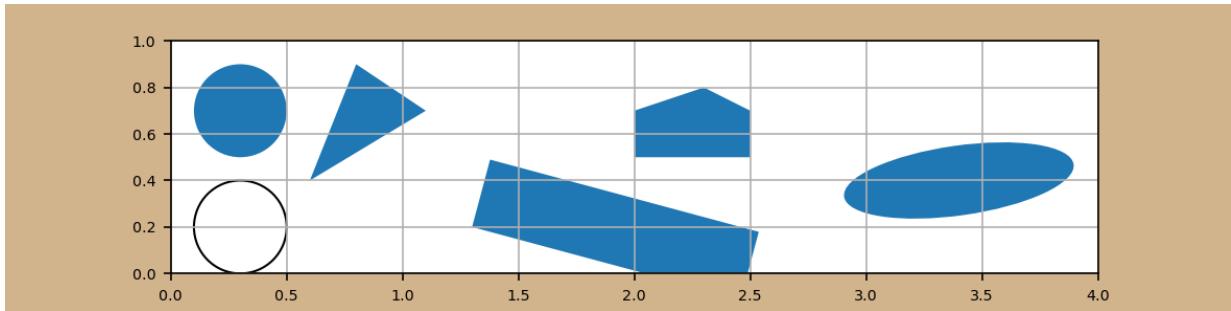
```
[11]: triangle = patches.Polygon([[.8, .9], [.6, .4], [1.1, .7]])
pentagon = patches.Polygon([[2, .5], [2, .7], [2.3, .8], [2.5, .7], [2.5, .5]])
ax.add_patch(triangle)
ax.add_patch(pentagon)
fig
```



63.6 Arc patches

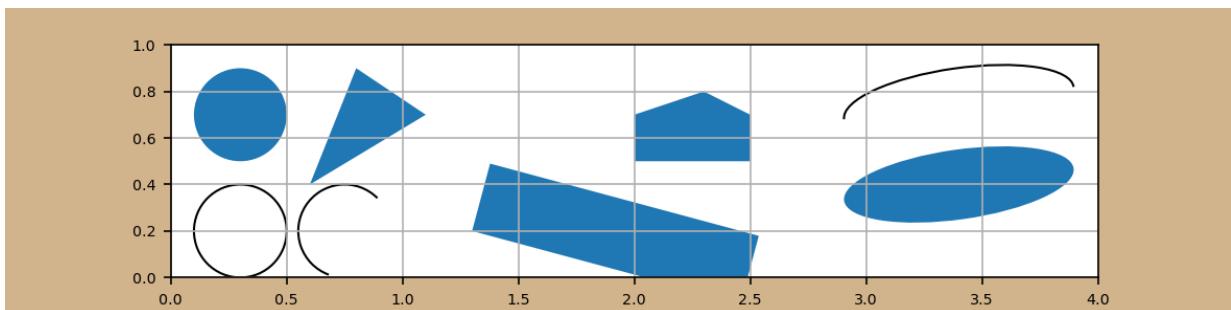
An arc is just the outer edge of an ellipse or circle. It is not filled in with any color. You can make a perfect circle using the `Arc` constructor by setting both the `height` and `width` parameters equal to each other, which will be the diameter (and not the radius) of the circle. The first argument is a tuple of its center. Below, a circle is created with the `Arc` constructor with a diameter of .4.

```
[12]: circle_arc = patches.Arc((.3, .2), width=.4, height=.4)
ax.add_patch(circle_arc)
fig
```



Instead of creating the entire edge of the circle or ellipse, you can draw a segment of it by specifying values for the parameters `theta1` and `theta2` in degrees. These control the start and ending angles. By default, they are set to 0 and 360 respectively, which means that the entire ellipse or circle will be drawn. The 0 degree mark is to the right of center with the degrees increasing in the counter-clockwise direction. Below, we create a segment of a circle, and a segment of an ellipse.

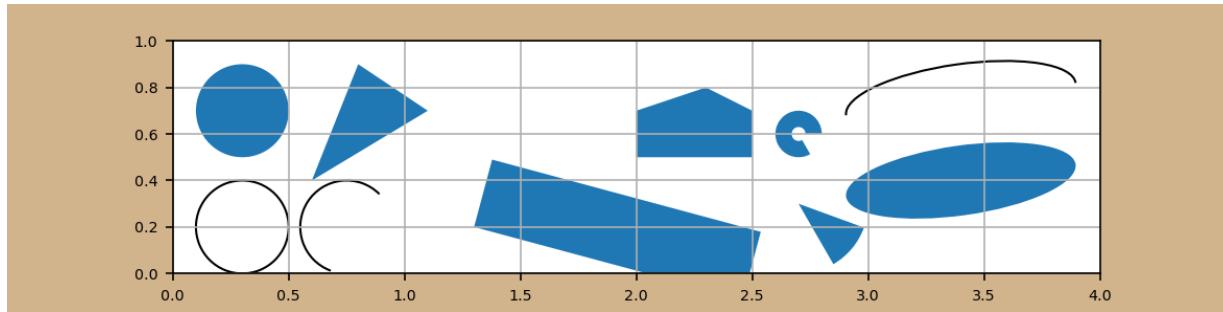
```
[13]: circle_arc_segment = patches.Arc((.75, .2), width=.4, height=.4, theta1=45, theta2=250)
ellipse_arc_segment = patches.Arc((3.4, .75), width=1, height=.3, angle=8,
                                 theta1=0, theta2=180)
ax.add_patch(circle_arc_segment)
ax.add_patch(ellipse_arc_segment)
fig
```



63.7 Wedge patches

A wedge is a section of a circle and looks like a slice of pizza. It is constructed with a center, radius, and start and stop angles. If the start and stop angles are 0 and 360, then a full circle is drawn. Optionally, you can use the `width` parameter to create a circular hole in the middle of the wedge to make it look like a donut. The radius of this hole is the radius minus the width. Here, we create a ‘pizza slice’ wedge and a ‘donut’ wedge with a small hole in it.

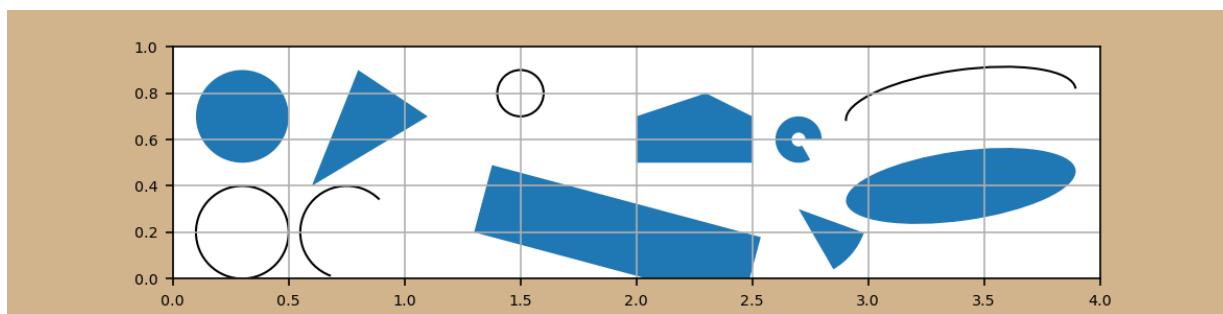
```
[14]: pizza_slice = patches.Wedge((2.7, .3), r=.3, theta1=300, theta2=340)
donut = patches.Wedge((2.7, .6), r=.1, theta1=0, theta2=300, width=.07)
ax.add_patch(pizza_slice)
ax.add_patch(donut)
fig
```



Not filling patches

By default, circle, ellipse, rectangle, polygon, and wedge patches are filled with a color. By setting the `fill` parameter to `False`, only the outline of the patch will be shown. Below, we create a small circle without any fill color. It’s also unnecessary to assign the patch first to a variable. You can place the constructor directly inside the `add_patch` method in a single line of code.

```
[15]: ax.add_patch(patches.Circle((1.5, .8), radius=.1, fill=False))
fig
```

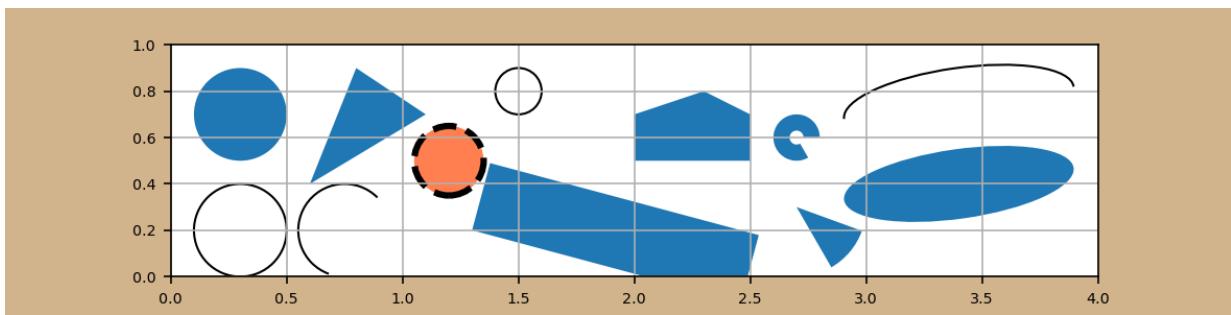


More patch properties

You can control the edge line width, color, and style, and face color with the following parameters, several of which are the same as they are for lines.

- `linewidth` or `lw` - edge width in points
- `edgecolor` or `ec` - edge color
- `linestyle` or `ls` - edge line style
- `facecolor` or `fc` - face color

```
[16]: ax.add_patch(patches.Circle((1.2, .5), radius=.15, lw=3,
                                ec='black', ls='dashed', fc='coral'))
fig
```



List of all patches

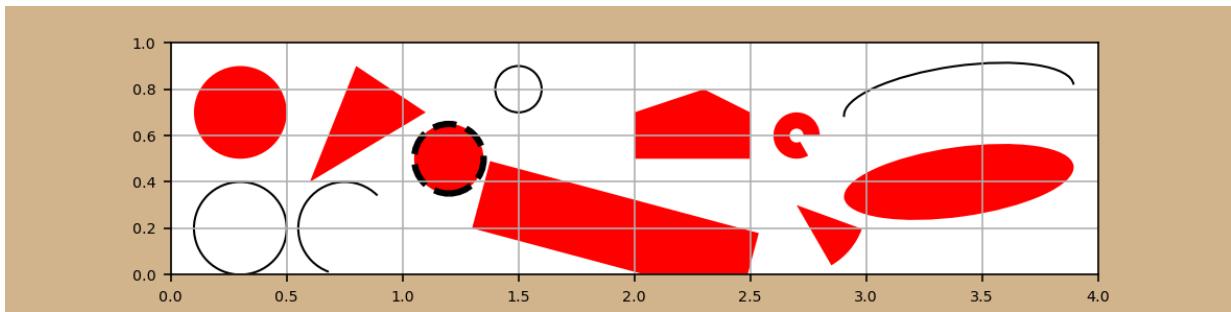
Use the `patches` attribute to retrieve a list of all the patches.

```
[17]: ax.patches
```

```
[17]: [<matplotlib.patches.Circle at 0x11dfc0190>,
        <matplotlib.patches.Ellipse at 0x1200d1d50>,
        <matplotlib.patches.Rectangle at 0x1200d82d0>,
        <matplotlib.patches.Polygon at 0x12035dad0>,
        <matplotlib.patches.Polygon at 0x12035db10>,
        <matplotlib.patches.Arc at 0x120360a50>,
        <matplotlib.patches.Arc at 0x120365890>,
        <matplotlib.patches.Arc at 0x120365150>,
        <matplotlib.patches.Wedge at 0x12035a110>,
        <matplotlib.patches.Wedge at 0x12035d110>,
        <matplotlib.patches.Circle at 0x12035d310>,
        <matplotlib.patches.Circle at 0x120371510>]
```

This list can be useful for iterating through and changing a property that they all have in common. Here, the face color of each patch is changed to red.

```
[18]: for patch in ax.patches:
    patch.set_fc('red')
fig
```



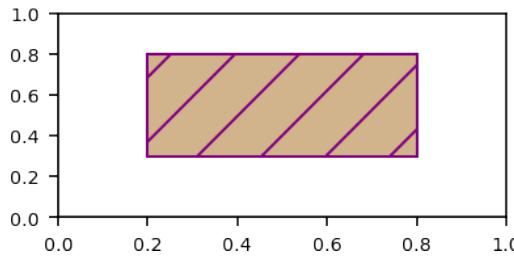
Hatching

Patches can be filled with patterns called ‘hatches’ by setting the `hatch` parameter to one of the following characters.

Symbol	Description
/	diagonal hatching
\	back diagonal
\	vertical
-	horizontal
+	crossed
x	crossed diagonal
o	small circle
O	large circle
.	dots
*	stars

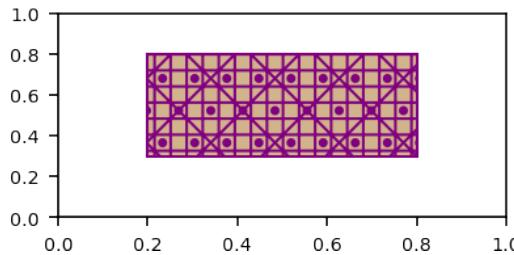
By default, the color of the hatch is black, but this can be set with `edgecolor/ec`. Below, we create a tan rectangle with purple ‘diagnonal hatching’ using the forward slash character.

```
[19]: fig, ax = plt.subplots(figsize=(3, 1.4))
rect = patches.Rectangle((.2, .3), width=.6, height=.5,
                        fc='tan', hatch='/', ec='purple')
ax.add_patch(rect);
```



You can increase the density of the hatches by repeating the same symbol. You can also combine hatching patterns together by using different hatch symbols. We use the `set_hatch` method from the rectangle patch object to change the hatching pattern.

```
[20]: rect.set_hatch('++x.')
fig
```



63.8 Matplotlib colors

We have already changed the colors of a few plotting elements. In this section, we will go deeper into how colors work and how to specify them. Nearly every plotting object has an option for you to change its color. There are two main ways to choose colors:

- Web colors - 140 named colors known to all modern web browsers
- RGB triples - Red, Green, and Blue intensities provided as floats or hexadecimal strings

Web colors

Web colors are a standardized set of 140 colors that all modern web browsers understand. They are also known as CSS4/X11 color names [with more information available here](#). matplotlib provides a dictionary of their names paired to their RGB string value in the `CSS4_COLORS` variable name found in the `colors` module. Let's output the string names of the first five of these colors.

```
[21]: from matplotlib import colors
list(colors.CSS4_COLORS)[:5]
```

```
[21]: ['aliceblue', 'antiquewhite', 'aqua', 'aquamarine', 'azure']
```

To help visualize all of the colors, we'll create one rectangle patch for each color, placing its name as a text object directly in the center of it. There are actually more than 140 names below, as the names 'gray' and 'grey' can be used interchangeably for the same color. For example, 'lightslategray' is the same color as 'lightslategrey'.

A few other things are changed below. The Jupyter Notebook configuration setting for 'bbox_inches' was set back to 'tight' to eliminate the excess padding. The axis labels, tick lines, and tick labels are hidden by passing the string 'off' to the `axis` method.

```
[22]: %config InlineBackend.print_figure_kwarg = {'bbox_inches': 'tight'}
fig, ax = plt.subplots(figsize=(8, 4))
ax.axis('off')
ax.set_xlim(0, 8)
ax.set_ylim(0, 19)
for i, color in enumerate(colors.CSS4_COLORS):
    x, y = divmod(i, 19)
    y = 18 - y
    ax.add_patch(patches.Rectangle((x, y), width=1, height=1, color=color))
    text_color = 'white' if color == 'black' else 'black'
    ax.text(x + .5, y + .5, color, ha='center', va='center',
            color=text_color, fontname='Arial Narrow')
```

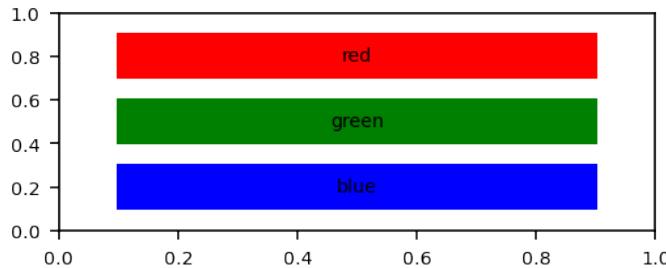
aliceblue	crimson	darkturquoise	honeydew	lightseagreen	mediumvioletred	peru	slategray
antiquewhite	cyan	darkviolet	hotpink	lightskyblue	midnightblue	pink	snow
aqua	darkblue	deeppink	indianred	lightslategray	mintcream	plum	springgreen
aquamarine	darkcyan	deepskyblue	indigo	lightslategrey	mistyrose	powderblue	steelblue
azure	darkgoldenrod	dimgray	ivory	lightsteelblue	moccasin	purple	tan
beige	darkgray	dimgrey	khaki	lightyellow	navajowhite	rebeccapurple	teal
bisque	darkgreen	dodgerblue	lavender	lime	navy	red	thistle
black	darkgrey	firebrick	lavenderblush	limegreen	oldlace	rosybrown	tomato
blanchedalmond	darkkhaki	floralwhite	lawngreen	linen	dive	royalblue	turquoise
blue	darkmagenta	forestgreen	lemonchiffon	magenta	olivedrab	saddlebrown	violet
blueviolet	darkolivegreen	fuchsia	lightblue	maroon	orange	salmon	wheat
brown	darkorange	gainsboro	lightcoral	mediumaquamarine	orangered	sandybrown	white
burlywood	darkorchid	ghostwhite	lightcyan	mediumblue	orchid	seagreen	whitesmoke
cadetblue	darkred	gold	lightgoldenrodyellow	mediumorchid	palegoldenrod	seashell	yellow
chartreuse	darksalmon	goldenrod	lightgray	mediumpurple	palegreen	sienna	yellowgreen
chocolate	darkseagreen	gray	lightgreen	mediumseagreen	paleturquoise	silver	
coral	darkslateblue	green	lightgrey	mediumslateblue	palevioletred	skyblue	
cornflowerblue	darkslategray	greenyellow	lightpink	mediumspringgreen	papayawhip	slateblue	
cornsilk	darkslategrey	grey	lightsalmon	mediumturquoise	peachpuff	slategray	

RGB triples

While the 140 web colors are sufficient for most tasks, many more colors can be described with RGB triples. These stand for the intensity of red, green, and blue components of the color. Each component of the triple may be provided as a float between 0 and 1, where 0 represents the lowest intensity and 1 represents the highest. We'll call these **RGB floats**. Each of the three intensities combines together to form the actual color.

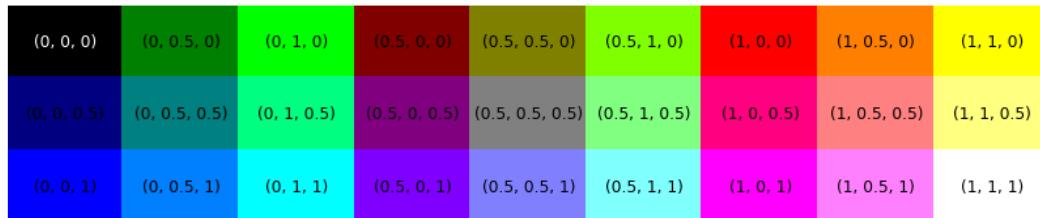
In matplotlib, the triple is provided as a three-item tuple where the intensities are in the order red, green, and blue. For example, `(1, 0, 0)` is an RGB triple for the color red. The triple `(0, 0, 0)` represents an absence of color, which forms black. The triple `(1, 1, 1)` has full intensity for each color, producing white. All float values between 0 and 1 are acceptable. An RGB triple of `(.8, .2, .4)` is valid and produces a color that will visually appear more red, as its intensity is greatest. Below, we create three rectangles using RGB triples (three-item tuples) for red, green, and blue instead of their string name. Note that the actual color green has .5 for the green intensity.

```
[23]: fig, ax = plt.subplots(figsize=(4, 1.5))
ax.add_patch(patches.Rectangle((.1, .7), width=.8, height=.2, color=(1, 0, 0)))
ax.text(.5, .8, 'red', ha='center', va='center')
ax.add_patch(patches.Rectangle((.1, .4), width=.8, height=.2, color=(0, .5, 0)))
ax.text(.5, .5, 'green', ha='center', va='center')
ax.add_patch(patches.Rectangle((.1, .1), width=.8, height=.2, color=(0, 0, 1)))
ax.text(.5, .2, 'blue', ha='center', va='center');
```



Below, we create 27 different RGB triple combinations that have either 0, .5, or 1 intensity for each of the three colors, and plot them as matplotlib rectangle patches.

```
[24]: vals = [0, .5, 1]
rgb_combos = [(r, g, b) for r in vals for g in vals for b in vals]
fig, ax = plt.subplots(figsize=(7, 1.5))
ax.axis('off')
ax.set_xlim(0, 9)
ax.set_ylim(0, 3)
for i, rgb in enumerate(rgb_combos):
    x, y = divmod(i, 3)
    y = 2 - y
    ax.add_patch(patches.Rectangle((x, y), width=1, height=1, color=rgb))
    text_color = 'white' if rgb == (0, 0, 0) else 'black'
    ax.text(x + .5, y + .5, rgb, ha='center', va='center', color=text_color, fontsize=6)
```



RGB hexadecimal strings

Alternatively, you may supply each of the RGB intensities as integers between 0 and 255. For instance, the triple `(255, 0, 0)` is equivalent to the intensity `(1, 0, 0)`. But, matplotlib does not allow you to use the actual integers in base 10. Instead, you must use base 16 for each integer, providing it with its **hexadecimal** representation. In base 10, the digits 0-9 are allowed in each place of a number. In base 16, the digits 0-15 are allowed in a single place. Instead of using two characters to represent the digits 10 - 15, they are referenced by the letters ‘a’ through ‘f’ in hexadecimal notation.

Hexadecimal examples

Using the 16 characters 0 through ‘f’ is the hexadecimal representation of numbers in base 16. It may be useful to look at some examples of numbers converted from base 10 to base 16 and vice-versa. Take the number 14 in base 10. This number is represented as `e` in base 16 (using hexadecimal representation). Now take the number `4c` in base 16. To convert this number to base 10, begin by taking the first digit `c`, which is 12 in base 10. The second digit, 4, is multiplied by 16 and then added to 12. This becomes $64 + 12$ or 76. The generic formula for converting a number in base 16 to base 10 is as follows:

- Start with the right-most digit and convert it to base 10
- Multiply it by 16 raised to the $n - 1$ power, where n is the place of the digit
- Move one digit to the left
- Repeat the procedure and sum all of the values calculated together

Conversion can be done with the built-in `int` constructor by passing it the hexadecimal string and setting the `base` parameter to 16.

```
[25]: int('4c', base=16)
```

```
[25]: 76
```

To convert from base 10 to base 16, use the built-in `hex` function. It returns the hexadecimal representation preceded by the characters ‘0x’. The highest integer intensity for any one color is 255, which has a hexadecimal representation of ‘ff’. Let’s verify this with the `hex` function.

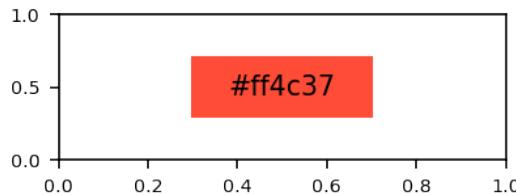
```
[26]: hex(255)
```

```
[26]: '0xff'
```

In matplotlib, the RGB hexadecimal triples are provided as a single string where each of the three intensities always uses two characters. For example, ‘00’ is used for intensity 0, ‘0f’ for 15, and ‘4c’ for 76. matplotlib requires that you place a '#' character at the beginning of the string. The three intensities follow one after the other. For example, ‘#ff4c37’ has intensity ‘ff’ for red, ‘4c’ for green and ‘37’ for blue. We’ll call these **RGB strings**. Let’s create a rectangle patch with this exact color.

```
[27]: fig, ax = plt.subplots(figsize=(3, 1))
ax.add_patch(patches.Rectangle((.3, .3), width=.4, height=.4, color='#ff4c37'))
```

```
ax.text(.5, .5, '#ff4c37', ha='center', va='center', fontsize=10);
```



Why use RGB strings?

The RGB float values are probably a more intuitive way to think about the intensity of a particular color than the RGB string. For instance, a red intensity of .8 is clearer than 'cc' (204 in base 10), its hexadecimal representation. All of the web colors have a standard hexadecimal representation and you will often see this representation used for the colors in other applications. matplotlib stores this representation as the values in the `CSS4_COLORS` dictionary. Let's retrieve the RGB string for a few of the named colors.

```
[28]: colors.CSS4_COLORS['red']
```

```
[28]: '#FF0000'
```

```
[29]: colors.CSS4_COLORS['deeppink']
```

```
[29]: '#FF1493'
```

```
[30]: colors.CSS4_COLORS['olivedrab']
```

```
[30]: '#6B8E23'
```

Converting from RGB floats to RGB strings

In the `colors` module, matplotlib provides the helper functions `to_rgb` and `to_hex` to convert between RGB floats and strings. Let's begin by converting from an RGB string to a three-item tuple float.

```
[31]: colors.to_rgb('#6B8E23')
```

```
[31]: (0.4196078431372549, 0.5568627450980392, 0.13725490196078433)
```

We can go the other direction by beginning with an RGB float and convert it to an RGB string. Each number is first multiplied by 255, then rounded to the nearest integer and then converted to a hexadecimal string.

```
[32]: colors.to_hex((.4, .2, .9))
```

```
[32]: '#6633e6'
```

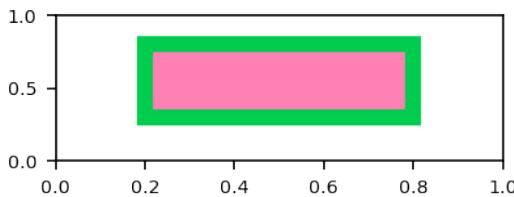
The function can take a named web color and convert it to its respective representation.

```
[33]: colors.to_rgb('seagreen')
```

```
[33]: (0.1803921568627451, 0.5450980392156862, 0.3411764705882353)
```

Let's create a patch with edge and face colors using RGB tuples.

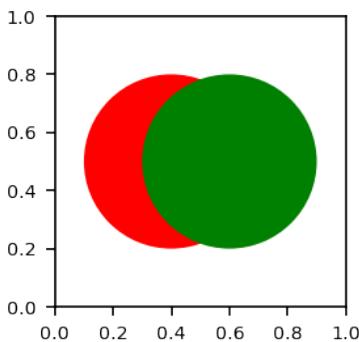
```
[34]: fig, ax = plt.subplots(figsize=(3, 1))
rect = patches.Rectangle((.2, .3), width=.6, height=.5, fc=(1, .5, .7),
                        ec=(0, .8, .3), lw=6)
ax.add_patch(rect);
```



63.9 Color transparency

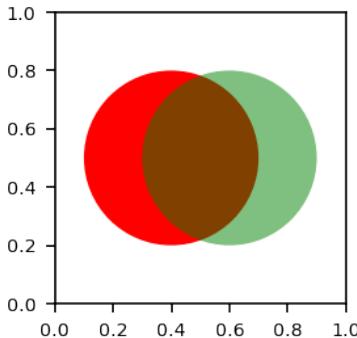
It is possible to control the transparency of each color by setting its `alpha` parameter to a float between 0 and 1, where 0 is completely transparent and 1 is completely opaque. By default, alpha values are set to 1. Two overlapping circle patches are created below.

```
[35]: fig, ax = plt.subplots(figsize=(2, 2))
ax.set_aspect('equal')
left_circle = patches.Circle((.4, .5), radius=.3, fc='red')
right_circle = patches.Circle((.6, .5), radius=.3, fc='green')
ax.add_patch(left_circle)
ax.add_patch(right_circle);
```



By default, the last patch added to the axes takes precedence over which one is visible whenever two or more occupy the same position. Both patches are completely opaque. If we lower the transparency of the green circle, we'll be able to see objects underneath it. Let's set it's alpha value to .5.

```
[36]: right_circle.set_alpha(.5)
fig
```

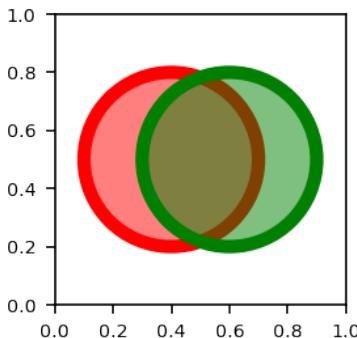


The outline of the red circle is now visible underneath. Notice that the portion of the green circle not overlapping the red circle has become lighter.

RGBA floats

Instead of using the `alpha` parameter directly, you can set colors to four-item tuples consisting of red, blue, green, and alpha values (RGBA floats). Below, we set the face color of each patch to a different level of transparency using an RGBA float. The edge colors are set as opaque.

```
[37]: fig, ax = plt.subplots(figsize=(2, 2))
ax.set_aspect('equal')
left_circle = patches.Circle((.4, .5), radius=.3, fc=(1, 0, 0, .5),
                             ec=(1, 0, 0), lw=5)
right_circle = patches.Circle((.6, .5), radius=.3, fc=(0, .5, 0, .5),
                             ec=(0, .5, 0), lw=5)
ax.add_patch(left_circle)
ax.add_patch(right_circle);
```



Values for alpha can be encoded as two-character hexadecimal strings just like the red, green, and blue values. We use the `to_hex` function again to convert the four-item tuple of the face color of the second circle above to an RGBA string by setting `keep_alpha` to True.

```
[38]: colors.to_hex((0, 1, 0, .3), keep_alpha=True)
```

```
[38]: '#00ff004c'
```

63.10 Layering with zorder

When multiple plotting objects occupy the same space, matplotlib follows special rules to determine the order of how they get layered on top of one another. Every plotting object has a `zorder` property that

is set to a number. The larger this number, the higher its plotting precedence is. Objects with a higher precedence get plotted on top of those with lower precedence. By default, patches have the lowest zorder of 1, followed by lines with 2, and text with 3. Let's verify the zorder of the circle patches from above by retrieving it with a getter method.

```
[39]: left_circle.get_zorder()
```

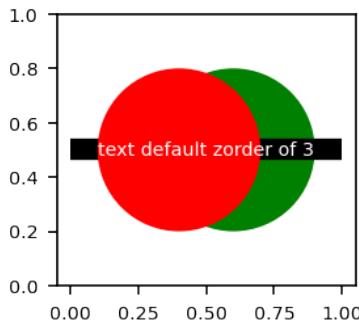
[39]: 1

```
[40]: right_circle.get_zorder()
```

[40]: 1

If two objects have the same zorder, then the object added last to the axes will be shown first. Above, the `right_circle` was added last, and therefore is placed on top of the other patch. In the following plot, the `left_circle` patch is given an explicit zorder of 3, which places it on top of the `right_circle`. A horizontal line is drawn across the entire span of the x-axis keeping its default zorder of 2. It's placed under the `left_circle`, but on top of the `right_circle`. Lastly, some text is added to the axes. By default, it has a zorder of 3, the same as `left_circle`, but because it is added last, it is placed on top.

```
[41]: fig, ax = plt.subplots(figsize=(2, 2))
ax.set_aspect('equal')
left_circle = patches.Circle((.4, .5), radius=.3, fc='red', zorder=3)
right_circle = patches.Circle((.6, .5), radius=.3, fc='green')
ax.add_patch(left_circle)
ax.add_patch(right_circle)
ax.hlines(y=.5, xmin=0, xmax=1, lw=8)
ax.set_ylim(0, 1)
ax.text(.1, .5, 'text default zorder of 3', va='center', color='white');
```

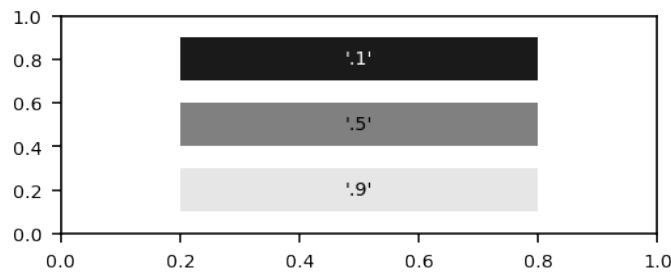


63.11 Gray scale

Besides the named web colors and the RGB triples, matplotlib allows you to use a string of a float value from 0 to 1 to select a shade of gray. Smaller float values correspond to dark shades, with black being the darkest, while larger float values correspond to lighter shades, with white being the lightest. For instance, the string `'.1'` represents a dark gray color. Below, three rectangle patches with varying shades of gray and added to the axes colored with their gray scale string.

```
[42]: fig, ax = plt.subplots(figsize=(4, 1.5))
ax.add_patch(patches.Rectangle((.2, .7), width=.6, height=.2, fc='.1'))
ax.text(.5, .8, "'0.1'", ha='center', va='center', color='white')
```

```
ax.add_patch(patches.Rectangle((.2, .4), width=.6, height=.2, fc='0.5'))
ax.text(.5, .5, "'0.5'", ha='center', va='center')
ax.add_patch(patches.Rectangle((.2, .1), width=.6, height=.2, fc='0.9'))
ax.text(.5, .2, "'0.9'", ha='center', va='center');
```



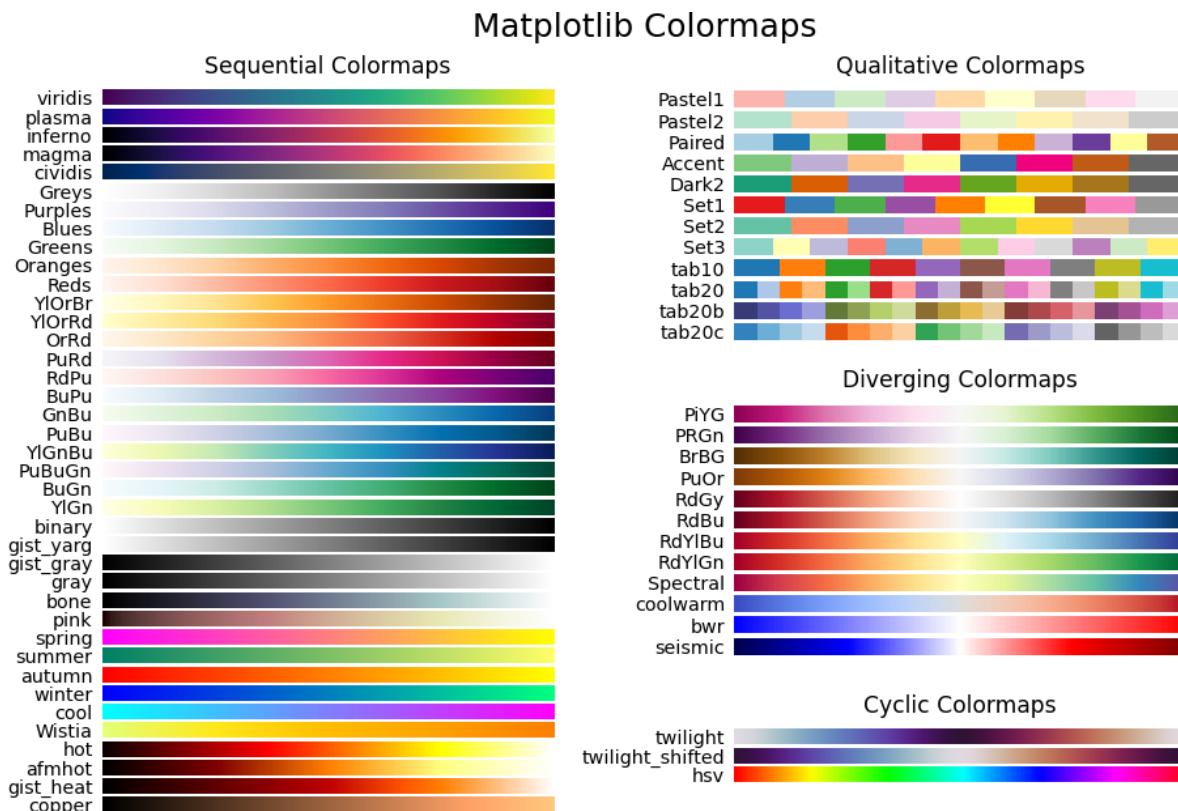
All gray scales have the same intensity for red, green, and blue. You can verify this by converting this string float value to an RGB triple.

[43]: `colors.to_rgb('0.7')`

[43]: `(0.7, 0.7, 0.7)`

63.12 Colormaps

A matplotlib **colormap** is a list of colors grouped together for various purposes. The image below shows most of the available colormaps in matplotlib grouped into categories. [Navigate to this page of the matplotlib documentation](#) to find all of the names.



All color maps can be directly accessed with their name from the `cm` module, which we import below.

```
[44]: from matplotlib import cm
```

Let's access the inferno colormap.

```
[45]: cm.inferno
```

```
[45]: <matplotlib.colors.ListedColormap at 0x11d519b90>
```

All colormaps have the attribute `N` that returns the number of colors it has.

```
[46]: cm.inferno.N
```

```
[46]: 256
```

There are 256 colors in the inferno colormap. Most colormaps also have 256 colors but some have 10 or 20 and a few have more. Each unique color can be accessed as an RGBA float by calling the colormap object as if it were a function and passing it an integer corresponding to one of the colors. The first color of the inferno colormap is returned below.

```
[47]: cm.inferno(0)
```

```
[47]: (0.001462, 0.000466, 0.013866, 1.0)
```

Let's get the 51st color.

```
[48]: cm.inferno(50)
```

```
[48]: (0.25162, 0.037705, 0.403378, 1.0)
```

And now the last color.

```
[49]: cm.inferno(255)
```

```
[49]: (0.988362, 0.998364, 0.644924, 1.0)
```

Integers less than 0 or greater than or equal to `N` are valid and return the first and last colors respectively. The integer -99 references the same RGBA as 0.

```
[50]: cm.inferno(-99)
```

```
[50]: (0.001462, 0.000466, 0.013866, 1.0)
```

The integer 256 references the same RGBA as 255.

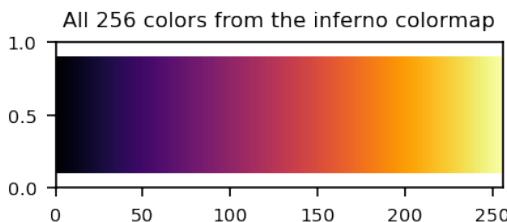
```
[51]: cm.inferno(256)
```

```
[51]: (0.988362, 0.998364, 0.644924, 1.0)
```

Let's visualize the inferno colormap by creating a thin rectangle for each of the 256 colors.

```
[52]: fig, ax = plt.subplots(figsize=(3, 1))
ax.set_title('All 256 colors from the inferno colormap', fontsize=8)
ax.set_xlim(0, 256)
for i in range(256):
```

```
ax.add_patch(patches.Rectangle((i, .1), height=.8, width=2, fc=cm.inferno(i)))
```



You can get a list of all the RGBA values of a colormap by passing it a sequence of all of its integers. Here, we get all colors for the inferno colormap and output the first five values.

```
[53]: inferno_color_list = cm.inferno(range(256))
inferno_color_list[:5]
```

```
[53]: array([[1.4620e-03, 4.6600e-04, 1.3866e-02, 1.0000e+00],
       [2.2670e-03, 1.2700e-03, 1.8570e-02, 1.0000e+00],
       [3.2990e-03, 2.2490e-03, 2.4239e-02, 1.0000e+00],
       [4.5470e-03, 3.3920e-03, 3.0909e-02, 1.0000e+00],
       [6.0060e-03, 4.6920e-03, 3.8558e-02, 1.0000e+00]])
```

Reversed colormaps

Each colormap is available in reverse order and can be accessed by appending '_r' to its name. For instance, 'inferno_r' is the reversed colormap of 'inferno'. Below, we verify that the first and last values of the colormaps are equal.

```
[54]: cm.inferno_r(0) == cm.inferno(255)
```

```
[54]: True
```

```
[55]: cm.inferno_r(255) == cm.inferno(0)
```

```
[55]: True
```

Using floats between 0 and 1

Instead of using the integers from 0 to $N - 1$ to reference each color in the colormap, you can use a float between 0 and 1 instead. These floats represent the relative position of the color in the colormap. For instance, a value of .5 corresponds to the middle of the colormap. Let's get the color for a value of .3.

```
[56]: cm.inferno(.3)
```

```
[56]: (0.416331, 0.090203, 0.432943, 1.0)
```

To make the conversion between decimals and integer, multiply by N and truncate the decimals to get an integer. Multiplying .3 by 256 equals 76.8 and truncating the decimals yields the integer 76. Let's verify it has the same RGBA value.

```
[57]: cm.inferno(76)
```

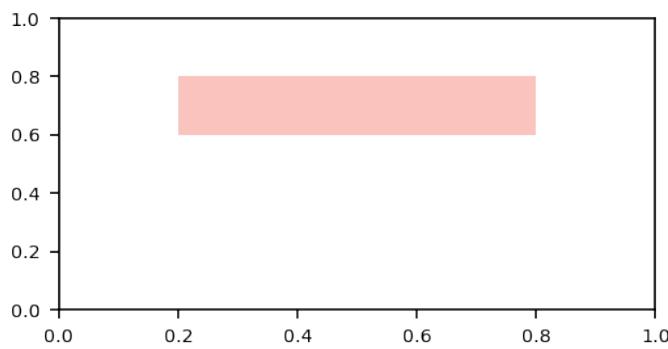
```
[57]: (0.416331, 0.090203, 0.432943, 1.0)
```

Colormaps become important when visualizing data. Only the mechanics of colormaps were presented in this chapter. In upcoming chapters, we will use them to give meaning to our data visualizations.

63.13 Filling between two lines

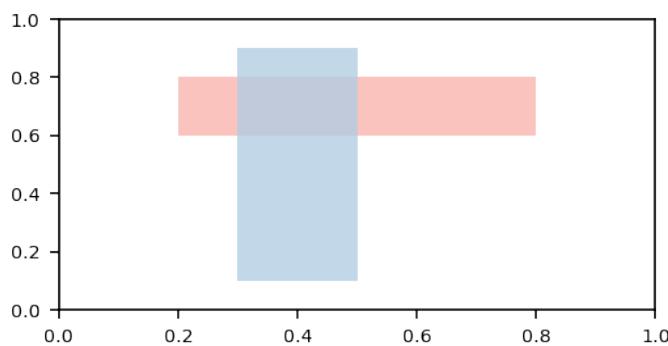
matplotlib provides the ability to fill the vertical area between two lines with a color using the `fill_between` method. At a minimum, you must set `x` as the first and last x-values and then use `y1` and `y2` as a single value for the lower and upper boundaries. Here, the area between the y-values .6 and .8 beginning at an x of .2 ending at .8 is filled with the first color of the Pastel1 colormap.

```
[58]: fig, ax = plt.subplots()
ax.set(xlim=(0, 1), ylim=(0, 1))
ax.fill_between(x=[.2, .8], y1=.6, y2=.8, fc=cm.Pastel1(0), alpha=.8);
```



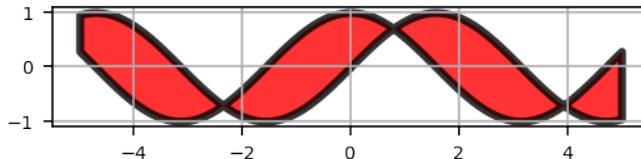
The `fill_betweenx` method fills the horizontal area between two lines in an analogous fashion.

```
[59]: ax.fill_betweenx(y=[.1, .9], x1=.3, x2=.5, fc=cm.Pastel1(1), alpha=.8)
fig
```



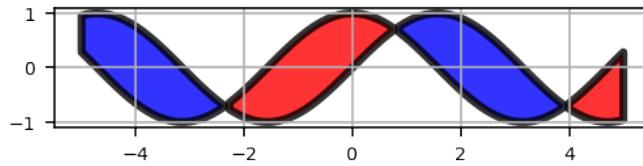
Arrays of equal size can be used to describe more complex lines. Here, the area between the cosine and sine of x-values between -5 and 5 are filled.

```
[60]: x = np.linspace(-5, 5, 100)
y1 = np.cos(x)
y2 = np.sin(x)
fig, ax = plt.subplots(figsize=(4, 1))
ax.set_aspect('equal')
ax.grid(True)
ax.fill_between(x=x, y1=y1, y2=y2, fc='red', alpha=.8, ec='black', linewidth=3);
```



The `where` parameter can be given an array of booleans that correspond to which x-values (or y-values with `fill_betweenx`) will be plotted. The most common case is to set it equal to `y1 > y2` and call `fill_between` a second time setting `where` to be `y1 < y2` to fill in the region with a different color.

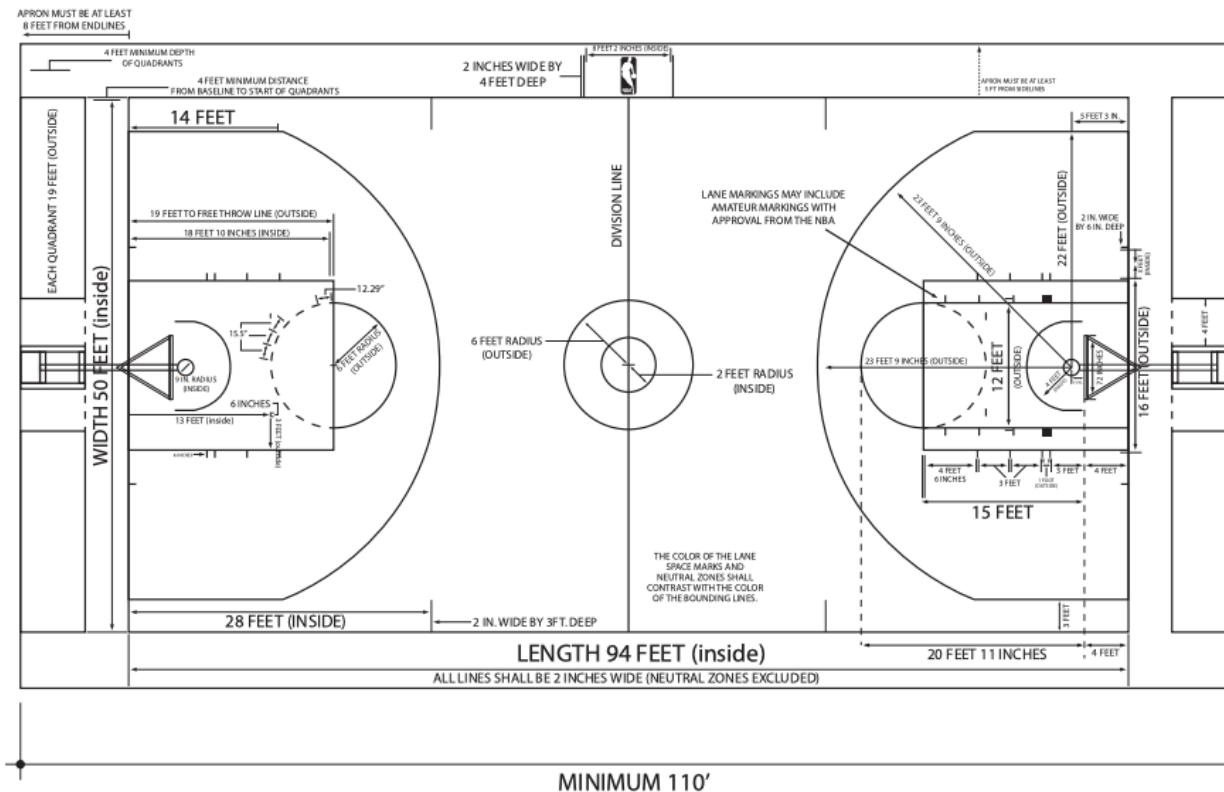
```
[61]: fig, ax = plt.subplots(figsize=(4, 1))
ax.set_aspect('equal')
ax.grid(True)
ax.fill_between(x=x, y1=y1, y2=y2, fc='red', where=y1 > y2,
                alpha=.8, ec='black', linewidth=3)
ax.fill_between(x=x, y1=y1, y2=y2, fc='blue', where=y1 < y2,
                alpha=.8, ec='black', linewidth=3);
```



63.14 Creating a basketball court

To help reinforce some of the material in this chapter, we will reproduce an image of a basketball court using matplotlib patches and horizontal and vertical lines. Take a look at the image below, which is an official image from the National Basketball Association with all the required dimensions of the court. [Navigate to this page](#) to see the image and read more about the court.

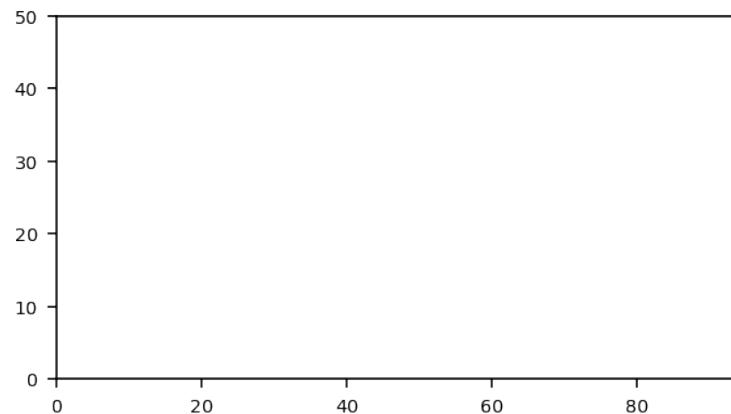
The image contains all of the markings on the court for things such as the playing area, half-court line, three-point line, free-throw line, position of the basket, and more. We will attempt to replicate this image, but leave out the words and anything outside of the actual playing area (marked by ‘LENGTH’ and ‘WIDTH’). I suggest attempting to complete this as an exercise on your own before looking at the code below.



MINIMUM 110'

The outer dimensions of an NBA court are 94 feet by 50 feet. Let's create a figure with a `figsize` similar to the ratio of the length to width of the court (2 to 1). We can then set the x-limits and y-limits so that they match exactly the length and width of the court. Each unit in each direction represents one foot. The aspect ratio is set to `equal` so that the physical distance in pixels along the x and y dimensions is the same.

```
[62]: fig, ax = plt.subplots(figsize=(5, 2.5))
ax.set_xlim(0, 94)
ax.set_ylim(0, 50)
ax.set_aspect('equal')
```



Vertical lines

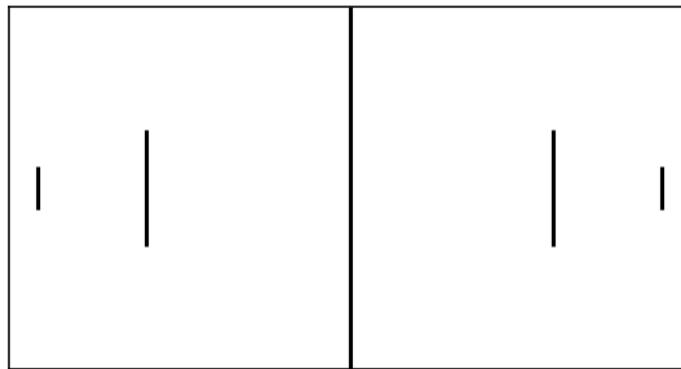
We will draw the following vertical lines using the `vlines` function.

- Half-court line - Runs vertically at center ($x = 47$) of the court
- Free throw lines - 19 feet from edge of court and 16 feet in length

- Backboard - 4 feet from edge of court and 6 feet in length

We also remove the x-axis and y-axis tick marks by setting them to an empty list.

