Master Data Analysis with Python - Intro to Pandas

by
Teddy Petrou

Contents

Part I

Intro to Pandas

Chapter 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, query, transform, aggregate, and visualize **tabular** data. Tabular refers to data that is two-dimensional, consisting of rows and columns. Commonly, we refer to this organized structure of data as a **table**. pandas is the tool that we will use to analyze data in nearly every chapter of this book.

1.1 Why pandas and not xyz?

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

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

Why is it named after an East Asian bear?

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

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

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

pandas is built directly on numpy

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

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

1.2 pandas operates on tabular data

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

The DataFrame and Series

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

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

1.3 pandas examples

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

- Reading data
- Filtering data
- Aggregating methods

- Non-Aggregating methods
- Aggregating within groups
- Cleaning data
- Joining data
- Time series analysis
- Visualization

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

The `head` method

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

1.4 Which pandas version to use?

The pandas library is under constant development and regularly releases new versions. Currently, pandas is on major version **1**, which was released in January, 2020. Before major version 1, pandas was on version **0**. Python libraries use the form **a.b.c** for version numbering where **a** represents the **major** version number. It is increased whenever there are major changes, with some being backward incompatible. **b** represents the **minor** version number and is incremented for smaller backward-compatible changes and enhancements. **c** represents the **micro** version number and is incremented mainly for bug fixes.

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

Import pandas and verify version number

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

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

[1]:
```
'1.3.4'
```

1.5 Reading data

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

3 rows × 11 columns

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 5,000 (seconds).

Single Condition

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy

3 rows × 11 columns

Multiple Conditions

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy
1954	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	LaSalle St & Washington St	...	Theater on the Lake	15.0	44.1	12.7	clear

3 rows × 11 columns

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
9	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	Lakeview Ave & Fullerton Pkwy	...	Racine Ave & Congress Pkwy	19.0	81.0	12.7	mostlycloudy
14	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	Morgan St & Lake St	...	Aberdeen St & Jackson Blvd	15.0	82.0	5.8	mostlycloudy
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy

3 rows × 11 columns

Using the `query` method

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy

3 rows × 11 columns

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy
1954	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	LaSalle St & Washington St	...	Theater on the Lake	15.0	44.1	12.7	clear

3 rows × 11 columns

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
9	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	Lakeview Ave & Fullerton Pkwy	...	Racine Ave & Congress Pkwy	19.0	81.0	12.7	mostlycloudy
14	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	Morgan St & Lake St	...	Aberdeen St & Jackson Blvd	15.0	82.0	5.8	mostlycloudy
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy

3 rows × 11 columns

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`, `max`, and `min` are all examples of aggregation methods. By default, calling these methods on a pandas DataFrame applies the aggregation to each column. Below, we use a dataset containing San Francisco employee compensation information. Only a subset of the columns are initially read into the DataFrame.

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

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

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

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

[10]:
```
salaries             53715.441133
overtime              4201.272687
other salaries        2816.296542
retirement           10484.755614
health and dental     9382.390735
dtype: float64
```

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

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

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

1.8 Non-aggregating methods

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

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

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

1.9 Aggregating within groups

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

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

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

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

Frequency of unique values in a single column

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

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

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

Single aggregation function

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

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

	mean_charges
sex	
female	13000.0
male	14000.0

Multiple aggregation functions

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

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

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

Multiple grouping columns

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

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

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

Pivot Tables

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

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

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

Styling DataFrames

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

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

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

1.10 Cleaning data

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

Options in the `read_csv` function

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

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

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

3 rows × 13 columns

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

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

	date	time	location	operator	flight_no	...	cn_ln	aboard	fatalities	ground	summary
0	September 17, 1908	17:18	Fort Myer, Virginia	Military - U.S. Army	NaN	...	1	2 (passengers:1 crew:1)	1 (passengers:1 crew:0)	0.0	During a demonstration flight, a U.S. Army fly...
1	September 07, 1909	NaN	Juvisy-sur-Orge, France	NaN	NaN	...	NaN	1 (passengers:0 crew:1)	1 (passengers:0 crew:0)	0.0	Eugene Lefebvre was the first pilot to ever be...
2	July 12, 1912	06:30	Atlantic City, New Jersey	Military - U.S. Navy	NaN	...	NaN	5 (passengers:0 crew:5)	5 (passengers:0 crew:5)	0.0	First U.S. dirigible Akron exploded just offsh...

3 rows × 13 columns

String manipulation

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

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

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

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

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

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

Reshaping into tidy form

Occasionally, you will have several columns of data that all belong in a single column. Take a look at the DataFrame below on the average arrival delay of airlines at different airports. All of the columns

with three-letter airport codes could be placed in the same column as they all contain the arrival delay which has the same units.

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

	airline	ATL	IAH	LAX	SFO
0	AA	4	11	3	3
1	DL	0	3	3	0
2	UA	5	9	4	5

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

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

	airline	airport	delay
0	AA	ATL	4
1	DL	ATL	0
2	UA	ATL	5
3	AA	IAH	11
4	DL	IAH	3
5	UA	IAH	9
6	AA	LAX	3
7	DL	LAX	3
8	UA	LAX	4
9	AA	SFO	3

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.

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

Output the first three rows of each DataFrame.

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

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

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

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

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

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

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

1.12 Time Series Analysis

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

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

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

Select a period of time

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

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

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

Group by time

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

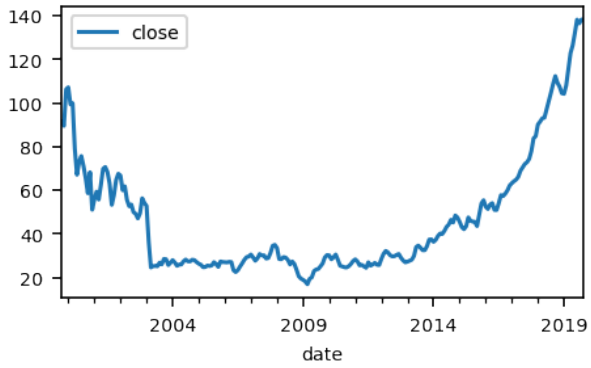
[32]:
```
msft_mc = msft.resample('M').agg({'close':'mean'})
msft_mc.head(3)
```

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

1.13 Visualization

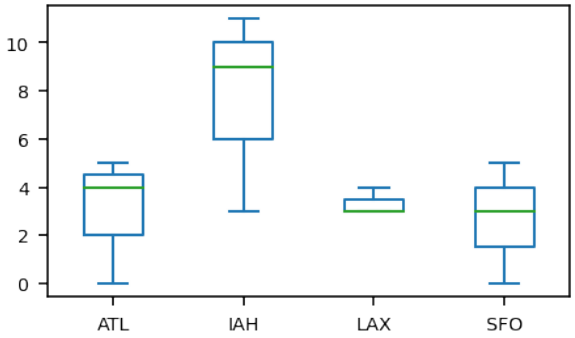
pandas provides basic visualization abilities by giving its users a few default plots. Below, we plot the average monthly closing price of Microsoft for the last 20 years.

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



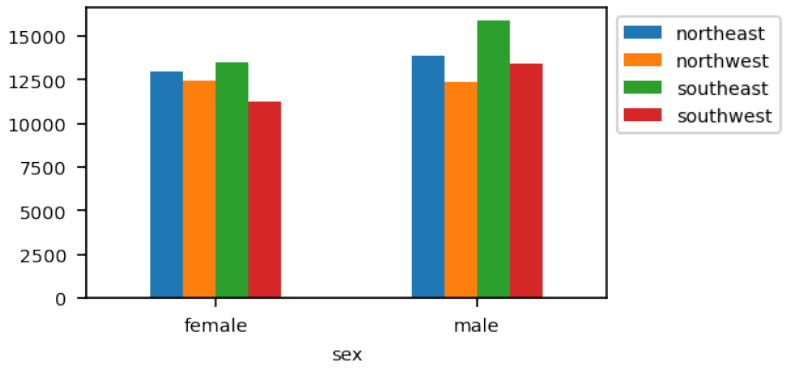
Below, we create a box plot of the average arrival delay by airport.

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



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

[35]:
```
pt.plot(kind='bar', rot=0).legend(bbox_to_anchor=(1, 1), loc='upper left');
```



1.14 Much More

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

Chapter 2

The DataFrame and Series

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

2.1 Reading external data with pandas

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

City of Chicago bike rides

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

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

```
gender,starttime,stoptime,tripduration,from_station_name,start_capacity,to_station_name,e
nd_capacity,temperature,wind_speed,events

Male,2013-06-28 19:01:00,2013-06-28 19:17:00,993,Lake Shore Dr & Monroe St,11.
↪0,Michigan
Ave & Oak St,15.0,73.9,12.7,mostlycloudy
```

```
Male,2013-06-28 22:53:00,2013-06-28 23:03:00,623,Clinton St & Washington Blvd,31.
↪0,Wells
St & Walton St,19.0,69.1,6.9,partlycloudy
```

Understanding the file location

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

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

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

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

Import pandas

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

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

Reading a CSV with the `read_csv` function

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

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

Display DataFrame in Jupyter Notebook

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

`head` and `tail` methods

A very useful and simple method is `head`, which returns the first 5 rows of the DataFrame by default. This avoids long default output and is something I highly recommend when doing data analysis within

a notebook. The `tail` method returns the last 5 rows by default. There will only be a few instances in the book where the `head` method is not used, as displaying up to 60 rows is far too many and will take up a lot of space on a screen or page.

