

# Pandas





# Table of Contents

---

## PART 1: GETTING STARTED

- 1: Introducing pandas: 5
- 2: A whirlwind tour of pandas: 14

## PART 2: THE PYTHON ECOSYSTEM

- 3: Python crash course: 47
- 4: NumPy crash course: 166

## PART 3: THE SERIES

- 5: The Series Object: 193
- 6: Series methods: 255



# Table of Contents

---

## **PART 4: THE DATAFRAME**

- 7: The DataFrame Object: 313
- 8: Filtering a DataFrame: 395

## **PART 5: WORKING WITH TEXT DATA**

- 9: Working with Text Data

## **PART 6: GROUPING, AGGREGATING AND MERGING DATA**

- 10: MultiIndex DataFrames
- 11: Reshaping and Pivoting
- 12: The GroupBy Object



# Table of Contents

---

13: Merging, Joining and Concatenating

## **PART 7: WORKING WITH DATES AND TIMES**

14: Working with Dates and Times

## **PART 8: INPUT AND OUTPUT**

15: Imports and Exports

16: Configuring Pandas

## **PART 9: VISUALIZATION**

17: Visualization



# Lesson 1 Introducing Pandas





# Introducing Pandas



This session covers:

- The role of data in the 21st century
- Popular solutions for data analysis including pandas, Excel, R, and SAS
- The advantages and disadvantages of pandas relative to its competitors

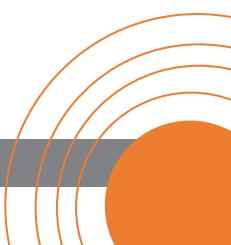


# Data in the 21st Century



"It is a capital mistake to theorize before one has data," advises Sherlock Holmes to his assistant John Watson in A Scandal in Bohemia, the first of Sir Arthur Conan Doyle's classic short stories pairing the duo.

"Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."





# Introducing pandas

---

- As demand has grown for workers skilled at gathering insights from mountains of data, so has the technical ecosystem of tools for doing so.
  - Today, pandas is one of the most popular software packages available for data analysis. An open source library built on top of the Python programming language, pandas enables developers to manipulate and analyze complex data sets with ease.
- 

# Introducing pandas

```
In [2]: populations = pd.read_csv("populations.csv")
populations.sort_values(by = "Population", ascending = False)
```

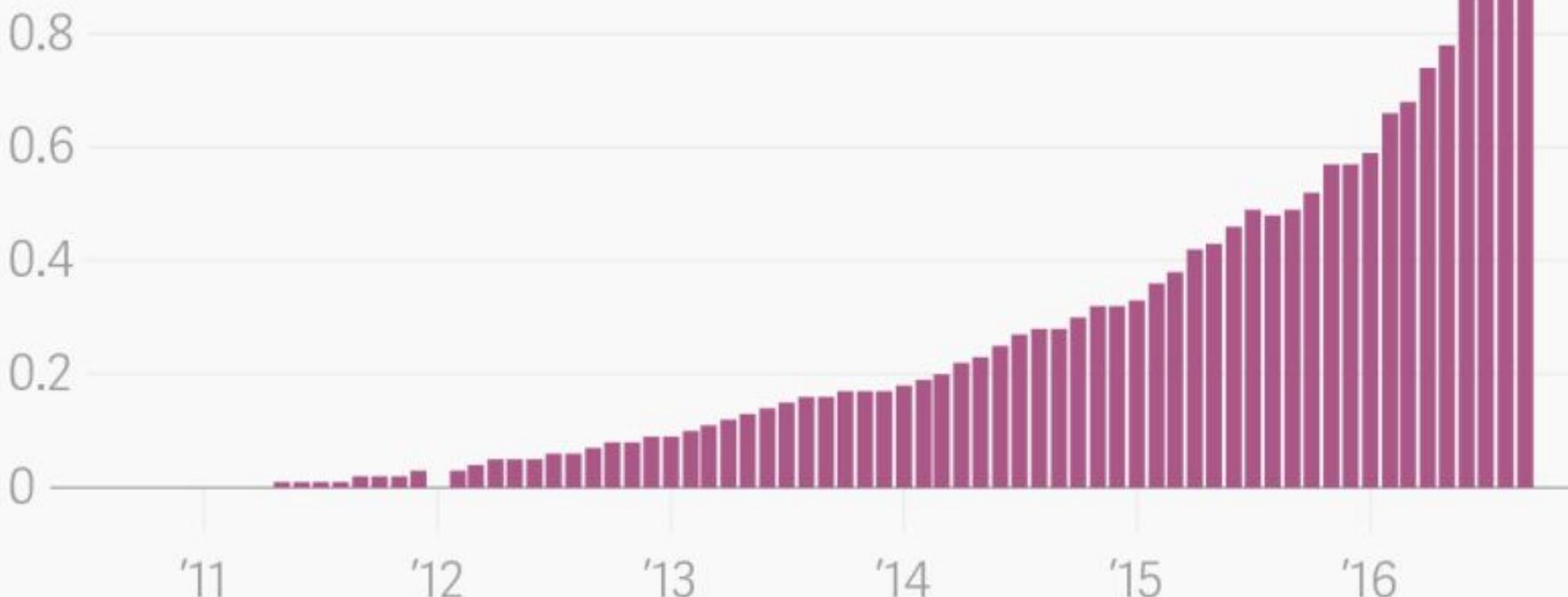
Out[2]:

	Country	Population
144	China	1433783686
21	India	1366417754
156	United States	329064917
76	Indonesia	270625568
147	Pakistan	216565318
79	Brazil	211049527
6	Nigeria	200963599
123	Bangladesh	163046161

# Introducing pandas

## The rise in popularity of Pandas

1.0% of all question views on Stack Overflow\*



ATLAS | Data: Stack Overflow | \* World Bank high-income countries

# Pandas vs Graphical Spreadsheet Applications

- What makes Python different from Excel or Sheets?  
Programming is inherently more verbal than it is visual; pandas requires that our instructions be commands instead of clicks.
- Furthermore, the commands have to be issued with the correct inputs in the correct order.
- The increased probability of errors guarantees a larger learning curve compared to one in a graphical spreadsheet application



# Pandas vs Its Competitors

---

- Among data science enthusiasts, pandas is more frequently compared to the open source programming language R and the proprietary software suite SAS.
  - Each option has its own community of advocates.
  - R is a focused language with a foundation in statistics while Python is a generalist language that can be used for a variety of use cases.
  - Predictably, the languages tend to attract users with familiarity in their respective domains
- 



# Summary



- Pandas is an open-source data analysis library built on top of the Python programming language
  - A library is a collection of tools and features that expand the functionality of a language
  - Pandas excels in performing complex operations on large data sets with a terse syntax
  - Competitors to Pandas include the statistical programming language R and the proprietary SAS software suite
- 

# Lesson 2: A whirlwind Tour of pandas





# A Whirlwind Tour of pandas



This session covers:

- The basics of the DataFrame and the Series, the two primary objects in pandas
- Importing CSV datasets
- Sorting and counting the values in a column of data
- Filtering a tabular dataset by one or more criteria
- Grouping a dataset and aggregating its values



# Importing a Dataset



- Our dataset is contained in a movies.csv file. The CSV (comma separated values) file format is a plain text file that separates each row of data with a line break and each row value with a comma.
- The first row in the file holds the column headers for the data. For example, the first three rows in movies.csv are:

```
Rank,Title,Studio,Gross,Year
1,Avengers: Endgame,Buena Vista,"$2,796.30",2019
2,Avatar,Fox,"$2,789.70",2009
```



# Importing a Dataset



- Let's begin by creating a new Jupyter Notebook inside the same directory as the movies.csv file.
- Our first step is to import the pandas library to get access to its features.
- We'll assign it the popular community alias pd

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



# Importing a Dataset

---

- Pandas can import a variety of file types include xlsx (Excel), sql (SQL), and hdf (Hierarchical Data Format).
- Each type has its own associated import method at the top-level of the library, We'll use the `read_csv` method here and pass it a string with the file name.
- Pandas will first look for the file in the same directory as the Notebook.
- I switched this simply by defining a different codec package in the `read_csv()` command:  
`encoding = 'unicode_escape'`

# Importing a Dataset

```
In [2] pd.read_csv("movies.csv")
```

```
Out [2]
```

<b>Rank</b>		<b>Title</b>	<b>Studio</b>	<b>Gross</b>	<b>Year</b>
0	1	Avengers: Endgame	Buena Vista	\$2,796.30	2019
1	2	Avatar	Fox	\$2,789.70	2009
2	3	Titanic	Paramount	\$2,187.50	1997
3	4	Star Wars: The Force Awakens	Buena Vista	\$2,068.20	2015
4	5	Avengers: Infinity War	Buena Vista	\$2,048.40	2018
...	...	...	...	...	...
777	778	Yogi Bear	Warner Brothers	\$201.60	2010
778	779	Garfield: The Movie	Fox	\$200.80	2004
779	780	Cats & Dogs	Warner Brothers	\$200.70	2001
780	781	The Hunt for Red October	Paramount	\$200.50	1990
781	782	Valkyrie	MGM	\$200.30	2008

782 rows × 5 columns

# Importing a Dataset

```
In [3] pd.read_csv("movies.csv", index_col = "Title")
```

```
Out [3]
```

Title	Rank	Studio	Gross	Year
Avengers: Endgame	1	Buena Vista	\$2,796.30	2019
Avatar	2	Fox	\$2,789.70	2009
Titanic	3	Paramount	\$2,187.50	1997
Star Wars: The Force Awakens	4	Buena Vista	\$2,068.20	2015
Avengers: Infinity War	5	Buena Vista	\$2,048.40	2018
...	...	...	...	...
Yogi Bear	778	Warner Brothers	\$201.60	2010
Garfield: The Movie	779	Fox	\$200.80	2004
Cats & Dogs	780	Warner Brothers	\$200.70	2001
The Hunt for Red October	781	Paramount	\$200.50	1990
Valkyrie	782	MGM	\$200.30	2008



# Importing a Dataset



- Without a variable assignment, Python will toss an object out of memory after it is processed.
- Let's assign a movies variable to the DataFrame so that can it be used after import.

```
In [4] movies = pd.read_csv("movies.csv", index_col = "Title")
```



# Manipulating a DataFrame

```
In [5] movies.head(4)
```

```
Out [5]
```

Title	Rank	Studio	Gross	Year
Avengers: Endgame	1	Buena Vista	\$2,796.30	2019
Avatar	2	Fox	\$2,789.70	2009
Titanic	3	Paramount	\$2,187.50	1997
Star Wars: The Force Awakens	4	Buena Vista	\$2,068.20	2015

...or view a slice of the end.

```
In [6] movies.tail(6)
```

```
Out [6]
```

Title	Rank	Studio	Gross	Year
21 Jump Street	777	Sony	\$201.60	2012
Yogi Bear	778	Warner Brothers	\$201.60	2010
Garfield: The Movie	779	Fox	\$200.80	2004
Cats & Dogs	780	Warner Brothers	\$200.70	2001
The Hunt for Red October	781	Paramount	\$200.50	1990
Valkyrie	782	MGM	\$200.30	2008



# Manipulating a DataFrame



- The DataFrame plays friendly with Python's built in functions.
- Let's find out how many rows of data are contained in our dataset.

In [7] len(movies)

Out [7] 782



# Manipulating a DataFrame



- The shape of the DataFrame (the number of rows and columns) as well as the total number of values are available as attributes.

In [8] `movies.shape`

Out [8] (782, 4)

In [9] `movies.size`

Out [9] 3128



# Manipulating a DataFrame



- How many dimensions does this dataset have? We can find out with the `ndim` attribute.

```
In [10] movies.ndim
```

```
Out [10] 2
```



# Manipulating a DataFrame

---

- A row can be accessed by its numeric index position.
- Let's say we wanted to discover the movie with a rank of 500.

```
In [11] movies.iloc[499]
```

```
Out [11] Rank 500
```

```
Studio Fox
```

```
Gross $288.30
```

```
Year 2018
```

```
Name: Maze Runner: The Death Cure, dtype: object
```



# Manipulating a DataFrame



- A row can also be accessed by its index label.
- Let's find out the information for the classic 1994 tearjerker Forrest Gump.

```
In [12] movies.loc["Forrest Gump"]
```

```
Out [12] Rank           119
          Studio        Paramount
          Gross        $677.90
          Year         1994
          Name: Forrest Gump, dtype: object
```



# Manipulating a DataFrame



- Unlike dictionary keys, index labels in a DataFrame can contain duplicates.
- There are actually two entries for 101 Dalmatians on the list, the 1961 original and the 1996 remake.

```
In [13] movies.loc["101 Dalmatians"]
```

```
Out [13]
```

Title	Rank	Studio	Gross	Year
101 Dalmatians	425	Buena Vista	\$320.70	1996
101 Dalmatians	708	Buena Vista	\$215.90	1961

# Manipulating a DataFrame

```
In [14] movies.sort_values("Year", ascending = False).head()
```

```
Out [14]
```

Title	Rank	Studio	Gross	Year
Avengers: Endgame	1	Buena Vista	\$2,796.30	2019
John Wick: Chapter 3 - Parab...	458	Lionsgate	\$304.70	2019
The Wandering Earth	114	China Film ...	\$699.80	2019
Toy Story 4	198	Buena Vista	\$519.80	2019
How to Train Your Dragon: The ...	199	Universal	\$519.80	2019

# Manipulating a DataFrame

- We can also sort by multiple columns. Let's sort the dataset first by the Studio alphabetically, then by the Year.

```
In [15] movies.sort_values(["Studio", "Year"]).head()
```

```
Out [15]
```

Title	Rank	Studio	Gross	Year
The Blair Witch Project	588	Artisan	\$248.60	1999
101 Dalmatians	708	Buena Vista	\$215.90	1961
The Jungle Book	755	Buena Vista	\$205.80	1967
Who Framed Roger Rabbit	410	Buena Vista	\$329.80	1988
Dead Poets Society	636	Buena Vista	\$235.90	1989

# Manipulating a DataFrame

- The index can also be sorted in either direction. That's helpful if we want the movies in alphabetical order.

```
In [16] movies.sort_index().head()
```

```
Out [16]
```

Title	Rank	Studio	Gross	Year
10,000 B.C.	536	Warner Brothers	\$269.80	2008
101 Dalmatians	708	Buena Vista	\$215.90	1961
101 Dalmatians	425	Buena Vista	\$320.70	1996
2 Fast 2 Furious	632	Universal	\$236.40	2003
2012	93	Sony	\$769.70	2009

# Counting Values in a Series

```
In [17] movies["Studio"]

Out [17] Title
        Avengers: Endgame           Buena Vista
        Avatar                      Fox
        Titanic                     Paramount
        Star Wars: The Force Awakens Buena Vista
        Avengers: Infinity War      Buena Vista
                                    ...
        Yogi Bear                   Warner Brothers
        Garfield: The Movie         Fox
        Cats & Dogs                Warner Brothers
        The Hunt for Red October   Paramount
        Valkyrie                    MGM
Name: Studio, Length: 782, dtype: object
```

- The `value_counts` method counts the occurrences of each unique value in a Series.
- Let's chain on the method to the end of our existing line of code.
- We'll a

```
In [18] movies["Studio"].value_counts().head(10)
```

Out [18]	Warner Brothers	132
	Buena Vista	125
	Fox	117
	Universal	109
	Sony	86
	Paramount	76
	Dreamworks	27
	Lionsgate	21
	New Line	16
	TriStar	11
	Name:	Studio, dtype: int64

# Filtering a Column by One or More Criteria

```
In [19] movies[movies["Studio"] == "Universal"]
```

```
Out [19]
```

<b>Title</b>	<b>Rank</b>	<b>Studio</b>	<b>Gross</b>	<b>Year</b>
Jurassic World	6	Universal	\$1,671.70	2015
Furious 7	8	Universal	\$1,516.00	2015
Jurassic World: Fallen Kingdom	13	Universal	\$1,309.50	2018
The Fate of the Furious	17	Universal	\$1,236.00	2017
Minions	19	Universal	\$1,159.40	2015
	...	...	...	...
The Break-Up	763	Universal	\$205.00	2006
Everest	766	Universal	\$203.40	2015
Patch Adams	772	Universal	\$202.30	1998
Kindergarten Cop	775	Universal	\$202.00	1990
Straight Outta Compton	776	Universal	\$201.60	2015

109 rows × 4 columns

We can also assign the condition to a variable to provide context to other readers.

```
In [20] released_by_universal = movies["Studio"] == "Universal"  
       movies[released_by_universal]
```

Rows can also be selected based on multiple criteria. For example, let's look at all the movies released by Universal Pictures in the year 2015.

```
In [21] released_by_universal = movies["Studio"] == "Universal"  
       released_in_2015 = movies["Year"] == 2015  
       movies[released_by_universal & released_in_2015]
```

Out [21]

<b>Title</b>	<b>Rank</b>	<b>Studio</b>	<b>Gross</b>	<b>Year</b>
Jurassic World	6	Universal	\$1,671.70	2015
Furious 7	8	Universal	\$1,516.00	2015
Minions	19	Universal	\$1,159.40	2015
Fifty Shades of Grey	165	Universal	\$571.00	2015
Pitch Perfect 2	504	Universal	\$287.50	2015
Ted 2	702	Universal	\$216.70	2015
Everest	766	Universal	\$203.40	2015
Straight Outta Compton	776	Universal	\$201.60	2015

# Filtering a Column by One or More Criteria

```
In [22] released_by_universal = movies["Studio"] == "Universal"  
released_in_2015 = movies["Year"] == 2015  
movies[released_by_universal | released_in_2015]
```

Out [22]

Title	Rank	Studio	Gross	Year
Star Wars: The Force Awakens	4	Buena Vista	\$2,068.20	2015
Jurassic World	6	Universal	\$1,671.70	2015
Furious 7	8	Universal	\$1,516.00	2015
Avengers: Age of Ultron	9	Buena Vista	\$1,405.40	2015
Jurassic World: Fallen Kingdom	13	Universal	\$1,309.50	2018
The Break-Up	... 763	Universal	\$205.00 ...	2006 ...
Everest	766	Universal	\$203.40	2015
Patch Adams	772	Universal	\$202.30	1998
Kindergarten Cop	775	Universal	\$202.00	1990
Straight Outta Compton	776	Universal	\$201.60	2015

140 rows × 4 columns

# Filtering a Column by One or More Criteria

```
In [23] before_1975 = movies["Year"] < 1975  
movies[before_1975]
```

```
Out [23]
```

Title	Rank	Studio	Gross	Year
The Exorcist	252	Warner Brothers	\$441.30	1973
Gone with the Wind	288	MGM	\$402.40	1939
Bambi	540	RKO	\$267.40	1942
The Godfather	604	Paramount	\$245.10	1972
101 Dalmatians	708	Buena Vista	\$215.90	1961
The Jungle Book	755	Buena Vista	\$205.80	1967

# Filtering a Column by One or More Criteria

```
In [24] time_range = movies["Year"].between(1983, 1986)  
      movies[time_range]
```

```
Out [24]
```

Title	Rank	Studio	Gross	Year
Return of the Jedi	222	Fox	\$475.10	1983
Back to the Future	311	Universal	\$381.10	1985
Top Gun	357	Paramount	\$356.80	1986
Indiana Jones and the Temple of Doom	403	Paramount	\$333.10	1984
Crocodile Dundee	413	Paramount	\$328.20	1986
Beverly Hills Cop	432	Paramount	\$316.40	1984
Rocky IV	467	MGM	\$300.50	1985
Rambo: First Blood Part II	469	TriStar	\$300.40	1985
Ghostbusters	485	Columbia	\$295.20	1984
Out of Africa	662	Universal	\$227.50	1985

# Filtering a Column by One or More Criteria

```
In [25] has_dark_in_title = movies.index.str.lower().str.contains("dark")
       movies[has dark in title]
```

```
Out [25]
```

Title	Rank	Studio	Gross	Year
Transformers: Dark of the Moon	23	Paramount	\$1,123.80	2011
The Dark Knight Rises	27	Warner Brothers	\$1,084.90	2012
The Dark Knight	39	Warner Brothers	\$1,004.90	2008
Thor: The Dark World	132	Buena Vista	\$644.60	2013
Star Trek Into Darkness	232	Paramount	\$467.40	2013
Fifty Shades Darker	309	Universal	\$381.50	2017
Dark Shadows	600	Warner Brothers	\$245.50	2012
Dark Phoenix	603	Fox	\$245.10	2019

# Grouping Data

```
In [26] movies["Gross"].str.replace("$", "").str.replace(", ", "")
```

```
Out [26] Title
```

Avengers: Endgame	2796.30
Avatar	2789.70
Titanic	2187.50
Star Wars: The Force Awakens	2068.20
Avengers: Infinity War	2048.40
...	
Yogi Bear	201.60
Garfield: The Movie	200.80
Cats & Dogs	200.70
The Hunt for Red October	200.50
Valkyrie	200.30
Name: Gross, Length: 782, dtype: object	

```
In [27]:  
    movies["Gross"].str.replace("$", "")  
        .str.replace(",","")  
        .astype(float)  
)
```

```
Out [27]: Title  
Avengers: Endgame      2796.3  
Avatar                2789.7  
Titanic               2187.5  
Star Wars: The Force Awakens 2068.2  
Avengers: Infinity War 2048.4  
...  
Yogi Bear              201.6  
Garfield: The Movie   200.8  
Cats & Dogs            200.7  
The Hunt for Red October 200.5  
Valkyrie               200.3  
Name: Gross, Length: 782, dtype: float64
```



# Grouping Data

---

- All of these operations are temporary and do not mutate the original Gross Series.
- Each time, pandas creates a copy of the original data, performs the operation, and returns the resulting object.
- We have to explicitly overwrite the Gross column with the new Series object to make these changes permanent.

```
In [28] movies["Gross"] = (movies["Gross"].str.replace("$", "")  
                         .str.replace(",","", "")  
                         .astype(float))
```



# Grouping Data



- Let's invoke the mean method to find the average box office gross of a movie on the list -- it's over 439 million dollars!

```
In [29] movies["Gross"].mean()
```

```
Out [29] 439.0308184143222
```



# Grouping Data



First, we group the values in **Studio**, which organizes all unique values from the column into their own clump.

```
In [30] studios = movies.groupby("Studio")
```

We can now add the totals of the **Gross** column per studio.

```
In [31] studios["Gross"].sum().head()
```

```
Out [31] Studio
        Artisan           248.6
        Buena Vista      73585.0
        CL                228.1
        China Film Corporation  699.8
        Columbia          1276.6
Name: Gross, dtype: float64
```



# Grouping Data

```
In [32] studios["Gross"].sum().sort_values(ascending = False).head()

Out [32] Studio
        Buena Vista      73585.0
        Warner Brothers   58643.8
        Fox                50420.8
        Universal          44302.3
        Sony               32822.5
Name: Gross, dtype: float64
```



# Summary



- Pandas can import a variety of different file formats including CSV, XLSX, and more.
  - Any row in a dataset can be accessed by its row number or an identifier.
  - Datasets can be sorted by one or more columns.
  - One or more conditions can be used to extract a subset of data from a larger dataset.
  - Pandas can aggregate data by grouping together identical values across a column.
- 

# Lesson 3 Python Crash Course





# Python Crash Course



This session covers:

- Core data types in Python including strings, integers, and floats
  - Built-in functions including len, str, float, and int
  - Custom functions with parameters and return values
  - The list, tuple, dictionary, and set data structures
  - Importing modules and packages into our program
- 



# Simple Data Types

- Data can be stored in a variety of different types. For example, a whole number like 5 is of a different type than a decimal point number like 5.46.
- Similarly, the aforementioned numeric values are different from a text value like "Bob".
- Part of a data analyst's skillset is identifying what type of data is needed to perform the desired analysis.



