Master Data Analysis with Python: Volume 1 - Parts 1 & 2

by
Ted Petrou

Contents

Part I

Intro to pandas

Chapter 1

What is pandas?

1.1 Overview

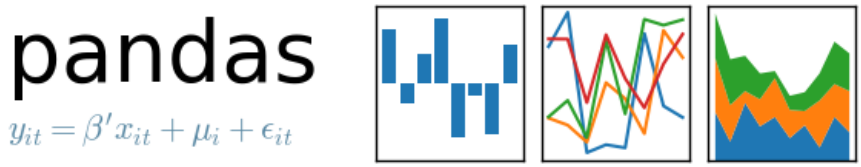
1.1.1 Objectives

- Know why pandas is suitable for data analysis in Python
- Identify a DataFrame as

1.1.2 Resources

- Official Documentation
- Package Overview
- Intro to Data Structures

1.2 Welcome to ....



1.2.1 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 meaning data that is two-dimensional with rows and columns; i.e. a table.

1.2.2 Why pandas and not xyz?

In this current age of data explosion, there are now many 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 very quickly.

1.2.3 Why is it named after an East Asian bear?

The pandas library was begun by Wes McKinney beginning in 2008 at a hedge fund named AQR. Finance speak is to call tabular data 'panel data' which smashed together becomes pandas. If you are really interested in the history, you can hear it from the creator himself.

1.2.4 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 structures lists and dictionaries, do not easily lend themselves to tabular data analysis in ways that humans can operate on them.

1.2.5 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.

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.

1.2.6 numpy tutorial in appendix

Although it is not necessary to understand numpy to perform data analysis with pandas, it is a major piece of the data science ecosystem in Python and it can be used alongside pandas. A thorough numpy tutorial is available in Appendix A.

1.3 pandas operates on tabular (table) data

There are numerous formats for data such as XML, JSON, raw bytes, and many others. But, for our purposes, we will only be examining what most people think of when they think of data - a table. pandas is built just for analyzing this tabular, rectangular, very deceptively normal concept of data. pandas has the capability to read in many different formats of data, but they all will be converted to tabular data.

1.3.1 The DataFrame and Series

The DataFrame and Series are the two primary pandas objects that we will be using throughout this course.

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

1.4 pandas examples

The rest of this chapter is dedicated to showing examples of what pandas is capable of doing. There will be 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 will provide just enough information so that you can logically follow what the end result is.

1.4.1 The `head` method

You will notice that many of the last lines of code end with the `head` method. This returns, by default, the first five rows. This helps keep the output compact.

1.5 Reading data

There will be multiple datasets used during the rest of this chapter. pandas can read in a variety of different data formats. 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.

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

Out[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

pandas stores its data in a **DataFrame** which is the topic of the next chapter.

1.6 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 5000 (seconds). This example is a single condition that gets tested for each row. Only the rows that meet this condition are returned.

1.6.1 Single Condition

```
In [2]: filt = bikes['tripduration'] > 5000
        bikes[filt].head()
```

Out[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299
335	323274	Subscriber	Male	2013-08-25 17:20:00	2013-08-25 19:26:00	7533
504	442585	Subscriber	Male	2013-09-08 03:43:00	2013-09-08 07:20:00	13037

1.6.2 Multiple Conditions

We can test for multiple conditions in a single row. The following example only returns riders that are female **and** have a `tripduration` greater than 5000.

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

Out[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299
1954	1103416	Subscriber	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050
2712	1416670	Subscriber	Female	2014-04-15 15:56:00	2014-04-16 14:09:00	79988
2915	1499497	Subscriber	Female	2014-04-25 13:01:00	2014-04-25 16:27:00	12351

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 5000.

```
In [4]: filt = filt1 | filt2
        bikes[filt].head()
```

Out[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
9	23558	Subscriber	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922
14	31121	Subscriber	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
20	42488	Subscriber	Female	2013-07-09 17:39:00	2013-07-09 17:55:00	943
21	42818	Subscriber	Female	2013-07-09 19:26:00	2013-07-09 19:38:00	726

1.7 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, min, and mix are all examples of aggregation functions. By default, calling these methods on a pandas DataFrame will apply the aggregation to each column. Below, we are using a dataset containing the percentage of undergraduate races for all US colleges.

```
In [5]: college = pd.read_csv('../data/college.csv', index_col='instnm')
        cr = college.loc[:, 'ugds_white':'ugds_unkn']
        cr.head()
```

Out[5]:

instnm	ugds_white	ugds_black	ugds_hisp	ugds_asian	ugds_aian
Alabama A & M University	0.0333	0.9353	0.0055	0.0019	0.0024
University of Alabama at Birmingham	0.5922	0.2600	0.0283	0.0518	0.0022
Amridge University	0.2990	0.4192	0.0069	0.0034	0.0000
University of Alabama in Huntsville	0.6988	0.1255	0.0382	0.0376	0.0143
Alabama State University	0.0158	0.9208	0.0121	0.0019	0.0010

The `mean` method takes the mean of each column.

```
In [6]: cr.mean()
```

Out[6]:

ugds_white	0.510207
ugds_black	0.189997
ugds_hisp	0.161635
ugds_asian	0.033544
ugds_aian	0.013813
ugds_nhpi	0.004569
ugds_2mor	0.023950
ugds_nra	0.016086
ugds_unkn	0.045181

```
dtype: float64
```

pandas allows you to aggregate rows as well. You must use the `axis` parameter to change the direction of the aggregation.

```
In [7]: cr.sum(axis=1).head()
```

Out[7]:

instnm	1.0000
Alabama A & M University	0.9999
University of Alabama at Birmingham	1.0000
Amridge University	1.0000
University of Alabama in Huntsville	1.0000

```
dtype: float64
```

1.8 Non-aggregating methods

There are methods that perform some calculation on the DataFrame that do not aggregate the data. They preserve the shape of the DataFrame. For example, the `round` method will round each number to a particular decimal place.

```
In [8]: cr.round(2).head()
```

Out[8]:

instnm	ugds_white	ugds_black	ugds_hisp	ugds_asian	ugds_aian
Alabama A & M University	0.03	0.94	0.01	0.00	0.00
University of Alabama at Birmingham	0.59	0.26	0.03	0.05	0.00
Amridge University	0.30	0.42	0.01	0.00	0.00
University of Alabama in Huntsville	0.70	0.13	0.04	0.04	0.01
Alabama State University	0.02	0.92	0.01	0.00	0.00

1.9 Aggregating within groups

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

```
In [9]: ins = pd.read_csv('../data/insurance.csv')
        ins.head()
```

Out[9]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

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

1.9.1 Count frequency of one column

```
In [10]: ins['region'].value_counts()
```

Out[10]:

southeast	364
northwest	325
southwest	325
northeast	324

```
Name: region, dtype: int64
```

Let's say we wish to find the mean changes for each of the unique values found in the `sex` column. The `groupby` method gives us this functionality.

```
In [11]: ins.groupby('sex').agg({'charges': 'mean'}).round(-3)
```

Out[11]:

	charges
sex	
female	13000.0
male	14000.0

1.9.2 Multiple aggregation functions

pandas allows us to perform multiple aggregations at the same time. Below, we find the mean, count the number of non-missing values, and the max of the `charges` column by each unique `sex`.

```
In [12]: ins.groupby('sex').agg({'charges': ['mean', 'count', 'max']}).round(0)
```

Out[12]:

	charges		
	mean	count	max
sex			
female	12570.0	662	63770.0
male	13957.0	676	62593.0

1.9.3 Multiple Grouping columns

pandas allows us form groups based on multiple columns. In the below example, each unique combination of `sex` and `region` for a group. There are 8 unique groups and for each of these groups the same aggregations as above are performed on the `charges` column.

```
In [13]: ins.groupby(['sex', 'region']).agg({'charges': ['mean', 'count', 'max']}).round(0)
```

Out[13]:

		charges		
		mean	count	max
sex	**region**			
female	**northeast**	12953.0	161	58571.0
	northwest	12480.0	164	55135.0
	southeast	13500.0	175	63770.0
	southwest	11274.0	162	48824.0
male	**northeast**	13854.0	163	48549.0
	northwest	12354.0	161	60021.0
	southeast	15880.0	189	62593.0
	southwest	13413.0	163	52591.0

1.9.4 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 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.

```
In [14]: pt = ins.pivot_table(index='sex', columns='region',
                               values='charges', aggfunc='mean').round(0)
         pt
```

Out[14]:

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

1.9.5 Styling DataFrames

To help make your data really pop-out, pandas provides ways to style your DataFrame in various ways. Below, the maximum value of each column is highlighted.

```
In [15]: pt.style.highlight_max()
```

```
Out[15]: <pandas.io.formats.style.Styler at 0x117f25668>
```

1.10 Tidying

Many datasets will be messy upon first inspection and pandas provides ways to clean them so that they are put into 'tidy' form.

1.10.1 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 by default pandas will read them in as strings.

```
In [16]: pc = pd.read_csv('../data/tidy/planecrashinfo.csv')
         pc.head(3)
```

Out[16]:

	date	time	location	operator	flight_no
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	?
1	September 07, 1909	?	Juvisy-sur-Orge, France	?	?
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	?

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

```
In [17]: pc = pd.read_csv('../data/tidy/planecrashinfo.csv', na_values='?')
         pc.head(3)
```

Out[17]:

	date	time	location	operator	flight_no
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	NaN
1	September 07, 1909	NaN	Juvisy-sur-Orge, France	NaN	NaN
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	NaN

1.10.2 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.

```
In [18]: aboard = pc['aboard']
         aboard.head()
```

Out[18]:

0	2 ă (passengers:1ă crew:1)
1	1 ă (passengers:0ă crew:1)
2	5 ă (passengers:0ă crew:5)
3	1 ă (passengers:0ă crew:1)
4	20 ă (passengers:?ă 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.

```
In [19]: aboard.str.extract(r'(\d+)?\D*(\d+)?\D*(\d+)?').head()
```

Out[19]:

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

1.10.3 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 average arrival delay of airlines at different airports. All 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.

```
In [20]: aad = pd.read_csv('../data/tidy/average_arrival_delay.csv').head()
         aad
```

Out[20]:

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

```
In [21]: aad.melt(id_vars='airline', var_name='airport', value_name='delay').head(10)
```

Out[21]:

	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

1.11 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.

```
In [22]: from sqlalchemy import create_engine
         engine = create_engine('sqlite:///../data/neurIPS.db')

         authors = pd.read_sql('Authors', engine)
         pa = pd.read_sql('PaperAuthors', engine)
```

Output the first 5 rows of each DataFrame.

```
In [23]: authors.head()
```

Out[23]:

	Id	Name
0	178	Yoshua Bengio
1	200	Yann LeCun
2	205	Avrim Blum
3	347	Jonathan D. Cohen
4	350	Samy Bengio

```
In [24]: pa.head()
```

Out[24]:

	Id	PaperId	AuthorId
0	1	5677	7956
1	2	5677	2649
2	3	5941	8299
3	4	5941	8300
4	5	5941	575

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.

```
In [25]: pa.merge(authors, how='left', left_on='AuthorId', right_on='Id').head()
```

Out[25]:

	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
3	4	5941	8300	8300	Aditya Menon
4	5	5941	575	575	Robert C. Williamson

1.12 Time Series Analysis

One of the original purposes of pandas was to do time series analysis. Below, we read in the last 5 years of Apple's stock price data with help from the excellent IEX Trading API.

```
In [26]: aapl = pd.read_json('https://api.iextrading.com/1.0/stock/aapl/chart/5y')
         aapl = aapl.set_index('date')
         aapl.head()
```

Out[26]:

date	change	changeOverTime	changePercent	close	high	label	low	open
2014-03-11	0.678987	0.000000	0.974	70.4057	70.7537	Mar 11, 14	69.9460	70.3217
2014-03-12	0.068289	0.000970	0.097	70.4740	70.5712	Mar 12, 14	69.8686	70.1982
2014-03-13	-0.782734	-0.010147	-1.111	69.6913	70.8746	Mar 13, 14	69.4956	70.5830
2014-03-14	-0.782740	-0.021265	-1.123	68.9085	69.7228	Mar 14, 14	68.6866	69.4470
2014-03-17	0.269229	-0.017440	0.391	69.1778	69.6020	Mar 17, 14	69.0609	69.3038

1.12.1 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 20, 2018 through March 5, 2018.

```
In [27]: aapl['2018-02-20':'2018-03-05']
```

Out[27]:

date	change	changeOverTime	changePercent	close	high	label	low
2018-02-20	-0.571300	1.404243	-0.336	169.2724	171.6463	Feb 20, 18	168.8489
2018-02-21	-0.768301	1.393330	-0.454	168.5041	171.5084	Feb 21, 18	168.4450
2018-02-22	1.408600	1.413337	0.836	169.9127	171.3409	Feb 22, 18	169.1345
2018-02-23	2.955000	1.455308	1.739	172.8677	173.0154	Feb 23, 18	170.9371
2018-02-26	3.418000	1.503854	1.977	176.2856	176.6993	Feb 26, 18	173.5670
2018-02-27	-0.571300	1.495740	-0.324	175.7143	177.7730	Feb 27, 18	175.4878
2018-02-28	-0.265950	1.491963	-0.151	175.4484	177.9059	Feb 28, 18	175.3794
2018-03-01	-3.073200	1.448313	-1.752	172.3752	177.0785	Mar 1, 18	170.0703
2018-03-02	1.191900	1.465241	0.691	173.5670	173.6557	Mar 2, 18	169.8634
2018-03-05	0.600850	1.473776	0.346	174.1679	175.0741	Mar 5, 18	171.9024

1.12.2 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.

```
In [28]: aapl_mc = aapl.resample('M').agg({'close':'mean'})
         aapl_mc.head()
```

Out[28]:

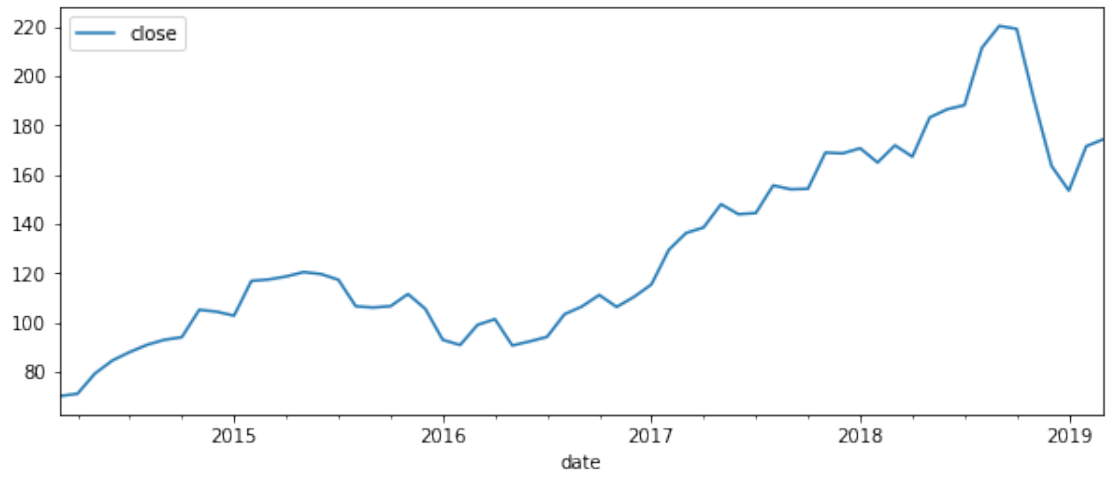
date	close
2014-03-31	70.166513
2014-04-30	71.060305
2014-05-31	79.266967
2014-06-30	84.539476
2014-07-31	87.980627