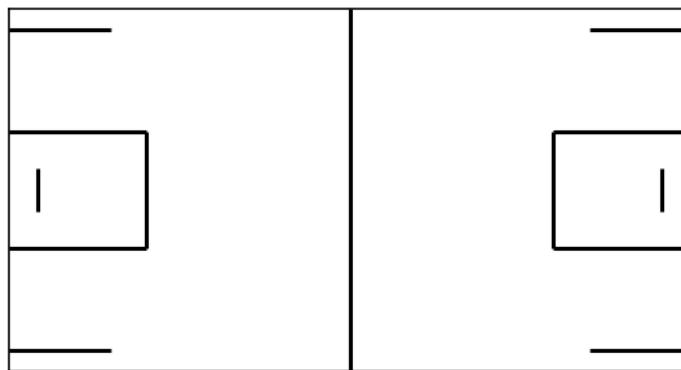
```
[63]: ax.set(xticks=[], yticks=[])
midcourt_line      = ax.vlines(x=47, ymin=0, ymax=50)
free_throw_lines = ax.vlines(x=[19, 75], ymin=17, ymax=33)
backboard         = ax.vlines(x=[4, 90], ymin=22, ymax=28)
fig
```



Horizontal lines

We add horizontal lines for the edges of the free throw area and for the corners of the three-point lines.

```
[64]: left_free_throw_box      = ax.hlines(y=[17, 33], xmin=0, xmax=19)
right_free_throw_box        = ax.hlines(y=[17, 33], xmin=75, xmax=94)
left_corner_three_point_line = ax.hlines(y=[3, 47], xmin=0, xmax=14)
right_corner_three_point_line = ax.hlines(y=[3, 47], xmin=80, xmax=94)
fig
```

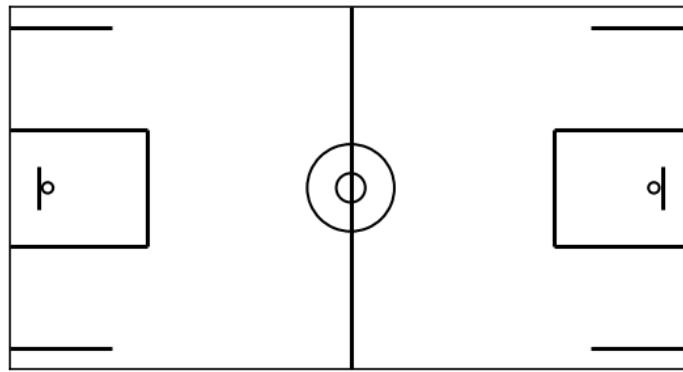


Circles

The basket is a circle with a radius of 9 inches or .75 feet. The two concentric circles at half-court could have been created with a two circle patches, but we opt to use a wedge with a hole in the middle. We don't want to fill these circles with any color, so we set the `fill` parameter to `False`.

```
[65]: left_basket  = patches.Circle((5.25, 25), radius=.75, fill=False)
right_basket = patches.Circle((94 - 5.25, 25), radius=.75, fill=False)
center_wedge = patches.Wedge((47, 25), r=6, theta1=0, theta2=360, width=4, fill=False)
```

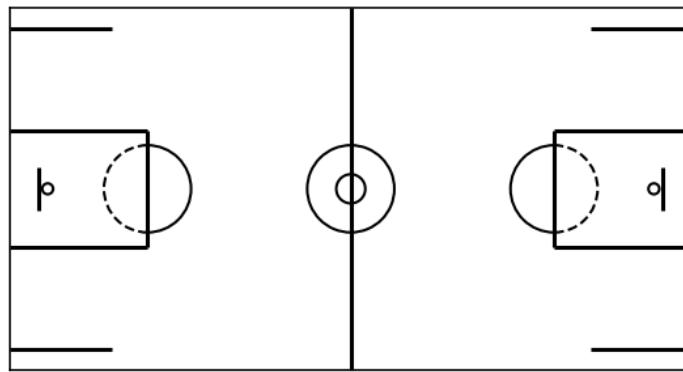
```
ax.add_patch(left_basket)
ax.add_patch(right_basket)
ax.add_patch(center_wedge)
fig
```



Free-throw arcs

There are circles around each free-throw line, but instead of using a `Circle` patch, we'll use an `Arc` patch so that we can make half of it dashed and the other half solid.

```
[66]: left_inner_arc = patches.Arc((19, 25), height=12, width=12,
                                 theta1=90, theta2=270, ls='dashed')
left_outer_arc = patches.Arc((19, 25), height=12, width=12,
                             theta1=270, theta2=90)
right_inner_arc = patches.Arc((75, 25), height=12, width=12,
                               theta1=270, theta2=90, ls='dashed')
right_outer_arc = patches.Arc((75, 25), height=12, width=12,
                              theta1=90, theta2=270)
ax.add_patch(left_inner_arc)
ax.add_patch(left_outer_arc)
ax.add_patch(right_inner_arc)
ax.add_patch(right_outer_arc)
fig
```

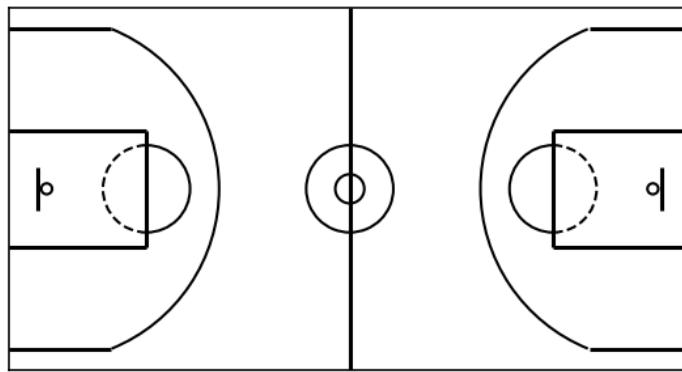


Three-point arcs

The remaining portion of the three-point line is a segment of a circle with radius of 23.75 (diameter of 47.5) feet from the center of the basket. We use an `Arc` patch to draw this, calculating the starting and ending

angles using some basic geometry.

```
[67]: deg = np.rad2deg(np.arccos(22 / 23.75))
left_three_point = patches.Arc((5.25, 25), height=47.5, width=47.5,
                               theta1=270 + deg, theta2=90 - deg)
right_three_point = patches.Arc((94 - 5.25, 25), height=47.5, width=47.5,
                                 theta1=90 + deg, theta2=270 - deg)
ax.add_patch(left_three_point)
ax.add_patch(right_three_point)
fig
```



63.15 Exercises

Exercise 1

Create a simple house with a roof, door, yard, and sun shining overhead with matplotlib patches.

[]:

Exercise 2

Convert the RGB float triple (.7, .03, .8) to a hexadecimal string. Do so without using matplotlib, then verify with one of the functions from the `colors` module.

[]:

Exercise 3

Convert the RGB string '#7cf2bb' to an RGB float triple. Do so without using matplotlib, then verify with one of the functions from the `colors` module.

[]:

Exercise 4

Create a circle that visually looks like a circle that has edge color of lightcoral and face color of forestgreen. Use the hexadecimal representation for these strings. Increase the edge width so that it is more visible.

[]:

Exercise 5

Create two circles of the same size vertically next to one another (same x-value, different y-value) without any overlap. Draw three vertical lines that pass through both circles. Place one of these lines underneath both circles, another on top of the top circle and below the bottom circle, and the last line on top of both circles.

[]:

Exercise 6

Create the flag of Iceland with matplotlib. Search online for its RGB color values.

[]:

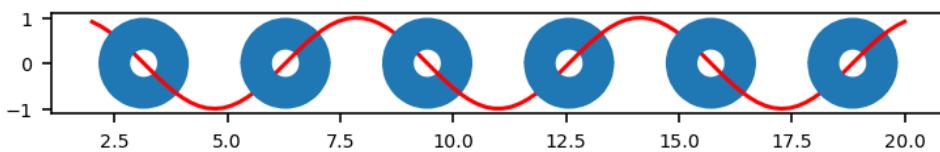
Exercise 7

Create a rainbow. Use a colormap that has the colors of the rainbow.

[]:

Exercise 8

Recreate the following image in matplotlib.



[]:

Chapter 64

Matplotlib Line Plots

In this chapter, we will plot actual data with matplotlib line and scatter plots. Thus far, we have yet to do any data visualization. We've focused on understanding how to create the figure and axes, and how to change its properties using the object-oriented approach. We've also covered how to add straight lines, text, and patches, and how to choose colors for them. None of these plotting objects came from data. In this chapter, we will use our data that we previously analyzed within a pandas DataFrame to make data visualizations with matplotlib.

64.1 Axes API

We'll continue to call methods from the axes object, but concentrate on those that plot data. It can be helpful to navigate to the [axes API page from matplotlib's official documentation](#) which contains around 300 different methods. The API page categorizes and groups each method by its functionality. In this chapter, we focus on the `plot` method from the [plotting section](#).

Plotting methods

All of the plotting methods accept data as input, and add matplotlib objects to the axes. Each of the plotting methods returns the plotting object(s) which we can assign to a variable name.

64.2 Line plots with the `plot` method

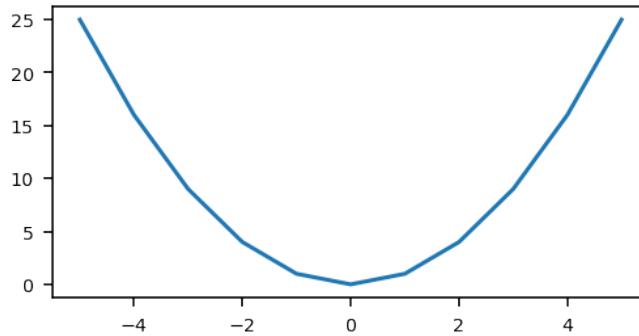
The `plot` method's primary purpose is to create line plots. It does have the ability to create scatter plots as well, but that task is best reserved for the `scatter` method. A more appropriate name for this method would have been `lineplot` as the name `plot` isn't descriptive. You'll have to make the association that `plot` really means line plot. All of the other plotting methods are descriptive of what kind of plot they actually create except this one.

The `plot` method is very flexible and can take a variety of different inputs. Instead of covering all of the possible ways to use the `plot` method, a single, straightforward approach will be given. The first two arguments to the `plot` method are the `x` and `y` coordinates of the data.

Below, a numpy array of the integers from -5 to 5 is created with the `arange` function. These values represent the coordinates of our line. The squared values of this array are used as the `y` coordinates. We call the `plot` method assigning the returned values to `line_plot_objects`. Each `x`-value is paired with its respective `y`-value. The points are connected with straight lines forming the image below.

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

```
import matplotlib.pyplot as plt
plt.style.use('..../mdap.mplstyle')
fig, ax = plt.subplots()
x = np.arange(-5, 6)
y = x ** 2
line_plot_objects = ax.plot(x, y)
```



What was returned?

The variable `line_plot_objects` was assigned the return value from the call to the `plot` method. Let's inspect this object and output its contents to the screen.

```
[2]: line_plot_objects
```

```
[2]: [<matplotlib.lines.Line2D at 0x123102750>]
```

matplotlib returns a list of `Line2D` objects. The `plot` method can produce many lines in a single call to it, which is why it returns the results as a list and not as a single object. Let's verify that this returned object is a list.

```
[3]: type(line_plot_objects)
```

```
[3]: list
```

Let's assign its only item to a variable name and output its type.

```
[4]: line = line_plot_objects[0]
type(line)
```

```
[4]: matplotlib.lines.Line2D
```

We now have access to our line, technically a `Line2D` object. Like all matplotlib objects, it has many properties, which can be accessed and changed with its getter and setter methods. We begin by getting the underlying data, which is returned as a tuple of numpy arrays.

```
[5]: line.get_data()
```

```
[5]: (array([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4,  5]),
      array([25, 16,  9,  4,  1,  0,  1,  4,  9, 16, 25]))
```

The color is retrieved as an RGB string.

```
[6]: line.get_color()
```

```
[6]: '#1f77b4'
```

All lines default to a zorder of 2, which we verify below.

```
[7]: line.get_zorder()
```

```
[7]: 2
```

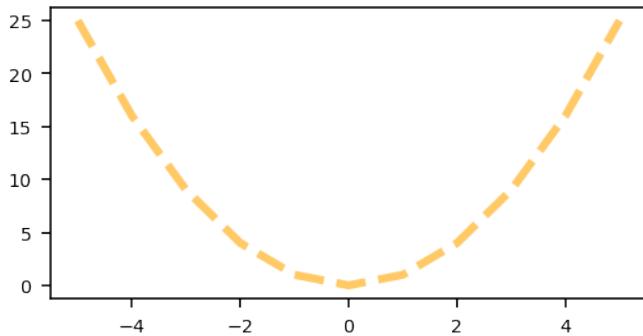
The `Line2D` object is not all that important to understand and you will rarely use it directly to create lines. Instead, you'll be using methods like `plot`, `hlines`, `vlines` and others.

Line properties

The line properties that we've covered in previous chapters remain the same and all work when using the `plot` method.

Property	Possible Values
<code>linewidth</code> or <code>lw</code>	width of line in points
<code>linestyle</code> or <code>ls</code>	'solid' or '-' (default), 'dashed' or '--', 'dotted' or ':', 'dashdot' or '-.'
<code>color</code> or <code>c</code>	line color
<code>alpha</code>	0 to 1 - opacity

```
[8]: fig, ax = plt.subplots()
ax.plot(x, y, color='orange', alpha=.6, ls='--', lw=3);
```



Markers

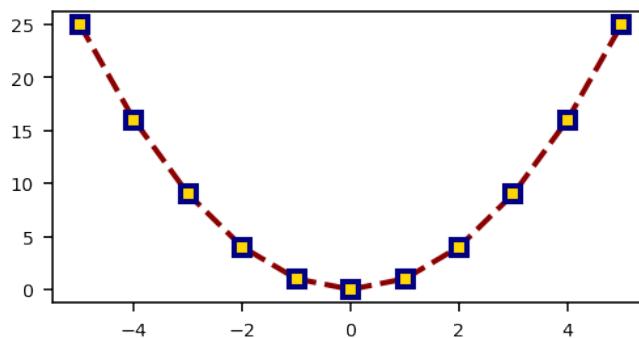
By default, the coordinates that describe the line have no special marker to denote their location. These coordinates can be visualized by providing one of a few dozen [marker styles](#) as a single character or digit. There are several other parameters available to control the marker appearance.

- `marker` - style of marker with some examples below
 - '.' - point
 - 'o' - circle
 - 's' - square
 - '+' - plus
 - 6 - caret up
- `markersize` or `ms` - size of marker in points

- `markerfacecolor` or `mfc` - face color of marker
- `markeredgecolor` or `mec` - edge color of marker
- `markeredgewidth` or `mew` - edge width of marker in points

We create another line plot using the same data adding square markers (denoted by ‘s’). The marker size, face color, edge color, and edge width are all set as well.

```
[9]: fig, ax = plt.subplots()
ax.plot(x, y, color='darkred', lw=2, ls='--',
         marker='s', ms=6, mfc='gold', mec='navy', mew=2);
```



64.3 Integration with pandas

In the official documentation, the vast majority of examples use numpy arrays as inputs for the matplotlib plotting methods. Alternatively, matplotlib allows the use of pandas Series and DataFrames as inputs. There’s even an alternative syntax that will be explained below that allows you to reference DataFrame columns by their string names. Let’s begin with examples from the flights dataset.

```
[10]: flights = pd.read_csv('../data/flights.csv', parse_dates=['date'])
flights.head(3)
```

	date	airline	origin	dest	dep_time	...	carrier_delay	weather_delay	nas_delay	security_delay	late_aircraft_delay
0	2018-01-01	UA	LAS	IAH	100	...	0	0	0	0	0
1	2018-01-01	WN	DEN	PHX	515	...	0	0	0	0	0
2	2018-01-01	B6	JFK	BOS	550	...	0	83	8	0	0

Average carrier delay per departure hour

Let’s run a calculation before plotting, such as finding the average carrier delay for each departure hour. First, we’ll round down each departure time to the nearest hour by creating the column `dep_hour`.

```
[11]: flights['dep_hour'] = flights['dep_time'] // 100
flights['dep_hour'].head(3)
```

```
[11]: 0      1
1      5
2      5
Name: dep_hour, dtype: int64
```

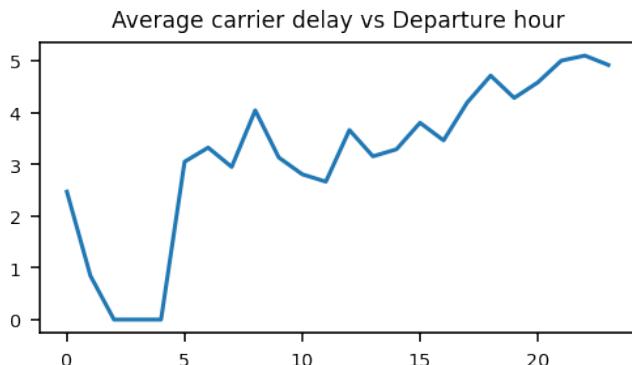
We use this new column to calculate the average carrier delay per departure hour.

```
[12]: avg_cd = flights.groupby('dep_hour').agg(avg_carrier_delay=('carrier_delay', 'mean'))
avg_cd = avg_cd.reset_index()
avg_cd.head(3)
```

dep_hour	avg_carrier_delay
0	0
1	1
2	2

We'll now make a line plot placing the departure hour as the x-values and the average carrier delay as the y-values. We'll use the old syntax, passing the x and y data as pandas Series.

```
[13]: fig, ax = plt.subplots()
x = avg_cd['dep_hour']
y = avg_cd['avg_carrier_delay']
ax.plot(x, y)
ax.set_title('Average carrier delay vs Departure hour');
```



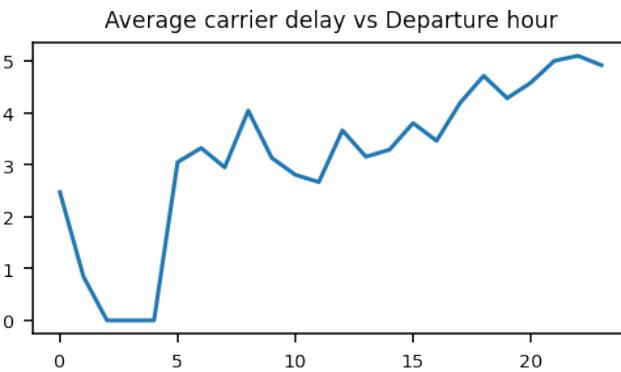
Alternative syntax for plotting with DataFrames

Most plotting methods have a `data` parameter that can be set to be a pandas DataFrame. Doing so allows you to use the column names as the x and y values. Creating a line plot from a DataFrame, `df`, with x-values in `col1` and y-values in `col2` has the following generic form:

```
ax.plot(col1, col2, data=df)
```

We recreate the above plot using this alternative syntax.

```
[14]: fig, ax = plt.subplots()
ax.plot('dep_hour', 'avg_carrier_delay', data=avg_cd)
ax.set_title('Average carrier delay vs Departure hour');
```



Plotting with strings

In our above plots, both the x and y values were numeric. It's possible to create plots where either the x or the y values are strings. Let's create a line plot of the total number of flights for each airline. The x-values will be the string abbreviation of the airline with the number of flights as the y-values. Let's calculate this result with the `value_counts` methods.

```
[15]: num_flights_series = flights['airline'].value_counts()
num_flights_series.head(3)
```

```
[15]: AA    16779
DL    13104
UA    11882
Name: airline, dtype: int64
```

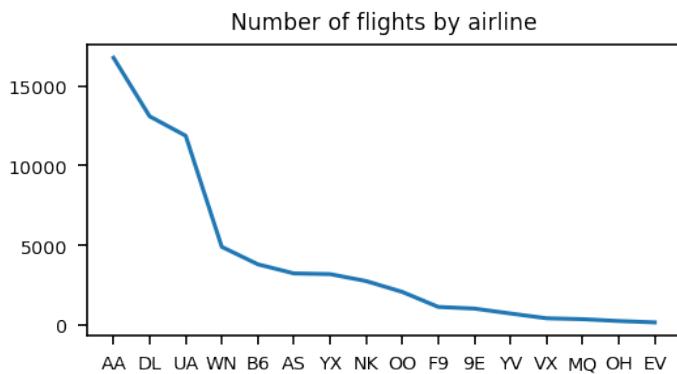
Let's convert this Series into a two-column DataFrame so that we can pass it to the `data` parameter.

```
[16]: df = num_flights_series.reset_index()
df.columns = ['airline', 'num_flights']
df.head(3)
```

	airline	num_flights
0	AA	16779
1	DL	13104
2	UA	11882

The string column `airline` is used for the x-values in the line plot below.

```
[17]: fig, ax = plt.subplots()
ax.plot('airline', 'num_flights', data=df)
ax.set_title('Number of flights by airline');
```



matplotlib places the strings on the x-axis in the order that they appear in the DataFrame. It can be helpful to understand what is happening behind the x-axis. Each unique string value is mapped to an integer beginning with 0. Let's get the underlying x-values of the tick marks.

```
[18]: ax.get_xticks()
```

```
[18]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

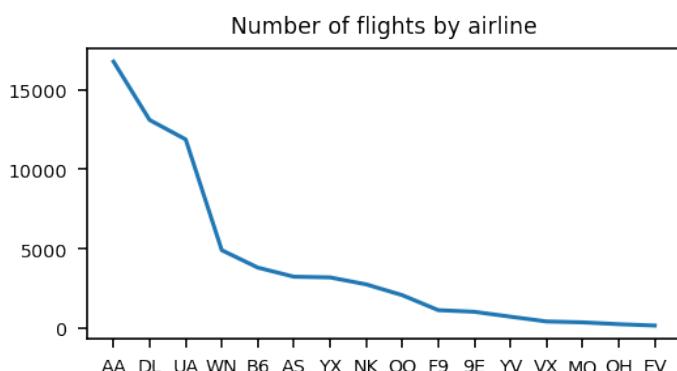
Normally, the labels for the tick marks are just the x-values themselves. When plotting strings, matplotlib uses those exact strings as the labels. Let's view a few of the underlying tick labels.

```
[19]: list(ax.get_xticklabels())[:3]
```

```
[19]: [Text(0, 0, 'AA'), Text(1, 0, 'DL'), Text(2, 0, 'UA')]
```

It's actually not necessary to convert the Series to a DataFrame. matplotlib uses the index values as the x-values in the plot. We pass the original Series computed from the `value_counts` method to the `plot` method as its only argument to produce the same plot as above.

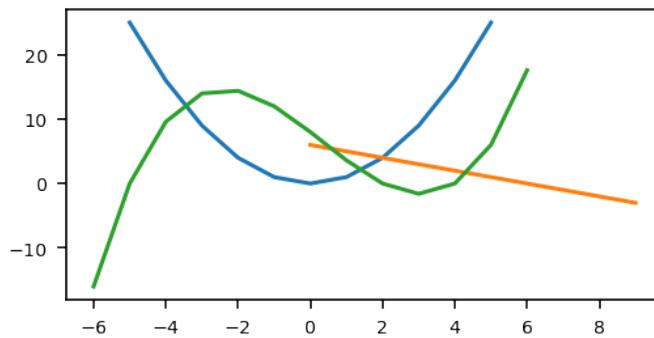
```
[20]: fig, ax = plt.subplots()
ax.plot(num_flights_series)
ax.set_title('Number of flights by airline');
```



Plotting multiple lines on the same axes

You can plot as many distinct lines on an axes as you like by repeatedly calling the `plot` method. Let's return to creating lines with simple algebraic functions like we did in our first example. Below, three different sets of x and y values are created and then plotted.

```
[21]: fig, ax = plt.subplots()
x1 = np.arange(-5, 6)
y1 = x1 ** 2
x2 = np.arange(0, 10)
y2 = -x2 + 6
x3 = np.arange(-6, 7)
y3 = .2 * (x3 + 5) * (x3 - 2) * (x3 - 4)
ax.plot(x1, y1)
ax.plot(x2, y2)
ax.plot(x3, y3);
```



Even though we did not assign the result of any of these lines to a variable, we can still access them with the `lines` attribute, which is available to all axes objects, and returns a list of all the lines.

```
[22]: ax.lines
```

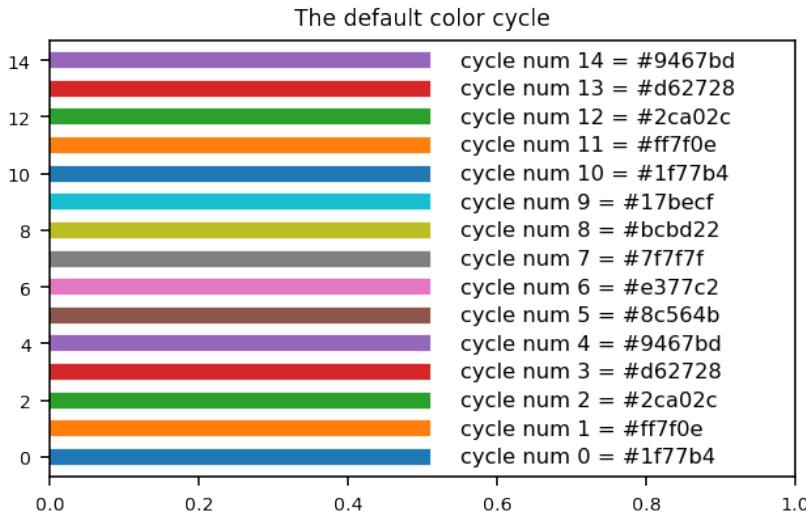
```
[22]: [<matplotlib.lines.Line2D at 0x12526cf10>,
 <matplotlib.lines.Line2D at 0x125712990>,
 <matplotlib.lines.Line2D at 0x125712ed0>]
```

Notice how each of the lines above was plotted with a different color. This happened without any explicit setting of the `color` parameter. matplotlib is designed such that each new call to one of the plotting methods results in an object with a different color. This automatic selection of new colors is referred to as the **color cycle**.

64.4 Color cycle

By default, the color cycle is a list of 10 different colors. Each time a new plotting method is called without explicitly setting a color, the next color in the color cycle is chosen. In the plot below, 15 lines are created without specifying a color. Using the `text` method, the cycle number and RGB string are placed to the right of the line. Notice that the colors begin repeating at cycle number 10.

```
[23]: fig, ax = plt.subplots(figsize=(5, 3))
x = [0, .5]
ax.set_xlim(0, 1)
ax.set_title('The default color cycle')
for i in range(15):
    y = [i, i]
    lines = ax.plot(x, y, lw=6)
    color = lines[0].get_color()
    ax.text(.55, i, s=f'cycle num {i} = {color}', fontsize=8, va='center')
```



The default color cycle is the the **tab10** colormap, which comes directly from the data visualization company Tableau. Let's verify this by listing the RGB hexadecimal strings for each of the colors in tab10. We use the `to_hex` method from the `colors` module to convert each RGB float.

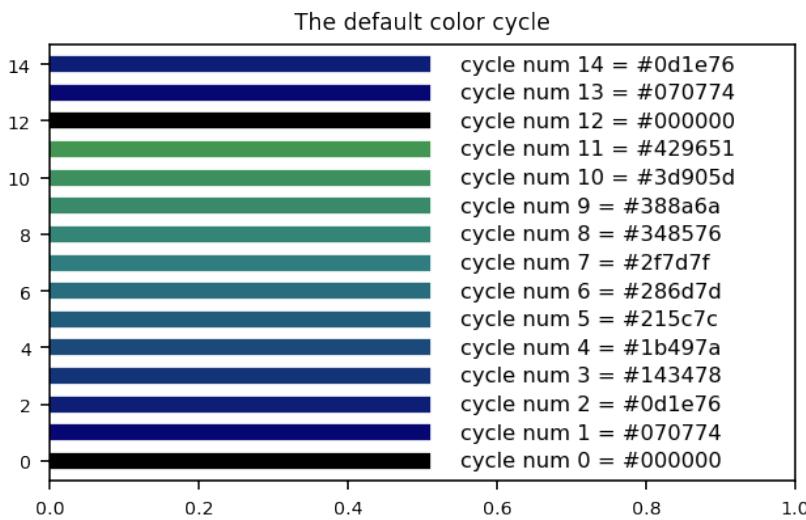
```
[24]: from matplotlib import colors, cm
print([colors.to_hex(cm.tab10(i)) for i in range(10)])
```

```
['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f',
 '#bcbd22', '#17becf']
```

Changing the color cycle

The color cycle of an axes can be changed with the `set_prop_cycle` method by setting the `color` parameter to a sequence of color values. Here, we retrieve every 10th value of the first 120 colors from the 'gist_earth' colormap for a total of 12 colors. These 12 colors are set as the new color cycle for this axes. The same 15 lines from above are produced below with the new color cycle. The color for line 12 is the same as line 0. The color cycle returns back to its default whenever a new axes is created.

```
[25]: fig, ax = plt.subplots(figsize=(5, 3))
x = [0, .5]
new_cycle = cm.gist_earth(range(0, 120, 10))
ax.set_prop_cycle(color=new_cycle)
ax.set_xlim(0, 1)
ax.set_title('The default color cycle')
for i in range(15):
    y = [i, i]
    lines = ax.plot(x, y, lw=6)
    color = colors.to_hex(lines[0].get_color())
    ax.text(.55, i, s=f'cycle num {i} = {color}', fontsize=8, va='center')
```



64.5 More line plots

Now that we understand the color cycle, let's get back to plotting multiple lines on the same axes. Using the flights dataset, let's find the number of flights every month for each of the top five busiest origin airports. We begin by creating a new column containing the first three letters of each month.

```
[26]: flights['month_name'] = flights['date'].dt.month_name().str[:3]
flights['month_name'].head(3)
```

```
[26]: 0      Jan
      1      Jan
      2      Jan
Name: month_name, dtype: object
```

To help pandas sort the data appropriately, we can convert this column to an ordered categorical. First, we'll need the unique values of the months in order. Since the data is already ordered by date, the `drop_duplicates` method should return the sequence we want.

```
[27]: months = flights['month_name'].drop_duplicates()
months
```

```
[27]: 0      Jan
      5165    Feb
      10009   Mar
      15420   Apr
      21018   May
      26770   Jun
      32636   Jul
      38546   Aug
      44407   Sep
      49774   Oct
      55396   Nov
      60641   Dec
Name: month_name, dtype: object
```

We can now make the conversion.

```
[28]: cat_dtype = pd.CategoricalDtype(categories=months, ordered=True)
flights['month_name'] = flights['month_name'].astype(cat_dtype)
```

The top five origin airports are now found.

```
[29]: top_5_origin = flights['origin'].value_counts().index[:5]
top_5_origin
```

```
[29]: Index(['ORD', 'LAX', 'ATL', 'SFO', 'BOS'], dtype='object')
```

We use the `query` method to filter our data to just these five airports and see that over 40,000 rows have been filtered out.

```
[30]: flights_busy = flights.query('origin in @top_5_origin')
len(flights) - len(flights_busy)
```

```
[30]: 43606
```

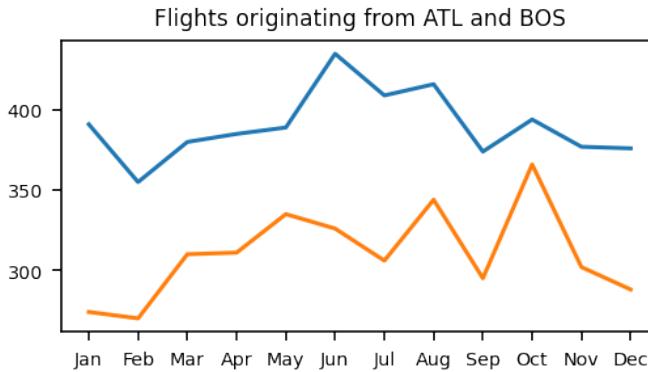
We can now use the `crosstab` function to count all the flights for each month for every origin airport. Because the month names are ordered categoricals, they will remain in that order in the index.

```
[31]: origin_flight_counts = pd.crosstab(index=flights_busy['month_name'],
                                         columns=flights_busy['origin'])
origin_flight_counts.head()
```

	origin	ATL	BOS	LAX	ORD	SFO
month_name						
Jan	391	274	384	428	310	
Feb	355	270	403	371	288	
Mar	380	310	377	361	302	
Apr	385	311	444	423	313	
May	389	335	420	464	336	

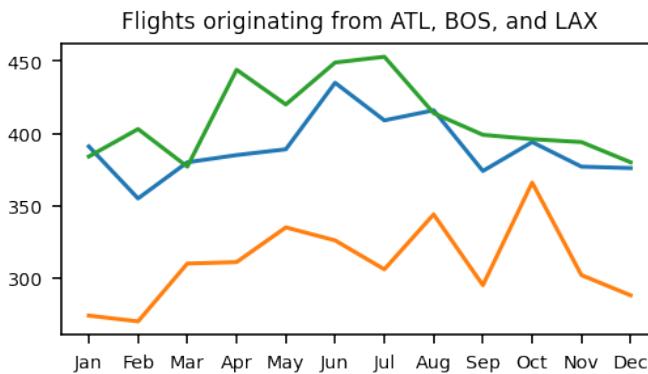
Let's plot the first two columns as line plots by calling the `plot` method twice, passing it each column as a Series.

```
[32]: fig, ax = plt.subplots()
atl = origin_flight_counts['ATL']
bos = origin_flight_counts['BOS']
ax.plot(atl)
ax.plot(bos)
ax.set_title('Flights originating from ATL and BOS');
```



Each unique axes keeps track of where it is in the color cycle. The first plot on each axes is given the first color of the cycle. Let's add the next origin airport to the plot, which uses the next color in the cycle.

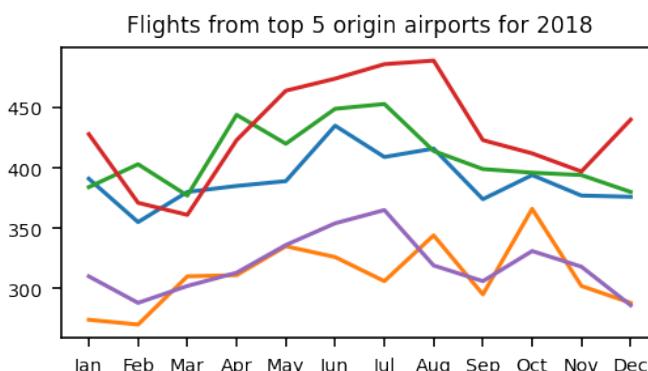
```
[33]: lax = origin_flight_counts['LAX']
ax.plot(lax)
ax.set_title('Flights originating from ATL, BOS, and LAX')
fig
```



Plotting DataFrames

Instead of plotting one column at a time, you can pass the entire DataFrame to the `plot` method. Each column will be plotted as a separate line and use the index as its x-values.

```
[34]: fig, ax = plt.subplots()
ax.plot(origin_flight_counts)
ax.set_title('Flights from top 5 origin airports for 2018');
```



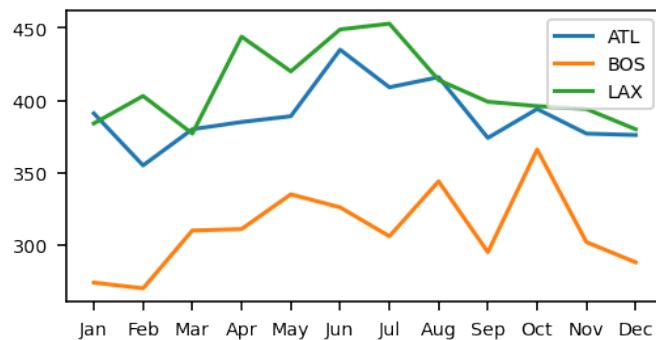
64.6 Adding a legend

It's impossible to determine the origin airport of each line just by looking at the plot. Legends are an important addition allowing you to label each object on the plot. matplotlib grants its users lots of power to create the exact legends they desire. Unfortunately, this extensive power is often difficult to use for beginners. In this section, we begin by creating legends with a simple, straightforward approach, slowly adding more complexity.

Simple legend with `label` parameter

Every plotting method has a `label` parameter that can be set to a string to identify the particular objects produced from that method. Calling the `legend` axes method creates a legend with the string used as the `label`. Let's make the same three calls to the `plot` method as we did above, but use the `label` parameter to identify each line before calling the `legend` method with no arguments.

```
[35]: fig, ax = plt.subplots()
ax.plot(atl, label='ATL')
ax.plot(bos, label='BOS')
ax.plot(lax, label='LAX')
ax.legend();
```

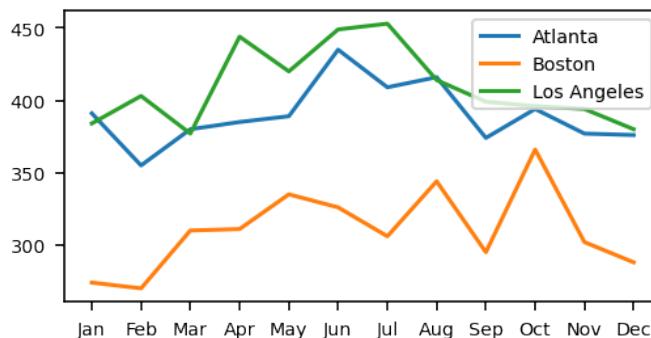


Setting the `label` isn't enough for the legend to appear. You must call the `legend` method if you want to have a legend. If you call `legend` without having set any of the labels, then that particular object will not appear in the legend.

Setting the labels in the `legend` method with a list

The `legend` method has a huge number of parameters that can be set to control its appearance. It's not strictly necessary to set the `label` parameter within the plotting method. Alternatively, you can set the `labels` parameter of the `legend` method to a list of strings.

```
[36]: fig, ax = plt.subplots()
ax.plot(atl)
ax.plot(bos)
ax.plot(lax)
labels=['Atlanta', 'Boston', 'Los Angeles']
ax.legend(labels=labels);
```

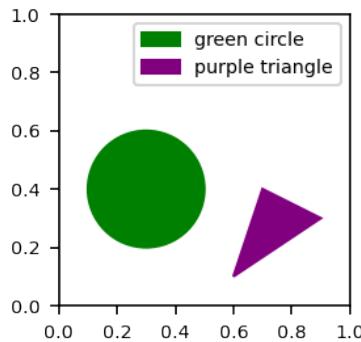


While this works, I prefer using the `label` parameter. With this method, you have to make sure that your labels are listed in the same order as the lines they reference. It's not quite as explicit as the previous method.

Labeling patches in the legend

All patch constructors have the same `label` parameter that can be used to identify it in the legend. A circle and triangle are added to the axes with a legend to identify them.

```
[37]: from matplotlib import patches
fig, ax = plt.subplots()
ax.set_aspect('equal')
p1 = patches.Circle((.3, .4), radius=.2, color='green', label='green circle')
p2 = patches.Polygon([(0.6, 0.1), (0.7, 0.4), (0.9, 0.3)], color='purple', label='purple triangle')
ax.add_patch(p1)
ax.add_patch(p2)
ax.legend();
```



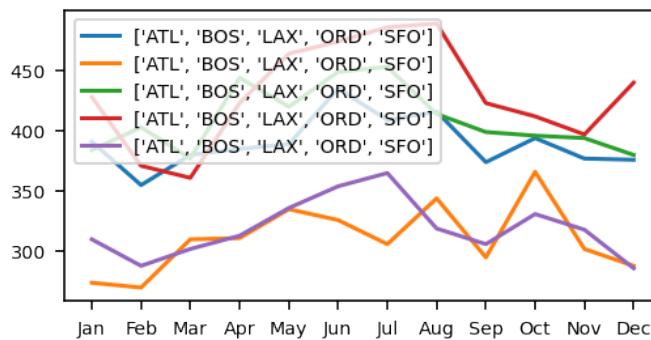
Legends when plotting DataFrames

We plotted all five origin airports with a single call to the `plot` method when passing it the DataFrame `origin_flight_counts` above. If we want to add a legend with a label for each line, we can't proceed as before since there is only a single `plot` method called which has a single `label` parameter.

You might think that setting the `label` parameter to a list would allow you to label each line individually. This unfortunately does not work and shows the label of each line as the entire list as is seen below. Both Series and Index objects have a `tolist` method to make the conversion to a list.

```
[38]: fig, ax = plt.subplots()
origin_cols = origin_flight_counts.columns.tolist()
```

```
ax.plot(origin_flight_counts, label=origin_cols)
ax.legend();
```



Legend entries - handle, label pairs

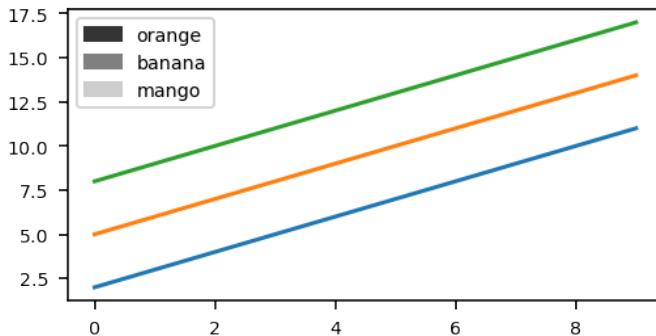
Before showing how to properly label the legend when plotting DataFrames, we'll need to cover more specifics of the legend itself. Every legend is composed of one or more **entries**. Each entry is composed of a **handle**, the colored marker on the left, and the **label**, the text to the right.

For all of our examples thus far, matplotlib has created the legend handles for us, and we've provided the labels as strings. You can actually provide both the handles and the labels yourself. The handles must be a plotting object, but don't have to be part of the plot or have anything to do with the data.

Here, three lines are added to the axes with the `plot` method and given a label. Three different patches that have nothing to do with the data, and are not added to the axes, are created and placed in a list. This list is passed to the `legend` method as the handles along with a list of strings for the labels. The handles and labels provided to the `legend` method override any handles and labels automatically generated from the plotting objects.

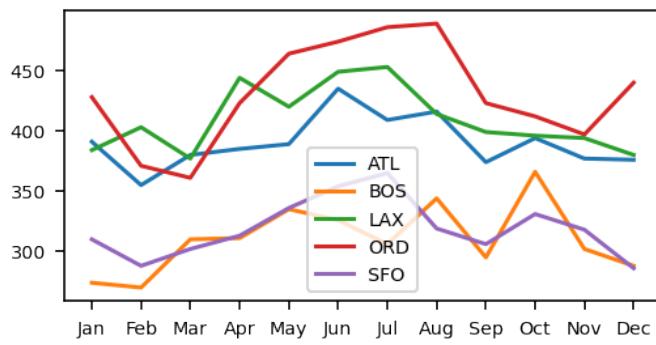
```
[39]: fig, ax = plt.subplots()
x = np.arange(10)
ax.plot(x, x + 2, label='Line 1')
ax.plot(x, x + 5, label='Line 2')
ax.plot(x, x + 8, label='Line 3')

# patches that are NOT added to the plot
p1 = patches.Circle((0, 0), radius=10, color='.2')
p2 = patches.Rectangle((5, -15), width=3, height=2, color='.5')
p3 = patches.Polygon([[100, 50], [30, 40], [500, -300]], color='.8')
handles = [p1, p2, p3]
labels = ['orange', 'banana', 'mango']
ax.legend(handles=handles, labels=labels);
```



In order to create the legend properly when plotting multiple lines using a DataFrame, you'll need to assign the list of lines returned from the `plot` method to a variable and use them as the handles in the `legend` method. The list of column names can be used as the labels.

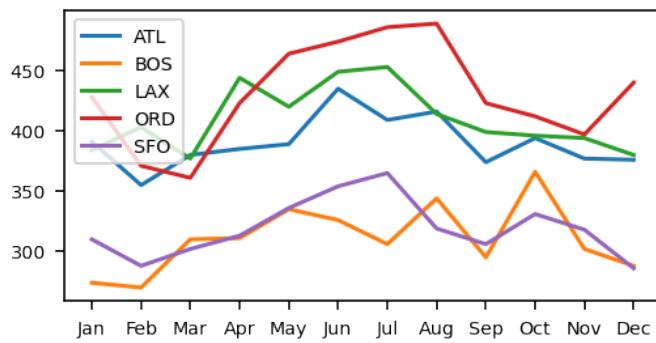
```
[40]: fig, ax = plt.subplots()
lines = ax.plot(origin_flight_counts)
ax.legend(handles=lines, labels=origin_cols);
```



Moving the legend

By default, matplotlib attempts to put the legend in the best possible location within the axes. You can specify the location of the legend by setting the `loc` parameter to a string combining a vertical ('`upper`', '`center`', '`lower`') and horizontal ('`left`', '`center`', '`right`') position. The string '`best`' is also a choice, but is the default. Let's move our legend to the upper left corner of the plot. Instead of creating a new figure, we'll just call the `legend` method again which replaces the current legend.

```
[41]: ax.legend(handles=lines, labels=origin_cols, loc='upper left')
fig
```

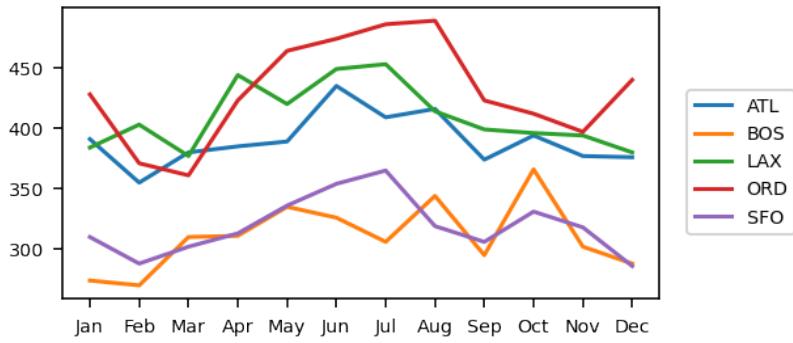


You can also place the legend in a specific location by setting the `bbox_to_anchor` parameter to a two-item

numeric tuple. This tuple defines coordinates where the legend will be ‘anchored’. The coordinates to define this point are relative to the axes where 0 is the beginning of the x and y axis and 1 is the end of the x and y axis. The coordinate (0, 0) corresponds to the lower left corner of the plot, with (.5, .5) corresponding to the center. These units have nothing to do with the actual data.

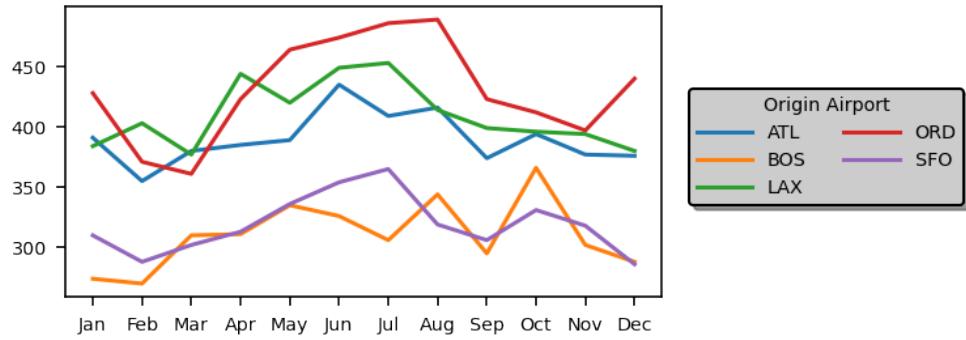
It’s possible to place the legend outside of the plotting surface by specifying an x or y coordinate outside of the range 0 to 1. Below, we anchor the upper left corner of the legend to the point (1.03, .75) which is just to the right of the axes.

```
[42]: ax.legend(handles=lines, labels=origin_cols, loc='upper left', bbox_to_anchor=(1.03, .75))
fig
```



Several more legend properties exist and are set below. Control the number of columns in the legend with `ncol` and the relative length of each marker with `handlelength` (which has default value of 2). The other parameters should be intuitive to understand. [Visit the documentation](#) to learn about all of the legend parameters.

```
[43]: ax.legend(handles=lines, labels=origin_cols, loc='upper left',
             bbox_to_anchor=(1.03, .75), title='Origin Airport', ncol=2,
             facecolor='.8', handlelength=3, edgecolor='black', shadow=True)
fig
```



Accessing the legend

The legend object can be accessed by assigning the result of the call to the `legend` method to a variable or from the `legend_` axes attribute.

```
[44]: legend = ax.legend_
type(legend)
```

[44]: `matplotlib.legend.Legend`

We've seen that other groups of plotting objects are accessed with attributes that end in 's' such as `lines`, `collections`, `patches`, and `texts`. `matplotlib` only allows a single legend per axes, therefore the name `legends` wouldn't quite work, so the developers chose `legend_` instead. As usual, all of the properties can be retrieved and changed with the getter and setter methods.

64.7 Exercises

Exercise 1

Create line plots of the average sale price per every 100 square feet of living area, garage area, and basement area. Place markers at every point and add a legend.

[]:

Exercise 2

For every neighborhood that has at least 100 homes, find the average sale price by each overall quality between 3 and 8 (inclusive). Plot each neighborhood as a line with overall quality on the x-axis and sale price on the y-axis. Use one of the [qualitative colormaps](#) other than the default tab10. Add a legend inside the bounds of the axes that has a frame, a title, and all labels on one row.

[]:

Chapter 65

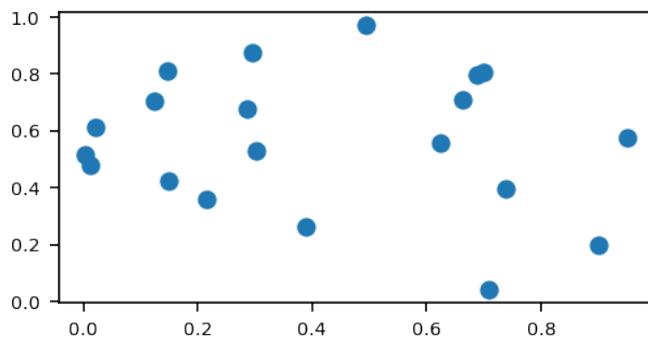
Matplotlib Scatter and Bar Plots

In this chapter, we'll cover how to create scatter and bar plots in matplotlib using data.

65.1 Scatter plots

A typical scatter plot consists of a sequence of paired x and y coordinates each plotted as a single point without being connected to one another. In matplotlib, the `scatter` method is usually the best way to create a scatter plot. Its first two arguments are the same as the `plot` method, the x and y values. Below, we create a scatter plot of 20 points using random values between 0 and 1 generated with numpy. We assign the result of the call to the `scatter` method to its own variable.

```
[1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('..../mdap.mplstyle')
fig, ax = plt.subplots()
x = np.random.rand(20)
y = np.random.rand(20)
scatter_obj = ax.scatter(x, y)
```



Each point is represented as a small circle of the same color. Let's find out what type of object gets returned.

```
[2]: type(scatter_obj)
```

```
[2]: matplotlib.collections.PathCollection
```

This `PathCollection` object represents all 20 of these points. As usual, it has getter and setter methods to view and change its properties. When calling the `get_facecolor` method, a two-dimensional numpy array

is returned and not a four-item tuple or an RGBA string as you might expect. With the `scatter` method, every single point can be a different color, so its possible a 20×4 array was returned with RGBA floats for each point.

```
[3]: scatter_obj.get_facecolor()
```

```
[3]: array([0.12156863, 0.46666667, 0.70588235, 1.          ]])
```

The default size of a scatter plot point is in points squared. This value is 36, which means it is 6 points or $1 / 12$ ($6 / 72$) of a figure inch in diameter. The `get_sizes` method returns this size as an array. Like the face color, each point can be a different size, so it could have returned an array of 20 values.

```
[4]: scatter_obj.get_sizes()
```

```
[4]: array([36.])
```

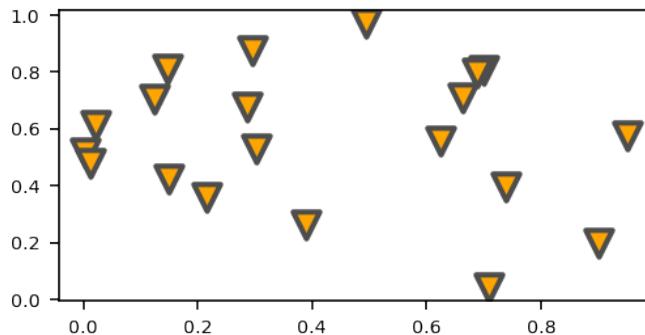
Scatter plot properties

There are several parameters you can set to customize the appearance of your scatter plot. The following are the most common:

- `s` - size of marker in points squared
- `c` - face color of marker
- `marker` - a single character or digit [marker style](#)
- `linewidth/lw` - width of marker edge in points
- `edgecolor/ec` - color of marker edge

Let's set each of these parameters to a value and plot the same data. The size of each marker is set to 100, but this is in points squared, meaning that the actual size is the square root of 100, or 10.

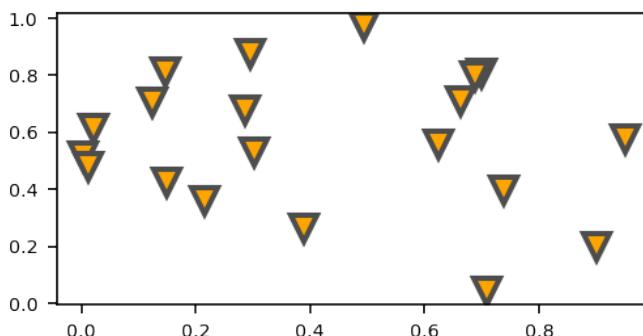
```
[5]: fig, ax = plt.subplots()
ax.scatter(x, y, s=100, c='orange', marker='v', lw=2, ec='.3');
```



Creating a scatter plot with the `plot` method

The primary purpose of the `plot` method is to create line plots. But, it can be used to create scatter plots by setting the `linestyle` or `ls` parameter to an empty string and setting the `marker` parameter to any valid value. Let's replicate the above scatter plot with the `plot` method. The parameters `ms`, `mew`, and `mec` are used instead of `s`, `lw` and `ec`. Note that the marker size for the `plot` method is given in actual points (10) and not points squared (100) as it was with `scatter`.

```
[6]: fig, ax = plt.subplots()
ax.plot(x, y, ms=10, c='orange', marker='v', mew=2, mec='.3', ls='');
```



Although the exact same output is produced, I don't recommend using the `plot` method to create scatter plots. With the `scatter` method, it's possible to control the properties of each individual point. With the `plot` method, all points must have the same properties.

Setting properties of individual points

Let's show examples of how we can change the properties of each individual point with the `scatter` method. We begin by reading in a few columns of the housing dataset, getting a random sample of 100 houses.

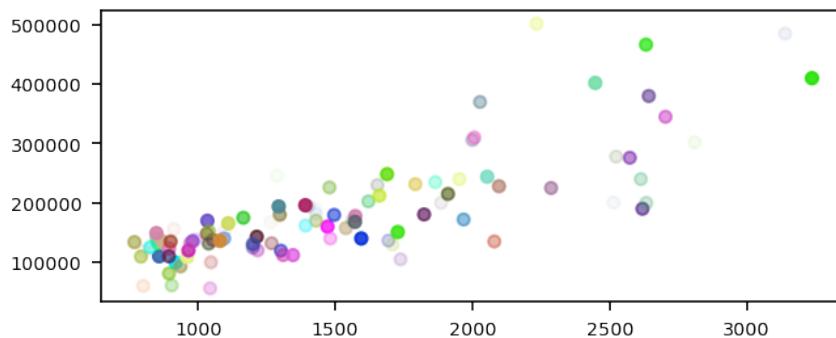
```
[7]: cols = ['OverallQual', 'Exterior1st', 'GrLivArea',
           'GarageArea', 'GarageCars', 'SalePrice']
housing = pd.read_csv('../data/housing.csv', usecols=cols)
housing = housing.sample(100, random_state=43).reset_index(drop=True)
housing.head(3)
```

	OverallQual	Exterior1st	GrLivArea	GarageCars	GarageArea	SalePrice
0	7	Wd Sdng	1426	1	216	189950
1	6	Wd Sdng	1298	2	403	180000
2	6	MetalSd	1694	2	576	136500

Setting the color of individual points

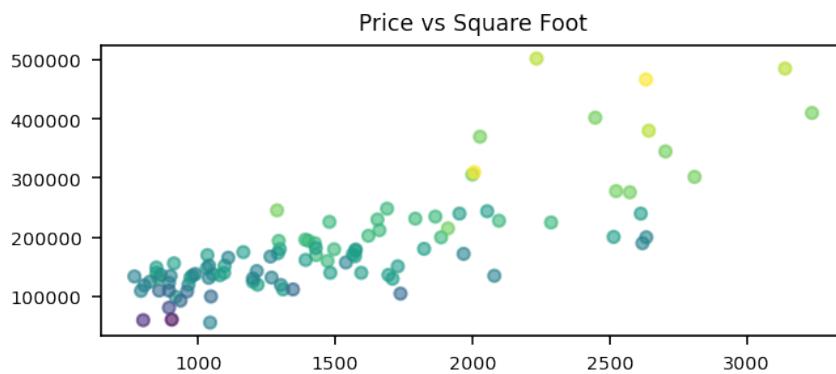
The color of each individual point is controlled by the `c` (and NOT the `color`) parameter. You can set it to a sequence of web colors or RGBA values the same length as the data. Below, we create an array of 100 random RGBA values and use them to color the points of the scatter plot between living area and sale price.