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	...	Clark St & Randolph St	31.0	72.0	16.1	mostlycloudy
4	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130	Damen Ave & Pierce Ave	...	Damen Ave & Pierce Ave	19.0	73.0	17.3	partlycloudy

5 rows × 11 columns

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

[5]:
```
bikes.tail()
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
50084	Male	2017-12-30 13:07:00	2017-12-30 13:34:00	1625	State St & Pearson St	...	Clark St & Elm St	27.0	5.0	16.1	partlycloudy
50085	Male	2017-12-30 13:34:00	2017-12-30 13:44:00	585	Halsted St & 35th St (*)	...	Union Ave & Root St	11.0	5.0	16.1	partlycloudy
50086	Male	2017-12-30 13:34:00	2017-12-30 13:48:00	824	Kingsbury St & Kinzie St	...	Halsted St & Blackhawk St (*)	20.0	5.0	16.1	partlycloudy
50087	Female	2017-12-31 09:30:00	2017-12-31 09:33:00	178	Clinton St & Lake St	...	Kingsbury St & Kinzie St	31.0	7.0	11.5	partlycloudy
50088	Male	2017-12-31 15:22:00	2017-12-31 15:26:00	214	Clarendon Ave & Leland Ave	...	Clifton Ave & Lawrence Ave	15.0	10.9	15.0	partlycloudy

5 rows × 11 columns

First and last `n` rows

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

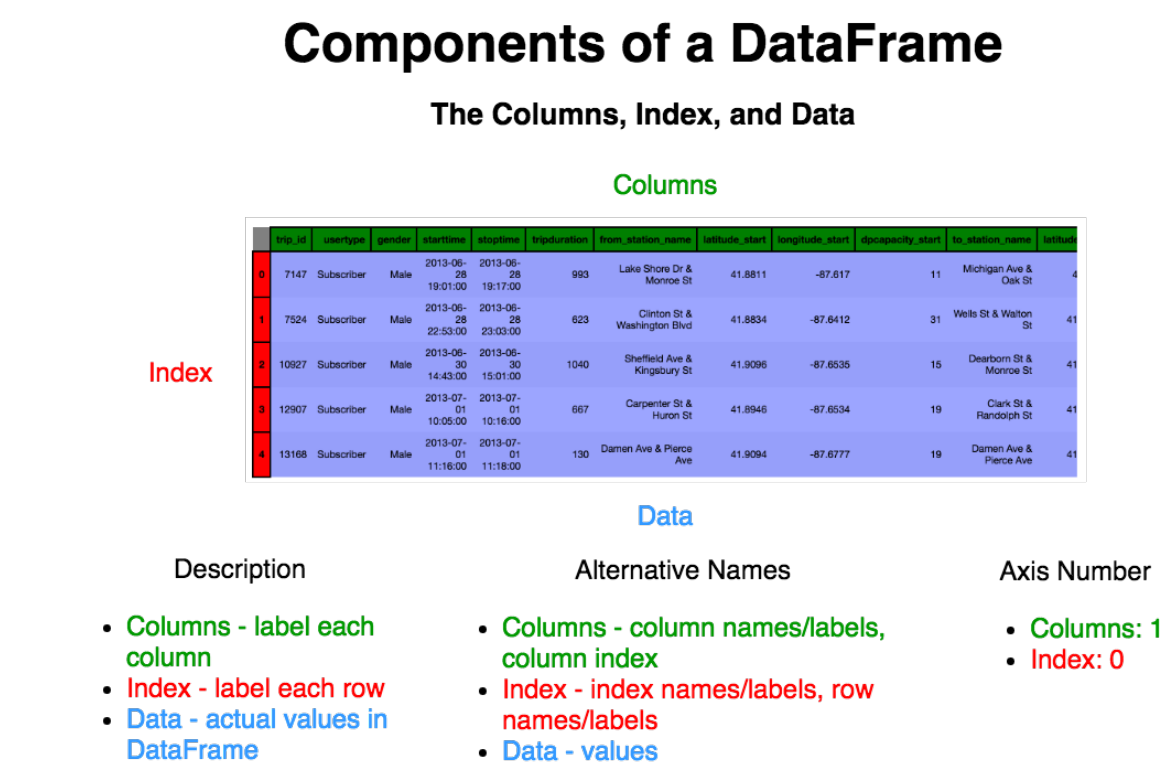
[6]:
```
bikes.head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

3 rows × 11 columns

2.2 Components of a DataFrame

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



The columns

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

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

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

The index

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

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

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

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

The data

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

The Axes

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

2.3 What type of object is `bikes`?

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

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

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

Fully-qualified name

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

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

Package vs Module

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

Sub-packages

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

Where are the packages located on my machine?

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

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

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

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

2.4 Select a single column from a DataFrame - a Series

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

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

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

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

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

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

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

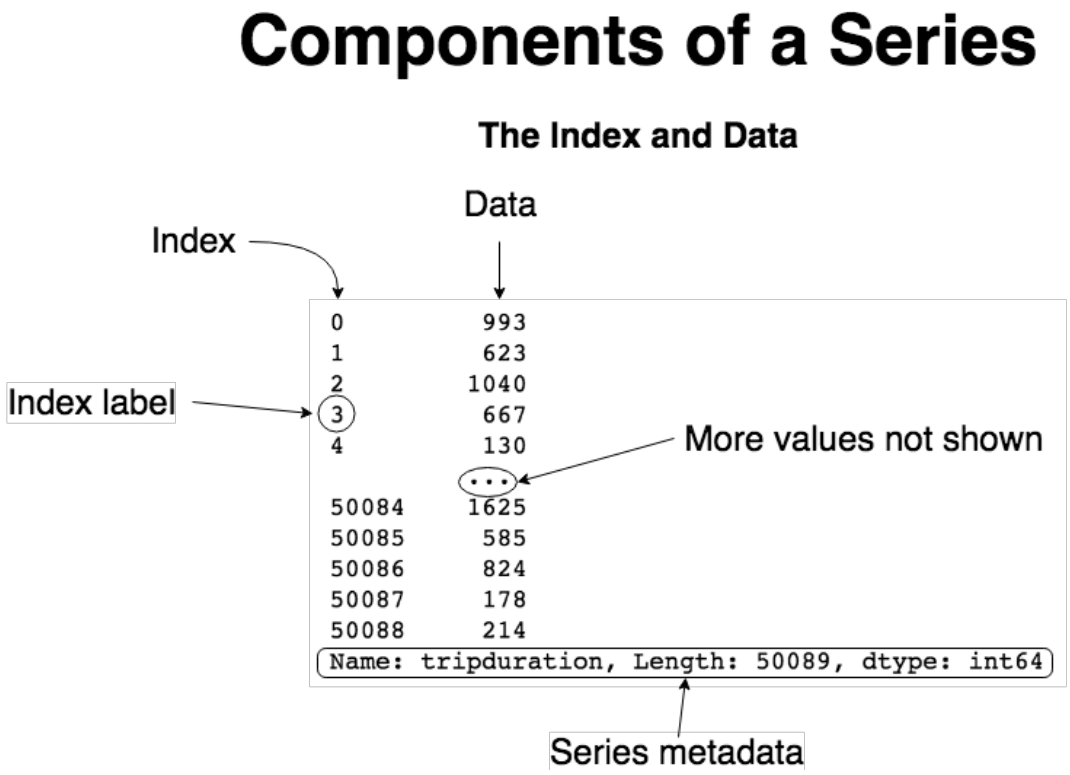
Let's verify that `td` has the type Series.

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

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

2.5 Components of a Series

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



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

The index

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

Output of Series vs DataFrame

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

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

2.6 Changing display options

pandas gives you the ability to change how the output on your screen is displayed. For instance, the default number of columns displayed for a DataFrame is 20, meaning that if your DataFrame has more

than 20 columns, then only the first and last 10 columns will be shown on the screen. All the other columns will be hidden and unable to be displayed. This is problematic as many DataFrames have more than 20 columns.

Get current option value with `get_option`

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

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

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

[12]: 10

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

[13]: 60

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

[14]: 50

Use the `set_option` function to change an option value

To change an option's 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 100 and the maximum number of rows to 4.

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

We now read in the housing dataset which contains 81 columns, all of which will be visible. Uncomment the lines to run them in your notebook.

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

2.7 Exercises

Use the `bikes` DataFrame for the following exercises.

Exercise 1

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

[17]:

Exercise 2

What type of object is `events`?

[18]:

Exercise 3

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

[19]:

Exercise 4

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

[20]:

Chapter 3

Data Types and Missing Values

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

3.1 Common data types

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

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

The importance of knowing the data type

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

The exception with the object data type

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

3.2 String data type - major enhancement to pandas 1.0

Before the release of pandas version 1.0, there was no dedicated string data type. This was a huge limitation and caused numerous problems. pandas still has the 'object' data type, which is capable of holding strings.