1.13 Visualization

pandas provides basic visualization abilities by giving its users a few nice default plots. Below, we plot the average monthly closing price of Apple for the last 5 years.

```
In [29]: %matplotlib inline
         aapl_mc.plot(kind='line', figsize=(10, 4))
```

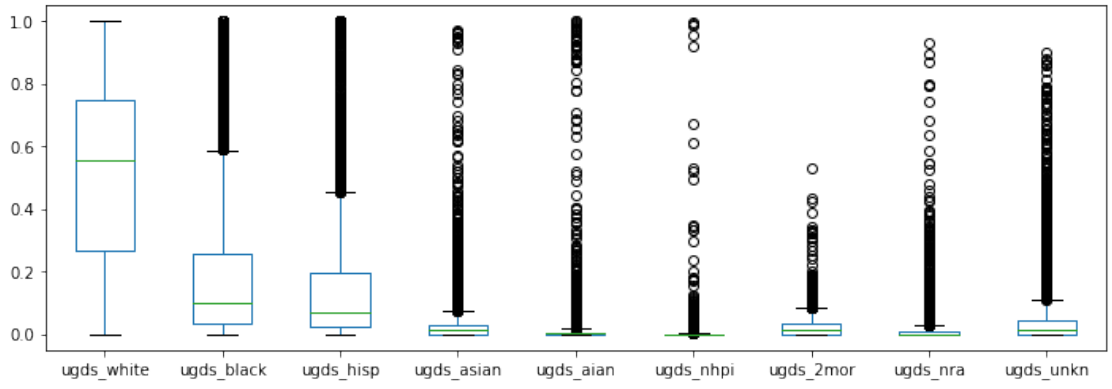
```
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x11b86fe10>
```



Here, we use the college race data to create a box plot of each of the race percentage columns.

```
In [30]: cr.plot(kind='box', figsize=(12, 4))
```

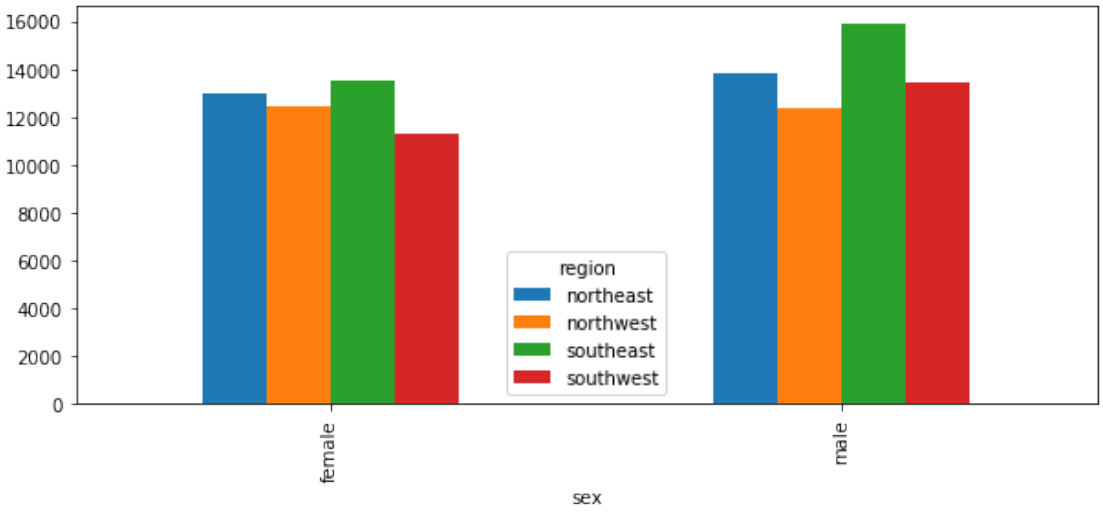
```
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x11b836f60>
```



We turn our pivot table from above into a bar graph.

```
In [31]: pt.plot(kind='bar', figsize=(10, 4))
```

```
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x11ce34630>
```



1.14 Much More

The above was just a small sampling that pandas has to offer, but does show many basic examples from many of the major sections of the library.

Chapter 2

The DataFrame and Series

2.1 Overview

2.1.1 Objectives

- Identify a DataFrame as a two-dimensional data structure with an **index**, **columns**, and **values**
- Identify a Series as a single dimensional data structure with an **index** and **values**
- Know the difference between the **index** and **values**

2.1.2 Resources

- Official Documentation
- Package Overview
- Intro to Data Structures

2.1.3 The DataFrame and Series

The DataFrame and Series are the two primary pandas objects that we will be using throughout this course.

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

2.2 Import pandas and read in data with the `read_csv` function

By convention, pandas is imported and aliased as `pd`. We will read in the `bikes` dataset with the `read_csv` function. Its first parameter is the location of the file relative to the current directory as a string. All of the data for this course is stored in the `data` directory one level above where this notebook is located. The two dots in the path passed to `read_csv` are interpreted as the directory immediately above the current one.

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

2.2.1 Display DataFrame in Jupyter Notebook

We assigned the output from the `read_csv` function to the `bikes` variable which now refers to our DataFrame object. Let's get a visual display of our DataFrame by writing the variable name as the last line in a code cell.

```
In [2]: bikes
```

Out[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

2.2.2 Default output

pandas defaults to outputting 60 rows and 20 columns. These display options (and many others) can be changed. This will be covered later.

2.3 Our first methods - `head` and `tail`

A very useful and simple method is `head`, which by default will return the first 5 rows of the DataFrame. 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.

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

Out[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

```
In [4]: bikes.tail()
```

Out[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
50084	17534938	Subscriber	Male	2017-12-30 13:07:00	2017-12-30 13:34:00	1625
50085	17534969	Subscriber	Male	2017-12-30 13:34:00	2017-12-30 13:44:00	585
50086	17534972	Subscriber	Male	2017-12-30 13:34:00	2017-12-30 13:48:00	824
50087	17535645	Subscriber	Female	2017-12-31 09:30:00	2017-12-31 09:33:00	178
50088	17536246	Subscriber	Male	2017-12-31 15:22:00	2017-12-31 15:26:00	214

2.3.1 First and Last `n` rows

Both the `head` and `tail` methods take a single integer parameter `n`, which controls the number of rows to return.

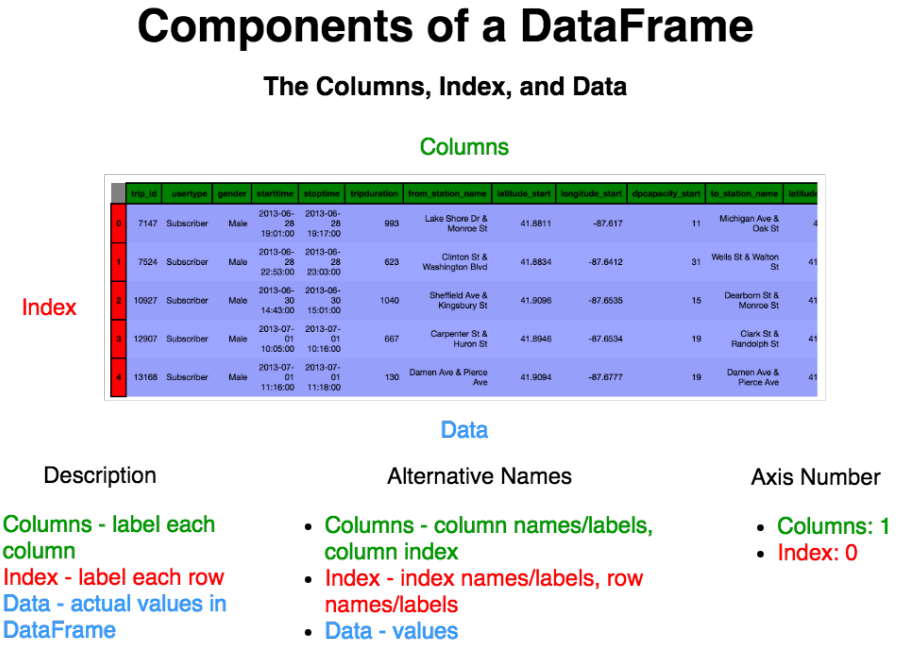
```
In [5]: bikes.head(3)
```

Out[5]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040

2.4 Components of a DataFrame - columns, index, and data

The DataFrame is composed of three separate components that you must know. The **columns**, the **index**, and the **data**. These terms will be used throughout the course 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 **index** provides a label for each row
- The **columns** provide a label for each column
- The **index** is also referred to as the **row names/labels**
- The **columns** are also referred to as the **column names/labels** or the **column index**
- An individual element of the index is referred to as an **index label/name** or **row label/name**
- An individual element of the columns is a **column name/label**
- The index and the columns are always in **bold font**
- Collectively the index and the columns are known as the **axes** (or individually as an **axis**)
- pandas uses integers to refer to each axis; 0 for the index and 1 for the columns. This is borrowed directly from numpy
- The actual **data** is always in normal font
- The **data** is also referred to as the **values**

2.5 What type of object is `bikes`

As we said previously, `bikes` is a DataFrame. Let's verify this:

```
In [6]: type(bikes)
```

```
Out[6]: pandas.core.frame.DataFrame
```

2.5.1 Fully-qualified name

Only the word after the last dot is the class name. The `bikes` variable has type `DataFrame`. Python always returns the location and module name of where the class was defined.

2.5.2 Location and module name?

The fully-qualified name holds the location in your computer where the class is defined. In this example, `pandas` is a directory that contains another directory `core` which contains a file `frame.py` which defines the `DataFrame` class.

2.5.3 Package, sub-package, and module

The top level directory of other files and directories containing Python files is technically called a **package**. In this example `pandas` is the package. All directories within the package are called **sub-packages** such as `core`. All Python files (those ending in .py) are called **modules**.

2.5.4 Where are the packages located?

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 the help of the built-in `site` module's `getsitepackages` function.

```
In [7]: import site
        site.getsitepackages()
```

```
Out[7]: ['/Users/Ted/anaconda3/lib/python3.6/site-packages']
```

2.6 Select a single column from a DataFrame - a Series

To select a single column from a DataFrame, pass the name of one of the columns to the brackets operator, `[]`. The returned object will be a pandas **Series**. Let's select the column name `tripduration`, assign it to a variable, and output it to the screen.

```
In [8]: trip_duration = bikes['tripduration']
        trip_duration
```

```
Out[8]:
0     993
1     623
2    1040
3     667
4     130
Name: tripduration, Length: 50089, dtype: int64
```

2.7 `head` and `tail` methods work the same with a Series

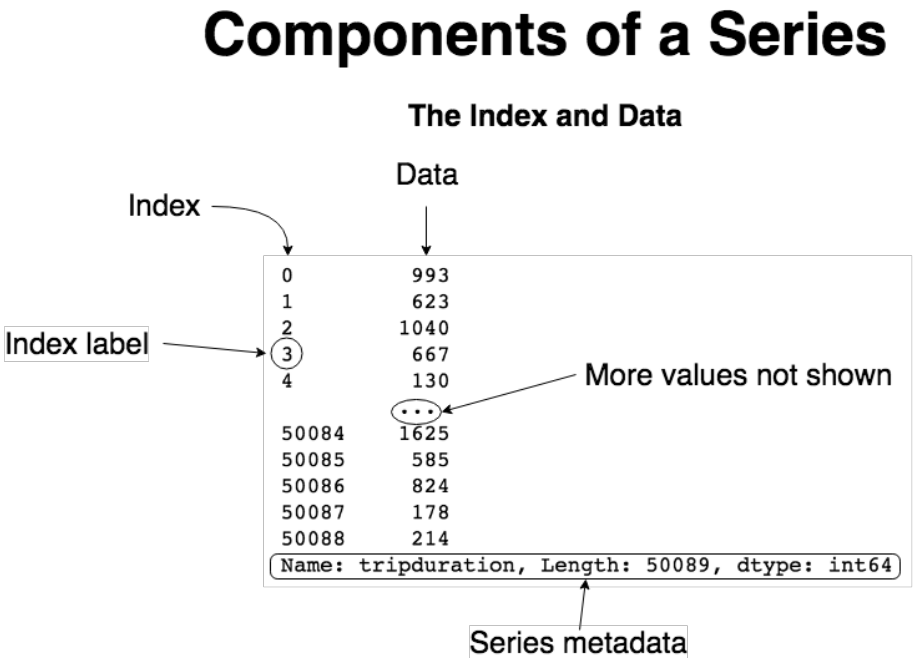
Use the `head` and `tail` methods to condense the output.

```
In [9]: trip_duration.tail(3)
```

```
Out[9]:
50086    824
50087    178
50088    214
Name: tripduration, dtype: int64
```

2.8 Components of a Series - the index and the data

A Series is simpler than a DataFrame with just a single dimension of data. It has two components - the **index** and the **data**. It is essentially a one-column DataFrame. Let's take a look at a stylized Series graphic.



The definition for the index and data components are the same as they are for a DataFrame.

2.8.1 Output of Series vs DataFrame

Notice that there is no nice HTML styling for the Series. It's just plain text. Also, below each Series will be some metadata on it - the **name**, **length**, and **dtype**.

- The **name** is not important right now. If the Series is 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. Each column of data must be of only one particular data type. These will be covered later.

It's important to note that this metadata is NOT part of the Series itself and is just some extra info pandas outputs for your information.

2.9 Exercises

Use the `bikes` DataFrame for the following:

2.9.1 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.

2.9.2 Exercise 2

What type of object is `events`?

2.9.3 Exercise 3

Select the last 2 rows of the `bikes` DataFrame and assign it to the variable `bikes_last_2`. What type of object is `bikes_last_2`?

Chapter 3

Data Types and Missing Values

3.1 Overview

3.1.1 Objectives

- Know all the possible column data types
- Know that each value in a column must be of the same data type
- Know the representations of missing values and which ones are used for each data type
- Know how to get metadata with `shape`, `size`, and `info`

3.2 Data Types

Each column of data in a pandas DataFrame has a data type. This is a very similar concept to types in Python. Just like every object has a type, every column has a data type.

3.2.1 Most Common Data Types

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

- boolean
- integer
- float
- object (mainly strings)
- datetime (a specific moment in time)

3.2.2 Other Data Types

There are three other data types that are less common. We will cover them when necessary.

- category
- timedelta (a specific amount of time)
- period (a specific time period)

3.2.3 More on the most common data types

boolean

Boolean columns contain only two values: `True` and `False`

integer

Whole numbers without a decimal

float

Numbers with decimals

object

The object data type is a bit confusing. Each value in an object column can be **any** Python object. But, nearly all of the time, object data type columns contain strings. They can contain any other Python object such as integers, floats, or even complex types such as lists or dictionaries. There is no specific data type for strings as there are in most other data processing packages. When you see that object is the data type, you should assume you have a string column.

datetime

A datetime is a specific moment in time with both a date (month, year, day) and a time (hour, minute, second, fraction of a second). All datetimes in pandas have nanosecond (1 billionth of a second) precision.

3.2.4 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.

3.2.5 A major exception with the object data type

The object data type, is unfortunately, an exception to the rule in the previous section. Although columns that have object data type are typically strings, there is no guarantee that each value will be a string. You could very well have an integer, list, or even another DataFrame as a value in an object column.

3.3 Missing Value Representation

3.3.1 `NaN`, `None`, and `NaT`

pandas represents missing values differently based on the data type of the column.

- `NaN` stands for not a number and is technically a floating point value
- `None` is the literal Python object `None`. This will only be found in object columns
- `NaT` stands for not a time and is used for missing values in datetime, timedelta, and period columns

3.3.2 Missing values for each data type

- **boolean and integer** - No representation for missing values exist for boolean and integer columns. This is unfortunate, but a current limitation (See the next section for an update on this limitation). If you have boolean or integer columns and need to also have missing values within them, then these columns will be coerced to a float.
- **floats** - Use only `NaN` as the missing value.
- **object** - Columns of object data type can contain any Python object so technically you may see `NaN`, `None`, or `NaT` but primarily you will see `None`.
- **datetime, timedelta, period** - Use only `NaT` as the missing value.