```
[8]: c = np.random.rand(100, 4)
fig, ax = plt.subplots(figsize=(5, 2))
ax.scatter('GrLivArea', 'SalePrice', c=c, data=housing, s=20);
```



It's possible to use column names to reference the color name, just like we do with the x and y values. The column values must either be numeric or named color strings. Let's use the overall quality column to color the points. The values in this column range from 1 to 10. Also, all points are made slightly transparent, which is important to do for scatter plots that have many overlapping values.

```
[9]: fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c='OverallQual',
                        data=housing, alpha=.6, s=20)
ax.set_title('Price vs Square Foot');
```



Colors chosen with default colormap

matplotlib uses a colormap to select the colors for the points. The default colormap is viridis, which contains 256 colors. matplotlib maps each value in the overall quality column to a single color of the colormap using the following steps:

- Calculate the range of the values (difference between maximum and minimum)
- Subtract the minimum from each value
- Divide each value by the range
- Multiply each value by the number of colors
- Truncate the decimal to get an integer

These integers are then passed to the colormap object to retrieve the RGB value. Since there are only 10 unique values of overall quality, we can follow the above steps ourselves to produce the colormap integer for each value and output the result as a DataFrame.

```
[10]: from matplotlib import cm
min_oq = housing['OverallQual'].min()
max_oq = housing['OverallQual'].max()
range_oq = max_oq - min_oq
oq_vals = np.arange(1, 11)
```

```
N = cm.viridis.N
cm_vals = ((oq_vals - min_oq) / range_oq * N // 1).astype('int64')
df_colormap = pd.DataFrame({'OverallQual': oq_vals, 'Colormap integer': cm_vals})
df_colormap
```

	OverallQual	Colormap integer
0	1	0
1	2	28
2	3	56
3	4	85
4	5	113
5	6	142
6	7	170
7	8	199
8	9	227
9	10	256

The lowest value always gets mapped to 0 and the highest to N , the total number of colors in the colormap. Let's apply the same transformation to all the actual values of overall quality to create a Series of colormap integers.

```
[11]: cm_values = (housing['OverallQual'] - min_oq) / range_oq * N // 1
cm_values = cm_values.astype('int64')
cm_values.head()
```

```
[11]: 0    170
      1    142
      2    142
      3    170
      4    113
Name: OverallQual, dtype: int64
```

These are the values matplotlib used to determine the color of each point in our scatter plot. Let's convert each of these integers to its RGBA value using the same alpha for each.

```
[12]: rgba_array = cm.viridis(cm_values, alpha=.6)
rgba_array[:3]
```

```
[12]: array([[0.20803 , 0.718701, 0.472873, 0.6       ],
       [0.119699, 0.61849 , 0.536347, 0.6       ],
       [0.119699, 0.61849 , 0.536347, 0.6       ]])
```

The above values are what we believe matplotlib used as RGBA colors for each point. Let's get the actual RGBA values from the plotting object with the `get_facecolor` method.

```
[13]: scatter_rgba = scatter_obj.get_facecolor()
scatter_rgba[:3]
```

```
[13]: array([[0.20803 , 0.718701, 0.472873, 0.6      ],
           [0.119699, 0.61849 , 0.536347, 0.6      ],
           [0.119699, 0.61849 , 0.536347, 0.6      ]])
```

The values from the arrays appear to be identical. Let's verify this by running the `assert_array_equal` method from numpy's `testing` module. If no exception is raised, then the arrays are equal.

```
[14]: np.testing.assert_array_equal(rgbarray, scatter_rgba)
```

Adding a legend to the scatter plot

Adding a legend to a scatter plot is simple if all the points represent the same thing. In that case, set the `label` parameter of the `scatter` method to a string and then call the `legend` method as before.

If different groups of points require different labels, more work has to be done. Call the `legend_elements` method from the object returned from `scatter`. This will automatically generate the handles and labels for you as a two-item tuple of lists. Below, we unpack these two returned objects into their own variable names.

```
[15]: handles, labels = scatter_obj.legend_elements()
```

Let's output the `handles` list to see what it contains.

```
[16]: handles
```

```
[16]: [<matplotlib.lines.Line2D at 0x1214ade50>,
        <matplotlib.lines.Line2D at 0x1214ada50>,
        <matplotlib.lines.Line2D at 0x1214ad5d0>,
        <matplotlib.lines.Line2D at 0x116f2cdd0>,
        <matplotlib.lines.Line2D at 0x1214dd950>,
        <matplotlib.lines.Line2D at 0x12205ff50>,
        <matplotlib.lines.Line2D at 0x12205fed0>,
        <matplotlib.lines.Line2D at 0x12205ff90>,
        <matplotlib.lines.Line2D at 0x12205fdd0>,
        <matplotlib.lines.Line2D at 0x12205fe90>]
```

It contains a list of 10 lines corresponding to the 10 unique values of overall quality. These lines are not on the axes. They are created from `legend_elements` just so you can pass them to the `legend` method. The color of the first line is the same color of the lowest overall quality value, which we verify below.

```
[17]: handle_0 = handles[0]
handle_0.get_color() == cm.viridis(0)
```

```
[17]: True
```

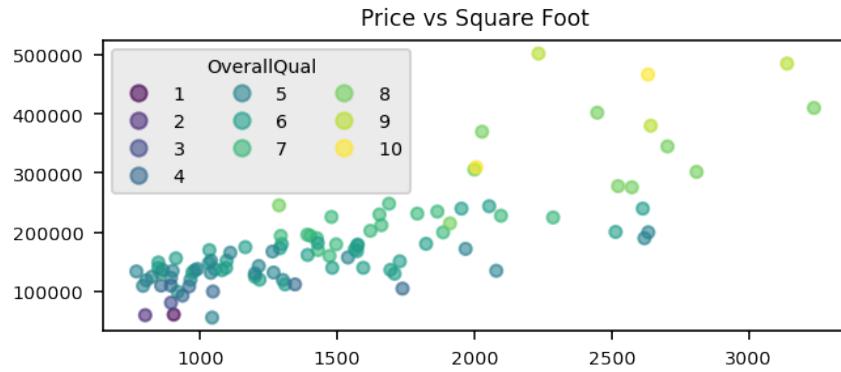
The labels are a list of 10 strings corresponding to the 10 unique values of overall quality. The first few are output below.

```
[18]: labels[:3]
```

```
[18]: ['$\\mathdefault{1}$', '$\\mathdefault{2}$', '$\\mathdefault{3}$']
```

These strings look strange, but essentially represent the integers 1 to 10. matplotlib allows you to write mathematical expressions in a special markup language called LaTeX by placing the content between dollar signs in a string. The details of this will not be presented here. The legend is created by passing these handles and labels to the `legend` method.

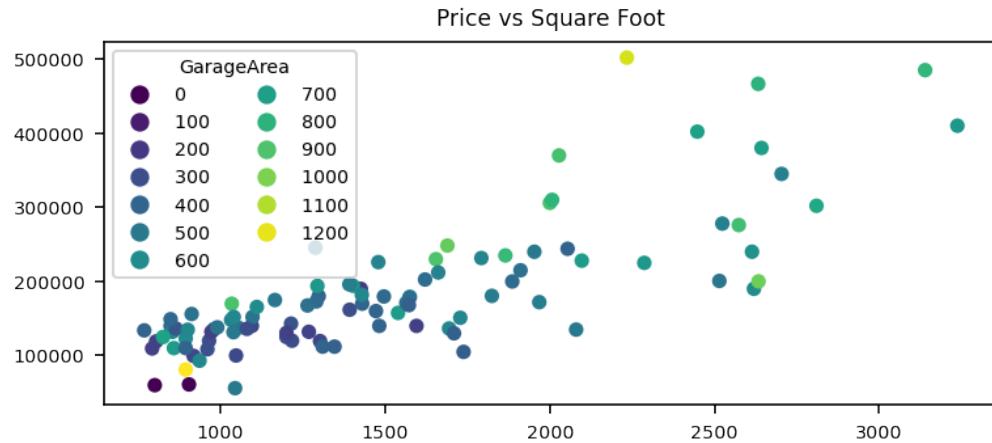
```
[19]: ax.legend(handles=handles, labels=labels, title='OverallQual',
             facecolor='.9', ncol=3, loc='upper left')
fig
```



Coloring by a column with more unique values

Let's now color each point by the size of the house's garage area, which is numeric and therefore valid. Each value of garage area will be mapped to a specific color from the viridis colormap using the rules from above. The `legend_elements` method is called to return the handles and labels again. It does not produce a legend entry for every unique value, but instead smartly chooses a reasonable number of entries given the data. You can control the exact number of entries to produce by setting the `num` parameter.

```
[20]: fig, ax = plt.subplots(figsize=(6, 2.5))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c='GarageArea', data=housing, s=20)
handles, labels = scatter_obj.legend_elements(num=12)
ax.legend(handles=handles, labels=labels, title='GarageArea', ncol=2)
ax.set_title('Price vs Square Foot');
```

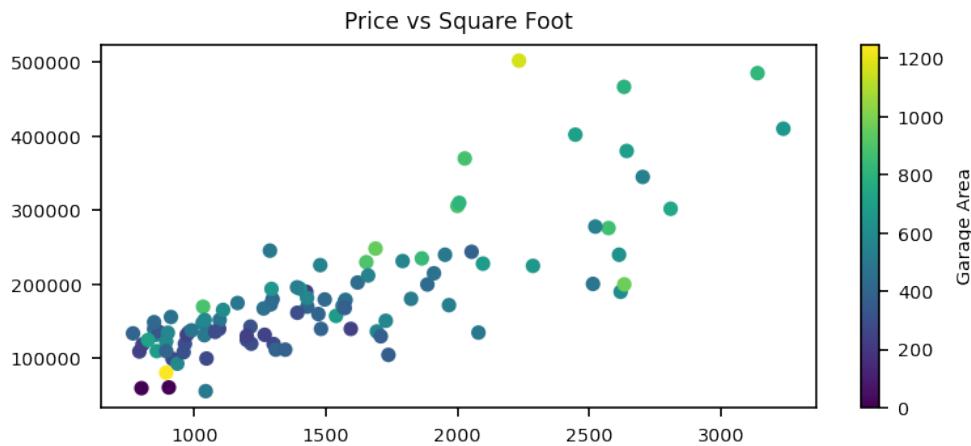


Using a colorbar

If a substantial number of unique values exist, a colorbar might be a better choice to label the meaning of colors. Call the `colorbar` `figure` method passing it the underlying plotting object. This is a rare instance

where a figure (and not an axes) method is used. The entire spectrum of the colormap with its corresponding values will be placed in a slim vertical bar on the right side of the figure.

```
[21]: fig, ax = plt.subplots(figsize=(6, 2.5))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c='GarageArea', data=housing, s=20)
fig.colorbar(scatter_obj, label='Garage Area')
ax.set_title('Price vs Square Foot');
```



Technically, an entire new axes was created. You can retrieve all axes from the figure as a list using an attribute of the same name.

```
[22]: fig.axes
```

```
[22]: [<matplotlib.axes._subplots.AxesSubplot at 0x1219f2250>,
<matplotlib.axes._subplots.AxesSubplot at 0x121b5b250>]
```

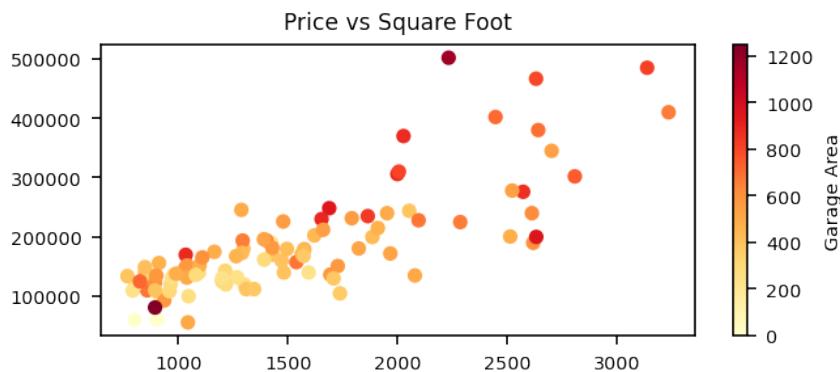
Using a different colormap

Take a look at all of the [colormaps in the official documentation](#). You'll notice that they are divided into the categories sequential, diverging, cyclic, and qualitative. Each category of colors is best suited for a particular type of data.

- sequential - data has a meaningful order - population, SAT score, quality of kitchen, salary, diamond clarity, etc...
- diverging - data has a clear point that divides the distribution - sea level, velocity, z-scores, etc...
- cyclic - data where first and last values have similar meaning - seasonal data, business cycles, etc...
- qualitative - data has no meaningful order - states, animals, brands, type of roof, department, etc...

Garage area is data with meaningful order with higher values meaning larger garages. The default colormap, viridis, is sequential so it is already a valid choice. But, there are several other sequential colormaps that we can choose from. Below, we use 'YlOrRd', which begins with shades of yellow, before moving to orange and red. We pass it as a string to the `cmap` parameter.

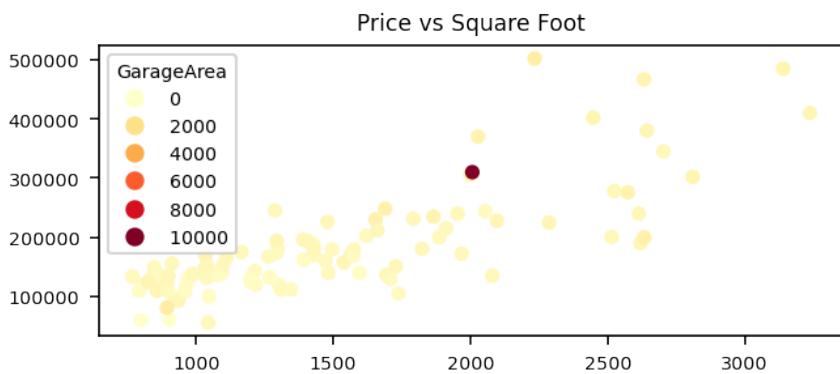
```
[23]: fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c='GarageArea',
                        cmap='YlOrRd', data=housing, s=20)
fig.colorbar(scatter_obj, label='Garage Area')
ax.set_title('Price vs Square Foot');
```



Limiting the range of values for the colormap

If there are outliers in the column used to create the colormap, then the diversity of colors shown can greatly diminish. For example, the largest garage area is currently around 1,000. If we change the garage area of one house to 10,000, then the distribution of colors would completely change. All the houses except one would be mapped to the first 10% of the colormap spectrum. Let's see how this would look.

```
[24]: housing.loc[67, 'GarageArea'] = 10_000
fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c='GarageArea',
                        cmap='YlOrRd', data=housing, s=20)
handles, labels = scatter_obj.legend_elements(num=5)
ax.legend(handles=handles, labels=labels, title='GarageArea')
ax.set_title('Price vs Square Foot');
```

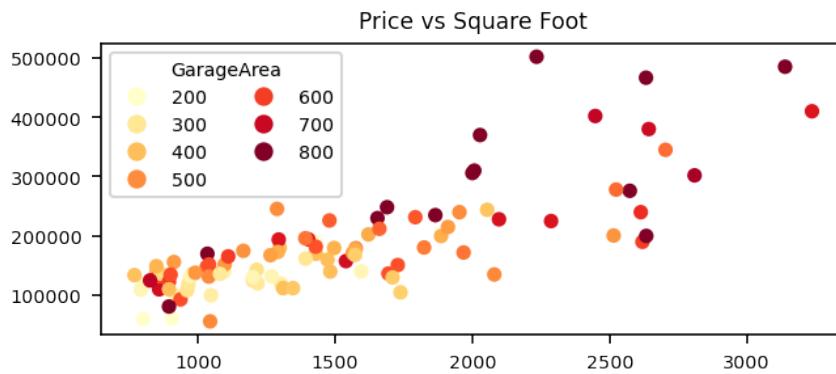


The coloring is now polarized and completely dominated by this one outlying garage area. You can set the minimum and maximum values of the range of the variable used for the colormap to consider with the `vmin` and `vmax` parameters of the `scatter` method. The values 200 and 800 are used below for this range. Garage areas below 200 and above 800 will be mapped to the first and last colors of the colormap respectively.

Instead of using an integer for the `num` parameter in `legend_elements`, a sequence can be used to provide the exact values you want. If you don't provide a sequence, then matplotlib will choose equally spaced values from 0 to 10,000.

```
[25]: fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c='GarageArea',
                        cmap='YlOrRd', data=housing, s=20, vmin=200, vmax=800)
handles, labels = scatter_obj.legend_elements(num=range(200, 900, 100))
ax.legend(handles=handles, labels=labels, title='GarageArea', ncol=2)
```

```
ax.set_title('Price vs Square Foot');
```



Using string columns to color

DataFrame columns with string values can also be used to color the points of a scatter plot, but you'll need to convert the values to integers. In the housing dataset, the `Exterior1st` column contains a string description of the primary exterior of each house.

```
[26]: housing['Exterior1st'].value_counts()
```

```
[26]:
```

VinylSd	35
MetalSd	20
Wd Sdng	15
HdBoard	13
Plywood	9
BrkFace	3
CemntBd	3
AsbShng	1
WdShing	1

Name: `Exterior1st`, dtype: int64

Attempting to use this column for the color directly results in an error.

```
[27]: ax.scatter('GrLivArea', 'SalePrice', c='Exterior1st', data=housing)
```

```
ValueError: 'c' argument must be a color, a sequence of colors, or a sequence of numbers, not 0 Wd Sdng
1    Wd Sdng
2    MetalSd
3    HdBoard
4    Plywood
...
95   VinylSd
96   MetalSd
97   BrkFace
98   Plywood
99   VinylSd
Name: Exterior1st, Length: 100, dtype: object
```

We need to map each value to an integer. The simplest way to do this is to convert it to the categorical data type. Since it has no inherent ordering, we convert it without providing a list of all the categories.

```
[28]: housing['Exterior1st'] = housing['Exterior1st'].astype('category')
```

The underlying integer values can be accessed with the `cat` accessor's `codes` attribute.

```
[29]: ext_codes = housing['Exterior1st'].cat.codes
ext_codes.head(3)
```

```
[29]: 0    7
      1    7
      2    4
      dtype: int8
```

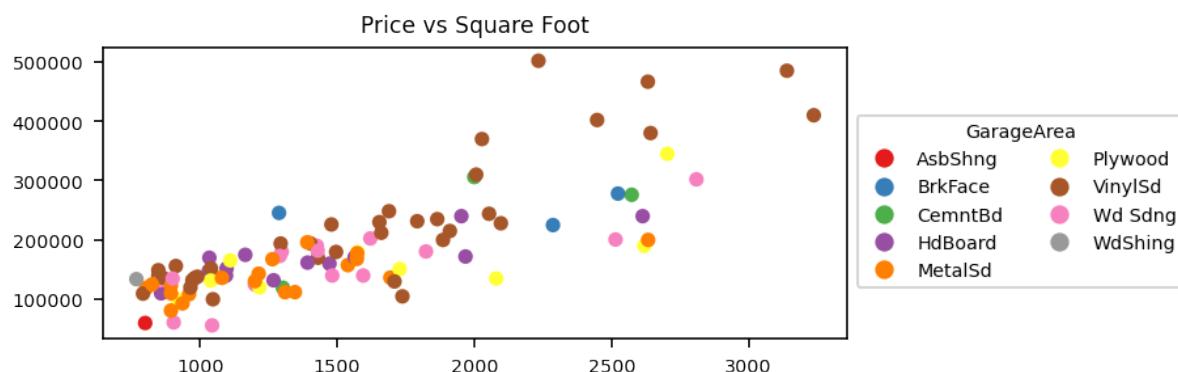
The category names need to be retrieved for the legend. We use the `cat` accessor again and convert the values to a list and display a few of the categories.

```
[30]: categories = housing['Exterior1st'].cat.categories.tolist()
categories[:5]
```

```
[30]: ['AsbShng', 'BrkFace', 'CemntBd', 'HdBoard', 'MetalSd']
```

This data is best plotted with a qualitative colormap as it has no inherent ordering. The Set1 colormap is chosen, which has 9 colors, the exact number of unique values of `Exterior1st`. The labels are set to the categories.

```
[31]: fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', c=ext_codes, cmap='Set1',
                        data=housing, s=20)
handles, labels = scatter_obj.legend_elements()
ax.legend(handles=handles, labels=categories, title='GarageArea',
           ncol=2, bbox_to_anchor=(1, .8))
ax.set_title('Price vs Square Foot');
```



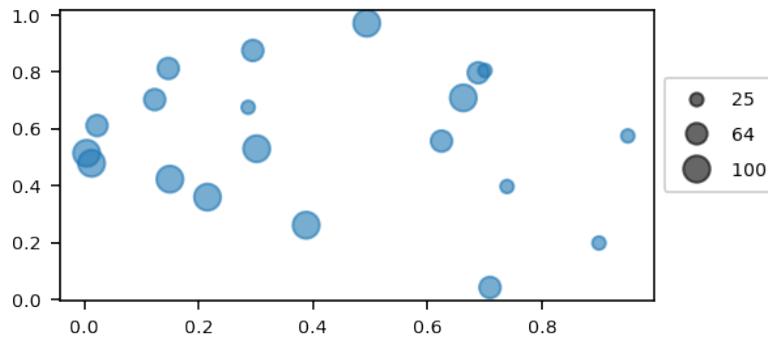
65.2 Change scatter plot point size

The size of each point of a scatter plot is provided as typographical points squared with the `s` parameter. You can set it as a single number, a sequence of numbers, or a DataFrame column name containing numeric values. Let's plot our 20 random values with a random size of either 25, 64, or 100. While these numbers

seem large, they are in points squared, so these translate to heights of 5, 8, and 10 points or .07, .11, and .14 figure-inches (dividing by 72).

When calling `legend_elements`, set `prop` to the string '`sizes`' so that it returns handles and labels referencing the size and not the color. Other legend parameters that control the spacing and padding are also used.

```
[32]: fig, ax = plt.subplots()
s = np.random.choice([25, 64, 100], 20)
scatter_obj = ax.scatter(x, y, s=s, alpha=.6)
handles, labels = scatter_obj.legend_elements(prop='sizes')
ax.legend(handles=handles, labels=labels, bbox_to_anchor=(1, .8),
          labelspacing=.9, borderpad=.7);
```



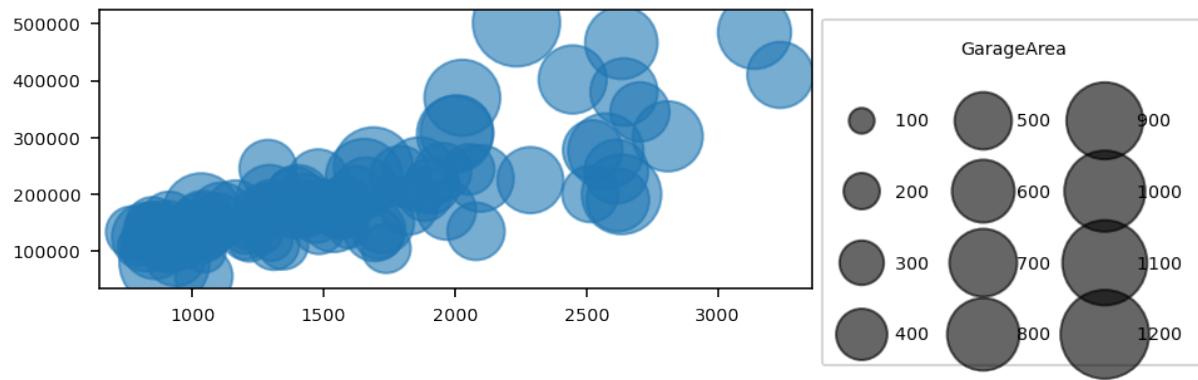
Let's go back to our housing dataset and use the values of garage area as the size of our points. Before doing so, we need to get an understanding of how large the points will be. Below, we calculate the figure-inches for the points that will represent the smallest and largest garage areas. We also return the garage area of the house that was set as 10,000 square feet to its original value.

```
[33]: housing.loc[67, 'GarageArea'] = 812
housing['GarageArea'].agg(['min', 'max']) ** .5 / 72
```

```
[33]: min      0.000000
max     0.490653
Name: GarageArea, dtype: float64
```

The houses with no garage area won't appear on the graph, while the largest garage will have a point that is about half of a figure-inch, which doesn't seem too large. Let's go ahead and make the plot and assess its appearance.

```
[34]: fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', s='GarageArea', alpha=.6, □
                        ↴data=housing)
handles, labels = scatter_obj.legend_elements(prop='sizes')
ax.legend(handles=handles, labels=labels, title='GarageArea', bbox_to_anchor=(1, 1),
          labelspacing=3, borderpad=1.2, ncol=3);
```



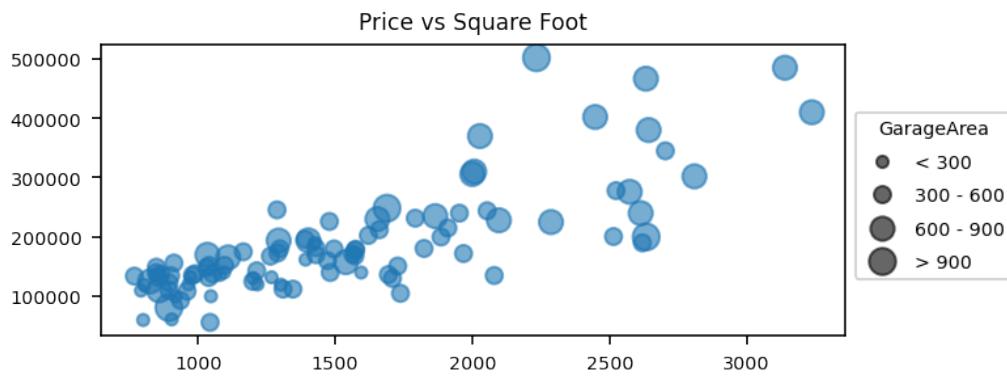
The points begin too large and there are far too many of them in the legend. It's not likely that your column of data will happen to align nicely to typographical points squared. Most of the time, you'll need to manually transform your data so that you get the exact point size you desire. One fairly simple way to do this is with the pandas `cut` function, which categorizes numeric data into bins. Pass it the Series you want to bin, the edges of the bins, and the labels for each bin. Use the marker size in points squared as the labels.

```
[35]: ga_bins = pd.cut(housing['GarageArea'], bins=[-1, 300, 600, 900, 1500],
                      labels=[20, 40, 80, 100])
ga_bins.head()
```

```
[35]: 0    20
1    40
2    40
3    40
4    40
Name: GarageArea, dtype: category
Categories (4, int64): [20 < 40 < 80 < 100]
```

We created four categories with five edge points for garage areas less than 300, between 300 and 600, between 600 and 900, and greater than 900 with corresponding integer values of 20, 40, 60, and 80 for the marker size. A list of descriptive strings is manually created for the legend.

```
[36]: fig, ax = plt.subplots(figsize=(5, 2))
scatter_obj = ax.scatter('GrLivArea', 'SalePrice', s=ga_bins, alpha=.6, data=housing)
handles, _ = scatter_obj.legend_elements(prop='sizes')
labels = ['< 300', '300 - 600', '600 - 900', '> 900']
ax.legend(handles=handles, labels=labels, title='GarageArea', bbox_to_anchor=(1, .8),
           labelspacing=.8)
ax.set_title('Price vs Square Foot');
```



65.3 Bar plots

Standard bar plots are a series of rectangles with a bottom on the x-axis. Each bar is defined by an x-value and a height. In matplotlib, bar plots involving only one group of data are fairly easy to produce, but become substantially more difficult when multiple groups are involved. Let's use the City of Houston employee dataset for the bar plot examples.

```
[37]: emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
emp.head(3)
```

	dept	title	hire_date	salary	sex	race
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black

Let's create a bar plot of the frequency of each department.

```
[38]: dept_counts = emp['dept'].value_counts()
dept_counts
```

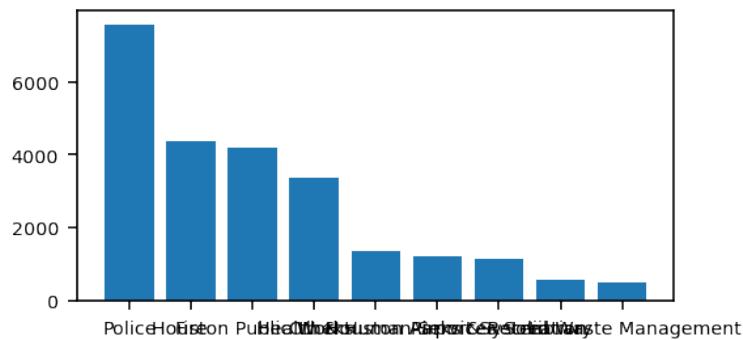
```
[38]: Police          7573
Fire            4376
Houston Public Works 4190
Other           3373
Health & Human Services 1353
Houston Airport System 1216
Parks & Recreation    1152
Library          563
Solid Waste Management 512
Name: dept, dtype: int64
```

To create a bar plot, you must supply both the `x` and the `height` parameters. Let's assign the unique departments to the variable name `x` and the values to `height`.

```
[39]: x = dept_counts.index
height = dept_counts.values
```

We can now call the `bar` method to plot the counts of each department. Like usual, we assign the result of the plotting method to a variable.

```
[40]: fig, ax = plt.subplots()
bar_obj = ax.bar(x=x, height=height)
```



Let's get the type of object returned.

```
[41]: type(bar_obj)
```

```
[41]: matplotlib.container.BarContainer
```

This `BarContainer` object is essentially a list of each of the bars. Nine bars were created, one for each department. We verify this with the `len` function.

```
[42]: len(bar_obj)
```

```
[42]: 9
```

Each individual item is a rectangle patch.

```
[43]: bar0 = bar_obj[0]
type(bar0)
```

```
[43]: matplotlib.patches.Rectangle
```

We've created these kinds of patches before. Let's use some of the getter methods to retrieve some properties.

```
[44]: bar0.get_width()
```

```
[44]: 0.8
```

```
[45]: bar0.get_xy()
```

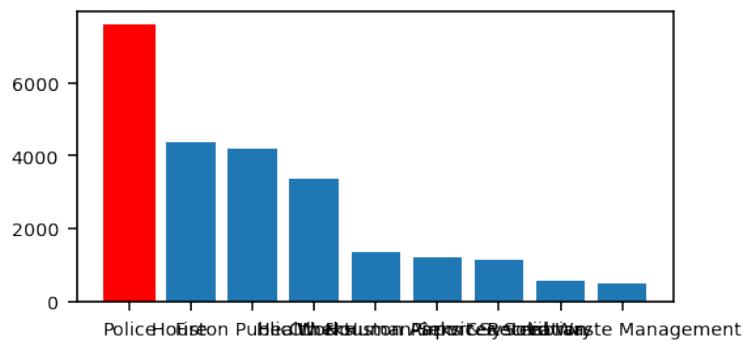
```
[45]: (-0.4, 0)
```

```
[46]: bar0.get_height()
```

```
[46]: 7573
```

We can set this first bar to a different color and output the figure to verify the change.

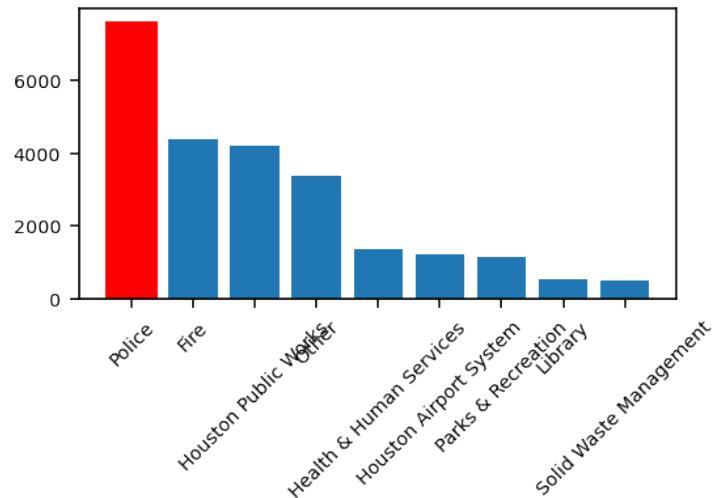
```
[47]: bar0.set_color('red')
fig
```



Rotating the ticks labels

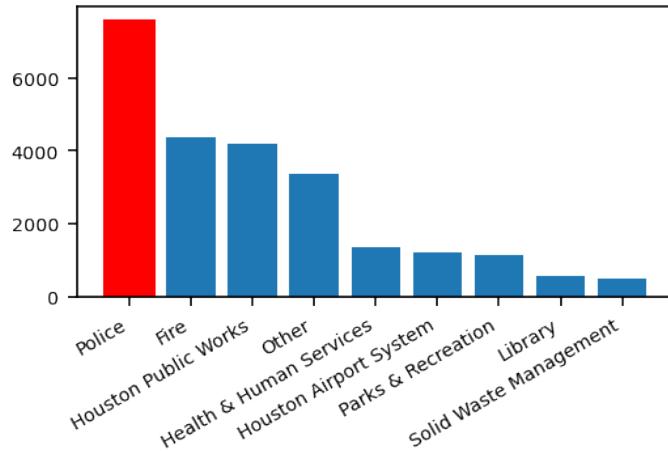
The tick labels overlap each other. Let's use the `tick_params` method to rotate the labels 45 degrees.

```
[48]: ax.tick_params(axis='x', labelrotation=45)
fig
```



The labels are text objects with center horizontal alignment making them difficult to determine which bar they reference when rotated. They would be better suited if they were aligned from the right. Unfortunately, there is no way to set the horizontal alignment with the `tick_params` method. You'll have to loop through each label individually and call its setter methods. The `get_xticklabels` method returns a list-like object of each label as a matplotlib text object. We rotate the labels a bit less and align them horizontally from the right.

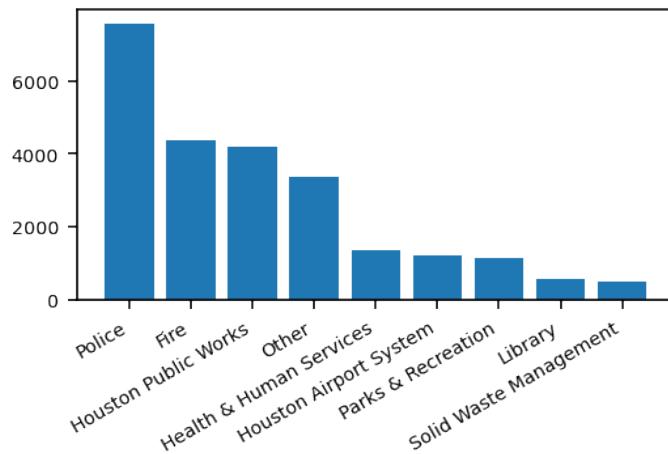
```
[49]: for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')
fig
```



Set rotation, alignment, and all other text properties with `set_xticklabels`

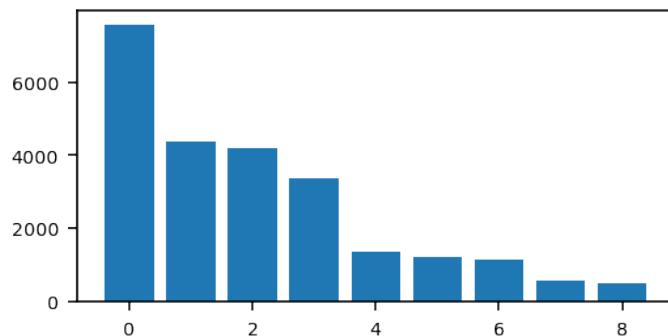
If the tick labels are already in a list or a sequence like a pandas index, then you can use the `set_xticklabels` method to change all of their text properties without a loop.

```
[50]: fig, ax = plt.subplots()
bar_obj = ax.bar(x=x, height=height)
ax.set_xticklabels(x, rotation=30, ha='right');
```



We used strings for the x-values, but these are mapped directly to integers beginning at 0. We could have duplicated the bars by using a sequence of integers instead. Notice that not all tick values are present. We'll learn in a future chapter how to place ticks at particular intervals.

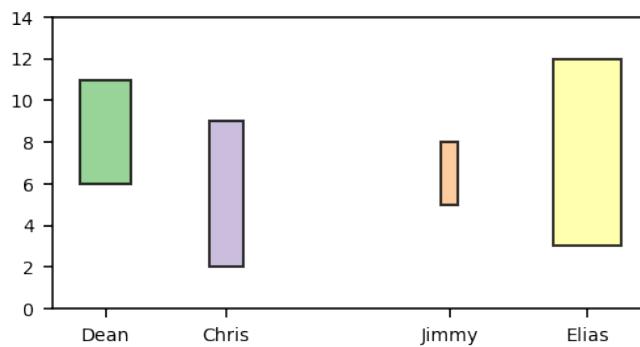
```
[51]: fig, ax = plt.subplots()
ax.bar(x=np.arange(9), height=height);
```



Customizing every bar

Every property of every bar can be customized. The width of each bar can be set with the `width` parameter (default .8). The starting y-value of each bar is set to 0 by default, but can be changed with the `bottom` parameter. Below, we create four bars of all different sizes and starting positions. The first four colors of the Accent colormap are used for the colors and the labels are set with the `tick_label` parameter.

```
[52]: x = [-12, -5, 8, 16]
height = [5, 7, 3, 9]
width = [3, 2, 1, 4]
bottom = [6, 2, 5, 3]
color = cm.Accent(range(4))
tick_labels = ['Dean', 'Chris', 'Jimmy', 'Elias']
fig, ax = plt.subplots()
ax.set_ylim(0, 14)
ax.bar(x=x, height=height, width=width, bottom=bottom, alpha=.8, color=color,
       ec='black', tick_label=tick_labels);
```



Plotting groups of bars

Plotting multiple groups of bars requires us to carefully place each set of bars at a precise location along the x-axis with a width that does not overlap the other bars. Let's first calculate the average salary for every department and race.

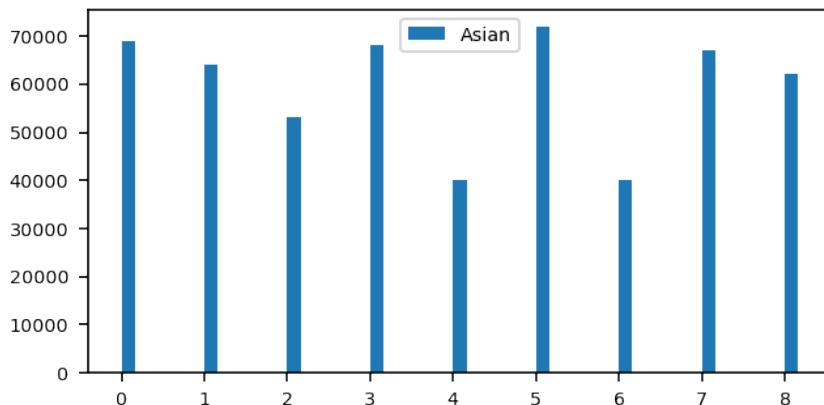
```
[53]: pt = emp.pivot_table(index='dept', columns='race',
                           values='salary', aggfunc='mean').round(-3)
pt.head(3)
```

	race	Asian	Black	Hispanic	Native American	White
dept						
Fire	69000.0	60000.0	57000.0		62000.0	62000.0
Health & Human Services	64000.0	57000.0	47000.0		45000.0	66000.0
Houston Airport System	53000.0	51000.0	46000.0		60000.0	69000.0

Each column can be plotted as bars to represent the average salary of each department for that race. A total of five groups of bars, one for each race, will be created. To accommodate each bar, we'll reduce the width to one-fifth of its original value.

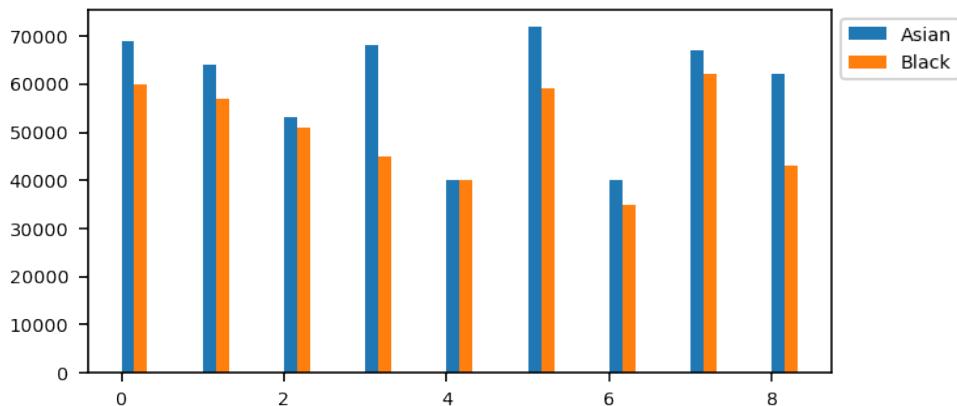
Below, we select the first column ('Asian') as a Series for the height and calculate the new width. The `label` parameter is set to the name of the race so that it appears in the legend. By default, the bars are centered around the given x-value. When the `align` parameter is set to '`edge`', the left edge of the bar will begin at the x-value.

```
[54]: m, n = pt.shape
x = np.arange(m)
height = pt['Asian']
orig_width = .8
width = orig_width / n
fig, ax = plt.subplots(figsize=(5, 2.5))
ax.bar(x, height, width=width, align='edge', label='Asian')
ax.legend();
```



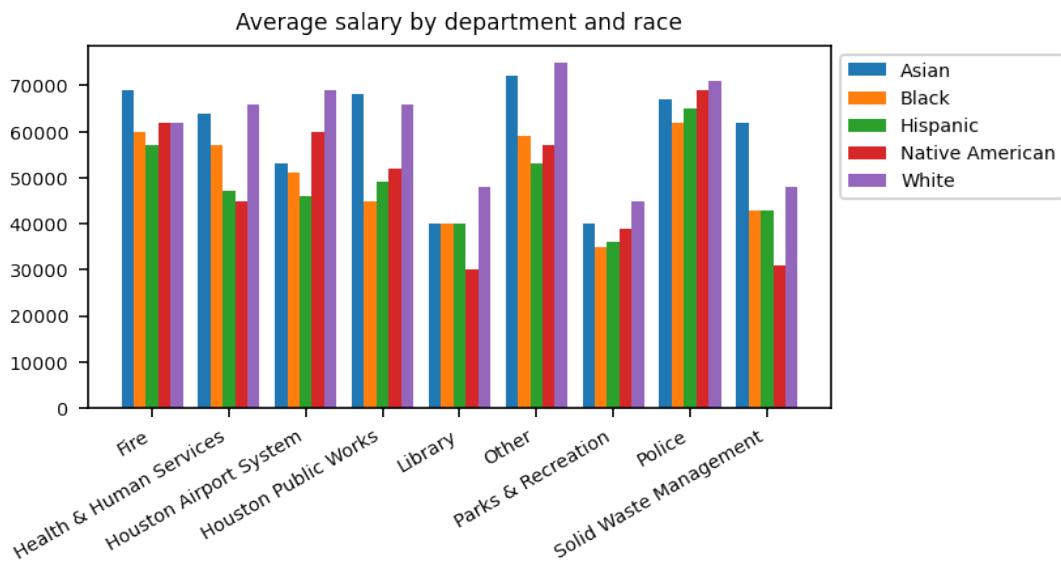
The x-values for the next column group need to be shifted to the right by the width of the bars so that they begin exactly where the last bar ends. We add the width to every x-value and then plot the next group. The `legend` method must be called again to update it.

```
[55]: height = pt['Black']
ax.bar(x + width, height, width=width, align='edge', label='Black')
ax.legend(bbox_to_anchor=(1, 1), loc='upper left')
fig
```



Instead of adding each group of bars one at a time, we can use a loop to iterate through the column names, calculating the new x-values. The tick marks are placed in the center of each group and given the department names as labels. The `set_xticklabels` method is used to rotate and align the labels.

```
[56]: fig, ax = plt.subplots(figsize=(5, 2.5))
for i, col in enumerate(pt.columns):
    x_new = x + width * i
    ax.bar(x_new, pt[col], width=width, label=col, align='edge')
ax.legend(bbox_to_anchor=(1, 1))
ax.set_xticks(x + orig_width / 2)
ax.set_xticklabels(pt.index, rotation=30, ha='right')
ax.set_title('Average salary by department and race');
```



Stacked bar plot

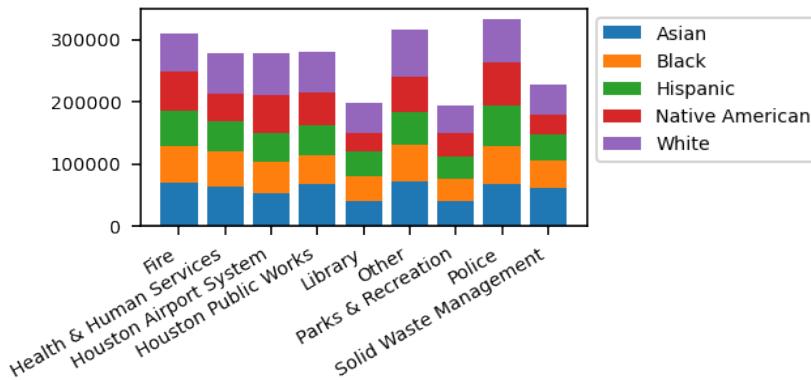
Instead of placing the bars directly next to each other, they can be stacked one on top of the other to create a stacked bar plot. In this instance, the `bottom` parameter must be used and updated each iteration. It is initially set to zero, but then accumulated for each group. The default width can be used.

```
[57]: fig, ax = plt.subplots(figsize=(3, 1.5))
bottom = np.zeros(9)
for col in pt.columns:
```

```

ax.bar(x, pt[col], bottom=bottom, label=col)
bottom += pt[col]
ax.legend(bbox_to_anchor=(1, 1))
ax.set_xticks(x)
ax.set_xticklabels(pt.index, rotation=30, ha='right');

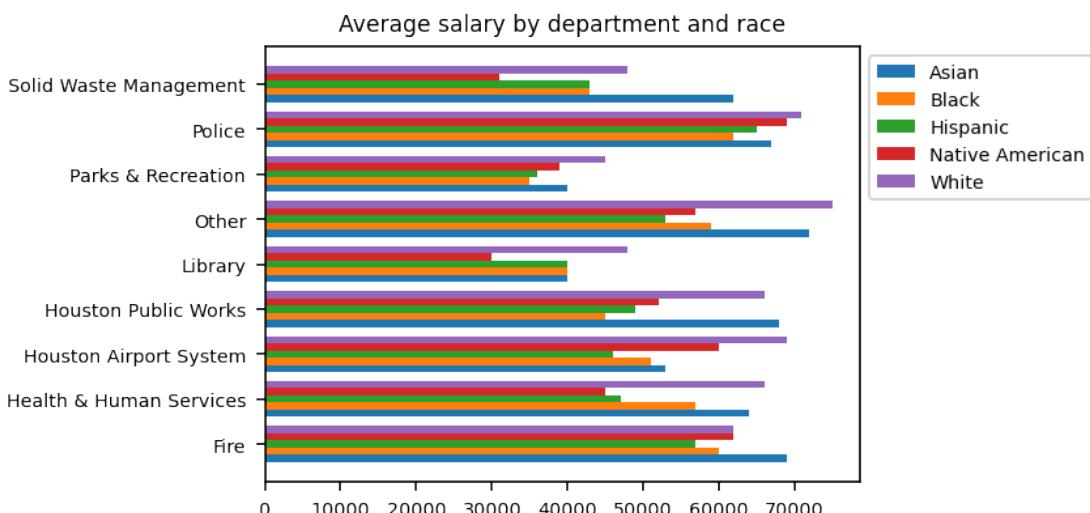
```



Horizontal bar plots

Horizontal bar plots are created with the `barrh` method in nearly the same manner. Only the names of the parameters are different. The same code from above is copied below. The parameters `x`, `width`, and `height` have been substituted with `y`, `height`, and `width`.

```
[58]: fig, ax = plt.subplots(figsize=(4, 3))
for i, col in enumerate(pt.columns):
    x_new = x + width * i
    ax.barrh(y=x_new, width=pt[col], height=width, label=col, align='edge')
ax.legend(bbox_to_anchor=(1, 1))
ax.set_yticks(x + orig_width / 2)
ax.set_yticklabels(pt.index)
ax.set_title('Average salary by department and race');
```



65.4 Exercises

Exercise 1

Read in the bikes dataset and select 500 rows of data at random. Filter for rides with a trip duration less than the 95th percentile. Remove any rows that have obviously bad data for temperature and wind speed. Make a scatter plot with temperature and trip duration as the x and y variables. Color by gender and size by wind speed using a qualitative color map. Use [this tutorial](#) to create two separate legends, one for gender, and the other for wind speed.

[]:

Chapter 66

Matplotlib Distribution Plots

There are various statistical methods available to understand how a sequence of data is distributed. The mean, median, variance, minimum, maximum, and various quantiles are all examples of statistics that describe how data is distributed. Often times, the best way to understand how data is distributed is through a visualization. In this chapter, we will visualize distributions of numeric columns of data with matplotlib. Let's begin by reading in a few of the columns of the housing dataset that all have units of square feet.

```
[1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('..../mdap.mplstyle')
cols = ['TotalBsmtSF', '1stFlrSF', '2ndFlrSF',
        'GrLivArea', 'GarageArea', 'WoodDeckSF']
housing = pd.read_csv('../data/housing.csv', usecols=cols)
housing.head(3)
```

	TotalBsmtSF	1stFlrSF	2ndFlrSF	GrLivArea	GarageArea	WoodDeckSF
0	856	856	854	1710	548	0
1	1262	1262	0	1262	460	298
2	920	920	866	1786	608	0

The simplest way to get basic non-visual descriptive statistics is to use the `describe` method.

```
[2]: housing.describe(percentiles=[.01, .1, .25, .5, .75, .9, .99]) \
    .style.format('{:.0f}')
```

	TotalBsmtSF	1stFlrSF	2ndFlrSF	GrLivArea	GarageArea	WoodDeckSF
count	1,460	1,460	1,460	1,460	1,460	1,460
mean	1,057	1,163	347	1,515	473	94
std	439	387	437	525	214	125
min	0	334	0	334	0	0
1%	0	520	0	692	0	0
10%	637	757	0	912	240	0
25%	796	882	0	1,130	334	0
50%	992	1,087	0	1,464	480	0
75%	1,298	1,391	728	1,777	576	168
90%	1,602	1,680	954	2,158	757	262
99%	2,155	2,219	1,419	3,123	1,003	505
max	6,110	4,692	2,065	5,642	1,418	857

66.1 Histograms

A histogram is one of the most common methods to visualize the distribution of a single column of numeric data. A histogram is constructed by first binning the numeric data into some number of bins and then counting the values that fall within each bin. Each bin is then plotted as a rectangular bar with width stretching from the left to right edge of the bin and the height equal to the count. Before we create a histogram with matplotlib, let's create the necessary summary statistics with pandas.

We use the `cut` function to create 10 bins of equal size of the `TotalBsmtSF` column. The bin widths are determined by finding the range of the distribution (maximum minus minimum) and dividing by the number of bins. The `precision` parameter controls the number of decimal places to use for the bins and is set to 0 to return integers. By default, bins include values that equal their right edge exactly and exclude them if they equal their left edge. We set `right` to `False` to reverse this so that it matches how matplotlib does its bin calculations.

```
[3]: tbsf_bins = pd.cut(housing['TotalBsmtSF'], bins=10, precision=0, right=False)
tbsf_bins.head(3)
```

```
[3]: 0    [611.0, 1222.0)
1    [1222.0, 1833.0)
2    [611.0, 1222.0)
Name: TotalBsmtSF, dtype: category
Categories (10, interval[float64]): [[0.0, 611.0) < [611.0, 1222.0) < [1222.0, 1833.0) <
[1833.0, 2444.0) ... [3666.0, 4277.0) < [4277.0, 4888.0) < [4888.0, 5499.0) < [5499.0,
6116.0)]
```

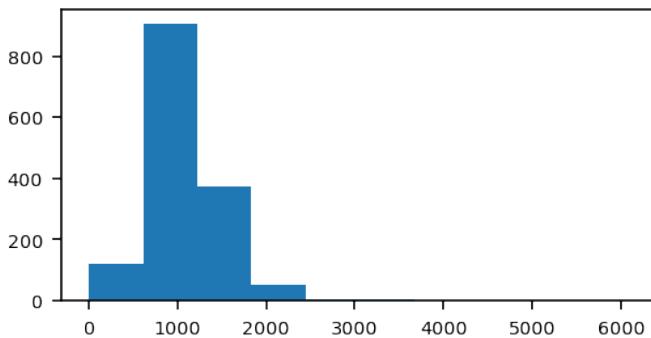
We can then call the `value_counts` method on this Series to get the counts within each bin. By default, the returned Series is sorted by count in descending order. With histograms, we want the bins (the index in this case) to be sorted, so we set `sort` to `False`.

```
[4]: tsbf_bin_count = tsbf_bins.value_counts(sort=False)
tsbf_bin_count
```

```
[4]: [0.0, 611.0)      121
[611.0, 1222.0)    907
[1222.0, 1833.0)   372
[1833.0, 2444.0)   52
[2444.0, 3055.0)   3
[3055.0, 3666.0)   4
[3666.0, 4277.0)   0
[4277.0, 4888.0)   0
[4888.0, 5499.0)   0
[5499.0, 6116.0)   1
Name: TotalBsmtSF, dtype: int64
```

Let's return to matplotlib and use the `hist` axes method to create a histogram of `TotalBsmtSF`. It has a single required parameter, `x`, which may be set as a Series. By default, 10 bins are used.

```
[5]: fig, ax = plt.subplots()
hist_return = ax.hist(housing['TotalBsmtSF'])
```



The bar heights and bin widths appear to match the counts from pandas. The `hist` method returns the bin counts (height), bin edges, and the patches used to create the rectangles as a three-item tuple. Let's unpack this into three separate variables.

```
[6]: counts, edges, patches = hist_return
```

The edges are contained in a one-dimensional numpy array. There will always be one more edge value than the number of bins. Let's verify that they match the pandas intervals from above.