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

3.3 Missing value representation

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

- `NaN` - "Not a Number"
- `NaT` - "Not a Time"

Missing values for each data type

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

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

Missing values in boolean and integer columns

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

3.4 New Integers and booleans data types in pandas 1.0

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

Pandas NA - A new missing value representation for pandas 1.0

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

3.5 Recommendation for Pandas 1.0 - Avoid the new data types

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

3.6 Finding the data type of each column

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

3 rows × 11 columns

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

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

[2]:
```
gender                object
starttime             object
stoptime              object
tripduration           int64
from_station_name     object
start_capacity       float64
to_station_name       object
end_capacity         float64
temperature          float64
wind_speed           float64
events                object
dtype: object
```

Object data types hold string columns

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

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

From the visual display of the bikes DataFrame above, it appears that both the `starttime` and `stoptime` columns are datetimes. However, the result of the `dtypes` attribute shows that they are

strings. Unfortunately, the `read_csv` function does not automatically read in these columns as datetimes. It requires that you provide it a list of columns that are datetimes to the `parse_dates` parameter, otherwise it will read them in as strings. Let's reread the data using the `parse_dates` parameter.

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

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

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

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

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

3.7 Getting more metadata

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

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

[4]:
```
(50089, 11)
```

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

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

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

[5]:
```
50089
```

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

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

[6]:
```
50089
```

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

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

[7]:
```
11
```

Total number of values with the `size` attribute

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

[8]:
```
bikes.size
```

[8]:
```
550979
```

Get data types plus more with the `info` method

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

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

The information is printed to the screen. It does not return any object.

[9]:
```
bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   gender             50089 non-null  object
 1   starttime          50089 non-null  datetime64[ns]
 2   stoptime           50089 non-null  datetime64[ns]
 3   tripduration       50089 non-null  int64
 4   from_station_name  50089 non-null  object
 5   start_capacity     50083 non-null  float64
 6   to_station_name    50089 non-null  object
 7   end_capacity       50077 non-null  float64
 8   temperature        50089 non-null  float64
 9   wind_speed         50089 non-null  float64
 10  events             50089 non-null  object
dtypes: datetime64[ns](2), float64(4), int64(1), object(4)
memory usage: 4.2+ MB
```

3.8 More data types

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

3.9 Exercises

Use the `bikes` DataFrame for the following:

Exercise 1

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

[10]:

Exercise 2

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

[11]:

Exercise 3

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.

[12]:

Chapter 4

Setting a Meaningful Index

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

4.1 Setting an index of a DataFrame

Instead of using the default index for your pandas DataFrame, you can call the `set_index` method to use one of the columns as the index. Let's read in a small dataset to show how this is done. Note the current index is just consecutive integers beginning from 0.

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

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

The `set_index` method

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

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

name	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

A new DataFrame copy is returned

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

[3]:
```
df
```

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

Assigning the result of `set_index` to a variable name

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

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

name	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

Number of columns decreased

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

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

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

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

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

4.2 Accessing the index, columns, and data

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

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

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

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

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

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

[9]:
```
array([['NY', 'blue', 'Steak', 30, 165, 4.6],
       ['TX', 'green', 'Lamb', 2, 70, 8.3],
       ['FL', 'red', 'Mango', 12, 120, 9.0],
       ['AL', 'white', 'Apple', 4, 80, 3.3],
       ['AK', 'gray', 'Cheese', 32, 180, 1.8],
       ['TX', 'black', 'Melon', 33, 172, 9.5],
       ['TX', 'red', 'Beans', 69, 150, 2.2]], dtype=object)
```

Find the type of these objects

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

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

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

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

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

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

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

Accessing the components does not change the DataFrame

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

[13]:
```
df2
```

name	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

pandas `Index` type

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

Two-dimensional numpy array

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

Operating with the DataFrame and not its components

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

4.3 Accessing the components of a Series

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

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

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

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

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

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

In a Series, the values are stored as a 1-dimensional numpy array.

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

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

The default index

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

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

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

3 rows × 22 columns

Integers in the index

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

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

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

We can also verify its type.

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

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

The RangeIndex

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

A numpy array underlies the index

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

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

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

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

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

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

4.4 Setting an index on read

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

Reread the movie dataset with the movie title as the index

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

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

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

3 rows × 21 columns

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

Access the new index and output its type

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

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

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

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

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

Select a value from the index

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

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

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

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

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

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

Selection with slice notation

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

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

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

Selection with a list of integers

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

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

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

4.5 Choosing a good index

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

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

Verifying uniqueness in the index

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

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

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

3 rows × 21 columns

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

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

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

4.6 Exercises

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

Exercise 1

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

[31]:
```

```

Exercise 2

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

[32]:

Exercise 3

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

[33]:

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?

[34]:

Chapter 5

Five-Step Process for Data Exploration

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

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

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

Apply to every part of the analysis

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

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

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

3 rows × 22 columns

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

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

[2]:
```
movie.set_index('title').head(3)
```

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

3 rows × 21 columns

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

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

Step 3: Assign the result to a variable

You would normally do this step in the same cell, but for this demonstration, we will place it in the cell below. Here, we assign the above operation to new variable name `movie2`.

[3]:
```
movie2 = movie.set_index('title')
```

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

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

[4]:
```
movie2.head(3)
```

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

3 rows × 21 columns

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

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

All steps in one cell

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

[5]:
```
movie2 = movie.set_index('title')
movie2.head(3)
```

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

3 rows × 21 columns

More examples

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

[6]:
```
avg_duration = movie.groupby('year').agg({'duration':'mean'})
avg_duration.head(3)
```

	duration
year	
1916.0	123.0
1920.0	110.0
1925.0	151.0

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

[7]:
```
top_years = avg_duration.sort_values('duration', ascending=False)
top_years.head(3)
```

	duration
year	
1963.0	153.875000
1925.0	151.000000
1939.0	149.333333

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

No strict requirement for one line of code

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

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

[8]:
```
cols = ['year', 'duration']
movie3 = movie[cols]
movie3.head(3)
```

	year	duration
0	2009.0	178.0
1	2007.0	169.0
2	2015.0	148.0

When to assign the result to a variable

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

When to create a new variable name

During step 3 of the first example, the result of our new dataset was assigned to `movie2`. We could have reassigned the result back to `movie` and continued on with our analysis. When first exploring data, I recommend creating a new variable name for each major result. By doing so, you have preserved each step of your work and are able to inspect it at a later date. Creating new variables makes it much easier to find errors at different places in your analysis.

When to reuse variable names

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

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

Continuously verifying results

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

Part II

Selecting Subsets of Data

Chapter 6

Selecting Subsets of Data from DataFrames with Just the Brackets

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

Examples of selections of subsets of data

The following images show different types of subset selection that are possible. The DataFrame on the left is the original with the selection highlighted. The DataFrame on the right is the result of the selection.

Selecting columns

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

Selection

name	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

Result

name	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

Selecting rows

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

Selection

name	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

Result

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

Simultaneous row and column selection

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

Selection

name	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

Result

name	color	age	height
Aaron	red	12	120
Dean	gray	32	180

6.1 pandas dual references - by label and by integer location

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

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

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

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

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

6.2 The three indexers [ ], loc, iloc

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

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

Terminology

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

Square brackets instead of parentheses

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

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

6.3 Begin with *just the brackets*

As we saw in a previous chapter, just the brackets are used to select a single column as a Series. We place the column name inside the brackets to return the Series. Let's read in a simple, small DataFrame and select a single column from it. We use the `index_col` parameter to set the index to the first column (integer location 0) on read.

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

name	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

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

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

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

6.4 Select multiple columns with a list

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

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

name	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

You must use an inner set of brackets

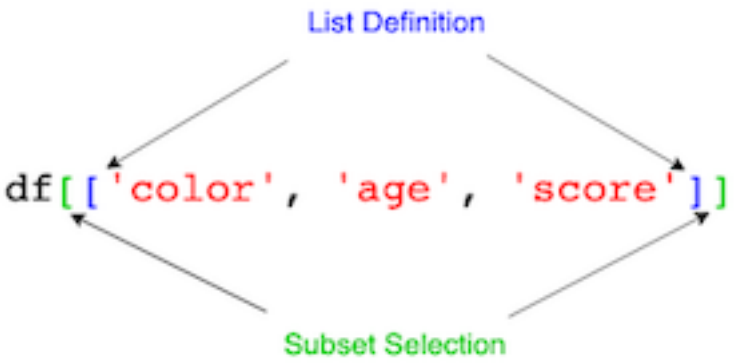
You might be tempted to do the following, which will NOT work. When selecting multiple columns, you must use a **list** to contain the names. Remember, that a list is defined by a set of square brackets, so the following raises an error.