# Numbers



- An integer is a whole number. It has no fractional or decimal component.

In [1] 20

Out [1] 20

- An integer can be any positive number, negative number, or zero. Negative numbers begin with a - sign.

In [2] -13

Out [2] -13



# Numbers



- A floating-point number (colloquially called a float) is a number with a fractional or decimal component.
- A dot is used to create a decimal point.

In [3] 7.349

Out [3] 7.349



# Strings



```
In [4] 'Good morning'  
Out [4] 'Good morning'  
  
In [5] "Good afternoon"  
Out [5] 'Good afternoon'  
  
In [6] """Good night"""  
Out [6] 'Good night'
```



# Strings



- Use the presence of quotes as a visual identifier for a string. Many beginners are confused by a value like "5", which is a string even though it holds a numeric character.
- An empty string is a string without characters.
- It's created with a pair of quotes with nothing between them.

In [8] ""

Out [8] "



# Strings



- Any character in a string can be extracted by its index position.
- After the string, write a pair of square brackets containing the index.
- In the next example, we pull out the "h" from "Python". "h" is the fourth character in sequence which means it has an index position of 3.

In [9] "Python"[3]

Out [9] 'h'



# Strings



- A negative value within the square brackets will pull relative to the end of the string.
- For example, a value of -1 will extract the last character, -2 will extract the second-to-last character, and so on.
- In the next example, we target the "t" in "Python", the fourth-to-last character.

```
In [10] "Python"[-4]  
Out [10] 't'
```





# Strings



- Note that the starting index is inclusive, which means its character will be included, while the ending index is exclusive, which means its character will be excluded

```
In [11] # Pull all characters from index position 2 up to
      # (but not including) index position 5. The character
      # "t" is at index position 2, the "h" is at 3, and
      # the "o" is at 4.

      "Python"[2:5]

Out [11] 'tho'
```



# Strings



- If the starting index is 0, it can be removed entirely from the square brackets.

```
In [12] "Python"[0:4]
        # is the same as
"Python)[:4]
```

```
Out [12] 'Pyth'
```



# Strings



- The end index can be removed if we want to extract from a starting index position until the end of a string.
- The example below shows two options for pulling out the characters from "h" (index position 3) to the end of the "Python" string.

```
In [13] "Python"[3:6]
        # is the same as
        "Python"[3:]
```

```
Out [13] 'hon'
```



# Strings



- Positive and negative index positions can be mixed and matched in a string slice.
- Let's pull from index position 1 ("y") up to the last character in the string ("n").

```
In [14] "Python"[1:-1]  
Out [14] 'ytho'
```



# Strings



- Index position 6 is excluded because the "Python" string's last index position is 5.

```
In [15] "Python"[0:6:2]
```

```
Out [15] 'Pto'
```



# Booleans



```
In [16] True
```

```
Out [16] True
```

```
In [17] False
```

```
Out [17] False
```

# Operators

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulo (remainder)



# Mathematical Operators



- The symbols in the table above are called operators.
- The value(s) that the operator works on are called operands.
- In the next example, + is the operator, and 3 and 5 are the operands.

In [18]  $3 + 5$

Out [18] 8



# Mathematical Operators

```
In [19] 3 - 5
```

```
Out [19] -2
```

```
In [20] 3 * 5
```

```
Out [20] 15
```

```
In [21] 3 ** 5      # 3 to the power of 5
```

```
Out [21] 243
```

```
In [22] 3 / 5
```

```
Out [22] 0.6
```



# Mathematical Operators



- The modulo operator ( % ) returns the remainder of a division. In the next example, 2 is the remainder of dividing 5 by 3.

```
In [23] 5 % 3
```

```
Out [23] 2
```

- Division of any two numeric values will return a floating-point number even if one value is evenly divisible by the other.

```
In [24] 4 / 1
```

```
Out [24] 4.0
```



# Mathematical Operators

---

- In mathematical lingo, the quotient is the result from dividing one number by another.
- Floor division is an alternate type of division that removes the decimal remainder from a quotient.
- It is performed with two forward slashes ( // ) and always returns an integer.

In [25] 8 / 3

Out [25] 2.6666666666666665

In [26] 8 // 3

Out [26] 2



# Mathematical Operators

---

- The addition and multiplication operators can also be used with strings.
- The plus sign combines two strings together into a new one.
- The technical word for this process is concatenation.

In [27] "race" + "car"

Out [27] 'racecar'

- The multiplication sign repeats a string a given number of times.

In [28] "Mahi" \* 2

Out [28] 'MahiMahi'



# Mathematical Operators

- A string can be concatenated to another string, and a number can be added to another number.
- But what happens when we try to add a string and a number together?

```
In [29]: 3 + "5"

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-9-d4e36ca990f8> in <module>
----> 1 3 + "5"

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



# Equality and Inequality Operators

```
In [30] 10 == 10
Out [30] True

In [31] 10 == 20
Out [31] False

In [32] "Hello" == "Hello"
Out [32] True

In [33] "Hello" == "Goodbye"
Out [33] False
```



# Equality and Inequality Operators



- Case sensitivity matters in comparisons of string equality.
- In the next example, one string starts with a capital "H" while the other starts with a lowercase "h".
- Python considers the two unequal.

```
In [34] "Hello" == "hello"
```

```
Out [34] False
```



# Equality and Inequality Operators

```
In [35] 10 != 20
Out [35] True

In [36] "Hello" != "Goodbye"
Out [36] True

In [37] 10 != 10
Out [37] False

In [38] "Hello" != "Hello"
Out [38] False
```



# Equality and Inequality Operators



```
In [39] -5 < 3      # Less than  
Out [39] True  
  
In [40] 5 > 7      # Greater than  
Out [40] False  
  
In [41] 11 <= 11. # Less than or equal to  
Out [41] True  
  
In [42] 4 >= 5.    # Greater than or equal to  
Out [42] False
```



# Variables

---

- A variable is assigned to an object with the assignment operator, a single equal sign ( = ).
- The next example assigns four variables (name, age, high\_school\_gpa, and is\_handsome) to four different data types (string, integer, floating-point, and Boolean respectively).

```
In [43] name = "Boris"
```

```
age = 28
```

```
high_school_gpa = 3.7
```

```
is_handsome = True
```



# Variables

---

- The execution of a Jupyter Notebook cell with a variable assignment does not yield any cell output.
- However, the variable is then available to be used in any cell.

In [44] name

Out [44] 'Boris'



# Variables



- As their name suggests, variables can vary over the course of a program's execution.
- Let's reassign the age variable to a new value of 35. After the cell executes, the variable's reference to its former value, 28, will be lost.

```
In [45] age = 35
```

```
age
```

```
Out [45] 35
```



# Variables



- The result, 45, is then assigned to the age variable.
- Finally, we output the most up-to-date value of age.

In [46] age = age + 10

age

Out [46] 45



# Variables



- Variables can thus be reassigned from an object of one type to another.
- In the example below, the `high_school_gpa` variable is reassigned from its original floating-point value of 3.7 to a string of "A+".

```
In [47] high_school_gpa = "A+"
```



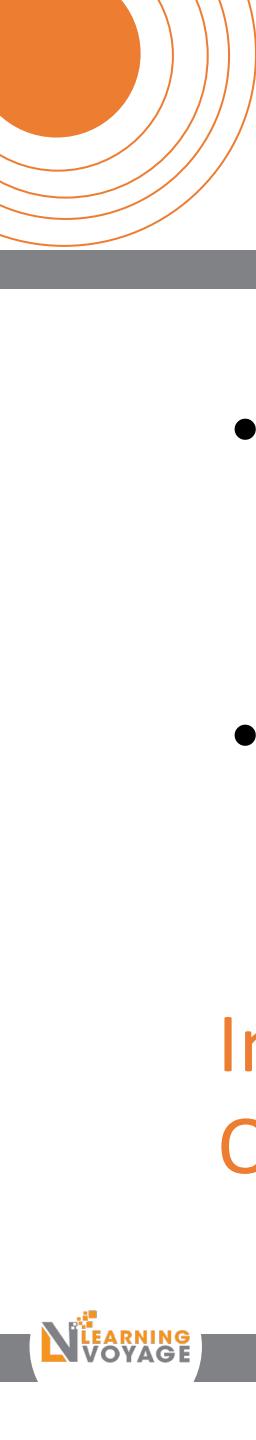
# Variables



- A `NameError` exception is raised when a variable does not exist in the program.
- This can happen when a variable is undeclared or mistyped by the user.

```
In [48]: last_name
-----
NameError                                 Traceback (most recent call last)
<ipython-input-5-e1aeda7b4fde> in <module>
----> 1 last_name

NameError: name 'last_name' is not defined
```



# Functions

- The `len` function expects one argument, the object whose length it should calculate.
- The next example passes the function a string argument of "Python is fun".

```
In [49] len("Python is fun")  
Out [49] 13
```



# Functions



Three more popular built-in functions in Python:

- `int`, which converts its argument to an integer
- `float`, which converts its argument to a floating-point number
- `str`, which converts its argument to a string



# Functions



```
In [50] int("20")
```

```
Out [50] 20
```

```
In [51] float("14.3")
```

```
Out [51] 14.3
```

```
In [52] str(5)
```

```
Out [52] '5'
```

# Functions

```
In [53]: int("xyz")
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-6-ed77017b9e49> in <module>  
----> 1 int("xyz")
```

```
ValueError: invalid literal for int() with base 10: 'xyz'
```



# Functions



```
In [54] value = 10  
       print(value)  
  
       value = value - 3  
       print(value)  
  
       value = value * 4  
       print(value)  
  
       value = value / 2  
       print(value)  
  
Out [54] 10  
        7  
        28  
        14
```



# Functions



- If a function accepts multiple arguments, every two subsequent ones must be separated by a comma.
- A space is often added after the comma for readability.
- When the print function is passed multiple arguments, it outputs all of them in sequence.
- In the next example, notice how all the three printed elements are separated by a space. We'll come back to that shortly.

In [55] `print("Cherry", "Strawberry", "Key Lime")`

Out [55] Cherry Strawberry Key Lime



# Functions

---

- Certain arguments require their parameter names to be explicitly written out.
- For example, the `sep` (separator) parameter to the `print` function customizes the string that is inserted in between every two subsequent printed values.
- We have to explicitly write out its name when invoking the `print` function. An argument is assigned to a keyword parameter with an equal sign.

```
In [56] print("Cherry", "Strawberry", "Key Lime", sep = "!")  
Out [56] Cherry!Strawberry!Key Lime
```



# Functions



- The two lines of code below thus produce the same output.

```
In [57] print("Cherry", "Strawberry", "Key Lime")
        print("Cherry", "Strawberry", "Key Lime", sep=" ")
Cherry Strawberry Key Lime
Cherry Strawberry Key Lime
```



# Functions



- Here's another example of a print function invocation with a different string argument passed to the sep parameter.

```
In [58] print("Cherry", "Strawberry", "Key Lime", sep="*!*")  
Out [58] Cherry*!*Strawberry*!*Key Lime
```



# Functions



- In the next example, we explicitly pass the same "\n" argument manually to the end parameter.

```
In [59] print("Cherry", "Strawberry", "Key Lime, end="\n")
print("Peach Cobbler")
Out [59] Cherry Strawberry Key Lime
Peach Cobbler
```



# Functions

- In the first invocation of the print function below, we separate its three elements with a "!" and end the output with a "\*\*\*".
- Because there is no line break, the second invocation of the print function continues where the first one left off

```
In [60] print("Cherry", "Strawberry", "Key Lime", sep="!", end="***")
       print("Peach Cobbler")
```

```
Cherry!Strawberry!Key Lime***Peach Cobbler
```



# Custom Functions

---

- Developers can declare custom functions in their programs as well.
  - The goal of each function should be to encapsulate a distinct piece of business logic in a single procedure.
  - A common mantra in software engineering circles is DRY, an acronym for "Don't Repeat Yourself".
  - It is a warning that duplication of the same logic or behavior can lead to a more unstable program.
  - The more places that a piece of code is repeated, the more places that you have to edit if the requirements change.
- 



# Custom Functions

- The `def` keyword defines a function. It is followed by a name, a pair of opening and closing parentheses, and a colon.
- Function and variable names with multiple words follow a `snake_case` naming convention which separates every two words with an underscore.
- The convention is inspired by the fact that the name visually resembles a slithering snake.

```
def convert_to_fahrenheit():
```



# Custom Functions



- As a review, a parameter is a name assigned to an expected function argument.
- We want to allow the convert\_to\_fahrenheit function to accept a single parameter, the temperature in Celsius.
- Let's call it celsius\_temp.

```
def convert_to_fahrenheit(celsius_temp):
```



# Custom Functions



Let's write out the logic for the conversion! The formula to convert a Celsius temperature to Fahrenheit is to multiply it by 9/5 and add 32 to the result.

```
def convert_to_fahrenheit(celsius_temp):  
    first_step = celsius_temp * (9 / 5)  
    fahrenheit_temperature = first_step + 32
```



# Custom Functions

- We need to use the return keyword to specify the Fahrenheit temperature as the final output from the function.

```
In [61] def convert_to_fahrenheit(celsius_temp):  
        first_step = celsius_temp * (9 / 5)  
        fahrenheit_temperature = first_step + 32  
    return fahrenheit_temperature
```



# Custom Functions

- We'll invoke the `convert_to_fahrenheit` function and pass in a sample argument of 10.
- Python will run through the body of the function with the `celsius_temp` parameter set to 10.
- The function returns a value of 50.0.

```
In [62] convert_to_fahrenheit(10)  
Out [62] 50.0
```



# Custom Functions



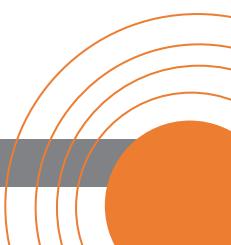
- We can also write out the parameter names explicitly.
- The code below is logically equivalent to the code previous.

```
In [63] convert_to_fahrenheit(celsius_temp = 10)  
Out [63] 50.0
```



# Objects and Methods

---

- All of the data types that we've explored so far -- integers, floats, Booleans, strings, exceptions, even functions -- are objects.
  - As a reminder, an object is a digital data structure, a container for storing, accessing, and manipulating some kind of information.
  - As programs grow in complexity, it becomes difficult to model real-world ideas with Python's base data types.
- 



# Attributes

Objects consist of attributes and methods. The former represents the object's state while the latter represents the object's behavior. An attribute is a characteristic or detail about the object. Imagine we were trying to model real world objects in Python. Here are a few examples:

- a Basketball object might have attributes like color, weight, and circumference
- a Restaurant object might have attributes like name, cuisine, and address
- a Car object might have attributes like year, model, and `is_used`



# Methods



A method is an action or command sent to an object. It can be described as a function that belongs to an object, a procedure that acts upon a data structure. A method may manipulate an object's state by altering the values of one or more of its attributes. If we were modelling the same set of real-world objects in Python,

- a Basketball object might have methods like `throw`, `shoot`, and `pass`
  - a Restaurant object might have attributes like `open`, `close`, and `deliver_food`
  - a Car object might have attributes like `drive`, `fill_up_tank`, and `park`
- 



# Methods



- We access a method on an object with a dot followed by the method name.
- Methods are invoked with a pair of parentheses, exactly like a function.

```
In [64] "Hello".upper()  
Out [64] "HELLO"
```



# Methods



- A variable is just a placeholder for a Python object.
- In the next example, we invoke the upper method on the string that the greeting variable is representing.
- The output is identical.

```
In [65] greeting = "Hello"  
      greeting.upper()  
Out [65] "HELLO"
```



# Methods



- We can prove this by outputting the value of greeting after the invocation of the upper method.
- It still has its original character casing.

In [66] greeting  
Out [66] 'Hello'



# Methods



- The replace method's two sequential arguments are (a) the substring to look for and (b) the value to replace all occurrences of it with.
- The next example swaps all occurrences of the letter "S" with the character "\$".

In [67] "Sally Sells Seashells by the Seashore".replace("S", "\$")

Out [67] '\$ally \$ells \$eashells by the \$eashore'



# Methods

---

- The return value does not have to be of the same data type as the object the method is invoked upon.
- For example, the `isspace` method is invoked on a string but returns a Boolean.
- It returns `True` if the string consists of only spaces and `False` otherwise.

```
In [68] ".isspace()
```

```
Out [68] True
```

```
In [69] "3 Amigos".isspace()
```

```
Out [69] False
```



# Additional String Methods



- It is important for Pandas users to learn how to effectively manipulate strings because text data can arrive in a variety of distorted forms.
- We saw an example of `upper` in the previous section.
- The complementary `lower` method returns a new string with all of the characters lowercased.

```
In [70] "1611 BROADWAY".lower()
```

```
Out [70] '1611 broadway'
```



# Additional String Methods

---

- There's even a swapcase method that returns a new string with the case of each character inverted.
- Uppercase letters become lowercase, and lowercase letters become uppercase.

```
In [71] "uPsIdE dOwN".swapcase()  
Out [71] 'UpSiDe DoWn'
```



# Additional String Methods

```
In [72] data = " 10/31/2019 "
       data.rstrip()
```

```
Out [72] ' 10/31/2019'
```

```
In [73] data.lstrip()
```

```
Out [73] '10/31/2019 '
```

```
In [74] data.strip()
```

```
Out [74] '10/31/2019'
```



# Additional String Methods

---

- The capitalize method capitalizes the first letter of a string.
- This often proves helpful when dealing with names, places or organizations.

In [75] "robert".capitalize()

Out [75] 'Robert'

- The title method capitalizes the first letter of every word in a string. It uses the presence of a space to distinguish where each word begins and ends.

In [76] "once upon a time".title()

Out [76] 'Once Upon A Time'



# Additional String Methods

---

- Multiple methods can be invoked in sequence on a single line. This is called method chaining.
- In the next example, the lower method returns a new string object upon which the title method is invoked.
- The return value is yet another string object.

```
In [77] "BENJAMIN FRANKLIN".lower().title()  
Out [77] 'Benjamin Franklin'
```



# Additional String Methods



```
In [78] "tuna" in "fortunate"
```

```
Out [78] True
```

```
In [79] "salmon" in "fortunate"
```

```
Out [79] False
```



# Additional String Methods

---

- The starts with and ends with methods check for a substring at the end or beginning of a string.

```
In [80] "factory".startswith("fact")
```

```
Out [80] True
```

```
In [81] "garage".endswith("rage")
```

```
Out [81] True
```



# Additional String Methods



- The count method returns the number of times a substring occurs within a string.
- Here, we could the number of "e" letters present in "celebrate".

```
In [82] "celebrate".count("e")  
Out [82] 3
```



# Additional String Methods

---

- The find and index methods return the first index position in which a substring is found.
- Remember that index positions start counting at 0.

```
In [83] "celebrate".find("e")
```

```
Out [83] 1
```

```
In [84] "celebrate".index("e")
```

```
Out [84] 1
```



# Additional String Methods



```
In [85] "celebrate".find("z")
```

```
Out [85] -1
```

```
In [86] "celebrate".index("z")
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-5-bf78a69262aa> in <module>
----> 1 "celebrate".index("z")
```

```
ValueError: substring not found
```



# Lists



- A list is declared by placing objects between a pair of opening and closing square brackets.
- Every two subsequent elements are separated with a comma. The next example creates a list of 5 strings in order.