```
[7]: edges
```

```
[7]: array([ 0.,  611., 1222., 1833., 2444., 3055., 3666., 4277., 4888.,
 5499., 6110.])
```

Only the last value is different, and this is because pandas always adds 1% to either the first or last bin to include the minimum or maximum value. With matplotlib, the first and last bins have both the left and right edges inclusive. Let's now verify the counts are the same.

```
[8]: counts
```

```
[8]: array([121., 907., 372., 52., 3., 4., 0., 0., 0., 1.])
```

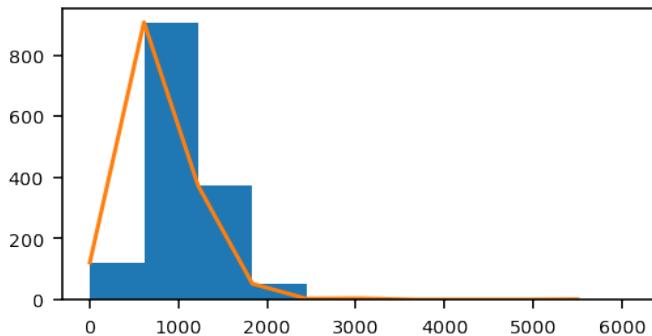
The patch object is a list-like object of all the 10 rectangle patches.

```
[9]: patches
```

```
[9]: <a list of 10 Patch objects>
```

We can use the `edges` and `counts` arrays to create a line plot connecting the upper left corners of each bar. The extra last edge is not included as there is one more edge than count.

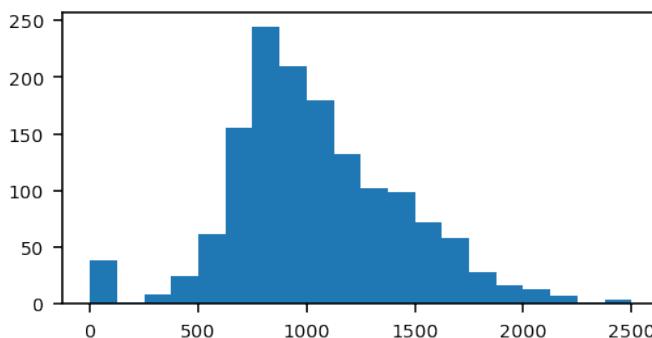
```
[10]: ax.plot(edges[:-1], counts)
fig
```



Customizing the histogram

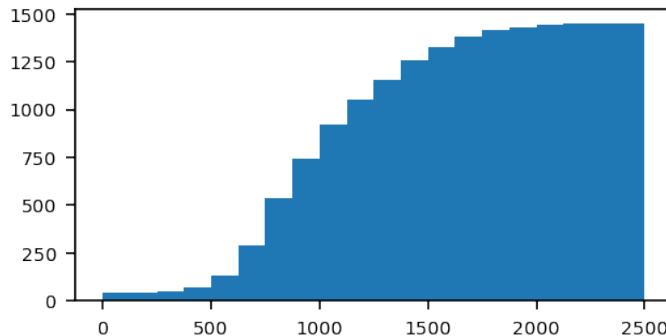
There are a large number of parameters available to the `hist` method to customize the appearance of the histogram. Our dataset has one extreme value several times larger than the others that's leaving the right side of the plot empty. We can set the `range` parameter to a two-item tuple of the minimum and maximum values to consider. The number of bins is controlled by the `bins` parameter and set to 20.

```
[11]: fig, ax = plt.subplots()
ax.hist(housing['TotalBsmtSF'], range=(0, 2500), bins=20);
```



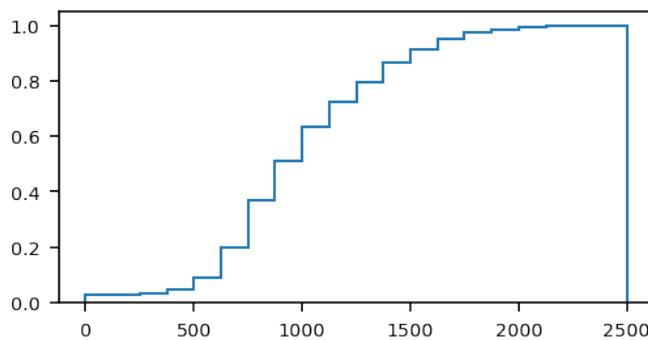
Instead of showing the count of each bin, the bar heights can equal the accumulated total up through that bin by setting the `cumulative` parameter to `True`.

```
[12]: fig, ax = plt.subplots()
ax.hist(housing['TotalBsmtSF'], range=(0, 2500), bins=20, cumulative=True);
```



Instead of showing the raw counts, the relative frequency can be shown by setting `density` to `True`. An outline of the heights can be drawn by setting the `histtype` to `'step'`.

```
[13]: fig, ax = plt.subplots()
ax.hist(housing['TotalBsmtSF'], range=(0, 2500), bins=20,
        cumulative=True, density=True, histtype='step');
```



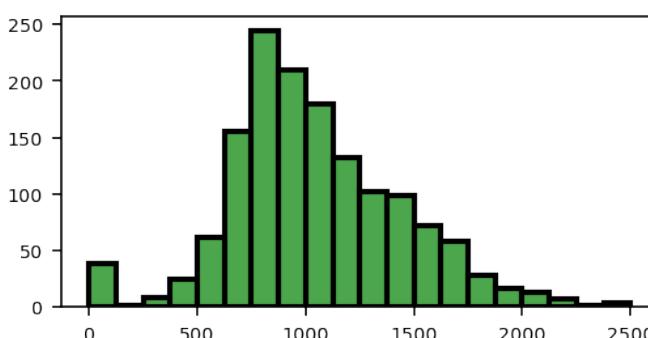
According to the histogram above, around 50% of homes have less than 1,000 total basement square feet. We can verify this with pandas.

```
[14]: (housing['TotalBsmtSF'] < 1000).mean()
```

```
[14]: 0.5089041095890411
```

All properties available to matplotlib patches are able to be set within the `hist` method as well. We set the face and edge colors and edge width below.

```
[15]: fig, ax = plt.subplots()
ax.hist(housing['TotalBsmtSF'], range=(0, 2500), bins=20,
        fc=(0, .5, 0, .7), ec='black', lw=2);
```



Multiple histograms

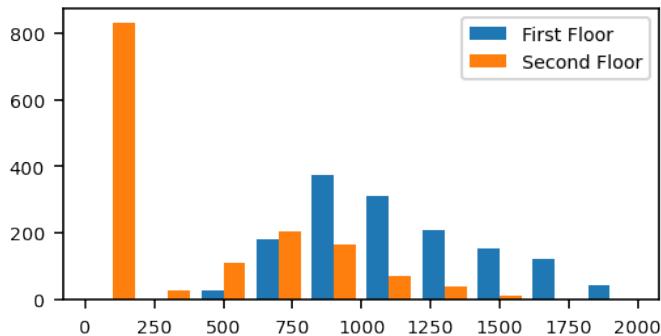
It's possible to simultaneously plot two or more independent histograms on the same axes. To do so, select the columns you want and retrieve the underlying numpy array with the `values` attribute. The `hist` method doesn't work well when passing it a DataFrame. Here, we select the first and second floor square footage columns and output the values for the first three homes.

```
[16]: x = housing[['1stFlrSF', '2ndFlrSF']].values
x[:3]
```

```
[16]: array([[ 856,  854],
       [1262,      0],
       [ 920,  866]])
```

This array is passed as the first argument to `hist`. The `label` parameter works differently than other plotting methods and can be set as a list that corresponds to each column. Each histogram is plotted with bars of the same color corresponding to the color cycle.

```
[17]: fig, ax = plt.subplots()
ax.hist(x, label=['First Floor', 'Second Floor'], range=(0, 2000))
ax.legend();
```



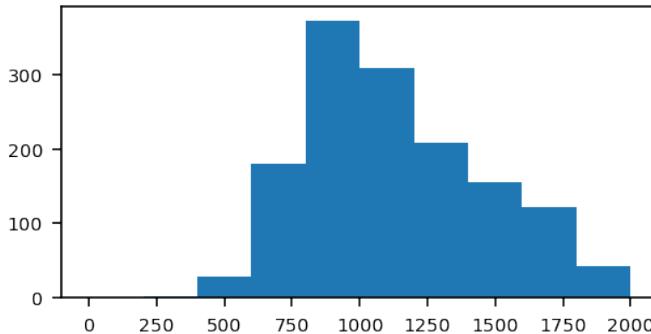
The bin edges are computed by first finding the minimum and maximum of the whole dataset regardless of the column.

```
[18]: housing[['1stFlrSF', '2ndFlrSF']].agg(['min', 'max'])
```

	1stFlrSF	2ndFlrSF
min	334	0
max	4692	2065

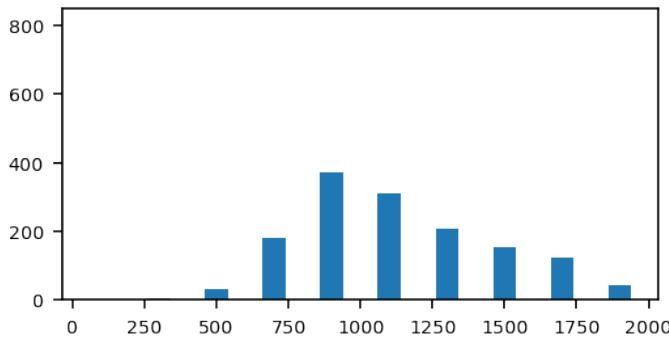
The minimum of 0 comes from the second floor with the maximum coming from the first floor. The entire range would have been (0, 4,692) if it had not been set explicitly. Each bar is about half of the width of what it would have been if it were the only column plotted. matplotlib automatically decreases the width so that each bar fits within the size of the bin. Let's plot a histogram of just the first floor so a comparison can be made.

```
[19]: fig, ax = plt.subplots()
ax.hist(housing['1stFlrSF'], range=(0, 2000));
```



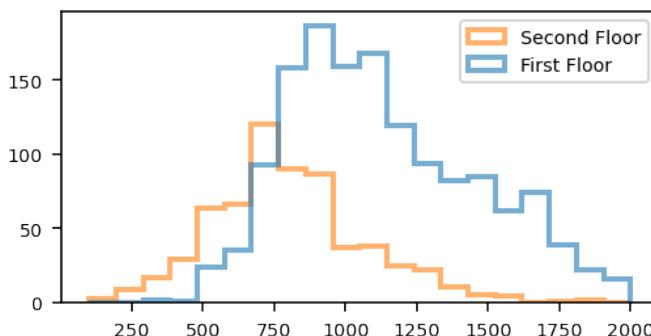
It's possible to plot this single histogram as it appeared in the plot above with both histograms by expanding the y-limit and using `rwidth` (relative width, defaulted to 1) to shrink the width.

```
[20]: fig, ax = plt.subplots()
ax.hist(housing['1stFlrSF'], range=(0, 2000), rwidth=.4)
ax.set_ylim(0, 850);
```



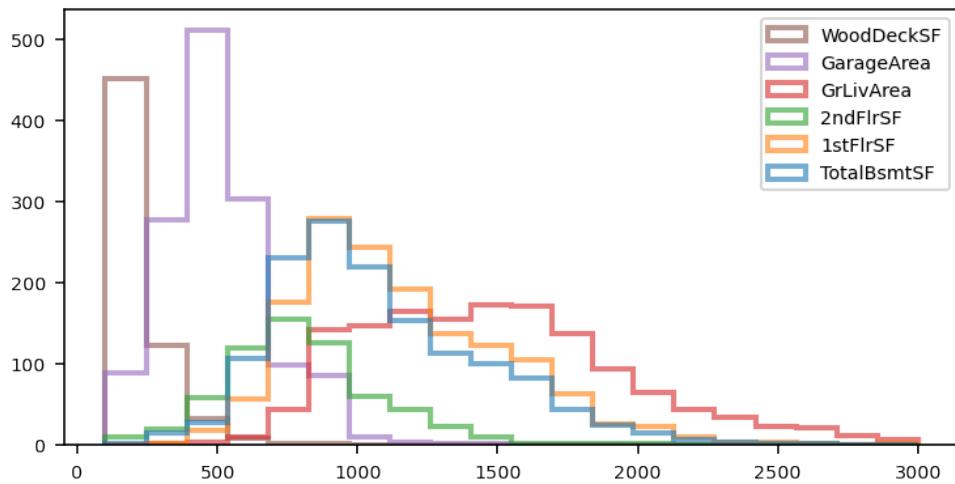
The plot with histograms for both variables is difficult to interpret. I prefer changing `histtype` to '`step`', which will use the entire width of the bin as the width of each horizontal step. We exclude the houses without a second floor below (those with 0 second floor square feet) with the `range` parameter.

```
[21]: fig, ax = plt.subplots()
ax.hist(x, label=['First Floor', 'Second Floor'], bins=20, range=(100, 2000),
        histtype='step', alpha=.6, lw=2)
ax.legend();
```



Histograms for each column can be plotted at once, though each additional column makes the plot considerably more difficult to read. The kernel density estimate plot, which will be introduced in the next part of the book, is a better choice when comparing multiple distributions like this.

```
[22]: fig, ax = plt.subplots(figsize=(6, 3))
ax.hist(housing.values, bins=20,label=housing.columns, range=(100, 3000),
        histtype='step', alpha=.6, lw=2)
ax.legend();
```



Setting the data parameter to a DataFrame

It's possible to set the `data` parameter to a DataFrame and use the column name as a string, but that syntax only works for a single column and cannot be extended to create multiple histograms.

66.2 Box and whisker plots

Box and whisker plots provide a different visual display for analyzing the distribution of a single column of data. A box plot is created by finding the 25th, 50th, and 75th percentiles of the distribution, which are also known as the first quartile, median, and third quartile. Let's do this manually for the `TotalBsmtSF` column.

```
[23]: quartiles = housing['TotalBsmtSF'].quantile([.25, .5, .75])
quartiles
```

```
[23]: 0.25    795.75
      0.50   991.50
      0.75  1298.25
Name: TotalBsmtSF, dtype: float64
```

The **interquartile range** (IQR) is calculated, which is the difference between the third and first quartiles.

```
[24]: q1 = quartiles.iloc[0]
median = quartiles.iloc[1]
q3 = quartiles.iloc[2]
iqr = q3 - q1
iqr
```

```
[24]: 502.5
```

The lower and upper whiskers are a distance of 1.5 times the IQR beyond their respective quartile.

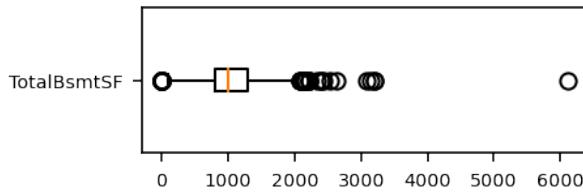
```
[25]: lower_whisker = q1 - 1.5 * iqr
       upper_whisker = q3 + 1.5 * iqr
       lower_whisker, upper_whisker
```

[25]: (42.0, 2052.0)

When creating a horizontal box and whisker plot, the “box” portion is formed by creating a rectangle that has a left edge at the first quartile and a right edge at the third quartile. A vertical line is placed at the median. The height of the rectangle is not consequential. Horizontal lines are drawn protruding from the left and right edges of the boxes to their respective whisker location. Any individual points outside of the whiskers are called **fliers** and are drawn individually.

Let’s use the `boxplot` method to create a horizontal box, whiskers, and fliers. It has a single required parameter, `x`, the sequence of values, which we set as a Series. By default, the box plot is created vertically, but setting `vert` to `False` makes it horizontal. The `labels` parameter can be set to a list of the column names. For now, we just have one.

```
[26]: fig, ax = plt.subplots(figsize=(3, 1))
x = housing['TotalBsmtSF']
box_return = ax.boxplot(x, vert=False, labels=['TotalBsmtSF'])
```



Box plots help identify extreme points better than histograms as we can clearly see exactly where they are located. The returned object is a dictionary containing six different groups of plotting objects. Let’s take a look at the different groups by accessing the keys.

```
[27]: box_return.keys()
```

[27]: dict_keys(['whiskers', 'caps', 'boxes', 'medians', 'fliers', 'means'])

Let’s retrieve the whiskers and see what type of object they are.

```
[28]: whiskers = box_return['whiskers']
whiskers
```

```
[28]: [matplotlib.lines.Line2D at 0x11e225990>,
<matplotlib.lines.Line2D at 0x11e225ed0>]
```

Both whiskers are plotted as a lines from the box edge to the whisker value. Let’s verify that the left whisker begins at the third quartile (795.75) and ends at the whisker (42) by getting the x-values.

```
[29]: whiskers[0].get_xdata()
```

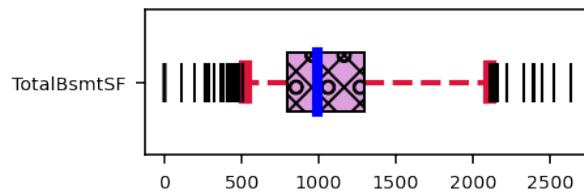
```
[29]: array([795.75, 105. ])
```

There’s a mismatch between our calculated value of the whisker and the location matplotlib used. This is because no house had total basement square feet between 42 and 105, so matplotlib uses the last known value less than the calculated whisker.

More box plot parameters

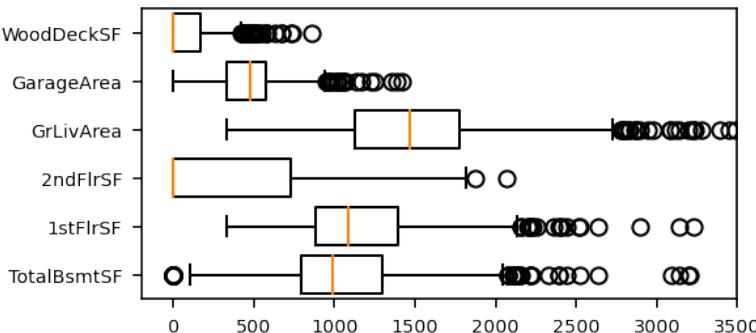
As usual, there are many different parameters that can be set to control every aspect of the box plot. Here, we use `whis` to set the whiskers at the 5th and 99th percentiles. The width of the box is increased and the box is turned into a patch object. We then change the properties of five distinct plotting elements.

```
[30]: fig, ax = plt.subplots(figsize=(3, 1))
x = housing.query('TotalBsmtSF < 3_000')['TotalBsmtSF']
ax.boxplot(x, vert=False, labels=['TotalBsmtSF'],
            whis=(5, 99), widths=.4, patch_artist=True,
            boxprops={'hatch': 'xox', 'fc': 'plum'},
            flierprops={'marker': '|', 'ms': 15},
            whiskerprops={'lw': 2, 'c': 'crimson', 'ls': '--'},
            medianprops={'lw': 4, 'c': 'blue'},
            capprops={'c': 'crimson', 'lw': 5});
```



All of the columns can be plotted simultaneously as individual box and whisker plots. By default, each box is placed exactly one y (or x) unit apart.

```
[31]: fig, ax = plt.subplots()
x = housing.values
ax.boxplot(x, vert=False, labels=housing.columns, widths=.8)
ax.set_xlim(-200, 3_500);
```



66.3 Exercises

[]:

Chapter 67

Best of the Rest of Matplotlib

matplotlib is an enormous library with tremendous capabilities to visualize almost anything. The previous chapters helped lay a strong foundation to help you understand the most fundamental concepts of matplotlib that arise nearly every time you use it. In this chapter, many more useful concepts will be covered.

67.1 Axes spines

Each of the four sides of the rectangular axes border are known collectively as **spines** and can be accessed individually through the **spines** attribute. Each spine is like a line in which you can control its color, width, and style. Let's begin by creating a scatter plot of sale price versus living area from our housing data.

```
[1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('..../mdap.mplstyle')
housing = pd.read_csv('../data/housing.csv')
fig, ax = plt.subplots(figsize=(6, 2))
ax.scatter('GrLivArea', 'SalePrice', data=housing, s=8, alpha=.6)
ax.set_title('Sale Price vs Living Area')
ax.set_xlabel('Living Area')
ax.set_ylabel('Sale Price');
```



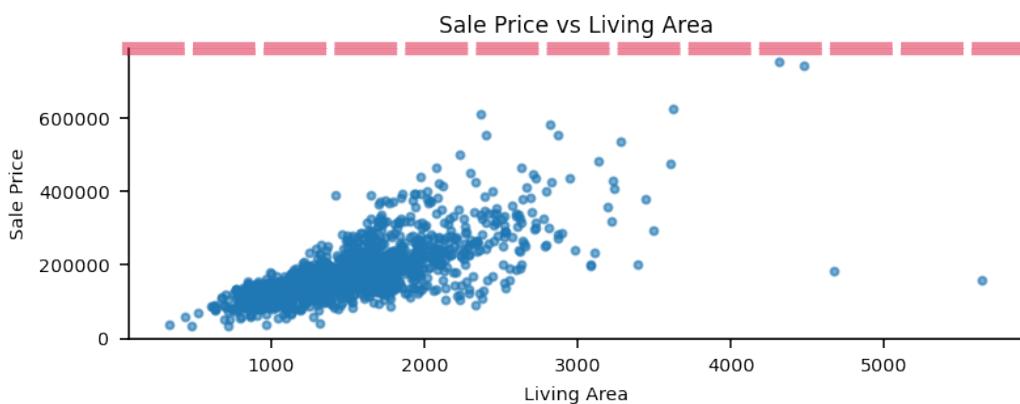
The spines are stored as an ordered dictionary with their position as the keys.

```
[2]: ax.spines
```

```
[2]: OrderedDict([('left', <matplotlib.spines.Spine at 0x11a33c550>),
                 ('right', <matplotlib.spines.Spine at 0x11ad7d7d0>),
                 ('bottom', <matplotlib.spines.Spine at 0x11ad7d950>),
                 ('top', <matplotlib.spines.Spine at 0x11ad7dad0>)])
```

Here, the top spine is selected and several of its properties are changed with its setter methods.

```
[3]: top_spine = ax.spines['top']
top_spine.set_color('crimson')
top_spine.set_alpha(.5)
top_spine.set_lw(5)
top_spine.set_ls('---')
fig
```



Making spines invisible

All matplotlib objects can be toggled between visible and invisible with the `set_visible` method, which takes a boolean. This is different from the `remove` method, which completely removes objects from the axes. The object still exists when it is made invisible. The seaborn library, which uses matplotlib, often removes spines for its plots. Let's do the same by making them invisible.

```
[4]: top_spine.set_visible(False)
ax.spines['right'].set_visible(False)
fig
```



67.2 The `xaxis` and `yaxis` objects

Nearly all of our methods thus far have been called from axes objects. There are two other objects within the axes, the `xaxis` and `yaxis`, that allow for a little more customization not available directly from the axes. These two objects provide the ability to make changes to the axis labels, tick locations, tick lines, and tick labels. Most of this functionality is already provided by the axes methods, so it's rare that you'll access these objects. However, there are some methods that provide unique functionality not available from axes methods. It's important to note that the `xaxis` and `yaxis` objects have nothing to do with the axes spines. Let's access the `xaxis` object as an attribute.

```
[5]: ax.xaxis
```

```
[5]: <matplotlib.axis.XAxis at 0x11ad7d710>
```

Overlapping methods

The majority of methods available to both these objects have nearly identical counterparts in the axes with nearly the same name. Below, we get the location and labels of the ticks, and the axis labels using both the axis and axes methods.

```
[6]: ax.xaxis.get_ticklocs()
```

```
[6]: array([ 0., 1000., 2000., 3000., 4000., 5000., 6000.])
```

```
[7]: ax.get_xticks()
```

```
[7]: array([ 0., 1000., 2000., 3000., 4000., 5000., 6000.])
```

```
[8]: list(ax.xaxis.get_ticklabels())
```

```
[8]: [Text(0.0, 0, '0'),
      Text(1000.0, 0, '1000'),
      Text(2000.0, 0, '2000'),
      Text(3000.0, 0, '3000'),
      Text(4000.0, 0, '4000'),
      Text(5000.0, 0, '5000'),
      Text(6000.0, 0, '6000')]
```

```
[9]: list(ax.get_xticklabels())
```

```
[9]: [Text(0.0, 0, '0'),
      Text(1000.0, 0, '1000'),
      Text(2000.0, 0, '2000'),
      Text(3000.0, 0, '3000'),
      Text(4000.0, 0, '4000'),
      Text(5000.0, 0, '5000'),
      Text(6000.0, 0, '6000')]
```

```
[10]: ax.yaxis.get_label_text()
```

```
[10]: 'Sale Price'
```

```
[11]: ax.get_ylabel()
```

```
[11]: 'Sale Price'
```

Unique `xaxis` and `yaxis` methods

I suggest accessing the `xaxis` and `yaxis` methods only when an axes method isn't available to do what you desire. Below, we use a couple methods unique to the `xaxis` and `yaxis` that move the label and ticks to the opposite side. We also make the right spine visible again. The upcoming sections contain more examples where the `xaxis` and `yaxis` must be accessed.

```
[12]: ax.yaxis.set_label_position('right')
ax.yaxis.set_ticks_position('right')
ax.spines['left'].set_visible(False)
ax.spines['right'].set_visible(True)
fig
```



67.3 Tick locators

For every plot that you create, matplotlib automatically sets the tick positions and tick labels for you. In this section, we'll cover how to use **tick locators** to add or remove ticks at precise intervals. Currently, there are five x-ticks, one for every 1,000 square feet of living area. matplotlib intelligently chose these ticks for us with an object called the `AutoLocator`. This is one of several classes of locators that give you control of the tick locations. You can retrieve the locator for each axis with the `get_major_locator` method available to both the `xaxis` and `yaxis` objects.

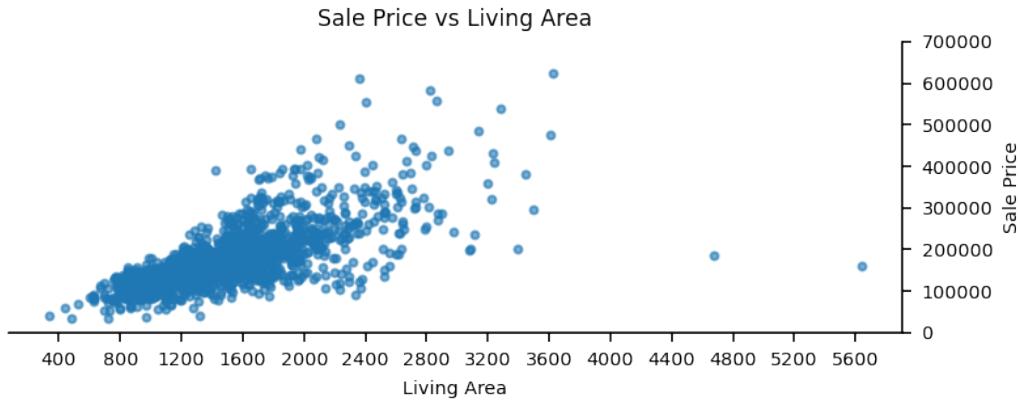
```
[13]: ax.xaxis.get_major_locator()
```

```
[13]: <matplotlib.ticker.AutoLocator at 0x11ace1b10>
```

The `ticker` module

The matplotlib `ticker` module provides several different locator classes that you can instantiate to control the location of the ticks. All of the tick locators can be found on [this page in the documentation](#). The `MultipleLocator` places ticks at every location that is a multiple of the number that it is instantiated with. Below, ticks are placed every 400 units along the x-axis. The `MaxNLocator` allows you to set the maximum number of ticks with the provided integer and is used for the y-axis to create 8 ticks.

```
[14]: from matplotlib import ticker
ax.xaxis.set_major_locator(ticker.MultipleLocator(400))
ax.yaxis.set_major_locator(ticker.MaxNLocator(nbins=8))
ax.set_ylim(0, 700_000)
fig
```

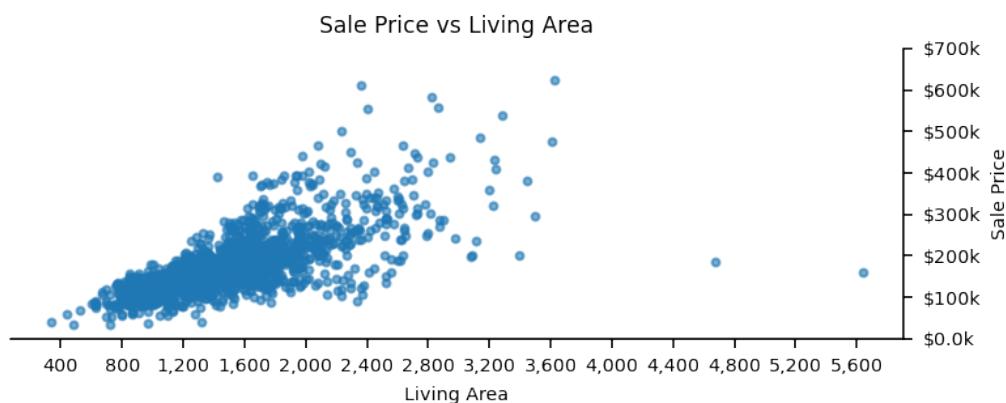


67.4 Tick formatters

Previously, the tick labels have been set manually by creating lists of strings or using the DataFrame `columns` attribute. Within the `ticker` module exist [many formatter classes](#) to change the display of the tick labels. These **tick formatter** classes are similar to the locators. A formatter class is instantiated and passed to the `set_major_formatter` method of the `xaxis` or `yaxis` object.

The `StrMethodFormatter` is a common formatter that allows you to change each label's appearance using syntax available to a string's `format` method. If you do not know the format string syntax, [visit the official Python documentation for help](#). The format specification is provided as a string within curly braces. `matplotlib` uses the variable name `x` which you have access to within the format. Below, we use a comma to separate every third digit of the x-tick labels and convert the y-tick labels into thousands of dollars.

```
[15]: ax.xaxis.set_major_formatter(ticker.StrMethodFormatter('{x:,.0f}'))
ax.yaxis.set_major_formatter(ticker.StrMethodFormatter('${x!s:.3}k'))
fig
```



67.5 Minor ticks

In the last two sections, the methods that set the location and format began with `set_major`. In matplotlib there are two types of ticks, major and minor. By default, only the major ticks are shown, and these were the ticks that we were working with above. To show the minor ticks for both the x and y axis, call the `minorticks_on` axes method.

```
[16]: ax.minorticks_on()
fig
```



The minor ticks are shorter and have no label. matplotlib automatically places them at reasonable intervals. They have their own locator and formatters, which we access now. The `NullFormatter` simply returns an empty string for each label.

```
[17]: ax.xaxis.get_minor_locator()
```

```
[17]: <matplotlib.ticker.AutoMinorLocator at 0x125a9f190>
```

```
[18]: ax.xaxis.get_minor_formatter()
```

```
[18]: <matplotlib.ticker.NullFormatter at 0x11a590410>
```

The same locator and formatter classes in the `ticker` module can be used to change the location and appearance of the minor ticks. Below, the locations for both the x and y minor ticks are changed to longer intervals. The y-axis minor tick labels are formatted using the `FuncFormatter` which allows you to write a custom function that is passed two arguments, the tick value and the integer position. The labels are reduced in size with the `tick_params` axes method. It's unlikely that your graph will need to show the actual labels for the minor ticks, but is done below as an example.

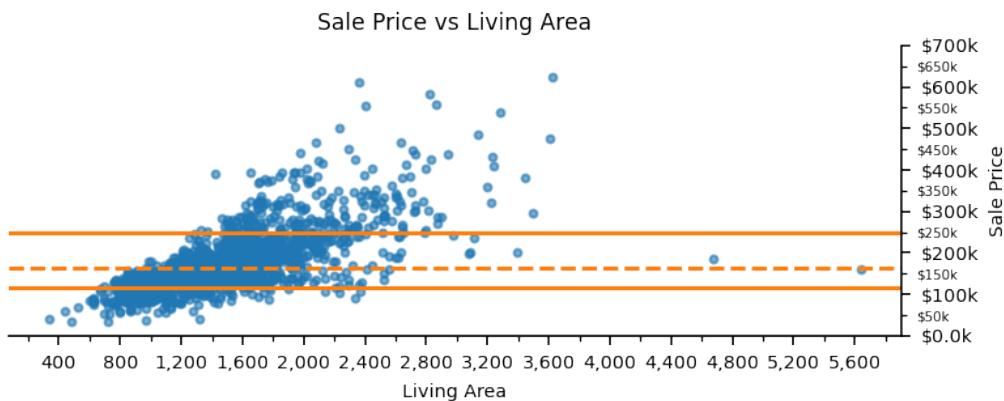
```
[19]: ax.xaxis.set_minor_locator(ticker.MultipleLocator(200))
ax.yaxis.set_minor_locator(ticker.MultipleLocator(50_000))
ax.yaxis.set_minor_formatter(ticker.FuncFormatter(lambda x, pos: f'${x // 1000:.0f}k'))
ax.tick_params(axis='y', which='minor', labelsize=5)
fig
```



67.6 Horizontal and vertical lines that span the axes

We previously learned about the `hlines` and `vlines` methods that place horizontal and vertical lines onto your axes. These methods require that you provide a beginning and ending point. Consider the scenario where you want a line to always span the width or height of your axes regardless of change in limits. With the methods `axhline` and `axvline`, you provide a single x or y value and by default a line will be drawn across the entire width or height of the axes. Here, we calculate the 15th, 50th, and 85th quantiles of sale price and unpack each as a scalar value. We then plot three horizontal lines that span the entire width of the axes.

```
[20]: from matplotlib import cm
q15, q50, q85 = housing['SalePrice'].quantile([.15, .5, .85])
ax.axhline(q15, color=cm.tab10(1))
ax.axhline(q50, ls='--', color=cm.tab10(1))
ax.axhline(q85, color=cm.tab10(1))
fig
```



Here, we duplicate the procedure for the x-axis variable drawing vertical lines that span the entire height of the axes. We also change the x-limits to prove that the horizontal lines plotted above still span the entire axes.

```
[21]: q15, q50, q85 = housing['GrLivArea'].quantile([.15, .5, .85])
ax.axvline(q15, color=cm.tab10(2))
ax.axvline(q50, ls='--', color=cm.tab10(2))
ax.axvline(q85, color=cm.tab10(2))
ax.set_xlim(-500, 3_000)
fig
```



67.7 Plotting with dates

matplotlib has the ability to make plots when one of the columns has datetime values. In this section, we'll make several plots that use datetime values along the x-axis. These plots tend to take more care to get the tick marks in the right location with the proper labels. We'll begin by reading in the stocks dataset converting the `date` column to a datetime placing it in the index.

```
[22]: stocks = pd.read_csv('../data/stocks/stocks10.csv', index_col=['date'],
                           parse_dates=['date'])
```

The closing price of Microsoft is plotted as a line plot below.

```
[23]: fig, ax = plt.subplots(figsize=(5, 2))
ax.plot(stocks['MSFT'])
ax.set_title('MSFT closing stock price 1999 - 2019');
```



While this plot is straightforward to interpret, there are some important details about the x-axis that need to be discussed. Let's take a look at the limits of the x-axis.

```
[24]: ax.get_xlim()
```

```
[24]: (729686.8, 737721.2)
```

matplotlib has chosen to convert the datetimes to floats by treating each year as 365 (or 366) units. The integer 1 corresponds to January 1, 1. If you multiply the beginning and end years on the plot by the average number of days per year, 365.25, you'll get values very close to those same floats.

```
[25]: 1999 * 365.25, 2021 * 365.25
```

[25]: (730134.75, 738170.25)

Converting floats to datetimes and vice-versa

matplotlib provides tools in its `dates` module to help convert floats to datetimes and vice-versa. The `num2date` function converts numbers to dates. Let's use it to verify that the number 1 is the date January 1, 1.

```
[26]: from matplotlib import dates  
dates.num2date(1)
```

[26]: datetime.datetime(1, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)

The returned value is a datetime object from the datetime standard library with units for year, month, day, hour, minute, second, and microseconds. Seconds and microseconds are not shown above, but are part of this object. Let's use this function (which also accepts a sequence) to determine the exact minimum and maximum dates of our x-axis.

```
[27]: dates.num2date(ax.get_xlim())
```

[27]: [datetime.datetime(1998, 10, 24, 19, 12, tzinfo=datetime.timezone.utc),
 datetime.datetime(2020, 10, 23, 4, 48, tzinfo=datetime.timezone.utc)]

The x-axis begins at October 10, 1998 at 7:12 pm and ends at October 23, 2020 at 4:48 am. There also exists the `date2num` function to do the opposite and take a datetime and convert it to a float. While it's perfectly valid to use datetimes from the datetime module, you can use numpy or pandas datetime objects as well. Below, we convert a pandas datetime object to a float.

```
[28]: dates.date2num(pd.Timestamp('2005-08-22 04:33:58'))
```

[28]: 732180.1902546296

There's also a `datestr2num` function to convert strings that are formatted like dates to floats. This saves the step of using an actual datetime object first.

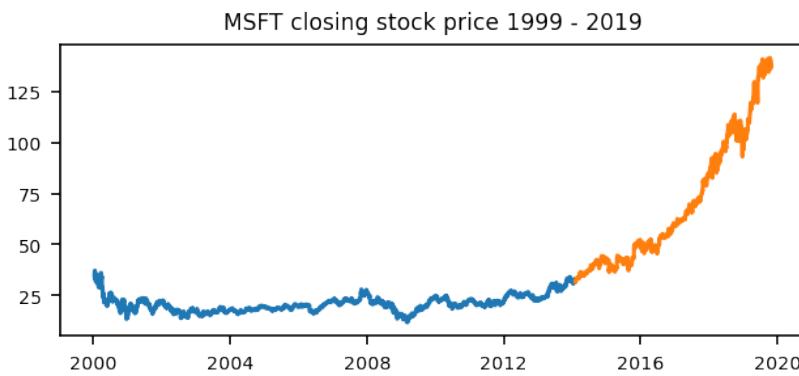
```
[29]: dates.datestr2num('2005-08-22 04:33:58')
```

[29]: 732180.1902546296

Stock price under CEOs Ballmer and Nadella

Microsoft has had two CEO's for almost the entirety of the twenty years of stock price data. Steve Ballmer took over from Bill Gates on January 13, 2000 before Satya Nadella took charge on February 4, 2014. Let's plot the stock price under each CEO as a different line.

```
[30]: fig, ax = plt.subplots(figsize=(5, 2))  
ax.plot(stocks.loc['2000-01-13':'2014-02-04', 'MSFT'])  
ax.plot(stocks.loc['2014-02-04':, 'MSFT'])  
ax.set_title('MSFT closing stock price 1999 - 2019');
```



Date locators and formatters

The above plot has major tick marks every four years. It might be nice to add minor ticks for every year. We could use the `minorticks_on` method to turn them on, but this turns them on for the y-axis as well. Instead, let's take a look at the minor tick locator.

```
[31]: ax.xaxis.get_minor_locator()
```

```
[31]: <matplotlib.ticker.NullLocator at 0x119976d10>
```

As the name implies, the `NullLocator` makes it so there are no ticks. There are special locator classes for datetimes in the `dates` module based on the unit of date measurement (year, month, day, hour, etc...). We'll use the `YearLocator` to make the minor tick labels visible. Grid lines for both minor and major ticks are turned on as well.

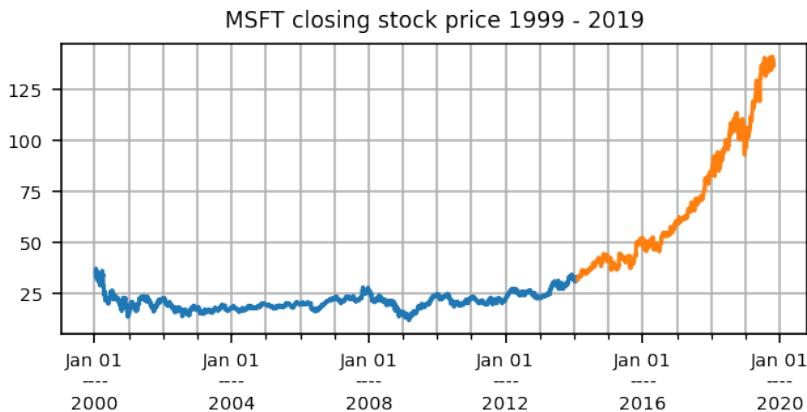
```
[32]: ax.xaxis.set_minor_locator(dates.YearLocator(1))
ax.grid(True, which='both')
fig
```



Datetimes also have their own set of formatter classes in the `dates` module. The most common way to format dates is by using a **formatting directive**, a single character code preceded by a percentage sign. Each directive corresponds with a specific string output. For instance '`%b`' represents the three character month name, and '`%m`' is the zero-padded month number (01, 02, etc...). [Consult the official Python documentation](#) for a list of all of the directives.

Create a string with any number of directives and use it to instantiate the `DateFormatter` class. Any other characters can appear in the string and will be interpreted literally. Below, we use three different string directives and add a couple of new lines.

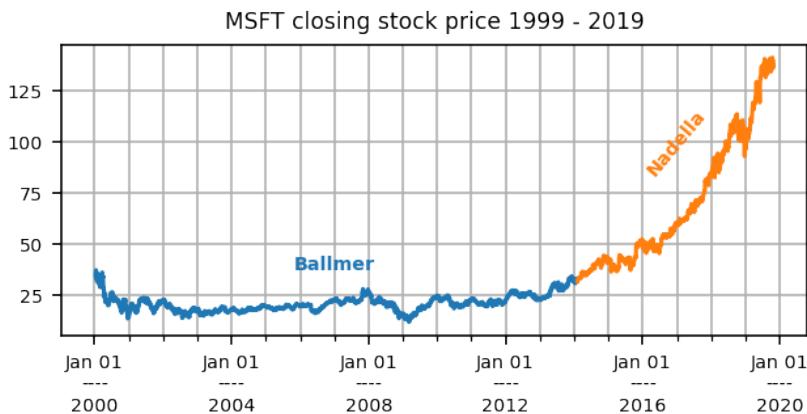
```
[33]: ax.xaxis.set_major_formatter(dates.DateFormatter('%b %d\n----\n%Y'))
fig
```



Adding objects to date plots

Adding new objects to the plot after the x-axis has turned into a date takes some care. You'll need to either use datetimes for the x-values or convert them to floats with `date2num` or `datestr2num`. Below, we add the names of each CEO at their midpoint of their time by using a pandas datetime object as the x-value. They are colored the same as their line color.

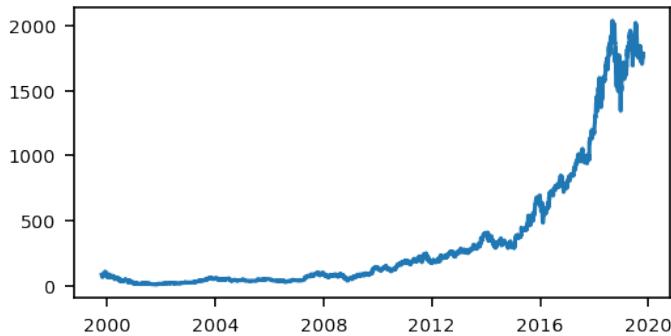
```
[34]: ballmer_middle = pd.Timestamp('2007-01-01')
nadella_middle = pd.Timestamp('2017-01-01')
colors = [line.get_color() for line in ax.lines]
ax.text(x=ballmer_middle, y=40, s='Ballmer', color=colors[0],
        ha='center', va='center', fontweight='bold')
ax.text(x=nadella_middle, y=100, s='Nadella', color=colors[1],
        rotation=50, ha='center', va='center', fontweight='bold')
fig
```



67.8 Using a different scale for the axis

By default, the x and y axis use a linear scale, meaning equal-sized sections represent the same number of units. A distance of one horizontal inch on the x or y axis represents the same number of units regardless of its position. In this section, we'll cover how to use different scales other than linear. We begin by making a line plot of Amazon's closing price for the last 20 years. The range of values vary greatly making it a candidate for using a different scale.

```
[35]: stocks = pd.read_csv('../data/stocks/stocks10.csv', index_col=['date'],
                         parse_dates=['date'])
fig, ax = plt.subplots()
ax.plot(stocks['AMZN']);
```



Let's verify that the current scale for both the x and y axis is linear with the `get_xscale` and `get_yscale` methods.

```
[36]: ax.get_xscale()
```

```
[36]: 'linear'
```

```
[37]: ax.get_yscale()
```

```
[37]: 'linear'
```

Logarithmic scale

The maximum closing price is more than 100 times the minimum, making it very difficult to discern stock price movements in the first two-thirds of the graph. The same one-day percentage move for prices on the higher end could appear around 100 times as large as the lowest. By using a logarithmic scale all one-day percentage movements represent the the same vertical distance on the graph regardless of the underlying price.

For example, let's compare the same percentage increase for stock prices of 50 and 1,500. For a 20% upward movement, the change in prices of 10 and 300 are very different when using a linear scale. A 20% movement in the above plot whenever Amazon's price was 50 would be indiscernible. This is why the line looks so flat for the lower prices. With a logarithmic scale, a 20% increase (and all percentage movements) represents the same vertical distance due to the following property of logs.

$$\log 1.2x - \log x = \log \frac{1.2x}{x} = \log 1.2$$

Below, we verify that the 20% movement results in the same increase when taking the logarithm with base 10 of both prices.

```
[38]: np.log10(60) - np.log10(50)
```

```
[38]: 0.07918124604762489
```

```
[39]: np.log10(1800) - np.log10(1500)
```

[39]: 0.07918124604762466

[40]: `np.log10(1.2)`

[40]: 0.07918124604762482

Manually computing the logarithmic scale

To help understand exactly how an axis gets transformed from a linear to a logarithmic scale, we will go through the procedure manually with six specific values. Two separate axes are created (see section below on how to create multiple axes within a figure) with the actual values plotted on one and the log of the values on the other.

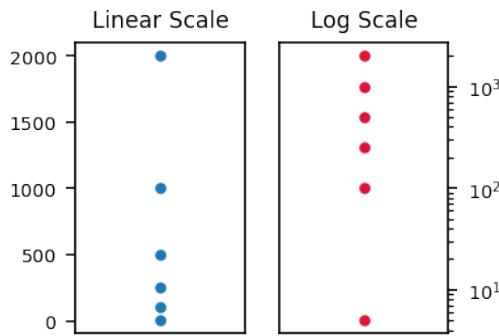
```
[41]: x = np.ones(6)
y = np.array([5, 100, 250, 500, 1000, 2000])
y_log = np.log10(y)
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(2.5, 2))
ax1.scatter(x, y, s=10)
ax2.scatter(x, y_log, s=10, c='crimson')
ax1.set_xticks([])
ax2.set_xticks([])
ax2.yaxis.set_ticks_position('right')
ax1.set_title('Actual Values')
ax2.set_title('Logged Values');
```



Using matplotlib to set the scale

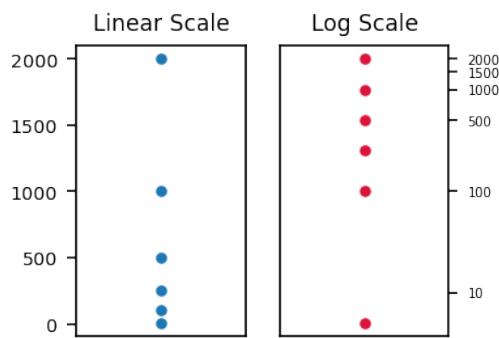
All we did above was apply a transformation to the data. The y-axis scale is still linear. To change the scale of the y-axis, and keep the data the same, pass the string '`'log'`' to the `set_yscale` method. We plot the actual values on each axes. Notice that the points are in the same exact location as the plot above. The only thing that changed are the tick marks and labels.

```
[42]: fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(2.5, 2))
ax1.scatter(x, y, s=10)
ax2.scatter(x, y, s=10, c='crimson')
ax2.set_yscale('log')
ax1.set_xticks([])
ax2.set_xticks([])
ax2.yaxis.set_ticks_position('right')
ax1.set_title('Linear Scale')
ax2.set_title('Log Scale');
```



Each major tick is formatted using scientific notation. Let's set the tick marks on our log scale axes to those of the linear scale to compare their locations.

```
[43]: ax2.set_yticks([10, 100, 500, 1000, 1500, 2000])
ax2.yaxis.set_major_formatter(ticker.ScalarFormatter())
ax2.yaxis.set_minor_locator(ticker.NullLocator())
ax2.tick_params(axis='y', labelsize='x-small')
fig
```



Let's go back and plot our Amazon data using a log scale. This uncovers a huge downswing the stock suffered in the early 2000's where it lost more than 90% of its value. This massive dip was essentially impossible to see with the original scale.

```
[44]: fig, ax = plt.subplots()
ax.plot(stocks['AMZN'])
ax.set_yscale('log')
ax.set_title('Amazon Closing Price - Log Scale');
```



67.9 Adding images

matplotlib has the ability to display images on its axes. Natively, matplotlib supports the reading of PNG images, but can read in many other formats using the [Pillow library](#). Install it by running `conda install pillow` from the command line. All images used in this section will be PNGs.

Reading in images as three-dimensional numpy arrays

Before you can add an image to your axes, you must read it in from its file location as a three-dimensional numpy array using the `imread` (image read) function from the `image` module. It reads in every pixel as an RGBA float (four values all between 0 and 1). It returns a three-dimensional numpy array with the shape $M \times N \times 4$ where M is the number of **vertical** pixels and N the number of **horizontal** pixels along with the 4 RGBA floats for each pixel. Let's read in an image of my son Niko swinging on a tire and output the shape of the returned array.

```
[45]: from matplotlib import image  
img_array = image.imread('images/niko.png')  
img_array.shape
```

```
[45]: (1453, 2829, 4)
```

This image has 1,453 vertical pixels and 2,829 horizontal pixels, meaning its width is about twice its height. This is the reverse order of how we normally read a pair of values that correspond to two dimensions in our coordinate plane. A pair of points usually has horizontal units first then vertical units. The top-left-hand corner pixel resides at the location $(0, 0)$ in our numpy array. Let's select it to retrieve its RGBA floats.

```
[46]: img_array[0, 0]
```

```
[46]: array([0.26666668, 0.34117648, 0.3372549 , 1.        ], dtype=float32)
```

Let's get the RGBA float for the bottom left-hand corner pixel.

```
[47]: img_array[1452, 0]
```

```
[47]: array([0.24705882, 0.34509805, 0.4627451 , 1.        ], dtype=float32)
```

All of these RGBA pixels can be plotted simultaneously with the `imshow` (image show) axes method. Let's add the image to a newly created axes.

```
[48]: fig, ax = plt.subplots()  
ax.imshow(img_array);
```



Several things take place whenever `imshow` is called. The x and y limits change to the exact dimensions of the image. The y-axis gets inverted so that the upper-left-hand corner is the point (0, 0). The aspect is automatically set to ‘equal’ (we did this manually when creating circle patches) from its default ‘auto’.

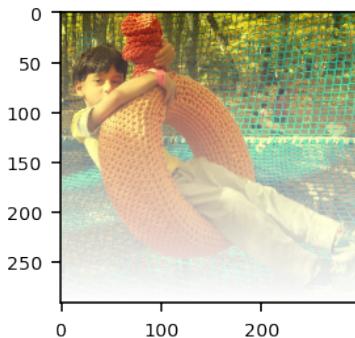
The original image has 2,829 horizontal pixels, but our figure has a DPI of 147 and with a width of 4 figure inches, this equates to just 588 horizontal pixels. matplotlib scales the image down so that it fits within the dimensions of the axes within the figure.

Changing values of the image array

Because images are read in as a numpy array of floats, you can easily manipulate the pixels using array operations. Here, we alter the image by doing the following:

- Select every fifth vertical pixel
- Select every fifth horizontal pixel beginning with the 2,000th through the 500th, flipping the image
- Divide the intensity of the blue values in half
- Transform the alpha values so that they begin at 1 on the top row and linearly go to 0 by the bottom.

```
[49]: fig, ax = plt.subplots()
a = img_array.copy()
a = a[::-5, 2000:500:-5, :]
a[:, :, 2] = a[:, :, 2] / 2
a[:, :, 3] = np.linspace(1, 0, len(a)).reshape(-1, 1)
ax.imshow(a);
```



Two-dimensional image arrays

The `imshow` method can take a two-dimensional numpy array of floats as well. When doing so, it uses the color map provided by the `cmap` parameter to convert each float to an RGBA value. Let’s select just the green values from our original image array and verify it is two-dimensional.

```
[50]: img_array_2d = img_array[:, :, 1]
img_array_2d.shape
```

```
[50]: (1453, 2829)
```

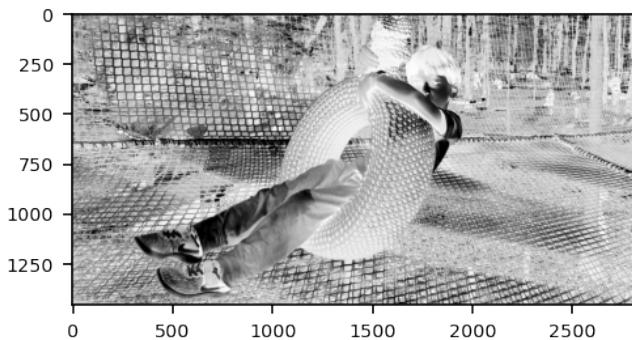
The first three rows and five columns (15 pixels) are selected and show below.

```
[51]: img_array_2d[:3, :5].round(2)
```

```
[51]: array([[0.34, 0.32, 0.37, 0.36, 0.36],
           [0.34, 0.32, 0.38, 0.37, 0.37],
           [0.35, 0.33, 0.33, 0.34, 0.36]], dtype=float32)
```

If not provided, the colormap chosen will be viridis. Let's choose 'Greys' to make this a grayscale image.