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

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

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

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



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

Use two lines of code to select multiple columns

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

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

name	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

Columns in any order

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

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

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

6.5 Summary of *just the brackets*

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

6.6 Exercises

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

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

title	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb	actor2	actor2_fb	actor3	actor3_fb	gross	genres	num_reviews	num_voted_users	plot_keywords	language	country	budget	imdb_score
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0	Joel David Moore	936.0	Wes Studi	855.0	760505847.0	Action|Adventure|Fantasy|Sci-Fi	723.0	886204	avatar|future|marine|native|paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	Johnny Depp	40000.0	Orlando Bloom	5000.0	Jack Davenport	1000.0	309404152.0	Action|Adventure|Fantasy	302.0	471220	goddess|marriage ceremony|marriage proposa|pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	Christoph Waltz	11000.0	Rory Kinnear	393.0	Stephanie Sigman	161.0	200074175.0	Action|Adventure|Thriller	602.0	275868	bomb|espionage|sequel|spy|terrorist	English	UK	245000000.0	6.8

Exercise 1

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

[8]:
```

```

Exercise 2

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

[9]:
```

```

Exercise 3

Select a single column as a DataFrame and not a Series

[10]:
```

```

Exercise 4

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

[11]:

Chapter 7

Selecting Subsets of Data from DataFrames with `loc`

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

7.1 Simultaneous row and column subset selection

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

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

Just the brackets cannot do this

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

`loc` primarily selects data by label

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

Read in data

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

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

name	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

Select two rows and three columns with `loc`

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

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

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

The possible types of row and column selections

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

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

Select two rows and a single column

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

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

[3]:
```
name
Dean      Cheese
Aaron      Mango
Name: food, dtype: object
```

Series returned

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

7.2 loc with slice notation

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

Example slices

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

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

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

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

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

Slice notation is inclusive of the stop label

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

Slice notation only works within the brackets attached to the object

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

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

```
  File "/var/folders/0x/whw882fj4qv0czzqzrngxvdh0000gn/T/ipykernel_41428/16
71112888.py", line 4
    rows = 'Jane':'Penelope'
                  ^
SyntaxError: invalid syntax
```

Slice both the rows and columns

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

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

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

Selecting all of the rows and some of the columns

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

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

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

Could have used *just the brackets*

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

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

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

A single colon is slice notation to select all values

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

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

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

Use a single colon to select all the columns

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

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

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

The above can be shortened

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

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

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

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

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

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

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

name	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

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

Changing the step size

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

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

name	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

Select a single row and a single column

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

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

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

Select a single row as a Series with `loc`

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

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

[15]:
```
state        TX
color     green
```

```
food       Lamb
age           2
height       70
score       8.3
Name: Niko, dtype: object
```

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

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

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

Confusing output

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

Selecting a single row as a DataFrame

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

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

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

7.3 Summary of the `loc` indexer

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

7.4 Exercises

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

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

title	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb	actor2	actor2_fb	actor3	actor3_fb	gross	genres	num_reviews	num_voted_users	plot_keywords	language	country	budget	imdb_score
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0	Joel David Moore	936.0	Wes Studi	855.0	760505847.0	Action|Adventure|Fantasy|Sci-Fi	723.0	886204	avatar|future|marine|native|paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	Johnny Depp	40000.0	Orlando Bloom	5000.0	Jack Davenport	1000.0	309404152.0	Action|Adventure|Fantasy	302.0	471220	goddess|marriage ceremony|marriage proposal|pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	Christoph Waltz	11000.0	Rory Kinnear	393.0	Stephanie Sigman	161.0	200074175.0	Action|Adventure|Thriller	602.0	275868	bomb|espionage|sequel|spy|terrorist	English	UK	245000000.0	6.8

Exercise 1

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

[19]:
```

```

Exercise 2

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

[20]:
```

```

Exercise 3

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

[21]:
```

```

Exercise 4

Repeat exercise 3, but return a DataFrame instead.

[22]:
```

```

Exercise 5

Select all columns for the movie 'The Dark Knight Rises'.

[23]:
```

```

Exercise 6

Repeat exercise 5 but return a DataFrame instead.

[24]:
```

```

Exercise 7

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

[25]:

Exercise 8

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

[26]:

Exercise 9

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

[27]:

Exercise 10

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

[28]:

Exercise 11

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

[29]:

Chapter 8

Selecting Subsets of Data from DataFrames with `iloc`

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

8.1 Simultaneous row and column subset selection

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

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

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

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

name	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

What is integer location?

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

Select using a list for both rows and columns

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

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

name	state	score
Aaron	FL	9.0
Dean	AK	1.8

The possible types of selections for `iloc`

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

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

Slice the rows and use a list for the columns

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

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

name	height	food
Aaron	120	Mango
Penelope	80	Apple

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

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

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

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

Select all of the rows and some of the columns

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

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

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

Cannot do this with *just the brackets*

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

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

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

Select some of the rows and all of the columns

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

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

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

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

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

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

Select a single row and a single column

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

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

[9]:
```
'Apple'
```

Select a single row and a single column as a DataFrame

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

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

name	food
Penelope	Apple

Select some rows and a single column

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

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

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

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

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

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

name	height
Aaron	120
Penelope	80
Christina	172

Select a single row as a Series

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

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

[13]:
```
state        FL
color       red
food      Mango
age          12
height      120
score       9.0
Name: Aaron, dtype: object
```

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

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

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

8.2 Summary of `iloc`

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

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

8.3 Exercises

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

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

title	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb	actor2	actor2_fb	actor3	actor3_fb	gross	genres	num_reviews	num_voted_users	plot_keywords	language	country	budget	imdb_score
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0	Joel David Moore	936.0	Wes Studi	855.0	760505847.0	Action|Adventure|Fantasy|Sci-Fi	723.0	886204	avatar|future|marine|native|paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	Johnny Depp	40000.0	Orlando Bloom	5000.0	Jack Davenport	1000.0	309404152.0	Action|Adventure|Fantasy	302.0	471220	goddess|marriage ceremony|marriage proposal|pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	Christoph Waltz	11000.0	Rory Kinnear	393.0	Stephanie Sigman	161.0	200074175.0	Action|Adventure|Thriller	602.0	275868	bomb|espionage|sequel|spy|terrorist	English	UK	245000000.0	6.8

Exercise 1

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

[16]:
```

```

Exercise 2

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

[17]:
```

```

Exercise 3

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

[18]:
```

```

Exercise 4

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

[19]:
```

```

Exercise 5

Return the result of exercise 4 as a DataFrame.

[20]:
```

```

Exercise 6

Select the last 5 rows of the last 5 columns.

[21]:
```

```

Exercise 7

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

[22]:
```

```

Exercise 8

Select the column with integer location 7 as a Series.

[23]:

Exercise 9

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

[24]:

Chapter 9

Selecting Subsets of Data from a Series

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

9.1 Series indexer rules

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

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

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

3 rows × 21 columns

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

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

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

Series subset selection with just the brackets

For DataFrames, we learned that *just the brackets* accepted either a single label or a list of labels and used this input to select one or more DataFrame columns. For a Series, *just the brackets* has different rules that you must follow to use it correctly. It allows selection by index label. For instance, we can select the `imdb_score` for the movie Avatar like this:

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

[3]:
```
7.9
```

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

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

[4]:
```
7.9
```

9.2 Use `loc` and `iloc` instead of just the brackets

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

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

9.3 Series subset selection with `loc`

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

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

Select a single value with `loc`

Select a single value by providing the `loc` indexer the name of the index. Here, we select the `imdb_score` of the movie Forrest Gump. When selecting a single value, just that value is returned and not a Series.

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

[5]:
```
8.8
```

Select multiple values using a list with `loc`

Provide the `loc` indexer a list of index labels to select multiple values. This will return a Series.

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

[6]:
```
title
Good Will Hunting    8.3
Home Alone           7.5
```

```
Meet the Parents     7.0
Name: imdb_score, dtype: float64
```

Select multiple values using slice notation with `loc`

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

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

[7]:
```
title
Home Alone          7.5
3 Men and a Baby    5.9
Tootsie             7.4
Top Gun             6.9
Name: imdb_score, dtype: float64
```

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

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

[8]:
```
title
Twins                      6.0
Scream: The TV Series      7.3
The Yellow Handkerchief    6.8
The Color Purple           7.8
Tidal Wave                 5.7
Name: imdb_score, dtype: float64
```

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

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

[9]:
```
title
Twins                                                  6.0
Ernest & Celestine                                     7.9
Welcome to the Rileys                                  7.0
Alpha and Omega 4: The Legend of the Saw Toothed Cave  6.0
Fast Times at Ridgemont High                           7.2
Young Frankenstein                                     8.0
Neal 'N' Nikki                                         3.3
Rise of the Entrepreneur: The Search for a Better Way  8.2
Name: imdb_score, dtype: float64
```

9.4 Series subset selection with `iloc`

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

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

Select a single value with `iloc`

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

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

[10]:
```
4.2
```

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

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

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

Select multiple values using a list with `iloc`

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

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

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

Select multiple values using slice notation with `iloc`

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

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

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

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

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

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

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

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

[15]:
```
title
The Life Aquatic with Steve Zissou    7.3
Ride Along 2                          5.9
Trainwreck                            6.3
Down to Earth                         5.4
The Duchess                           6.9
Name: imdb_score, dtype: float64
```

9.5 Summary of Series subset selection

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

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

9.6 Exercises

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

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

[16]:
```
title
Avatar                                         178.0
Pirates of the Caribbean: At World's End       169.0
Spectre                                        148.0
```

```
The Dark Knight Rises                          164.0
Star Wars: Episode VII - The Force Awakens       NaN
Name: duration, dtype: float64
```

Exercise 1

How long was the movie 'Titanic'?