```
In [87] backstreet_boys = ["Nick", "AJ", "Brian", "Howie",  
"Kevin"]
```



# Lists



The length of a list is equal to its number of elements. Remember the trusty `len` function? It can help us figure how many members are in the greatest boy band of all time.

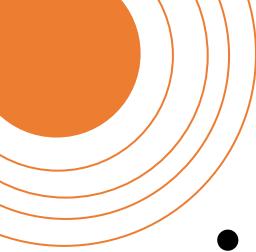
```
In [88] len(backstreet_boys)
```

```
Out [88] 5
```

An empty list is a list without elements. It has a length of 0.

```
In [89] []
```

```
Out [89] []
```



# Lists

- In the three-item favorite\_foods list below, the string "Sushi" occupies index position 0, the string "Steak" occupies index position 1, and the string "Barbeque" occupies index position 2.

In [90] favorite\_foods = ["Sushi", "Steak", "Barbeque"]

- A list element is accessed by its index position. Pass the index in between a pair of square brackets after the list (or the variable that represents it).

In [91] favorite\_foods[1]

Out [91] 'Steak'



# Lists

```
In [92] favorite_foods[1:3]
Out [92] ['Steak', 'Barbeque']

In [93] favorite_foods[:2]
Out [93] ['Sushi', 'Steak']

In [94] favorite_foods[2:3]
Out [94] ['Barbeque']

In [95] favorite_foods[0:3:2]
Out [95] ['Sushi', 'Barbeque']
```





# Lists



- The append method adds a new element to the end of a list.

```
In [96] favorite_foods.append("Burrito")  
      favorite_foods
```

```
Out [96] ['Sushi', 'Steak', 'Barbeque', 'Burrito']
```



# Lists



- Lists include a variety of mutational methods.
- The extend method accepts a list as an argument.
- It adds all the elements from the argument list to the end of the list that the method is called upon.

```
In [97] favorite_foods.extend(["Tacos", "Pizza", "Cheeseburger"])
favorite_foods

Out [97] ['Sushi', 'Steak', 'Barbeque', 'Burrito', 'Tacos', 'Pizza',
'Cheeseburger']
```



# Lists



```
In [97] favorite_foods.insert(2, "Pasta")
        favorite_foods
```

```
Out [97] ['Sushi',
           'Steak',
           'Pasta',
           'Barbeque',
           'Burrito',
           'Tacos',
           'Pizza',
           'Cheesburger']
```



# Lists



- The `in` keyword checks for the inclusion of an element within a list. "Pizza" exists in our `favorite_foods` list while "Caviar" does not.

```
In [98] "Pizza" in favorite_foods
```

```
Out [98] True
```

```
In [99] "Caviar" in favorite_foods
```

```
Out [99] False
```



# Lists



- The `not in` keyword checks for exclusion or, in other words, the absence of an element within a list.
- It returns the inverse of the `in` operator.

```
In [100] "Pizza" not in favorite_foods
```

```
Out [100] False
```

```
In [101] "Caviar" not in favorite_foods
```

```
Out [101] True
```



# Lists



- The count method returns the number of times an element appears in the list.

```
In [102] favorite_foods.append("Pasta")
         print(favorite_foods)
```

```
['Sushi', 'Steak', 'Pasta', 'Barbeque', 'Burrito', 'Tacos',
 'Pizza', 'Cheeseburger', 'Pasta']
```

```
In [103] favorite_foods.count("Pasta")
```

```
Out [103] 2
```



# Lists

---

- The remove method deletes the first occurrence of an element from the list.
- Note that subsequent occurrences will not be removed.

```
In [104] favorite_foods.remove("Pasta")
          favorite_foods
```

```
Out [104] ['Sushi',
            'Steak',
            'Barbeque',
            'Burrito',
            'Tacos',
            'Pizza',
            'Cheeseburger',
            'Pasta']
```



# Lists



- Let's target the other "Pasta" value at the end of the list.
- The pop method removes and returns the last element from the list.

```
In [105] favorite_foods.pop()  
Out [105] 'Pasta'  
  
In [106] favorite_foods  
Out [106] ['Sushi', 'Steak', 'Barbeque', 'Burrito', 'Tacos', 'Pizza',  
          'Cheeseburger']
```



# Lists



- The pop method also accepts an integer argument with the index position whose value should be deleted.
- The next example removes the "Barbeque" value at index position 2.

```
In [107] favorite_foods.pop(2)

Out [107] 'Barbeque'

In [108] favorite_foods

Out [108] ['Sushi', 'Steak', 'Burrito', 'Tacos', 'Pizza', 'Cheeseburger']
```



# Lists



- A list can hold any object including other lists. Our next list holds three nested lists, each of which contain 3 integers.

```
In [109] spreadsheet = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
]
```



# List Iteration

---

- To iterate means to move over an object's elements one at a time. For example, we can iterate over the individual elements of a list in sequence.
- This traversal of a list's items is accomplished with a for in loop. The syntax looks like this:

```
In [110]: for season in ["Winter", "Spring", "Summer", "Fall"]:  
    print(season)
```

```
Winter  
Spring  
Summer  
Fall
```



# List Iteration



- In the next example, we use list iteration to sum up the lengths of the strings in the same list.
- Inside the for loop block, we calculate the length of an individual string, then add it to a running total.

```
In [111] letter_count = 0

        for season in ["Winter", "Spring", "Summer", "Fall"]:
            letter_count = letter_count + len(season)

        letter_count

Out [111] 22
```



# List Comprehension



- List comprehension is a shorthand syntax to create a list from another iterable object.
- Most often, it's used to create a list from another list.
- Imagine we have a list of 6 numbers.

In [112] numbers = [4, 8, 15, 16, 23, 42]





# List Comprehension



```
In [113] squares = []

    for number in numbers:
        squares.append(number ** 2)

squares

Out [113] [16, 64, 225, 256, 529, 1764]
```



# List Comprehension

In the next example, we still iterate over the `numbers` list and assign each list element to a number variable. We declare what we'd like to do with each number before the `for` keyword. With list comprehension, the calculation is moved to the beginning and the `for` in logic is moved to the end.

```
In [114] squares = [number ** 2 for number in numbers]
          squares
```

```
Out [114] [16, 64, 225, 256, 529, 1764]
```

List comprehension is considered the more "Pythonic" way to perform an operation like this. The "Pythonic" way is the collection of best practices adopted by Python developers over time.

# Converting a String to a List and Vice Versa

- The split method slices a string into multiple strings based on occurrences of a delimiter, a sequence of one or more characters that marks a boundary.
- Let's say we receive a string in our program representing an address:

```
In [115] empire_state_bldg = "20 West 34th Street, New  
York, NY, 10001"
```

# Converting a String to a List and Vice Versa

- This string uses a comma as a delimiter to separate the street, the city, the state and the zip code.
- With the split method, we can return a list consisting of these individual elements.

```
In [116] empire_state_bldg.split(",")
```

```
Out [116] ['20 West 34th Street', ' New York', ' NY', ' 10001']
```

# Converting a String to a List and Vice Versa

- While we could iterate over the list's elements and call the strip on each one to remove its whitespace, a more optimal solution is to add the space to our delimiter argument to split.

In [117] empire\_state\_bldg.split("", "")

Out [117] ['20 West 34th Street', 'New York', 'NY', '10001']

# Converting a String to a List and Vice Versa

The process also works in reverse. We can concatenate the elements of a list into a single string. Imagine we had our address stored in a list.

```
In [118] chrysler_bldg = ["405 Lexington Ave", "New York", "NY", "10174"]
```

Invoke the `join` method on a string specifying the delimiter and pass in a list as the argument. All of the list's elements will be *joined* together, separated by the delimiter.

```
In [119] ", ".join(chrysler_bldg)
```

```
Out [119] '405 Lexington Ave, New York, NY, 10174'
```



# Tuples



- A tuple is essentially an immutable list. It is a data structure for storing objects in order that cannot have elements added to it or removed from it after creation.
- The only technical requirement for declaring a tuple is separating multiple elements with commas.

In [120] "Rock", "Pop", "Country"

Out [120] ('Rock', 'Pop', 'Country')



# Tuples



- Usually, however, a tuple will be declared with a pair of parentheses.
- It makes it easier to identify the data structure.

In [121] ("Rock", "Pop", "Country")

Out [121] ('Rock', 'Pop', 'Country')



# Dictionaries



- A dictionary is a Python object that is used to model the same idea.
- It is a mutable, unordered collection of key-value pairs.
- Each key serves as a identifier for a value. Keys must be unique while values can contain duplicates.
- A dictionary is declared with a pair of curly braces ( {} ).  
The example below creates an empty dictionary.

In [122] {}

Out [122] {}



# Dictionaries



- Let's create a sample restaurant menu in Python. Inside the curly braces, we can assign a key to its value with a colon.
- The example below declares a dictionary with one key-value pair.
- The string key "Cheeseburger" is assigned to the floating-point value 7.99.

```
In [123] { "Cheeseburger": 7.99 }  
Out [123] {'Cheeseburger': 7.99}
```



# Dictionaries

When declaring a dictionary with multiple key-value pairs, subsequent pairs must be separated by commas. Let's expand our dictionary to hold 3 key-value pairs. Notice that the value for the "French Fries" and "Soda" key is identical.

```
In [124] menu = {"Cheeseburger": 7.99, "French Fries": 2.99, "Soda": 2.99}  
menu
```

```
Out [124] {'Cheeseburger': 7.99, 'French Fries': 2.99, 'Soda': 2.99}
```

We can count the number of key-value pairs in a dictionary by passing it to Python's built-in **len** function.

```
In [125] len(menu)
```

```
Out [125] 3
```



# Dictionaries

- A key is used to retrieve a value from a dictionary. Place a pair of square brackets containing the key immediately after the dictionary.
- This is an identical syntax to accessing a character in a string or an element in a list by its index position.
- With a dictionary, however, the keys can be of any immutable data type: integers, floats, strings, Booleans, and more.
- The example below extracts the value for the "French Fries" key.

```
In [126] menu["French Fries"]  
Out [126] 2.99
```



# Dictionaries



- A `KeyError` exception will be raised if the key does not exist in the dictionary.
- This is yet another error built into Python.

```
In [127]: menu["Steak"]

-----
KeyError                                     Traceback (most recent call last)
<ipython-input-19-0ad3e3ec4cd7> in <module>
----> 1 menu["Steak"]

KeyError: 'Steak'
```



# Dictionaries



- As always, case sensitivity matters. If a single character is mismatched, Python will not be able to find it.

```
In [128]: menu["soda"]

-----
KeyError                                     Traceback (most recent call last)
<ipython-input-20-47940ceca824> in <module>
----> 1 menu["soda"]

KeyError: 'soda'
```



# Dictionaries



- A dictionary is a mutable data structure. Key-value pairs can be added to or removed from the dictionary after it has been created.
- To add a new pair, provide the key in square brackets and assign a value to it with the assignment operator ( = ).

```
In [129] menu["Taco"] = 0.99  
menu
```

```
Out [129] {'Cheeseburger': 7.99, 'French Fries': 2.99, 'Soda': 1.99,  
'Taco': 0.99}
```



# Dictionaries

---

- If the key already exists in the dictionary, its value will be overwritten.
- The example below overwrites the value of the "Cheeseburger" key from 7.99 to 9.99.

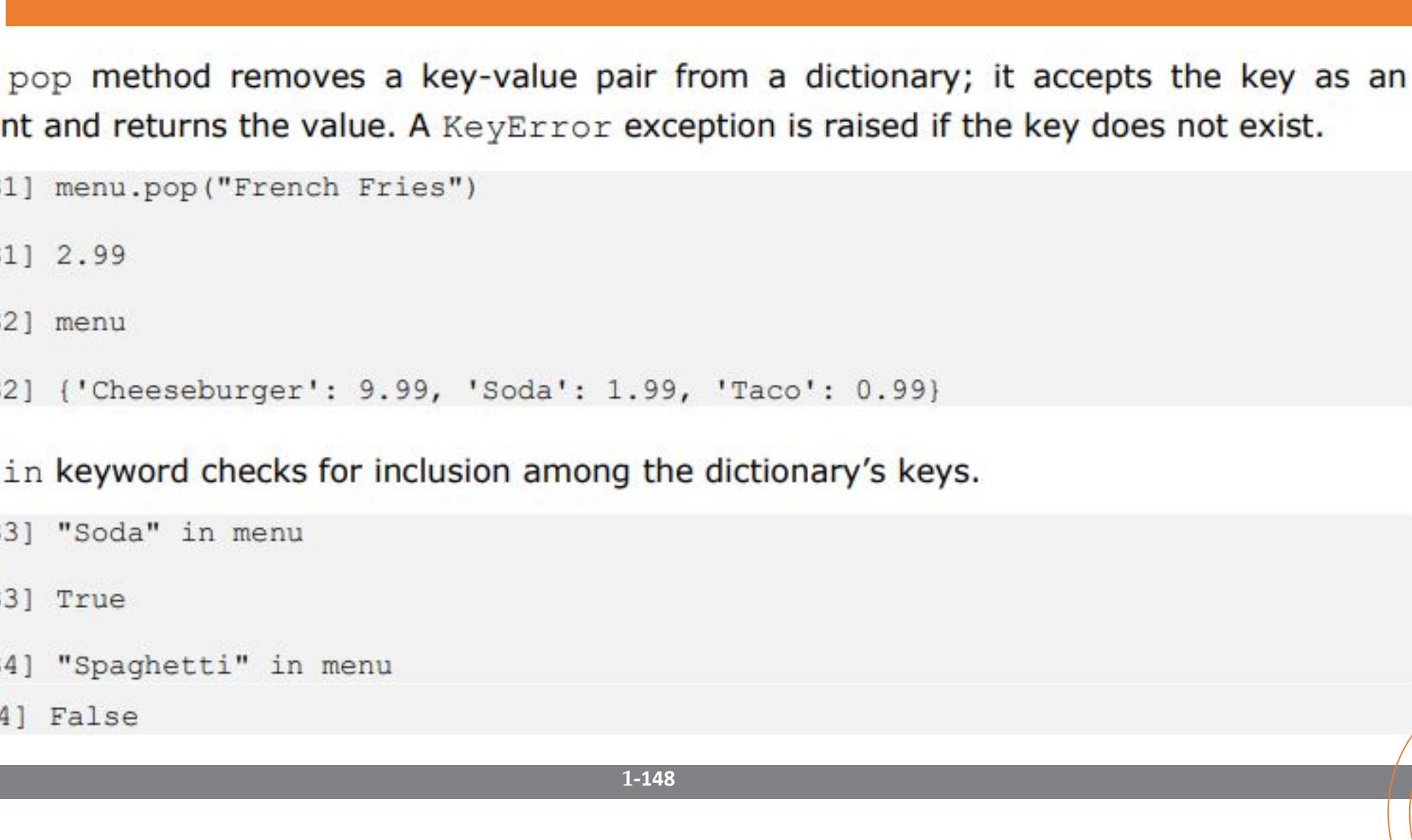
```
In [130] print(menu["Cheeseburger"])

menu["Cheeseburger"] = 9.99
print(menu["Cheeseburger"])

Out [129] 7.99
         9.99
```



# Dictionaries



The `pop` method removes a key-value pair from a dictionary; it accepts the key as an argument and returns the value. A `KeyError` exception is raised if the key does not exist.

```
In [131] menu.pop("French Fries")  
Out [131] 2.99  
  
In [132] menu  
  
Out [132] {'Cheeseburger': 9.99, 'Soda': 1.99, 'Taco': 0.99}
```

The `in` keyword checks for inclusion among the dictionary's keys.

```
In [133] "Soda" in menu  
Out [133] True  
  
In [134] "Spaghetti" in menu  
Out [134] False
```



# Dictionaries



- The in operator can be used in combination with the return value from the values method.

```
In [135] 1.99 in menu.values()
```

```
Out [135] True
```

```
In [136] 499.99 in menu.values()
```

```
Out [136] False
```



# Dictionary Iteration



```
In [137]: capitals = {  
    "New York": "Albany",  
    "Florida": "Tallahassee",  
    "California": "Sacramento"  
}  
  
for state, capital in capitals.items():  
    print("The capital of " + state + " is " + capital + ".")
```

The capital of New York is Albany.  
The capital of Florida is Tallahassee.  
The capital of California is Sacramento.





# Sets



- List and dictionary objects help solve the problems of order and association.
- One additional problem that data analysts frequently have to deal with is uniqueness.
- A set is an unordered, mutable collection of elements that prohibits duplicates.
- It's created by a pair of curly braces. Subsequent elements within the braces are separated by commas.

In [138] favorite\_numbers = { 4, 8, 15, 16, 23, 42 }



# Sets



- The only way to create an empty set is with the built-in set function.
- This is because Python will interpret a pair of curly braces as an empty dictionary.

In [139] `set()`

Out [139] `set()`



# Sets



The add method adds a new element into the set.

```
In [140] favorite_numbers.add(100)
         favorite_numbers

Out [140] {4, 8, 15, 16, 23, 42, 100}
```

An element will only be added to a set if it is not already contained within it. The example below attempts to adds 15, which is an existing value in the set. Python will not raise an exception, but the set will remain unchanged.

```
In [141] favorite_numbers.add(15)
         favorite_numbers

Out [141] {4, 8, 15, 16, 23, 42, 100}
```



# Sets



- The elements of a set are unordered.
- Attempting to access an element by index position will lead to a `TypeError` exception

```
In [142]: favorite_numbers[2]
```

```
-----
TypeError                                                 Traceback (most recent call last)
<ipython-input-17-e392cd51c821> in <module>
----> 1 favorite_numbers[2]

TypeError: 'set' object is not subscriptable
```



# Set Operations



- In addition to uniqueness, sets are ideal for identifying similarities and differences between multiple collections of data.
- Let's define two sets of strings.

```
In [143] candy_bars = { "Milky Way", "Snickers", "100 Grand"  
}  
sweet_things = { "Sour Patch Kids", "Reeses Pieces",  
"Snickers" }
```





# Set Operations



- The intersection method returns a new set with elements found in both of the original sets.
- The & symbol performs the same logic.

```
In [144] candy_bars.intersection(sweet_things)
```

```
Out [144] {'Snickers'}
```

```
In [145] candy_bars & sweet_things
```

```
Out [145] {'Snickers'}
```



# Set Operations



- The union method returns a set that combines all elements of the two sets.
- The | symbol performs the same logic. Keep in mind that duplicates like "Snickers" will only appear once.

```
In [146] candy_bars.union(sweet_things)
```

```
Out [146] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Snickers', 'Sour Patch Kids'}
```

```
In [147] candy_bars | sweet_things
```

```
Out [147] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Snickers', 'Sour Patch Kids'}
```



# Set Operations

---

- The difference method returns a set of elements that are present in the set the method is called on but not present in the set passed in as an argument.
- The - symbol can be used as an alternative syntax.

```
In [148] candy_bars.difference(sweet_things)
```

```
Out [148] {'100 Grand', 'Milky Way'}
```

```
In [149] candy_bars - sweet_things
```

```
Out [149] {'100 Grand', 'Milky Way'}
```



# Set Operations



- The `symmetric_difference` method returns a set with elements found in either of the sets but not both.
- The `^` syntax accomplishes the same result.

```
In [150] candy_bars.symmetric_difference(sweet_things)
Out [150] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Sour Patch Kids'}
In [151] candy_bars ^ sweet_things
Out [151] {'100 Grand', 'Milky Way', 'Reeses Pieces', 'Sour Patch Kids'}
```





# Modules, Classes, and Datetimes

---

The syntax for importing a module is identical for native modules and for external packages like pandas.

Write the import keyword followed by the module or package's name.

Let's import Python's datetime module for working with dates and times.

In [152] import datetime



# Modules, Classes, and Datetimes

---

- An alias is an alternate name for a module. It designates a shortcut so that we don't have to write out the complete module name each time we reference it.
- Aliases are assigned with the `as` keyword.
- The name is up to us, but certain aliases have established themselves as favorites among the Python community.

In [153] import datetime as dt



# Modules, Classes, and Datetimes



- Like everything else in Python, a module is an object. Its attributes represent the variables, functions and classes that are defined within that file.
- Attributes are accessed with dot syntax.
- For example, the MAXYEAR attribute tells us the maximum year a date in Python can support. It's a long way off from now!

In [154] `dt.MAXYEAR`

Out [154] 9999



# Modules, Classes, and Datetimes

---

- The syntax for creating an instance from a class is similar to that of invoking a function.
- Write the class name followed by a pair of parentheses populated with arguments.
- The first three arguments for the datetime constructor are mandatory and represent the year, month, and day the object will represent.
- The example below creates a datetime object for Leonardo Da Vinci's birthday on April 15th, 1452.

```
In [155] da_vinci_birthday = dt.datetime(1452, 4, 15)
```



# Modules, Classes, and Datetimes



Like all objects, datetime objects have their own set of attributes and methods.

```
In [156] da_vinci_birthday.year
```

```
Out [156] 1452
```

One example of a method on a datetime object is `weekday`, which returns the day of the week represented as an integer. 0 represents Sunday and 6 represents Saturday. It looks like Da Vinci was born on a Wednesday!