3.3.3 Integer NaN update for pandas 0.24

With the release of pandas version 0.24 (February 2019), missing value representation was made possible for a special kind of integer data type called **Int64Dtype**. There is still no missing value representation for the default integer data type. For more on this new addition, see chapter **X**.

3.3.4 Even more data types

pandas is a library under constant flux and has implemented a couple of newer data types for very specific situations. The **SparseDType** is used for data that contains a vast majority of values as 0 or some other constant. The other is **IntervalDtype** for creating values that represent an entire interval, such as all the values between 3 and 5. Both of these data types are newer additions to the library, are rarely used, and will be covered in chapter **X**.

3.4 Finding the data types of each column

The `dtypes` DataFrame attribute (NOT a method) returns the data type of each column. Let's get the data types of our `bikes` DataFrame. Note that the returned data is a Series with the column names now in the index and the data type as the values.

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

Out[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

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

Out[2]:

trip_id	int64
usertype	object
gender	object
starttime	object
stoptime	object

```
Length: 19, dtype: object
```

3.4.1 Think string whenever you see object

pandas does not have a string data type like most databases, but when you see **object** you should assume that the column consists entirely of strings.

3.4.2 Why are `starttime` and `stoptime` object data types?

If you look at the output of the `bikes` DataFrame, it's apparent that both the `starttime` and `stoptime` columns are datetimes but the output from `dtypes` from the above states that they are objects.

When reading in a text file, such as `bikes.csv`, you must explicitly tell pandas which columns are datetimes. We can force pandas to read these columns as datetimes with the `parse_dates` parameter of the

`read_csv` function. It accepts a list of the column names we would like to make datetimes. Let's reread the data:

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

Out[3]:

trip_id	int64
usertype	object
gender	object
starttime	datetime64[ns]
stoptime	datetime64[ns]

```
Length: 19, dtype: object
```

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

Booleans, integers, floats, datetimes and timedeltas all use a particular amount of memory for each of their values. The memory is measured in **bits**. By default, pandas uses 64 bits to represent integers, floats, datetimes, and timedeltas. pandas will use 64 bits regardless if you have a 32 or 64 bit machine.

It is possible to use a different number of bits for integers and floats. Integers can be either 8, 16, 32, or 64 bits while floats can be 16, 32, 64, or 128. For instance, 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. It's also rare to see anything other than 64 bit integers or floats since that is the default. You would need to manually change their size to get a different type.

Booleans are stored as 8-bits, which is a single **byte**. Datetimes and timedeltas are always stored as 64 bits. **Objects** can store any Python object, so there is no set amount of memory for each of their values.

3.5 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.

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

```
Out[4]: (50089, 19)
```

3.5.1 Total number of elements with the `size` attribute

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

```
In [5]: bikes.size
```

```
Out[5]: 951691
```

3.5.2 Get Data Types plus more with the `info` method

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

```
In [6]: bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 19 columns):
trip_id              50089 non-null int64
usertype             50089 non-null object
gender               50089 non-null object
starttime            50089 non-null datetime64[ns]
stoptime             50089 non-null datetime64[ns]
tripduration         50089 non-null int64
from_station_name    50089 non-null object
latitude_start       50083 non-null float64
longitude_start      50083 non-null float64
dpcapacity_start     50083 non-null float64
to_station_name      50089 non-null object
latitude_end         50077 non-null float64
longitude_end        50077 non-null float64
dpcapacity_end       50077 non-null float64
temperature          50089 non-null float64
visibility           50089 non-null float64
wind_speed           50089 non-null float64
precipitation        50089 non-null float64
events               50089 non-null object
dtypes: datetime64[ns](2), float64(10), int64(2), object(5)
memory usage: 7.3+ MB
```

3.6 Exercises

Use the `bikes` DataFrame for the following:

3.6.1 Exercise 1

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

3.6.2 Exercise 2

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

3.6.3 Exercise 3

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

3.6.4 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

4.1 Overview

4.1.1 Objectives

- Extract the components of a DataFrame and verify their type
- Know that a `RangeIndex` is the default DataFrame index
- Select values from the index like a list
- Understand what makes a meaningful index
- Use the `index_col` parameter of `read_csv` to set an index on read
- Use the `set_index` method to set an index after read

4.2 Extracting the components of a DataFrame

The DataFrame consists of three components - the index, columns, and data. It is possible to extract each component and assign them into their own variable. Let's read in a small dataset to show how this is done. Notice that when we read in the data, we choose the first column to be the index with the `index_col` parameter. More on this later.

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

Out[1]:

	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	9.0
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

4.2.1 The attributes `index`, `columns`, and `values`

The index, columns, and data are each separate objects. Notice that each of these objects are extracted as attributes and NOT methods. Let's assign them as their own variables.

```
In [2]: index = df.index
        columns = df.columns
        data = df.values
```

4.2.2 View these objects

Let's output each of these objects:

```
In [3]: index
```

```
Out[3]: Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina', 'Cornelia'], dtype='object')
```

```
In [4]: columns
```

```
Out[4]: Index(['state', 'color', 'food', 'age', 'height', 'score'], dtype='object')
```

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

```
Out[5]: 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)
```

4.2.3 What are these objects?

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

```
In [6]: type(index)
```

```
Out[6]: pandas.core.indexes.base.Index
```

```
In [7]: type(columns)
```

```
Out[7]: pandas.core.indexes.base.Index
```

```
In [8]: type(data)
```

```
Out[8]: numpy.ndarray
```

4.2.4 pandas `Index` type

pandas has a special type of object called an `Index`. This object is similar to a list or a one dimensional array. You can think of it as a sequence of labels for either the rows or the columns. You will not deal with this object directly much, so we will not go into further details about it here. Notice that the both the index and columns are of the same type.

4.2.5 numpy's `ndarray`

The data is stored as a numpy `ndarray` (which stands for n-dimensional array). It is this array that is doing the bulk of the workload in pandas.

4.2.6 Operating with the DataFrame as a whole

You will rarely need to operate with these components directly and instead be working with the entire DataFrame.

4.3 Extracting the components of a Series

Similarly, we can extract the two Series components - the index and the data. Let's first select a single column as a Series:

```
In [9]: color = df['color']
        color
```

Out[9]:

Jane	blue
Niko	green
Aaron	red
Penelope	white
Dean	gray
Christina	black
Cornelia	red

```
Name: color, dtype: object
```

```
In [10]: color.index
```

```
Out[10]: Index(['Jane', 'Niko', 'Aaron', 'Penelope', 'Dean', 'Christina', 'Cornelia'], dtype='object')
```

```
In [11]: color.values
```

```
Out[11]: array(['blue', 'green', 'red', 'white', 'gray', 'black', 'red'],
               dtype=object)
```

4.4 More on the index

The index is an important (and sometimes confusing) part of both the Series and DataFrame. It provides us with a label for each row. It is always **bold** and is NOT a column of data. It is a separate component of our DataFrame.

4.4.1 The default index

If you don't specify an index when first reading in a DataFrame, then pandas will create one for you as integers beginning at 0. An index always exists even if it just appears to be the row number. Let's read in the movie dataset without setting an index.

```
In [12]: movie = pd.read_csv('../data/movie.csv')
         movie.head()
```

Out[12]:

	title	year	color	content_rating	duration
0	Avatar	2009.0	Color	PG-13	178.0
1	Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0
2	Spectre	2015.0	Color	PG-13	148.0
3	The Dark Knight Rises	2012.0	Color	PG-13	164.0
4	Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN

4.4.2 Notice the integers in the index

These integers are the default index labels for each of the rows. Let's examine the underlying index object.

```
In [13]: idx = movie.index
         idx

Out[13]: RangeIndex(start=0, stop=4916, step=1)

In [14]: type(idx)

Out[14]: pandas.core.indexes.range.RangeIndex
```

4.4.3 A RangeIndex

pandas has many different types of index objects. A `RangeIndex` is similar to a Python `range` object. The values of a RangeIndex are not actually stored in memory and only accessed when requested.

4.4.4 Select a value from the index

The index is a complex object on its own and has many features (many more than a Python list). We will not cover it in-depth because it is used infrequently. That said, the minimum we should know about an index is how to select values from it. We use **integer location**, just like it were a Python list, to make selections. Let's select a single value from it.

```
In [15]: idx[5]

Out[15]: 5
```

4.4.5 A numpy array underlies the index

To get the underlying numpy array, use the `values` attribute. This is similar to how we get the underlying data from a pandas DataFrame.

```
In [16]: idx.values

Out[16]: array([   0,    1,    2, ..., 4913, 4914, 4915])
```

If you don't assign the index to a variable, you can retrieve the array from the DataFrame by chaining the attributes together like this:

```
In [17]: movie.index.values

Out[17]: array([   0,    1,    2, ..., 4913, 4914, 4915])
```

4.5 Setting an index on read

pandas allows us to use one of the columns as the index when reading in the data.

4.5.1 Setting an index when reading in the data with `read_csv`

The `read_csv` function gives us dozens of parameters that allow us to read in a wide variety of csv 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.

4.5.2 Reread the movie dataset with the movie title as the index

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

```
In [18]: movie = pd.read_csv('../data/movie.csv', index_col='title')
         movie.head()
```

Out[18]:

title	year	color	content_rating	duration
Avatar	2009.0	Color	PG-13	178.0
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0
Spectre	2015.0	Color	PG-13	148.0
The Dark Knight Rises	2012.0	Color	PG-13	164.0
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN

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 - **title** is NOT a column name, but rather the **name of the index**.

4.5.3 Extract the new index and output its type

We again have an `Index` object.

```
In [19]: idx2 = movie.index
         idx2
```

```
Out[19]: 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)
```

```
In [20]: type(idx2)
```

```
Out[20]: pandas.core.indexes.base.Index
```

4.6 Selecting values from this index

Just like we did with our `RangeIndex`, we use the brackets operator to select a single index value.

```
In [21]: idx2[105]
```

```
Out[21]: 'Poseidon'
```

4.6.1 Selection with slice notation

As with Python lists, you can select a range of values using slice notation with the three components, start, stop, and step separated by a colon like this - `start:stop:step`

```
In [22]: idx2[100:120:4]
```

```
Out[22]: Index(['The Fast and the Furious', 'The Sorcerer's Apprentice', 'Warcraft',
                'Transformers', 'Hancock'],
               dtype='object', name='title')
```

4.6.2 Selection with a list of integers

You can select multiple individual values with a list of integers.

```
In [23]: nums = [1000, 453, 713, 2999]
         idx2[nums]
```

```
Out[23]: Index(['The Life Aquatic with Steve Zissou', 'Daredevil', 'Daddy Day Care',
                'The Ladies Man'],
               dtype='object', name='title')
```

4.7 Choosing a good index

First, it's never necessary to choose an index for your DataFrames. You can complete all of your analysis with just the default `RangeIndex`. Setting a column to be an index can help identify the rows such as we did with the movie titles above.

I suggest choosing columns that are both **unique** and **descriptive**. Although uniqueness is not enforced, it does help when needing to identify one particular row.

4.8 Setting the index after read with the `set_index` method

It is possible to set the index after reading the data with the `set_index` method. Pass it the name of the column you would like to use as the index. Below, we read in our data without setting an index.

```
In [24]: movie = pd.read_csv('../data/movie.csv')
         movie = movie.set_index('title')
         movie.head()
```

Out[24]:

title	year	color	content_rating	duration
Avatar	2009.0	Color	PG-13	178.0
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0
Spectre	2015.0	Color	PG-13	148.0
The Dark Knight Rises	2012.0	Color	PG-13	164.0
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN

4.8.1 Reassigned `movie` variable

Notice above that we reassigned the variable name `movie` as the result of the `set_index` command. This is because `set_index` makes an entire new copy of the data. It does not change the original DataFrame. We say the operation **does NOT happen in-place**.

4.9 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.

4.9.1 Get current option value with `get_option`

You can retrieve any option with the `get_option` function. Notice that this is not a DataFrame method. It is a function that you access directly from `pd`. It is not necessary to remember the option names. They are all available in the docstrings of the `get_option` function. Below are three of the most common options to change.

```
In [25]: pd.get_option('display.max_columns')

Out[25]: 20

In [26]: pd.get_option('display.max_rows')

Out[26]: 60

In [27]: pd.get_option('display.max_colwidth')

Out[27]: 50
```

4.9.2 Use the `set_option` function to change an option value

To set a new option value, use the `set_option` function. You can set as many options as you would like at one time. It's 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 40 and the maximum number of rows to 8. We will now be able to view all the columns in the movie DataFrame.

```
In [28]: pd.set_option('display.max_columns', 40, 'display.max_rows', 8)
         movie
```

Out[28]:

title	year	color	content_rating	duration
Avatar	2009.0	Color	PG-13	178.0
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0
Spectre	2015.0	Color	PG-13	148.0
The Dark Knight Rises	2012.0	Color	PG-13	164.0
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN

4.9.3 All available options

See the documentation for all the available options.

4.10 Exercises

4.10.1 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?

4.10.2 Exercise 2

Use `set_index` to set the index and keep the column as part of the data

4.10.3 Exercise 3

Assign the index of the movie DataFrame that has the titles in the index to its own variable. Output the last 10 movies titles.

4.10.4 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?

4.10.5 Exercise 5

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

Chapter 5

Five-Step Process for Data Exploration

5.1 Overview

A major pain point for beginners is writing too many lines of code in a single cell. When you are learning, you need 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

5.1.1 Apply to every part of the analysis

You can apply this process to every part of your data analysis. Let's see this process in action with a few examples. We will start by reading in the data.

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

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

In this step, we make a call to the `read_csv` function.

```
In [2]: pd.read_csv('../data/bikes.csv')
```

Out[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

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

Looking above, the output appears to be correct. Of course, we can't inspect every single value, but we can do a sanity check to see if indeed a reasonable-looking DataFrame is produced.

5.1.4 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.

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

5.1.5 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.

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

Out[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

5.1.6 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 line 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 will 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.

5.1.7 More examples

Let's see another simple example of the five-step process for data exploration in the notebook. Instead of writing each of the five steps in their own cell, the final result is shown with an explanation that follows.

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

Out[5]:

trip_id	usertype	gender	starttime	stoptime	tripduration
7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

In this part of the analysis, we want to set one of the columns as the index. During step 1, we write a single line of code, `bikes.set_index('trip_id')`. In step 2, we manually verify that the output looks correct. In step 3, we assign the result to a variable with `bikes = bikes.set_index('trip_id')`. In step 4, we output the head as another line of code, and in step 5, we move on to the next cell.

5.1.8 No strict requirement for one line of code

The above examples each had a single main line of code followed by outputting the head of the DataFrame. Often times there will be a few more very simple lines of code that can be written in the same cell. You should not strictly adhere 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 block has three lines of code. The first is very simple and creates a list of column names as strings. This is an instance where multiple lines of code are easily interpreted.

```
In [6]: cols = ['gender', 'tripduration']
        bikes_gt = bikes[cols]
        bikes_gt.head()
```

Out[6]:

trip_id	gender	tripduration
7147	Male	993
7524	Male	623
10927	Male	1040
12907	Male	667
13168	Male	130

5.1.9 When to assign the result to a variable

Not all operations on our data will 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 have immediate access to the previous result.

5.1.10 When to create a new variable name

In the second example, `bikes` was reassigned to itself. We did this because we no longer needed the original DataFrame. In the third example, we created an entirely new variable, `bikes_gt`. This was done because we wanted to keep the `bikes` DataFrame. Creating new variables also makes it easier to trace the flow of work. Debugging is easier as well since we will have preserved the result of the cell in its own variable (assuming we did not overwrite it in a later cell).