[17]:

Exercise 2

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

[18]:

Exercise 3

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

[19]:

Exercise 4

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

[20]:

Exercise 5

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

[21]:

Read in bikes dataset

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

[22]:
```
bikes = pd.read_csv('../data/bikes.csv')
wind = bikes['wind_speed']
wind.head()
```

[22]:
```
0    12.7
1     6.9
2    16.1
3    16.1
4    17.3
Name: wind_speed, dtype: float64
```

Exercise 6

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

[23]:

Exercise 7

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

[24]:

Exercise 8

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

[25]:

Chapter 10

Boolean Selection Single Conditions

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

Examples of boolean selection

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

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

All queries have a logical condition

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

Keep or discard an entire row of data

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

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

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

Begin with a small DataFrame

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

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

```
df
```

name	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

10.1 Manually filtering the data

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

Create a list of booleans

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

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

Variable name `filt`

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

Place this list in just the brackets

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

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

name	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

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

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

10.2 Operator overloading

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

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

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

10.3 Practical boolean selection

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

Creating boolean Series from column data

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

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

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

3 rows × 11 columns

Create a boolean Series

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

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

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

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

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

[6]:
```
50089
```

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

[7]:
```
50089
```

Manually verify correctness

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

Complete the boolean selection

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	...	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy

3 rows × 11 columns

How many rows have a trip duration greater than 1000?

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

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

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

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

[10]:
```
50089
```

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

[11]:
```
10178
```

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

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

[12]:
```
0.20319830701351593
```

10.4 Boolean selection in one line

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	...	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy

3 rows × 11 columns

10.5 Single condition expression

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

[14]:
```
filt = bikes['from_station_name'] == 'State St & Van Buren St'
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
7	Male	2013-07-03 09:07:00	2013-07-03 09:16:00	505	State St & Van Buren St	...	Franklin St & Jackson Blvd	27.0	64.0	5.8	cloudy
20	Female	2013-07-09 17:39:00	2013-07-09 17:55:00	943	State St & Van Buren St	...	State St & 16th St	15.0	82.9	9.2	mostlycloudy
55	Male	2013-07-16 15:31:00	2013-07-16 15:37:00	363	State St & Van Buren St	...	Daley Center Plaza	47.0	91.0	8.1	mostlycloudy

3 rows × 11 columns

10.6 Summary of single condition boolean selection

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

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

10.7 Exercises

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

Exercise 1

Find all the rides with temperature below 0.

[15]:

Exercise 2

Find all the rides with wind speed greater than 30.

[16]:

Exercise 3

Find all the rides that began from station 'Millennium Park'.

[17]:

Exercise 4

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

[18]:

Exercise 5

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

[19]:

Exercise 6

Did any rides happen in temperature over 100 degrees?

[20]:

Read in new data

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

[21]:
```
import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

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

3 rows × 21 columns

Exercise 7

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

[22]:

Exercise 8

Select movies with an IMDB score greater than 9.

[23]:

Exercise 9

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

[24]:

Chapter 11

Boolean Selection Multiple Conditions

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

11.1 Logical operators

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

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

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

[1]:
```
False
```

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

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

[2]:
```
True
```

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

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

[3]:
```
False
```

Different logical operators for boolean Series

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

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

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

3 rows × 11 columns

Our first multiple condition expression

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	...	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy

3 rows × 11 columns

11.2 Multiple conditions in one line

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

Use parentheses to separate conditions

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

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

Same results

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	...	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy

3 rows × 11 columns

11.3 Using an or condition

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
9	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	Lakeview Ave & Fullerton Pkwy	...	Racine Ave & Congress Pkwy	19.0	81.0	12.7	mostlycloudy

3 rows × 11 columns

11.4 Inverting a condition with the not operator

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	...	Clark St & Randolph St	31.0	72.0	16.1	mostlycloudy

3 rows × 11 columns

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	...	Clark St & Randolph St	31.0	72.0	16.1	mostlycloudy

3 rows × 11 columns

Invert a more complex condition

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	...	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	...	Clark St & Randolph St	31.0	72.0	16.1	mostlycloudy

3 rows × 11 columns

Even more complex conditions

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

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

filt2 = ((bikes['gender'] == 'Female') &
         (bikes['tripduration'] >= 2000) &
         (bikes['tripduration'] <= 3000))
filt = filt1 | filt2
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy
173	Female	2013-08-08 08:49:00	2013-08-08 09:31:00	2502	Sheffield Ave & Addison St	...	Dearborn St & Adams St	19.0	71.1	10.4	mostlycloudy
258	Female	2013-08-17 22:10:00	2013-08-17 22:53:00	2566	Millennium Park	...	Theater on the Lake	15.0	69.1	5.8	clear

3 rows × 11 columns

11.5 Many equality conditions in a single column

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

[12]:
```
filt = ((bikes['events'] == 'rain') |
        (bikes['events'] == 'snow') |
        (bikes['events'] == 'tstorms') |
        (bikes['events'] == 'sleet'))
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	...	State St & Harrison St	19.0	82.9	5.8	rain
78	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	Michigan Ave & Pearson St	...	Millennium Park	35.0	82.4	11.5	tstorms
79	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	Carpenter St & Huron St	...	Carpenter St & Huron St	19.0	82.4	11.5	tstorms

3 rows × 11 columns

Use the `isin` method instead

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	...	State St & Harrison St	19.0	82.9	5.8	rain
78	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	Michigan Ave & Pearson St	...	Millennium Park	35.0	82.4	11.5	tstorms
79	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	Carpenter St & Huron St	...	Carpenter St & Huron St	19.0	82.4	11.5	tstorms

3 rows × 11 columns

Combining `isin` with other filters

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2344	Female	2014-03-19 07:23:00	2014-03-19 08:00:00	2181	Seeley Ave & Roscoe St	...	Franklin St & Lake St	23.0	43.0	6.9	rain
7697	Male	2014-09-12 14:20:00	2014-09-12 14:57:00	2213	Damen Ave & Pierce Ave	...	California Ave & Division St	15.0	52.0	12.7	rain
8357	Male	2014-09-30 08:21:00	2014-09-30 08:58:00	2246	Damen Ave & Melrose Ave	...	Wood St & Taylor St	15.0	46.9	11.5	rain

3 rows × 11 columns

11.6 Exercises

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

Exercise 1

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

[15]:
```

```

Exercise 2

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

[16]:
```

```

Exercise 3

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

[17]:
```

```

Exercise 4

Find all the rides with temperature greater than 90 or trip duration greater than 2000 or wind speed greater than 20.

[18]:

Exercise 5

Invert the condition from exercise 4.

[19]:

Exercise 6

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

[20]:

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

[21]:
```
import pandas as pd
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

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

3 rows × 21 columns

Exercise 7

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

[22]:

Exercise 8

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

[23]:

Exercise 9

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

[24]:

Exercise 10

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

[25]:

Exercise 11

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

[26]:

Exercise 12

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

[27]:

Exercise 13

Select all movies from the 1970's.

[28]:

Chapter 12

Boolean Selection More

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

12.1 Boolean selection on a Series

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

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

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

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

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

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

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

[3]:
```
filt1 = temp < 0
filt2 = temp > 95
```

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

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

Set the index as `starttime`

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

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

starttime	gender	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
2013-06-28 19:01:00	Male	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	11.0	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
2013-06-28 22:53:00	Male	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	31.0	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2013-06-30 14:43:00	Male	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	15.0	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

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

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

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

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

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

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

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

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

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

12.2 The `between` method

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

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

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

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

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

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

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

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

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	...	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy

3 rows × 11 columns

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

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

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	...	State St & Harrison St	19.0	82.9	5.8	rain
78	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	Michigan Ave & Pearson St	...	Millennium Park	35.0	82.4	11.5	tstorms
79	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	Carpenter St & Huron St	...	Carpenter St & Huron St	19.0	82.4	11.5	tstorms

3 rows × 11 columns

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

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

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

	events	tripduration
45	rain	727
112	rain	1395
124	rain	442
161	rain	890
498	rain	978

Now let's find all female riders with trip duration greater than 5,000 when it was cloudy. We'll only return the columns used during the boolean selection.

[13]:
```
filt1 = bikes['gender'] == 'Female'
filt2 = bikes['tripduration'] > 5000
filt3 = bikes['events'] == 'cloudy'
filt = filt1 & filt2 & filt3
cols = ['gender', 'tripduration', 'events']
bikes.loc[filt, cols]
```

	gender	tripduration	events
2712	Female	79988	cloudy
14455	Female	7197	cloudy
22868	Female	13205	cloudy
36441	Female	19922	cloudy

12.4 Column to column comparisons

So far, we created filters by comparing each of our column values to a single scalar value. It is possible to do element-by-element comparisons by comparing two columns to one another. For instance, the total bike capacity at each station at the start and end of the ride is stored in the `start_capacity` and `end_capactiy` columns. If we wanted to test whether there was more capacity at the start of the ride vs the end, we would do the following:

[14]:
```
filt = bikes['start_capacity'] > bikes['end_capacity']
```

Let's use this filter with `loc` to return all the rows where the start capacity is greater than the end.