```
In [157] da_vinci_birthday.weekday()
```

```
Out [157] 3
```



# Summary



- Python's simple data types include integers, floating-point numbers, strings, and Booleans.
  - Operators are used for mathematical operations like addition and subtraction as well as comparing the equality or inequality of two objects.
  - A variable is a name assigned to an object.
  - The value of a variable can change over the course of a program's execution.
- 

# Lesson 4 NumPy crash course





# NumPy crash course



This session covers:

- One-dimensional and multi-dimensional data structures in NumPy
  - Generating a range of sequential numeric values
  - Common attributes on NumPy's core ndarray object
  - Generating random 1D, 2D, and 3D arrays of integers and floating-point values
  - The NaN object for representing absence
- 

# Dimensions

	Temperature
New York	38
Chicago	36
San Francisco	51
Miami	73

# Dimensions

	Monday	Tuesday	Wednesday	Thursday	Friday
New York	38	41	35	32	35
Chicago	36	39	31	27	25
San Francisco	51	52	50	49	53
Miami	73	74	72	71	74



# Dimensions



## Week 1

	Monday	Tuesday	Wednesday	Thursday	Friday
New York	38	41	35	32	35
Chicago	36	39	31	27	25
San Francisco	51	52	50	49	53
Miami	73	74	72	71	74

# Dimensions

## Week 2

	Monday	Tuesday	Wednesday	Thursday	Friday
New York	40	42	38	36	28
Chicago	32	28	25	31	25
San Francisco	49	55	54	51	48
Miami	75	78	73	76	71



# The ndarray Object



- Let's begin by creating a new Jupyter Notebook and importing the numpy library.
- It's often assigned the alias np.

In [1] import numpy as np

# Generating a Numeric Range with the `arrange` Method

- When `arange` is invoked with one argument, NumPy will set 0 as the lower bound, the point at which the range begins.
- The lower bound is inclusive; its value will be included.

```
In [2] np.arange(3)  
Out [2] array([0, 1, 2])
```

# Generating a Numeric Range with the `arrange` Method

- We can also pass arange two arguments to mark the lower and upper bounds of the range.
- The starting point will remain inclusive while the endpoint will remain exclusive.
- In the next example, notice how 2 is included but 6 is not.

```
In [3] np.arange(2, 6)
Out [3] array([2, 3, 4, 5])
```

# Generating a Numeric Range with the `arrange` Method

- The first two arguments correspond to start and stop keyword parameters whose names can be written out explicitly.
- The code samples above and below are identical.

```
In [4] np.arange(start = 2, stop = 6)  
Out [4] array([2, 3, 4, 5])
```

# Generating a Numeric Range with the `arrange` Method

- To reverse the values, provide a `[::-1]` after the `ndarray` object.
- This syntax is inspired by Python's list-slicing syntax. It iterates across the range from the last value to the first value
- in steps of 1. Below, we use it to create a list of descending numbers.

```
In [5] np.arange(start = 2, stop = 6)[::-1]  
Out [5] array([5, 4, 3, 2])
```

# Generating a Numeric Range with the `arrange` Method

The `arange` function accepts an optional third parameter, `step`, which represents the interval or stride between every two values. It's easiest to think about it mathematically. Start at the lower bound and add the `step` value until you reach the upper bound. The example below moves from 0 to 111 (exclusive) in gaps of 10.

```
In [6] np.arange(start = 0, stop = 111, step = 10)  
Out [6] array([ 0,  10,  20,  30,  40,  50,  60,  70,  80,  90, 100, 110])
```

Let's save that last `ndarray` to a `tens` variable. We'll use it shortly.

```
In [7] tens = np.arange(start = 0, stop = 111, step = 10)
```



# Attributes on a ndarray Object



- The shape attribute returns a tuple with the dimensions of the array.
- The length of the tuple is equal to the object's number of dimensions.
- The output below indicates that tens is a onedimensional array that holds 12 values.

In [8] tens.shape

Out [8] (12,)



# Attributes on a ndarray Object



- The number of dimensions is also accessible via the ndim attribute.

```
In [9] tens.ndim  
Out [9] 1
```

- The number of elements is accessible via the size attribute.

```
In [10] tens.size  
Out [10] 12
```



# The reshape Method



- Currently, our 12-element range is one-dimensional.
- We can think of it as a single row of 12 values.

```
In [11] tens  
Out [11] array([  0,   10,   20,   30,   40,   50,   60,   70,   80,   90,  100,  
                 110])
```

# The reshape Method

```
In [12] tens.reshape(4, 3)
```

```
Out [12] array([[ 0, 10, 20],  
[ 30, 40, 50],  
[ 60, 70, 80],  
[ 90, 100, 110]])
```

```
In [13] tens.reshape(4, 3).ndim
```

```
Out [13] 2
```



# The reshape Method

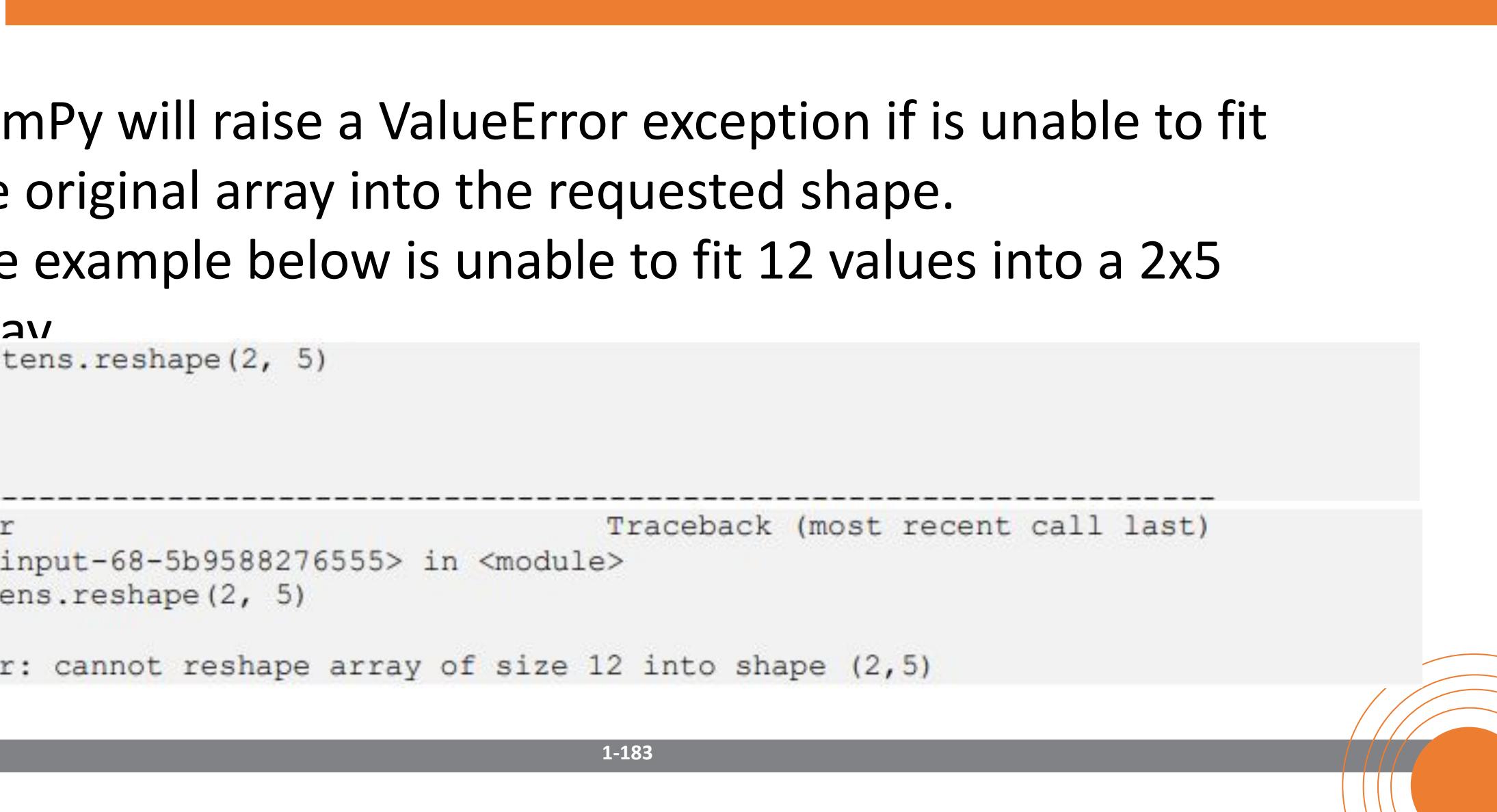


- The product of the arguments must equal the number of elements within the array.  $4 \times 3$  works because it is equal to 12.
- Another valid example is a two-dimensional array with 2 rows and 6 columns.

```
In [14] tens.reshape(2, 6)
Out [14] array([[ 5, 15, 25, 35, 45, 55],
 [ 65, 75, 85, 95, 105, 115]])
```



# The reshape Method



- NumPy will raise a ValueError exception if is unable to fit the original array into the requested shape.
- The example below is unable to fit 12 values into a 2x5

`array`

```
In [15] tens.reshape(2, 5)
```

```
Out [15]
```

```
-----  
ValueError                                     Traceback (most recent call last)  
<ipython-input-68-5b9588276555> in <module>  
----> 1 tens.reshape(2, 5)  
  
ValueError: cannot reshape array of size 12 into shape (2, 5)
```



# The reshape Method



```
In [16] tens.reshape(2, 3, 2)
```

```
Out [16] array([[[ 5, 15],  
                  [ 25, 35],  
                  [ 45, 55]],  
  
                  [[ 65, 75],  
                  [ 85, 95],  
                  [105, 115]]])
```

```
In [17] tens.reshape(2, 3, 2).ndim
```

```
Out [17] 3
```



# The randint Function



- The randint function is ideal for generating one or more random numbers.
- With a single argument, it returns a random integer from 0 up to the value.
- The example below will return a value between 0 and 4.

```
In [18] np.random.randint(5)
```

```
Out [18] 3
```



# The randint Function



- Two arguments create an inclusive lower bound and an exclusive upper bound from which NumPy will select a number.

```
In [19] np.random.randint(1, 10)  
Out [19] 9
```



# The randint Function

---

- What if we want to generate an entire array of random integers? The third argument specifies the desired shape of the array.
- We can pass it a single integer or a one-element list to create a one-dimensional array.

```
In [20] np.random.randint(1, 10, 3)
```

```
Out [20] array([4, 6, 3])
```

```
In [21] np.random.randint(1, 10, [3])
```

```
Out [21] array([9, 1, 6])
```



# The randint Function



- To create a multi-dimensional ndarray, pass a list with two or more values.
- The example below populates a two-dimensional 3x5 array with values between 1 and 10 (exclusive).

```
In [22] np.random.randint(1, 10, [3, 5])
```

```
Out [22] array([[2, 9, 8, 8, 7],  
                 [9, 8, 7, 3, 2],  
                 [4, 4, 5, 3, 9]])
```

# The randn Function

```
In [23] np.random.randn(3)

Out [23] array([-1.04474993,  0.46965268, -0.74204863])

In [24] np.random.randn(2, 4)

Out [24] array([[[-0.35139565,  1.15677736,  1.90854535,  0.66070779],
   [-0.02940895, -0.86612595,  1.41188378, -1.20965709]]]

In [25] np.random.randn(2, 4, 3)

Out [25] array([[[ 0.38281118,  0.54459183,  1.49719148],
   [-0.03987083,  0.42543538,  0.11534431],
   [-1.38462105,  1.54316814,  1.26342648],
   [ 0.6256691 ,  0.51487132,  0.40268548]],

   [[-0.24774185, -0.64730832,  1.65089833],
   [ 0.30635744,  0.21157744, -0.5644958 ],
   [ 0.35393732,  1.80357335,  0.63604068],
   [-1.5123853 ,  1.20420021,  0.22183476]]])
```



# The nan Object



- The NumPy library's `nan` object is used to represent a missing or invalid value.
- We'll see it appear later in the course when we import a dataset with blank cells.
- For now, we can access it directly as an attribute on the `np` module.

In [26] `np.nan`

Out [26] `nan`



# The nan Object



- NaN is short for "not a number", a generic term for missing data.
- Note that two NaN values are considered unequal.

```
In [27] np.nan == np.nan
```

```
Out [27] False
```



# Summary



- Dimensions refer to the number of points of reference that are needed to extract a single value from a data structure.
- The NumPy library's ndarray object is an n-dimensional array that Pandas uses to store its data under the hood.

# Lesson 5 The Series Object





# The Series Object



This session covers:

- Instantiating Series objects from lists, dictionaries, and more
- Creating a custom index for a Series
- Accessing attributes and invoking methods on a Series
- Performing mathematical operations on a Series
- Passing the Series to Python's built-in functions



# Overview of a Series



- Let's create some Series objects, shall we? We'll begin with an import of pandas and numpy; the latter library will be used later to generate some random data.
- The popular community aliases for pandas and numpy are pd and np.

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



# Modules, Classes and Instances

---

- A module is a Python file that organizes a collection of related code.
- Like everything else in the language, it is an object that can hold attributes, or pieces of internal data.
- In our case, the pd module contains the top-level exports of the pandas library, a combination of 100+ classes, functions, exceptions, constants and more.
- Any of these attributes can be accessed with dot syntax like so:

`module.attribute`

# Modules, Classes and Instances

```
In [ ]: pd.S
          Series
          SparseArray
          SparseDataFrame
          SparseDtype
          SparseSeries
In [ ]:
In [ ]:
```



# Modules, Classes and Instances



- An object is instantiated by placing a pair of parentheses after its class name.

```
In [2] pd.Series()  
Out [2] Series([], dtype: float64)
```

# Populating the Series with Values

```
In [3] ice_cream_flavors = ["Chocolate", "Vanilla", "Strawberry",
                           "Rum Raisin"]

pd.Series(ice_cream_flavors)

Out [3] 0      Chocolate
        1      Vanilla
        2    Strawberry
        3   Rum Raisin
       dtype: object
```

# Populating the Series with Values

```
pd.Series()  
  
Init signature:  
pd.Series(  
    data=None,  
    index=None,  
    dtype=None,  
    name=None,  
    copy=False,  
    fastpath=False,  
)
```

: The parameters and the default arguments for the Series class



# Populating the Series with Values

- Parameters and arguments can also be tied together explicitly.
- Write the name of the parameter followed by an equal sign

```
In [4]: pd.Series(data = ice_cream_flavors)

Out [4]: 0      Chocolate
          1      Vanilla
          2    Strawberry
          3   Rum Raisin
dtype: object
```



# Customizing the Index

---

- Let's take a look at our Series object again.
  - The incrementing list of numbers on the left-side is called the index.
  - It serves the same purpose as an index position in a list: a numeric indicator of the element's place in line.
  - We can also view it as comparable to a key in a dictionary: an identifier or name connected to each value.
  - The Series combines the best features of both data structures.
- 



# Customizing the Index

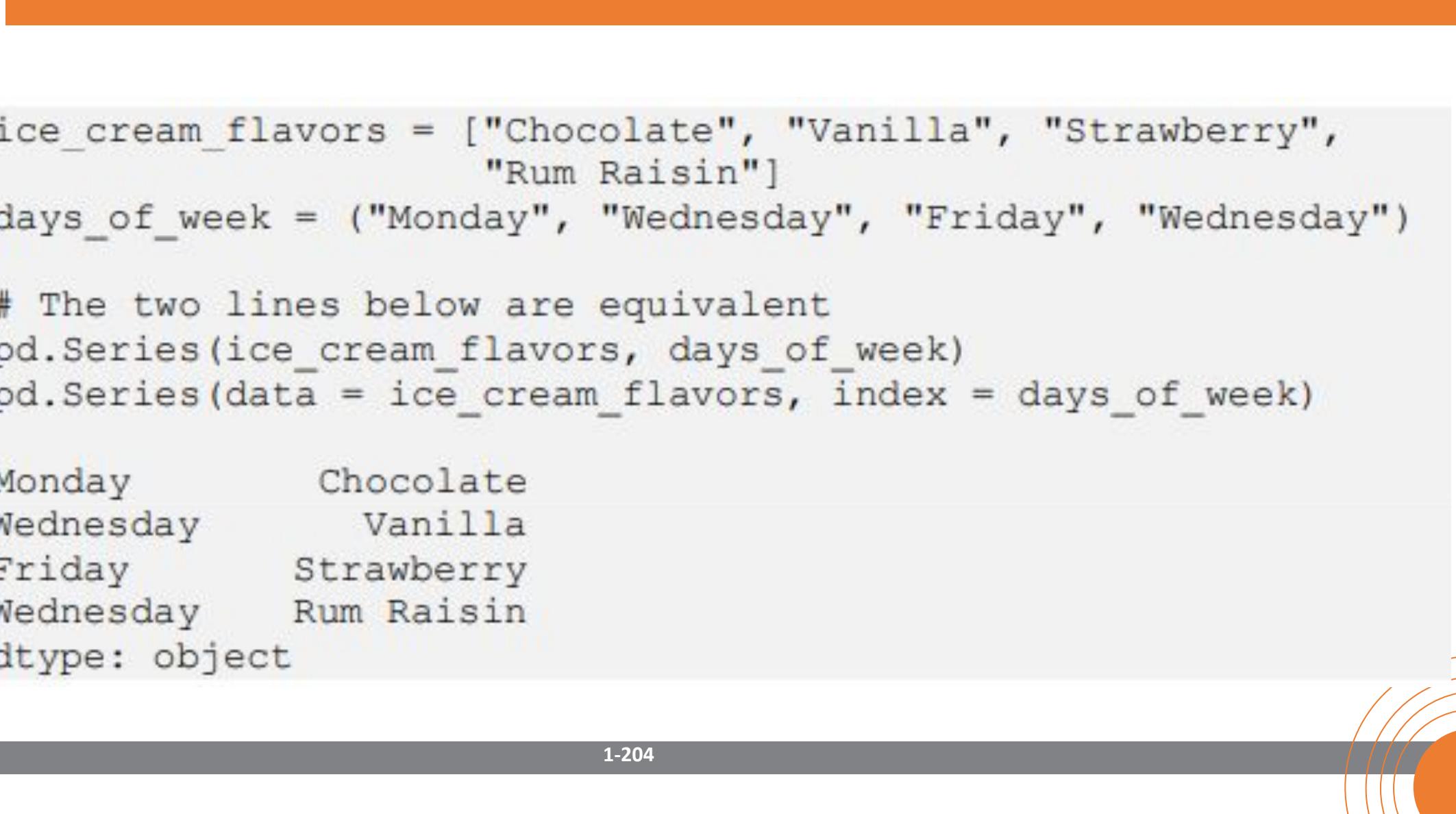
- In the example below, `ice_cream_flavors` is a list and `days_of_week` is a tuple, but both objects have a length of

```
In [5] ice_cream_flavors = ["Chocolate", "Vanilla", "Strawberry",
                           "Rum Raisin"]
      days_of_week = ("Monday", "Wednesday", "Friday", "Saturday")
      pd.Series(data = ice_cream_flavors, index = days_of_week)

Out [5] Monday           Chocolate
       Wednesday        Vanilla
       Friday            Strawberry
       Saturday          Rum Raisin
       dtype: object
```



# Customizing the Index



```
In [6] ice_cream_flavors = ["Chocolate", "Vanilla", "Strawberry",
                           "Rum Raisin"]
      days_of_week = ("Monday", "Wednesday", "Friday", "Wednesday")

      # The two lines below are equivalent
      pd.Series(ice_cream_flavors, days_of_week)
      pd.Series(data = ice_cream_flavors, index = days_of_week)

Out [6] Monday           Chocolate
       Wednesday        Vanilla
       Friday            Strawberry
       Wednesday        Rum Raisin
       dtype: object
```



# Customizing the Index



- As mentioned earlier, keyword arguments permit values to be passed in any order.
- Thus, the line of code below offers yet another way to return the same Series as the one above

```
pd.Series(index = days_of_week, data = ice_cream_flavors)
```

```
In [7] stock_prices = [985.32, 950.44]
      times = ["Open", "Close"]
      pd.Series(data = stock_prices, index = times)

Out [7] Open    985.32
        Close   950.44
        dtype: float64

In [8] bunch_of_bools = [True, False, False]
      pd.Series(bunch_of_bools)

Out [8] 0    True
        1   False
        2   False
        dtype: bool

In [9] lucky_numbers = [4, 8, 15, 16, 23, 42]
      pd.Series(lucky_numbers)

Out [9] 0     4
        1     8
        2    15
        3    16
        4    23
        5    42
        dtype: int64
```



# Creating a Series with Missing Values

---

- In the example below, we create a Series from a list that includes one "missing value".
- In the visual output, we can see NaN at index position 2.
- Get used to this trio of letters; we're going to be seeing them a lot throughout the text.

```
In [10] temperatures = [94, 88, np.nan, 91]
        pd.Series(data = temperatures)

Out [10] 0    94.0
         1    88.0
         2      NaN
         3    91.0
dtype: float64
```

# Creating a Series with Missing Values

- Python's None object will also be converted to a NaN when passed as a value into a Series

```
In [11] pd.Series(data = ["Hello", None])  
  
Out [11] 0      Hello  
          1      None  
         dtype: object
```



# Create a Series from Python Objects

---

## Dictionaries

- When passed a dictionary, pandas will use each key as the corresponding index label in the Series.