5.1.11 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 is producing the desired result is important. Doing this also provides feedback to help you think about what avenues to explore next.

Chapter 6

Solutions

6.1 2. The DataFrame and Series

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

        pd.options.display.max_columns = 40
        bikes = pd.read_csv('../data/bikes.csv')
```

6.1.1 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.

```
In [2]: events = bikes['events']
        events.head(10)
```

```
Out[2]:
0    mostlycloudy
1    partlycloudy
2    mostlycloudy
3    mostlycloudy
4    partlycloudy
5    mostlycloudy
6          cloudy
7          cloudy
8          cloudy
9    mostlycloudy
Name: events, dtype: object
```

6.1.2 Exercise 2

What type of object is **events**?

```
In [3]: # it's a Series
        type(events)
```

```
Out[3]: pandas.core.series.Series
```

6.1.3 Exercise 3

Select the last 2 rows of the `bikes` DataFrame and assign it to the variable `bikes_last_2`. What type of object is `bikes_last_2`?

```
In [4]: # it's a DataFrame
        bikes_last_2 = bikes.tail(2)
        type(bikes_last_2)
```

```
Out[4]: pandas.core.frame.DataFrame
```

6.2 3. Data Types and Missing Values

6.2.1 Exercise 1

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

```
In [5]: # a Series
        type(bikes.dtypes)
```

```
Out[5]: pandas.core.series.Series
```

6.2.2 Exercise 2

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

```
In [6]: # a tuple of rows, columns
        type(bikes.shape)
```

```
Out[6]: tuple
```

6.2.3 Exercise 3

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

The object `None` is returned. What you see is just output printed to the screen.

```
In [7]: info_return = bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 19 columns):
trip_id               50089 non-null int64
usertype              50089 non-null object
gender                50089 non-null object
starttime             50089 non-null object
stoptime              50089 non-null object
tripduration          50089 non-null int64
from_station_name     50089 non-null object
latitude_start        50083 non-null float64
longitude_start       50083 non-null float64
dpcapacity_start      50083 non-null float64
to_station_name       50089 non-null object
latitude_end          50077 non-null float64
longitude_end         50077 non-null float64
dpcapacity_end        50077 non-null float64
temperature           50089 non-null float64
```

```
visibility           50089 non-null float64
wind_speed           50089 non-null float64
precipitation        50089 non-null float64
events               50089 non-null object
dtypes: float64(10), int64(2), object(7)
memory usage: 7.3+ MB
```

```
In [8]: type(info_return)
```

```
Out[8]: NoneType
```

6.2.4 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.

```
In [9]: bikes.info(memory_usage='deep')
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 19 columns):
trip_id              50089 non-null int64
usertype             50089 non-null object
gender               50089 non-null object
starttime            50089 non-null object
stoptime             50089 non-null object
tripduration         50089 non-null int64
from_station_name    50089 non-null object
latitude_start       50083 non-null float64
longitude_start      50083 non-null float64
dpcapacity_start     50083 non-null float64
to_station_name      50089 non-null object
latitude_end         50077 non-null float64
longitude_end        50077 non-null float64
dpcapacity_end       50077 non-null float64
temperature          50089 non-null float64
visibility           50089 non-null float64
wind_speed           50089 non-null float64
precipitation        50089 non-null float64
events               50089 non-null object
dtypes: float64(10), int64(2), object(7)
memory usage: 28.9 MB
```

6.3 4. Setting a meaningful index

6.3.1 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?

```
In [10]: movies = pd.read_csv('../data/movie.csv', index_col='director_name')
         movies.head()
```

Out[10]:

director_name	title	year	color	content_rating
James Cameron	Avatar	2009.0	Color	PG-13
Gore Verbinski	Pirates of the Caribbean: At World's End	2007.0	Color	PG-13
Sam Mendes	Spectre	2015.0	Color	PG-13
Christopher Nolan	The Dark Knight Rises	2012.0	Color	PG-13
Doug Walker	Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN

Director name isn't unique. There aren't any other good column names to use as an index.

6.3.2 Exercise 2

Use `set_index` to set the index and keep the column as part of the data

```
In [11]: movies = pd.read_csv('../data/movie.csv')
         movies = movies.set_index('title', drop=False)
         movies.head(3)
```

Out[11]:

title	title	year	color
Avatar	Avatar	2009.0	Color
Pirates of the Caribbean: At World's End	Pirates of the Caribbean: At World's End	2007.0	Color
Spectre	Spectre	2015.0	Color

6.3.3 Exercise 3

Assign the index of the movie DataFrame that has the titles in the index to its own variable. Output the last 10 movies titles.

```
In [12]: index = movies.index
         index[-10:]
```

```
Out[12]: Index(['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')
```

6.3.4 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?

```
In [13]: movies = pd.read_csv('../data/movie.csv', index_col=-5)
         movies.head()
```

Out[13]:

plot_keywords	title
avatar | future | marine | native | paraplegic	Avatar
goddess | marriage ceremony | marriage proposal | pir...	Pirates of the Caribbean: At World's End
bomb | espionage | sequel | spy | terrorist	Spectre
deception | imprisonment | lawlessness | police offic...	The Dark Knight Rises
NaN	Star Wars: Episode VII - The Force Awakens

It chooses the column name with that integer location.

6.3.5 Exercise 5

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

```
In [14]: pd.reset_option('all')
```

```
html.border has been deprecated, use display.html.border instead
(currently both are identical)


: boolean
    use_inf_as_null had been deprecated and will be removed in a future
    version. Use `use_inf_as_na` instead.



/Users/Ted/anaconda3/lib/python3.6/site-packages/pandas/core/config.py:615: FutureWarning: html.border h
(currently both are identical)

  warnings.warn(d.msg, FutureWarning)
/Users/Ted/anaconda3/lib/python3.6/site-packages/pandas/core/config.py:615: FutureWarning:
: boolean
    use_inf_as_null had been deprecated and will be removed in a future
    version. Use `use_inf_as_na` instead.

  warnings.warn(d.msg, FutureWarning)
```

Part II

Selecting Subsets of Data

Chapter 7

Selecting Subsets of Data from DataFrames with just the brackets

7.1 Overview

7.1.1 Objectives

- Know the three indexers `[ ]`, `loc`, and `iloc` are used to select subsets of data
- The primary purpose of *just the brackets* is to select columns of a DataFrame

7.1.2 Resources

- Read Indexing and Selecting - **up to but not including Selection By Callable**

7.2 Selecting Subsets of Data

One of the most common tasks during a data analysis is to select subsets of the dataset. In pandas, this means selecting particular rows and/or columns from our DataFrame (or Series).

7.2.1 Examples of Selections of Subsets of Data

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

7.2.2 Selection of 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

Resulting 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

7.2.3 Selection of rows

	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

Resulting DataFrame:

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

7.2.4 Simultaneous selection of rows and 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

Resulting DataFrame:

	color	age	height
Aaron	red	12	120
Dean	gray	32	180

7.3 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 columns 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 and ends at n-1 for each row and column. Take a look above at our sample 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 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 INTEGER.

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

The documentation uses the term **indexing** frequently. This term is essentially just a one-word phrase to say **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).

7.4 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 what makes the subset selection.

We will go in-depth on how to make selections with each of these indexers. Each indexer has different rules for how it works. All our selections will look similar to the following, except they will have something placed within the brackets.

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

7.4.1 Terminology

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

7.5 Begin with *just the brackets*

As we saw in the last notebook, 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.

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

Out[1]:

	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	9.0
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

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

Out[2]:

Jane	blue
Niko	green
Aaron	red
Penelope	white
Dean	gray
Christina	black
Cornelia	red

```
Name: color, dtype: object
```

7.6 Select Multiple Columns with a List

You can select multiple columns by placing them in a list inside of just the brackets. Notice that a DataFrame and NOT a Series is returned:

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

Out[3]:

	color	age	score
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

7.6.1 You must use an inner set of brackets

You might be tempted to do the following which will NOT work. You must pass the columns names as a **list**. Remember that a list is defined by a set of square brackets.

```
In [4]: # NO! An exception is raised
        df['color', 'age', 'score']


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

7.6.2 Use two lines of code to select multiple columns

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

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

Out[5]:

	color	age	score
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

7.6.3 Column order does not matter

You can create new DataFrames in any column order you wish. They do need not match the original column order.

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

Out[6]:

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

7.7 Exercises

For the following exercises, make sure to use the movie dataset with `title` set as the index. It's good practice to shorten your output with the `head` method.

7.7.1 Exercise 1

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

7.7.2 Exercise 2

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

7.7.3 Exercise 3

Select a single column as a DataFrame and not a Series

7.7.4 Exercise 4

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

Chapter 8

Selecting Subsets of Data from DataFrames with `loc`

8.1 Overview

8.1.1 Objectives

- `loc` can select rows, columns, or rows and columns simultaneously
- `loc` selects primarily by **label**

8.2 Subset selection with `loc`

The `loc` indexer selects data in a different manner than *just the brackets*. We must learn its set of rules.

8.2.1 Simultaneous row and column subset selection with `loc`

The `loc` indexer can select rows and columns simultaneously. You cannot do this with *just the brackets*. This is done by separating the row and column selections with a **comma**. The selection will look something like this:

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

8.2.2 `loc` only selects data by LABEL

Very importantly, `loc` primarily selects data by the **LABEL** of the rows and columns. Provide `loc` with the label of the rows and/or columns you would like to select.

8.2.3 Select two rows and three columns with `loc`

If we wanted to select the rows `Dean` and `Cornelia` along with the columns `age`, `state`, and `score` we would do this:

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

Out[1]:
```

	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	9.0
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

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

Out[2]:

	age	state	score
Dean	32	AK	1.8
Cornelia	69	TX	2.2

8.2.4 The possible types of selections for `loc`

Row or column selections can be any of the following:

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

We can use any of these for either row or column selections with `loc`.

8.2.5 Select two rows and a single column:

Let's use a list for the rows and a string for the column. The row selection is `['Dean', 'Aaron']` and the column selection is `food`. Note how this returns a Series since we are selecting exactly a single column.

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

Out[3]:

Dean	Cheese
Aaron	Mango

```
Name: food, dtype: object
```

8.3 Use slice notation to select a range of rows

We have seen slice notation when working with Python lists. This same notation is allowed with DataFrames. Let's choose all of the rows from `Jane` to `Penelope` with slice notation along with the columns `state` and `color`.

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

Out[4]:

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

8.3.1 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.

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


        SyntaxError: invalid syntax
```

8.3.2 Use the `slice` function to separate out the selection in a different line

There is a built-in `slice` function that you can use to assign your selection to a variable. It takes the same three values **start**, **stop**, and **step**, but this time as function parameters.

```
In [6]: rows = slice('Jane', 'Penelope')
        cols = ['state', 'color']
        df.loc[rows, cols]
```

Out[6]:

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

8.3.3 Slice both the rows and columns

```
In [7]: df.loc[:'Dean', 'height':]
```

Out[7]:

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

Use `None` to denote an empty part of the slice.

```
In [8]: rows = slice(None, 'Dean')
        cols = slice('height', None)
        df.loc[rows, cols]
```

Out[8]:

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

8.3.4 Slices with `loc` are inclusive of the stop value

Notice that the stop value is included in the returned DataFrame. When slicing Python lists, the last element is **excluded**.

8.3.5 Use slice notation or the slice function?

Almost no one uses the `slice` function, so you will probably want to use slice notation. That said, the slice function does help separate the row and column selections into their own lines of code.

8.3.6 Selecting all of the rows and some of the columns

It is possible to select all of the rows by using a single colon. Here, we select all of the rows and two of the columns.

```
In [9]: cols = ['food', 'color']
        df.loc[:, cols]
```

Out[9]:

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

Equivalently, we could use the slice function like this:

```
In [10]: rows = slice(None)
         cols = ['food', 'color']
         df.loc[rows, cols]
```

Out[10]:

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

8.3.7 The above is not necessary! Use *just the brackets*

You would never see two columns with all the rows selected like that. This is exactly what *just the brackets* are built for.

```
In [11]: cols = ['food', 'color']
         df[cols]
```

Out[11]:

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

8.3.8 A single colon is slice notation for select all

That single colon might be intimidating but it is technically slice notation that selects all items. See the following example with a list:

```
In [12]: a_list = [1, 2, 3, 4, 5, 6]
         a_list[:]
```

```
Out[12]: [1, 2, 3, 4, 5, 6]
```

8.3.9 Use a single colon to select all the columns

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

```
In [13]: rows = ['Penelope','Cornelia']
         df.loc[rows, :]
```

Out[13]:

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

8.3.10 The above can be shortened

By default, pandas will select all of the columns if you only provide a row selection. Providing the colon is not necessary and the following will do the same:

```
In [14]: rows = ['Penelope', 'Cornelia']
         df.loc[rows]
```

Out[14]:

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

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

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

Similarly, we can slice from Niko through Dean while selecting all of the columns. We do not provide a specific column selection. By default, Pandas returns all of the columns.

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

Out[15]:

	state	color	food	age	height	score
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

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

8.3.12 Other slicing examples

You can slice in a variety of ways such as selecting every other row by setting the step size to 2:

```
In [16]: df.loc['Niko':'Christina':2]
```

Out[16]:

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

Omitting the start value to include all rows until the stop value:

```
In [17]: df.loc[:'Penelope']
```

Out[17]:

	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	9.0
Penelope	AL	white	Apple	4	80	3.3

Omitting the stop value to keep all rows after the start value:

```
In [18]: df.loc['Aaron':]
```

Out[18]:

	state	color	food	age	height	score
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

8.4 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.

```
In [19]: rows = 'Jane'
         cols = 'state'
         df.loc[rows, cols]

Out[19]: 'NY'
```

8.4.1 Select a single row as a Series with `loc`

The `loc` indexer will return 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.

```
In [20]: df.loc['Niko']
```

Out[20]:

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

```
Name: Niko, dtype: object
```

8.4.2 Is this confusing?

Think about why this output may be confusing.

8.5 Summary of `loc`

- Primarily uses labels
- Can select rows and columns simultaneously
- Selection can be a single label, a list of labels, a slice of labels, or a boolean Series/array
- Put a comma between row and column selections

8.6 Exercises

8.6.1 Exercise 1

Read in the movie dataset and set the title column as the index. Select all columns for the movie 'The Dark Knight Rises'.

8.6.2 Exercise 2

Select all columns for the movies 'Tangled' and 'Avatar'.

8.6.3 Exercise 3

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

8.6.4 Exercise 4

Can you tell what the data type of the `year` column is by just looking at its values?

```
In [21]: # Turn this into a markdown cell and write your answer here
```

8.6.5 Exercise 5

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

8.6.6 Exercise 6

Select every 100th movie between 'Tangled' and 'Forrest Gump'. Why doesn't 'Forrest Gump' appear in the results?

Chapter 9

Selecting Subsets of Data from DataFrames with `iloc`

9.1 Overview

9.1.1 Objectives

- `iloc` can select rows, columns, or rows and columns simultaneously
- `iloc` selects only by **integer location**

9.2 Getting started with `iloc`

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

9.2.1 Simultaneous row and column subset selection with `iloc`

Selection with iloc will look like the following:

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

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

Out[1]:

	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	9.0
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

9.2.2 Use 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.

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

Out[2]:

	state	score
Aaron	FL	9.0
Dean	AK	1.8

9.2.3 The possible types of selections for `iloc`

Row or column selections can be any of the following:

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

9.2.4 Slice the rows and use a list for the columns

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

Out[3]:

	height	food
Jane	165	Steak
Aaron	120	Mango
Dean	180	Cheese
Cornelia	150	Beans

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

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

Out[4]:

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

9.2.6 Selecting some rows and all of the columns

If you leave the column selection empty, then all of the columns will be selected.