[15]:
```
cols = ['start_capacity', 'end_capacity']
bikes.loc[filt, cols].head(3)
```

	start_capacity	end_capacity
1	31.0	19.0
6	31.0	19.0
8	31.0	15.0

Boolean selection with `iloc` does not work

The pandas developers decided not to allow boolean selection with `iloc`. The following raises an error.

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

```
NotImplementedError: iLocation based boolean indexing on an integer type is
not available
```

12.5 Finding missing values with `isna`

The `isna` method called from either a DataFrame or a Series returns `True` for every value that is missing and `False` for any other value. Let's see this in action by calling `isna` on the start capacity column.

[17]:
```
bikes['start_capacity'].isna().head(3)
```

[17]:
```
0    False
1    False
2    False
Name: start_capacity, dtype: bool
```

Filtering for missing values

We can now use this boolean Series to select all the rows where the capacity start column is missing. Verify that those values are indeed missing.

[18]:
```
filt = bikes['start_capacity'].isna()
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
17566	Male	2015-09-06 07:52:00	2015-09-06 07:55:00	207	Clark St & 9th St (AMLI)	...	Federal St & Polk St	19.0	75.0	4.6	mostlycloudy
17605	Female	2015-09-07 09:52:00	2015-09-07 09:57:00	293	Clark St & 9th St (AMLI)	...	Wabash Ave & 8th St	19.0	81.0	8.1	mostlycloudy
17990	Male	2015-09-15 08:25:00	2015-09-15 08:33:00	473	Clark St & 9th St (AMLI)	...	Franklin St & Monroe St	27.0	68.0	9.2	mostlycloudy

3 rows × 11 columns

`isnull` is an alias for `isna`

There is an identical method named `isnull` that you will see in other tutorials. It is an **alias** of `isna` meaning it does the exact same thing but has a different name. Either one is suitable to use, but I prefer `isna` because of the similarity to **NaN**, the representation of missing values. There are also other methods such as `dropna` and `fillna` that have 'na' in their names.

12.6 Exercises

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

Exercise 1

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

[19]:
```

```

Exercise 2

Select all wind speed values between 12 and 16.

[20]:
```

```

Exercise 3

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

[21]:
```

```

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

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

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

3 rows × 21 columns

Exercise 4

Select all the movies such that the Facebook likes for actor 2 are greater than those for actor 1.

[23]:

Exercise 5

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

[24]:

Exercise 6

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

[25]:

Exercise 7

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

[26]:

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.

[27]:

Chapter 13

Filtering with the `query` Method

The previous chapters on boolean selection showed us how to filter our DataFrames and Series based on their values. We created conditions, usually involving the comparison operators, resulting in boolean Series and passed them to the *just the brackets* or `loc` indexers to filter the data.

In this chapter we cover the `query` method, which enables us to also make selections based on the values of the DataFrame or Series. The `query` method is easier and more intuitive to use than boolean selection, but doesn't provide as much functionality to filter the data. Still, it is a good method to know about to make your subset selections more readable.

13.1 The `query` method

The `query` method allows you to filter the data by writing the condition as a string. For instance, you would use the string `'tripduration > 1000'` to select all rows of the `bikes` dataset that have a `tripduration` greater than 1,000. Let's read in the bikes dataset and run this command now.

[1]:
```
import pandas as pd
bikes = pd.read_csv('../data/bikes.csv', parse_dates=['starttime', 'stoptime'])
bikes.query('tripduration > 1000').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	...	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy

3 rows × 11 columns

Less syntax and more readable

The `query` method generally uses less syntax than boolean selection and is usually more readable. Reproducing the last result with boolean selection would look like this:

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

This looks a bit clumsy with the name `bikes` written twice right next to one another. The `query` method has its own set of rules for what constitutes a correctly written condition within the string you pass it. The rest of this chapter covers all of the available functionality of the `query` method. This syntax only works within the `query` method and is not allowed anywhere else in pandas.

13.2 Use strings `and`, `or`, `not`

Unlike boolean selection, you can use the strings `and`, `or`, and `not` instead of the operators `&`, `|`, and `~` which further aides readability with `query`. Let's select all rides with `tripduration` greater than 1,000 and `temperature` greater than 85.

[2]:
```
bikes.query('tripduration > 1000 and temperature > 85').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
53	Male	2013-07-16 13:04:00	2013-07-16 13:28:00	1435	Canal St & Jackson Blvd	...	Canal St & Jackson Blvd	35.0	90.0	8.1	mostlycloudy
60	Male	2013-07-17 10:23:00	2013-07-17 10:40:00	1024	Clinton St & Washington Blvd	...	Larrabee St & Menomonee St	15.0	88.0	5.8	partlycloudy

3 rows × 11 columns

13.3 Chained comparisons

Let's say we want to find all rides where the temperature was between 50 and 60 degrees. You can do this with `query` by using the `and` operator.

[3]:
```
bikes.query('temperature >= 50 and temperature <= 60').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
590	Female	2013-09-13 07:55:00	2013-09-13 08:01:00	319	Greenview Ave & Fullerton Ave	...	Sheffield Ave & Fullerton Ave	15.0	54.0	15.0	partlycloudy
591	Male	2013-09-13 08:04:00	2013-09-13 08:16:00	738	Lincoln Ave & Armitage Ave	...	Larrabee St & Kingsbury St	27.0	57.9	13.8	partlycloudy
592	Female	2013-09-13 08:04:00	2013-09-13 08:14:00	599	Orleans St & Elm St	...	Sheffield Ave & Kingsbury St	15.0	57.9	13.8	partlycloudy

3 rows × 11 columns

While this syntax is valid, there is a more compact way. You can use a **chained comparison** to make the string even more readable and concise. A chained comparison places the column name between two comparison operators. The following implies that 50 is less than or equal to the temperature and the temperature is less than or equal to 60 which is equivalent to our previous selection.

[4]:
```
bikes.query('50 <= temperature <= 60').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
590	Female	2013-09-13 07:55:00	2013-09-13 08:01:00	319	Greenview Ave & Fullerton Ave	...	Sheffield Ave & Fullerton Ave	15.0	54.0	15.0	partlycloudy
591	Male	2013-09-13 08:04:00	2013-09-13 08:16:00	738	Lincoln Ave & Armitage Ave	...	Larrabee St & Kingsbury St	27.0	57.9	13.8	partlycloudy
592	Female	2013-09-13 08:04:00	2013-09-13 08:14:00	599	Orleans St & Elm St	...	Sheffield Ave & Kingsbury St	15.0	57.9	13.8	partlycloudy

3 rows × 11 columns

13.4 Reference strings with quotes

If you would like to reference a literal string within `query`, you need to surround it with quotes, or else pandas will attempt to use it as a column name. Let's select all rides by a 'Female' with a trip duration greater than 2,000.

[5]:
```
bikes.query('gender == "Female" and tripduration > 2000').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy
173	Female	2013-08-08 08:49:00	2013-08-08 09:31:00	2502	Sheffield Ave & Addison St	...	Dearborn St & Adams St	19.0	71.1	10.4	mostlycloudy

3 rows × 11 columns

Forgetting quotes

If you do not use quotes around your literal string, then pandas assumes that value is a column name. The following raises an error. It believes you are accessing a column name Female, which doesn't exist.

[6]:
```
bikes.query('gender == Female and tripduration > 2000')
```

```
UndefinedVariableError: name 'Female' is not defined
```

13.5 Column to column comparisons

It is possible to compare each value in one column with each value in another column. Here, we filter for all the rides where there were more bikes at the start than at the end.

[7]:
```
bikes.query('start_capacity > end_capacity').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	...	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
6	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565	Clark St & Randolph St	...	Ravenswood Ave & Irving Park Rd	19.0	66.0	15.0	cloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy

3 rows × 11 columns

13.6 Use 'in' for multiple equalities

You can check whether each value in a column is equal to one or more other values by using the word 'in' within your query. Use the syntax for creating a list within the query string to contain all the values you'd like to check. The following tests whether the weather event was snow or rain.

[8]:
```
bikes.query('events in ["snow", "rain"]').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	...	State St & Harrison St	19.0	82.9	5.8	rain
112	Male	2013-07-26 19:10:00	2013-07-26 19:33:00	1395	Larrabee St & Kingsbury St	...	Damen Ave & Pierce Ave	19.0	66.9	12.7	rain
124	Male	2013-07-30 18:53:00	2013-07-30 19:00:00	442	Canal St & Jackson Blvd	...	Racine Ave & Congress Pkwy	19.0	69.1	3.5	rain

3 rows × 11 columns

There are multiple syntaxes for the above that all work the same, but I prefer using the above as it is most similar to the `isin` method used during boolean selection.