```
[52]: fig, ax = plt.subplots()
ax.imshow(img_array_2d, cmap='Greys');
```

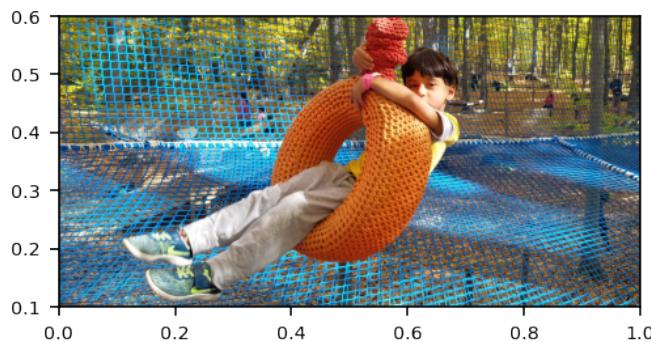


Adding images to a specific position in the axes

By default, the `imshow` method changes the x and y limits to those of the image dimensions regardless of the other plotting objects that are already on the axes. To place an image in a specific rectangle of coordinates, set the `extent` parameter of `imshow` to a four-item list of the left, right, bottom, and top coordinates. If you do not set the limits of your axis when doing so, then matplotlib will use the limits of the last image it receives.

Below, the image of Niko is read and assigned to a new variable name. It is then plotted within the bounds of the rectangle defined from `extent`. Notice how matplotlib automatically sets the limits of the x and y axis to those of the rectangle limits.

```
[53]: array_niko = image.imread('images/niko.png')
fig, ax = plt.subplots()
ax.imshow(array_niko, extent=[0, 1, .1, .6]);
```



An image of my daughter Penelope climbing a rock is read in and plotted within its own rectangle in a different non-overlapping location as the image above. Again, matplotlib adjusts the limits to be those of the last image added to the plot.

```
[54]: array_penelope = image.imread('images/penelope.png')
ax.imshow(array_penelope, extent=[1, 2, .3, .8])
fig
```



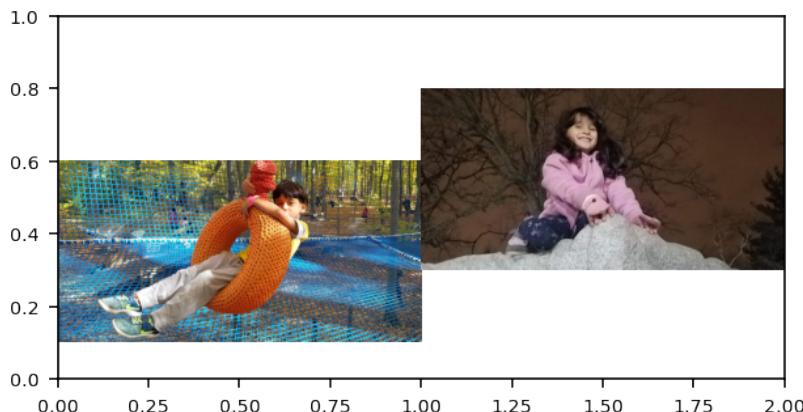
Our axes still has two images on it, but we can only see one. We can verify this by accessing the `images` attribute.

```
[55]: ax.images
```

```
[55]: [<matplotlib.image.AxesImage at 0x11b64e850>,
<matplotlib.image.AxesImage at 0x11b257290>]
```

Setting the limits of the axes prevents matplotlib from using just the limits defined by `extent`.

```
[56]: fig, ax = plt.subplots(figsize=(5, 2.5))
ax.set_xlim(0, 2)
ax.set_ylim(0, 1)
img_niko = ax.imshow(array_niko, extent=[0, 1, .1, .6])
img_penelope = ax.imshow(array_penelope, extent=[1, 2, .3, .8]);
```

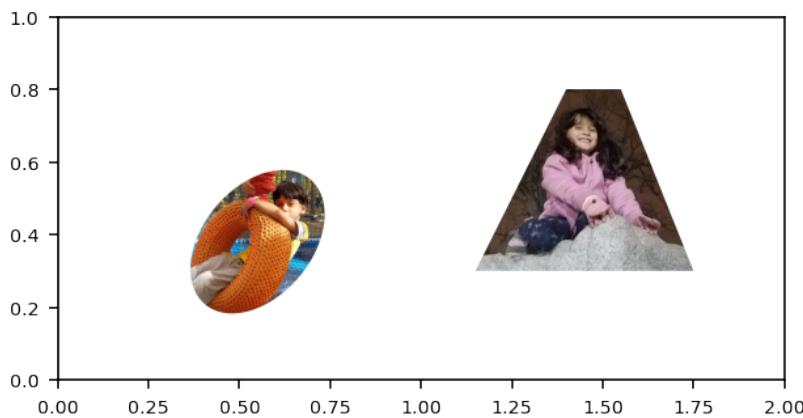


Clipping the image

Images can be clipped by matplotlib patches. To do so, create the patch with `alpha` set to 0 (or `fill` set to `False`) and add it to the axes. Then pass the patch to the `set_clip_path` image method. Above, the images were assigned to variable names.

```
[57]: from matplotlib import patches
niko_clip = patches.Ellipse((.55, .38), width=.3, height=.45, angle=-40, alpha=0)
penelope_clip = patches.Polygon([(1.4, .8), [1.15, .3], [1.75, .3], [1.55, .8]], alpha=0)
ax.add_patch(niko_clip)
ax.add_patch(penelope_clip)
```

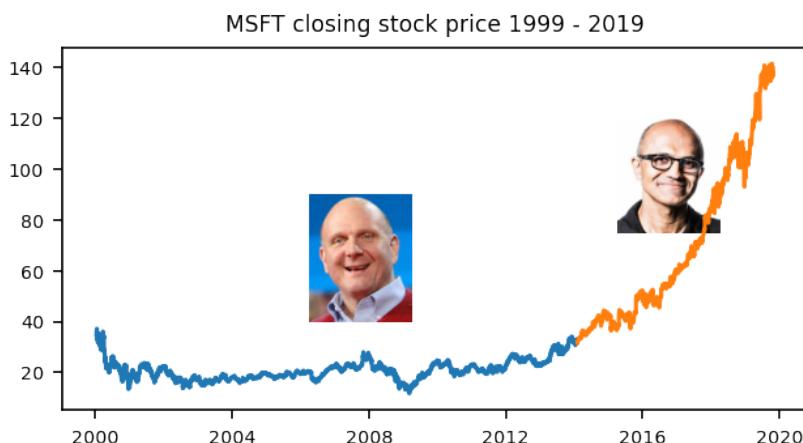
```
img_niko.set_clip_path(niko_clip)
img_penelope.set_clip_path(penelope_clip)
fig
```



Adding Microsoft's CEO images

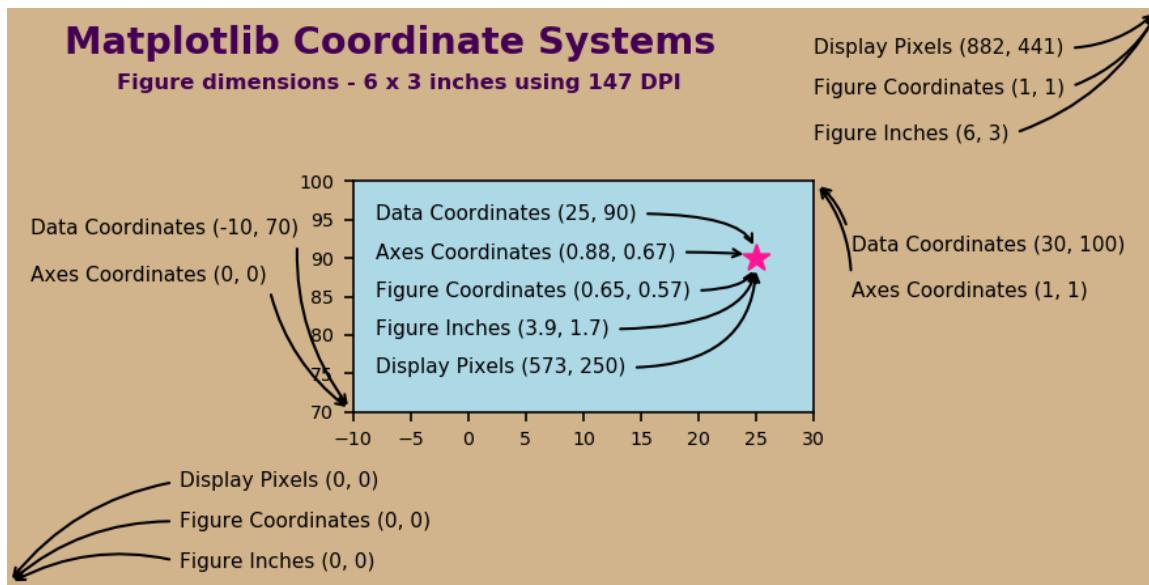
In this example, we'll add images of each CEO to the Microsoft closing price graph from above. Instead of setting the x and y limits, we'll set the `aspect` parameter from `imshow` to 'auto' instead of the default 'equal'. To get the correct x-coordinates for the rectangle defined in `extent`, we use the `datestr2num` function to convert strings directly to floats, saving the step of creating a datetime object.

```
[58]: array_ballmer = image.imread('images/ballmer.png')
array_nadella = image.imread('images/nadella.png')
fig, ax = plt.subplots(figsize=(5, 2.5))
ax.plot(stocks.loc['2000-01-13':'2014-02-04', 'MSFT'])
ax.plot(stocks.loc['2014-02-04':, 'MSFT'])
ax.set_title('MSFT closing stock price 1999 - 2019')
b_left, b_right = dates.datestr2num('2006'), dates.datestr2num('2009')
n_left, n_right = dates.datestr2num('2015'), dates.datestr2num('2018')
ax.imshow(array_ballmer, extent=[b_left, b_right, 40, 90], aspect='auto')
ax.imshow(array_nadella, extent=[n_left, n_right, 75, 120], aspect='auto');
```



67.10 Coordinate systems

In each matplotlib figure, a single point may be referenced using one of several different **coordinate systems**. matplotlib employs five different coordinate systems that are explained visually in the image below. Take a look at the point plotted as a star. Each of the coordinate systems references that point with a different pair of x and y coordinates.



Coordinate system definitions

Data - Each axes defines its own data coordinate system with the x and y limits (retrieved with the `get_xlim` and `get_ylim` methods). The bottom left-hand and top-right hand corners have coordinates of `(xmin, ymin)` and `(xmax, ymax)`. The data coordinate system is the only one we've used thus far and is the primary system used for placing objects on our axes.

Axes - The bottom left-hand and top-right hand corners of the axes always have coordinates of `(0, 0)` and `(1, 1)`. Points in the axes coordinate system are given as relative units to the width and height of the axes.

Figure - Similar to the axes coordinate system, the bottom left-hand and top right-hand corners of the figure always have coordinates `(0, 0)` and `(1, 1)`.

Figure-Inches - The bottom left-hand and top right-hand corners have coordinates `(0, 0)` and `(width, height)` where `width` and `height` are set during construction with the `figsize` parameter.

Display - The display coordinate system has units of pixels. Multiplying the figure inches by the DPI returns the upper right-hand corner value.

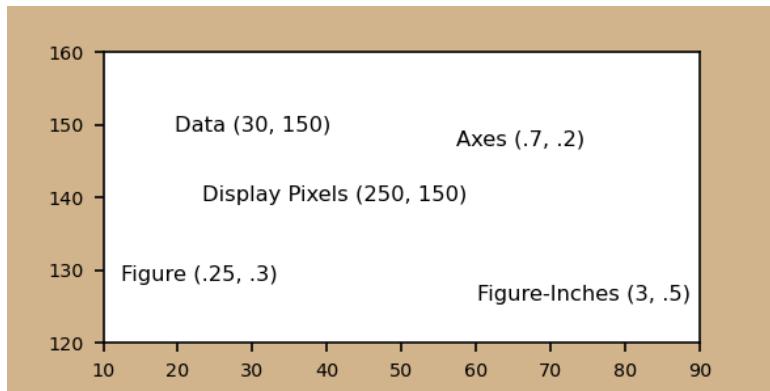
Plotting with each coordinate system

Nearly all axes methods use the data coordinate system by default. All numbers passed to these axes methods are treated as data coordinates. These methods can be set to use any other coordinate system by setting the `transform` parameter to a specific transformation object. You can access the transformation object from each coordinate system using the following attributes:

- **Data** - `ax.transData`
- **Axes** - `ax.transAxes`
- **Figure** - `fig.transFigure`
- **Figure-Inches** - `fig.dpi_scale_trans`
- **Display** - None

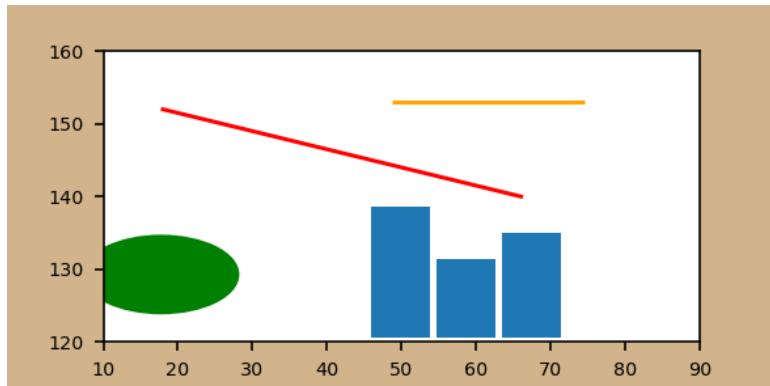
Let's create our default figure and axes and add text to different places using each of the coordinate systems by setting the `transform` parameter. As a reminder, our figure measures 4 x 2 figure-inches using 147 DPI making it 588 x 294 pixels. The notebook display 'bbox_inches' is set to `None` so that the exact dimensions of the figure are displayed.

```
[59]: %config InlineBackend.print_figure_kwarg = {'bbox_inches': None}
fig, ax = plt.subplots(facecolor='tan')
ax.set_xlim(10, 90)
ax.set_ylim(120, 160)
kws = {'ha': 'center', 'va': 'center', 'fontsize': 8}
ax.text(x=30, y=150, s='Data (30, 150)', transform=ax.transData, **kws)
ax.text(x=.7, y=.7, s='Axes (.7, .2)', transform=ax.transAxes, **kws)
ax.text(x=.25, y=.3, s='Figure (.25, .3)', transform=fig.transFigure, **kws)
ax.text(x=3, y=.5, s='Figure-Inches (3, .5)', transform=fig.dpi_scale_trans, **kws)
ax.text(x=250, y=150, s='Display Pixels (250, 150)', transform=None, **kws);
```



Although we set the `transform` parameter to `ax.transData`, it wasn't necessary as that is the default. Below, we call four different axes methods setting the coordinate system to something other than the default.

```
[60]: fig, ax = plt.subplots(facecolor='tan')
ax.set_xlim(10, 90)
ax.set_ylim(120, 160)
ax.plot([.1, .7], [.8, .5], transform=ax.transAxes, color='red')
ax.add_patch(patches.Circle((.2, .3), radius=.1, transform=fig.transFigure, color='green'))
ax.hlines(y=1.5, xmin=2, xmax=3, transform=fig.dpi_scale_trans, color='orange')
ax.bar(x=[300, 350, 400], height=[100, 60, 80], width=45, bottom=40, transform=None);
```



Transforming to and from different coordinate systems

Each of the transformation objects offers a way to transform points in one coordinate system to display pixels with its `transform` method. Using the same figure and axes above, let's transform the point (30, 150) to display pixels.

```
[61]: ax.transData.transform([30, 150])
```

```
[61]: array([187.425, 203.2275])
```

Knowing this, we could have added the text 'Data (30, 150)' from a previous plot with the following:

```
ax.text(x=187, y=203, s='Data (30, 150)', transform=None, **kws)
```

Let's continue using the various transformation objects to convert to display pixels. Here we transform the right endpoint of the first line plotted above from axes coordinates to pixels.

```
[62]: ax.transAxes.transform(.8, .5)
```

```
[62]: array([438.06, 147.735])
```

The center of the circle, originally in figure coordinates, is transformed to pixels.

```
[63]: fig.transFigure.transform(.2, .3)
```

```
[63]: array([117.6, 88.2])
```

The left endpoint of the horizontal line, originally in figure-inches, is transformed to pixels.

```
[64]: fig.dpi_scale_trans.transform([2, 1.5])
```

```
[64]: array([294., 220.5])
```

A different transformation object, `ax.transLimits`, transforms points from the data to axes coordinate system (and not to the display coordinate system).

```
[65]: ax.transLimits.transform([30, 150])
```

```
[65]: array([0.25, 0.75])
```

Inverting a transformation

Each transformation object has an `inverted` method which creates a new transformation object capable of inverting the transformation with its own `transform` method. Below, we transform the display pixels back to their data coordinates.

```
[66]: ax.transData.inverted().transform([187.425, 203.2275])
```

```
[66]: array([30., 150.])
```

Let's verify that the top right-hand corner display coordinate has the correct figure-inches and figure coordinates.

```
[67]: fig.dpi_scale_trans.inverted().transform([588, 294])
```

```
[67]: array([4., 2.])
```

```
[68]: fig.transFigure.inverted().transform([588, 294])
```

```
[68]: array([1., 1.])
```

We can now go from one coordinate system to another by first transforming to pixels and then using an inverted transformer to go to the other coordinate system. Here, we transform the figure coordinates of $(.2, .3)$ to data coordinates.

```
[69]: pixels = fig.transFigure.transform([.2, .3])
ax.transData.inverted().transform(pixels).round(1)
```

```
[69]: array([ 17.7, 129.3])
```

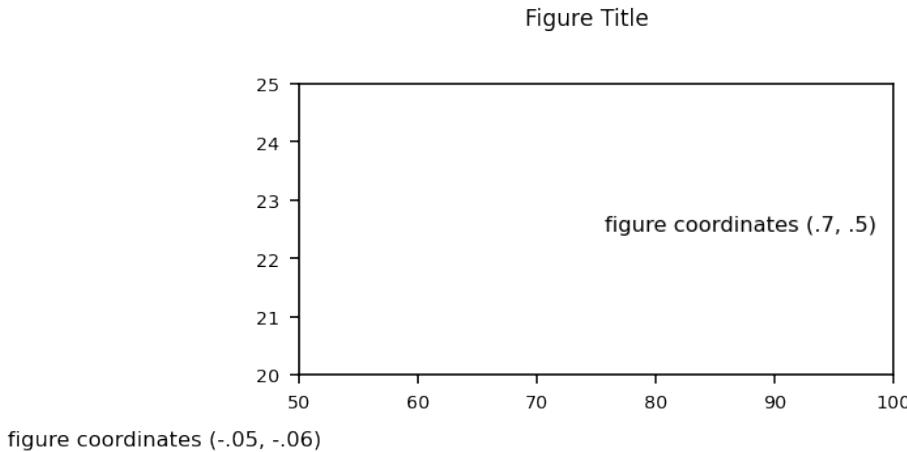
67.11 Figure methods

Nearly all of the methods called thus far have come from the axes object, which will be the case the vast majority of the time. In this section, we'll call several figure methods that change the figure in some way. Figures have far fewer methods than axes and are not capable of directly plotting data.

You can add a title to the entire figure with the `suptitle` method (think of “super” title) and text with the `text` method. Both of these methods accept x and y values given as figure coordinates and not data coordinates as before. The figure is a different object than the axes and can contain multiple axes, so the concept of data coordinates for it do not make sense.

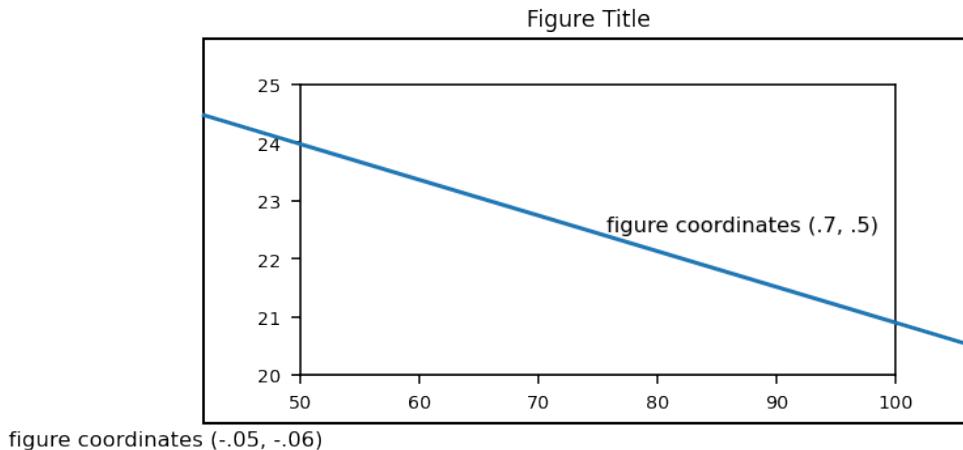
A title is added to the figure with a y-value figure coordinate of 1.02. While this value is outside of the figure, it is still visible in the display because we set ‘bbox_inches’ back to ‘tight’ which includes any plotting objects regardless of their location. Just because an object isn’t contained within the boundaries of the figure, doesn’t mean it isn’t able to be drawn. Two pieces of text are created, one within and the other outside the figure boundary.

```
[70]: %config InlineBackend.print_figure_kwarg = {'bbox_inches': 'tight'}
fig, ax = plt.subplots()
ax.set_xlim(50, 100)
ax.set_ylim(20, 25)
fig.suptitle(x=.5, y=1.02, t='Figure Title', va='bottom')
fig.text(x=.7, y=.5, s='figure coordinates (.7, .5)', ha='center', fontsize=8)
fig.text(x=-.05, y=-.06, s='figure coordinates (-.05, -.06)', ha='center', fontsize=8);
```



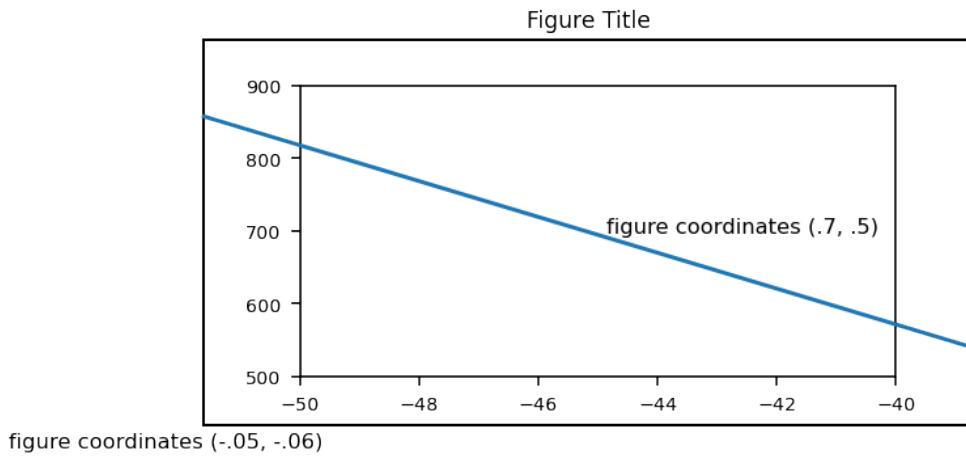
It is rare that you'll need to add plotting objects such as patches or lines to a figure, but it is possible. You'll have to import the module that creates the underlying object and then add it to the figure with the `add_artist` method. By default, the points used to create these objects will be in figure coordinates. We add a line across the entire length of the figure using the `Line2D` constructor from the `lines` module. A rectangle patch is added so that it outlines the border of the figure.

```
[71]: from matplotlib import lines
line = lines.Line2D(xdata=[0, 1], ydata=[.8, .2])
fig.add_artist(line)
rect = patches.Rectangle((0, 0), width=1, height=1, fill=False)
fig.add_artist(rect)
fig
```



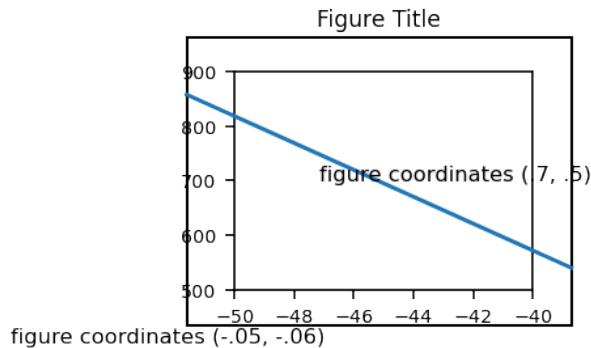
One reason to add objects to figures instead of axes is to keep them in the same place regardless of the x and y data limits. Let's change the limits of the axes dramatically to verify this has no effect on the figure objects.

```
[72]: ax.set_ylim(500, 900)
ax.set_xlim(-50, -40)
fig
```



Changing the size of the figure keeps the figure objects in the same relative position.

```
[73]: fig.set_size_inches(2, 1.5)
fig
```

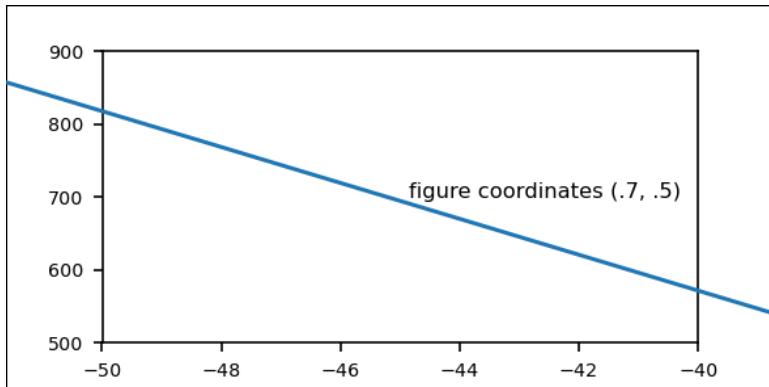


Saving the figure to a file

The figure can be permanently saved on disk with the `savefig` method. The first argument is the filename given as a string. The filename extension dictates the type of file created. Possible values are png, jpeg, svg, eps, pdf, and more. We save the figure in the images folder in this current directory as a png.

```
[74]: fig.set_size_inches(4, 2)
fig.savefig('images/simple_figure.png')
```

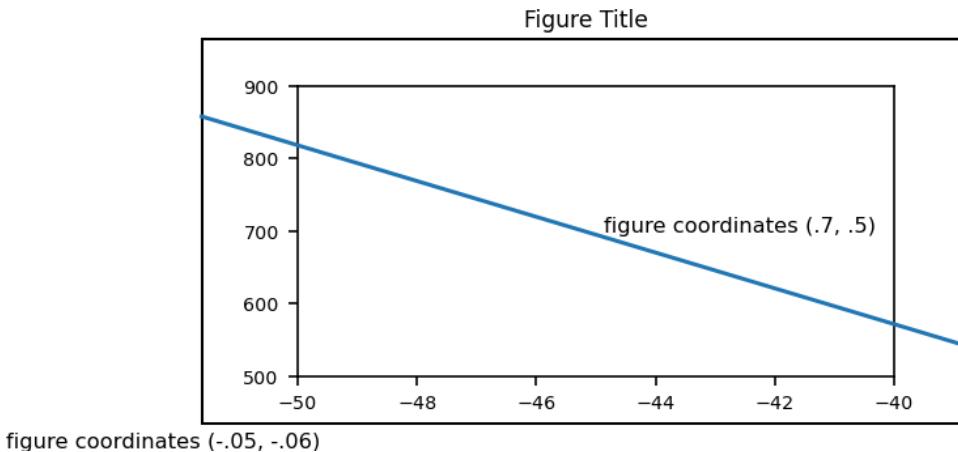
By default, the exact dimensions of the figure are saved. Any objects plotted outside the figure boundaries are cut off. The image we just saved is shown below. Notice that the figure title and one of the text objects did not get saved to the image.



By setting the `bbox_inches` parameter to '`'tight'`', matplotlib will include all of the items plotted regardless of their location. When choosing this option, there is still a small amount of padding around the figure (.1 inches by default). Use the `pad_inches` parameter to control it.

```
[75]: fig.savefig('images/simple_figure_tight.png', bbox_inches='tight', pad_inches=0)
```

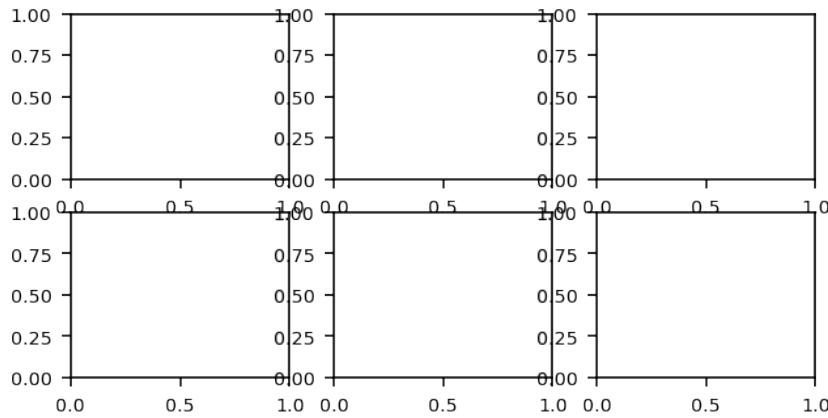
The updated image is shown below with both of the previously missing items. These `bbox_inches` from the `savefig` method are the same setting that Jupyter Notebook uses when displaying the images. The only difference is that the default settings for the notebook have `bbox_inches` set to '`'tight'`', while `savefig` uses '`'None'`'.



67.12 Creating a grid of axes

In all of our previous work, we created a single axes within a single figure. In this section, we will create multiple axes within a single figure. The simplest way to create a grid of axes within a single figure is to set the `nrows` and `ncols` parameters of the `subplots` function. By default, these values are each set to 1, so when we used this function previously, a 1×1 grid of subplots was created. Let's begin by creating a figure with two rows and three columns, making a total of six axes.

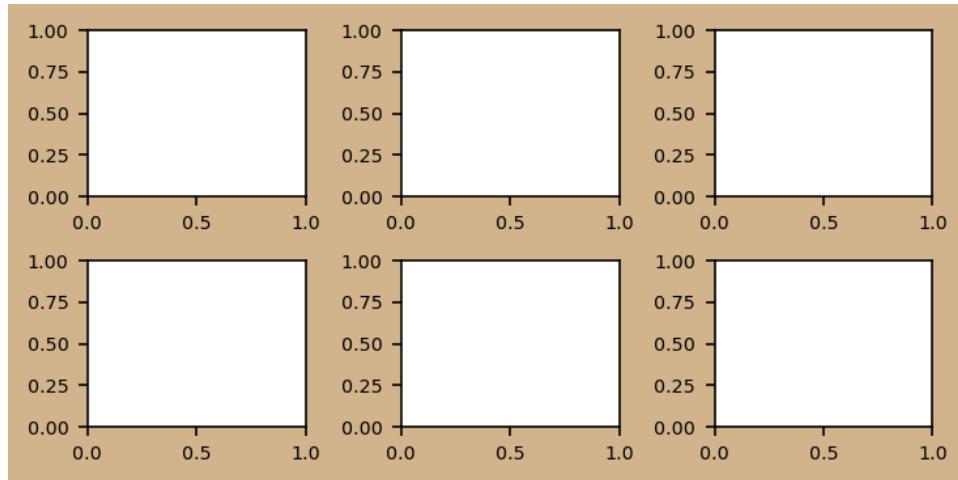
```
[76]: fig, ax_array = plt.subplots(nrows=2, ncols=3, figsize=(5, 2.5))
```



Clean up the layout and set the face color of the figure

The axes are packed quite closely together above with the x and y tick labels overlapping one another. matplotlib doesn't take the labels of each axes into account when displaying the plots. To force matplotlib to consider all of the labels and provide ample space to be given between each axes, we must set the `tight_layout` property to `True`. This can be done with a setter method, but can also be done when first constructing the figure.

```
[77]: fig, ax_array = plt.subplots(nrows=2, ncols=3, figsize=(5, 2.5),
                               tight_layout=True, facecolor='tan')
```



Multiple Axes returned as a numpy array

Whenever you create multiple axes like this with the `subplots` function, the second item returned will be a numpy array of axes objects. Previously, only one axes was created, and it was returned as the axes itself, not wrapped up in an array. Let's verify that this new second object is indeed a numpy array.

```
[78]: type(ax_array)
```

```
[78]: numpy.ndarray
```

The array has the same shape as our grid, two rows by three columns.

```
[79]: ax_array.shape
```

[79]: (2, 3)

If we output the array, we will see six different axes objects.

[80]: ax_array

```
[80]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1404e2090>,
   <matplotlib.axes._subplots.AxesSubplot object at 0x125ff1ed0>,
   <matplotlib.axes._subplots.AxesSubplot object at 0x1403a36d0>],
  [<matplotlib.axes._subplots.AxesSubplot object at 0x11b9abc10>,
   <matplotlib.axes._subplots.AxesSubplot object at 0x125e94c50>,
   <matplotlib.axes._subplots.AxesSubplot object at 0x125d1ea90>]],  
dtype=object)
```

It's much easier to work with the axes directly, so let's assign each one to a variable by selecting it using integer location.

[81]:

```
ax1 = ax_array[0, 0]
ax2 = ax_array[0, 1]
ax3 = ax_array[0, 2]
ax4 = ax_array[1, 0]
ax5 = ax_array[1, 1]
ax6 = ax_array[1, 2]
```

Instead of selecting each axes with its row and column location, use the `flatten` method to return a one-dimensional array of all six axes. By default, they are flattened one row at a time. We can unpack each axes as its own variable easier this way.

[82]:

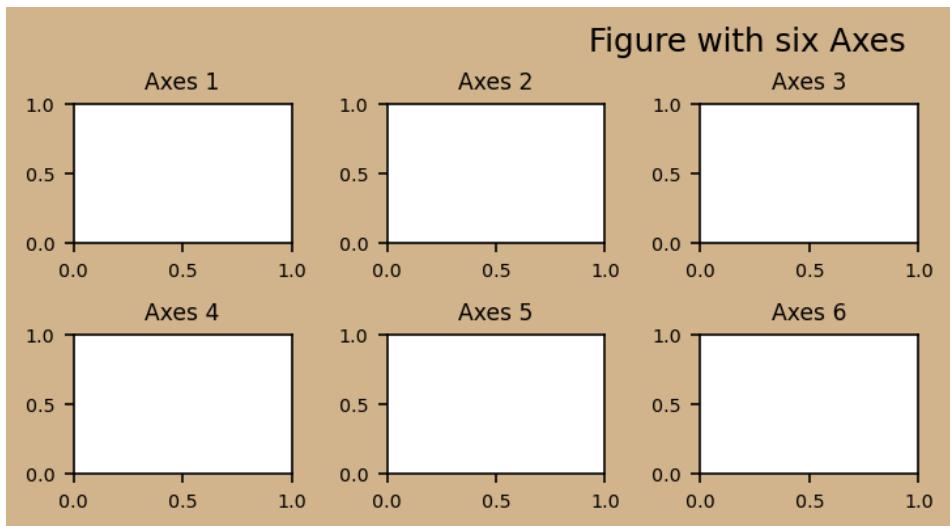
```
ax_flat = ax_array.flatten()
ax1, ax2, ax3, ax4, ax5, ax6 = ax_flat
```

Set the title of each axes to identify it

To help identify each axes, let's set the title of each one. We could use the variable name to reference each axes, but choose to iterate through the flattened array of axes instead. Notice that the height of each axes has automatically adjusted a bit to accommodate the titles due to the `tight_layout` being set to `True`.

[83]:

```
for i, ax in enumerate(ax_flat):
    ax.set_title(f'Axes {i + 1}')
fig.suptitle('Figure with six Axes', x=.95, y=1.05, fontsize=12, ha='right')
fig
```



67.13 Exercises

[]:

Part XII

Visualization with Pandas and Seaborn

Chapter 68

Plotting with pandas Series

Both pandas Series and DataFrames have a `plot` method capable of creating a variety of plots with the data they contain. pandas directly uses matplotlib for all of its plotting and does not have any plotting capabilities on its own. pandas simply calls matplotlib's plotting functions internally, supplying them the arguments for you. pandas provides only a small subset of the total available types of plots that matplotlib offers. pandas does not give you full control over the plots it creates. However, it does return the underlying matplotlib axes object, which you can assign to a variable, and then use to customize the plot however you wish.

Series plots

In this chapter, we cover plotting with the simpler pandas Series. All plotting runs through the `plot` method with the `kind` parameter controlling the type of plot. Set the `kind` parameter equal to one of the following strings:

- `line` - line plot (default)
- `bar` - vertical bar plot
- `barch` - horizontal bar plot
- `box` - box plot
- `hist` - histogram
- `kde` - kernel density estimation plot
- `pie` - pie plot
- `area` - area plot

For all of these plots, the Series `index` is used as the x-values and the Series `values` as the y-values. We begin by reading in the stocks dataset and selecting Amazon's closing price as a Series.

```
[1]: import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('..../mdap.mplstyle')
stocks = pd.read_csv('..../data/stocks/stocks10.csv', index_col='date',
                     parse_dates=['date'])
amzn = stocks['AMZN']
amzn.head(3)
```

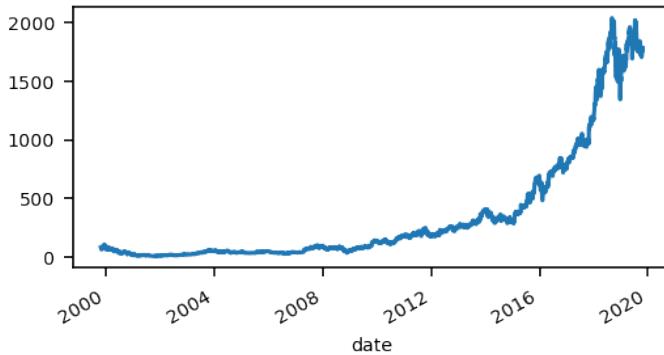
```
[1]: date
1999-10-25    82.75
1999-10-26    81.25
1999-10-27    75.94
```

```
Name: AMZN, dtype: float64
```

68.1 Line plots

We'll now create a line plot of these prices. The `kind` method is defaulted to '`line`', but since that's not an intuitive default, it is set explicitly below. pandas uses matplotlib to create the figure and axes for you and returns the axes which we assign to a variable.

```
[2]: ax = amzn.plot(kind='line')
```



Let's verify that we have a matplotlib axes.

```
[3]: type(ax)
```

```
[3]: matplotlib.axes._subplots.AxesSubplot
```

All axes have a `figure` attribute that you can access to retrieve the figure.

```
[4]: fig = ax.figure
```

The figure properties will be equal to those in the run configuration settings. Let's verify the size and DPI.

```
[5]: fig.get_size_inches()
```

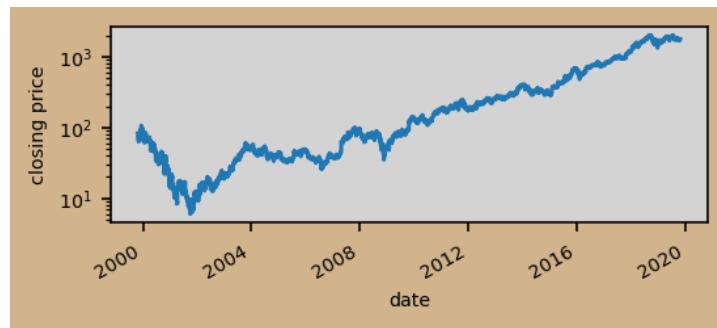
```
[5]: array([4., 2.])
```

```
[6]: fig.get_dpi()
```

```
[6]: 147.0
```

All axes and figure methods can now be called to update the plot.

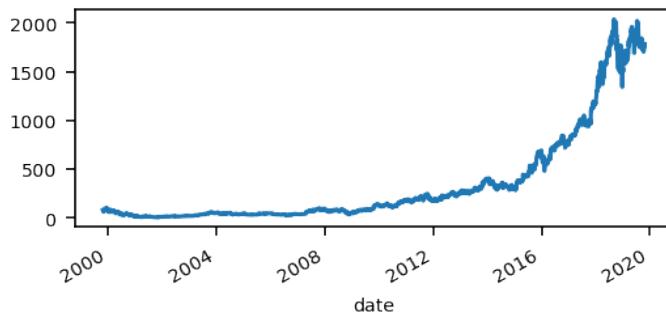
```
[7]: ax.set_ylabel('closing price')
ax.set_facecolor('lightgray')
ax.set_yscale('log')
fig.set_facecolor('tan')
fig.set_size_inches(4, 1.5)
fig
```



Recreating plot with matplotlib

Let's replicate this plot using matplotlib directly. We must manually set the x-axis label and move the tick labels. pandas uses the index name for the x-axis label.

```
[8]: fig, ax = plt.subplots(figsize=(4, 1.5))
ax.plot(amzn)
ax.set_xlabel('date')
for label in ax.get_xticklabels():
    label.set_rotation(30)
    label.set_ha('right')
```



Plotting parameters

There are a substantial number of parameters available to the `plot` method to customize its appearance. Setting these might make it so that you won't have to use matplotlib directly. Below, we set the figure size, use a log scale for the y-axis, add grid lines, a legend, and a title. The rotation and size of the ticks are controlled by `rot` and `fontsize`. Any other parameter not part of the `plot` documentation is passed to the underlying matplotlib plotting method, which is `ax.plot` in this instance. Here, `c`, `ls`, and `lw` change the property of the line itself.

```
[9]: amzn.plot(kind='line', figsize=(5, 2), logy=True, grid=True, legend=True,
            title='Amazon Closing Price', rot=15, fontsize=6,
            c='crimson', ls='--', lw=1);
```



68.2 Bar plots

Bar plots are created by setting the `kind` parameter to the string '`bar`' or '`bard`'. Each value in the Series will be plotted as a bar and labeled with its corresponding index value. Let's calculate the number of times Amazon's stock had a greater than 5% positive movement from the previous day's close for each year. We begin by finding the percentage change and test whether it meets our criteria.

```
[10]: amzn_up_down = amzn.pct_change(1) > .05
amzn_up_down.head(3)
```

```
[10]: date
1999-10-25    False
1999-10-26    False
1999-10-27    False
Name: AMZN, dtype: bool
```

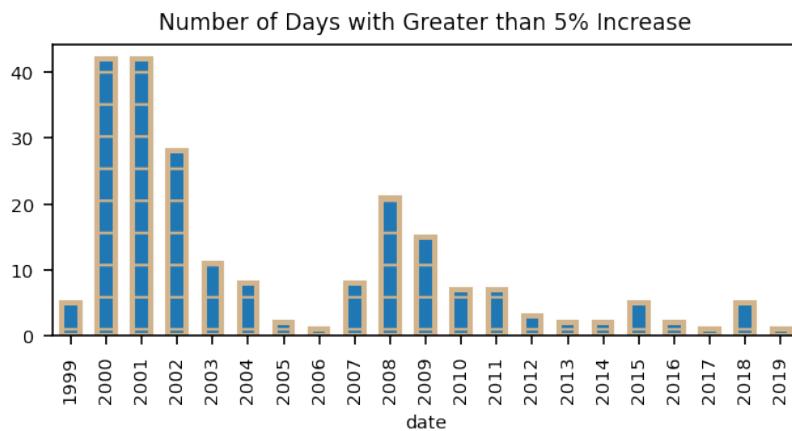
We then group by year and sum the `True` values to get the count by year.

```
[11]: num_big_up_days = amzn_up_down.resample('Y', kind='period').sum()
num_big_up_days.head(3)
```

```
[11]: date
1999      5.0
2000     42.0
2001     42.0
Freq: A-DEC, Name: AMZN, dtype: float64
```

This Series can now be made into a bar plot. In this case, the extra parameters (`lw`, `ec`, and `hatch`) are forwarded to the matplotlib axes `bar` method.

```
[12]: num_big_up_days.plot(kind='bar', figsize=(5, 2),
                         title='Number of Days with Greater than 5% Increase',
                         lw=2, ec='tan', hatch='-' );
```



68.3 Distribution plots

Box plots, histograms, and KDEs are the available distribution plots to pandas Series. Let's use the salary column from the City of Houston employee dataset for these examples.

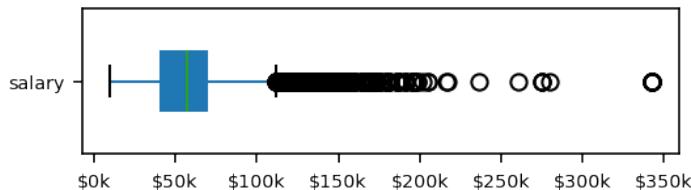
```
[13]: emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
sal = emp['salary']
sal.head(3)
```

```
[13]: 0    87545.38
1    82182.00
2    49275.00
Name: salary, dtype: float64
```

Box and whisker plots

A horizontal box plot is created by using the `vert` parameter, which is forwarded to the axes `boxplot` method along with `widths` and `patch_artist`. The ticks are also formatted to be in thousands of dollars.

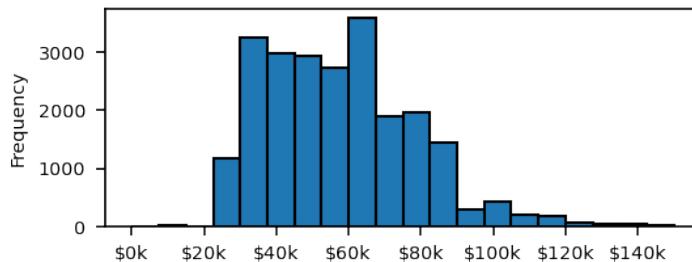
```
[14]: from matplotlib import ticker
ax = sal.plot(kind='box', figsize=(4, 1), vert=False, widths=.4, patch_artist=True)
conv_dollar = lambda x, pos: f'${x // 1000:.0f}k'
ax.xaxis.set_major_formatter(ticker.FuncFormatter(conv_dollar))
```



Histograms

A histogram of salaries is created below, with the tail end eliminated using the `range` parameter. Box plots are much better tools for analyzing extreme values, while histograms are much better at analyzing the “middle” of the data (such as those within the whiskers of the box plot). This is why the range of values is bounded below. In general, with box plots, you would not want to limit the range.

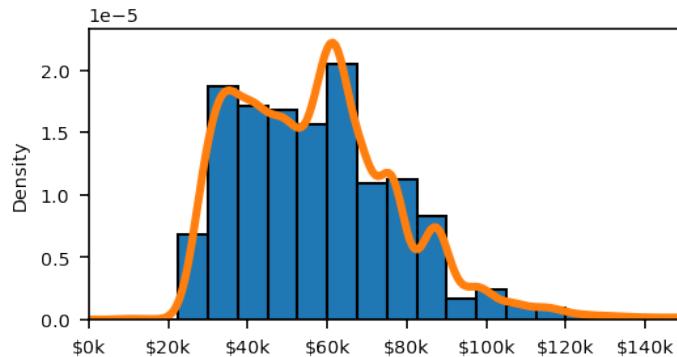
```
[15]: ax = sal.plot(figsize=(4, 1.5), kind='hist', bins=20, ec='black', range=(0, 150_000))
ax.xaxis.set_major_formatter(ticker.FuncFormatter(conv_dollar))
```



KDEs

A kernel density plot (KDE) is essentially a histogram with an infinite number of bins. A KDE plot is the only type of plot that does not have an equivalent matplotlib axes method. It estimates the probability density function of a distribution. Multiple calls to the `plot` method in a single cell will place each plot on the same axes. Below, we plot a histogram and KDE of the same salary data. You can see how closely the KDE curve matches the histogram. In order for the KDE and histogram to have the same units, we set `density` to True.

```
[16]: ax = sal.plot(kind='hist', bins=20, ec='black', range=(0, 150_000), density=True)
sal.plot(kind='kde', xlim=(0, 150_000), lw=3)
ax.xaxis.set_major_formatter(ticker.FuncFormatter(conv_dollar))
```



68.4 Pie Charts

Pie charts are circles with wedge areas for each value in the Series corresponding to its proportion of the whole. Let's count the number of employees whose salary fall into a particular range. The `cut` function is used to create the buckets with the `value_counts` method doing the counting.

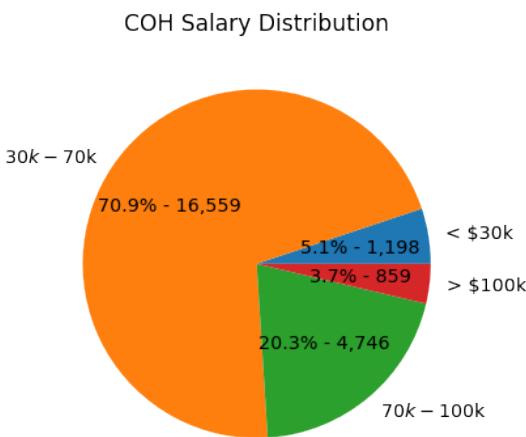
```
[17]: sals = pd.cut(sal, bins=[0, 30_000, 70_000, 100_000, sal.max() + 1],
                  labels=['< $30k', '$30k - $70k', '$70k - $100k', '> $100k'])
sal_ct = sals.value_counts(sort=False)
sal_ct
```

< \$30k	1198
\$30k - \$70k	16559
\$70k - \$100k	4746
> \$100k	859

```
Name: salary, dtype: int64
```

For pie charts, the index values are used as the labels for each wedge. The `autopct` parameter is forwarded to `ax.pie` and can be set to a function that is passed the percentage of each wedge. It returns a formatted percentage and the raw count. By default, pandas use the name of the Series as the y-axis label. A title looks more appropriate so it is removed by setting it to an empty string.

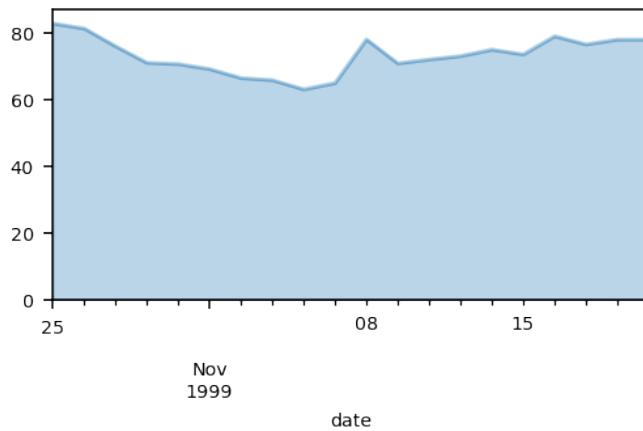
```
[18]: ax = sal_ct.plot(kind='pie', figsize=(3, 3), title='COH Salary Distribution',
                      autopct=lambda x: f'{x:.1f}% - {x / 100 * sal_ct.sum():,.0f}')
ax.set_ylabel('');
```



68.5 Area Plots

Area plots are like line plots, but fill the area between the x-axis and the line with a color. It's equivalent in matplotlib is `stackplot`. Area plots are much more useful when using DataFrames, as you'll see in the next chapter. Below, the first 20 trading days of Amazon are plotted as an area plot.

```
[19]: amzn.head(20).plot(kind='area', alpha=.3);
```

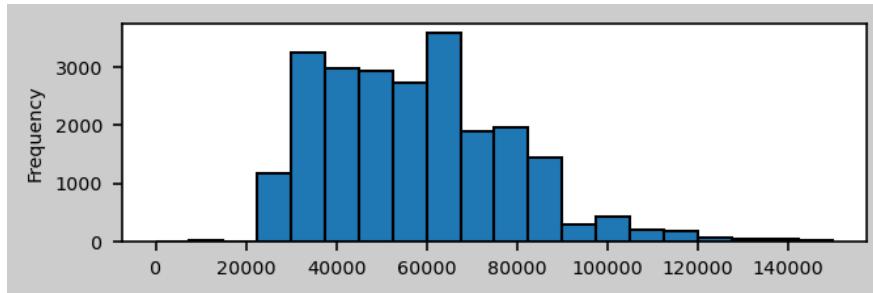


68.6 Adding a plot to a previously made axes

For all of the above plots, we let pandas create the figure and single axes. It's possible for us to create the figure and axes (possibly more than one) first and then tell pandas to use that particular axes with the `ax`

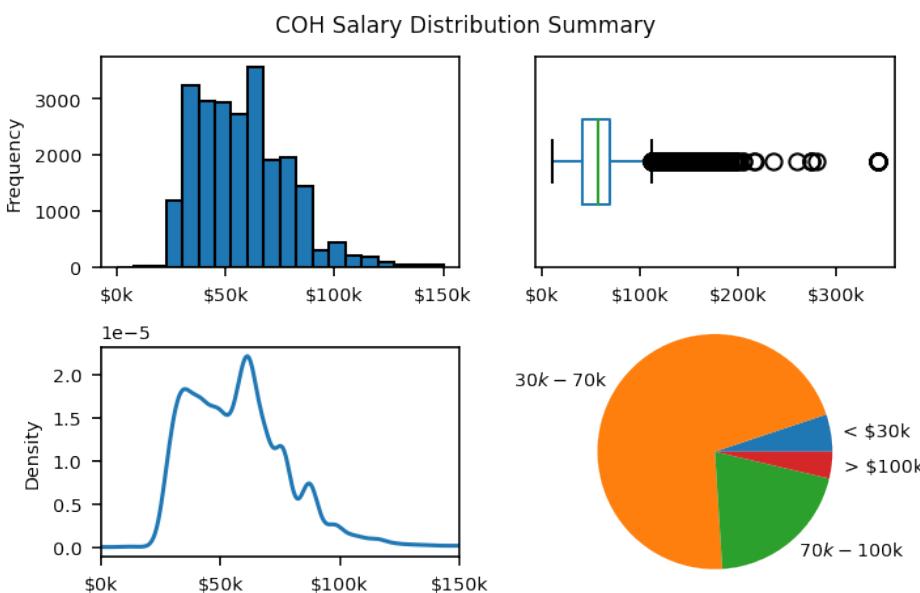
parameter. Below, we create our figure and axes first and then place a histogram of the salaries on that axes.

```
[20]: fig, ax = plt.subplots(figsize=(5, 1.5), facecolor='0.8')
sal.plot(kind='hist', bins=20, ec='black', range=(0, 150_000), ax=ax);
```



This becomes especially useful when placing plots into a figure with multiple axes. Here, four axes are created and unpacked into separate variables with the help of the numpy array `flatten` method. pandas is used to place each plot on each axes. The ticks are formatted and located appropriately.