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

Out[5]:

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Aaron	FL	red	Mango	12	120	9.0

It is possible to rewrite the above with both row and column selections by using a colon to represent a slice of all of the columns. Just as with `loc`, this can be instructive and reinforce the concept that the `iloc` does simultaneous row and column selection with the row selection coming first.

```
In [6]: df.iloc[rows, :]
```

Out[6]:

	state	color	food	age	height	score
Penelope	AL	white	Apple	4	80	3.3
Aaron	FL	red	Mango	12	120	9.0

9.2.7 Select all of the rows and some of the columns

```
In [7]: cols = [1, 5]
        df.iloc[:, cols]
```

Out[7]:

	color	score
Jane	blue	4.6
Niko	green	8.3
Aaron	red	9.0
Penelope	white	3.3
Dean	gray	1.8
Christina	black	9.5
Cornelia	red	2.2

9.2.8 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 `1` and `5`.

```
In [8]: cols = [1, 5]
        df[cols]
```

```
KeyError: "None of [Int64Index([1, 5], dtype='int64')] are in the [columns]"
```

9.2.9 Select some rows and a single column

Note that a Series is returned whenever a single row or column is selected.

```
In [9]: rows = [2, 3, 5]
        cols = 4
        df.iloc[rows, cols]
```

Out[9]:

Aaron	120
Penelope	80
Christina	172

```
Name: height, dtype: int64
```

9.2.10 A trick to select a single row 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.

```
In [10]: rows = [2, 3, 5]
         cols = [4]
         df.iloc[rows, cols]
```

Out[10]:

	height
Aaron	120
Penelope	80
Christina	172

9.2.11 Select a single row as a Series with `iloc`

By passing a single integer to `iloc`, it will select one row as a Series:

```
In [11]: df.iloc[2]
```

Out[11]:

state	FL
color	red
food	Mango
age	12
height	120
score	9

```
Name: Aaron, dtype: object
```

9.3 Summary of `iloc`

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

9.4 Exericses

- Use the movie dataset for the following exercises

9.4.1 Exercise 1

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

9.4.2 Exericse 2

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

9.4.3 Exercise 3

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

Chapter 10

Selecting Subsets of Data from a Series

10.1 Overview

This notebook will teach you how to select a subset of data from a Series.

10.1.1 Objectives

- Learn how to select subsets from a Series using `loc` and `iloc`

10.2 Using Dot Notation to Select a Column as a Series

Previously we learned how to use *just the brackets* to select a single column as a Series. Another common way to do this uses dot notation. Place the column name following a dot after the name of your DataFrame.

Let's read in the movie dataset, set the index as the title and then select the year with dot notation.

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

Out[1]:

title	year	color	content_rating	duration
Avatar	2009.0	Color	PG-13	178.0
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0
Spectre	2015.0	Color	PG-13	148.0
The Dark Knight Rises	2012.0	Color	PG-13	164.0
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN

```
In [2]: movie.year.head()
```

Out[2]:

title	2009.0
Avatar	2007.0
Pirates of the Caribbean: At World's End	2015.0
Spectre	2012.0
The Dark Knight Rises	NaN

```
Name: year, dtype: float64
```

10.2.1 I don't recommend doing this

Although this is valid pandas syntax I don't recommend using this notation for the following reasons: * You cannot select columns with spaces in them * You cannot select columns that have the same name as a pandas method such as `count` * You cannot use a variable name that is assigned to the name of a column

Using *just the brackets* **always** works so I recommend doing the following instead:

```
In [3]: movie['year'].head()
```

```
Out[3]:
```

title	2009.0
Avatar	2007.0
Pirates of the Caribbean: At World's End	2015.0
Spectre	2012.0
The Dark Knight Rises	NaN

```
Name: year, dtype: float64
```

10.2.2 Why even know about this?

pandas is written differently by different people and you will definitely see this syntax around so it's important to be aware of it. It also has the advantage of providing tab-completion help when chaining a method to the end. Place your cursor at the end of the following two lines and press tab. Only the one that selects via dot notation will show the available methods. This helps me remember what methods are possible, so sometimes I will use this to find the method I need and then change the syntax back to the brackets.

```
In [4]: # place your cursor after the dot and press tab
        movie.year.

In [5]: # place your cursor after the dot and press tab
        movie['year'].
```

10.3 Selecting Subsets of Data From a Series

Selecting subsets of data from a Series is very similar to that as a DataFrame. Since there are no columns in a Series, there isn't a need to use *just the brackets*. Instead, you can do all of your subset selection with `loc` and `iloc`. Let's select the column for IMDB scores as a Series and output the head.

```
In [6]: imdb = movie['imdb_score']
        imdb.head()
```

```
Out[6]:
```

title	7.9
Avatar	7.1
Pirates of the Caribbean: At World's End	6.8
Spectre	8.5
The Dark Knight Rises	7.1

```
Name: imdb_score, dtype: float64
```

10.3.1 Selection with a scalar, a list, and a slice

Just like with a DataFrame, both `loc` and `iloc` accept either a single scalar, a list, or a slice. The `loc` indexer also accepts a boolean Series/array which will be covered in a later chapter. Let's select the movie IMDB score for 'Forrest Gump':

```
In [7]: imdb.loc['Forrest Gump']
```

```
Out[7]: 8.8
```

Select both 'Forrest Gump' and 'Avatar' with a list. Notice that a Series is returned.

```
In [8]: locs = ['Forrest Gump', 'Avatar']
        imdb.loc[locs]
```

```
Out[8]:
```

title	8.8
Forrest Gump	7.9

```
Name: imdb_score, dtype: float64
```

Select every 100th movie from 'Avatar' to 'Forrest Gump' with slice notation:

```
In [9]: imdb.loc['Avatar':'Forrest Gump':100]
```

```
Out[9]:
```

title	7.9
Avatar	6.7
The Fast and the Furious	7.5
Harry Potter and the Sorcerer's Stone	6.7
Epic	4.8
102 Dalmatians	5.6
Pompeii	6.3
Wall Street: Money Never Sleeps	5.5
Hop	6.5

```
Name: imdb_score, dtype: float64
```

10.3.2 Repeat with `iloc`

Select a single score

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

```
Out[10]: 6.9
```

Select multiple scores with a list

```
In [11]: ilocs = [10, 20, 30]
         imdb.iloc[ilocs]
```

```
Out[11]:
```

title	6.9
Batman v Superman: Dawn of Justice	7.5
The Hobbit: The Battle of the Five Armies	7.8

```
Name: imdb_score, dtype: float64
```

Select multiple scores with a slice

```
In [12]: imdb.iloc[3000:3050:10]
```

```
Out[12]:
```

title	6.8
Quartet	5.4
The Guru	6.2
Machine Gun McCain	6.3
The Blue Butterfly	6.9

```
Name: imdb_score, dtype: float64
```

10.3.3 Trouble with *just the brackets*

It is possible to use just the brackets to make the same selections as above. See the following examples:

```
In [13]: imdb['Forrest Gump']
```

```
Out[13]: 8.8
```

```
In [14]: imdb['Avatar':'Forrest Gump':100]
```

Out[14]:

title	
Avatar	7.9
The Fast and the Furious	6.7
Harry Potter and the Sorcerer's Stone	7.5
Epic	6.7
102 Dalmatians	4.8
Pompeii	5.6
Wall Street: Money Never Sleeps	6.3
Hop	5.5
	6.5

```
Name: imdb_score, dtype: float64
```

```
In [15]: ilocs = [10, 20, 30]
         imdb[ilocs]
```

Out[15]:

title	
Batman v Superman: Dawn of Justice	6.9
The Hobbit: The Battle of the Five Armies	7.5
	7.8

```
Name: imdb_score, dtype: float64
```

```
In [16]: imdb[3000:3050:10]
```

Out[16]:

title	
Quartet	6.8
The Guru	5.4
Machine Gun McCain	6.2
The Blue Butterfly	6.3
	6.9

```
Name: imdb_score, dtype: float64
```

10.3.4 Can you spot the problem?

The major issue is that using *just the brackets* is **ambiguous** and **not explicit**. We don't know if we are selecting by label or by integer location. With `loc` and `iloc`, it is clear what our intentions are. I suggest using `loc` and `iloc` for clarity.

10.4 Comparison to Python Lists and Dictionaries

It may be helpful to compare pandas ability to make selections by label and integer location to that of Python lists and dictionaries. Python lists allow for selection of data only through **integer location**. You can use a single integer or slice notation to make the selection but NOT a list of integers. Let's see examples of subset selection of lists using integers:

```
In [17]: a_list = [10, 5, 3, 89, 20, 44, 37]

In [18]: a_list[4]

Out[18]: 20

In [19]: a_list[-3:]

Out[19]: [20, 44, 37]
```

10.4.1 Selection by label with Python dictionaries

All values in each dictionary are labeled by a key. We use this key to make single selections. Dictionaries only allow selection with a single label. Slices and lists of labels are not allowed.

```
In [20]: d = {'a':1, 'b':2, 't':20, 'z':26, 'A':27}
         d['a']

Out[20]: 1

In [21]: d['A']

Out[21]: 27
```

10.4.2 pandas has the power of lists and dictionaries

DataFrames and Series are able to make selections with integers like a list and with labels like a dictionary.

10.5 Exercises

10.5.1 Exercise 1

Read in the bikes dataset. We will be using it for the rest of the questions. Select the wind speed column as a Series and assign it to a variable and output the head. What kind of index does this Series have?

10.5.2 Exercise 2

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

10.5.3 Exercise 3

Copy and paste your answer to problem 2 below but use `loc` instead. Do you get the same result? Why not?

10.5.4 Exercise 4

Read in the movie dataset and set the index to be the title. Select `actor1` as a Series. Who is the `actor1` for 'My Big Fat Greek Wedding'?

10.5.5 Exercise 5

Find `actor1` for your favorite two movies?

10.5.6 Exercise 6

Select the last 10 values from `actor1` using two different ways?

Chapter 11

Boolean Indexing Single Conditions

11.1 Overview

11.1.1 Objectives

- Boolean Indexing or Boolean Selection is the selection of a subset of a Series/DataFrame based on the **values** themselves and not the row/column labels or integer location
- Boolean means **True** or **False**
- Each row of the DataFrame will be kept or discarded based on the boolean value aligned with it
- Boolean selection is a two-step process
  - First, create a **filter** - a sequence of True/False values the same length as the DataFrame/Series
  - Second, pass this filter to one of the indexers `[ ]` or `loc`
- Boolean selection does not work with `iloc`
- The indexing operators are overloaded—change functionality depending on what is passed to them
- The filter is commonly created by comparing a column of data (a Series) against some scalar value

11.2 Boolean Indexing

Boolean indexing, also referred to as **Boolean Selection**, 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.

11.2.1 Examples of Boolean Indexing

Let's see some examples of actual questions (in plain English) that boolean indexing can help us answer from the bikes dataset.

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

The term **query** is used to refer to these sorts of questions.

11.2.2 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.

11.2.3 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, otherwise it is discarded.

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

When you perform boolean indexing, each row of the DataFrame (or value of a Series) will have a True or False value associated with it depending on whether or not it meets the condition. True/False values are known as boolean. The documentation refers to the entire procedure as boolean indexing. Since we are using the booleans to select subsets of data, it is sometimes referred to as **boolean selection**.

11.2.5 Beginning with a small DataFrame

We will perform our first boolean indexing on a dataset of 5 rows. Let's assign the head of the bikes dataset to its own variable. The `bikes_head` DataFrame has five rows in it.

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

Out[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

11.3 Manual filtering of the data

Let's find all the rides with a trip duration greater than 900. We will do this manually by inspecting the data.

11.3.1 Create a list of booleans

By inspecting the data, we see that the 1st and 3rd rows have a trip duration greater than 900. A list of 5 boolean values is created, one for each row. The first 1st and 3rd values are `True`. The others are `False`.

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

11.3.2 Variable name `filt`

The variable name `filt` will be used throughout the book to contain the sequence of booleans. `filt` simply stands for filter. Being consistent with variables makes your code easier to understand.

11.3.3 Pass this list into just the brackets

The above list has `True` in both the 1st and 3rd position. These will be the rows that are kept during boolean indexing. To formally do boolean indexing, we place the list inside the brackets.

```
In [3]: bikes_head[filt]
```

Out[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040

11.3.4 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. Now, all of a sudden, this example is showing that entire rows are selected with boolean values. This is what makes pandas, unfortunately, a confusing library to use.

11.4 Operator Overloading

Just the brackets is **overloaded**. This means, that depending on the inputs, pandas will do something completely different. Here are the rules for the different objects you pass to the brackets.

- **string**—return a column as a Series
- **list of strings**—return all those columns as a DataFrame
- **sequence of booleans**—select all rows where True
- **slice**—select rows (can do both label and integer location—confusing!) I never do this as it is ambiguous. This has not been covered yet.

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

11.4.1 Using booleans in a Series and not a list

Instead of using a list to contain our booleans, we can store them in a Series. This produces the same output. Below, we use the Series constructor to create a Series object.

```
In [4]: filt = pd.Series([True, False, True, False, False])
        filt
```

Out[4]:

0	True
1	False
2	True
3	False
4	False

```
dtype: bool
```

11.4.2 Use the boolean Series to do the boolean selection

Placing the Series directly in the brackets will again select only the rows which have True values in the Series.

```
In [5]: bikes_head[filt]
```

Out[5]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040

11.5 Practical Boolean Selection

We will almost never create boolean lists/Series manually like we did above but instead use the actual data to create them.

11.5.1 Creating boolean Series from column data

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

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

11.5.2 Create a boolean Series

Let's create a boolean Series by determining which rows have a trip duration of over 1000 seconds.

```
In [6]: filt = bikes['tripduration'] > 1000
        filt.head(10)
```

```
Out[6]:
0    False
1    False
2     True
3    False
4    False
5    False
6    False
7    False
8     True
9    False
Name: tripduration, dtype: bool
```

11.5.3 Manually verify correctness

Let's output the head of the `tripduration` Series to manually verify that indeed integer locations 2 and 8 are the ones greater than 1000.

```
In [7]: bikes['tripduration'].head(10)
```

```
Out[7]:
0     993
1     623
2    1040
3     667
4     130
5     660
6     565
7     505
8    1300
9     922
Name: tripduration, dtype: int64
```

11.5.4 Complete our boolean indexing

We created our boolean Series, `filt`, using the greater than comparison operator on the `tripduration` column. We can now pass this result into just the brackets to filter the entire DataFrame. Verify that all `tripduration` values are greater than 1000.

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

Out[8]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523
11	24673	Subscriber	Male	2013-07-04 18:13:00	2013-07-04 18:42:00	1697
12	26214	Subscriber	Male	2013-07-05 10:02:00	2013-07-05 10:40:00	2263

11.5.5 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 retrieve the `shape` of the DataFrame.

```
In [9]: bikes.shape

Out[9]: (50089, 19)

In [10]: bikes_duration_1000 = bikes[filt]
         bikes_duration_1000.shape

Out[10]: (10178, 19)
```

About 20% of the rides are longer than 1000 seconds.

11.6 Boolean selection in one line

Often, you will see boolean selection happen in a single line of code instead of the multiple lines we used above. Put the expression for the filter directly inside the brackets.

```
In [11]: bikes[bikes['tripduration'] > 1000].head()
```

Out[11]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523
11	24673	Subscriber	Male	2013-07-04 18:13:00	2013-07-04 18:42:00	1697
12	26214	Subscriber	Male	2013-07-05 10:02:00	2013-07-05 10:40:00	2263

I recommend assigning the filter as a separate variable to help with readability.