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

Use 'not in' to invert the condition

You can invert the result of an 'in' clause by placing the word 'not' before it. Here, we find all the rides that did not have the weather events cloudy, partly cloudy or mostly cloudy.

[9]:
```
bikes.query('events not in ["cloudy", "partlycloudy", "mostlycloudy"]').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
25	Female	2013-07-11 08:17:00	2013-07-11 08:31:00	830	Wabash Ave & Roosevelt Rd	...	Daley Center Plaza	47.0	73.9	8.1	clear
26	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043	State St & Harrison St	...	Racine Ave & 18th St	15.0	64.9	0.0	clear
33	Male	2013-07-12 17:22:00	2013-07-12 17:34:00	730	Clark St & Congress Pkwy	...	Racine Ave & Congress Pkwy	19.0	79.0	10.4	clear

3 rows × 11 columns

13.7 Arithmetic operations within query

It is possible to write arithmetic operations within `query` just as you would outside of it. For instance, if we wanted to find all the rides such that there were 20 or more bikes at the start station than at the end, we do the following.

[10]:
```
bikes.query('start_capacity - end_capacity >= 20').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
54	Male	2013-07-16 15:13:00	2013-07-16 15:18:00	347	Daley Center Plaza	...	State St & Van Buren St	27.0	91.0	8.1	mostlycloudy
66	Male	2013-07-17 20:56:00	2013-07-17 21:14:00	1073	Millennium Park	...	Morgan St & 18th St	15.0	86.0	9.2	partlycloudy
116	Male	2013-07-27 09:54:00	2013-07-27 09:56:00	121	Daley Center Plaza	...	LaSalle St & Washington St	15.0	60.8	11.5	cloudy

3 rows × 11 columns

Filtering for right triangles

Let's read in the triangles dataset which contains the lengths of each side of a triangle as the columns `a`, `b`, and `c`.

[11]:
```
triangles = pd.read_csv('../data/triangles.csv')
triangles.head()
```

	a	b	c
0	2	3	4
1	3	2	4
2	3	4	5
3	3	5	6
4	3	6	7

We can use the `query` method to find all the right triangles, those that satisfy the Pythagorean Theorem. We write the condition using the arithmetic and comparison operators.

[12]:
```
triangles.query('a ** 2 + b ** 2 == c ** 2').head()
```

	a	b	c
2	3	4	5
5	4	3	5
14	5	12	13
21	6	8	10
33	7	24	25

The syntax is quite a bit nicer than the boolean selection alternative.

[13]:
```
filt = triangles['a'] ** 2 + triangles['b'] ** 2 == triangles['c'] ** 2
triangles[filt].head()
```

	a	b	c
2	3	4	5
5	4	3	5
14	5	12	13
21	6	8	10
33	7	24	25

13.8 Reference variable names with the `@` symbol

By default, all words within the query string attempt to reference a column name. You can, however, reference a variable name by preceding it with the `@` symbol. Let's assign the variable name `min_length` to 5,000 and reference it in a query to find all the rides where trip duration was greater than it.

[14]:
```
min_length = 5000
bikes.query('tripduration > @min_length').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy

3 rows × 11 columns

13.9 Using the index with `query`

You can even use the word `index` to make comparisons against the index as if it were a normal column. In the bikes DataFrame, the index is just the integers beginning at 0. Here, we select only the `events` that were 'cloudy' for an index value greater than 4,000.

[15]:
```
bikes.query('index > 4000 and events == "cloudy" ').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
4007	Male	2014-06-07 14:07:00	2014-06-07 14:31:00	1434	Lake Shore Dr & North Blvd	...	Halsted St & Roscoe St	15.0	82.0	13.8	cloudy
4008	Male	2014-06-07 14:58:00	2014-06-07 15:19:00	1258	Theater on the Lake	...	Sheridan Rd & Buena Ave	15.0	82.0	13.8	cloudy
4009	Male	2014-06-07 15:23:00	2014-06-07 15:28:00	297	Sheffield Ave & Addison St	...	Pine Grove Ave & Waveland Ave	23.0	80.1	13.8	cloudy

3 rows × 11 columns

Referencing named index

If your DataFrame has an index that is named, which happens when a column is set as the index, then you can use that name within `query` just as if it were a regular column name. Here, we create a new DataFrame that has the `from_station_name` as the index.

[16]:
```
bikes_idx = bikes.set_index('from_station_name')
bikes_idx.head(3)
```

from_station_name	gender	starttime	stoptime	tripduration	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
Lake Shore Dr & Monroe St	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	11.0	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
Clinton St & Washington Blvd	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	31.0	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
Sheffield Ave & Kingsbury St	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	15.0	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

Notice the name 'from_station_name' directly above the index. This is the name for the index and what can be referenced when using `query`. Let's filter for trip ids greater than 200,000.

[17]:
```
bikes_idx.query('from_station_name == "Theater on the Lake"').head(3)
```

from_station_name	gender	starttime	stoptime	tripduration	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
Theater on the Lake	Male	2013-08-23 17:57:00	2013-08-23 18:16:00	1166	15.0	Lincoln Ave & Roscoe St	19.0	79.0	9.2	partlycloudy
Theater on the Lake	Female	2013-08-24 15:31:00	2013-08-24 15:59:00	1661	15.0	Fairbanks Ct & Grand Ave	15.0	84.9	6.9	partlycloudy
Theater on the Lake	Male	2013-09-07 14:28:00	2013-09-07 14:37:00	540	15.0	Sheffield Ave & Fullerton Ave	15.0	88.0	10.4	mostlycloudy

13.10 Use backticks to reference column names with spaces

pandas allows DataFrames to have column names with spaces in them. In order to use a column name containing spaces within `query`, you'll need to surround it with backticks. If you don't use the backticks you'll get an error. Let's read in the San Francisco employee compensation dataset which contains multiple column names that have spaces.

[18]:
```
sf_emp = pd.read_csv('../data/sf_employee_compensation.csv')
sf_emp.head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
1	2013	General Administration & Finance	Planner 2	67941.06	0.00	0.0	13030.23	10047.52	5608.37
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20

Let's find all the employees that are in the organization group of 'Public Protection'.

[19]:
```
sf_emp.query('`organization group` == "Public Protection"').head(3)
```

	year	organization group	job	salaries	overtime	other salaries	retirement	health and dental	other benefits
0	2013	Public Protection	Personnel Technician	71414.01	0.00	0.0	14038.58	12918.24	5872.04
2	2013	Public Protection	Firefighter	116956.72	59975.43	19037.3	24796.44	15788.97	3222.20
7	2013	Public Protection	Police Officer	78591.02	2050.18	832.9	15383.49	11004.42	4471.61

Selecting columns with `query`

Unfortunately the `query` method does not give us the ability to select a subset of the columns when filtering the data. You would have to do normal column selection after calling the method. Here, we use *just the brackets* to select three columns after finding all the rides where the weather was snow or rain.

[20]:
```
cols = ['starttime', 'temperature', 'events']
bikes.query('events in ["snow", "rain"]')[cols].head()
```

	starttime	temperature	events
45	2013-07-15 16:43:00	82.9	rain
112	2013-07-26 19:10:00	66.9	rain
124	2013-07-30 18:53:00	69.1	rain
161	2013-08-05 17:09:00	68.0	rain
498	2013-09-07 16:09:00	81.0	rain

13.11 Summary

The `query` method provides an alternative to boolean selection to filter the data based on the values. Here are the rules for the string you provide.

- The expression in the string must evaluate as True or False for every row
- Column names may be accessed directly with their name
- Often you will use one of the comparison operators to create a condition
- Use chained comparison operators to shorten syntax
- Use `and`, `or`, and `not` to create more complex conditions
- To use a literal string, surround it with quotes
- Use `in` to test multiple equalities. Provide the test values in a list
- All arithmetic operators work just as they do outside of the string
- Use the `@` character to reference a variable name
- Reference the index with the string 'index' or the index's name
- Use backticks to reference a column name with spaces in it

13.12 Exercises

Use the bikes dataset for the first few exercises.

Exercise 1

Use the `query` method to select trip durations between 5,000 and 10,000.

[21]:

Exercise 2

Use the `query` method to select trip durations between 5,000 and 10,000 when the weather was snow or rain. Retrieve the same data with boolean selection.

[22]:

Exercise 3

Use the `query` method to select trip durations between 5,000 and 10,000 when it was snow or rain. Create a list outside of the `query` method to hold the weather and reference that variable with `@` within `query`.

[23]:

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

[24]:
```
import pandas as pd
pd.set_option('display.max_columns', 50)
movie = pd.read_csv('../data/movie.csv', index_col='title')
movie.head(3)
```

title	year	color	content_rating	duration	director_name	director_fb	actor1	actor1_fb	actor2	actor2_fb	actor3	actor3_fb	gross	genres	num_reviews	num_voted_users	plot_keywords	language	country	budget	imdb_score
Avatar	2009.0	Color	PG-13	178.0	James Cameron	0.0	CCH Pounder	1000.0	Joel David Moore	936.0	Wes Studi	855.0	760505847.0	Action|Adventure|Fantasy|Sci-Fi	723.0	886204	avatar|future|marine|native|paraplegic	English	USA	237000000.0	7.9
Pirates of the Caribbean: At World's End	2007.0	Color	PG-13	169.0	Gore Verbinski	563.0	Johnny Depp	40000.0	Orlando Bloom	5000.0	Jack Davenport	1000.0	309404152.0	Action|Adventure|Fantasy	302.0	471220	goddess|marriage ceremony|marriage proposal|pi...	English	USA	300000000.0	7.1
Spectre	2015.0	Color	PG-13	148.0	Sam Mendes	0.0	Christoph Waltz	11000.0	Rory Kinnear	393.0	Stephanie Sigman	161.0	200074175.0	Action|Adventure|Thriller	602.0	275868	bomb|espionage|sequel|spy|terrorist	English	UK	245000000.0	6.8

Exercise 4

Use the `query` method to find all movies where the total number of Facebook likes for all three actors is greater than 50,000.

[25]:

Exercise 5

Select all the movies where the number of user voters is less than 10 times the number of reviews.

[26]:

Exercise 6

Select all the movies made in the 1990's that were rated R with an IMDB score greater than 8.

[27]:

Chapter 14

Miscellaneous Subset Selection

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

I personally do not use the methods described in this chapter and suggest that you also avoid them. They are all valid syntax and some pandas users do actually use them, so you may find them valuable.

14.1 Selecting a column with dot notation

In my opinion, the best way to select a single column from a DataFrame as a Series is by placing the name of the column within *just the brackets*. There's actually an alternative way to select a single column of data and that is with dot notation. Let's read in the the `sample_data2.csv` dataset.

[1]:
```
import pandas as pd
df = pd.read_csv('../data/sample_data2.csv')
df
```

	name	average score	max
0	Niko	99	100
1	Penelope	100	102
2	Aria	88	93

Place the name of the column directly after the dot as if it were an attribute.

[2]:
```
df.name
```

[2]:
```
0        Niko
1    Penelope
2        Aria
Name: name, dtype: object
```

This produces an identical result as using *just the brackets*.

[3]:
```
df['name']
```

[3]:
```
0        Niko
1    Penelope
2        Aria
Name: name, dtype: object
```

Avoid dot notation - use just the brackets

Although this method for column selection requires less syntax and is used by many pandas users, it has many downsides. The following is a partial list of the functionality that is impossible using dot notation.

- Select column names with spaces
- Select column names that have the same name as methods
- Select columns with variables
- Select columns that begin with numbers
- Select columns that are non-strings

Examples of some of the above scenarios will now be covered. Using dot notation does not allow you to select columns with spaces. Selecting the column `average score` raises a syntax error.

[4]:
```
df.average score
```