```
[21]: fig, ax_array = plt.subplots(nrows=2, ncols=2, figsize=(5, 3), tight_layout=True)
ax1, ax2, ax3, ax4 = ax_array.flatten()
sal.plot(kind='hist', bins=20, ec='black', range=(0, 150_000), ax=ax1)
sal.plot(kind='box', vert=False, widths=.4, ax=ax2, yticks[])
sal.plot(kind='kde', xlim=(0, 150_000), ax=ax3)
sal_ct.plot(kind='pie', ax=ax4, radius=1.4)
ax1.xaxis.set_major_formatter(ticker.FuncFormatter(conv_dollar))
ax2.xaxis.set_major_formatter(ticker.FuncFormatter(conv_dollar))
ax3.xaxis.set_major_formatter(ticker.FuncFormatter(conv_dollar))
ax3.xaxis.set_major_locator(ticker.MultipleLocator(50_000))
ax4.set_ylabel('')
fig.suptitle('COH Salary Distribution Summary', y=1.04);
```



Chapter 69

Plotting with pandas DataFrames

The pandas DataFrame has the same `plot` method as the Series, and uses the same `kind` parameter to select a plot. All of the Series plots are available to the DataFrame plus scatter and hexbin (a scatter plot that bins data into hexagons). One of the main differences with the DataFrame `plot` method, is that you can choose the columns to use along each axis with the `x` and `y` parameters.

You can also make plots by not supplying either of these parameters, and if you do, then each column will be plotted independently as the y-values, using the same index for the x-values. Just like most operations in pandas, plotting is **column-based** and you can think of each column as an independent Series. Let's begin by reading in the stocks dataset without putting the date in the index.

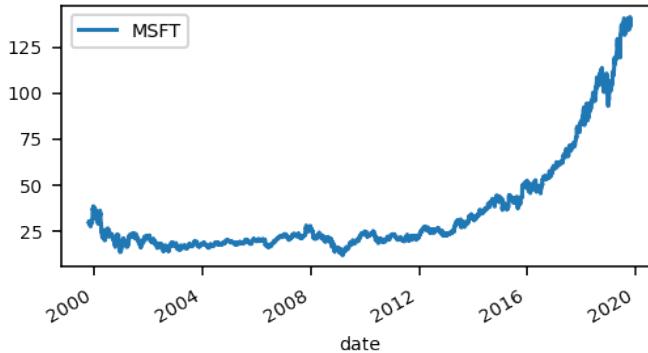
```
[1]: import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('.../.../mdap.mplstyle')
stocks = pd.read_csv('.../data/stocks/stocks10.csv', parse_dates=['date'])
stocks.head(3)
```

	date	MSFT	AAPL	SLB	AMZN	...	XOM	WMT	T	FB	V
0	1999-10-25	29.84	2.32	17.02	82.75	...	21.45	38.99	16.78	NaN	NaN
1	1999-10-26	29.82	2.34	16.65	81.25	...	20.89	37.11	17.28	NaN	NaN
2	1999-10-27	29.33	2.38	16.52	75.94	...	20.80	36.94	18.27	NaN	NaN

69.1 Line plots

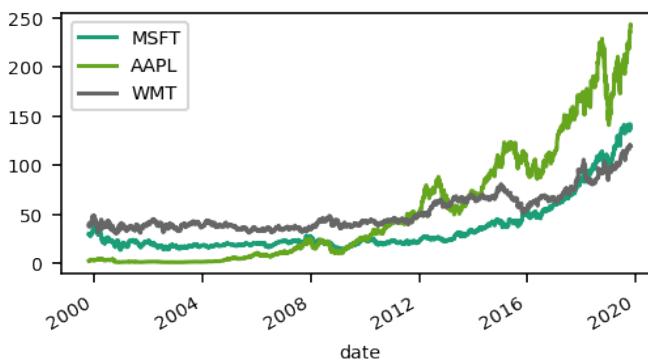
Most kinds of DataFrame plots allow you to choose between explicitly setting `x` and `y` or implicitly using the index as the x-values and each column as the y-values. Line plots allow both. Here, we plot Microsoft's closing price for each date in the dataset. By default, the column name used for the y-values will be used as the legend label.

```
[2]: stocks.plot(x='date', y='MSFT', kind='line');
```



More than one independent line can be plotted by using a list for y. A specific qualitative colormap is chosen.

```
[3]: ax = stocks.plot(x='date', y=['MSFT', 'AAPL', 'WMT'], kind='line', cmap='Dark2')
```



The returned object is a matplotlib axes which we can use to access each of the lines.

```
[4]: ax.lines
```

```
[4]: [<matplotlib.lines.Line2D at 0x116b37d90>,
 <matplotlib.lines.Line2D at 0x117655ad0>,
 <matplotlib.lines.Line2D at 0x116d42890>]
```

Implicitly plotting all columns

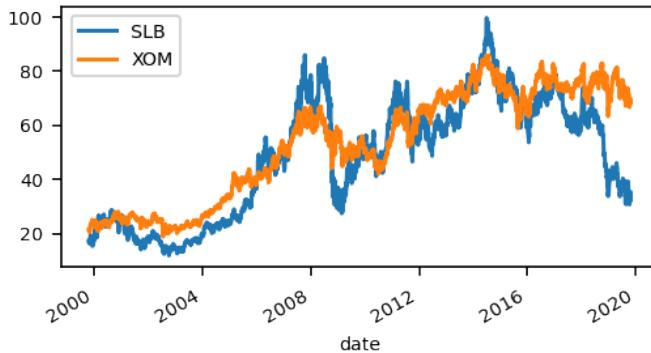
Let's put the date in the index and select the two oil companies, Schlumberger and ExxonMobil.

```
[5]: oil_stocks = stocks.set_index('date')[['SLB', 'XOM']]
oil_stocks.head(3)
```

	SLB	XOM
date		
1999-10-25	17.02	21.45
1999-10-26	16.65	20.89
1999-10-27	16.52	20.80

We'll call the `plot` method without providing either `x` or `y`. Each column of values is implicitly used as the `y`-values for an independent line with the index as the `x`-values for each.

```
[6]: oil_stocks.plot(kind='line');
```



69.2 Bar plots

Bar plots work just like line plots. You can explicitly set `x` and `y` or you can implicitly allow pandas to use the index as the `x`-values and each column as a set of bars of the same color. We'll use the life expectancy data which contains the average life expectancy from nearly every country from 2000 to 2016.

```
[7]: life_exp = pd.read_csv('../data/life_expectancy.csv')
life_exp.head(3)
```

	year	country	sex	life_expectancy
0	2000	Afghanistan	all	55.9
1	2000	Afghanistan	female	57.3
2	2000	Afghanistan	male	54.6

Let's find the five countries with the lowest life expectancy for all sexes in the year 2000 and plot every fourth year of data. We use `query` to filter the data and `nsmallest` to select the five lowest countries by life expectancy.

```
[8]: low_le = life_exp.query('year == 2000 and sex == "all"') \
    .nsmallest(5, 'life_expectancy')
low_le
```

	year	country	sex	life_expectancy
435	2000	Sierra Leone	all	39.8
159	2000	Eritrea	all	43.9
543	2000	Zambia	all	44.4
93	2000	Central African Republic	all	45.4
408	2000	Rwanda	all	45.7

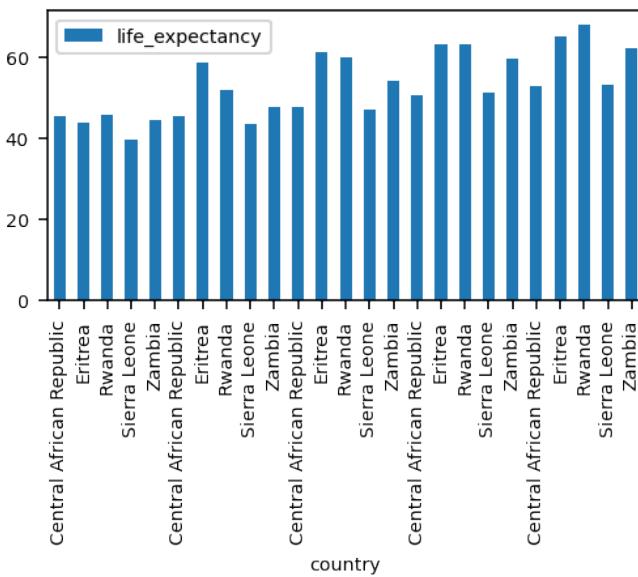
We can now run another query on the original data to get life expectancy every fourth year.

```
[9]: low_countries = low_le['country']
low_le_4th = life_exp.query("country in @low_countries and year % 4 == 0 and sex == 'all'")
low_le_4th.head(10)
```

	year	country	sex	life_expectancy
93	2000	Central African Republic	all	45.4
159	2000	Eritrea	all	43.9
408	2000	Rwanda	all	45.7
435	2000	Sierra Leone	all	39.8
543	2000	Zambia	all	44.4
2289	2004	Central African Republic	all	45.5
2355	2004	Eritrea	all	58.7
2604	2004	Rwanda	all	51.9
2631	2004	Sierra Leone	all	43.4
2739	2004	Zambia	all	47.6

Getting the plot we desire isn't possible with the current structure of the data. Using the countries as the x-values allows us to infer the year, but still doesn't make a good plot.

```
[10]: low_le_4th.plot(x='country', y='life_expectancy', kind='bar');
```



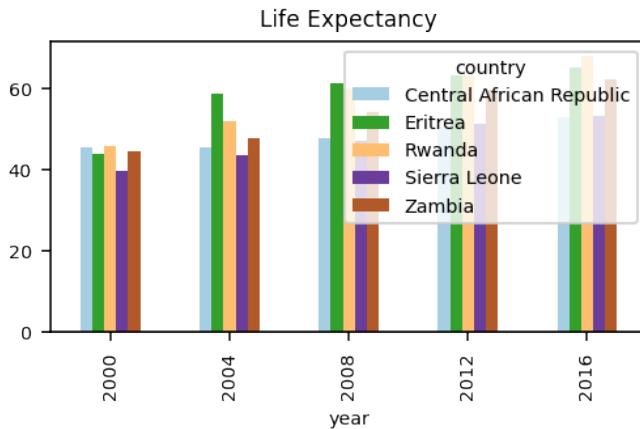
We need to make each column a country with values equal to the life expectancy. Let's use the `pivot` method to change the structure of the data.

```
[11]: country_le = low_le_4th.pivot(index='year', columns='country', values='life_expectancy')
country_le
```

country	Central African Republic	Eritrea	Rwanda	Sierra Leone	Zambia
year					
2000	45.4	43.9	45.7	39.8	44.4
2004	45.5	58.7	51.9	43.4	47.6
2008	47.8	61.3	59.9	47.1	54.0
2012	50.7	63.3	63.1	51.1	59.6
2016	53.0	65.0	68.0	53.1	62.3

We can now use pandas implicit plotting so that every column is plotted as its own set of unique bars.

```
[12]: ax = country_le.plot(kind='bar', title='Life Expectancy', cmap='Paired')
```



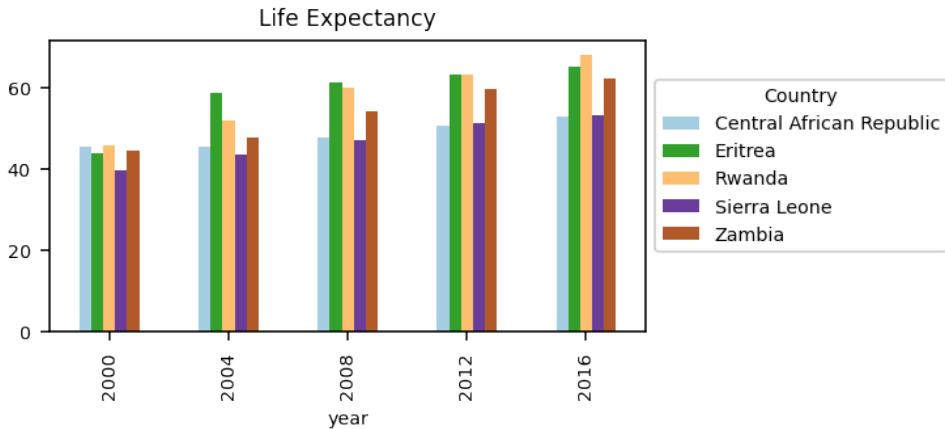
Each country now has its own set of bars, but the legend is taking up too much space. pandas internally supplies the `label` to each set of bars and calls matplotlib's axes `legend` method for us. We can verify that the label has been set by retrieving the list of containers (there are five total, one for each country), selecting one of the elements and using a getter method to return the label.

```
[13]: ax.containers[0].get_label()
```

```
[13]: 'Central African Republic'
```

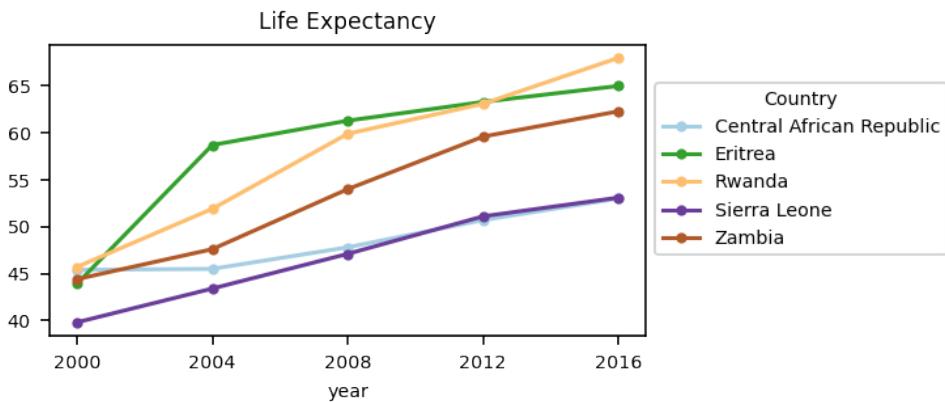
Since the labels are already set, we can just call the `legend` method again from the axes and use `bbox_to_anchor` to specify an exact location for the legend.

```
[14]: ax.legend(bbox_to_anchor=(1, .9), loc='upper left', title='Country')
ax.figure
```



A line plot is a better choice to see the trend by country. Adding markers is important to denote the exact position of the data we are plotting. Each tick, which was originally placed two years apart, is changed to every four years.

```
[15]: from matplotlib import ticker
ax = country_le.plot(kind='line', title='Life Expectancy', cmap='Paired', marker='.')
ax.legend(bbox_to_anchor=(1, .9), title='Country')
ax.xaxis.set_major_locator(ticker.MultipleLocator(4))
```



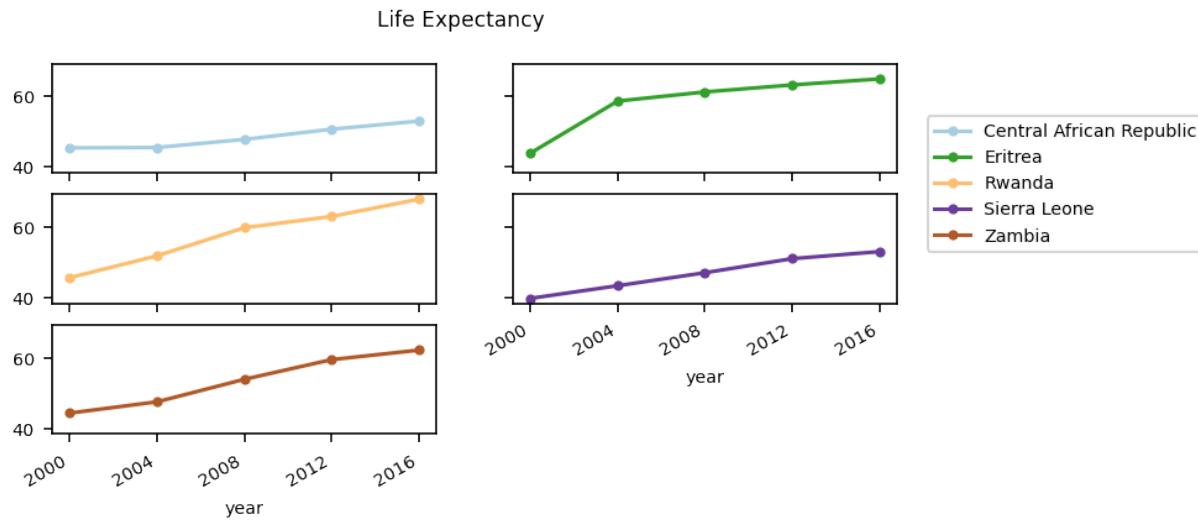
69.3 Plotting on separate axes

Instead of plotting DataFrame columns on the same axes, each one can be plotted on a separate axes by setting the `subplots` parameter to `True`. Set the `layout` parameter to a two-item integer tuple of the number of rows and columns. The boolean parameters `sharex` and `sharey`, when set to `True`, ensure that the limits and ticks of the respective axis are the same.

Each country is plotted on its own axes. By sharing both the x and y axis, comparing the countries becomes much easier. Also, only the left axes have y-tick labels. It appears as though only five axes were created, one for each country, but any axes that doesn't contain data has the attribute `visible` set to `False`. pandas returns a numpy array with each of the six axes we created. The original x-axis tick marks were set at five year intervals. Since all of the x-axes are shared, changing one changes them all.

```
[16]: ax_array = country_le.plot(kind='line', cmap='Paired', title='Life Expectancy',
                               marker='.', figsize=(6, 3), layout=(3, 2), subplots=True,
                               legend=False, sharex=True, sharey=True)
ax1 = ax_array[0, 0]
```

```
fig = ax1.figure
fig.legend(bbox_to_anchor=(.92, .8), loc='upper left', bbox_transform=fig.transFigure)
ax1.xaxis.set_major_locator(ticker.MultipleLocator(4))
ax1.minorticks_off()
```



69.4 Distribution plots

The same three distribution plots, box, histogram, and KDE are available for DataFrames. They work a little differently as the other plots as they disregard the `x` parameter and only use `y`. Let's read in the neighborhood and sale price along with several columns containing information on square footage from the housing dataset.

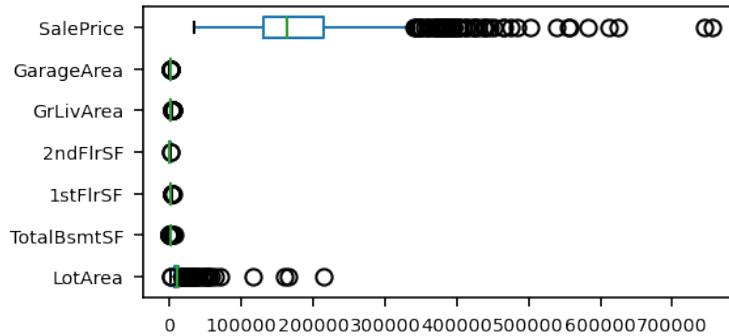
```
[17]: cols = ['Neighborhood', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF',
           'GrLivArea', 'GarageArea', 'LotArea', 'SalePrice']
housing = pd.read_csv('../data/housing.csv', usecols=cols)
housing.head(3)
```

	LotArea	Neighborhood	TotalBsmtSF	1stFlrSF	2ndFlrSF	GrLivArea	GarageArea	SalePrice
0	8450	CollgCr	856	856	854	1710	548	208500
1	9600	Veenker	1262	1262	0	1262	460	181500
2	11250	CollgCr	920	920	866	1786	608	223500

Box plots

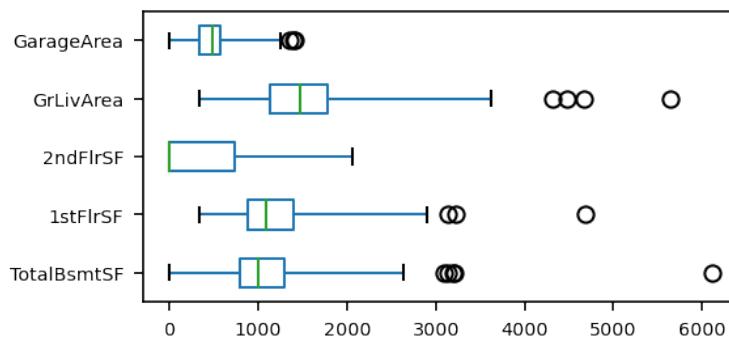
Without specifying `x` or `y`, a box plot is created for each numeric column. All non-numeric columns, such as neighborhood, are silently dropped. The x-limits (or y-limits when vertical) will range from the overall lowest to highest values from all variables, so a single variable like `LotArea` or `SalePrice` which are around 1,000 times greater than the others can dominate the plot.

```
[18]: housing.plot(kind='box', vert=False);
```



You can choose specific columns to plot with the y (and NOT x) parameter by setting it to a list. We also extend the whiskers to three times the interquartile range.

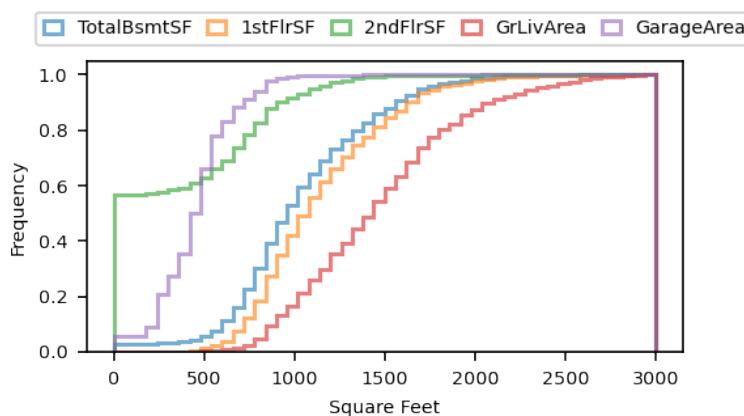
```
[19]: housing.plot(y=cols[1:-2], kind='box', vert=False, whis=3);
```



Histograms

Each of the five columns in the last box plot is made into a cumulative step histogram with the relative frequency on the y-axis. From the graph you can see that close to 60% of houses have no second floor and that nearly all houses have less than 1,000 total basement square feet.

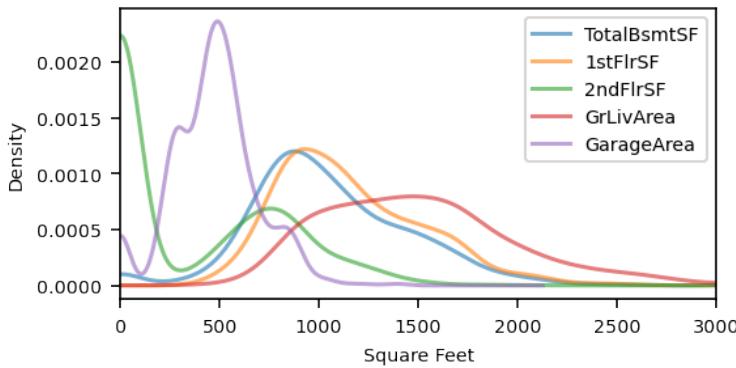
```
[20]: ax = housing.plot(y=cols[1:-2], kind='hist', bins=50, range=(0, 3_000), alpha=.6,
                      histtype='step', cumulative=True, lw=1.5, density=True)
ax.set_xlabel('Square Feet')
ax.legend(bbox_to_anchor=(-.1, 1.2), loc='upper left', ncol=5,
          handlelength=1, columnspacing=.8);
```



KDEs

KDEs work similarly as histograms. Each column is plotted as an independent KDE.

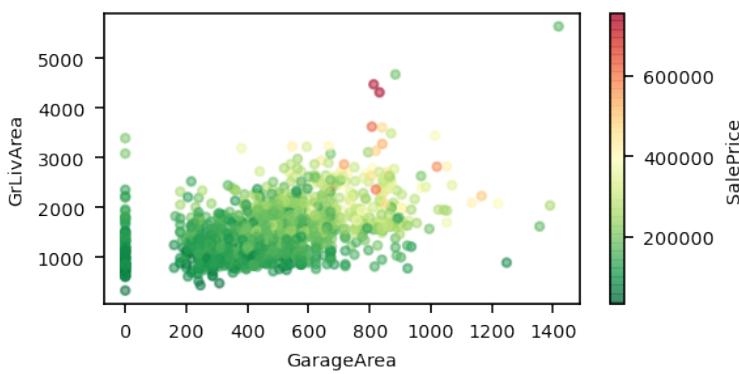
```
[21]: ax = housing.plot(y=cols[1:-2], kind='kde', alpha=.6, lw=1.5, xlim=(0, 3_000), legend=True)
ax.set_xlabel('Square Feet');
```



69.5 Scatter and Hexbin

Scatter and hexbin plots are only possible with DataFrames, not with Series. They both work differently than the plots above and **require** you to set `x` and `y` to a single column name. Below, we make a scatter plot of garage area versus living area using the `c` parameter to color by sale price. A colorbar is also created automatically. There's currently a bug in pandas requiring us to manually make the x-axis label visible and to set the tick position.

```
[22]: ax = housing.plot(x='GarageArea', y='GrLivArea', kind='scatter', s=10,
                      c='SalePrice', cmap='RdYlGn_r', alpha=.5)
ax.xaxis.label.set_visible(True)
ax.tick_params(axis='x', labelbottom=True)
```

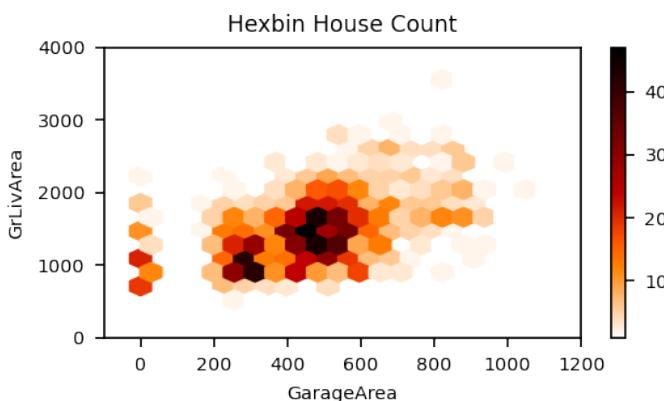


Hexbin plots are similar to scatter plots. They divide up the region they are plotting into equally-sized hexagons, and count the number of observations that fall within the bounds of each hexagon. These hexagons are then filled in with a color corresponding to the number of observations in that bin. pandas uses the `matplotlib hexbin` plot, so check out its documentation to learn about all of its parameters.

The `gridsize` parameter controls the number of hexagons to create along the `x` and `y` directions. If a single integer is provided, it is used as the number in the `x` direction with matplotlib computing the number in the `y`-direction so that the hexagons are regular. Set it to a tuple to control the exact number in both directions.

By default, pandas uses the BuGn sequential colormap. Below, we choose a different colormap, the reverse of `gist_heat`. The `mincnt` parameter controls the minimum count needed for a hexagon to be shown. This is useful when you have a colormap that does not have white as its minimum value. It prevents filling hexagons with a color if they have no count.

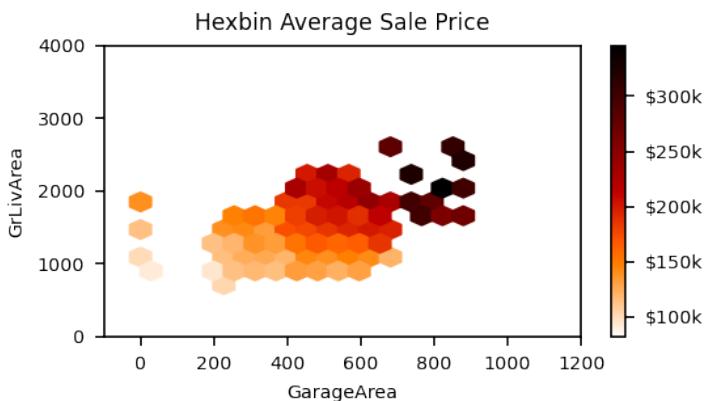
```
[23]: ax = housing.plot(x='GarageArea', y='GrLivArea', kind='hexbin',
                      xlim=(-100, 1_200), ylim=(0, 4_000), title='Hexbin House Count',
                      cmap='gist_heat_r', gridsize=25, mincnt=1)
ax.xaxis.label.set_visible(True)
ax.tick_params(axis='x', labelbottom=True)
```



pandas automatically adds a colorbar to the figure for us. The darkest hexagons have 40 or more houses with those particular combinations for garage area and living area. Hexbins are quite useful alternatives for scatter plots whenever there are many overlapping points. They can help get a better feel for the actual distribution.

The above plot merely counted the observations in each hexagon. It's possible to aggregate another variable and use the result for the color instead of the count. Set the `C` parameter to the column name you want to aggregate. By default, the average value of this variable will be calculated. Below, the average sale price is used to color the hexagons. The colorbar axes is selected and labels are formatted.

```
[24]: ax = housing.plot(x='GarageArea', y='GrLivArea', C='SalePrice', kind='hexbin',
                      xlim=(-100, 1_200), ylim=(0, 4_000), title='Hexbin Average SalePrice',
                      cmap='gist_heat_r', gridsize=25, mincnt=5)
ax.xaxis.label.set_visible(True)
ax.tick_params(axis='x', labelbottom=True)
ax_colorbar = ax.figure.axes[-1]
ax_colorbar.yaxis.set_major_formatter(ticker.StrMethodFormatter('${x:.3}k'))
```



Relative representation of each sex by experience and salary

In this section, we'll divide City of Houston employees into bins by experience and salary and report the relative representation of each sex. For instance, 70% of employees are male. If the group of employees with 10 years of experience and a salary of 80,000 is 60% male, then males are underrepresented by 10%. Let's read in the data and create a column for years of experience.

```
[25]: emp = pd.read_csv('../data/employee.csv', parse_dates=['hire_date'])
emp['experience'] = 2019 - emp['hire_date'].dt.year
emp.head(3)
```

	dept	title	hire_date	salary	sex	race	experience
0	Police	POLICE SERGEANT	2001-12-03	87545.38	Male	White	18
1	Other	ASSISTANT CITY ATTORNEY II	2010-11-15	82182.00	Male	Hispanic	9
2	Houston Public Works	SENIOR SLUDGE PROCESSOR	2006-01-09	49275.00	Male	Black	13

Let's get the relative frequency of each sex and verify that 70% of employees are male.

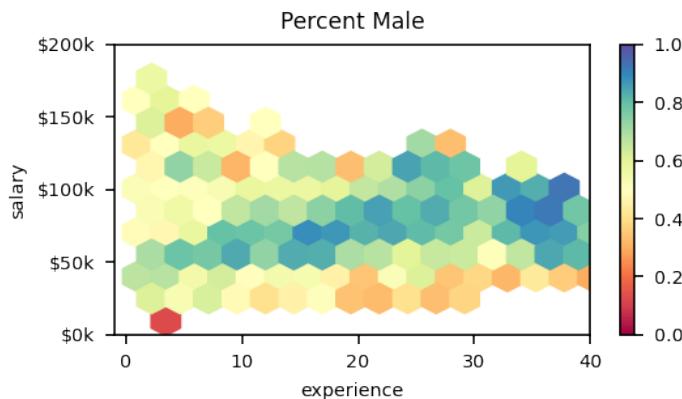
```
[26]: emp['sex'].value_counts(normalize=True).round(2)
```

```
[26]: Male      0.7
Female    0.3
Name: sex, dtype: float64
```

Before finding the relative representation of each sex, we will find the percent male per group using a hexbin plot. We create a numeric binary column equal to 1 for male employees and pass this to the `C` parameter, which is aggregated with the mean function by default. The divergent colormap 'Spectral' is chosen since there are two unique values for sex. To ensure that the midpoint of the colormap is .5, `vmin` and `vmax` are set to 0 and 1. A minimum of 5 employees is set for the hexagon to be shown.

```
[27]: emp['is_male'] = (emp['sex'] == 'Male') * 1
ax = emp.plot(x='experience', y='salary', C='is_male', kind='hexbin',
               title='Percent Male', xlim=(-1, 40), ylim=(0, 200_000),
               gridsize=20, cmap='Spectral', vmin=0, vmax=1, mincnt=5)
ax.xaxis.label.set_visible(True)
```

```
ax.tick_params(axis='x', labelbottom=True)
ax.yaxis.set_major_formatter(ticker.FuncFormatter(lambda x, pos: f'${x / 1000:.0f}k'))
```

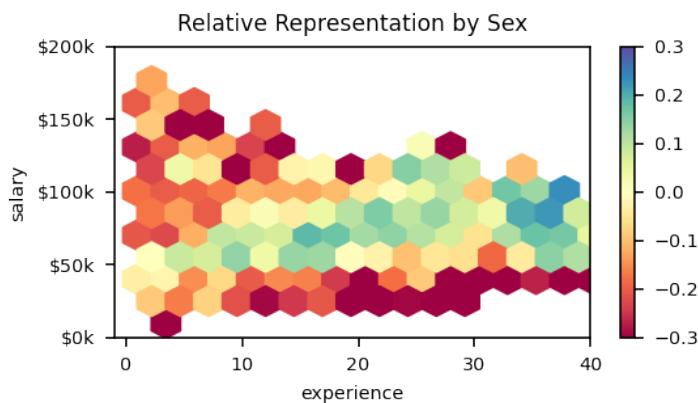


To get the relative representation of each group, we'll need to write our own custom aggregation function and pass it to the `reduce_C_function` parameter. matplotlib implicitly passes this function a list of all the values of the `C` column for that group.

In our custom function `relative_male`, we calculate the relative frequency of males and subtract the overall relative frequency of males calculated above (.7). This gives us a metric of over/under representation of each sex per group. Values greater than 0 have an over-representation of males (more than 70%) while those less than 0 are under-represented.

```
[28]: def relative_male(x):
    return pd.Series(x).mean() - .7

ax = emp.plot(x='experience', y='salary', C='is_male', kind='hexbin',
               title='Relative Representation by Sex', xlim=(-1, 40), ylim=(0, 200_000),
               gridsize=20, cmap='Spectral', vmin=-.3, vmax=.3, mincnt=5,
               reduce_C_function=relative_male)
ax.xaxis.label.set_visible(True)
ax.tick_params(axis='x', labelbottom=True)
ax.yaxis.set_major_formatter(ticker.FuncFormatter(lambda x, pos: f'${x / 1000:.0f}k'))
```



69.6 Area plots

Area plots are a great way to show emerging trends over time. In this section, we'll make different area plots based on the currently ongoing COVID-19 pandemic. Let's read in data collected by the World Health

Organization (WHO) with new deaths by day for each country.

```
[29]: c19_new_deaths = pd.read_csv('../data/WHO/new_deaths.csv', index_col='date',
                                 parse_dates=['date'])
c19_new_deaths.tail(3)
```

	World	Afghanistan	Albania	Algeria	Andorra	...	Vatican	Venezuela	Vietnam	Zambia	Zimbabwe
date											
2020-03-23	1660.0	0.0	0.0	5.0	0.0	...	0.0	0.0	0.0	0.0	0.0
2020-03-24	1764.0	1.0	2.0	2.0	0.0	...	0.0	0.0	0.0	0.0	1.0
2020-03-25	2200.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0

To get the total deaths for each date, take the cumulative sum of each column.

```
[30]: c19_total = c19_new_deaths.cumsum()
c19_total.tail(3)
```

	World	Afghanistan	Albania	Algeria	Andorra	...	Vatican	Venezuela	Vietnam	Zambia	Zimbabwe
date											
2020-03-23	14601.0	0.0	2.0	15.0	0.0	...	0.0	0.0	0.0	0.0	0.0
2020-03-24	16365.0	1.0	4.0	17.0	0.0	...	0.0	0.0	0.0	0.0	1.0
2020-03-25	18565.0	1.0	5.0	17.0	0.0	...	0.0	0.0	0.0	0.0	1.0

Making a plot containing all of the more than 180 countries would be very messy. Let's select the five countries with most deaths at the current time by sorting horizontally by the last index date.

```
[31]: last_date = c19_total.index[-1]
c19_total_top = c19_total.sort_values(last_date, axis=1, ascending=False)
c19_total_top = c19_total_top.iloc[:, :6]
c19_total_top.tail(3)
```

	World	Italy	China	Spain	Iran	France
date						
2020-03-23	14601.0	5476.0	3276.0	1720.0	1685.0	674.0
2020-03-24	16365.0	6077.0	3283.0	2182.0	1812.0	860.0
2020-03-25	18565.0	6820.0	3287.0	2696.0	1934.0	1100.0

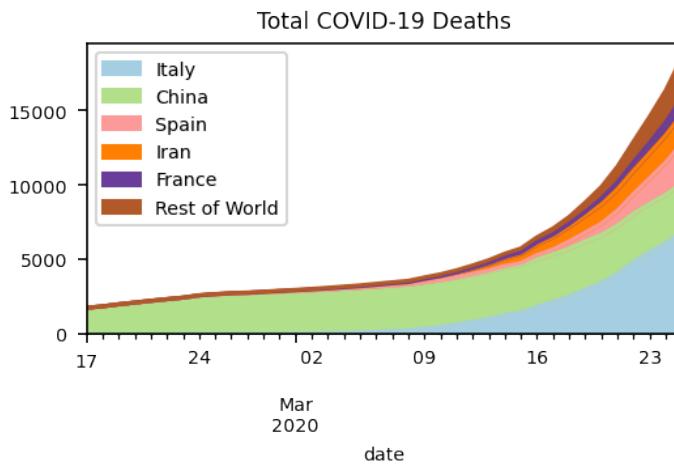
Let's create a column called 'Rest of World' and set it equal to the total of all deaths outside of these countries and then drop the 'World' column.

```
[32]: c19_total_top['Rest of World'] = c19_total_top['World'] - c19_total_top.iloc[:, 1:].
    ↪sum(axis=1)
c19_total_top = c19_total_top.drop(columns='World')
c19_total_top.tail(3)
```

	Italy	China	Spain	Iran	France	Rest of World
date						
2020-03-23	5476.0	3276.0	1720.0	1685.0	674.0	1770.0
2020-03-24	6077.0	3283.0	2182.0	1812.0	860.0	2151.0
2020-03-25	6820.0	3287.0	2696.0	1934.0	1100.0	2728.0

We can now create an area plot that draws a line for each column beginning with the first column. Each subsequent column uses the cumulative total of all y-values before it. The area between each lines is filled in with a different color. This vertical area represents the number of deaths for each country at each date. We begin the plot in mid-February after a larger number of deaths had already occurred.

```
[33]: c19_total_top_recent = c19_total_top['2020-02-17':]
c19_total_top_recent.plot(kind='area', title='Total COVID-19 Deaths', cmap='Paired');
```



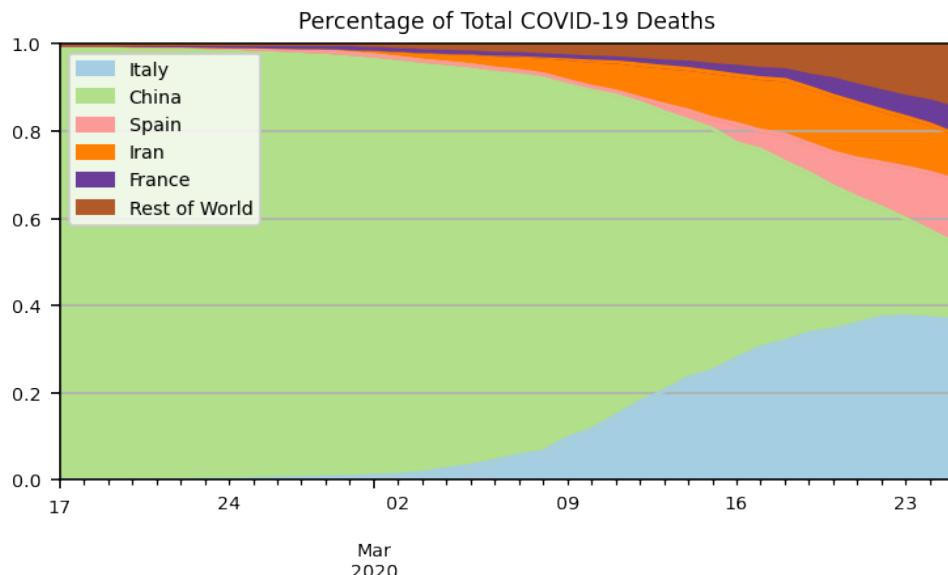
Instead of plotting the total number of deaths, we can plot each country's percentage of the total world deaths per day. To get these percentages, we first sum horizontally to get the total deaths in the world and then divide each row by its corresponding world total. We must use the `div` method to change the direction of the division so that the rows align.

```
[34]: total_world = c19_total_top_recent.sum(axis=1)
c19_perc_top_recent = c19_total_top_recent.div(total_world, axis=0)
c19_perc_top_recent.tail(3).round(2)
```

	Italy	China	Spain	Iran	France	Rest of World
date						
2020-03-23	0.38	0.22	0.12	0.12	0.05	0.12
2020-03-24	0.37	0.20	0.13	0.11	0.05	0.13
2020-03-25	0.37	0.18	0.15	0.10	0.06	0.15

Another area plot can be made. The height of the stacked lines must sum to 1 each day. Using a relative metric allows us to use more space on the plotting surface and show emerging trends clearer.

```
[35]: ax = c19_perc_top_recent.plot(kind='area', title='Percentage of Total COVID-19 Deaths',
                                    cmap='Paired', figsize=(6, 3), legend=True, ylim=(0, 1))
ax.set_xlabel('')
ax.grid(True, axis='y')
```



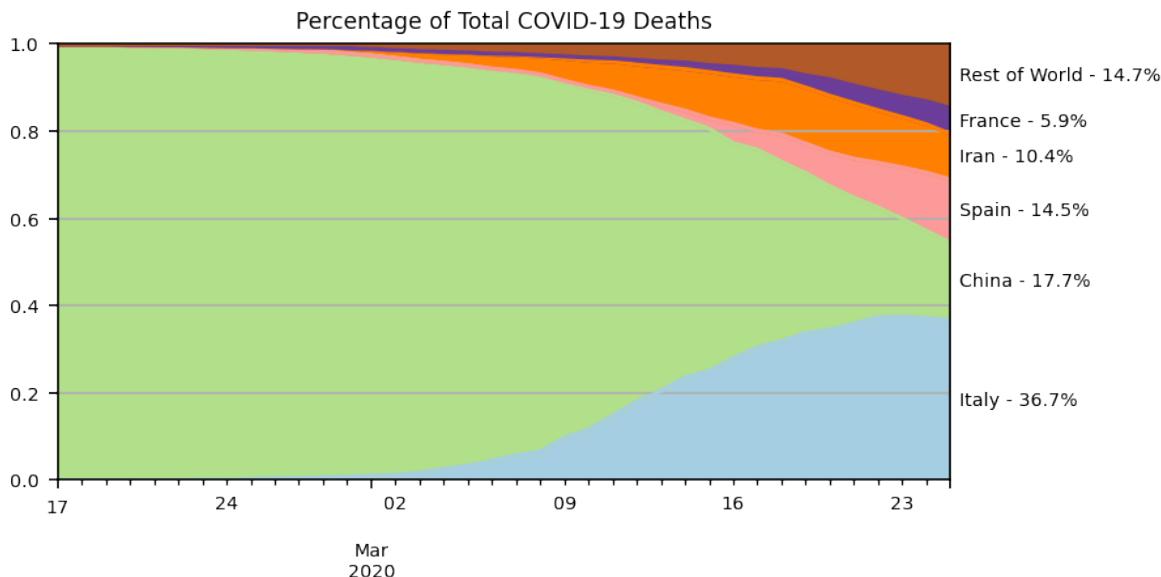
Instead of having a legend, we can place text on the right side of the axes denoting the country and the last percentage. Let's get the last row of data.

```
[36]: last_perc = c19_perc_top_recent.iloc[-1]
last_perc.round(3)
```

```
[36]: Italy          0.367
China          0.177
Spain          0.145
Iran           0.104
France         0.059
Rest of World  0.147
Name: 2020-03-25 00:00:00, dtype: float64
```

We remove the legend and iterate through the country name and associated value by converting the Series to a dictionary. We use axes coordinates (`ax.transAxes`) to place the text just outside of the axes as the y-values are in the same units for both the data and axes coordinates, making it possible to use the percentages to make the placement.

```
[37]: ax.legend_.remove()
y = 0
for country, perc in last_perc.to_dict().items():
    ax.text(x=1.01, y=y + perc / 2, s=f'{country} - {perc:.1%}',
            ha='left', va='center', transform=ax.transAxes)
    y += perc
ax.figure
```



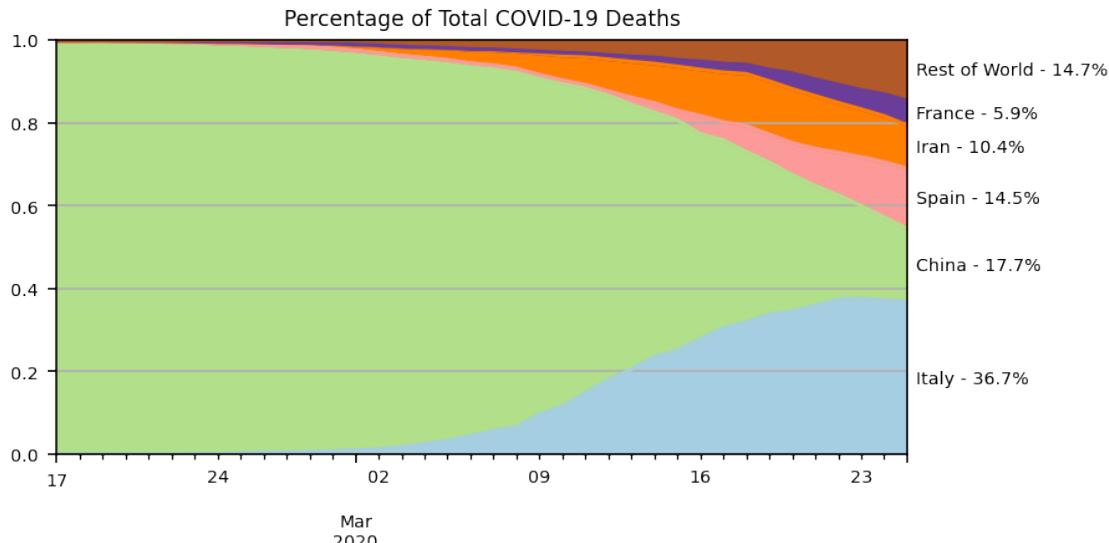
matplotlib allows you to add tables of data directly below your plots with the `table` axes method. We create our table by selecting five rows spaced out at even intervals and then transpose it so the dates are horizontal.

```
[38]: step = len(c19_total_top_recent) // 5 + 1
table = c19_total_top_recent[::-step].T.astype('int64')
table
```

date	2020-02-17	2020-02-25	2020-03-04	2020-03-12	2020-03-20
Italy	0	6	80	827	3407
China	1771	2665	2983	3172	3254
Spain	0	0	0	47	767
Iran	0	12	77	354	1284
France	1	1	4	48	372
Rest of World	3	14	58	170	800

The axes `table` method is now called needing the cell data passed as a numpy array. Other parameters are used to set the row and column labels. The `bbox` parameter sets the rectangular region (left, bottom, width, height) where the table will be placed.

```
[39]: ax.table(cellText=table.values, rowLabels=table.index,
            colLabels=table.columns.astype('str'), bbox=(0, -.8, 1, .5))
ax.figure
```



	2020-02-17	2020-02-25	2020-03-04	2020-03-12	2020-03-20
Italy	0	6	80	827	3407
China	1771	2665	2983	3172	3254
Spain	0	0	0	47	767
Iran	0	12	77	354	1284
France	1	1	4	48	372
Rest of World	3	14	58	170	800

We can collect different insights by plotting the percentage of new deaths by country instead of the total. Let's begin by going back to our original data and finding the top 9 countries by total number of deaths.

```
[40]: top10 = c19_new_deaths.sum().nlargest(10)
top10
```

```
[40]: World      18565.0
Italy       6820.0
China      3287.0
Spain      2696.0
Iran       1934.0
France     1100.0
United States 801.0
United Kingdom 422.0
Netherlands 276.0
Germany    149.0
dtype: float64
```

We select these countries and then calculate a three day rolling average of the new deaths. We do this because daily data is erratic and subject to errors. Smoothing it out using three days of data will prevent jagged spikes in our plot.

```
[41]: new_deaths_top = c19_new_deaths[top10.index].copy()
new_deaths_top_avg = new_deaths_top.rolling(3, center=True, min_periods=1).mean()
new_deaths_top_avg.tail(3).round(0)
```

	World	Italy	China	Spain	Iran	France	United States	United Kingdom	Netherlands	Germany
date										
2020-03-23	1705.0	682.0	7.0	393.0	126.0	137.0	110.0	53.0	36.0	27.0
2020-03-24	1875.0	664.0	7.0	457.0	126.0	179.0	154.0	63.0	47.0	27.0
2020-03-25	1982.0	672.0	6.0	488.0	124.0	213.0	165.0	70.0	48.0	28.0

We calculate the ‘Rest of World’ column again.

```
[42]: new_deaths_top_avg['World'] = new_deaths_top_avg['World'] - new_deaths_top_avg.iloc[:,1:-1].sum(1)
new_deaths_top_avg = new_deaths_top_avg.rename(columns={'World': 'Rest of World'})
new_deaths_top_avg.tail(3).round(0)
```

	Rest of World	Italy	China	Spain	Iran	France	United States	United Kingdom	Netherlands	Germany
date										
2020-03-23	134.0	682.0	7.0	393.0	126.0	137.0	110.0	53.0	36.0	27.0
2020-03-24	151.0	664.0	7.0	457.0	126.0	179.0	154.0	63.0	47.0	27.0
2020-03-25	168.0	672.0	6.0	488.0	124.0	213.0	165.0	70.0	48.0	28.0

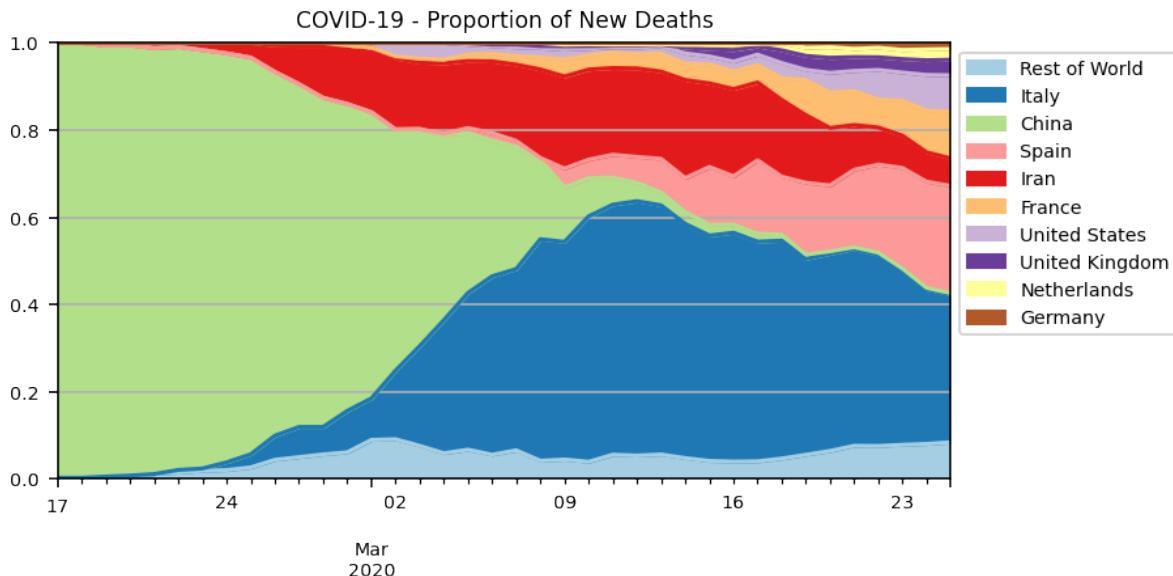
Dividing by the daily total yields the proportion per day.

```
[43]: new_deaths_top_perc = new_deaths_top_avg.div(new_deaths_top_avg.sum(axis=1), axis=0)
new_deaths_top_perc.tail(3).round(2)
```

	Rest of World	Italy	China	Spain	Iran	France	United States	United Kingdom	Netherlands	Germany
date										
2020-03-23	0.08	0.40	0.0	0.23	0.07	0.08	0.06	0.03	0.02	0.02
2020-03-24	0.08	0.35	0.0	0.24	0.07	0.10	0.08	0.03	0.02	0.01
2020-03-25	0.08	0.34	0.0	0.25	0.06	0.11	0.08	0.04	0.02	0.01

We make another area plot to reveal a different emerging trend. China’s contribution to the daily total has shrunk from all to nearly none.

```
[44]: pct_daily_final = new_deaths_top_perc['2020-02-17':]
ax = pct_daily_final.plot(kind='area', figsize=(6, 3), ylim=(0, 1), legend=False,
                           cmap='Paired', title='COVID-19 - Proportion of New Deaths')
ax.grid(axis='y')
ax.legend(bbox_to_anchor=(1, 1), loc='upper left')
ax.set_xlabel('');
```



Chapter 70

Seaborn Axes Plots

This chapter introduces the seaborn visualization library in Python. seaborn has a high-level, easy-to-use interface for creating powerful and beautiful visualizations. Like pandas, seaborn relies entirely on matplotlib to do all of the actual plotting. The library is fairly minimal and exposes relatively few functions.

70.1 The seaborn API

Visit the [seaborn API page](#) to get a nice overview of the library. The top of the API shows the below sections that contain most of the plotting function in the library. The other sections cover less important topics such as making specific grids and aesthetics. This chapter focuses only on the plots in these sections.

- Relational
- Categorical
- Distribution
- Regression
- Matrix

Axes and Grid plots

All of the seaborn plotting functions return either a matplotlib axes or a seaborn grid. As the name implies, these axes plots use a single matplotlib axes. Grid plots are more complex and are composed of a matplotlib figure with multiple axes. From the five sections in the API above, `relplot`, `catplot`, `lmplot` and `clustermap` are the only grid plots. These will be the focus of the following chapter. In this chapter, we focus on the axes plots.

A different categorization of plots

If you look at the seaborn API, you'll notice a section on categorical plots. Unfortunately, this isn't labeled properly as there are multiple distribution plotting functions such as box and violin plots in that section. Instead of using seaborn's classification, I like to divide the plotting functions into the following three categories:

- **Distribution plots** - These plots show the distribution of some set of points of a continuous valued variable. Examples of these are box, violin, histogram, and KDE plots.
- **Grouping and aggregating plots** - These plots group by some categorical variable and aggregate another. Examples of these plots include bar, count, and point plots.
- **Raw data plots** - These plots do not change the underlying data other than display it. Some examples are scatter and line plots along with heatmaps.

70.2 seaborn integration with pandas

seaborn is tightly integrated with pandas. Nearly all seaborn plotting functions contain a `data` parameter that accepts a pandas DataFrame. This allows you to use **strings** of the column names for the function arguments.

The four common seaborn plotting function parameters - `x`, `y`, `hue`, and `data`

Most seaborn plotting function signatures look similar and use the parameters `x`, `y`, `hue`, and `data`. The syntax will often look like one of the following lines of code, where `x`, `y`, and `hue` are all optional and set to a column name if used.