11.7 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 the brackets which completes our selection.

```
In [12]: filt = bikes['events'] == 'cloudy'
         bikes[filt].head()
```

Out[12]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
6	18880	Subscriber	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565
7	19689	Subscriber	Male	2013-07-03 09:07:00	2013-07-03 09:16:00	505
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
19	40879	Subscriber	Male	2013-07-09 13:14:00	2013-07-09 13:20:00	384

11.8 Exercises

11.8.1 Exercise 1

Read in the movie dataset and set the index to be the title. Select all movies that have Tom Hanks as `actor1`. How many of these movies has he starred in?

11.8.2 Exercise 2

Select movies with and IMDB score greater than 9.

Chapter 12

Boolean Indexing Multiple Conditions

12.1 Overview

12.1.1 Objectives

- Create complex filters with the and (`&`), or (`|`), and not (`~`) logical operators
- Wrap each filter with parentheses when more than one occur at the same line
- For readability, assign each filter to its own variable
- Use the `isin` method to test for multiple equalities in the same column

12.2 Multiple condition expression

So far, our boolean selections have involved a single condition. You can 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.

12.2.1 Use `&, | , ~`

Although Python uses the keywords `and`, `or`, and `not`, these will not work when with pandas. You must use the following operators:

- `&` for and
- `|` for or
- `~` for not

12.2.2 Our first multiple condition expression

Let's find all the rides longer than 1,000 seconds when it was cloudy. We assign each condition to separate variables and then combine them with the and operator, `&`.

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

```
Out[1]:
```

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

```
In [2]: filt1 = bikes['tripduration'] > 1000
        filt2 = bikes['events'] == 'cloudy'
        filt = filt1 & filt2
        bikes[filt].head(3)
```

Out[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
80	90932	Subscriber	Female	2013-07-22 07:59:00	2013-07-22 08:19:00	1224

12.3 Multiple conditions in one line

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

12.3.1 Use parentheses to separate conditions

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

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

12.3.2 Same results

We can then place this expression inside of just the brackets to get the same results:

```
In [3]: bikes[(bikes['tripduration'] > 1000) & (bikes['events'] == 'cloudy')].head(3)
```

Out[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
80	90932	Subscriber	Female	2013-07-22 07:59:00	2013-07-22 08:19:00	1224

Again, I prefer assigning each condition to its own variable for better readability.

12.4 Using an `or` condition

Let's find all the rides that were done by females **or** had trip durations longer than 1,000 seconds. For the or condition, we use the pipe character `|`.

```
In [4]: filt1 = bikes['tripduration'] > 1000
        filt2 = bikes['gender'] == 'Female'
        filt = filt1 | filt2
        bikes[filt].head(3)
```

Out[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
9	23558	Subscriber	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922

12.5 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 1000, we could do it like this:

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

Out[5]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667

Of course, inverting single conditions is pretty pointless as we can simply use the less than or equal to operator instead like this:

```
In [6]: filt = bikes['tripduration'] <= 1000
        bikes[filt].head(3)
```

Out[6]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667

12.5.1 Invert a more complex condition

Typically, we will save the not operator for reversing more complex conditions. Let's reverse 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.

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

Out[7]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667

12.5.2 Even more complex conditions

It is possible to build extremely complex conditions to select rows of your DataFrame that meet a very specific condition. For instance, we can select males riders with trip duration between 1,000 and 2,000 seconds along with female riders with trip duration between 5,000 and 10,000 seconds. With multiple conditions, its probably best to break out the logic into multiple steps:

```
In [8]: filt1 = ((bikes['gender'] == 'Male') &
                 (bikes['tripduration'] >= 1000) &
                 (bikes['tripduration'] <= 2000))

        filt2 = ((bikes['gender'] == 'Female') &
                 (bikes['tripduration'] >= 5000) &
                 (bikes['tripduration'] <= 10000))
        filt = filt1 | filt2
        bikes[filt].head(10)
```

Out[8]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523
11	24673	Subscriber	Male	2013-07-04 18:13:00	2013-07-04 18:42:00	1697
13	30404	Subscriber	Male	2013-07-06 09:43:00	2013-07-06 10:06:00	1365
26	51130	Subscriber	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043
34	54257	Subscriber	Male	2013-07-12 18:13:00	2013-07-12 18:40:00	1616
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
41	64257	Subscriber	Male	2013-07-15 06:26:00	2013-07-15 06:44:00	1125
47	67013	Subscriber	Male	2013-07-15 19:10:00	2013-07-15 19:34:00	1463

12.6 Lots of equality conditions in a single column - use `isin`

Occasionally, we will 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.

```
In [9]: filt = ((bikes['events'] == 'rain') |
                (bikes['events'] == 'snow') |
                (bikes['events'] == 'tstorms') |
                (bikes['events'] == 'sleet'))

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

Out[9]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727
78	89180	Subscriber	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809
79	89228	Subscriber	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999

Instead, use the `isin` method and pass it a list (or a set) of all the acceptable values:

```
In [10]: filt = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
         bikes[filt].head(3)
```

Out[10]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727
78	89180	Subscriber	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809
79	89228	Subscriber	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999

12.6.1 Combining `isin` with other filters

You can use the resulting boolean Series from the `isin` method in the same way you would from the logical operators. For instance, If we wanted to find all the rides that had the same events and had a duration greater than 10,000 we would do the following:

```
In [11]: filt1 = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
         filt2 = bikes['tripduration'] > 2000
         filt = filt1 & filt2
         bikes[filt].head()
```

Out[11]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2344	1266453	Subscriber	Female	2014-03-19 07:23:00	2014-03-19 08:00:00	2181
7697	3557596	Subscriber	Male	2014-09-12 14:20:00	2014-09-12 14:57:00	2213
8357	3801419	Subscriber	Male	2014-09-30 08:21:00	2014-09-30 08:58:00	2246
8506	3846762	Subscriber	Male	2014-10-04 12:33:00	2014-10-04 14:06:00	5568
11267	4822906	Subscriber	Male	2015-04-10 17:25:00	2015-04-10 18:00:00	2074

12.7 Exercises

12.7.1 Exercise 1

Select all movies from the 1970s.

12.7.2 Exercise 2

Select all movies from the 1970s that had IMDB scores greater than 8

12.7.3 Exercise 3

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

12.7.4 Exercise 4

Select movies that are either rated PG-13 or were made after 2010.

12.7.5 Exercise 5

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

12.7.6 Exercise 6

Reverse the condition from problem 4. In words, what have you selected.

Chapter 13

Boolean Indexing More

13.1 Overview

13.1.1 Objectives

- Boolean Selection with the brackets on a Series
- Using the `between` method instead of an `and` condition
- Simultaneously select rows with boolean selection and columns with a list of names with `loc`
- Select rows with missing values with the `isna` method

13.2 Boolean Selection on a Series

All the examples thus far have taken place on the bikes DataFrame. 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.

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

```
In [2]: temp = bikes['temperature']
        temp.head()
```

```
Out[2]:
0    73.9
1    69.1
2    73.0
3    72.0
4    73.0
Name: temperature, dtype: float64
```

Let's select temperatures greater than 90

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

```
Out[3]:
54    91.0
55    91.0
56    91.0
61    93.0
62    93.0
```

```
Name: temperature, dtype: float64
```

Select temperature less than 0 or greater than 95

```
In [4]: filt1 = temp < 0
        filt2 = temp > 95
        filt = filt1 | filt2
        temp[filt].head()
```

Out[4]:

395	96.1
396	96.1
397	96.1
1871	-2.0
2049	-2.0

```
Name: temperature, dtype: float64
```

13.2.1 Re-read data with `starttime` in the index

The default index is not very helpful. Let's reread the data with `starttime` in the index. While, this column may not be unique it does provide us with useful labels for each row.

```
In [5]: bikes = pd.read_csv('../data/bikes.csv',
                            parse_dates=['starttime', 'stoptime'],
                            index_col='starttime')
        bikes.head()
```

Out[5]:

starttime	trip_id	usertype	gender	stoptime	tripduration
2013-06-28 19:01:00	7147	Subscriber	Male	2013-06-28 19:17:00	993
2013-06-28 22:53:00	7524	Subscriber	Male	2013-06-28 23:03:00	623
2013-06-30 14:43:00	10927	Subscriber	Male	2013-06-30 15:01:00	1040
2013-07-01 10:05:00	12907	Subscriber	Male	2013-07-01 10:16:00	667
2013-07-01 11:16:00	13168	Subscriber	Male	2013-07-01 11:18:00	130

```
In [6]: temp2 = bikes['temperature']
        temp2.head()
```

Out[6]:

starttime	73.9
2013-06-28 19:01:00	69.1
2013-06-28 22:53:00	73.0
2013-06-30 14:43:00	72.0
2013-07-01 10:05:00	73.0

```
Name: temperature, dtype: float64
```

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

```
In [7]: filt = temp2 > 90
        temp2[filt].head()
```

Out[7]:

starttime	
2013-07-16 15:13:00	91.0
2013-07-16 15:31:00	91.0
2013-07-16 16:35:00	93.0
2013-07-17 17:08:00	93.0

```
Name: temperature, dtype: float64
```

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

```
In [8]: filt1 = temp2 < 0
        filt2 = temp2 > 95
        filt = filt1 | filt2
        temp2[filt2].head()
```

Out[8]:

starttime	
2013-08-30 15:33:00	96.1
2013-08-30 15:37:00	96.1

```
Name: temperature, dtype: float64
```

13.3 The `between` method

The `between` method return 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 (inclusive), we do the following:

```
In [9]: filt = temp2.between(50, 60)
        filt.head()
```

Out[9]:

starttime	
2013-06-28 19:01:00	False
2013-06-28 22:53:00	False
2013-06-30 14:43:00	False
2013-07-01 10:05:00	False

```
Name: temperature, dtype: bool
```

```
In [10]: temp2[filt].head()
```

Out[10]:

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
2013-09-13 08:06:00	57.9

```
Name: temperature, dtype: float64
```

13.4 Simultaneous boolean selection of rows and column labels with `loc`

The `loc` indexer was thoroughly covered in an earlier notebook and will now be covered here to simultaneously select rows and columns. Earlier, we saw how `loc` made selections by label. It can accept boolean Series or boolean numpy arrays.

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 older examples of boolean selection with `loc`.

```
In [11]: filt = bikes['tripduration'] > 1000
         bikes.loc[filt].head()
```

Out[11]:

starttime	trip_id	usertype	gender	stoptime	tripduration
2013-06-30 14:43:00	10927	Subscriber	Male	2013-06-30 15:01:00	1040
2013-07-03 15:21:00	21028	Subscriber	Male	2013-07-03 15:42:00	1300
2013-07-04 17:17:00	24383	Subscriber	Male	2013-07-04 17:42:00	1523
2013-07-04 18:13:00	24673	Subscriber	Male	2013-07-04 18:42:00	1697
2013-07-05 10:02:00	26214	Subscriber	Male	2013-07-05 10:40:00	2263

```
In [12]: filt = bikes['events'].isin(['rain', 'snow', 'tstorms', 'sleet'])
         bikes.loc[filt].head()
```

Out[12]:

starttime	trip_id	usertype	gender	stoptime	tripduration
2013-07-15 16:43:00	66336	Subscriber	Male	2013-07-15 16:55:00	727
2013-07-21 16:35:00	89180	Subscriber	Male	2013-07-21 17:06:00	1809
2013-07-21 16:47:00	89228	Subscriber	Male	2013-07-21 17:03:00	999
2013-07-23 00:16:00	95044	Subscriber	Female	2013-07-23 00:26:00	563
2013-07-26 19:10:00	111568	Subscriber	Male	2013-07-26 19:33:00	1395

13.4.1 Separate row and column selection with a comma for `loc`

The nice benefit of `loc` is that it allows us to simultaneously do boolean selection along the rows and select column by label. Let's select just the events rain and snow and only the columns `events` and `tripduration`.

```
In [13]: filt = bikes['events'].isin(['rain', 'snow'])
         cols = ['events', 'tripduration']
         bikes.loc[filt, cols].head()
```

Out[13]:

starttime	events	tripduration
2013-07-15 16:43:00	rain	727
2013-07-26 19:10:00	rain	1395
2013-07-30 18:53:00	rain	442
2013-08-05 17:09:00	rain	890
2013-09-07 16:09:00	rain	978

13.5 Column to Column Comparisons

So far, we have 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, if we wanted to test whether there were more capacity at the start of the ride vs the end, we would do the following:

```
In [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.

```
In [15]: cols = ['dpcapacity_start', 'dpcapacity_end']
         bikes.loc[filt, cols].head()
```

Out[15]:

starttime	dpcapacity_start	dpcapacity_end
2013-06-28 22:53:00	31.0	19.0
2013-07-02 17:47:00	31.0	19.0
2013-07-03 15:21:00	31.0	15.0
2013-07-07 00:06:00	19.0	15.0
2013-07-08 17:06:00	23.0	19.0

13.5.1 Boolean selection with `iloc` does not work

The pandas developers decided not to allow boolean selection with `iloc`.

```
In [16]: bikes.iloc[filt]
```

```
ValueError: iLocation based boolean indexing cannot use an indexable as a mask
```

13.6 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.

```
In [17]: bikes['dpcapacity_start'].isna().head()
```

Out[17]:

starttime	
2013-06-28 19:01:00	False
2013-06-28 22:53:00	False
2013-06-30 14:43:00	False
2013-07-01 10:05:00	False

```
Name: dpcapacity_start, dtype: bool
```

13.6.1 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.

```
In [18]: filt = bikes['dpcapacity_start'].isna()
         bikes[filt]
```

Out[18]:

starttime	trip_id	usertype	gender	stoptime	tripduration
2015-09-06 07:52:00	7319012	Subscriber	Male	2015-09-06 07:55:00	207
2015-09-07 09:52:00	7341764	Subscriber	Female	2015-09-07 09:57:00	293
2015-09-15 08:25:00	7468970	Subscriber	Male	2015-09-15 08:33:00	473
2015-10-03 21:43:00	7780399	Subscriber	Female	2015-10-03 22:04:00	1268
2015-11-06 08:53:00	8207553	Subscriber	Female	2015-11-06 09:08:00	868
2015-11-18 08:55:00	8317161	Subscriber	Male	2015-11-18 09:01:00	359

13.6.2 `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 its the similarity to **NaN**, the representation of missing values. There are also other methods such as `dropna` and `fillna` that use the 'na' in their method names.

13.7 Exercises

13.7.1 Exercise 1

Select the wind speed column a a Series and assign it to a variable. Are there any negative wind speeds?

13.7.2 Exercise 2

Select all wind speed between 12 and 16.

13.7.3 Exercise 3

Select the events and gender columns for all trip durations longer than 1,000 seconds.

13.7.4 Exercise 4

Read in the movie dataset with the title as the index. We will use this DataFrame for the rest of the problems. Select all the movies such that the Facebook likes for actor 2 are greater than those for actor 1.

13.7.5 Exercise 5

Select the year, content rating, and IMDB score columns for movies from the year 2016 with IMDB score less than 4.

13.7.6 Exercise 6

Select all the movies that are missing values for content rating.

13.7.7 Exercise 7

Select all the movies that are missing both the gross and budget. Return just those columns to verify that those values are indeed missing.

13.7.8 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 14

Miscellaneous Subset Selection

14.1 Overview

In this chapter, a few more methods for subset selection are described. The methods used in this notebook do not add any additional functionality to pandas, but are covered for completeness.

14.1.1 Objectives

- query method
- slicing with just the brackets
- `at/iat` selection for selecting a single cell

14.2 Additional Subset Selections

Believe it or not, there are still a few more ways to select subsets of data. I personally do not use the methods described in this chapter as each one of them provides no more functionality over the previously covered methods. These methods are presented for completeness. They are all valid syntax and many pandas users do actually use them so you may find them valuable.