```
  File "/var/folders/0x/whw882fj4qv0czzqzrngxvdh0000gn/T/ipykernel_42667/15
88428431.py", line 4
    df.average score
               ^
SyntaxError: invalid syntax
```

The only way to select this column is with *just the brackets*.

[5]:
```
df['average score']
```

[5]:
```
0     99
1    100
2     88
Name: average score, dtype: int64
```

Dot notation is unable to select columns that are the same name as methods. For instance, `max` is a method that all DataFrames have. In this particular DataFrame, it also the name of the column. Attempting to select it via dot notation will access the method.

[6]:
```
df.max
```

[6]:
```
<bound method NDFrame._add_numeric_operations.<locals>.max of        name  average␣
↪score
max
0      Niko             99  100
1  Penelope            100  102
2      Aria             88   93>
```

Again, the only way to select this column is with *just the brackets*.

[7]:
```
df['max']
```

[7]:
```
0    100
1    102
2     93
Name: max, dtype: int64
```

Dot notation is unable to select a column using a variable name. Let's say we assign the variable `col` to the string 'name' which is the name of the first column. Attempting to select it via dot notation raises an error.

[8]:
```
col = 'name'
df.col
```

```
AttributeError: 'DataFrame' object has no attribute 'col'
```

Once again, use *just the brackets*.

[9]:
```
df[col]
```

[9]:
```
0        Niko
1    Penelope
2        Aria
Name: name, dtype: object
```

Video with 10 reasons why using the brackets are superior

There are actually many more reasons to use the brackets over dot notation. If you are interested in hearing all of my reasons, watch this video.

14.2 Selecting rows with just the brackets using slice notation

So far, we have covered three ways to select subsets of data with *just the brackets*. You can use a single string, a list of strings, or a boolean Series. Let's quickly review those ways right now using the bikes dataset.

[10]:
```
bikes = pd.read_csv('../data/bikes.csv')
```

A single string

[11]:
```
bikes['tripduration'].head(3)
```

[11]:
```
0     993
1     623
2    1040
Name: tripduration, dtype: int64
```

A list of strings

[12]:
```
cols = ['gender', 'tripduration']
bikes[cols].head(3)
```

	gender	tripduration
0	Male	993
1	Male	623
2	Male	1040

A boolean Series

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

[13]:
```
filt = bikes['tripduration'] > 5000
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	...	Millennium Park	35.0	79.0	13.8	cloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	...	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	...	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy

3 rows × 11 columns

Using a slice

It is possible to use slice notation within just the brackets. For example, the following selects the rows beginning at location 2 up to location 10 with a step size of 3. You can even use slice notation when the index is strings.

[14]:
```
bikes[2:10:3]
```

	gender	starttime	stoptime	tripduration	from_station_name	...	to_station_name	end_capacity	temperature	wind_speed	events
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	...	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy
5	Male	2013-07-01 12:37:00	2013-07-01 12:48:00	660	California Ave & 21st St	...	Clark St & Wrightwood Ave	15.0	73.0	17.3	mostlycloudy
8	Male	2013-07-03 15:21:00	2013-07-03 15:42:00	1300	Clinton St & Washington Blvd	...	Wood St & Division St	15.0	71.1	0.0	cloudy

3 rows × 11 columns

I do not recommend using slicing with *just the brackets*

Although slicing with *just the brackets* seems simple, I do not recommend using it. This is because it is ambiguous and can make selections either by integer location or by label. I always prefer explicit, unambiguous methods. Both `loc` and `iloc` are unambiguous and explicit. Meaning that even without knowing anything about the DataFrame, you would be able to explain exactly how the selection will take place. If you do want to slice the rows, then use `loc` if you are using labels or `iloc` if you are using integer location, but do not use *just the brackets*.

14.3 Selecting a single cell with `at` and `iat`

pandas provides two more rarely seen indexers, `at`, and `iat`. These indexers are analogous to `loc` and `iloc` respectively, but only select a single cell of a DataFrame. Since they only select a single cell, you must pass both a row and column selection as either a label (`loc`) or an integer location (`iloc`). Let's see an example of each.

[15]:
```
bikes.at[40, 'temperature']
```

[15]: 87.1

[16]:
```
bikes.iat[-30, 5]
```

[16]: 23.0

The current index labels for `bikes` is integers which is why the number 40 was used above. It is the label for a row, but also happens to be an integer.

What's the purpose of these indexers?

All usages of `at` and `iat` may be replaced with `loc` and `iloc` and would produce the exact same results. The `at` and `iat` indexers are optimized to select a single cell of data and therefore provide slightly better performance than `loc` or `iloc`. Let's verify this below.

[17]:
```
bikes.loc[40, 'temperature']
```

[17]: 87.1

[18]:
```
bikes.iloc[-30, 5]
```

[18]: 23.0

I never use these indexers

Personally, I never use these specialty indexers as the performance advantage for a single selection is minor. It would require a case where single element selections were happening in great numbers to see any significant improvement and doing so is rare in data analysis.

Much bigger performance improvement using numpy directly

If you truly wanted a large performance improvement for single-cell selection, you would select directly from numpy arrays and not a pandas DataFrame. Below, the data is extracted into the underlying numpy array with the `values` attribute. We then time the performance of selecting with numpy and also with `iat` and `iloc` on a DataFrame.

The timing is done using the magic command `%time`. This is a special command only available in a Jupyter Notebook (or IPython shell). The **Wall time** provides the total time it took to complete the operation. On my machine, `iat` shows a negligible improvement over `iloc`, but selecting with numpy is about 15x as fast. There is no comparison here, if you care about performance for selecting a single cell of data, use numpy.

[19]:
```
values = bikes.values
```

[20]:
```
%time values[-30, 5]
```

```
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 7.15 µs
```

[20]:
```
23.0
```

[21]:
```
%time bikes.iat[-30, 5]
```

```
CPU times: user 144 µs, sys: 1 µs, total: 145 µs
Wall time: 150 µs
```

[21]:
```
23.0
```

[22]:
```
%time bikes.iloc[-30, 5]
```

```
CPU times: user 134 µs, sys: 0 ns, total: 134 µs
Wall time: 138 µs
```

[22]:
```
23.0
```

14.4 Exercises

Exercise 1

Provide several example column names that are not possible to select using dot notation.

[23]:
```

```

Exercise 2

Use the `%time` magic function to compare the performance difference between `loc` and `at` and between `iloc` and `iat`.

[24]:
```

```