```
>>> sns.plotting_func(x='col1', data=df)
>>> sns.plotting_func(y='col1', data=df)
>>> sns.plotting_func(x='col1', y='col2', data=df)
>>> sns.plotting_func(x='col1', y='col2', hue='col3', data=df)
```

70.3 Distribution Plots

We'll begin our adventure in seaborn by making distribution plots. We've seen how to make box plots, histograms and KDEs both directly with matplotlib and with pandas. seaborn uses the functions `boxplot` for box plots, `distplot` for histograms and univariate KDEs, and `kdeplot` for bivariate KDEs. By convention, seaborn is imported as `sns`. Let's begin by reading in Airbnb listing data from Washington, D.C.

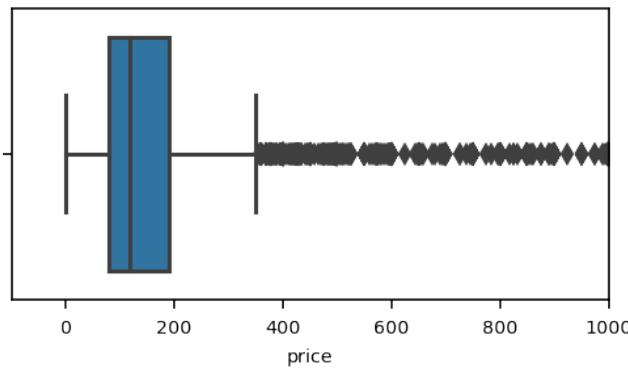
```
[1]: import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
plt.style.use('.../../../mdap.mplstyle')
airbnb = pd.read_csv('.../data/airbnb.csv')
airbnb.head(3)
```

	<code>id</code>	<code>neighborhood</code>	<code>property_type</code>	<code>accommodates</code>	<code>bathrooms</code>	...	<code>response_time</code>	<code>minimum_nights</code>	<code>maximum_nights</code>	<code>latitude</code>	<code>longitude</code>
0	3362	Shaw	Townhouse	16	3.5	...	within an hour	2	365	38.90982	77.02016
1	3663	Brightwood Park	Townhouse	4	3.5	...	Nan	3	30	38.95888	77.02554
2	3670	Howard University	Townhouse	2	1.0	...	Nan	2	30	38.91842	77.02750

Univariate distribution plots

Univariate distribution plots involve a single numeric variable. For these univariate plots, just one of the `x` or `y` parameters needs to be set to the DataFrame column name. Choosing `x` creates a horizontal plot with `y` creating a vertical one. Let's create a simple horizontal box plot of the price and assign the result (which is an axes) to a variable. Because of the presence of many values much larger than the median, we shorten the `x-limits`.

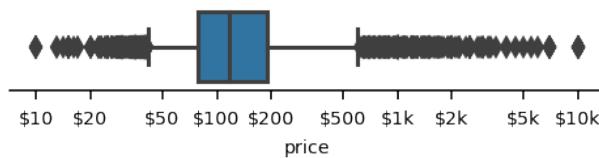
```
[2]: ax = sns.boxplot(x='price', data=airbnb)
ax.set_xlim(-100, 1000);
```



Let's do some work to change the appearance of this box plot. The height of the box is unnecessarily large. We'll control this by making the figure height much smaller. An `ax` parameter exists for all seaborn axes plots that can be set to an previously created axes.

The maximum price of the data is 10,000 and using a linear scale would compress the above data into less than 10% of the axes width. We'll use a log scale instead, which places major ticks every power of 10 (along with minor ticks) and uses scientific notation as the format. We use the `ticker` module to specify more major tick locations, remove the minor ticks, and format the x-axis labels as dollars. All spines but the bottom are made invisible.

```
[3]: from matplotlib import ticker
fig, ax = plt.subplots(figsize=(4, .6))
sns.boxplot(x='price', data=airbnb, whis=(5, 95), ax=ax)
ax.set_xscale('log')
ax.xaxis.set_major_locator(ticker.LogLocator(base=10, subs=(1, 2, 5)))
func = lambda x, pos: f'${x:.0f}' if x < 1000 else f'${x // 1000:.0f}k'
ax.xaxis.set_major_formatter(ticker.FuncFormatter(func))
ax.xaxis.set_minor_locator(ticker.NullLocator())
ax.yaxis.set_major_locator(ticker.NullLocator())
ax.spines['left'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```



Recreate box plot with matplotlib

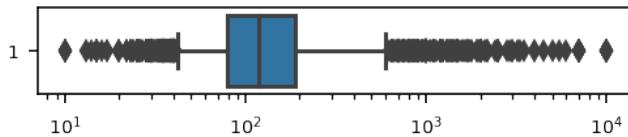
seaborn uses your current matplotlib run configuration settings for all of your plots. It creates the above plot by calling matplotlib's `boxplot` method, providing it many different settings for the median line, box, whiskers, caps (vertical lines at the end of the whiskers), and fliers. A recreation of the above seaborn plot is done below with matplotlib.

```
[4]: fig, ax = plt.subplots(figsize=(4, .6))
ax.boxplot(x='price', data=airbnb, whis=(5, 95), widths=.8, vert=False,
           patch_artist=True,
           medianprops={'color': '.25', 'lw': 1.5},
           boxprops={'ec': '.25', 'lw': 1.5, 'fc': '#3274a1'},
```

```

whiskerprops={'color': '.25', 'lw': 1.5},
capprops={'color': '.25', 'lw': 1.5},
flierprops={'marker': 'd', 'mfc': '.25', 'mec': '.25', 'ms': 5})
ax.set_xscale('log')

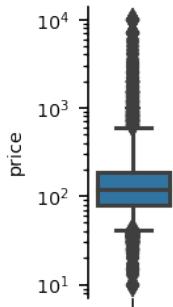
```



Vertical plots

To make a vertical plot, set the `y` parameter to the data you want to plot. To remove spines, you can use the seaborn `despine` function instead of accessing each spine directly like we did above. By default, the top and left spines are removed. We tell it to remove the bottom spine as well.

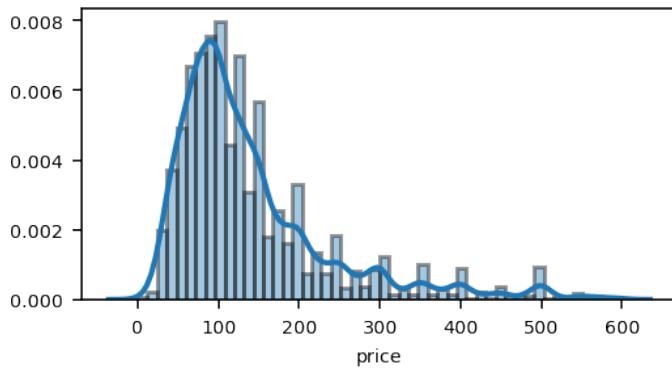
```
[5]: fig, ax = plt.subplots(figsize=(.6, 2))
sns.boxplot(y='price', data=airbnb, ax=ax, whis=(5, 95))
ax.set_yscale('log')
sns.despine(ax=ax, bottom=True)
```



Histograms and KDEs

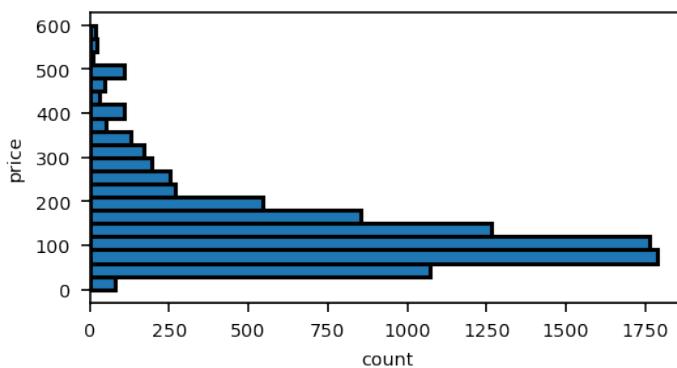
Both histograms and KDEs are made by the seaborn `distplot` function, which has a different signature than most of the other plotting functions. There is no `data` parameter. Instead, you pass the data as a Series as the first argument. By default, both a histogram and KDE are plotted together. The histogram and the KDE can both be tweaked further by setting the `hist_kws` and `kde_kws` parameters to dictionaries mapping the underlying matplotlib parameters to their values. Below we plot a histogram and KDE of the rental price of each listing under \$600.

```
[6]: price_600 = airbnb.query('price < 600')['price']
sns.distplot(price_600, hist_kws={'lw': 1.5, 'ec': 'black'}, kde_kws={'lw': 2});
```



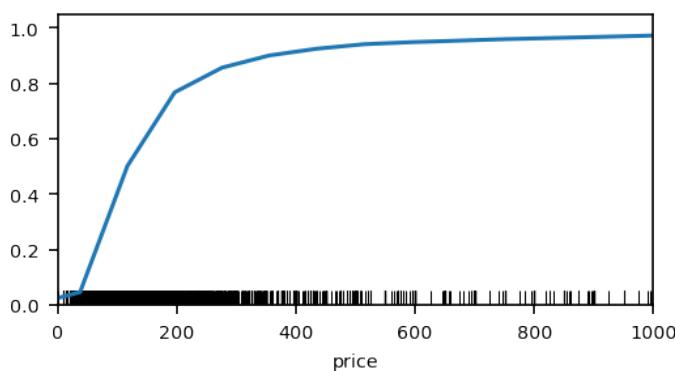
You can toggle the histogram and kde on and off by setting the boolean `hist` and `kde` parameters. We turn off the KDE and plot the histogram with its price going along the y-axis. When turning the KDE off, the raw counts of each bin are shown. seaborn attempts to use a reasonable number of bins, but they can be specified exactly with `bins`.

```
[7]: ax = sns.distplot(price_600, bins=20, hist_kws={'lw': 1.5, 'ec': 'black', 'alpha': 1},
                     kde=False, vertical=True)
ax.set_xlabel('count');
```



We can plot just the KDE by itself by setting `hist` to `False`. Theres also a third type of plot possible; a rug plot which places a small vertical line at every occurrence on the x-axis. These dashes are like the fliers on a box plot and good for detecting outliers. A cumulative KDE of all prices is created. We can see that the vast majority of listings have a price under 600 dollars.

```
[8]: ax = sns.distplot(airbnb['price'], hist=False, rug=True,
                     kde_kws={'cumulative': True}, rug_kws={'lw': .5, 'color':'black'})
ax.set_xlim(0, 1000);
```



Bivariate KDE plots

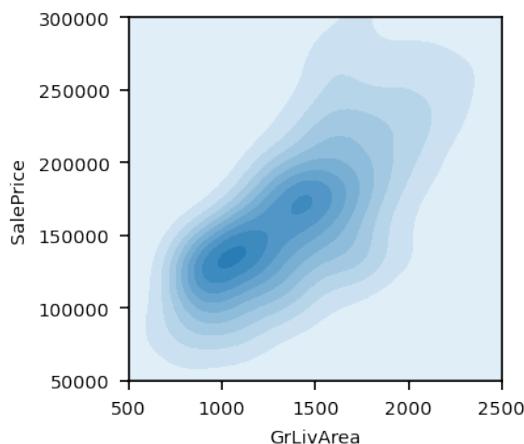
A bivariate KDE estimates the density of two numeric values co-occurring from two different variables. The `kdeplot` function in seaborn produces a bivariate KDE as contour lines of different colors along a sequential color map. Let's construct an example and then explain it further. We read in a few columns from the housing dataset, which has more continuous variables that make for better examples.

```
[9]: cols = ['OverallQual', 'GrLivArea', 'SalePrice']
housing = pd.read_csv('../data/housing.csv', usecols=cols)
housing.head(3)
```

	OverallQual	GrLivArea	SalePrice
0	7	1710	208500
1	6	1262	181500
2	7	1786	223500

The `kdeplot` function forces you to pass each column as a Series to the `data` and `data2` parameters. Here, we'll estimate the distribution of living area and sale price. We choose to shade in the contours and clip both the x and y limits to be where the majority of data is located.

```
[10]: fig, ax = plt.subplots(figsize=(2.5, 2.5))
sns.kdeplot(data=housing['GrLivArea'], data2=housing['SalePrice'],
             shade=True, clip=((500, 2_500), (50_000, 300_000)), ax=ax);
```

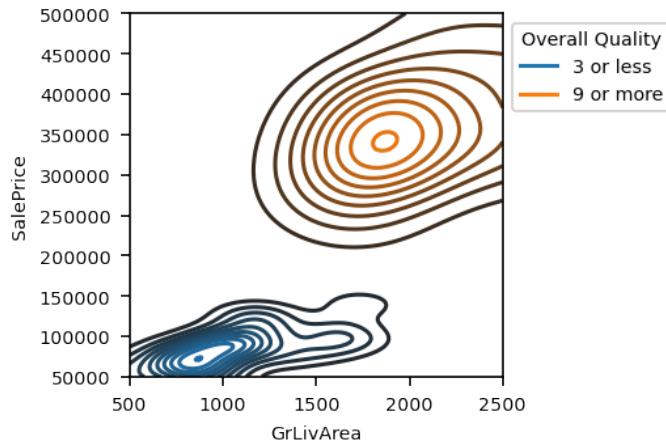


The darkest areas represent the greatest concentration of data. All contours with the same color have approximately the same probability of occurrence. In our dataset, houses around 1,000 square feet of living area priced at around \$140,000 are more common than others.

It's possible to plot multiple bivariate KDE's on the same axes. Here we use the same two variables to construct the KDE, but filter by those with the lowest and highest overall quality into separate DataFrames. The distributions for each group are quite distinct and do not overlap. We choose just to plot the contours without shading.

```
[11]: fig, ax = plt.subplots(figsize=(2.5, 2.5))
df = housing.query('OverallQual <= 3')
sns.kdeplot(data=df['GrLivArea'], data2=df['SalePrice'],
             clip=((500, 2_500), (50_000, 300_000)), ax=ax, label='3 or less')
```

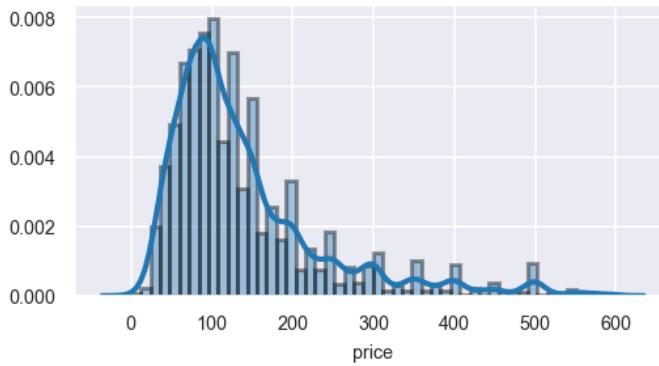
```
df = housing.query('OverallQual >= 9')
sns.kdeplot(data=df['GrLivArea'], data2=df['SalePrice'],
             clip=((500, 2_500), (50_000, 500_000)), ax=ax, label='9 or more')
ax.legend(bbox_to_anchor=(1, 1), loc='upper left', title='Overall Quality');
```



70.4 Seaborn style sheets

There are several seaborn style sheets that alter the appearance of a plot by changing some of the matplotlib configuration settings. Call the `set_style` function with one of the strings ‘darkgrid’, ‘whitegrid’, ‘dark’, ‘white’, or ‘ticks’. We will use the darkgrid for the remaining portion of this chapter which uses a light gray background color and white grid lines along with removing the tick marks. We’ll plot the same histogram and KDE from above to view the contrast in styles.

```
[12]: sns.set_style('darkgrid')
sns.distplot(price_600, hist_kws={'lw': 1.5, 'ec': 'black'}, kde_kws={'lw': 2});
```



70.5 Other distribution plots

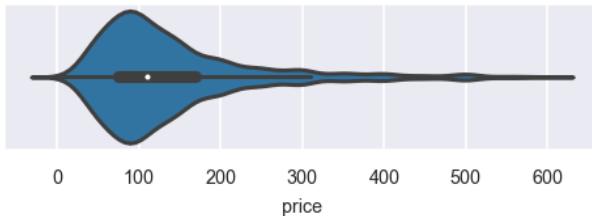
The following functions are also capable of producing univariate distribution plots and work similarly as `boxplot`. We’ll only cover `violinplot` below.

- `violinplot`
- `stripplot`
- `swarmplot`
- `boxenplot`

Violin plots

Violin plots produce the same KDE as `distplot` but duplicate the curve so that it appears both above and below the baseline. The final shape often resembles a violin because many distributions have a single peak and a long tail like the list price. `seaborn` uses the `matplotlib violinplot` method to actually create the plot.

```
[13]: fig, ax = plt.subplots(figsize=(4, 1))
sns.violinplot(x='price', data=airbnb.query('price < 600'), ax=ax);
```



To prove that the violin plot really is a KDE, we use `distplot` to create just the KDE. The shape of the curve is exactly the same.

```
[14]: fig, ax = plt.subplots(figsize=(4, .5))
sns.distplot(price_600, hist=False, ax=ax)
```

```
[14]: <matplotlib.axes._subplots.AxesSubplot at 0x11ef9da10>
```



Look back up at the original violin plot and you'll see a small white circle inside of a rectangle with a line running through it. `seaborn` creates a miniature box plot without fliers inside of the violin plot.

70.6 Automatic grouping by category

Thus far it would seem that `seaborn` isn't all that useful or necessary as `pandas` can duplicate all of the above plots besides the violin plot. The big benefit from `seaborn` comes when you want to group or split the data without manually doing so with `DataFrame` operations. `seaborn` automatically splits data into independent groups just by providing it the `DataFrame` column name.

For instance, let's say we are interested in making a box plot of listing price for each of the neighborhoods. This isn't possible to do using `pandas` without a for-loop iterating through each unique neighborhood name, filtering the data for that neighborhood and making a box plot. With `seaborn`, we just need to supply the grouping variable and plotting variable to the `x` and `y` parameters like this:

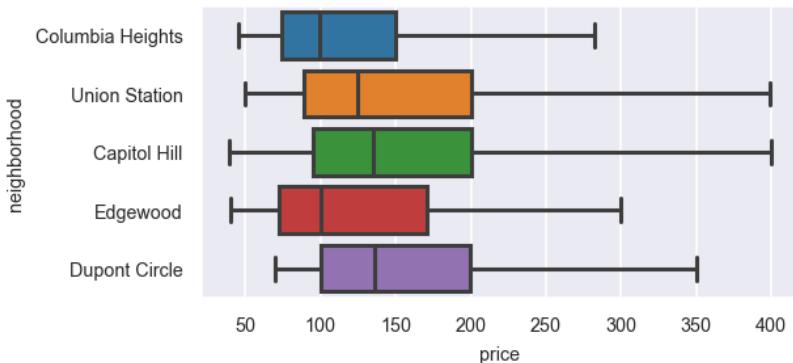
```
sns.boxplot(x='price', y='neighborhood', data=airbnb)
```

`seaborn` chooses the column that is of the object data type to use as the **grouping variable** and creates a different box plot for each of the unique values in this column using the other variable as the data. Internally, it uses `pandas` to make these splits, before using `matplotlib` to make the box plots.

Since there are dozens of unique neighborhoods, we'll plot just the five most common neighborhoods. It's not even necessary to filter the data first with `seaborn`, as you can set the `order` parameter to the subset of

categories that you want to plot. The boxes will also appear in that order. The fliers are not shown so that focus is drawn to the middle of the distribution to better compare neighborhood prices.

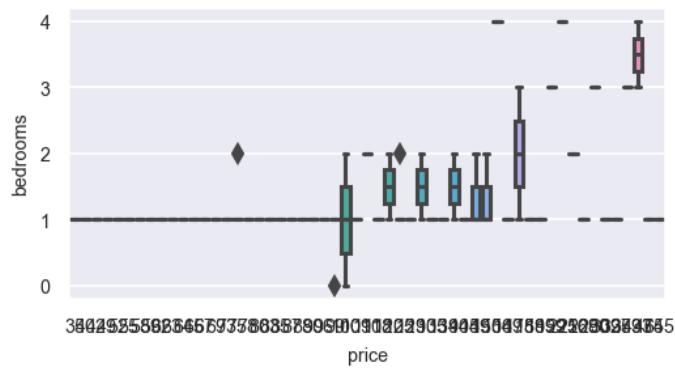
```
[15]: top5 = airbnb['neighborhood'].value_counts().index[:5]
sns.boxplot(x='price', y='neighborhood', data=airbnb, order=top5,
            whis=(5, 90), showfliers=False);
```



Grouping with two numeric variables

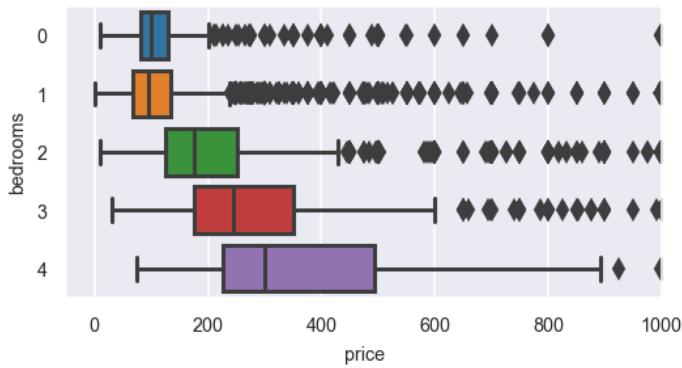
It's possible that your grouping variable is numeric, and when this happens seaborn automatically chooses `x` as the grouping variable. Below is an attempt to make a box plot of price for each unique number of bedrooms, but seaborn does the opposite, and makes a box plot for each unique price.

```
[16]: sns.boxplot(x='price', y='bedrooms', data=airbnb.head(100));
```



To force seaborn to use the variable set to `y` as the grouping column, set `orient` to '`h`' (horizontal). Set it to '`v`' to force a vertical plot. We filter the listings to show only those with four or less bedrooms by using the `order` parameter.

```
[17]: ax = sns.boxplot(x='price', y='bedrooms', data=airbnb, orient='h',
                     order=[0, 1, 2, 3, 4])
ax.set_xlim(-50, 1_000);
```

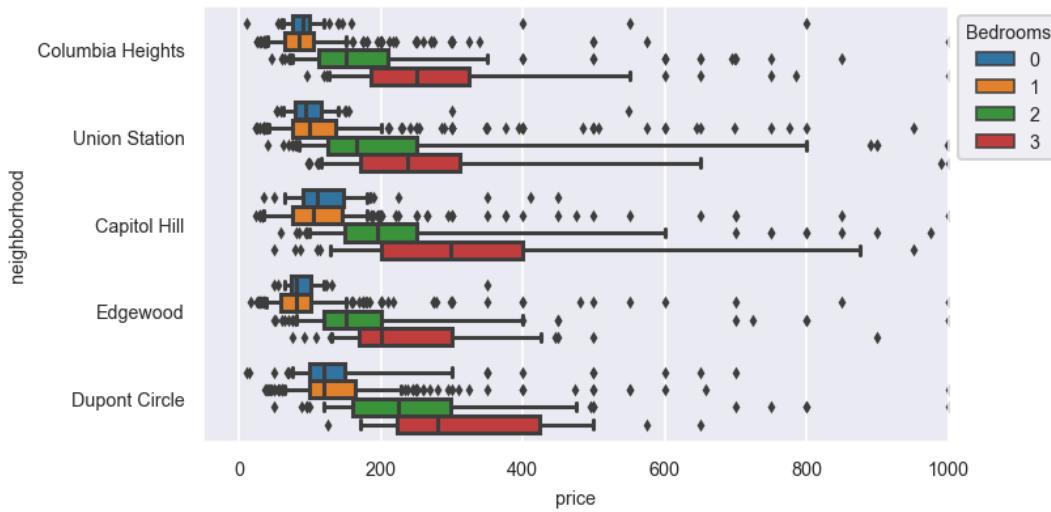


70.7 Grouping within groups with hue

Let's say we were interested in creating box plots for different numbers of bedroom for each neighborhood. `seaborn` allows you to create groups within your first group by setting the `hue` parameter to a column name. Below, we initially group by neighborhood. Within each neighborhood we will group by number of bedrooms and create a box plot for each of these groups.

Just as we used `order` to filter the main grouping column, we can use `hue_order` to filter this other grouping column. Both are used to limit the total number of combinations below. Also, the `fliersize` is decreased from its default value of 5 to fit better in this cramped plot.

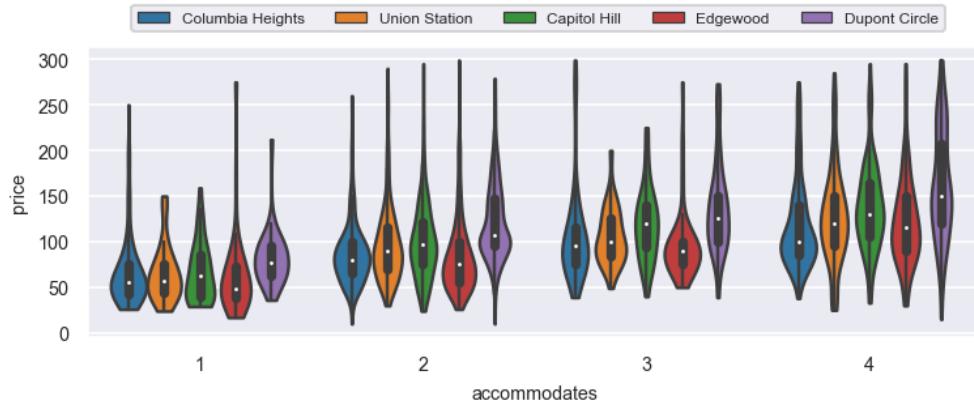
```
[18]: fig, ax = plt.subplots(figsize=(5, 3))
sns.boxplot(x='price', y='neighborhood', hue='bedrooms', data=airbnb, order=top5,
             hue_order=[0, 1, 2, 3], whis=(5, 90), fliersize=2, ax=ax)
ax.legend(bbox_to_anchor=(1, 1), loc='upper left', title='Bedrooms')
ax.set_xlim(-50, 1_000);
```



Grouping with violin plots

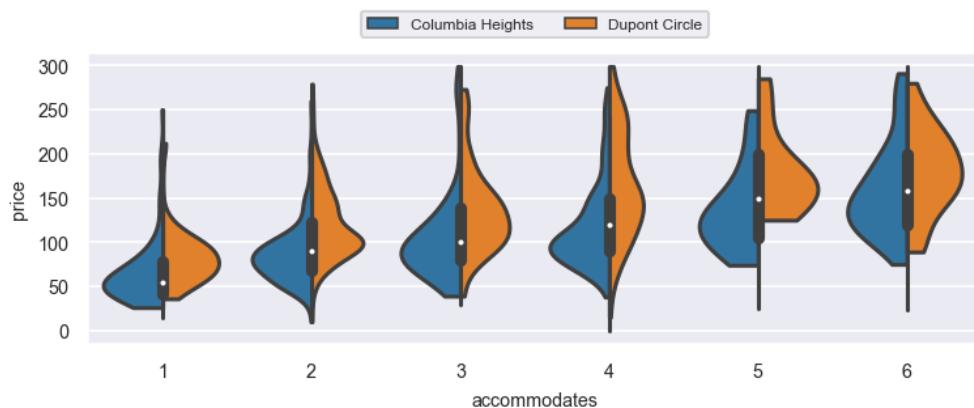
Violin plots can group and split data in the same manner. Here, we make vertical plots grouped first by the number of persons that the listing can accommodate before splitting into each of the top 5 most frequent neighborhoods. The data is filtered for listings under \$300. The `cut` parameter, when set to 0, does not show any part of the violin where actual data is not present.

```
[19]: fig, ax = plt.subplots(figsize=(6, 2))
airbnb_300 = airbnb.query('price < 300')
sns.violinplot(x='accommodates', y='price', data=airbnb_300, hue='neighborhood',
                 order=[1, 2, 3, 4], hue_order=top5, cut=0, linewidth=1.2, ax=ax)
ax.legend(bbox_to_anchor=(.04, 1.1), ncol=5, loc='center left', fontsize='small');
```



Violin plots have an interesting feature that allows you to make a comparison whenever there are exactly two categories for the hue parameter. Setting `split` to `True` creates a violin with the KDE's of each category on either side. Below, we compare prices of two neighborhoods for several different values of accommodates.

```
[20]: fig, ax = plt.subplots(figsize=(6, 2))
airbnb_300 = airbnb.query('price < 300')
neighs = ['Columbia Heights', 'Dupont Circle']
sns.violinplot(x='accommodates', y='price', data=airbnb_300, hue='neighborhood', split=True,
                 order=[1, 2, 3, 4, 5, 6], hue_order=neighs, cut=0, ax=ax)
ax.legend(bbox_to_anchor=(.5, 1.1), ncol=5, loc='center', fontsize='small');
```



70.8 Tidy data

seaborn is built to work with tidy data. It does all the grouping and aggregating for you. This is what makes it powerful. Once your data is tidy, you can create many different plots directly from seaborn without any further manipulation.

Comparison to pandas

The pandas `plot` method works quite different than seaborn plotting functions. In pandas, each column of the DataFrame is plotted as its own line, bar, box, histogram, etc... With pandas, you are responsible for grouping and aggregating the data on your own before plotting. With seaborn, the grouping and aggregating happens internally.

70.9 Grouping and Aggregating Plots

Our next category of possible plots are those that group and aggregate. These plots will create a single point statistic for some numeric column of data. The following are the grouping and aggregating functions, all found in the Categorical section of the API:

- `barplot`
- `pointplot`
- `countplot`
- `lineplot`

Univariate grouping and aggregating plots

Let's begin again like we did with the distribution plots by using a single numeric variable. The `barplot` function aggregates by taking the `mean` of the values by default. Below, we call the `barplot` function to create a single bar of the mean of the prices for all the listings.

```
[21]: fig, ax = plt.subplots(figsize=(1, 1.5))
sns.barplot(y='price', data=airbnb, ax=ax)
ax.set_title('Average Price for All Listings');
```



Notice the small black line appearing at the top of the bar. It represents the 95% confidence interval of the calculated statistic. It is computed by a procedure called bootstrapping which randomly samples the data with replacement to create an entire ‘new’ dataset. The mean of this new dataset is calculated. By default, 1,000 of these new datasets are produced and means calculated for each one. The 2.5 and 97.5 percentiles are found from these 1,000 values and used to draw the line above. Let's do this procedure ourselves with pandas by first calculating 1,000 different means and storing them in a Series.

```
[22]: price = airbnb['price']
means = pd.Series(price.sample(frac=1, replace=True).mean() for i in range(1000))
means.head()
```

```
[22]: 0    221.387929
1    206.218851
2    208.785575
3    218.109048
4    219.356079
```

```
dtype: float64
```

We can use the `quantile` method to find the range of the 95% confidence interval.

```
[23]: means.quantile([.025, .975])
```

```
[23]: 0.025    205.816705
0.975    227.685616
dtype: float64
```

These numbers should be very close to the y-values from the little black line in our plot from above, which we retrieve below.

```
[24]: ax.lines[0].get_ydata()
```

```
[24]: array([205.35413788, 227.48583459])
```

The number of new datasets used during bootstrapping can be controlled with the `n_boot` parameter and the confidence interval with `ci`. Let's make the same bar plot, but make an 80% confidence interval with less bootstrapped datasets.

```
[25]: fig, ax = plt.subplots(figsize=(1, 1.5))
sns.barplot(y='price', data=airbnb, ax=ax, ci=80, n_boot=200)
ax.set_title('Average Price for All Listings (80% CI)');
```



The `estimator` parameter

Both `barplot` and `pointplot` have an `estimator` parameter, which is used to choose the **aggregating function** and is defaulted to the numpy mean function. Unfortunately, seaborn doesn't allow you to use the string names of the function like pandas, so you'll have to use the numpy function directly. Here, we take the median of all listings and choose not to show the confidence interval by setting `ci` to `None`.

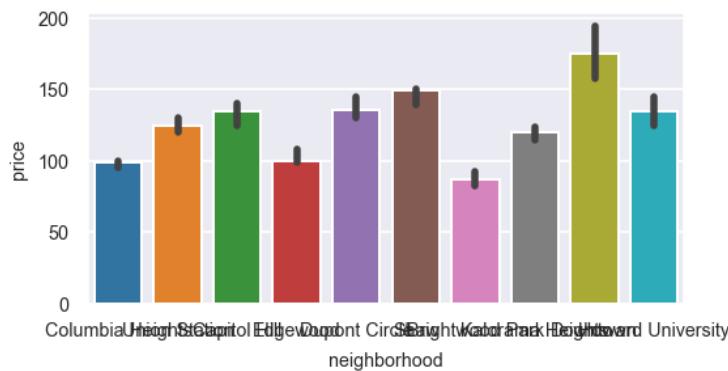
```
[26]: fig, ax = plt.subplots(figsize=(1, 1.5))
sns.barplot(y='price', data=airbnb, estimator=np.median, ci=None)
ax.set_title('Median Price for All Listings');
```



Grouping by one variable

Just as we did with box and violin plots, we can group data by providing both the `x` and `y` parameter. `seaborn` uses the categorical column as the grouping column and aggregates the numeric column. Here, we get the median price of each neighborhood.

```
[27]: top10 = airbnb['neighborhood'].value_counts().index[:10]
sns.barplot(x='neighborhood', y='price', data=airbnb, estimator=np.median,
             order=top10);
```

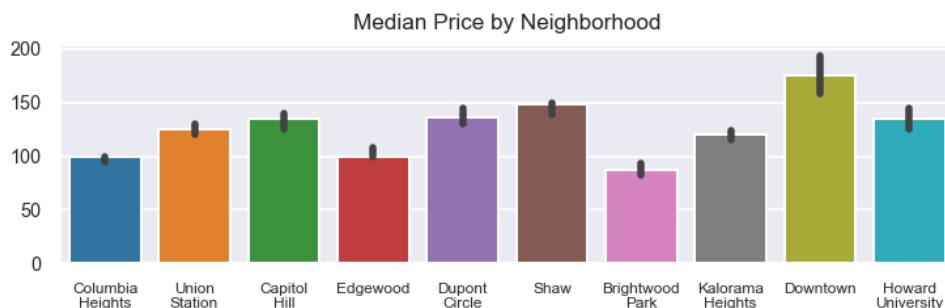


The bars show the correct median for each neighborhood, but the presentation is sloppy. `seaborn` does not provide the option to format tick labels so we'll do it ourselves. We could rotate the labels, but instead choose to use the standard library's `textwrap` module's `fill` function to wrap them at the given character width. This function also sets the font size.

```
[28]: import textwrap
def wrap_labels(ax, width, fontsize='medium'):
    labels = []
    for label in ax.get_xticklabels():
        text = label.get_text()
        labels.append(textwrap.fill(text, width=width, break_long_words=False))
    ax.set_xticklabels(labels, fontsize=fontsize, rotation=0)
```

We recreate the same plot, but make it wider and use the new labels at a smaller size.

```
[29]: fig, ax = plt.subplots(figsize=(6, 1.5))
sns.barplot(x='neighborhood', y='price', data=airbnb, estimator=np.median, order=top10)
ax.set(title='Median Price by Neighborhood', xlabel='', ylabel='')
wrap_labels(ax, width=10, fontsize='small')
```



We can further split each of these neighborhoods into more groups with the `hue` parameter. Here, we find the median price by neighborhood for listings that can accommodate up to five persons. The confidence

interval is also removed as it can take quite a long time to compute with many different groupings.

```
[30]: fig, ax = plt.subplots(figsize=(6, 1.5))
sns.barplot(x='neighborhood', y='price', data=airbnb, estimator=np.median,
             hue='accommodates',
             order=top10, hue_order=[1, 2, 3, 4, 5], ci=None, ax=ax)
ax.legend(bbox_to_anchor=(1, 1), loc='upper left', title='Accommodates')
ax.set(title='Median Price by Neighborhood', xlabel='', ylabel='')
wrap_labels(ax, width=10, fontsize='small')
```



Point plots

The `pointplot` function behaves similarly to `barplot` but instead of creating bars, it places points at the calculated statistic. When not specifying `hue`, a single line connects the calculated statistic for each group. The `scale` parameter controls the relative size of the line and point and is set to 1 by default. The confidence interval of the statistic is placed as a vertical line through the point. Use `errwidth` and `capsize` to control its appearance.

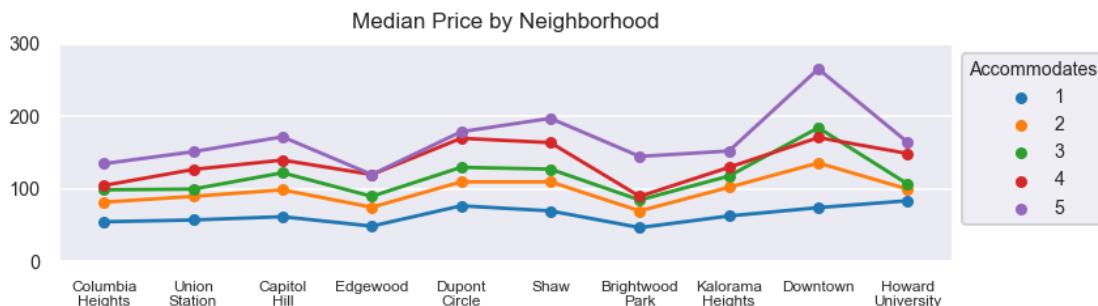
```
[31]: fig, ax = plt.subplots(figsize=(6, 1.5))
sns.pointplot(x='neighborhood', y='price', data=airbnb, estimator=np.mean,
               order=top10, scale=.5, errwidth=1, capsize=.2)
ax.set(title='Mean Price by Neighborhood', xlabel='', ylabel='')
wrap_labels(ax, width=10, fontsize='small')
```



Notice that the y-axis does not begin at zero like it did with bar plots, so it can give the illusion of greater differences between neighborhoods than actually exist. If you split the data using `hue`, a separate line for each unique category is created. Here, we recreate the last bar plot as a point plot and manually set the y-axis so it begins at 0.

```
[32]: fig, ax = plt.subplots(figsize=(6, 1.5))
sns.pointplot(x='neighborhood', y='price', data=airbnb, estimator=np.median,
               hue='accommodates',
```

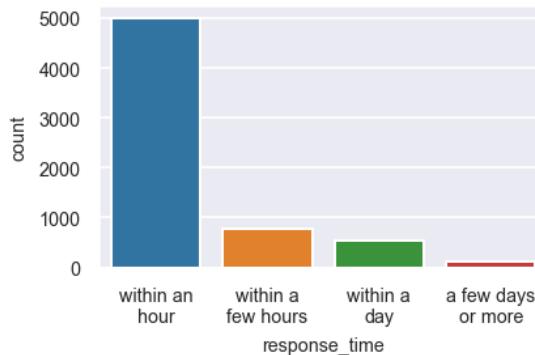
```
order=top10, hue_order=[1, 2, 3, 4, 5], ci=None, scale=.5, ax=ax)
wrap_labels(ax, width=10, fontsize='small')
ax.legend(bbox_to_anchor=(1, 1), loc='upper left', title='Accommodates')
ax.set(title='Median Price by Neighborhood', xlabel='', ylabel='', ylim=(0, 300));
```



Count plots

The `countplot` function can be thought of as a specific case of `barplot` that calculates the size of each group as its only aggregating function. Let's count the frequency of each unique response time.

```
[33]: fig, ax = plt.subplots(figsize=(3, 1.8))
sns.countplot(x='response_time', data=airbnb, ax=ax)
wrap_labels(ax, width=10)
```



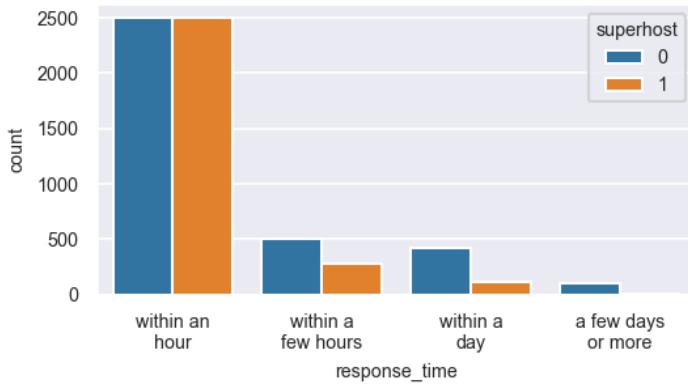
This is the same exact calculation produced by the Series `value_counts` method.

```
[34]: airbnb['response_time'].value_counts()
```

```
[34]: within an hour      4992
within a few hours     777
within a day           533
a few days or more    111
Name: response_time, dtype: int64
```

The `countplot` does not allow you to pass it both `x` and `y`, but you can use `hue` to further split the data. Here, we split by the binary variable '`superhost`'. Airbnb defines a "superhost" as an experienced host who offers exceptional experiences.

```
[35]: fig, ax = plt.subplots()
sns.countplot(x='response_time', data=airbnb, hue='superhost', ax=ax)
wrap_labels(ax, width=10)
```



This plot of raw counts can be misleading, since there are not the same total number of each host type. You might wrongly infer from the graph that both types of host have the same rate of response for “within an hour”. Let’s replicate the raw counts using pandas `crosstab` function.

```
[36]: pd.crosstab(index=airbnb['response_time'], columns=airbnb['superhost'])
```

	superhost	0	1
response_time			
a few days or more	105	6	
within a day	419	114	
within a few hours	504	273	
within an hour	2497	2495	

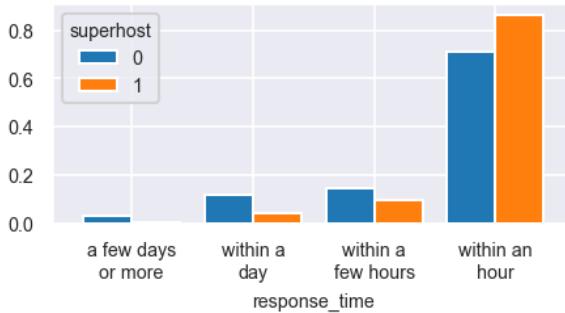
Instead, normalizing by the total count of each host type could get us a more accurate comparison between them. Now, we can see that 86.4% of superhosts answer within an hour vs 70.8% of non-superhosts.

```
[37]: df = pd.crosstab(index=airbnb['response_time'], columns=airbnb['superhost'],
                      normalize='columns')
df.round(3)
```

	superhost	0	1
response_time			
a few days or more	0.030	0.002	
within a day	0.119	0.039	
within a few hours	0.143	0.095	
within an hour	0.708	0.864	

Surprisingly, this plot isn’t easily manageable in seaborn as `countplot` does not do any normalization, so we plot the DataFrame above directly in pandas.

```
[38]: ax = df.plot(kind='bar', figsize=(3.5, 1.5), width=.8)
wrap_labels(ax, width=10)
```



Line plots

The seaborn `lineplot` function is similar to `pointplot`, but does not draw markers at every point. It only aggregates the y variable and expects x to be either numeric or datetime but not categorical. Let's read in the full COVID-19 dataset, which contains both new and total case and death data for nearly every country in the world.

```
[39]: full_covid = pd.read_csv('../data/WHO/full_covid_data.csv', parse_dates=['date'])
full_covid.head(3)
```

	date	country	new_cases	new_deaths	total_cases	total_deaths
0	2019-12-31	Afghanistan	0	0	0	0
1	2020-01-01	Afghanistan	0	0	0	0
2	2020-01-02	Afghanistan	0	0	0	0

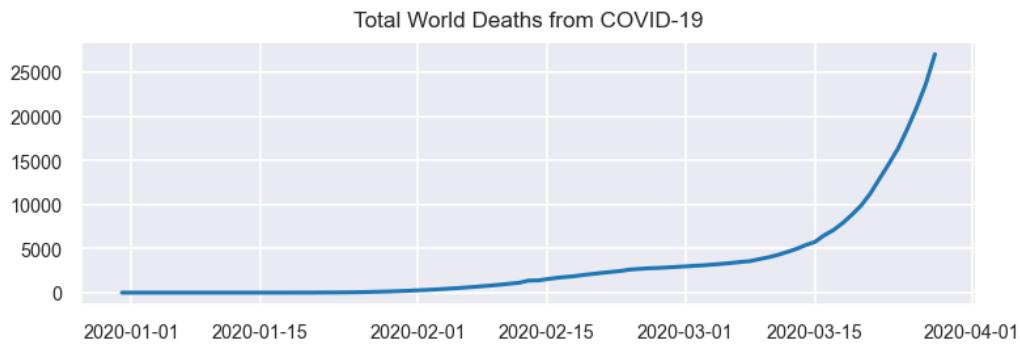
This dataset is tidy and contains total deaths in a single column. It's not possible to make line plots of total world deaths directly in pandas. You'd have to group and aggregate the data first like this:

```
[40]: full_covid.groupby('date').agg({'total_deaths': 'sum'}).tail(4)
```

total_deaths	
	date
2020-03-25	18558
2020-03-26	20981
2020-03-27	23665
2020-03-28	27083

Instead, we'll use seaborn to do the grouping and aggregation for us. Here, we set the `estimator` to the numpy `sum` function.

```
[41]: fig, ax = plt.subplots(figsize=(6, 1.8))
sns.lineplot(x='date', y='total_deaths', data=full_covid, ci=None, estimator=np.sum, ↴
             ax=ax)
ax.set(xlabel='', ylabel='', title='Total World Deaths from COVID-19');
```



If we are interested in making line plots by country using pandas, we'd need to pivot the data first.

```
[42]: full_covid.pivot(index='date', columns='country', values='total_deaths').iloc[:3, :10]
```

country	Afghanistan	Albania	Algeria	Andorra	Angola	Anguilla	Antigua and Barbuda	Argentina	Armenia	Aruba
date										
2019-12-31	0.0	NaN	0.0	NaN	NaN	NaN	NaN	NaN	0.0	NaN
2020-01-01	0.0	NaN	0.0	NaN	NaN	NaN	NaN	NaN	0.0	NaN
2020-01-02	0.0	NaN	0.0	NaN	NaN	NaN	NaN	NaN	0.0	NaN

Because we are using seaborn this isn't necessary. Since there are nearly 200 countries, we will select just the top 7 countries with most deaths. We first get the last date of data.

```
[43]: last_date = full_covid['date'].max()
last_date
```

```
[43]: Timestamp('2020-03-28 00:00:00')
```

We filter for the countries that have the highest total on this date.

```
[44]: highest_deaths = full_covid.query('date == @last_date').nlargest(7, 'total_deaths')
highest_deaths
```

	date	country	new_cases	new_deaths	total_cases	total_deaths
3615	2020-03-28	Italy	5959	971	86498	9136
6313	2020-03-28	Spain	7871	769	64059	4858
1429	2020-03-28	China	134	3	82213	3301
3258	2020-03-28	Iran	2926	144	32332	2378
2399	2020-03-28	France	3809	299	32964	1995
7156	2020-03-28	United States	18695	411	104686	1707
7067	2020-03-28	United Kingdom	2885	181	14543	759

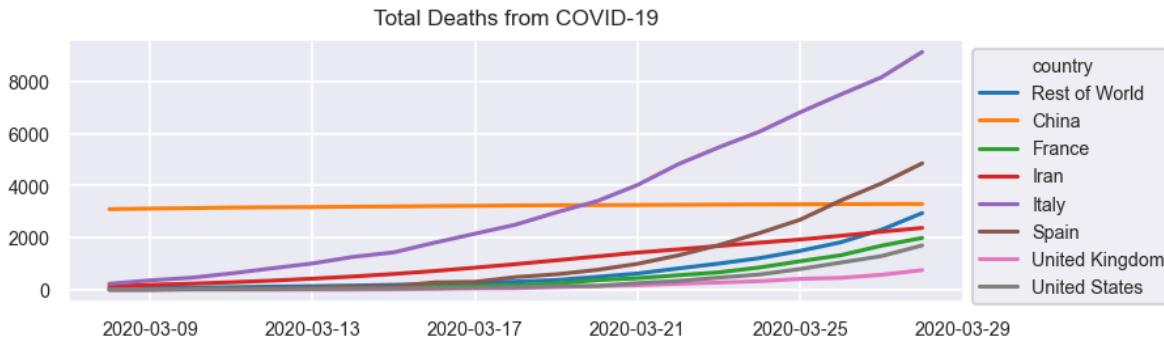
We set all the other country values to 'Rest of World' in the original dataset.

```
[45]: filt = ~full_covid['country'].isin(highest_deaths['country'])
full_covid.loc[filt, 'country'] = 'Rest of World'
full_covid.head(3)
```

	date	country	new_cases	new_deaths	total_cases	total_deaths
0	2019-12-31	Rest of World	0	0	0	0
1	2020-01-01	Rest of World	0	0	0	0
2	2020-01-02	Rest of World	0	0	0	0

We can now make a line plot by country using `hue`. Note that the sum aggregation is only doing an actual summation for 'Rest of World'. The other countries have exactly one row for each date.

```
[46]: fig, ax = plt.subplots(figsize=(6, 1.8))
sns.lineplot(x='date', y='total_deaths', data=full_covid.query('date > "2020-03-07"'),
              hue='country', ci=None, estimator=np.sum, ax=ax)
ax.legend(bbox_to_anchor=(1, 1))
ax.set(xlabel='', ylabel='', title='Total Deaths from COVID-19');
```



70.10 Raw data plots

The last major category of plots are those that do not change the underlying data. They simply plot the raw data as it is. In this section, we'll create scatter plots with regression lines running through them, as well as heat maps using the following functions.

- `scatterplot`
- `regplot`
- `heatmap`

Scatter plots

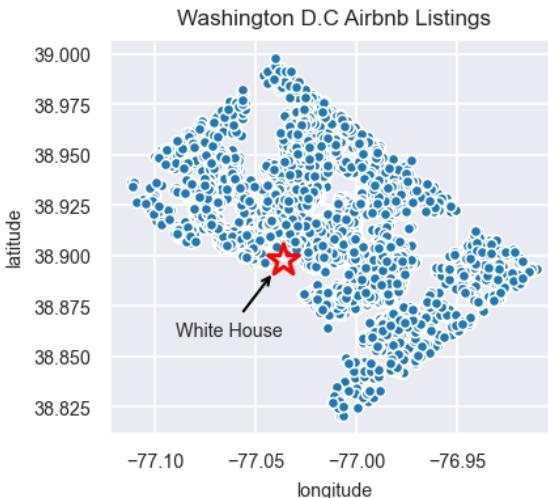
The `scatterplot` function provides many different options to size, color, and style points according to different variables. Let's begin by creating a map of each Airbnb listing using the longitude and latitude points and place a marker at the exact location of the White House. The `s` parameter sets the size of each marker in points squared.

```
[47]: wh_coords = -77.0365, 38.8977
fig, ax = plt.subplots(figsize=(3, 3))
```

```

ax.set_aspect('equal')
sns.scatterplot(x='longitude', y='latitude', data=airbnb, s=16, ax=ax)
ax.scatter(*wh_coords, marker='*', c='white', ec='red', lw=1.5, s=150)
ax.annotate('White House', xy=wh_coords, xytext=(-77.09, 38.86),
            arrowprops={'arrowstyle': '->', 'shrinkB': 7, 'color': 'black'})
ax.set_title('Washington D.C Airbnb Listings');

```



Because there are over 9,000 listings in this dataset, let's filter for listings that are within one mile of the White House. To do so exactly requires the use of the [haversine formula](#), but since this area of the world is so small and not close to the poles, we'll just use some basic trigonometry to calculate the distance. We first calculate the distance in degrees between the White House and every listing.

```
[48]: dist_degree = ((airbnb['longitude'] - wh_coords[0]) ** 2 +
                     (airbnb['latitude'] - wh_coords[1]) ** 2) ** .5
dist_degree.head(3)
```

```
[48]: 0    0.020344
      1    0.062154
      2    0.022590
      dtype: float64
```

The earth is approximately 25,000 miles in circumference so we can get the number of miles per degree by dividing by 360.

```
[49]: miles_per_degree = 25000 / 360
miles_per_degree
```

```
[49]: 69.44444444444444
```

Multiplying this number by the distance in degrees returns the number of miles from the White House for each listing.

```
[50]: airbnb['miles_from_wh'] = (dist_degree * miles_per_degree).round(2)
airbnb['miles_from_wh'].head(3)
```

```
[50]: 0    1.41
      1    4.32
```

```
2      1.57
Name: miles_from_wh, dtype: float64
```

Let's do a sanity check and get the minimum and maximum distances. Washington D.C. is a fairly small place originally carved into a perfect 10 by 10 mile square with corners rotated 45 degrees to appear diamond-shaped. The lower-left portion was returned to Virginia which accounts for its current shape and size of 68 square miles, down from it's original 100. The White House is very nearly in the center of the original square, so every listing should be no more than 10 miles from it.

```
[51]: airbnb['miles_from_wh'].agg(['min', 'max'])
```

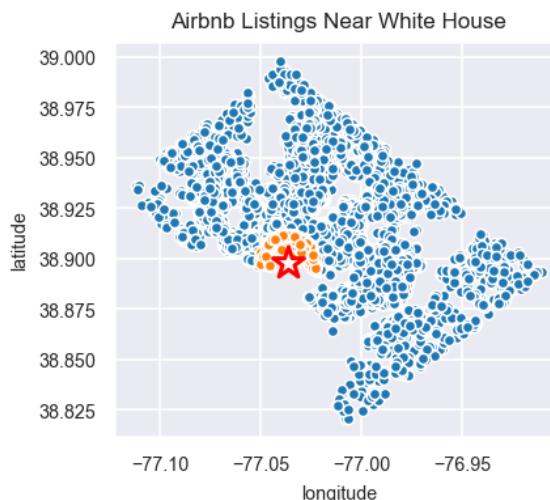
```
[51]: min    0.17
      max    8.78
      Name: miles_from_wh, dtype: float64
```

To provide more verification that our calculations are on the right track, let's recreate the plot above adding a second scatter plot for just those listings within 1 mile of the White House. Because we will be creating several similar scatter plots, a function is created to setup the figure, axes, grid, aspect, and title as well as to place the White House.

```
[52]: def setup_wh_plot(figsize=(3, 3)):
    wh_coords = -77.0365, 38.8977
    fig, ax = plt.subplots(figsize=figsize)
    ax.set_aspect('equal')
    ax.scatter(*wh_coords, marker='*', c='white', ec='red', lw=1.5, s=150, zorder=3)
    ax.set_title('Airbnb Listings Near White House')
    return ax
```

We filter for our new data, then use the above function which returns the axes, before finally plotting two separate scatter plots. seaborn automatically uses the next color in the color cycle when plotting on the same axes.

```
[53]: airbnb_wh = airbnb.query('miles_from_wh < 1')
ax = setup_wh_plot()
sns.scatterplot(x='longitude', y='latitude', data=airbnb, s=16, ax=ax)
sns.scatterplot(x='longitude', y='latitude', data=airbnb_wh, s=16, ax=ax);
```



The color, size, and marker style can correspond to specific columns by setting the `hue`, `size`, and `style`

parameters. Here, we color by neighborhood, size by price, and style by superhost. We'll zoom in on only the listings within one mile of the White House.

```
[54]: ax = setup_wh_plot(figsize=(4, 4))
sns.scatterplot(x='longitude', y='latitude', data=airbnb_wh, ax=ax,
                 hue='neighborhood', size='price', style='superhost')
ax.legend(bbox_to_anchor=(1, 1));
```

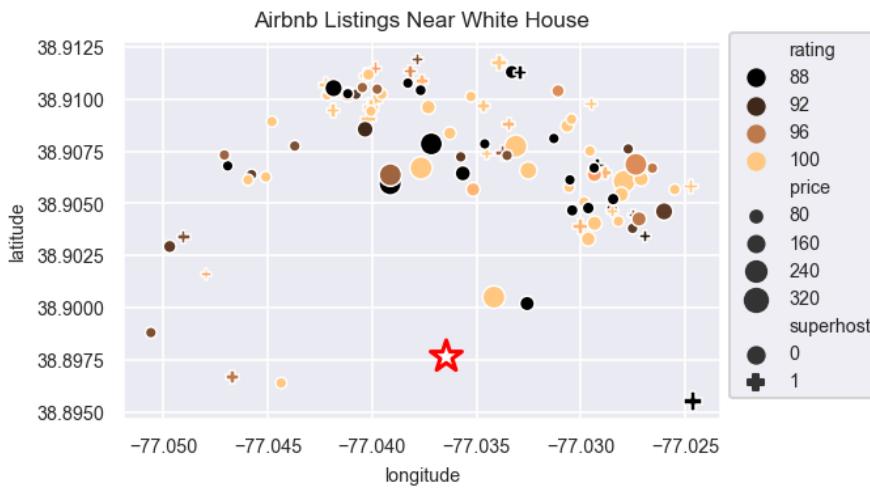


Each of the values for `hue`, `size`, and `style` can be customized. If either `hue` or `size` are numeric then you may set the `hue_norm` or `size_norm` parameter to a two-item tuple of the minimum and maximum value of the range. Values outside of the range given will be colored or sized to the minimum or maximum reference.