```
In [12] calorie_info = {  
    "Cereal": 125,  
    "Chocolate Bar": 406,  
    "Ice Cream Sundae": 342  
}  
  
diet = pd.Series(calorie_info)  
  
diet  
  
Out [12] Cereal          125  
Chocolate Bar        406  
Ice Cream Sundae     342  
dtype: int64
```



# Create a Series from Python Objects

---

- We can access these composed objects as attributes on our Series object.
- For example, the values attribute will return the ndarray object that stores the values.
- We can pass in the object to Python's built-in type function to see the class it is constructed from.

```
In [13] diet.values  
  
Out [13] array([125, 406, 342])  
  
In [14] type(diet.values)  
  
Out [14] numpy.ndarray
```



# Create a Series from Python Objects



- Similarly, the `index` attribute on a Series returns the Index object it stores internally

```
In [15] diet.index  
  
Out [15] Index(['Cereal', 'Chocolate Bar', 'Ice Cream Sundae'],  
              dtype='object')  
  
In [16] type(diet.index)  
  
Out [16] pandas.core.indexes.base.Index
```



# Create a Series from Python Objects



Some attributes can return helpful details about an object. For example, the `size` attribute can tell us the total number of values in the Series. Null values (`NaN`) will be counted here.

```
In [17] diet.size
```

```
Out [17] 3
```

A complementary attribute, `shape`, returns a tuple of the dimensions of any pandas data structure. For a 1-dimensional object like the Series, the tuple's only value will be its size. The comma after the 3 is a standard visual output for one-element tuples in Python.

```
In [18] diet.shape
```

```
Out [18] (3,)
```



# Create a Series from Python Objects



- The `is_unique` attribute returns True if there are no duplicate values in the Series

```
In [19] diet.unique
```

```
Out [19] True
```

```
In [20] pd.Series(data = [3, 3]).is_unique
```

```
Out [20] False
```



# Create a Series from Python Objects

---

- The `is_monotonic` attribute return True if all of the values are increasing.

```
In [21] pd.Series(data = [1, 3, 6]).is_monotonic
```

```
Out [21] True
```

```
In [22] pd.Series(data = [1, 6, 3]).is_monotonic
```

```
Out [22] False
```

# Tuples

- A Series can accept a tuple as its data source as well.
- As a reminder, a tuple is a similar data structure to a list except it is immutable.
- Elements cannot be added, removed or replaced in a tuple once it has been declared.

```
In [23] pd.Series(data = ("Red", "Green", "Blue"))
```

```
Out [23] 0      Red
          1      Green
          2      Blue
          dtype: object
```



# Tuples



- To use one or more tuples as actual Series values, wrap them in a larger container such a list.

```
In [24] pd.Series(data = [ ("Red", "Green", "Blue"), ("Orange", "Yellow") ])
```

```
Out [24] 0      (Red, Green, Blue)
          1      (Orange, Yellow)
         dtype: object
```



# Sets



```
In [25] my_set = { "Ricky", "Bobby" }  
pd.Series(my_set)
```

---

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-25-bf85415a7772> in <module>  
      1 my_set = { "Ricky", "Bobby" }  
----> 2 pd.Series(my_set)
```

```
TypeError: 'set' type is unordered
```



# Sets



- If the data in your program arrives as a set, one solution is to transform it into a suitable data structure like a list.

```
In [26] pd.Series(list(my_set))
```

```
Out [26] 0      Ricky
          1      Bobby
dtype: object
```



# NumPy Arrays

```
In [27]: data = np.random.randint(1, 101, 10)
        data

Out [27]: array([27, 16, 13, 83, 3, 38, 34, 19, 27, 66])

In [28]: pd.Series(data)

Out [28]: 0    27
          1    16
          2    13
          3    83
          4     3
          5    38
          6    34
          7    19
          8    27
          9    66
         dtype: int64
```

# NumPy Arrays

```
In [29]: pd.Series(np.random.randn(5, 10))
```

```
-----  
Exception                                                 Traceback (most recent call last)  
<ipython-input-16-917a11418a68> in <module>  
----> 1 pd.Series(np.random.randn(5, 10))
```

```
Exception: Data must be 1-dimensional
```



# Retrieving the First and Last Rows



- If an attribute can be considered a variable that belongs to an object, then a method can be described as a function that belongs to an object.
  - It is an action or command that we ask the object to perform.
  - Attributes define an object's state while methods define an object's behavior.
  - Methods will typically involve some kind of analysis, calculation or manipulation of the object's data.
- 

# Retrieving the First and Last Rows

```
In [30]: values = range(0, 500, 5)
         nums = pd.Series(data = values)
         nums
```

```
Out [30]: 0      0
          1      5
          2     10
          3     15
          4     20
          ...
          95    475
          96    480
          97    485
          98    490
          99    495
Length: 100, dtype: int64
```



# Retrieving the First and Last Rows

---

- Let's begin our exploration with one of the simplest methods available on a Series.
- The head method returns one or more rows from the beginning or the top of the dataset.
- It accepts a single argument n that represents the number of rows to extract.

```
In [31] nums.head(3)

Out [31] 0      0
          1      5
          2     10
dtype: int64
```



# Retrieving the First and Last Rows

---

- In this scenario, the n parameter has a default argument of 5.
- If a head method invocation does not explicitly pass an argument f In [32] nums.head ()

```
Out [32] 0      0
          1      5
          2     10
          3     15
          4     20
dtype: int64
```

# Retrieving the First and Last Rows

```
In [33]: nums.tail(3)
```

```
Out [33]:
```

97	485
98	490
99	495

```
dtype: int64
```

```
In [34]: nums.tail()
```

```
Out [34]:
```

95	475
96	480
97	485
98	490
99	495

```
dtype: int64
```



# Mathematical Operations

Let's begin by defining a Series from a list of ascending numbers. In the middle, we'll sneak-in a np.nan to represent a missing value. We'll assign the variable s, a common one-character variable for a Series.

```
In [35] s = pd.Series([1, 2, 3, np.nan, 4, 5])
```

The sum method returns the sum of all of the values. Missing values are excluded.

```
In [36] s.sum()
```

```
Out [36] 15.0
```

Most methods include a skipna parameter that can be set to False to include missing values. Because a null value cannot be added to any other, the return value will be another null value.

```
In [37] s.sum(skipna = False)
```

```
Out [37] nan
```



# Mathematical Operations



```
In [38] s.sum(min_count = 3)
```

```
Out [38] 15.0
```

```
In [39] s.sum(min_count = 5)
```

```
Out [39] 15.0
```

```
In [40] s.sum(min_count = 6)
```

```
Out [40] nan
```



# Mathematical Operations



- The product method multiplies the Series values together.  
Like sum, it accepts skipna and min\_count parameters.

In [41] s.product()

Out [41] 120.0

# Mathematical Operations

```
In [42] s.cumsum()
```

```
Out [42] 0      1.0
         1      3.0
         2      6.0
         3      NaN
         4     10.0
         5     15.0
dtype: float64
```



# Mathematical Operations



- If the skipna parameter is passed an argument of False here, the returned Series will have a cumulative sum up to the index with the first missing value, then nan for the remaining values.

```
In [43] s.cumsum(skipna = False)
```

```
Out [43] 0    1.0
          1    3.0
          2    6.0
          3    NaN
          4    NaN
          5    NaN
dtype: float64
```

# Mathematical Operations

- We may also want to see the shifts from one value to another by percentage.
- The pct\_change method is optimal here.

```
In [44] s.pct_change()  
  
Out [44] 0           NaN  
          1  1.000000  
          2  0.500000  
          3  0.000000  
          4  0.333333  
          5  0.250000  
  
dtype: float64
```



# Mathematical Operations



- The mean method returns the average of the Series' values.
- The average is calculated by diving the sum of values by the number of values (in this case, 15 divided by 5).

In [45] s.mean()  
Out [45] 3.0



# Mathematical Operations



- Half of the values will fall above it and half of the values will fall below it.

In [46] s.median()

Out [46] 3.0

- The std method returns the standard deviation of the values, a statistical measure of the amount of variation in the data.

In [47] s.std()

Out [47] 1.5811388300841898



# Mathematical Operations



- The max and min methods retrieve the largest and smallest value from the Series.

```
In [48] s.max()
```

```
Out [48] 5.0
```

```
In [49] s.min()
```

```
Out [49] 1.0
```



# Mathematical Operations



```
In [50] animals = pd.Series(["koala", "aardvark", "zebra"])
       animals.max()

Out [50] 'zebra'

In [51] animals.min()

Out [51] 'aardvark'
```



# Mathematical Operations



- The powerful describe method returns a Series of popular statistical evaluations including the count, mean, standard deviation and more.

```
In [52] s.describe()

Out [52] count      5.000000
          mean       3.000000
          std        1.581139
          min        1.000000
          25%       2.000000
          50%       3.000000
          75%       4.000000
          max        5.000000
          dtype: float64
```



# Mathematical Operations



- The selection of a random assortment means that the order of values in the new Series may not match the order of values in the original Series.

```
In [53] s.sample(3)

Out [53] 1    2
          3    4
          2    3
dtype: int64
```



# Mathematical Operations



- The method accepts a `n` parameter with a default argument of 1.
- A Series with one row of data will be returned when `sample` is invoked without an explicit argument.

```
In [54] s.sample()
```

```
Out [54] 0    1
          dtype: int64
```



# Mathematical Operations



```
In [55] authors = pd.Series(["Hemingway", "Orwell", "Dostoevsky",
                           "Fitzgerald", "Orwell"])
        authors.unique()

Out [55] array(['Hemingway', 'Orwell', 'Dostoevsky', 'Fitzgerald'], dtype=object)
```

The `nunique` method returns the number of unique values in the Series.

```
In [56] authors.nunique()

Out [56] 4
```

```
In [57]: s1 = pd.Series(data = [5, np.nan, 15], index = ["A", "B", "C"])
         s1

Out[57]: A    5.0
          B    NaN
          C   15.0
         dtype: float64

In [58]: s1 + 3

Out[58]: A    8.0
          B    NaN
          C   18.0
         dtype: float64

In [59]: s1 - 5

Out[59]: A    0.0
          B    NaN
          C   10.0
         dtype: float64

In [60]: s1 * 2

Out[60]: A   10.0
          B    NaN
          C   30.0
         dtype: float64

In [61]: s1 / 2

Out[61]: A    2.5
          B    NaN
          C    7.5
         dtype: float64
```



# Arithmetic Operations

---

- The floor division operator ( `//` ) removes any digits after the decimal point following a division.
- For example, the plain division of 15.0 by 4 yields a result of 3.75. In comparison, floor division chops off the .75, leading to a final result of 3.

```
In [62] s1 // 4  
  
Out [62] A      1.0  
        B      NaN  
        C      3.0  
dtype: float64
```



# Arithmetic Operations



- The modulo operator ( % ) returns the remainder of a division.
- In the example below, the value of 5 at index label A leaves a remainder of 2 when divided by 3.

In [63] s1 % 3

Out [63] A 2.0

B NaN

C 0.0

dtype: float64



# Broadcasting



- In the example below, the s1 and s2 Series have the same three-element index.
- The elements at index A (1 and 2), index B (2 and 5), and index C (3 and 6) are thus added together.

```
In [64] s1 = pd.Series([1, 2, 3], index = ["A", "B", "C"])
        s2 = pd.Series([4, 5, 6], index = ["A", "B", "C"])

        s1 + s2

Out [64] A    5
          B    7
          C    9
          dtype: int64
```



# Broadcasting



```
In [65]: s1 = pd.Series(data = [3, 6, np.nan, 12])
          s2 = pd.Series(data = [2, 6, np.nan, 12])
```

```
s1 == s2
```

```
Out [65]: 0      False
           1      True
           2      False
           3      True
          dtype: bool
```

```
In [66]: s1 != s2
```

```
Out [66]: 0      True
           1      False
           2      True
           3      False
          dtype: bool
```



# Broadcasting



```
In [67] s1 = pd.Series(data = [5, 10, 15], index = ["A", "B", "C"])
       s2 = pd.Series(data = [4, 8, 12, 14], index=["B", "C", "D", "E"])

       s1 + s2

Out [67] A      NaN
          B     14.0
          C    23.0
          D      NaN
          E      NaN
          dtype: float64
```

# Passing the Series to Python's Built-In Functions

- Let's create a small Series of strings representing cities in the United States.

```
In [68] cities = pd.Series(data = ["San Francisco", "Los Angeles", "Las Vegas"])
```

- The len function returns the number of rows in a Series. NaN values will be included in the count.

```
In [69] len(cities)
```

```
Out [69] 3
```

The type function returns the class than an object is constructed from.

```
In [70] type(cities)
```

```
Out [70] pandas.core.series.Series
```

# Passing the Series to Python's Built-In Functions

```
dir(cities)

['__',
 '_AXIS_ALIASES',
 '_AXIS_TALIASES',
 '_AXIS_LEN',
 '_AXIS_NAMES',
 '_AXIS_NUMBERS',
 '_AXIS_ORDERS',
 '_AXIS_REVERSED',
 '_AXIS_SLICENAP',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_prepare__',
 '__array_priority__',
 '__array_wrap__',
 '__bool__',
 '__bytes__',
 '__class__',
 '__contains__',
 '__delitem__',
 '__eq__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__gt__',
 '__hash__',
 '__iter__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setitem__',
 '__str__',
 '__subclasshook__']
```

# Passing the Series to Python's Built-In Functions

- The list function / class returns a Python list with the values of the Series.

```
In [71] list(cities)
```

```
Out [71] ['San Francisco', 'Los Angeles', 'Las Vegas']
```

- The dict function / class converts the Series into a dictionary. The index labels are used as the dictionary keys.

```
In [72] dict(cities)
```

```
Out [72] {0: 'San Francisco', 1: 'Los Angeles', 2: 'Las Vegas'}
```

# Passing the Series to Python's Built-In Functions

- The max function returns the greatest value from the Series.
- If the Series consists of strings, max returns the last alphabetically sorted value.

```
In [73] max(cities)
```

```
Out [73] 'San Francisco'
```

- The min function returns the smallest value from the Series.
- If the Series consists of strings, min returns the first value in alphabetical order.

```
In [74] min(cities)
```

# Passing the Series to Python's Built-In Functions

```
In [75] "Las Vegas" in cities  
Out [75] False  
  
In [76] 2 in cities  
Out [76] True  
  
In [77] "Las Vegas" in cities.values  
Out [77] True
```

# Passing the Series to Python's Built-In Functions

- The `not in` keyword validates that an element is not contained in a Series.

```
In [78] 100 not in cities
```

```
Out [78] True
```

```
In [79] "Paris" not in cities.values
```

```
Out [79] True
```



# Coding Challenges / Exercises

---

- Declare a colors list with 4 strings representing your favorite colors. Instantiate a
- Series object with the values from the colors list.

```
In [80] colors = ["Red", "Blue", "White", "Navy"]  
pd.Series(colors)
```

```
Out [80] 0 Red  
1 Blue  
2 White  
3 Navy  
dtype: object
```

# Coding Challenges / Exercises

```
In [81] pd.Series(data = [1, 2, 3, 4], index = colors)

Out [81] Red      1
          Blue     2
          White    3
          Navy     4
          dtype: int64
```



# Summary



- A Series is a one-dimensional labelled array that consists of values and an index.
- A Series can be constructed from lists, dictionaries, tuples and NumPy arrays.
- The head and tail methods retrieve the first and last rows of the Series.

# Lesson 6 Series Methods





# Series Methods



This lesson covers:

- Importing CSV datasets
- Sorting Series values in ascending and descending order
- Retrieving the largest and smallest values in a Series
- Mutating a Series inplace
- Counting occurrences of unique values in a Series
- Applying an operation to every value in a Series



# Importing a Dataset with the `read_csv` Method

- As always, our first step is to launch a fresh Jupyter Notebook and import pandas.
- Make sure to create the Notebook in the same directory as the CSV files you downloaded for this course.

In [1] import pandas as pd

```
In [2] pd.read_csv(filepath_or_buffer = "pokemon.csv")
      pd.read_csv("pokemon.csv")
```

```
Out [2]
```

	Pokemon	Type
0	Bulbasaur	Grass / Poison
1	Ivysaur	Grass / Poison
2	Venusaur	Grass / Poison
3	Charmander	Fire
4	Charmeleon	Fire
...	...	...
804	Stakataka	Rock / Steel
805	Blacephalon	Fire / Ghost
806	Zeraora	Electric
807	Meltan	Steel
808	Melmetal	Steel
809	rows × 2 columns	

# Importing a Dataset with the `read_csv` Method

```
In [3] pd.read_csv("pokemon.csv", index_col = "Pokemon")
```

```
Out [3]
```

Pokemon	Type
Bulbasaur	Grass / Poison
Ivysaur	Grass / Poison
Venusaur	Grass / Poison
Charmander	Fire
Charmeleon	Fire

# Importing a Dataset with the `read_csv` Method

```
In [4] pd.read_csv("pokemon.csv", index_col = "Pokemon", squeeze = True)
```

```
Out [4] Pokemon
       Bulbasaur      Grass / Poison
       Ivysaur        Grass / Poison
       Venusaur       Grass / Poison
       Charmander     Fire
       Charmeleon     Fire
       ...
       Stakataka      Rock / Steel
       Blacephalon    Fire / Ghost
       Zeraora        Electric
       Meltan          Steel
       Melmetal        Steel
       Name: Type, Length: 809, dtype: object
```

# Importing a Dataset with the `read_csv` Method

The last step is to assign the Series to a variable so it can be reused throughout the Notebook.

```
In [5] pokemon = pd.read_csv("pokemon.csv", index_col = "Pokemon", squeeze = True)
```

Curious if there are any NaN values in the Series? The `hasnans` attribute informs us there are no missing values in either the values or the index.

```
In [6] pokemon.hasnans
```

```
Out [6] False
```

```
In [7] pokemon.index.hasnans
```

```
Out [7] False
```

# Importing a Dataset with the `read_csv` Method

```
In [8] pd.read_csv("google_stocks.csv").head()
```

```
Out [8]
```

	Date	Close
0	2004-08-19	49.98
1	2004-08-20	53.95
2	2004-08-23	54.50
3	2004-08-24	52.24
4	2004-08-25	52.80

# Importing a Dataset with the `read_csv` Method

```
In [9] google = pd.read_csv("google_stocks.csv", index_col = "Date",
                           parse_dates = ["Date"], squeeze = True)

                           google.head()
```

```
Out [9] Date
        2004-08-19    49.98
        2004-08-20    53.95
        2004-08-23    54.50
        2004-08-24    52.24
        2004-08-25    52.80
Name: Close, dtype: float64
```

# Importing a Dataset with the `read_csv` Method

```
In [10] pd.read_csv("revolutionary_war.csv").tail()
```

```
Out [10]
```

	Battle	Start Date	State
227	Siege of Fort Henry	9/11/1782	Virginia
228	Grand Assault on Gibraltar	9/13/1782	NaN
229	Action of 18 October 1782	10/18/1782	NaN
230	Action of 6 December 1782	12/6/1782	NaN
231	Action of 22 January 1783	1/22/1783	Virginia

# Importing a Dataset with the `read_csv` Method

```
In [11] pd.read_csv("revolutionary_war.csv",
                     index_col = "Start Date",
                     parse_dates = ["Start Date"]).tail()
```

Out [11]

Start Date	Battle	State
1782-09-11	Siege of Fort Henry	Virginia
1782-09-13	Grand Assault on Gibraltar	NaN
1782-10-18	Action of 18 October 1782	NaN
1782-12-06	Action of 6 December 1782	NaN
1783-01-22	Action of 22 January 1783	Virginia

# Importing a Dataset with the `read_csv` Method

```
In [12] pd.read_csv("revolutionary_war.csv",
                     index_col = "Start Date",
                     parse_dates = ["Start Date"],
                     usecols = ["State", "Start Date"],
                     squeeze = True).tail()
```

```
Out [12] Start Date
          1782-09-11    Virginia
          1782-09-13        NaN
          1782-10-18        NaN
          1782-12-06        NaN
          1783-01-22    Virginia
          Name: State, dtype: object
```

# Importing a Dataset with the `read_csv` Method

```
In [13] pokemon = pd.read_csv("pokemon.csv",
                               index_col = "Pokemon",
                               squeeze = True)

google = pd.read_csv("google_stocks.csv",
                     index_col = "Date",
                     parse_dates = ["Date"],
                     squeeze = True)

battles = pd.read_csv("revolutionary_war.csv",
                      index_col = "Start Date",
                      parse_dates = ["Start Date"],
                      usecols = ["State", "Start Date"],
                      squeeze = True)
```

# Sorting a Series

## Sorting by Values with the `sort_values` Method

```
In [14] google.sort_values()  
  
Out [14] Date  
2004-09-03      49.82  
2004-09-01      49.94  
2004-08-19      49.98  
2004-09-02      50.57  
2004-09-07      50.60  
...  
2019-04-23     1264.55  
2019-10-25     1265.13  
2018-07-26     1268.33  
2019-04-26     1272.18  
2019-04-29     1287.58  
Name: Close, Length: 3824, dtype: float64
```

# Sorting a Series

```
In [15] pokemon.sort_values()

Out [15] Pokemon
        Illumise           Bug
        Silcoon            Bug
        Pinsir             Bug
        Burmy              Bug
        Wurmple            Bug
                    ...
        Tirtouga          Water / Rock
        Relicanth         Water / Rock
        Corsola            Water / Rock
        Carracosta         Water / Rock
        Empoleon           Water / Steel
Name: Type, Length: 809, dtype: object
```