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

Out[1]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

14.3 The `query` method

The `query` method allows you to make boolean selections by writing the filter as a string. For instance, you would pass the string `'tripduration > 1000'` to select all rows of the `bikes` dataset that have a `tripduration` less than 1000. Let's see this command now.

```
In [2]: bikes.query('tripduration > 1000').head()
```

Out[2]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
8	21028	Subscriber	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300
10	24383	Subscriber	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523
11	24673	Subscriber	Male	2013-07-04 18:13:00	2013-07-04 18:42:00	1697
12	26214	Subscriber	Male	2013-07-05 10:02:00	2013-07-05 10:40:00	2263

14.3.1 Less syntax and more readable

The `query` method generally uses less syntax than boolean selection and is usually more readable. For instance, to reproduce the above with boolean selection in a single line would look like the following:

```
bikes[bikes['tripduration'] > 1000]
```

This looks a bit clumsy with the name `bikes` written twice right next to one another.

14.3.2 Use strings `and`, `or`, `not`

Unlike boolean selection, you can use the strings `and`, `or`, and `not` instead of the operators which further aides readability with `query`. Let's select `tripduration` greater than 1000 and `temperature` greater than 85.

```
In [3]: bikes.query('tripduration > 1000 and temperature > 85').head()
```

Out[3]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
53	68864	Subscriber	Male	2013-07-16 13:04:00	2013-07-16 13:28:00	1435
60	71812	Subscriber	Male	2013-07-17 10:23:00	2013-07-17 10:40:00	1024
66	73852	Subscriber	Male	2013-07-17 20:56:00	2013-07-17 21:14:00	1073
69	76516	Subscriber	Female	2013-07-18 17:22:00	2013-07-18 17:40:00	1071

14.3.3 Use the `@` symbol to reference a variable name

By default, all words within the querty string will attempt to reference the column name. You can, however, reference a variable name by preceding it with the `@` symbol. Let's assign the variable `x` to 5000 and reference it in a query.

```
In [4]: x = 5000
        bikes.query('tripduration > @x').head()
```

Out[4]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299
335	323274	Subscriber	Male	2013-08-25 17:20:00	2013-08-25 19:26:00	7533
504	442585	Subscriber	Male	2013-09-08 03:43:00	2013-09-08 07:20:00	13037

14.3.4 Reference strings with quotation marks

If you would like to reference a literal string within a query, you need to wrap it in quotes, or else pandas will attempt to use it as a column name. Let's select all 'Female' riders.

```
In [5]: bikes.query('gender == "Female"').head()
```

Out[5]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
9	23558	Subscriber	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922
14	31121	Subscriber	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610
20	42488	Subscriber	Female	2013-07-09 17:39:00	2013-07-09 17:55:00	943
21	42818	Subscriber	Female	2013-07-09 19:26:00	2013-07-09 19:38:00	726
22	43804	Subscriber	Female	2013-07-10 09:18:00	2013-07-10 09:33:00	895

14.3.5 Use 'in' for multiple equalities

You can query for multiple equalities with the word 'in' within your query like this:

```
In [6]: bikes.query('events in ["snow", "rain"]').head()
```

Out[6]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727
112	111568	Subscriber	Male	2013-07-26 19:10:00	2013-07-26 19:33:00	1395
124	130156	Subscriber	Male	2013-07-30 18:53:00	2013-07-30 19:00:00	442
161	164536	Subscriber	Male	2013-08-05 17:09:00	2013-08-05 17:23:00	890
498	437985	Subscriber	Female	2013-09-07 16:09:00	2013-09-07 16:26:00	978

There are multiple syntaxes for the above that all work the same.

- `bikes.query('["snow", "rain"] in events')`
- `bikes.query('["snow", "rain"] == events')`
- `bikes.query('events == ["snow", "rain"]')`

14.3.6 Use 'not in' to invert the condition

You can the result of an 'in' clause by placing the word 'not' before it.

```
In [7]: bikes.query('events not in ["cloudy", "partlycloudy", "mostlycloudy"]').head()
```

Out[7]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
25	47798	Subscriber	Female	2013-07-11 08:17:00	2013-07-11 08:31:00	830
26	51130	Subscriber	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043
33	53963	Subscriber	Male	2013-07-12 17:22:00	2013-07-12 17:34:00	730
45	66336	Subscriber	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727
48	67632	Subscriber	Male	2013-07-16 03:21:00	2013-07-16 03:44:00	1376

14.3.7 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. Here, we select only the `events` that were 'cloudy' for an index value greater than 4000.

```
In [8]: bikes.query('index > 4000 and events =="cloudy" ').head()
```

Out[8]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
4007	2003400	Subscriber	Male	2014-06-07 14:07:00	2014-06-07 14:31:00	1434
4008	2004978	Subscriber	Male	2014-06-07 14:58:00	2014-06-07 15:19:00	1258
4009	2005778	Subscriber	Male	2014-06-07 15:23:00	2014-06-07 15:28:00	297
4010	2007397	Subscriber	Female	2014-06-07 16:17:00	2014-06-07 16:27:00	592
4011	2007534	Subscriber	Female	2014-06-07 16:25:00	2014-06-07 16:37:00	698

14.3.8 Use multiple comparison operators in a row

You can test that a column is contained between two values without using 'and'. Place the column name between the two less than (or greater than) signs like is done below.

```
In [9]: bikes.query('5000 < tripduration < 6000').head()
```

Out[9]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
3406	1715861	Subscriber	Male	2014-05-19 09:26:00	2014-05-19 10:50:00	5059
4046	2024090	Subscriber	Male	2014-06-08 17:31:00	2014-06-08 18:54:00	5020
4506	2201068	Subscriber	Male	2014-06-21 10:45:00	2014-06-21 12:17:00	5518
7072	3291588	Subscriber	Male	2014-08-25 17:51:00	2014-08-25 19:22:00	5459

14.4 Why I avoid query

The `query` method appears to provide a more readable approach to filtering our data based on the values, but it currently lacks the ability to reference column names with spaces. For instance, if we had a column name of 'trip duration' then we would have no way to reference it with `query`.

Using boolean selection as shown in previous chapters works for every situation, so I only use it. There has been some discussion amongst the pandas developers to add this feature of selecting column names with spaces in the library, but it has yet to be built.

14.5 Slicing with just the brackets

So far, we have covered three ways to select subsets of data with just the brackets. With a single string, a list of strings, and a boolean Series. Let's quickly review those ways right now.

14.5.1 A single string

```
In [10]: bikes['tripduration'].head()
```

Out[10]:

0	993
1	623
2	1040
3	667
4	130

```
Name: tripduration, dtype: int64
```

14.5.2 A list of strings

```
In [11]: cols = ['trip_id', 'tripduration']
         bikes[cols].head()
```

Out[11]:

	trip_id	tripduration
0	7147	993
1	7524	623
2	10927	1040
3	12907	667
4	13168	130

14.5.3 A boolean Series

The previous two examples, selected columns. Boolean Series select rows.

```
In [12]: filt = bikes['tripduration'] > 5000
         bikes[filt].head()
```

Out[12]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
18	40924	Subscriber	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299
335	323274	Subscriber	Male	2013-08-25 17:20:00	2013-08-25 19:26:00	7533
504	442585	Subscriber	Male	2013-09-08 03:43:00	2013-09-08 07:20:00	13037

14.5.4 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 4.

```
In [13]: bikes[2:10:4]
```

Out[13]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
6	18880	Subscriber	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565

You can even use slice notation when the index is strings.

14.5.5 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 and `iloc` if you are using integer location, but do not use just the brackets.

14.6 Select 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.

```
In [14]: bikes.at[40, 'temperature']
```

```
Out[14]: 87.1
```

```
In [15]: bikes.iat[-30, 5]
```

```
Out[15]: 389
```

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.

14.6.1 What's the purpose of these indexers?

All the appearances of `at` and `iat` may be directly replaced with `loc` and `iloc` in your data analysis and the code would produce the exact same results. Let's verify this below.

```
In [16]: bikes.loc[40, 'temperature']
```

```
Out[16]: 87.1
```

```
In [17]: bikes.iloc[-30, 5]
```

```
Out[17]: 389
```

These `at` and `iat` indexers are optimized to select a single cell of data and therefore provide slightly better performance than `loc` or `iloc`.

14.6.2 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.

14.6.3 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. From here we time the performance for selecting with numpy, `iat`, and `iloc`. 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.

```
In [18]: values = bikes.values
```

```
In [19]: %timeit -n 5 values[-30, 5]
```

```
199 ns ś 144 ns per loop (mean ś std. dev. of 7 runs, 5 loops each)
```

```
In [20]: %timeit -n 5 bikes.iat[-30, 5]
```

```
8.03 µs ± 3.55 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

```
In [21]: %timeit -n 5 bikes.iloc[-30, 5]
```

```
11.6 µs ± 3.43 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

14.7 Exercises

14.7.1 Exercise 1

Use the `query` method to select trip durations between 5000 and 10000 when it was partlycloudly or mostlycloudly. Create a set to contain the possible events and assign it a some variable name. Reference this variable within the query string. Then redo the operation again using boolean selection.

Chapter 15

Solutions

15.1 1. Selecting Subsets of Data from DataFrames with just the brackets

```
In [1]: movie = pd.read_csv('../data/movie.csv', index_col='title')
```

15.1.1 Exercise 1

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

```
In [2]: movie['director_name'].head()
```

Out[2]:

title	James Cameron
Avatar	Gore Verbinski
Pirates of the Caribbean: At World's End	Sam Mendes
Spectre	Christopher Nolan
The Dark Knight Rises	Doug Walker

```
Name: director_name, dtype: object
```

15.1.2 Exercise 2

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

```
In [3]: movie[['director_name', 'director_fb']].head()
```

Out[3]:

title	director_name	director_fb
Avatar	James Cameron	0.0
Pirates of the Caribbean: At World's End	Gore Verbinski	563.0
Spectre	Sam Mendes	0.0
The Dark Knight Rises	Christopher Nolan	22000.0
Star Wars: Episode VII - The Force Awakens	Doug Walker	131.0

15.1.3 Exercise 3

Select a single column as a DataFrame and not a Series

```
In [4]: # make a one item list
        col = ['director_name']
        movie[col].head()
```

Out[4]:

title	director_name
Avatar	James Cameron
Pirates of the Caribbean: At World's End	Gore Verbinski
Spectre	Sam Mendes
The Dark Knight Rises	Christopher Nolan
Star Wars: Episode VII - The Force Awakens	Doug Walker

15.2 2. Selecting Subsets of Data from DataFrames with `loc`

15.2.1 Exercise 1

Read in the movie dataset and set the title column as the index. Select all columns for the movie 'The Dark Knight Rises'.

```
In [5]: movie = pd.read_csv('../data/movie.csv', index_col='title')
        movie.head(3)
```

Out[5]:

title	year	color	content_rating	duration	director_name
Avatar	2009.0	Color	PG-13	178.0	James Cameron
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes

```
In [6]: movie.loc['The Dark Knight Rises']
```

Out[6]:

year	2012
color	Color
content_rating	PG-13
duration	164
director_name	Christopher Nolan

```
Name: The Dark Knight Rises, Length: 21, dtype: object
```

15.2.2 Exercise 2

Select all columns for the movies 'Tangled' and 'Avatar'.

```
In [7]: movie.loc[['Tangled', 'Avatar']]
```

Out[7]:

title	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb
Tangled	2010.0	Color	PG	100.0	Nathan Greno	15.0	Brad Garrett	799.0
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0

15.2.3 Exercise 3

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

```
In [8]: movie.loc[['Tangled', 'Avatar'], ['year', 'imdb_score']]
```

Out[8]:

title	year	imdb_score
Tangled	2010.0	7.8
Avatar	2009.0	7.9

15.2.4 Exercise 4

Can you tell what the data type of the `year` column is by just looking at its values?

Yes, because it has a decimal value it must be a float. Integers do not have decimals

15.2.5 Exercise 5

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

```
In [9]: # yes, its missing many values. 4432 non-missing vs 4916 total
        movie.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4916 entries, Avatar to My Date with Drew
Data columns (total 21 columns):
year                4810 non-null float64
color               4897 non-null object
content_rating      4616 non-null object
duration            4901 non-null float64
director_name       4814 non-null object
director_fb         4814 non-null float64
actor1              4909 non-null object
actor1_fb           4909 non-null float64
actor2              4903 non-null object
actor2_fb           4903 non-null float64
actor3              4893 non-null object
actor3_fb           4893 non-null float64
gross               4054 non-null float64
genres              4916 non-null object
num_reviews         4867 non-null float64
num_voted_users     4916 non-null int64
plot_keywords       4764 non-null object
language            4904 non-null object
country             4911 non-null object
budget              4432 non-null float64
imdb_score          4916 non-null float64
dtypes: float64(10), int64(1), object(10)
memory usage: 1004.9+ KB
```

15.2.6 Exercise 6

Select every 100th movie between 'Tangled' and 'Forrest Gump'. Why doesn't 'Forrest Gump' appear in the results?

```
In [10]: # Forrest Gump is not a multiple of 100 away from Tangled
         movie.loc['Tangled':'Forrest Gump':100]
```

Out[10]:

title	year	color	content_rating	duration	director_name	director_fb
Tangled	2010.0	Color	PG	100.0	Nathan Greno	15.0
Shrek the Third	2007.0	Color	PG	93.0	Chris Miller	50.0
X-Men 2	2003.0	Color	PG-13	134.0	Bryan Singer	0.0
Cloud Atlas	2012.0	Color	R	172.0	Tom Tykwer	670.0
Divergent	2014.0	Color	PG-13	139.0	Neil Burger	168.0
Hidalgo	2004.0	Color	PG-13	136.0	Joe Johnston	394.0
Doom	2005.0	Color	R	113.0	Andrzej Bartkowiak	43.0
Gone Girl	2014.0	Color	R	149.0	David Fincher	21000.0
Sabrina, the Teenage Witch	NaN	Color	TV-G	22.0	NaN	NaN

15.3 3. Selecting Subsets of Data from DataFrames with `iloc`

15.3.1 Exercise 1

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

```
In [11]: movie.iloc[[10, 5, 1]]
```

Out[11]:

title	year	color	content_rating	duration	director_name
Batman v Superman: Dawn of Justice	2016.0	Color	PG-13	183.0	Zack Snyder
John Carter	2012.0	Color	PG-13	132.0	Andrew Stanton
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski

15.3.2 Exercise 2

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

```
In [12]: movie.iloc[:, [10, 5, 1]].head()
```

Out[12]:

title	actor3	director_fb	color
Avatar	Wes Studi	0.0	Color
Pirates of the Caribbean: At World's End	Jack Davenport	563.0	Color
Spectre	Stephanie Sigman	0.0	Color
The Dark Knight Rises	Joseph Gordon-Levitt	22000.0	Color
Star Wars: Episode VII - The Force Awakens	NaN	131.0	NaN

15.3.3 Exercise 3

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

```
In [13]: movie.iloc[100:105, 5]
```

Out[13]:

title	357.0
The Fast and the Furious	21000.0
The Curious Case of Benjamin Button	905.0
X-Men: First Class	508.0
The Hunger Games: Mockingjay - Part 2	226.0

```
Name: director_fb, dtype: float64
```

15.4 4. Selecting Subsets of Data - Series

15.4.1 Exercise 1

Read in the bikes dataset. We will be using it for the next few questions. Select the wind speed column as a Series and assign it to a variable and output the head. What kind of index does this Series have?

```
In [14]: bikes = pd.read_csv('../data/bikes.csv')
         bikes.head()
```

Out[14]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130

```
In [15]: wind = bikes['wind_speed']
         wind.head()
```

Out[15]:

0	12.7
1	6.9
2	16.1
3	16.1
4	17.3

```
Name: wind_speed, dtype: float64
```

This index is a **RangeIndex**

```
In [16]: wind.index
```

```
Out[16]: RangeIndex(start=0, stop=50089, step=1)
```

```
In [17]: wind.loc[4:10]
```

```
Out[17]:
4    17.3
5    17.3
6    15.0
7     5.8
8     0.0
9    12.7
10    9.2
Name: wind_speed, dtype: float64
```

15.4.2 Exercise 2

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

```
In [18]: wind.iloc[4:10]
```

```
Out[18]:
4    17.3
5    17.3
6    15.0
7     5.8
8     0.0
9    12.7
Name: wind_speed, dtype: float64
```

15.4.3 Exercise 3

Copy and paste your answer to Exercise 2 below but use `loc` instead. Do you get the same result? Why not?

```
In [19]: wind.loc[4:10]
```

```
Out[19]:
4    17.3
5    17.3
6    15.0
7     5.8
8     0.0
9    12.7
10    9.2
Name: wind_speed, dtype: float64
```

This is tricky - the index in this case contains integers and not strings. So the labels themselves are also integers and happen to be the same integers corresponding to integer location. The reason `.iloc` and `.loc` produce different results is that `.loc` always includes the last value when slicing.