In the scatter plot below, we choose to color by rating, a numeric variable which has valid values between 0 and 100. But, more than 85% of ratings are greater than or equal to 90 with a full 30% getting the top rating of 100%. Using the default scale of 0 to 100 wouldn't show much difference between a rating of 90 and 100 though 90 is in the 15th percentile. Therefore, we set `hue_norm` to be `(90, 100)`. All ratings below 90 are colored the same value as 90. Use `palette` to set the matplotlib color map.

The same approach is used for size, which is controlled by price. There are some extreme outliers that force nearly every point to be the smallest size. We use `size_norm` to size based on a limited range. Finally, the marker for each unique value in the `style` column can be set by using a dictionary passed to `markers`. Since there are so many points, a random sample of 100 listings that accommodate at most two persons is selected.

```
[55]: ax = setup_wh_plot(figsize=(4, 4))
df = airbnb_wh.query('accommodates <= 2').dropna(subset=['rating']) \
    .sample(n=100, random_state=1)
sns.scatterplot(x='longitude', y='latitude', data=df, ax=ax,
                 hue='rating', hue_norm=(90, 100),
                 size='price', size_norm=(100, 300),
                 style='superhost', markers={0: 'o', 1: 'P'},
                 palette='copper')
ax.legend(bbox_to_anchor=(1, 1.05));
```



Heat maps

A heat map is a rectangular grid made of individual rectangles that each get colored based on a reference value. seaborn does not group or aggregate within its `heatmap` function and the `x` and `y` parameters do not exist. The first argument passed is the DataFrame who's values are used directly to correspond to the particular color of the chosen sequential colormap.

You'll probably need to do some data manipulation with pandas before creating a heat map as tidy data is not well-suited for it. Below, we filter listings that accommodate five or less and whose neighborhood has more than 300 total listings. We then create a pivot table of median price.

```
[56]: df = airbnb.query('accommodates <= 5') \
    .groupby('neighborhood').filter(lambda x: len(x) > 300)
median_price = df.pivot_table(index='neighborhood', columns='accommodates',
                               values='price', aggfunc='median').round(-1).astype('int64')
median_price
```

	accommodates	1	2	3	4	5
neighborhood						
Brightwood Park	50	70	80	90	140	
Capitol Hill	60	100	120	140	170	
Columbia Heights	60	80	100	100	140	
Downtown	70	140	180	170	270	
Dupont Circle	80	110	130	170	180	
Edgewood	50	80	90	120	120	
Kalorama Heights	60	100	120	130	150	
Shaw	70	110	130	160	200	
Union Station	60	90	100	130	150	
West End	70	130	1000	170	220	

Notice the outlier of 1,000 for West End listings that accommodate three persons. This one value will

dominate the heat map if we use the defaults. Instead, we set `vmax` to the second highest value to cap the range used to determine color. Setting `annot` to `True` annotates each cell with its value and is formatted with `fmt`.

```
[57]: ax = sns.heatmap(median_price, cmap='OrRd', annot=True, fmt=',.0f', vmax=270)
ax.set_title('Median Price');
```

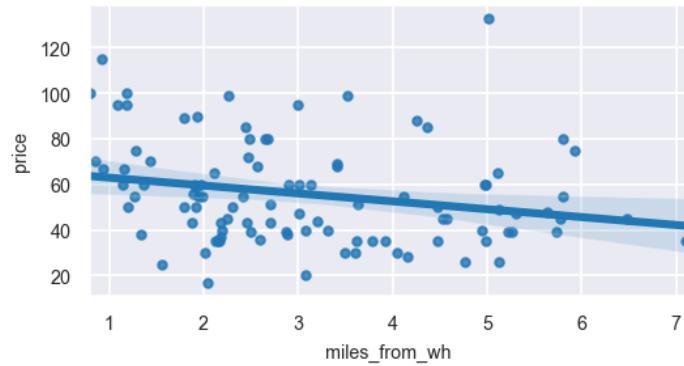


70.11 Scatter plots with linear regression lines using regplot

The `regplot` function creates scatter plots and places a linear regression line through the cloud of points. It might be interesting to investigate the relationship between distance from the White House and price. Perhaps listings closer to the White House would be in higher demand and cost more. To help test this hypothesis, we'll select listings that accommodate exactly one person. Since there are a large number of points, we'll select 100 random points.

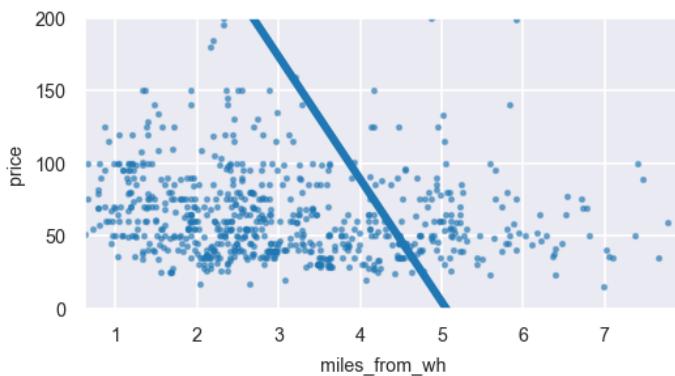
seaborn allows you to control the scatter points and line with the `scatter_kws` and `line_kws` parameters. Set them each to a dictionary mapping the underlying property to its value. We use them to set the size of each point and the width of the line.

```
[58]: df = airbnb.query('accommodates == 1 and price < 500').sample(100, random_state=1)
sns.regplot(x='miles_from_wh', y='price', data=df,
            scatter_kws={'s': 10}, line_kws={'lw': 3});
```



The line appears to have a slight downward trend suggesting that miles from the White House might have a minor effect on the price of the listing. Notice the light-blue region surrounding the line. This represents the 95% confidence interval for the regression line. Instead of using a random sample of the data, let's use all points for one-person listings.

```
[59]: df = airbnb.query('accommodates == 1')
ax = sns.regplot(x='miles_from_wh', y='price', data=df, ci=None, marker='.',
                  scatter_kws={'s': 10, 'alpha': .5}, line_kws={'lw': 3})
ax.set_ylim(0, 200);
```

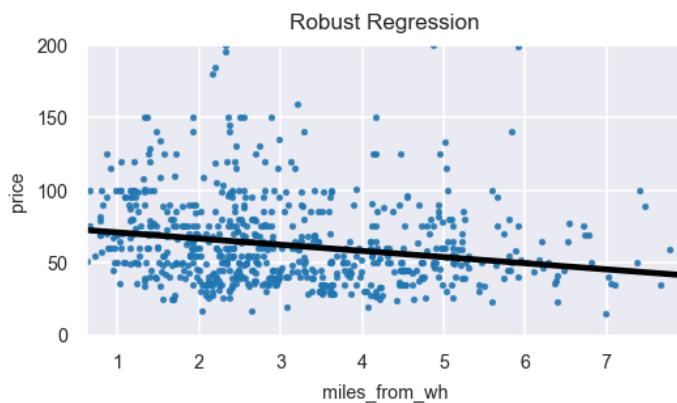


Choosing different types of linear regression

The regression line looks bizarre and is now predicting a price less than 0 for listings more than five miles from the White House. The reason for this is the presence of extreme outliers. Specifically, there are 10 listings priced at exactly \$7,000. The default algorithm used to find the formula for the line is called ordinary least squares (OLS), which attempts to minimize the sum of squared errors (distance between line and point). This metric is greatly influenced by outliers causing the regression line to be pulled far away from the central cluster of points.

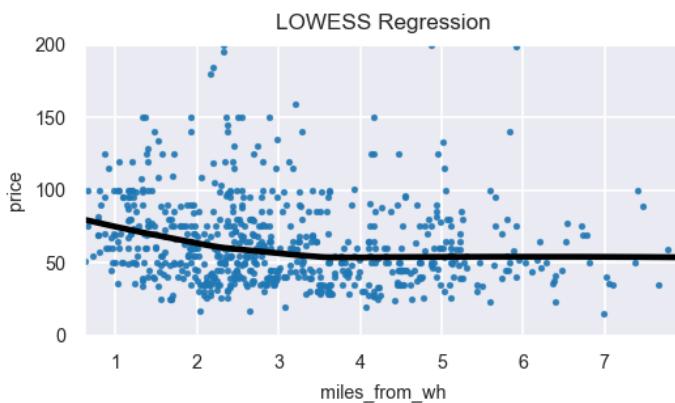
seaborn allows other algorithms besides OLS to compute the line of best fit. Set `robust` to `True` to perform robust regression, which greatly reduces the effect of outliers on the line. Robust regression is much more computationally intense, so we do not calculate a confidence interval.

```
[60]: df = airbnb.query('accommodates == 1')
ax = sns.regplot(x='miles_from_wh', y='price', data=df, marker='.',
                  ci=None, robust=True, scatter_kws={'s': 10}, line_kws={'color': 'black'})
ax.set(ylim=(0, 200), title='Robust Regression');
```



Another form of regression is LOWESS, or locally weighted sum of squares, that builds many regression models for every x-value weighing points closer to the x-value more than those far away. It's able to model highly non-linear movements as it's not constrained to a particular form. We set `lowess` to `True` and observe that the effect on price seems to only be significant for houses within three miles of the White House.

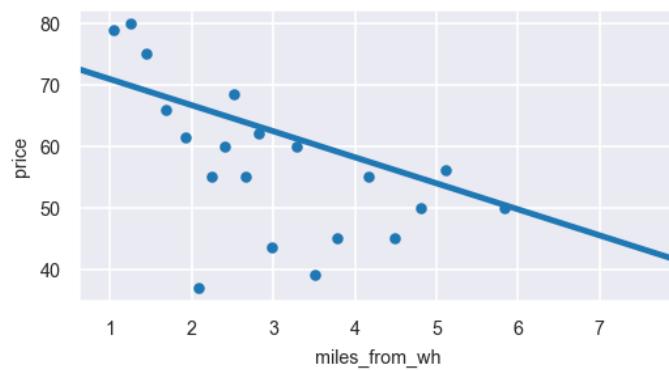
```
[61]: ax = sns.regplot(x='miles_from_wh', y='price', data=df, marker='.', ci=None,
                     lowess=True, scatter_kws={'s': 10}, line_kws={'color': 'black'});
ax.set(ylim=(0, 200), title='LOWESS Regression');
```



Grouping x and aggregating y

Instead of plotting each and every point, you can group the data into bins based on the x-values and aggregate the y-values. Choose evenly-sized bins by setting the `x_bins` parameter to an integer. Alternatively, set it to an array whose values will be the midpoint of the bins. Set `x_estimator` to the numpy aggregation function you want to use and `x_ci` for the confidence interval. The regression is calculated from the original data and NOT from these aggregated values. Here, we cut the data into 20 evenly-sized bins calculating the median price of each. The same robust regression line from above is plotted.

```
[62]: sns.regplot(x='miles_from_wh', y='price', x_bins=20, x_estimator=np.median, x_ci=None,
                 robust=True, ci=None, data=df, scatter_kws={'s': 10});
```



70.12 Ordered categorical data

The order of the grouping variable as it appears on your plot depends on its type. seaborn uses the natural ordering for numeric variables and the order of appearance for object/string variables. If you use an ordered categorical variable, then it will use that order. Let's read in three columns of the housing dataset to show how this works.

```
[63]: cols = ['OverallQual', 'HeatingQC', 'SalePrice']
housing = pd.read_csv('../data/housing.csv', usecols=cols)
housing.head(3)
```

	OverallQual	HeatingQC	SalePrice
0	7	Ex	208500
1	6	Ex	181500
2	7	Ex	223500

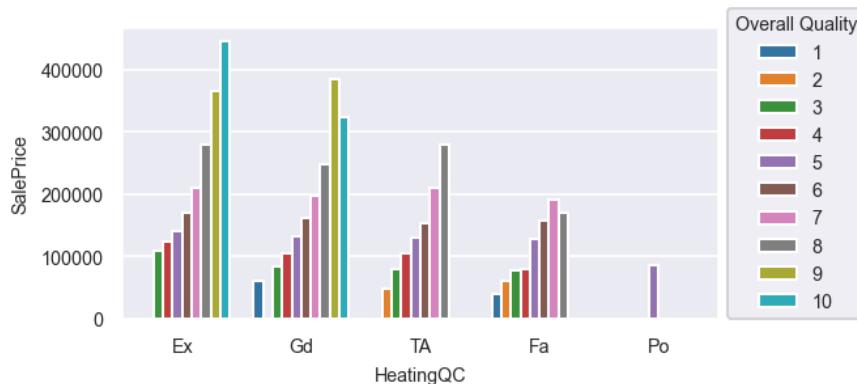
If we were to make a plot using 'HeatingQC' as a grouping variable, then seaborn would use the order of appearance of each unique value. We can find the first instance of each value with `drop_duplicates`.

```
[64]: housing['HeatingQC'].drop_duplicates()
```

```
[64]: 0      Ex
3      Gd
12     TA
29     Fa
325    Po
Name: HeatingQC, dtype: object
```

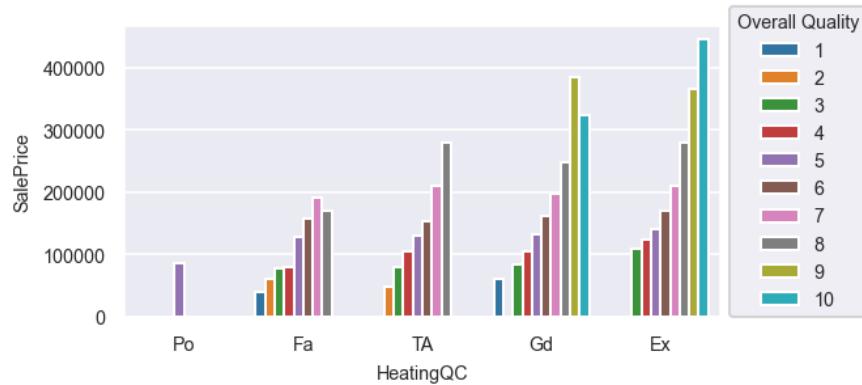
When we plot the average sale price by heating quality, the group follows the order from above. We also group by overall quality, but since this is a numeric variable, seaborn wisely chooses to use its natural ordering and not its order of appearance.

```
[65]: ax = sns.barplot(x='HeatingQC', y='SalePrice', hue='OverallQual', data=housing,
                     estimator=np.mean, ci=None)
ax.legend(bbox_to_anchor=(1, 1.1), title='Overall Quality');
```



For columns like heating quality that have an inherent natural ordering, it's best to use that ordering when plotting. We can do so by setting the `hue_order` parameter, but it's better to change its data type to ordered categorical in pandas first which seaborn will respect when plotting. We overwrite the 'HeatingQC' with its new data type and make the same plot.

```
[66]: hc_cat = pd.CategoricalDtype(['Po', 'Fa', 'TA', 'Gd', 'Ex'], ordered=True)
housing['HeatingQC'] = housing['HeatingQC'].astype(hc_cat)
ax = sns.barplot(x='HeatingQC', y='SalePrice', hue='OverallQual', data=housing,
                  estimator=np.mean, ci=None)
ax.legend(bbox_to_anchor=(1, 1.1), title='Overall Quality');
```



70.13 Exercises

[]:

Chapter 71

Seaborn Grid Plots

In this chapter, we continue our coverage of seaborn by focusing on plotting functions that return ‘grids’. These are seaborn objects that wrap matplotlib’s figure and usually contain more than one axes. There are a few different kinds of grids; some plot each group on a different axes, while others enhance the main plot on the axes.

71.1 Grids by categories

Many grid plotting functions split the data by unique category values. Take a look at the [Categorical section](#) of the API. The very top function, `catplot`, is the only grid plot in that group. The others are axes plots. The `catplot` function is capable of making each of the other plots in its section. It can be thought of as a ‘metaplottting’ function, as it just uses the other underlying plots in its section. Let’s start with `catplot` and read in our Airbnb listings data setting the style to `whitegrid`.

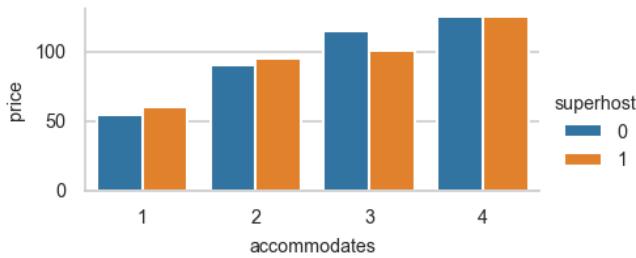
```
[1]: import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
plt.style.use('.../.../mdap.mplstyle')
sns.set_style('whitegrid')
airbnb = pd.read_csv('.../data/airbnb.csv')
airbnb.head(3)
```

	id	neighborhood	property_type	accommodates	bathrooms	...	response_time	minimum_nights	maximum_nights	latitude	longitude
0	3362	Shaw	Townhouse	16	3.5	...	within an hour	2	365	38.90982	-77.02016
1	3663	Brightwood Park	Townhouse	4	3.5	...	Nan	3	30	38.95888	-77.02554
2	3670	Howard University	Townhouse	2	1.0	...	Nan	2	30	38.91842	-77.02750

Set the `kind` parameter

The `kind` parameter of `catplot` must be set to one of the following options: ‘point’, ‘bar’, ‘strip’, ‘swarm’, ‘box’, ‘violin’, or ‘boxen’. Let’s create a bar plot of the median price by number of persons each listing accommodates of those that accommodate four or less further splitting by superhost. The following creates a simple seaborn grid with a single axes, which we assign to a variable.

```
[2]: grid = sns.catplot(kind='bar', x='accommodates', y='price', hue='superhost',
    ↪data=airbnb,
    ↪order=[1, 2, 3, 4], estimator=np.median, ci=None, height=1.5, ↪
    ↪aspect=2)
```



Let's verify that a seaborn grid was returned.

```
[3]: type(grid)
```

```
[3]: seaborn.axisgrid.FacetGrid
```

Access the underlying matplotlib figure with the `fig` attribute.

```
[4]: fig = grid.fig
type(fig)
```

```
[4]: matplotlib.figure.Figure
```

Return all axes in the figure as a list with the `axes` attribute.

```
[5]: fig.axes
```

```
[5]: [<matplotlib.axes._subplots.AxesSubplot at 0x1157b4710>]
```

We can change properties of this axes and then output the figure again.

```
[6]: fig.axes[0].set(xlabel='', ylabel='', title='Median Price by Number Accommodates')
fig
```



Control figure size with `height` and `aspect`

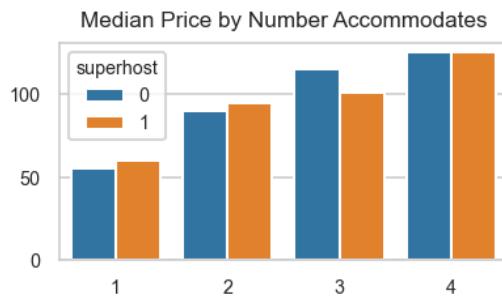
The parameters `height` and `aspect` are used to control the height and width of each axes and ultimately, the entire figure. By default, `height` is set to 5 inches. The `aspect` controls the ratio of width to height and is set to 1 by default. Multiplying the height by the aspect ratio gets the width of each axes (5 x 5 by default). Above, the height was set to 1.5 inches and aspect to 2 which made the width 3 inches. Let's verify the dimensions. The legend is responsible for adding a little extra width.

```
[7]: fig.get_size_inches()
```

```
[7]: array([3.50464853, 1.5])
```

The above plot can be duplicated directly with the `barplot` axes function.

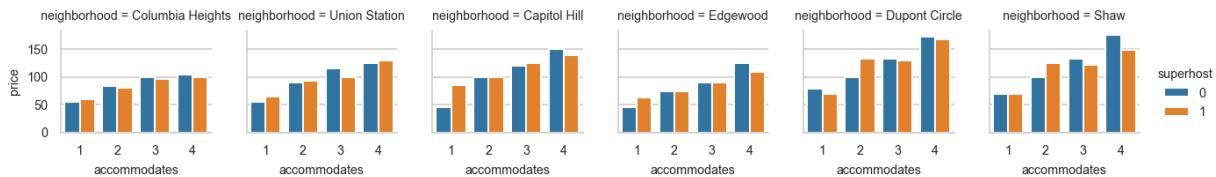
```
[8]: fig, ax = plt.subplots(figsize=(3, 1.5))
sns.barplot(x='accommodates', y='price', data=airbnb, hue='superhost',
             order=[1, 2, 3, 4], estimator=np.median, ci=None, ax=ax)
ax.set(xlabel='', ylabel='', title='Median Price by Number Accommodates');
```



Creating multiple axes with `row` and `col`

The main reason to use `catplot` is to split your data into multiple axes using the `row` and/or `col` parameters. These should be set to a categorical column name. We set `col` to neighborhood which would create a new axes for each neighborhood. But, we limit it to just the most frequent six neighborhoods with `col_order`.

```
[9]: top6 = airbnb['neighborhood'].value_counts().index[:6]
grid = sns.catplot(kind='bar', x='accommodates', y='price', data=airbnb,
                    order=[1, 2, 3, 4], hue='superhost', col='neighborhood',
                    col_order=top6, estimator=np.median, ci=None, height=1.5)
```



We have six axes each with a height and width of 1.5, so the entire figure size should be around 9 x 1.5 plus some extra width for the legend.

```
[10]: grid.fig.get_size_inches()
```

```
[10]: array([9.50464853, 1.5])
```

By default, all axes will be placed on a single row. Change this by setting the maximum number of axes per row with `col_wrap`. The seaborn `grid` object has a number of helper methods to set multiple properties at once. Here, we pass `set_titles` a string template to display just the neighborhood name. We also add a title to the entire figure by accessing the figure and calling its `suptitle` method.

```
[11]: grid = sns.catplot(kind='bar', x='accommodates', y='price', data=airbnb,
                      order=[1, 2, 3, 4], hue='superhost', col='neighborhood',
                      col_order=top6, col_wrap=3, estimator=np.median, ci=None,
```

```
height=1.5, aspect=1.2)
grid.set_titles('{col_name}')
grid.fig.suptitle('Washington D.C. Airbnb Listings by Neighborhood', y=1.03);
```



We made each plot a bit wider by changing the aspect ratio to 1.2. Now that there are two rows, the total height of the figure has doubled from 1.5 to 3. Let's verify these changes in the figure size.

```
[12]: grid.fig.get_size_inches()
```

```
[12]: array([5.90464853, 3.])
```

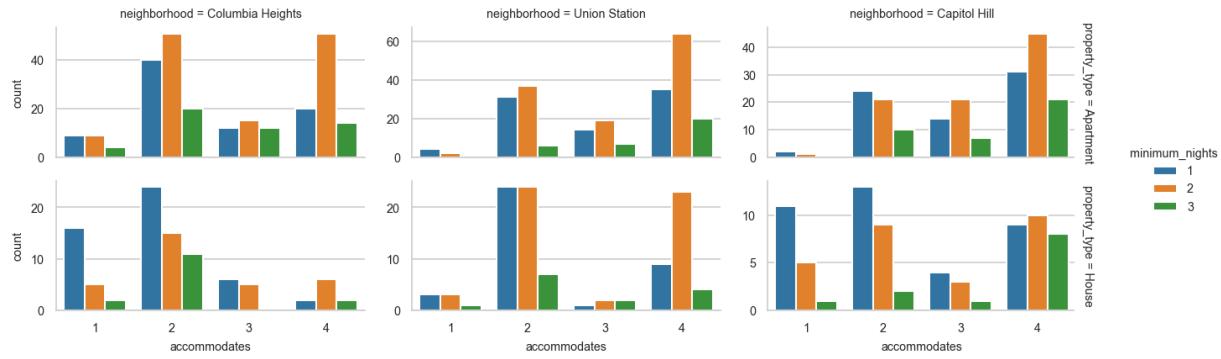
It's possible to set both `row` and `col` simultaneously to different categorical variables. The size of the grid is determined automatically from the number of unique values in each. Let's use `catplot` to create a separate axes for each unique combination of neighborhood and property type counting the number of listings by persons accommodated and minimum nights. First, we get the two most common property types.

```
[13]: properties = airbnb['property_type'].value_counts().index[:2]
properties
```

```
[13]: Index(['Apartment', 'House'], dtype='object')
```

We substantially cut down on the number of axes and bars by using the parameters that end in `order`. By default, all x and y axis limits are shared. The `sharex` and `sharey` boolean parameters can be set to `False` so that each axes sets its own limits. Here, each axes sets its own y-axis limits, though this isn't usually good practice as it makes for more difficult comparisons. Setting `margin_titles` to `True` places the titles of each row along the top and right side of the grid instead of on every axes.

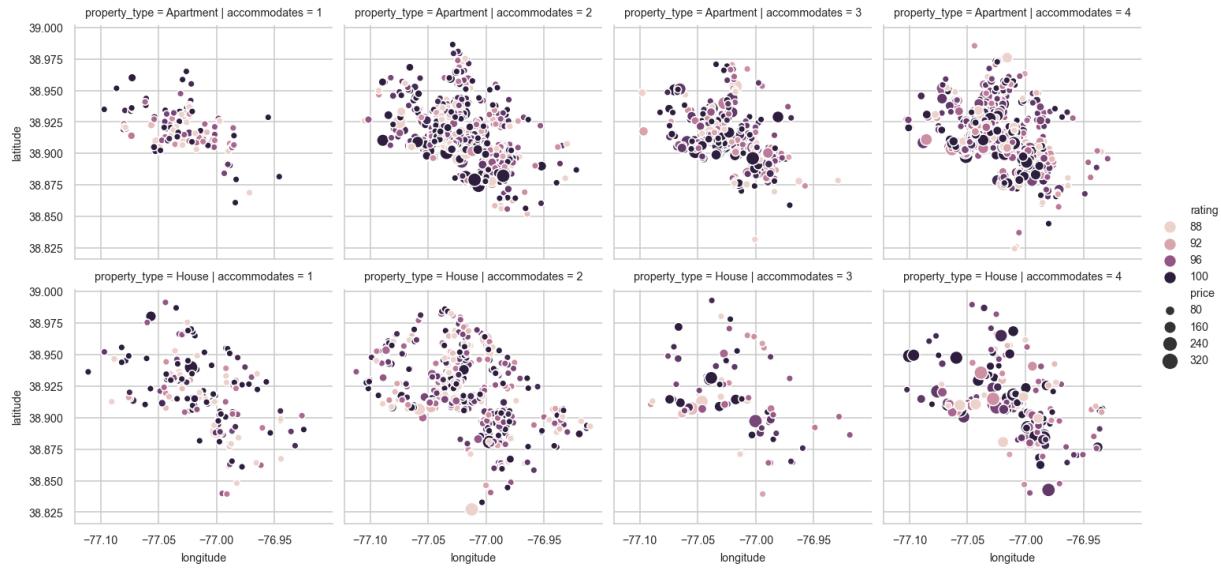
```
[14]: sns.catplot(kind='count', x='accommodates', data=airbnb,
                 order=[1, 2, 3, 4], hue='minimum_nights', hue_order=[1, 2, 3],
                 col='neighborhood', col_order=top6[:3],
                 row='property_type', row_order=properties,
                 height=1.5, aspect=2, sharey=False, margin_titles=True);
```



71.2 Scatter and line plot grids

The `relplot` function creates scatter and line plots within a grid in a very similar fashion as `catplot`. Choose the underlying plot by setting `kind` to either '`scatter`' or '`line`'. Use the `row` and `col` parameters to split the data into separate axes. Here, we plot the location of every listing by property type and persons it accommodates. The size is controlled by price and color by rating.

```
[15]: sns.relplot(x='longitude', y='latitude', kind='scatter', data=airbnb,
                 size='price', size_norm=(100, 300),
                 hue='rating', hue_norm=(90, 100),
                 col='accommodates', col_order=[1,2,3, 4],
                 row='property_type', row_order=properties, height=2.5);
```



Line plots with `relplot`

Let's make a single line plot on each axes showing the total number of COVID-19 deaths for the first 10 days after the 50th death for each country.

```
[16]: full_covid = pd.read_csv('../data/WHO/full_covid_data.csv', parse_dates=['date'])
full_covid.tail(3)
```

	date	country	new_cases	new_deaths	total_cases	total_deaths
7317	2020-03-26	Zimbabwe	1	0	3	1
7318	2020-03-27	Zimbabwe	0	0	3	1
7319	2020-03-28	Zimbabwe	2	0	5	1

We begin by selecting the countries with 10 or more days of over 50 deaths.

```
[17]: df = full_covid.query('total_deaths >= 50')
df = df.groupby('country').filter(lambda x: len(x) >= 10)
df.tail(3)
```

	date	country	new_cases	new_deaths	total_cases	total_deaths
7154	2020-03-26	United States	13963	249	69194	1050
7155	2020-03-27	United States	16797	246	85991	1296
7156	2020-03-28	United States	18695	411	104686	1707

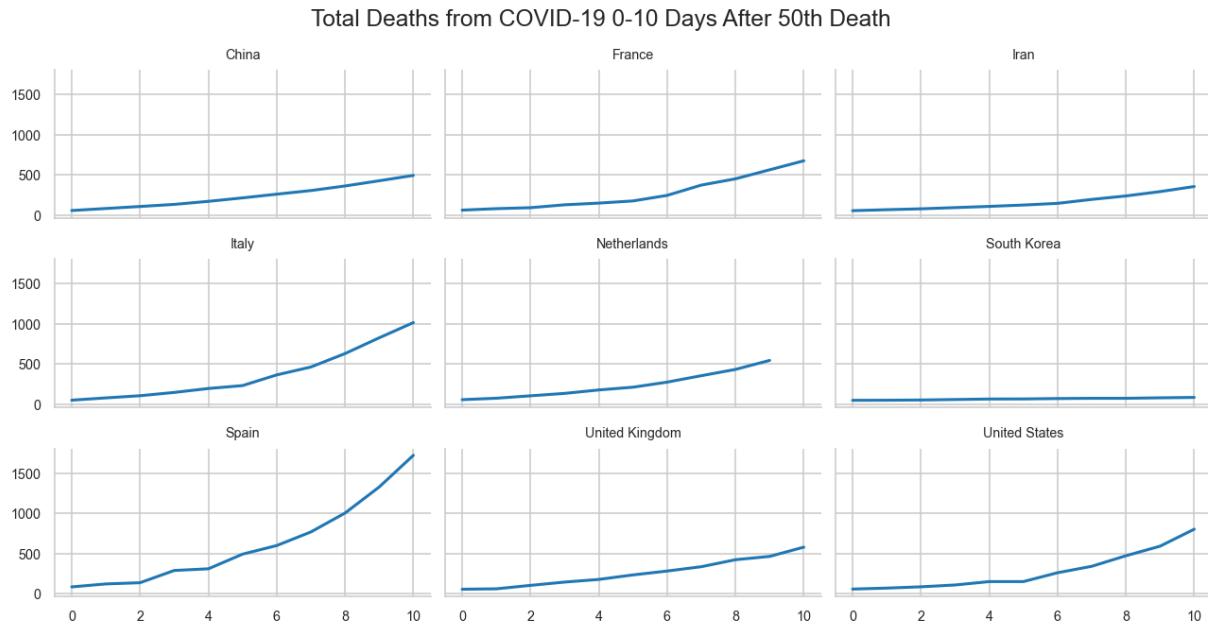
We then add a column that tracks the number of days after the 50th death for each country and filter for the first 10 days.

```
[18]: df['days_after_50'] = df.groupby('country').cumcount()
df = df.query('days_after_50 <= 10')
df.tail(3)
```

	date	country	new_cases	new_deaths	total_cases	total_deaths	days_after_50
7151	2020-03-23	United States	8459	131	35206	471	8
7152	2020-03-24	United States	11236	119	46442	590	9
7153	2020-03-25	United States	8789	211	55231	801	10

We can now use `relplot` to make a line plot of each country on its own axes by setting the `col` parameter. The aggregation function is set to `sum`, but since there's only one number per country per date, it doesn't really do anything.

```
[19]: grid = sns.relplot(x='days_after_50', y='total_deaths', data=df, kind='line',
                       col='country', col_wrap=3, estimator=np.sum, ci=None, height=1.5,
                       aspect=2)
grid.set_titles('{col_name}')
grid.set_ylabels('')
grid.set_xlabels('')
grid.fig.suptitle('Total Deaths from COVID-19 0-10 Days After 50th Death', y=1.04,
                  fontsize=12);
```



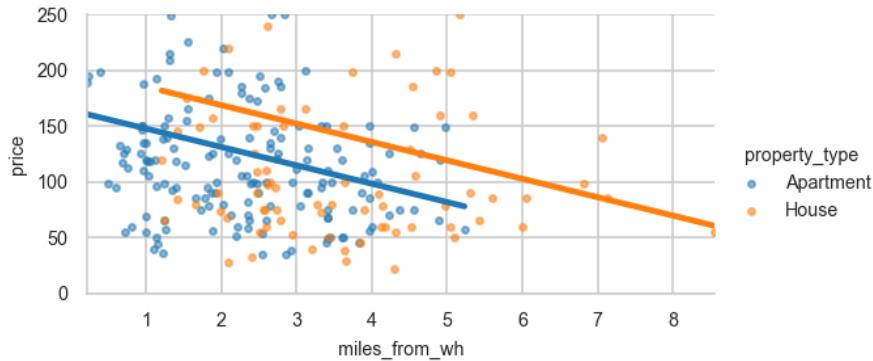
71.3 Regression grid plots

Following in the footsteps of `catplot` and `relplot` is `lmplot`, which makes grids of regression plots by using the `row` and `col` parameters. It essentially allows you run `regplot` on multiple different axes. Let's add a column for the miles from the White House to our Airbnb data again to test the relationship between it and price.

```
[20]: wh_coords = -77.0365, 38.8977
dist_degree = ((airbnb['longitude'] - wh_coords[0]) ** 2 +
                (airbnb['latitude'] - wh_coords[1]) ** 2) ** .5
miles_per_degree = 25000 / 360
airbnb['miles_from_wh'] = (dist_degree * miles_per_degree).round(2)
```

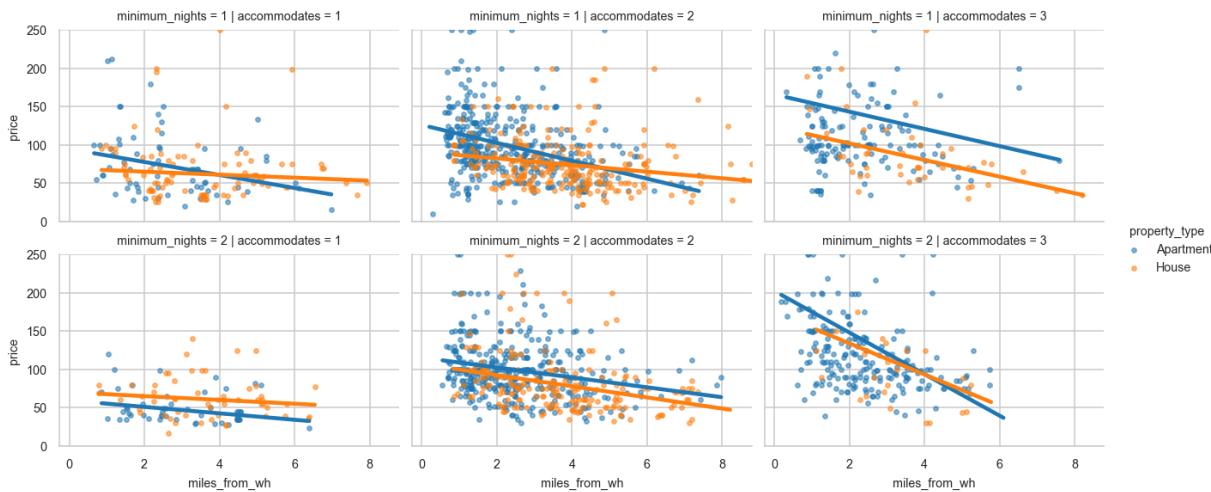
The `lmplot` function has an option for `hue`, which is not available to `regplot`. We'll use it to build two separate regression lines - one for two different property types. A small subset of the data is used because of the large number of points.

```
[21]: df = airbnb.sample(n=400, random_state=1)
grid = sns.lmplot(x='miles_from_wh', y='price', data=df, hue='property_type', ▾
                   hue_order=properties,
                   robust=True, ci=None, height=2, aspect=2, scatter_kws={'s': 6, ▾
                   'alpha': .5})
grid.set(ylim=(0, 250));
```



Here, we use the full power of `lmplot` and split the data by persons accommodated and minimum nights.

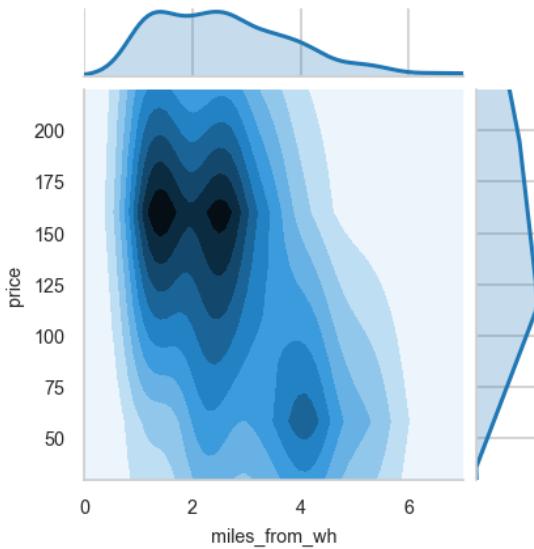
```
[22]: grid = sns.lmplot(x='miles_from_wh', y='price', data=airbnb,
                      hue='property_type', hue_order=properties,
                      col='accommodates', col_order=[1, 2, 3],
                      row='minimum_nights', row_order=[1, 2],
                      robust=True, ci=None, height=2, aspect=1.5,
                      scatter_kws={'s': 6, 'alpha': .5})
grid.set(ylim=(0, 250));
```



71.4 Bivariate distributions grids

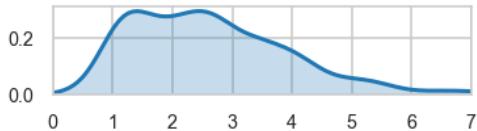
The `jointplot` function creates a grid with three matplotlib axes. The main axes contains the joint distribution of two numeric variables as either a scatter plot, KDE, or hexbin chosen with `kind`. Above and to the right of the main axes are univariate distributions of each variable as either a histogram or KDE. Here, we construct a bivariate KDE of miles from the White House and price for listings that accommodate four persons.

```
[23]: df = airbnb.query('accommodates == 4')
grid = sns.jointplot(x='miles_from_wh', y='price', data=df, height=3,
                     kind='kde', xlim=(0, 7), ylim=(30, 220), shade=True)
```



Above the plot lies the univariate KDE for miles from the White House. The univariate KDE for price is to the right. Individually, these are called the marginal distributions. It is easy to replicate the marginal KDEs to verify them. Here, we find the KDE of miles from the White House.

```
[24]: fig, ax = plt.subplots(figsize=(2.8, .6))
sns.kdeplot(df['miles_from_wh'], ax=ax, shade=True, legend=False)
ax.set_xlim(0, 7);
```



Each plot is stored on its own axes. Let's verify that there are three axes.

```
[25]: grid.fig.axes
```

```
[25]: [<matplotlib.axes._subplots.AxesSubplot at 0x1174f2b50>,
<matplotlib.axes._subplots.AxesSubplot at 0x117563490>,
<matplotlib.axes._subplots.AxesSubplot at 0x117acb710>]
```

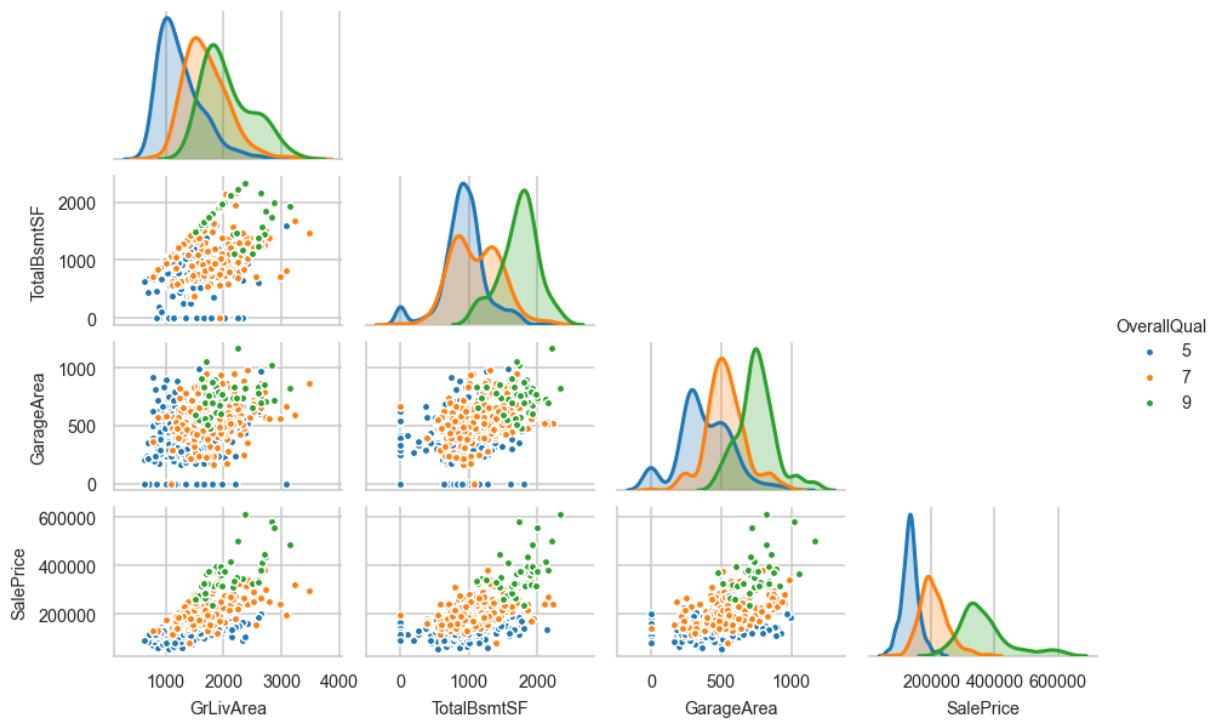
71.5 Scatter plot grids of multiple column combinations

The `pairplot` function allows you to plot any number of combinations of numeric variables as a scatter plot. Select the columns with the `vars` parameter. Here, each pair of combinations of four columns of the housing dataset are plotted as scatter plots and colored by three values of overall quality using `hue`.

By default, a square grid is created with each pair plotted twice, once as each x and y variable. This redundancy can be eliminated by setting `corner` to `True`. The diagonal plots are different as scatter plots with the same x and y variable would be pointless. A univariate histogram or KDE is produced instead and chosen with `diag_kind`.

```
[26]: housing = pd.read_csv('../data/housing.csv')
cols = ['GrLivArea', 'TotalBsmtSF', 'GarageArea', 'SalePrice']
sns.pairplot(housing, vars=cols, hue='OverallQual', corner=True,
```

```
kind='scatter', diag_kind='kde',
hue_order=[5, 7, 9], height=1, aspect=1.5, plot_kws={'s': 10});
```



71.6 Hierarchical cluster map

The `clustermap` function is one of the most powerful and interesting options seaborn has available. It finds similar rows and columns in your DataFrame using hierarchical clustering and creates a heat map to display the results. One use case is when you are exploring similarities between columns. The meetup dataset is used here, which contains 30 different groups from Meetup.com in the Houston area. Each row represents a single person's membership. Let's read in the data and observe the first few members and groups.

```
[27]: meetup = pd.read_csv('../data/meetup.csv', index_col='member_id')
meetup.iloc[:3, :5]
```

	Houston Arts + Culture	PyHou - Houston Python Enthusiasts!	TIBCO Houston User Group	Houston Beer Lovers- Let's discover beer around Houston	Houston Energy Data Science Meetup
member_id					
16691721	0	1	0	0	1
230046555	0	0	0	0	1
186076150	0	0	0	0	1

Let's say we are interested in finding the similarity between groups based on the fraction of members that are in-common between both. We can find the correlation coefficient between each column by calling the `corr` method. The correlation between the first five groups is displayed below.

```
[28]: meetup_corr = meetup.corr()
meetup_corr.iloc[:5, :5].round(3)
```

	Houston Arts + Culture	PyHou - Houston Python Enthusiasts!	TIBCO Houston User Group	Houston Beer Lovers- Let's discover beer around Houston	Houston Energy Data Science Meetup
Houston Arts + Culture	1.000	-0.086	0.004	0.236	-0.177
PyHou - Houston Python Enthusiasts!	-0.086	1.000	-0.053	-0.091	0.126
TIBCO Houston User Group	0.004	-0.053	1.000	0.011	0.029
Houston Beer Lovers- Let's discover beer around Houston	0.236	-0.091	0.011	1.000	-0.122
Houston Energy Data Science Meetup	-0.177	0.126	0.029	-0.122	1.000

We can then find all groups that are most similar to one another by sorting each column's values. Let's find the groups most similar to the Houston Energy Data Science Meetup. As we would expect, these are all data science groups.

```
[29]: meetup_corr['Houston Energy Data Science Meetup'].sort_values(ascending=False).head()
```

```
[29]: Houston Energy Data Science Meetup      1.000000
Houston Machine Learning                  0.373626
Houston R Users Group                    0.344886
Big Data Houston                         0.315321
Open Source Data Science                 0.132448
Name: Houston Energy Data Science Meetup, dtype: float64
```

The `clustermap` displays all of this data as a heat map. The maximum correlation value of 1, achieved only when computing the correlation of one group to itself will severely skew the distribution of color that the heat map uses. Let's get the highest correlation that is not 1 and assign it to `vmax`.

```
[30]: vmax = meetup_corr.replace(1, 0).max().max()
vmax
```

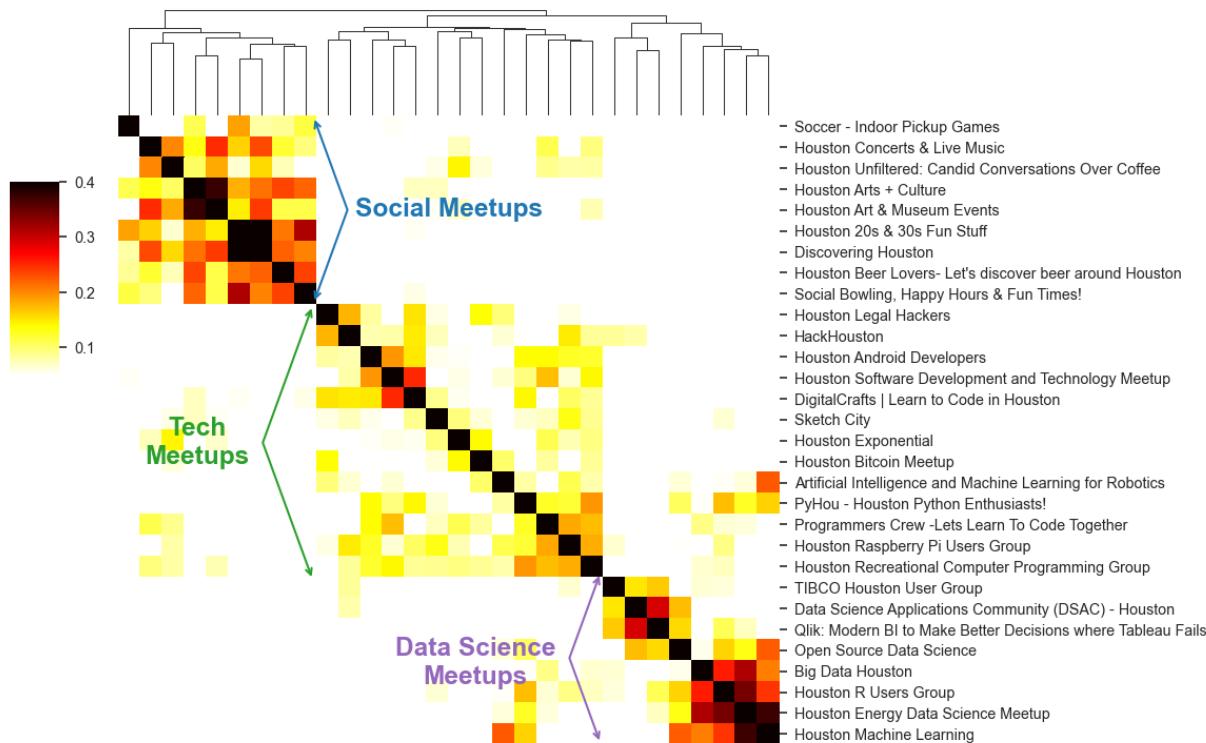
```
[30]: 0.4001769391824954
```

We now pass the pairwise correlation matrix between each group into the `clustermap` function. It rearranges the column order so that columns in the same cluster are next to one another in the plot. It is possible that groups are negatively correlated with one another, but we are interested in finding those similar to one another. Because of this, we set the color for any correlation below .05 to the first color of the colormap, which is white for `hot_r`. We also set the last color to be mapped to the maximum correlation found above.

```
[31]: from matplotlib import cm
grid = sns.clustermap(meetup_corr, vmin=.05, vmax=vmax, cmap='hot_r', figsize=(8, 5),
                      cbar_pos=(0.02, 0.5, 0.04, 0.25), xticklabels=[], 
                      yticklabels=True,
```

```
dendrogram_ratio=.15)

ax = grid.ax_heatmap
arrowprops = {'arrowstyle': '<->', 'connectionstyle': 'angle,angleA=110,angleB=70'}
annot_kwargs = {'s': '', 'xycoords': ax.transAxes, 'textcoords': ax.transAxes,
                'arrowprops': arrowprops}
text_kwargs = {'transform': ax.transAxes, 'size': 12, 'weight': 'bold',
               'ha': 'center', 'va': 'center'}
colors = cm.tab10([0, 2, 4])
arrowprops['color'] = colors[0]
ax.annotate(xy=(.295, .7), xytext=(.295, 1), **annot_kwargs)
arrowprops['color'] = colors[1]
ax.annotate(xy=(.295, .7), xytext=(.295, .26), **annot_kwargs)
arrowprops['color'] = colors[2]
ax.annotate(xy=(.73, .275), xytext=(.73, 0), **annot_kwargs)
ax.text(x=.5, y=.85, s='Social Meetups', **text_kwargs, color=colors[0])
ax.text(x=.12, y=.48, s='Tech\\nMeetups', **text_kwargs, color=colors[1])
ax.text(x=.54, y=.13, s='Data Science\\nMeetups', **text_kwargs, color=colors[2])
grid.ax_row_dendrogram.set_visible(False)
```



A sophisticated procedure called hierarchical clustering is used to find similar meetup groups. There is no definitive number of clusters returned with hierarchical clustering. It's up to you to decide what constitutes a cluster. Both the heat map and the dendrogram (tree diagram) can be used to determine the number of clusters. The heat map is usually a better source to see clusters and they will be visible as square regions where most of the columns are all correlated relatively high with one another. I chose three broad categories for clusters and labeled them by looking at the names of the meetup groups. This selection was done arbitrarily.

Four axes are returned - one for the heatmap and color bar and one for each dendrogram. Normally you would see a dendrogram to the left of the heatmap and x-axis tick labels below, but these were removed since they are duplicates to the dendrogram above and the y-axis tick labels to the right.