# Sorting a Series



- In pandas, as in Python, lowercase characters are sorted after uppercase characters.
- In the example below, the string "adam" is placed after the string "Ben".

```
In [16]: pd.Series(data = ["Adam", "adam", "Ben"]).sort_values()  
  
Out [16]: 0      Adam  
           .  
           2      Ben  
           1      adam  
          dtype: object
```

```
In [17] google.sort_values(ascending = False).head()
```

```
Out [17] Date
```

2019-04-29	1287.58
2019-04-26	1272.18
2018-07-26	1268.33
2019-10-25	1265.13
2019-04-23	1264.55

Name: Close, dtype: float64

```
In [18] pokemon.sort_values(ascending = False).head()
```

```
Out [18] Pokemon
```

Empoleon	Water / Steel
Carracosta	Water / Rock
Corsola	Water / Rock
Relicanth	Water / Rock
Tirtouga	Water / Rock

Name: Type, dtype: object

# Sorting a Series

```
In [19] battles.sort_values()  
  
Out [19] Start Date  
        1781-09-06    Connecticut  
        1779-07-05    Connecticut  
        1777-04-27    Connecticut  
        1777-09-03    Delaware  
        1777-05-17    Florida  
                    ...  
        1782-08-08      NaN  
        1782-08-25      NaN  
        1782-09-13      NaN  
        1782-10-18      NaN  
        1782-12-06      NaN  
Name: State, Length: 232, dtype: object
```



# Sorting a Series



```
In [20] battles.sort_values(na_position = "first")  
  
Out [20] Start Date  
1775-09-17      NaN  
1775-12-31      NaN  
1776-03-03      NaN  
1776-03-25      NaN  
1776-05-18      NaN  
...  
1781-07-06      Virginia  
1781-07-01      Virginia  
1781-06-26      Virginia  
1781-04-25      Virginia  
1783-01-22      Virginia  
Name: State, Length: 232, dtype: object
```

# Sorting a Series

```
In [21] battles.dropna().sort_values()
```

```
Out [21] Start Date
```

```
1781-09-06    Connecticut
```

```
1779-07-05    Connecticut
```

```
1777-04-27    Connecticut
```

```
1777-09-03    Delaware
```

```
1777-05-17    Florida
```

```
...
```

```
1782-08-19    Virginia
```

```
1781-03-16    Virginia
```

```
1781-04-25    Virginia
```

```
1778-09-07    Virginia
```

```
1783-01-22    Virginia
```

```
Name: State, Length: 162, dtype: object
```

# Sorting by Index with the `sort_index` Method

```
In [22]: pokemon.sort_index()  
  
Out [22]: Pokemon  
Abomasnow           Grass / Ice  
Abra                Psychic  
Absol               Dark  
Accelgor             Bug  
Aegislash           Steel / Ghost  
...  
Zoroark              Dark  
Zorua                Dark  
Zubat                Poison / Flying  
Zweilous             Dark / Dragon  
Zygarde              Dragon / Ground  
Name: Type, Length: 809, dtype: object
```

# Sorting by Index with the `sort_index` Method

```
In [23] battles.sort_index(ascending = False, na_position = "first")
```

```
Out [23] Start Date
        NaT      New Jersey
        NaT      Virginia
        NaT      NaN
        NaT      NaN
        1783-01-22      Virginia
                      ...
        1775-04-20      Virginia
        1775-04-19      Massachusetts
        1775-04-19      Massachusetts
        1774-12-14      New Hampshire
        1774-09-01      Massachusetts
Name: State, Length: 232, dtype: object
```

# Retrieving the Smallest and Largest Values with the `nsmallest` and `nlargest` Methods

```
In [24] google.nlargest(n = 5) # is the same as  
       google.nlargest()  
  
Out [24] Date  
        2019-04-29    1287.58  
        2019-04-26    1272.18  
        2018-07-26    1268.33  
        2019-10-25    1265.13  
        2019-04-23    1264.55  
Name: Close, dtype: float64
```

# Retrieving the Smallest and Largest Values with the nsmallest and nlargest Methods

```
In [25] google.nsmallest(n = 6) # is the same as  
       google.nsmallest(6)
```

```
Out [25] Date  
        2004-09-03    49.82  
        2004-09-01    49.94  
        2004-08-19    49.98  
        2004-09-02    50.57  
        2004-09-07    50.60  
        2004-08-30    50.81  
Name: Close, dtype: float64
```

# Overwriting a Series with the `inplace` Parameter

```
In [26]: battles.head(3)
```

```
Out [26]: Start Date
           1774-09-01    Massachusetts
           1774-12-14    New Hampshire
           1775-04-19    Massachusetts
           Name: State, dtype: object
```

```
In [27]: battles.sort_values().head(3)
```

```
Out [27]: Start Date
           1781-09-06    Connecticut
           1779-07-05    Connecticut
           1777-04-27    Connecticut
           Name: State, dtype: object
```

```
In [28]: battles.head(3)
```

```
Out [28]: Start Date
           1774-09-01    Massachusetts
           1774-12-14    New Hampshire
           1775-04-19    Massachusetts
           Name: State, dtype: object
```

# Overwriting a Series with the `inplace` Parameter

```
In [29] battles.head(3)

Out [29] Start Date
        1774-09-01    Massachusetts
        1774-12-14    New Hampshire
        1775-04-19    Massachusetts
        Name: State, dtype: object

In [30] battles.sort_values(inplace = True)

In [31] battles.head(3)

Out [31] Start Date
        1781-09-06    Connecticut
        1779-07-05    Connecticut
        1777-04-27    Connecticut
        Name: State, dtype: object
```

# Counting Values with the value\_counts Method

```
In [32] pokemon.value_counts()
```

```
Out [32] Normal          65
         Water           61
         Grass           38
         Psychic         35
         Fire            30
         ..
         Fire / Dragon   1
         Dark / Ghost   1
         Steel / Ground 1
         Fire / Psychic  1
         Dragon / Ice    1
Name: Type, Length: 159, dtype: int64
```

# Counting Values with the value\_counts Method

- The length of the value\_counts Series will be equal to the number of unique values from the pokemon Series.
- As a reminder, the nunique method returns this piece of information.

```
In [33] len(pokemon.value_counts())
```

```
Out [33] 159
```

```
In [34] pokemon.nunique()
```

```
Out [34] 159
```

# Counting Values with the value\_counts Method

```
In [35] pokemon.value_counts(ascending = True)
```

```
Out [35] Rock / Poison          1
          Ghost / Dark           1
          Ghost / Dragon          1
          Fighting / Steel        1
          Rock / Fighting          1
          ..
          Fire                      30
          Psychic                   35
          Grass                     38
          Water                     61
          Normal                    65
```

# Counting Values with the `value_counts` Method

- The `normalize` parameter can be set to `True` to return the frequencies of each unique value.
- The frequency represents what portion of the dataset a given value makes up.

```
In [36] pokemon.value_counts(normalize = True)
```

```
Out [36] Normal          0.080346
          Water           0.075402
          Grass            0.046972
          Psychic          0.043263
          Fire             0.037083
```

# Counting Values with the value\_counts Method

```
In [37] pokemon.value_counts(normalize = True) * 100
```

```
Out [37] Normal          8.034611
         Water           7.540173
         Grass            4.697157
         Psychic          4.326329
         Fire             3.708282
```

- Normal Pokémons make up 8.03% of the dataset, Water make up 7.54%, and so on.

# Counting Values with the value\_counts Method

```
In [38] (pokemon.value_counts(normalize = True) * 100).round(2)
```

```
Out [38] Normal          8.03
         Water           7.54
         Grass           4.70
         Psychic         4.33
         Fire            3.71
         ...
         Rock / Fighting  0.12
         Fighting / Steel 0.12
         Ghost / Dragon   0.12
         Ghost / Dark     0.12
         Rock / Poison    0.12
         Name: Type, Length: 159, dtype: float64
```

# Counting Values with the `value_counts` Method

- A Series with numeric values will work similarly.
- In the example below, we can see that no stock price appears more than three times in our google dataset.

```
In [39] google.value_counts().head()
```

```
Out [39] 237.04      3
          288.92      3
          287.68      3
          290.41      3
          194.27      3
```

# Counting Values with the `value_counts` Method

- Let's begin by determining the smallest and largest values within the Series with the `max` and `min` methods.
- An alternative solution is to pass the Series into Python's built-in `max` and `min` functions.

```
In [40] google.max()
```

```
Out [40] 1287.58
```

```
In [41] google.min()
```

```
Out [41] 49.82
```

# Counting Values with the `value_counts` Method

```
In [42] bins = [0, 200, 400, 600, 800, 1000, 1200, 1400]
      google.value_counts(bins = bins)

Out [42] (200.0, 400.0)      1568
        (-0.001, 200.0]       595
        (400.0, 600.0]       575
        (1000.0, 1200.0]     406
        (600.0, 800.0]       380
        (800.0, 1000.0]      207
        (1200.0, 1400.0]     93
        Name: Close, dtype: int64
```

# Counting Values with the value\_counts Method

```
In [43] google.value_counts(bins = bins).sort_index()

Out [43] (-0.001, 200.0)      595
          (200.0, 400.0)      1568
          (400.0, 600.0)      575
          (600.0, 800.0)      380
          (800.0, 1000.0)     207
          (1000.0, 1200.0)    406
          (1200.0, 1400.0)    93
Name: Close, dtype: int64
```

# Counting Values with the value\_counts Method

- An alternative solution is to pass a value of False to the sort parameter of the value\_counts method.
- This will yield the same result.

```
In [44] google.value_counts(bins = bins, sort = False)
```

```
Out [44] (-0.001, 200.0)      595
          (200.0, 400.0)      1568
          (400.0, 600.0)      575
          (600.0, 800.0)      380
          (800.0, 1000.0)     207
          (1000.0, 1200.0)     406
          (1200.0, 1400.0)     93
Name: Close, dtype: int64
```

# Counting Values with the value\_counts Method

```
In [45] google.value_counts(bins = 6, sort = False)
```

```
Out [45] (48.581, 256.113]           1204
          (256.113, 462.407]           1104
          (462.407, 668.7]            507
          (668.7, 874.993]            380
          (874.993, 1081.287]          292
          (1081.287, 1287.58]          337
          Name: Close, dtype: int64
```

# Counting Values with the value\_counts Method

- What about our battles dataset? We can use the `value_counts` method to see which states had the most battles in the Revolutionary War.

```
In [46] battles.value_counts().head()
```

```
Out [46] South Carolina      31
          New York            28
          New Jersey           24
          Virginia             21
          Massachusetts        11
          Name: State, dtype: int64
```

# Counting Values with the value\_counts Method

- NaN values will be missing from the list by default.
- Pass the dropna parameter an argument of False to count null values as a distinct category.

```
In [47] battles.value_counts(dropna = False).head()
```

```
Out [47]  NaN                  70  
          South Carolina        31  
          New York              28  
          New Jersey             24  
          Virginia              21  
          Name: State, dtype: int64
```

```
In [48] battles.index  
  
Out [48]  
  
DatetimeIndex(['1774-09-01', '1774-12-14', '1775-04-19', '1775-04-19',  
               '1775-04-20', '1775-05-10', '1775-05-27', '1775-06-11',  
               '1775-06-17', '1775-08-08',  
               ...  
               '1782-08-08', '1782-08-15', '1782-08-19', '1782-08-26',  
               '1782-08-25', '1782-09-11', '1782-09-13', '1782-10-18',  
               '1782-12-06', '1783-01-22'],  
              dtype='datetime64[ns]', name='Start Date', length=232,  
              freq=None)  
  
In [49] battles.index.value_counts()  
  
Out [49] 1775-04-19      2  
          1781-05-22      2  
          1781-04-15      2  
          1782-01-11      2  
          1780-05-25      2  
          ..  
          1778-05-20      1  
          1776-06-28      1  
          1777-09-19      1  
          1778-08-29      1  
          1777-05-17      1  
          Name: Start Date, Length: 217, dtype: int64
```



# Invoking a Function on Every Series Value with the apply Method



- A function can be treated like any other object in Python.
- This is a tricky concept for some because a function is a more abstract data type than a concrete value like an integer.
- But anything that you can do with an object like a number, you can also do with a function.



# Invoking a Function on Every Series Value with the apply Method

---

- The example below declares a `funcs` list that stores 3 of Python's built-in functions.
- Notice that the functions are not invoked within the list.
- These are references to the functions themselves.
- In analogous terms, we have a cookbook of 3 recipes here, but we haven't started baking anything yet.

In [50] `funcs = [len, max, min]`

# Invoking a Function on Every Series Value with the apply Method

- For each iteration, we invoke the current function being referenced by func and pass in the google Series.
- The output thus includes the length of the Series followed by its maximum and minimum values.

```
In [51] funcs = [len, max, min]
        for func in funcs:
            print(func(google))

Out [51] 3824
        1287.58
        49.82
```

# Invoking a Function on Every Series Value with the apply Method

```
In [52]: round(99.2)  
Out [52]: 99  
  
In [53]: round(99.49)  
Out [53]: 99  
  
In [54]: round(99.5)  
Out [54]: 100
```

# Invoking a Function on Every Series Value with the apply Method

```
In [55] google.apply(func = round)
        google.apply(round)

Out [55] Date
        2004-08-19      50
        2004-08-20      54
        2004-08-23      54
        2004-08-24      52
        2004-08-25      53
                    ...
        2019-10-21    1246
        2019-10-22    1243
        2019-10-23    1259
        2019-10-24    1261
        2019-10-25    1265
Name: Close, Length: 3824, dtype: int64
```

# Invoking a Function on Every Series Value with the apply Method

- We can use the in operator to check for the presence of the substring "/" in the input string. If it is found, we'll return the string "Multi". Otherwise, we'll return the string "Single".

```
In [56] def single_or_multi(types):  
        if "/" in types:  
            return "Multi"  
  
        return "Single"
```

# Invoking a Function on Every Series Value with the apply Method

- Let's get ready to pass the single\_or\_multi function to the apply method. Here's a quick refresher of our pokemon dataset.

```
In [57] pokemon.head(4)
```

```
Out [57] Pokemon
         Bulbasaur      Grass / Poison
         Ivysaur        Grass / Poison
         Venusaur       Grass / Poison
         Charmander     Fire
         Name: Type, dtype: object
```

# Invoking a Function on Every Series Value with the apply Method

```
In [58] pokemon.apply(single_or_multi)

Out [58] Pokemon
        Bulbasaur      Multi
        Ivysaur       Multi
        Venusaur      Multi
        Charmander     Single
        Charmeleon    Single
                      ...
        Stakataka     Multi
        Blacephalon   Multi
        Zeraora       Single
        Meltan        Single
        Melmetal      Single
Name: Type, Length: 809, dtype: object
```

# Invoking a Function on Every Series Value with the apply Method

- We have a new Series object now! Let's find out how many Pokémon fall into each classification by using `value_counts`.

```
In [59] pokemon.apply(single_or_multi).value_counts()
```

```
Out [59] Multi      405
          Single     404
          Name: Type, dtype: int64
```



# Coding Challenge: Deriving Insights from a Series

## Problem

- Let's tackle a challenge that combines several ideas introduced in the last two lessons. Our goal is to find out which day of the week saw the most battles during the Revolutionary War.
- The final output should be a Series with the days of the week (i.e. Sunday, Monday) as index labels and a count of battles on each day as the values.

# Coding Challenge: Deriving Insights from a Series

- When working with a datetime object, invoke the method strftime on it with a string argument of "%A" to return a day of a week (i.e. "Sunday"). See the example below

```
In [60] from datetime import datetime  
today = datetime.now()  
today.strftime("%A")  
  
Out [60] 'Sunday'
```

# Solution

- Our first step is to import the dataset. Let's remind ourselves of its original shape.

```
In [61] pd.read_csv("revolutionary_war.csv").head()
```

```
Out [61]
```

	Battle	Start Date	State
0	Powder Alarm	9/1/1774	Massachusetts
1	Storming of Fort William and Mary	12/14/1774	New Hampshire
2	Battles of Lexington and Concord	4/19/1775	Massachusetts
3	Siege of Boston	4/19/1775	Massachusetts
4	Gunpowder Incident	4/20/1775	Virginia

# Solution

- Finally, the squeeze parameter will create a Series instead of a DataFrame.

```
In [62] days_of_war = pd.read_csv("revolutionary_war.csv",
                                   usecols = ["Start Date"],
                                   parse_dates=["Start Date"],
                                   squeeze = True)

days_of_war.head()

Out [62] 0    1774-09-01
         1    1774-12-14
         2    1775-04-19
         3    1775-04-19
         4    1775-04-20
Name: Start Date, dtype: datetime64[ns]
```



# Solution



- Let's declare that function now.

```
In [63] def day_of_week(date):  
        return date.strftime("%A")
```

We can now pass this function as the first argument to the `apply` method. It should be invoked against every value, except...

```
In [64] days_of_war.apply(day_of_week)
```

```
-----  
ValueError                                     Traceback (most recent call last)  
<ipython-input-411-c133befd2940> in <module>  
----> 1 days_of_war.apply(day_of_week)  
  
ValueError: NaTType does not support strftime
```

# Solution

```
In [65] days_of_war.dropna().apply(day_of_week)

Out [65] 0      Thursday
        1      Wednesday
        2      Wednesday
        3      Wednesday
        4      Thursday
        ...
       227     Wednesday
       228     Friday
       229     Friday
       230     Friday
       231     Wednesday
Name: Start Date, Length: 228, dtype: object
```

# Solution

```
In [66] days_of_war.dropna().apply(day_of_week).value_counts()

Out [66]
Saturday      39
Friday         39
Wednesday     32
Thursday       31
Sunday         31
Tuesday        29
Monday         27
Name: Start Date, dtype: int64
```



# Summary



- The `read_csv` method imports a CSV's contents into a pandas data structure.
  - Parameters to `read_csv` can customize the included columns, the column that serves as the index, and the datatypes of columns
  - The `sort_values` and `sort_index` methods can sort the values or the index of a Series in ascending or descending order
- 

# Lesson 7 The DataFrame Object





# The DataFrame Object

---

This lesson covers:

- Instantiating a DataFrame object from a dictionary and a numpy ndarray
  - Importing a multidimensional dataset with the `read_csv` method
  - Sorting one or more columns in a DataFrame
  - Accessing rows and columns from a DataFrame
  - Setting and resetting the index of a DataFrame
  - Renaming column and index values
- 



# Overview of a DataFrame



- The workhorse of the Pandas library, the DataFrame is a 2-dimensional data structure consisting of rows and columns.
  - Two points of reference are needed to extract any given value from the dataset.
  - A DataFrame can be described as a grid or a table of data, similar to one you'd find in a spreadsheet application like Excel
- 



# Creating A DataFrame from a Dictionary



- As always, let's begin by importing Pandas.
- We'll also be using the NumPy library for some random data generation. It is commonly assigned the alias np.

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

# Creating A DataFrame from a Dictionary

```
In [2] city_data = {  
    "City": ["New York City", "Paris", "Barcelona", "Rome"],  
    "Country": ["United States", "France", "Spain", "Italy"],  
    "Population": [8600000, 2141000, 5515000, 2873000]  
}  
  
cities = pd.DataFrame(city_data)  
cities
```

Out [2]

	<b>City</b>	<b>Country</b>	<b>Population</b>
0	New York City	United States	8600000
1	Paris	France	2141000
2	Barcelona	Spain	5515000
3	Rome	Italy	2873000

# Creating A DataFrame from a Dictionary

What if we wanted the data flipped around, with our column headers serving as the index labels? There are a few options available here. On our existing DataFrame, we can either invoke the transpose method or access its T attribute.

```
In [3] cities.transpose() # is the same as  
      cities.T
```

```
Out [3]
```

	0	1	2	3
City	New York City	Paris	Barcelona	Rome
Country	United States	France	Spain	Italy
Population	8600000	2141000	5515000	2873000

# Creating A DataFrame from a Dictionary

- The method's orient parameter can be passed an argument of "index" to orient the headers as index labels

```
In [4] pd.DataFrame.from_dict(data = city_data, orient = "index")
```

```
Out [4]
```

	0	1	2	3
City	New York City	Paris	Barcelona	Rome
Country	United States	France	Spain	Italy
Population	8600000	2141000	5515000	2873000

# Creating A DataFrame from a Numpy ndarray

- The DataFrame constructor also accepts a NumPy ndarray object.
- Let's say we want to create a 3x5 DataFrame of integers between 1 and 100 (inclusive).
- We can begin by using the randint function on its random module to create our data.

```
In [5] data = np.random.randint(1, 101, [3, 5])
      data
```

```
Out [5] array([[25, 22, 80, 43, 42],
               [40, 89, 7, 21, 25],
               [89, 71, 32, 28, 39]])
```

# Creating A DataFrame from a Numpy ndarray

- Next, we pass our ndarray into the DataFrame constructor.
- Just like with the rows, Pandas will assign each column a numeric index if a set of custom column headers is not provided.

```
In [6] pd.DataFrame(data = data)
```

```
Out [6]
```

	0	1	2	3	4
0	25	22	80	43	42
1	40	89	7	21	25
2	89	71	32	28	39

# Creating A DataFrame from a Numpy ndarray

- This is a 3x5 table, so we must provide 3 labels for the indices.

```
In [7] index = ["Morning", "Afternoon", "Evening"]
       temperatures = pd.DataFrame(data = data, index = index)
       temperatures
```

```
Out [7]
```

	0	1	2	3	4
Morning	25	22	80	43	42
Afternoon	40	89	7	21	25
Evening	89	71	32	28	39

# Creating A DataFrame from a Numpy ndarray

```
In [8] index = ["Morning", "Afternoon", "Evening"]
       columns = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
       temperatures = pd.DataFrame(data = data,
                                      index = index,
                                      columns = columns)
       temperatures
```

Out [8]

	<b>Monday</b>	<b>Tuesday</b>	<b>Wednesday</b>	<b>Thursday</b>	<b>Friday</b>
Morning	25	22	80	43	42
Afternoon	40	89	7	21	25
Evening	89	71	32	28	39

# Creating A DataFrame from a Numpy ndarray

- Both the row and column indices are allowed to contain duplicates

```
In [9] index = ["Morning", "Afternoon", "Morning"]
       columns = ["Monday", "Tuesday", "Wednesday", "Tuesday", "Friday"]
       pd.DataFrame(data = data, index = index, columns = columns)
```

```
Out [9]
```

	<b>Monday</b>	<b>Tuesday</b>	<b>Wednesday</b>	<b>Tuesday</b>	<b>Friday</b>
<b>Morning</b>	25	22	80	43	42
<b>Afternoon</b>	40	89	7	21	25
<b>Morning</b>	89	71	32	28	39



# Similarities between Series and DataFrames



- Many of the Series attributes and methods introduced in the previous lessons are also available on a DataFrame.
- Of course, the implementations of them are often different; we're dealing with multiple columns now! Let's import our first dataset to try them out!