15.4.4 Exercise 4

Read in the movie dataset and set the index to be the title. Select `actor1` as a Series. Who is the `actor1` for 'My Big Fat Greek Wedding'?

```
In [20]: movie = pd.read_csv('../data/movie.csv', index_col='title')
         actor1 = movie['actor1']
```

```
In [21]: actor1.loc['My Big Fat Greek Wedding']
```

```
Out[21]: 'Nia Vardalos'
```

15.4.5 Exercise 5

Find `actor1` for your favorite two movies?

```
In [22]: actor1.loc[['Titanic', 'Blood Diamond']]
```

Out[22]:

title	Leonardo DiCaprio
Titanic	Leonardo DiCaprio

```
Name: actor1, dtype: object
```

15.4.6 Exercise 6

Select the last 10 values from `actor1` using two different ways?

```
In [23]: actor1.iloc[-10:]
```

Out[23]:

title	Shane Carruth
Primer	Ian Gamazon
Cavite	Carlos Gallardo
El Mariachi	Richard Jewell
The Mongol King	Kerry Bishé
Newlyweds	Eric Mabius
Signed Sealed Delivered	Natalie Zea
The Following	Eva Boehnke
A Plague So Pleasant	Alan Ruck
Shanghai Calling	John August

```
Name: actor1, dtype: object
```

```
In [24]: actor1.tail(10)
```

Out[24]:

title	Shane Carruth
Primer	Ian Gamazon
Cavite	Carlos Gallardo
El Mariachi	Richard Jewell
The Mongol King	Kerry Bishé
Newlyweds	Eric Mabius
Signed Sealed Delivered	Natalie Zea
The Following	Eva Boehnke
A Plague So Pleasant	Alan Ruck
Shanghai Calling	John August

```
Name: actor1, dtype: object
```

15.5 5. Boolean Indexing Single Conditions

```
In [25]: import pandas as pd
```

```
In [26]: movie = pd.read_csv('../data/movie.csv', index_col='title')
         movie.head(3)
```

Out[26]:

title	year	color	content_rating	duration	director_name
Avatar	2009.0	Color	PG-13	178.0	James Cameron
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes

15.5.1 Exercise 1

Read in the movie dataset and set the index to be the title. Select all movies that have Tom Hanks as `actor1`. How many of these movies has he starred in?

```
In [27]: import pandas as pd
```

```
In [28]: filt = movie['actor1'] == 'Tom Hanks'
         hanks_movies = movie[filt]
         hanks_movies.head(3)
```

Out[28]:

title	year	color	content_rating	duration	director_name	director_fb	actor1
Toy Story 3	2010.0	Color	G	103.0	Lee Unkrich	125.0	Tom Hanks
The Polar Express	2004.0	Color	G	100.0	Robert Zemeckis	0.0	Tom Hanks
Angels & Demons	2009.0	Color	PG-13	146.0	Ron Howard	2000.0	Tom Hanks

He's starred in 24 movies

```
In [29]: hanks_movies.shape
```

```
Out[29]: (24, 21)
```

15.5.2 Exercise 2

Select movies with and IMDB score greater than 9.

```
In [30]: filt= movie['imdb_score'] > 9
         movie[filt]
```

Out[30]:

title	year	color	content_rating	duration	director_name	director_fb
The Shawshank Redemption	1994.0	Color	R	142.0	Frank Darabont	0.0
Towering Inferno	NaN	Color	NaN	65.0	John Blanchard	0.0
Dekalog	NaN	Color	TV-MA	55.0	NaN	NaN
The Godfather	1972.0	Color	R	175.0	Francis Ford Coppola	0.0
Kickboxer: Vengeance	2016.0	NaN	NaN	90.0	John Stockwell	134.0

15.6 6. Boolean Indexing Multiple Conditions

15.6.1 Exercise 1

Select all movies from the 1970s.

```
In [31]: filt1 = movie['year'] >= 1970
         filt2 = movie['year'] <= 1979
         filt = filt1 & filt2
         movie[filt].head()
```

Out[31]:

title	year	color	content_rating	duration	director_name
All That Jazz	1979.0	Color	R	123.0	Bob Fosse
Superman	1978.0	Color	PG	188.0	Richard Donner
Solaris	1972.0	Black and White	PG	115.0	Andrei Tarkovsky
Mean Streets	1973.0	Color	R	112.0	Martin Scorsese
Star Trek: The Motion Picture	1979.0	Color	PG	143.0	Robert Wise

15.6.2 Exercise 2

Select all movies from the 1970s that had IMDB scores greater than 8

```
In [32]: filt1 = movie['year'] >= 1970
         filt2 = movie['year'] <= 1979
         filt3 = movie['imdb_score'] > 8

         filt = filt1 & filt2 & filt3
         movie[filt].head(3)
```

Out[32]:

title	year	color	content_rating	duration	director_name	director_fb
Solaris	1972.0	Black and White	PG	115.0	Andrei Tarkovsky	0.0
Apocalypse Now	1979.0	Color	R	289.0	Francis Ford Coppola	0.0
The Deer Hunter	1978.0	Color	R	183.0	Michael Cimino	517.0

15.6.3 Exercise 3

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

```
In [33]: filt = movie['content_rating'].isin(['R', 'PG-13', 'PG'])
         movie[filt].head(3)
```

Out[33]:

title	year	color	content_rating	duration	director_name
Avatar	2009.0	Color	PG-13	178.0	James Cameron
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes

15.6.4 Exercise 4

Select movies that are either rated PG-13 or were made after 2010.

```
In [34]: filt1 = movie['content_rating'] == 'PG-13'
         filt2 = movie['year'] > 2010
         filt = filt1 | filt2

         movie[filt].head()
```

Out[34]:

title	year	color	content_rating	duration	director_name
Avatar	2009.0	Color	PG-13	178.0	James Cameron
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes
The Dark Knight Rises	2012.0	Color	PG-13	164.0	Christopher Nolan
John Carter	2012.0	Color	PG-13	132.0	Andrew Stanton

15.6.5 Exercise 5

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

```
In [35]: filt1 = movie['actor1_fb'] > 10000
         filt2 = movie['actor2_fb'] > 10000
         filt3 = movie['actor3_fb'] > 10000
         filt = filt1 | filt2 | filt3

         movie[filt].head()
```

Out[35]:

title	year	color	content_rating	duration	director_name
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes
The Dark Knight Rises	2012.0	Color	PG-13	164.0	Christopher Nolan
Spider-Man 3	2007.0	Color	PG-13	156.0	Sam Raimi
Avengers: Age of Ultron	2015.0	Color	PG-13	141.0	Joss Whedon

15.6.6 Exercise 6

Reverse the condition from Exercise 4. In words, what have you selected.

The following selects non-PG-13 movies made in the year 2010 or before.

```
In [36]: filt1 = movie['content_rating'] == 'PG-13'
         filt2 = movie['year'] > 2010
         filt = filt1 | filt2

         movie[~filt].head()
```

Out[36]:

title	year	color	content_rating	duration	director_name
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN	Doug Walker
Tangled	2010.0	Color	PG	100.0	Nathan Greno
Harry Potter and the Half-Blood Prince	2009.0	Color	PG	153.0	David Yates
The Chronicles of Narnia: Prince Caspian	2008.0	Color	PG	150.0	Andrew Adamson
Alice in Wonderland	2010.0	Color	PG	108.0	Tim Burton

15.7 7. Boolean Indexing More

```
In [37]: import pandas as pd
         bikes = pd.read_csv('../data/bikes.csv')
```

15.7.1 Exercise 1

Select the wind speed column a a Series and assign it to a variable. Are there any negative wind speeds?

```
In [38]: wind = bikes['wind_speed']
         wind.head()
```

```
Out[38]:
0    12.7
1     6.9
2    16.1
3    16.1
4    17.3
Name: wind_speed, dtype: float64
```

Yes, there is really strong negative wind! Or maybe its just bad data...

```
In [39]: filt = wind < 0
         wind[filt].head()
```

```
Out[39]:
22990   -9999.0
27168   -9999.0
28368   -9999.0
29308   -9999.0
29309   -9999.0
Name: wind_speed, dtype: float64
```

15.7.2 Exercise 2

Select all wind speed between 12 and 16.

```
In [40]: filt = wind.between(12, 16)
         wind[filt].head()
```

```
Out[40]:
0    12.7
6    15.0
9    12.7
18   13.8
19   13.8
Name: wind_speed, dtype: float64
```

15.7.3 Exercise 3

Select the events and gender columns for all trip durations longer than 1,000 seconds.

```
In [41]: filt = bikes['tripduration'] > 1000
         cols = ['events', 'gender']
         bikes.loc[filt, cols].head()
```

Out[41]:

	events	gender
2	mostlycloudy	Male
8	cloudy	Male
10	mostlycloudy	Male
11	mostlycloudy	Male
12	partlycloudy	Male

15.7.4 Exercise 4

Read in the movie dataset with the title as the index. We will use this DataFrame for the rest of the Exercises. Select all the movies such that the Facebook likes for actor 2 are greater than those for actor 1.

```
In [42]: movie = pd.read_csv('../data/movie.csv', index_col='title')
         movie.head()
```

Out[42]:

title	year	color	content_rating	duration
Avatar	2009.0	Color	PG-13	178.0
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0
Spectre	2015.0	Color	PG-13	148.0
The Dark Knight Rises	2012.0	Color	PG-13	164.0
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN

There are none!

```
In [43]: filt = movie['actor2_fb'] > movie['actor2_fb']
         movie[filt]
```

Out[43]:

Empty DataFrame Columns: Index(['year', 'color', 'content_rating', 'duration', 'director_name', 'director_fb', 'actor1

15.7.5 Exercise 5

Select the year, content rating, and IMDB score columns for movies from the year 2016 with IMDB score less than 4.

```
In [44]: filt1 = movie['year'] == 2016
         filt2 = movie['imdb_score'] < 4
         filt = filt1 & filt2
         cols = ['year', 'content_rating', 'imdb_score']

         movie.loc[filt, cols]
```

Out[44]:

title	year	content_rating	imdb_score
Fifty Shades of Black	2016.0	R	3.5
Cabin Fever	2016.0	Not Rated	3.7
God's Not Dead 2	2016.0	PG	3.4

15.7.6 Exercise 6

Select all the movies that are missing values for content rating.

```
In [45]: filt = movie['content_rating'].isna()
         movie[filt].head(3)
```

Out[45]:

title	year	color	content_rating	duration	director_name
Star Wars: Episode VII - The Force Awakens	NaN	NaN	NaN	NaN	Doug Walker
Godzilla Resurgence	2016.0	Color	NaN	120.0	Hideaki Anno
Harry Potter and the Deathly Hallows: Part II	2011.0	Color	NaN	NaN	Matt Birch

15.7.7 Exercise 7

Select all the movies that are missing both the gross and budget. Return just those columns to verify that those values are indeed missing.

```
In [46]: filt = movie['gross'].isna() & movie['budget'].isna()
         cols = ['gross', 'budget']
         movie.loc[filt, cols].head()
```

Out[46]:

title	gross	budget
Star Wars: Episode VII - The Force Awakens	NaN	NaN
The Lovers	NaN	NaN
Godzilla Resurgence	NaN	NaN
Harry Potter and the Deathly Hallows: Part II	NaN	NaN
Harry Potter and the Deathly Hallows: Part I	NaN	NaN

15.7.8 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 Exercise 7 with this function.

```
In [47]: def find_missing(df, col1, col2):
             filt = df[col1].isna() & df[col2].isna()
             cols = [col1, col2]

             return df.loc[filt, cols]
```

```
In [48]: movie_missing = find_missing(movie, 'gross', 'budget')
         movie_missing.head()
```

Out[48]:

title	gross	budget
Star Wars: Episode VII - The Force Awakens	NaN	NaN
The Lovers	NaN	NaN
Godzilla Resurgence	NaN	NaN
Harry Potter and the Deathly Hallows: Part II	NaN	NaN
Harry Potter and the Deathly Hallows: Part I	NaN	NaN

15.8 8. Miscellaneous Subset Selection

```
In [49]: import pandas as pd
         bikes = pd.read_csv('../data/bikes.csv')
```

15.8.1 Exercise 1

Use the query method to select trip durations between 5000 and 10000 when it was partlycloudly or mostlycloudly. Create a set to contain the possible events and assign it a some variable name. Reference this variable within the query string. Then redo the operation again using boolean selection.

```
In [50]: event_types = {"partlycloudy", "mostlycloudy"}
         bikes.query('5000 <= tripduration <= 10000 and events in @event_types').head()
```

Out[50]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299
3406	1715861	Subscriber	Male	2014-05-19 09:26:00	2014-05-19 10:50:00	5059
3476	1744550	Subscriber	Male	2014-05-21 13:05:00	2014-05-21 15:11:00	7592
4046	2024090	Subscriber	Male	2014-06-08 17:31:00	2014-06-08 18:54:00	5020

```
In [51]: filt1 = bikes['tripduration'].between(5000, 10000)
         filt2 = bikes['events'].isin(event_types)
         filt = filt1 & filt2
         bikes[filt].head()
```

Out[51]:

	trip_id	usertype	gender	starttime	stoptime	tripduration
40	61401	Subscriber	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274
77	87005	Subscriber	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299
3406	1715861	Subscriber	Male	2014-05-19 09:26:00	2014-05-19 10:50:00	5059
3476	1744550	Subscriber	Male	2014-05-21 13:05:00	2014-05-21 15:11:00	7592
4046	2024090	Subscriber	Male	2014-06-08 17:31:00	2014-06-08 18:54:00	5020

```
In [52]: bikes.trip_id.between
```

```
Out[52]: <bound method Series.between of 0           7147
         1            7524
         2           10927
         3           12907
         4           13168
         5           13595
         6           18880
         7           19689
         8           21028
         9           23558
         10          24383
         11          24673
         12          26214
         13          30404
         14          31121
         15          33998
         16          38628
```

```
17          38729
18          40924
19          40879
20          42488
21          42818
22          43804
23          45077
24          46108
25          47798
26          51130
27          51210
28          51823
29          52547
           ...
50059    17525739
50060    17525743
50061    17526542
50062    17527722
50063    17528311
50064    17528461
50065    17528527
50066    17528614
50067    17529175
50068    17529292
50069    17529402
50070    17530881
50071    17530946
50072    17531067
50073    17531700
50074    17532063
50075    17532624
50076    17533419
50077    17533484
50078    17533656
50079    17533757
50080    17534057
50081    17534131
50082    17534773
50083    17534831
50084    17534938
50085    17534969
50086    17534972
50087    17535645
50088    17536246
Name: trip_id, Length: 50089, dtype: int64>
```