# Importing a CSV File with the `read_csv` Method

```
In [10] pd.read_csv("nba.csv")
```

```
Out [10]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	9/26/96	1445697
1	Christian Wood	Detroit Pistons	PF	9/27/95	1645357
2	PJ Washington	Charlotte Hornets	PF	8/23/98	3831840
3	Derrick Rose	Detroit Pistons	PG	10/4/88	7317074
4	Marial Shayok	Philadelphia 76ers	G	7/26/95	79568
...	...	...	...	...	...
445	Austin Rivers	Houston Rockets	PG	8/1/92	2174310
446	Harry Giles	Sacramento Kings	PF	4/22/98	2578800
447	Robin Lopez	Milwaukee Bucks	C	4/1/88	4767000
448	Collin Sexton	Cleveland Cavaliers	PG	1/4/99	4764960
449	Ricky Rubio	Phoenix Suns	PG	10/21/90	16200000
450	rows x 5 columns				

# Importing a CSV File with the `read_csv` Method

```
In [11]: pd.read_csv("nba.csv", parse_dates = ["Birthday"])
```

```
Out [11]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
2	PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
3	Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
4	Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568
...	...	...	...	...	...
445	Austin Rivers	Houston Rockets	PG	1992-08-01	2174310
446	Harry Giles	Sacramento Kings	PF	1998-04-22	2578800
447	Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
448	Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
449	Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000
450	rows x 5 columns				

# Importing a CSV File with the `read_csv` Method

- Much better! We can assign the DataFrame to a `nba` variable and get to work.

```
In [12] nba = pd.read_csv("nba.csv", parse_dates =  
["Birthday"])
```

# Shared and Exclusive Attributes between Series and DataFrames

- Let's try accessing our trusty dtypes attribute on nba.
- It will return a Series object with the DataFrame's columns and their respective data types.
- As a reminder, object is internal pandas lingo for strings

```
In [13] nba.dtypes
```

```
Out [13] Name          object
          Team         object
          Position      object
          Birthday    datetime64[ns]
          Salary        int64
          dtype: object
```

# Shared and Exclusive Attributes between Series and DataFrames

- We can count the number of columns with each data type by invoking the `value_counts` method on the resulting Series

```
In [14] nba.dtypes.value_counts()
```

```
Out [14] object          3
           datetime64[ns]    1
           int64             1
           dtype: int64
```



# Shared and Exclusive Attributes between Series and DataFrames



- The index attribute returns the underlying Index object for a DataFrame.
- Let's take a look at what kind of index Pandas is using for our nba dataset.

In [15] nba.index

Out [15] RangeIndex(start=0, stop=450, step=1)

# Shared and Exclusive Attributes between Series and DataFrames

```
In [16] nba.values

Out [16] array([['Shake Milton', 'Philadelphia 76ers', 'SG',
                  Timestamp('1996-09-26 00:00:00'), 1445697],
                  ['Christian Wood', 'Detroit Pistons', 'PF',
                  Timestamp('1995-09-27 00:00:00'), 1645357],
                  ['PJ Washington', 'Charlotte Hornets', 'PF',
                  Timestamp('1998-08-23 00:00:00'), 3831840],
                  ...,
                  ['Robin Lopez', 'Milwaukee Bucks', 'C',
                  Timestamp('1988-04-01 00:00:00'), 4767000],
                  ['Collin Sexton', 'Cleveland Cavaliers', 'PG',
                  Timestamp('1999-01-04 00:00:00'), 4764960],
                  ['Ricky Rubio', 'Phoenix Suns', 'PG',
                  Timestamp('1990-10-21 00:00:00'), 16200000]],
                  dtype=object)
```

# Shared and Exclusive Attributes between Series and DataFrames

The DataFrame also has an exclusive `columns` attribute which returns an Index object containing the headers.

```
In [17] nba.columns  
  
Out [17] Index(['Name', 'Team', 'Position', 'Birthday', 'Salary'],  
              dtype='object')
```

Both the horizontal and vertical indices are collected in a list referenced by the `axes` attribute.

```
In [18] nba.axes  
  
Out [18] [RangeIndex(start=0, stop=450, step=1),  
          Index(['Name', 'Team', 'Position', 'Birthday', 'Salary'],  
                 dtype='object')]
```

# Shared and Exclusive Attributes between Series and DataFrames

The `shape` attribute returns a tuple with the dimensions of the DataFrame. This one has 450 rows by 5 columns.

```
In [19] nba.shape  
Out [19] (450, 5)
```

The `ndim` attribute returns the number of dimensions.

```
In [20] nba.ndim  
Out [20] 2
```

# Shared and Exclusive Attributes between Series and DataFrames

- The size attribute calculates the total number of values in the dataset, including missing ones.
- It will be equal to the product of the number of rows and the number of columns.

```
In [21] nba.size
```

```
Out [21] 2250
```

```
In [22] len(nba.index) * len(nba.columns)
```

```
Out [22] 2250
```

# Shared and Exclusive Attributes between Series and DataFrames

```
In [23] nba.count()
```

```
Out [23] Name      450  
          Team     450  
          Position  450  
          Birthday 450  
          Salary    450  
          dtype: int64
```

```
In [24] nba.count().sum()
```

```
Out [24] 2250
```

# Shared Methods between Series and

```
In [25] nba.head(2)
```

```
Out [25]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357

```
In [26] nba.tail(n = 3)
```

```
Out [26]
```

	Name	Team	Position	Birthday	Salary
447	Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
448	Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
449	Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000

The `sample` method extracts a number of random rows from the DataFrame.

```
In [27] nba.sample(3)
```

```
Out [27]
```

Name	Team	Position	Birthday	Salary
Al Horford	Philadelphia 76ers	C	1986-06-03	28000000
Tristan Thompson	Cleveland Cavaliers	C	1991-03-13	18539130
Jusuf Nurkic	Portland Trail Blazers	C	1994-08-23	12000000



# Shared Methods between Series and DataFrames



- The `nunique` method returns a Series object with a count of unique values found in each column.

```
In [28] nba.nunique()
```

```
Out [28] Name        450
          Team         30
          Position      9
          Birthday     430
          Salary        269
          dtype: int64
```



# Shared Methods between Series and DataFrames

```
In [29] nba.max()
```

```
Out [29] Team           Washington Wizards
          Position        SG
          Birthday       2000-12-23 00:00:00
          Salary          40231758
          dtype: object
```

```
In [30] nba.min()
```

```
Out [30] Team           Atlanta Hawks
          Position        C
          Birthday       1977-01-26 00:00:00
          Salary          79568
          dtype: object
```



# Shared Methods between Series and DataFrames



- The argument can be either a string or a list of strings representing column names.

```
In [31] nba.nlargest(n = 4, columns = "Birthday")  
Out [31]
```

	Name	Team	Position	Birthday	Salary
205	Stephen Curry	Golden State Warriors	PG	1988-03-14	40231758
38	Chris Paul	Oklahoma City Thunder	PG	1985-05-06	38506482
219	Russell Westbrook	Houston Rockets	PG	1988-11-12	38506482
251	John Wall	Washington Wizards	PG	1990-09-06	38199000

# Shared Methods between Series and DataFrames

- The smallest datetime values are those occur the earliest in chronological order.

```
In [32] nba.nsmallest(3, columns = ["Birthday"])
```

```
Out [32]
```

	Name	Team	Position	Birthday	Salary
98	Vince Carter	Atlanta Hawks	PF	1977-01-26	2564753
196	Udonis Haslem	Miami Heat	C	1980-06-09	2564753
262	Kyle Korver	Milwaukee Bucks	PF	1981-03-17	6004753



# Shared Methods between Series and DataFrames



```
In [33] nba.sum(numeric_only = True)  
Out [33] Salary    3444112694  
          dtype: int64
```

Yup, the total combined salaries of these 450 NBA players is a cool 3.44 billion. We can find the average salary with the `mean` method.

```
In [34] nba.mean()  
Out [34] Salary    7.653584e+06  
          dtype: float64
```



# Shared Methods between Series and DataFrames

---

- Other statistical calculations like median and statistical deviation are also available.
- They will automatically filter for only numeric columns.

```
In [35] nba.median()  
  
Out [35] Salary    3303074.5  
          dtype: float64  
  
In [36] nba.std()  
  
Out [36] Salary    9.288810e+06  
          dtype: float64
```

# Sorting a DataFrame

```
In [37]: nba.sort_values("Name")      # is the same as  
       nba.sort_values(by = "Name")  
  
Out [37]
```

	Name	Team	Position	Birthday	Salary
52	Aaron Gordon	Orlando Magic	PF	1995-09-16	19863636
101	Aaron Holiday	Indiana Pacers	PG	1996-09-30	2239200
437	Abdel Nader	Oklahoma City Thunder	SF	1993-09-25	1618520
81	Adam Mokoka	Chicago Bulls	G	1998-07-18	79568
399	Admiral Schofield	Washington Wizards	SF	1997-03-30	1000000
...	...	...	...	...	...
159	Zach LaVine	Chicago Bulls	PG	1995-03-10	19500000
302	Zach Norvell	Los Angeles Lakers	SG	1997-12-09	79568
312	Zhaire Smith	Philadelphia 76ers	SG	1999-06-04	3058800
137	Zion Williamson	New Orleans Pelicans	F	2000-07-06	9757440
248	Zylan Cheatham	New Orleans Pelicans	SF	1995-11-17	79568

# Sorting a DataFrame

```
In [38] nba.sort_values("Birthday", ascending = False).head()
```

```
Out [38]
```

	Name	Team	Position	Birthday	Salary
136	Sekou Doumbouya	Detroit Pistons	SF	2000-12-23	3285120
432	Talen Horton-Tucker	Los Angeles Lakers	GF	2000-11-25	898310
137	Zion Williamson	New Orleans Pelicans	F	2000-07-06	9757440
313	RJ Barrett	New York Knicks	SG	2000-06-14	7839960
392	Jalen Lecque	Phoenix Suns	G	2000-06-13	898310

# Sort by Multiple Columns

```
In [39] nba.sort_values(by = ["Team", "Name"] )
```

```
Out [39]
```

	Name	Team	Position	Birthday	Salary
359	Alex Len	Atlanta Hawks	C	1993-06-16	4160000
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000
276	Brandon Goodwin	Atlanta Hawks	PG	1995-10-02	79568
438	Bruno Fernando	Atlanta Hawks	C	1998-08-15	1400000
194	Cam Reddish	Atlanta Hawks	SF	1999-09-01	4245720
...	...	...	...	...	...
418	Jordan McRae	Washington Wizards	PG	1991-03-28	1645357
273	Justin Robinson	Washington Wizards	PG	1997-10-12	898310
428	Moritz Wagner	Washington Wizards	C	1997-04-26	2063520
21	Rui Hachimura	Washington Wizards	PF	1998-02-08	4469160
36	Thomas Bryant	Washington Wizards	C	1997-07-31	8000000

# Sort by Multiple Columns

```
In [40] nba.sort_values(["Team", "Name"], ascending = False)
```

```
Out [40]
```

	Name	Team	Position	Birthday	Salary
36	Thomas Bryant	Washington Wizards	C	1997-07-31	8000000
21	Rui Hachimura	Washington Wizards	PF	1998-02-08	4469160
428	Moritz Wagner	Washington Wizards	C	1997-04-26	2063520
273	Justin Robinson	Washington Wizards	PG	1997-10-12	898310
418	Jordan McRae	Washington Wizards	PG	1991-03-28	1645357
...	...	...	...	...	...
194	Cam Reddish	Atlanta Hawks	SF	1999-09-01	4245720
438	Bruno Fernando	Atlanta Hawks	C	1998-08-15	1400000
276	Brandon Goodwin	Atlanta Hawks	PG	1995-10-02	79568
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000
359	Alex Len	Atlanta Hawks	C	1993-06-16	4160000

```
In [41] nba.sort_values(by = ["Team", "Salary"],  
                      ascending = [True, False])
```

Out [41]

	Name	Team	Position	Birthday	Salary
111	Chandler Parsons	Atlanta Hawks	SF	1988-10-25	25102512
28	Evan Turner	Atlanta Hawks	PG	1988-10-27	18606556
167	Allen Crabbe	Atlanta Hawks	SG	1992-04-09	18500000
213	De'Andre Hunter	Atlanta Hawks	SF	1997-12-02	7068360
339	Jabari Parker	Atlanta Hawks	PF	1995-03-15	6500000
...	...	...	...	...	...
80	Isaac Bonga	Washington Wizards	PG	1999-11-08	1416852
399	Admiral Schofield	Washington Wizards	SF	1997-03-30	1000000
273	Justin Robinson	Washington Wizards	PG	1997-10-12	898310
283	Garrison Mathews	Washington Wizards	SG	1996-10-24	79568
353	Chris Chiozza	Washington Wizards	PG	1995-11-21	79568

As always, the `inplace` parameter mutates the original DataFrame instead of returning a copy. There will be no output produced in Jupyter Notebook

```
In [42] nba.sort_values(by = ["Team", "Salary"],  
                      ascending = [True, False],  
                      inplace = True)
```

# Sort by Index

- The `sort_index` method sorts a DataFrame by its index values.

```
In [43]: nba.sort_index().head() # is the same as  
nba.sort_index(ascending = True).head()
```

```
Out [43]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
2	PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
3	Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
4	Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568

# Sort by Index

We can also reverse the sort order by passing `False` to the `ascending` parameter.

```
In [44] nba.sort_index(ascending = False).head()
```

```
Out [44]
```

	Name	Team	Position	Birthday	Salary
449	Ricky Rubio	Phoenix Suns	PG	1990-10-21	16200000
448	Collin Sexton	Cleveland Cavaliers	PG	1999-01-04	4764960
447	Robin Lopez	Milwaukee Bucks	C	1988-04-01	4767000
446	Harry Giles	Sacramento Kings	PF	1998-04-22	2578800
445	Austin Rivers	Houston Rockets	PG	1992-08-01	2174310

Finally, we can make any of these changes permanent with the `inplace` parameter.

```
In [45] nba.sort_index(inplace = True)
```



# Sort by Column Index

- To sort the columns in order, pass an argument of either 1 or "columns" to the axis parameter of the `sort_index`

```
In [46] nba.sort_index(axis = 1).head() # is the same as  
      nba.sort_index(axis = "columns").head()
```

```
Out [46]
```

	<b>Birthday</b>	<b>Name</b>	<b>Position</b>	<b>Salary</b>	<b>Team</b>
0	1996-09-26	Shake Milton	SG	1445697	Philadelphia 76ers
1	1995-09-27	Christian Wood	PF	1645357	Detroit Pistons
2	1998-08-23	PJ Washington	PF	3831840	Charlotte Hornets
3	1988-10-04	Derrick Rose	PG	7317074	Detroit Pistons
4	1995-07-26	Marial Shayok	G	79568	Philadelphia 76ers



# Sort by Column Index



- How about sorting the columns in reverse alphabetical order? As always, it's just a matter of combining the right method with the right arguments.

```
In [47]: nba.sort_index(axis = "columns", ascending = False).head()  
Out [47]
```

	<b>Team</b>	<b>Salary</b>	<b>Position</b>	<b>Name</b>	<b>Birthday</b>
0	Philadelphia 76ers	1445697	SG	Shake Milton	1996-09-26
1	Detroit Pistons	1645357	PF	Christian Wood	1995-09-27
2	Charlotte Hornets	3831840	PF	PJ Washington	1998-08-23
3	Detroit Pistons	7317074	PG	Derrick Rose	1988-10-04
4	Philadelphia 76ers	79568	G	Marial Shayok	1995-07-26

# Setting a New Index

```
In [48] nba.set_index(keys = "Name") # is the same as  
nba.set_index("Name")
```

```
Out [48]
```

Name		Team	Position	Birthday	Salary
Shake Milton		Philadelphia 76ers	SG	1996-09-26	1445697
Christian Wood		Detroit Pistons	PF	1995-09-27	1645357
PJ Washington		Charlotte Hornets	PF	1998-08-23	3831840
Derrick Rose		Detroit Pistons	PG	1988-10-04	7317074
Marial Shayok		Philadelphia 76ers	G	1995-07-26	79568
...		...	...	...	...
Austin Rivers		Houston Rockets	PG	1992-08-01	2174310
Harry Giles		Sacramento Kings	PF	1998-04-22	2578800
Robin Lopez		Milwaukee Bucks	C	1988-04-01	4767000
Collin Sexton		Cleveland Cavaliers	PG	1999-01-04	4764960
Ricky Rubio		Phoenix Suns	PG	1990-10-21	16200000



# Setting a New Index

Looks good! Let's make the operation permanent.

```
In [49] nba.set_index(keys = "Name", inplace = True)
```

If we know the column we'd like to use as the index when importing a dataset, we can also pass its name as a string to the `read_csv` method's `index_col` parameter.

```
In [50] nba = pd.read_csv("nba.csv",
                           parse_dates = ["Birthday"],
                           index_col = "Name")
```

# Select a Single Column from a DataFrame

```
In [51]: nba.Salary
```

```
Out [51]: Name
           Shake Milton      1445697
           Christian Wood     1645357
           PJ Washington      3831840
           Derrick Rose       7317074
           Marial Shayok      79568
           ...
           Austin Rivers      2174310
           Harry Giles        2578800
           Robin Lopez         4767000
           Collin Sexton       4764960
           Ricky Rubio        16200000
Name: Salary, Length: 450, dtype: int64
```

# Select a Single Column from a DataFrame

```
In [52] nba.Salary.to_frame()
```

```
Out [52]
```

Name	Salary
Shake Milton	1445697
Christian Wood	1645357
PJ Washington	3831840
Derrick Rose	7317074
Marial Shayok	79568
...	...
Austin Rivers	2174310
Harry Giles	2578800
Robin Lopez	4767000
Collin Sexton	4764960
Ricky Rubio	16200000

# Select a Single Column from a DataFrame

```
In [53] nba["Position"]
```

```
Out [53] Name
```

Shake Milton	SG
Christian Wood	PF
PJ Washington	PF
Derrick Rose	PG
Marial Shayok	G
	..
Austin Rivers	PG
Harry Giles	PF
Robin Lopez	C
Collin Sexton	PG
Ricky Rubio	PG

Name: Position, Length: 450, dtype: object

# Select Multiple Columns from a DataFrame

```
In [54] nba[["Salary", "Birthday"]]
```

```
Out [54]
```

Name	Salary	Birthday
Shake Milton	1445697	1996-09-26
Christian Wood	1645357	1995-09-27
PJ Washington	3831840	1998-08-23
Derrick Rose	7317074	1988-10-04
Marial Shayok	79568	1995-07-26
...	...	...
Austin Rivers	2174310	1992-08-01
Harry Giles	2578800	1998-04-22
Robin Lopez	4767000	1988-04-01
Collin Sexton	4764960	1999-01-04
Ricky Rubio	16200000	1990-10-21

# Select Multiple Columns from a DataFrame

```
In [55] # Select only string columns  
nba.select_dtypes(include = "object")
```

```
Out [55]
```

	Name	Team	Position
0	Shake Milton	Philadelphia 76ers	SG
1	Christian Wood	Detroit Pistons	PF
2	PJ Washington	Charlotte Hornets	PF
3	Derrick Rose	Detroit Pistons	PG
4	Marial Shayok	Philadelphia 76ers	G
...	...	...	...
445	Austin Rivers	Houston Rockets	PG
446	Harry Giles	Sacramento Kings	PF
447	Robin Lopez	Milwaukee Bucks	C
448	Collin Sexton	Cleveland Cavaliers	PG
449	Ricky Rubio	Phoenix Suns	PG

```
In [56] # Exclude string and integer columns  
nba.select_dtypes(exclude = ["object", "int"])
```

```
Out [56]
```

# Select Multiple Columns from a DataFrame

	<b>Birthday</b>
0	1996-09-26
1	1995-09-27
2	1998-08-23
3	1988-10-04
4	1995-07-26
...	...
445	1992-08-01
446	1998-04-22
447	1988-04-01
448	1999-01-04
449	1990-10-21



# Extract Rows by Index Label



The `loc` attribute accepts the label of a row to extract. It is declared with a pair of square brackets containing the index label and returns a Series object holding the values of the row with that label. As with everything else in Python, the search is case-sensitive.

```
In [57] nba.loc["LeBron James"]
```

```
Out [57] Team           Los Angeles Lakers
          Position        PF
          Birthday      1984-12-30 00:00:00
          Salary         37436858
          Name: LeBron James, dtype: object
```

# Extract Rows by Index Label

- We can also pass a list in between the square brackets to extract multiple rows. The result will be a DataFrame.

```
In [58] nba.loc[['Kawhi Leonard', 'Paul George']]
```

```
Out [58]
```

Name			Team	Position	Birthday	Salary
Kawhi Leonard		Los Angeles Clippers		SF	1991-06-29	32742000
Paul George		Los Angeles Clippers		SF	1990-05-02	33005556

# Extract Rows by Index Label

- The rows will be returned in the order the index labels appear in the list, not the order the labels appear in the DataFrame.

```
In [59] nba.loc[["Paul George", "Kawhi Leonard"]]
```

```
Out [59]
```

Name		Team	Position	Birthday	Salary
Paul George		Los Angeles	Clippers	SF 1990-05-02	33005556
Kawhi Leonard		Los Angeles	Clippers	SF 1991-06-29	32742000

# Extract Rows by Index Label

```
In [60] nba.sort_index().loc["Otto Porter":"Patrick Beverley"]
```

```
Out [60]
```

Name		Team	Position	Birthday	Salary
Otto Porter		Chicago Bulls	SF	1993-06-03	27250576
PJ Dozier		Denver Nuggets	PG	1996-10-25	79568
PJ Washington		Charlotte Hornets	PF	1998-08-23	3831840
Pascal Siakam		Toronto Raptors	PF	1994-04-02	2351838
Pat Connaughton		Milwaukee Bucks	SG	1993-01-06	1723050
Patrick Beverley	Los Angeles Clippers		PG	1988-07-12	12345680

# Extract Rows by Index Label

```
In [61] nba.sort_index().loc["Zach Collins":]
```

```
Out [61]
```

Name		Team	Position	Birthday	Salary
Zach Collins		Portland Trail Blazers	C	1997-11-19	4240200
Zach LaVine		Chicago Bulls	PG	1995-03-10	19500000
Zach Norvell		Los Angeles Lakers	SG	1997-12-09	79568
Zhaire Smith		Philadelphia 76ers	SG	1999-06-04	3058800
Zion Williamson		New Orleans Pelicans	F	2000-07-06	9757440
Zylan Cheatham		New Orleans Pelicans	SF	1995-11-17	79568

# Extract Rows by Index Label

```
In [62] nba.sort_index().loc[:"Al Horford"]
```

```
Out [62]
```

Name		Team	Position	Birthday	Salary
Aaron Gordon		Orlando Magic	PF	1995-09-16	19863636
Aaron Holiday		Indiana Pacers	PG	1996-09-30	2239200
Abdel Nader	Oklahoma City Thunder		SF	1993-09-25	1618520
Adam Mokoka	Chicago Bulls		G	1998-07-18	79568
Admiral Schofield	Washington Wizards		SF	1997-03-30	1000000
Al Horford	Philadelphia 76ers		C	1986-06-03	28000000

A `KeyError` exception will be raised if an index label does not exist in the DataFrame.

```
In [63] nba.loc["Bugs Bunny"]
```

```
-----
```

```
KeyError
```

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

```
KeyError: 'Bugs Bunny'
```

# Extract Rows by Index Position

```
In [64] nba.iloc[300]
```

```
Out [64] Team          Denver Nuggets
         Position       PF
         Birthday     1999-04-03 00:00:00
         Salary        1416852
         Name: Jarred Vanderbilt, dtype: object
```

```
In [65] nba.iloc[[100, 200, 300, 400]]
```

```
Out [65]
```

Name	Team	Position	Birthday	Salary
Brian Bowen	Indiana Pacers	SG	1998-10-02	79568
Marco Belinelli	San Antonio Spurs	SF	1986-03-25	5846154
Jarred Vanderbilt	Denver Nuggets	PF	1999-04-03	1416852
Louis King	Detroit Pistons	F	1999-04-06	79568

In [66] nba.iloc[400:404]

Out [66]

Name		Team	Position	Birthday	Salary
Louis King		Detroit Pistons	F	1999-04-06	79568
Kostas Antetokounmpo		Los Angeles Lakers	PF	1997-11-20	79568
Rodions Kurucs		Brooklyn Nets	PF	1998-02-05	1699236
Spencer Dinwiddie		Brooklyn Nets	PG	1993-04-06	10605600

In [67] nba.iloc[:2]

Out [67]

Name		Team	Position	Birthday	Salary
Shake Milton		Philadelphia 76ers	SG	1996-09-26	1445697
Christian Wood		Detroit Pistons	PF	1995-09-27	1645357

In [68] nba.iloc[447:]

Out [68]

Name		Team	Position	Birthday	Salary
Robin Lopez		Milwaukee Bucks	C	1988-04-01	4767000
Collin Sexton		Cleveland Cavaliers	PG	1999-01-04	4764960
Ricky Rubio		Phoenix Suns	PG	1990-10-21	16200000

# Extract Rows by Index Position

Negative numbers can also be passed for one or both of the values. The next example extracts from the 10th -to-last row up to but not including the 6 th -to-last row.

```
In [69]: nba.iloc[-10:-6]
```

```
Out [69]
```

Name	Team	Position	Birthday	Salary
Jared Dudley	Los Angeles Lakers	PF	1985-07-10	2564753
Max Strus	Chicago Bulls	SG	1996-03-28	79568
Kevon Looney	Golden State Warriors	C	1996-02-06	4464286
Willy Hernangomez	Charlotte Hornets	C	1994-05-27	1557250

# Extract Rows by Index Position

```
In [70] nba.iloc[0:10:2]
```

```
Out [70]
```

Name		Team	Position	Birthday	Salary
Shake Milton		Philadelphia 76ers	SG	1996-09-26	1445697
PJ Washington		Charlotte Hornets	PF	1998-08-23	3831840
Marial Shayok		Philadelphia 76ers	G	1995-07-26	79568
Kendrick Nunn		Miami Heat	SG	1995-08-03	1416852
Brook Lopez		Milwaukee Bucks	C	1988-04-01	12093024



# Extract Values from Specific Columns

---

- Both the loc and iloc attributes accept a second argument representing the column(s) to extract.
- In the example below, we use loc to identify the value at the intersection of the "Giannis Antetokounmpo" index label and the Team column.

```
In [71] nba.loc["Giannis Antetokounmpo", "Team"]
Out [71] 'Milwaukee Bucks'
```

# Extract Values from Specific Columns

```
In [72] # Extract the row with a "James Harden" index label and the  
# values from the "Position" and "Birthday" columns  
  
nba.loc["James Harden", ["Position", "Birthday"]]  
  
Out [72] Position          PG  
        Birthday   1989-08-26 00:00:00  
        Name: James Harden, dtype: object  
  
In [73] # Extract the rows with "Russell Westbrook" and "Anthony Davis"  
# index labels and values from the "Team" and "Salary" columns  
  
nba.loc[  
        ["Russell Westbrook", "Anthony Davis"],  
        ["Team", "Salary"]  
    ]  
Out [73]
```

Name	Team	Salary
Russell Westbrook	Houston Rockets	38506482
Anthony Davis	Los Angeles Lakers	27093019

# Extract Values from Specific Columns

- List slicing syntax can also be used to extract multiple columns without explicitly writing out all of their names.
- We have four columns in our dataset; let's extract all columns from Position to Salary. Both endpoints will be inclusive.

```
In [74] nba.loc["Joel Embiid", "Position":"Salary"]
```

```
Out [74] Position          C
          Birthday     1994-03-16 00:00:00
          Salary        27504630
          Name: Joel Embiid, dtype: object
```

# Extract Values from Specific Columns

The column names must be passed in the order they appear in the DataFrame. The code below yields an empty result because the **Salary** column comes after the **Position** column.

```
In [75] nba.loc["Joel Embiid", "Salary":"Position"]
```

```
Out [75] Series([], Name: Joel Embiid, dtype: object)
```

Each DataFrame column is assigned an index position. In our current DataFrame, **Team** has an index of 0, **Position** has an index of 1, and so on.

```
In [76] nba.columns
```

```
Out [76] Index(['Team', 'Position', 'Birthday', 'Salary'], dtype='object')
```

The index position of a column can also be passed as the second argument to iloc.

```
In [77] nba.iloc[57, 3]
```

```
Out [77] 796806
```

# Extract Values from Specific Columns

```
In [78] nba.iloc[100:104, :3]
```

```
Out [78]
```

Name	Team	Position	Birthday
Brian Bowen	Indiana	Pacers	SG 1998-10-02
Aaron Holiday	Indiana	Pacers	PG 1996-09-30
Troy Daniels	Los Angeles	Lakers	SG 1991-07-15
Buddy Hield	Sacramento	Kings	SG 1992-12-17



# Extract Values from Specific Columns

---

- Two alternatives attributes, at and iat, are available when we want to extract a single value from a DataFrame.
- The at attribute accepts the row and columns labels, while the iat attributes accepts the row and column indices.

```
In [79] nba.at["Austin Rivers", "Birthday"]
```

```
Out [79] Timestamp('1992-08-01 00:00:00')
```

```
In [80] nba.iat[263, 1]
```

```
Out [80] 'PF'
```

# Extract Values from Specific Columns

```
In [81]: %timeit  
       nba.at["Austin Rivers", "Birthday"]  
  
6.38 µs ± 53.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)  
  
In [82]: %timeit  
       nba.loc["Austin Rivers", "Birthday"]  
  
9.12 µs ± 53.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)  
  
In [83]: %timeit  
       nba.iat[263, 1]  
  
4.7 µs ± 27.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)  
  
In [84]: %timeit  
       nba.iloc[263, 1]  
  
7.41 µs ± 39.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```



# Extract Value from Series



```
In [85] nba["Salary"].loc["Damian Lillard"]
```

```
Out [85] 29802321
```

```
In [86] nba["Salary"].at["Damian Lillard"]
```

```
Out [86] 29802321
```

```
In [87] nba["Salary"].iloc[234]
```

```
Out [87] 2033160
```

```
In [88] nba["Salary"].iat[234]
```

```
Out [88] 2033160
```

# Rename Column or Row

```
In [89] nba.columns  
Out [89] Index(['Team', 'Position', 'Birthday', 'Pay'], dtype='object')  
In [90] nba.columns = ["Team", "Position", "Date of Birth", "Pay"]  
nba.head(1)  
Out [90]
```

Name	Team	Position	Date of Birth	Pay
Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697

The rename method is an alternate option. We can pass its columns parameter a dictionary with keys representing the existing column names and values representing their new names.

```
In [91] nba.rename(columns = { "Date of Birth": "Birthday" })
```

```
Out [91]
```

Name		Team	Position	Birthday	Pay
Shake Milton		Philadelphia 76ers	SG	1996-09-26	1445697
Christian Wood		Detroit Pistons	PF	1995-09-27	1645357
PJ Washington		Charlotte Hornets	PF	1998-08-23	3831840
Derrick Rose		Detroit Pistons	PG	1988-10-04	7317074
Marial Shayok		Philadelphia 76ers	G	1995-07-26	79568
...	...	...	...	...	...
Austin Rivers		Houston Rockets	PG	1992-08-01	2174310
Harry Giles		Sacramento Kings	PF	1998-04-22	2578800
Robin Lopez		Milwaukee Bucks	C	1988-04-01	4767000
Collin Sexton		Cleveland Cavaliers	PG	1999-01-04	4764960
Ricky Rubio		Phoenix Suns	PG	1990-10-21	16200000

As always, use the inplace parameter to make the operation permanent.

```
In [92] nba.rename(  
    columns = { "Date of Birth": "Birthday" },  
    inplace = True  
)
```

# Rename Column or Row

```
In [93] nba.loc["Giannis Antetokounmpo"]

Out [93] Team           Milwaukee Bucks
          Position        PF
          Birthday       1994-12-06 00:00:00
          Pay            25842697
          Name: Giannis Antetokounmpo, dtype: object

In [94] nba.rename(
          index = { "Giannis Antetokounmpo": "Greek Freak" },
          inplace = True
        )

In [95] nba.loc["Greek Freak"]

Out [95] Team           Milwaukee Bucks
          Position        PF
          Birthday       1994-12-06 00:00:00
          Pay            25842697
          Name: Greek Freak, dtype: object
```

# Resetting an Index

What if we wanted another column to serve as the index of our DataFrame? We *could* invoke the `set_index` method again with a different column but, unfortunately, that would lose the current index of player names.

```
In [96]: nba.set_index("Team").head()
```

```
Out [96]
```

	<b>Position</b>	<b>Birthday</b>	<b>Salary</b>
<b>Team</b>			
Philadelphia 76ers	SG	1996-09-26	1445697
Detroit Pistons	PF	1995-09-27	1645357
Charlotte Hornets	PF	1998-08-23	3831840
Detroit Pistons	PG	1988-10-04	7317074
Philadelphia 76ers	G	1995-07-26	79568

# Resetting an Index

In order to preserve the player's names, we need to first re-integrate the existing index as a regular column in our DataFrame. The `reset_index` method moves an existing index's values into a column and generates a fresh sequential index.

```
In [97] nba.reset_index().head()
```

```
Out [97]
```

	Name	Team	Position	Birthday	Salary
0	Shake Milton	Philadelphia 76ers	SG	1996-09-26	1445697
1	Christian Wood	Detroit Pistons	PF	1995-09-27	1645357
2	PJ Washington	Charlotte Hornets	PF	1998-08-23	3831840
3	Derrick Rose	Detroit Pistons	PG	1988-10-04	7317074
4	Marial Shayok	Philadelphia 76ers	G	1995-07-26	79568

# Resetting an Index

- Now we're in the clear to use the `set_index` method.

```
In [98] nba.reset_index().set_index("Team").head()  
Out [98]
```

		Name	Position	Birthday	Salary
Team					
Philadelphia 76ers	Shake Milton	SG	1996-09-26	1445697	
Detroit Pistons	Christian Wood	PF	1995-09-27	1645357	
Charlotte Hornets	PJ Washington	PF	1998-08-23	3831840	
Detroit Pistons	Derrick Rose	PG	1988-10-04	7317074	
Philadelphia 76ers	Marial Shayok	G	1995-07-26	79568	



# Resetting an Index



- The `reset_index` method also accepts an `inplace` parameter. Be careful, however.
- If the parameter is set to `True`, the method will not return a new `DataFrame` and thus the `set_index` method cannot be chained on in sequence.
- We'll have to rely on two separate method calls in sequence.

```
In [99] nba.reset_index(inplace = True)
        nba.set_index("Name", inplace = True)
```

# Coding Challenge

```
In [100] nfl = pd.read_csv("nfl.csv", parse_dates = ["Birthday"])
nfl
```

```
Out [100]
```

	Name	Team	Position	Birthday	Salary
0	Tremon Smith	Philadelphia Eagles	RB	1996-07-20	570000
1	Shawn Williams	Cincinnati Bengals	SS	1991-05-13	3500000
2	Adam Butler	New England Patriots	DT	1994-04-12	645000
3	Derek Wolfe	Denver Broncos	DE	1990-02-24	8000000
4	Jake Ryan	Jacksonville Jaguars	OLB	1992-02-27	1000000
...	...	...	...	...	...
1650	Bashaud Breeland	Kansas City Chiefs	CB	1992-01-30	805000
1651	Craig James	Philadelphia Eagles	CB	1996-04-29	570000
1652	Jonotthan Harrison	New York Jets	C	1991-08-25	1500000
1653	Chuma Edoga	New York Jets	OT	1997-05-25	495000
1654	Tajae Sharpe	Tennessee Titans	WR	1994-12-23	2025000
1655	rows × 5 columns				



# Coding Challenge

- What are the two ways we can overwrite the index of the DataFrame to store the player names?
- Our first option is to invoke the `set_index` method on our existing DataFrame with an `inplace` argument of `True`.

In [101] `nfl.set_index("Name", inplace = True)`

- Our second option is to use the `index_col` parameter with the `read_csv` method when importing the dataset.

```
In [102] nfl = pd.read_csv("nfl.csv", index_col = "Name",
                           parse_dates = ["Birthday"])
```

# Coding Challenge

- The results will be the same either way.

```
In [103] nfl.head()
```

```
Out [103]
```

Name		Team	Position	Birthday	Salary
Tremon Smith		Philadelphia Eagles	RB	1996-07-20	570000
Shawn Williams		Cincinnati Bengals	SS	1991-05-13	3500000
Adam Butler		New England Patriots	DT	1994-04-12	645000
Derek Wolfe		Denver Broncos	DE	1990-02-24	8000000
Jake Ryan		Jacksonville Jaguars	OLB	1992-02-27	1000000



# Coding Challenge

3. How can we get a count of the number of players per team in this dataset?

We can invoke the `value_counts` method on the **Team** column. Extract the Series with either dot syntax or square brackets. The results below are truncated for brevity.

```
In [104] nfl.Team.value_counts()      # is the same as  
        nfl["Team"].value_counts()
```

```
Out [104] New York Jets          58  
          Washington Redskins     56  
          Kansas City Chiefs     56  
          San Francisco 49Ers    55  
          New Orleans Saints     55
```



# Coding Challenge

4. Who are the five highest paid players in this dataset?

The `sort_values` method can sort the **Salary** column for us. To modify its default ascending sort order, we pass a `False` argument to the `ascending` parameter.

```
In [105] nfl.sort_values("Salary", ascending = False).head()
```

```
Out [105]
```

Name		Team	Position	Birthday	Salary
Kirk Cousins		Minnesota Vikings	QB	1988-08-19	27500000
Jameis Winston		Tampa Bay Buccaneers	QB	1994-01-06	20922000
Marcus Mariota		Tennessee Titans	QB	1993-10-30	20922000
Derek Carr		Oakland Raiders	QB	1991-03-28	19900000
Jimmy Garoppolo		San Francisco 49Ers	QB	1991-11-02	17200000

# Coding Challenge

```
In [106] nfl.sort_values(by = ["Team", "Salary"],  
                        ascending = [True, False])
```

```
Out [106]
```

Name		Team	Position	Birthday	Salary
Chandler Jones		Arizona Cardinals	OLB	1990-02-27	16500000
Patrick Peterson		Arizona Cardinals	CB	1990-07-11	11000000
Larry Fitzgerald		Arizona Cardinals	WR	1983-08-31	11000000
David Johnson		Arizona Cardinals	RB	1991-12-16	5700000
Justin Pugh		Arizona Cardinals	G	1990-08-15	5000000
	...	...	...	...	...
Ross Pierschbacher		Washington Redskins	C	1995-05-05	495000
Kelvin Harmon		Washington Redskins	WR	1996-12-15	495000
Wes Martin		Washington Redskins	G	1996-05-09	495000
Jimmy Moreland		Washington Redskins	CB	1995-08-26	495000
Jeremy Reaves		Washington Redskins	SS	1996-08-29	495000

```
In [107] nfl.reset_index(inplace = True)
         nfl.set_index(keys = "Team", inplace = True)
         nfl.head(3)
```

```
Out [107]
```

<b>Team</b>		<b>Name</b>	<b>Position</b>	<b>Birthday</b>	<b>Salary</b>
Philadelphia Eagles		Tremon Smith	RB	1996-07-20	570000
Cincinnati Bengals		Shawn Williams	SS	1991-05-13	3500000
New England Patriots		Adam Butler	DT	1994-04-12	645000

Next, we can use the loc attribute to isolate all players on the New York Jets.

```
In [108] nfl.loc["New York Jets"].head()
```

```
Out [108]
```

<b>Team</b>		<b>Name</b>	<b>Position</b>	<b>Birthday</b>	<b>Salary</b>
New York Jets		Bronson Kaufusi	DE	1991-07-06	645000
New York Jets		Darryl Roberts	CB	1990-11-26	1000000
New York Jets		Jordan Willis	DE	1995-05-02	754750
New York Jets		Quinnen Williams	DE	1997-12-21	495000
New York Jets		Sam Ficken	K	1992-12-14	495000



# Coding Challenge



- The last step is to sort the Birthday column and extract the top record. This sort is only possible because we converted the values to store datetime objects.

```
In [109] nfl.loc["New York Jets"].sort_values("Birthday").head(1)
```

```
Out [109]
```

<b>Team</b>		<b>Name</b>	<b>Position</b>	<b>Birthday</b>	<b>Salary</b>
New York Jets	Ryan Kalil		C	1985-03-29	2400000



# Summary



- The DataFrame is a two-dimensional data structure consisting of rows and columns.
  - The DataFrame shares attributes and methods with the Series, some of which operate differently due to the differences between the two objects.
  - The `sort_values` method sorts one or more columns. A different sort order can be applied to each column sort.
- 

# Lesson 8 Filtering a DataFrame





# Filtering a DataFrame

---

This lesson covers:

- Reducing the memory usage of a DataFrame
  - Extracting a subset of rows from a DataFrame based on one or more conditions
  - Filtering for rows that include or exclude null values
  - Selecting values that fall between a range
  - Removing duplicate and null values from a DataFrame
- 