Principles and Techniques of Data Science

Data 100

Bella Crouch Yash Dave Kanu Grover Ishani Gupta
Sakshi Kolli Minh Phan Milad Shafaie Matthew Shen
Lillian Weng

Table of contents

Welcome

About the Course Notes

This text offers supplementary resources to accompany lectures presented in the Spring 2024 Edition of the UC Berkeley course Data 100: Principles and Techniques of Data Science.

New notes will be added each week to accompany live lectures. See the full calendar of lectures on the course website.

If you spot any typos or would like to suggest any changes, please fill out the Data 100 Content Feedback Form (Spring 2024). Note that this link will only work if you have an @berkeley.edu email address. If you're not a student at Berkeley and would like to provide feedback, please email us at **data100.instructors@berkeley.edu**.

1 Introduction

> **Learning Outcomes**
>
> - Acquaint yourself with the overarching goals of Data 100
> - Understand the stages of the data science lifecycle

Data science is an interdisciplinary field with a variety of applications and offers great potential to address challenging societal issues. By building data science skills, you can empower yourself to participate in and drive conversations that shape your life and society as a whole, whether that be fighting against climate change, launching diversity initiatives, or more.

The field of data science is rapidly evolving; many of the key technical underpinnings in modern-day data science have been popularized during the early 21$^{\text{st}}$ century, and you will learn them throughout the course. It has a wide range of applications from science and medicine to sports.

While data science has immense potential to address challenging problems facing society by enhancing our critical thinking, it can also be used obscure complex decisions and reinforce historical trends and biases. This course will implore you to consider the ethics of data science within its applications.

Data science is fundamentally human-centered and facilitates decision-making by quantitatively balancing tradeoffs. To quantify things reliably, we must use and analyze data appropriately, apply critical thinking and skepticism at every step of the way, and consider how our decisions affect others.

Ultimately, data science is the application of data-centric, computational, and inferential thinking to:

- Understand the world (science).
- Solve problems (engineering).

A true mastery of data science requires a deep theoretical understanding and strong grasp of domain expertise. This course will help you build on the former – specifically, the foundation of your technical knowledge, allowing you to take data and produce useful insights on the world's most challenging and ambiguous problems.

Course Goals

- Prepare you for advanced Berkeley courses in **data management, machine learning, and statistics.**
- Enable you to launch a career as a data scientist by providing experience working with **real-world data, tools, and techniques**.
- Empower you to apply computational and inferential thinking to address **real-world problems**.

Some Topics We'll Cover

- `pandas` and `NumPy`
- Exploratory Data Analysis
- Regular Expressions
- Visualization
- Sampling
- Model Design and Loss Formulation
- Linear Regression
- Gradient Descent
- Logistic Regression
- Clustering
- PCA

Prerequisites

To ensure that you can get the most out of the course content, please make sure that you are familiar with:

- Using Python.
- Using Jupyter notebooks.
- Inference from Data 8.
- Linear algebra

To set you up for success, we've organized concepts in Data 100 around the **data science lifecycle**: an *iterative* process that encompasses the various statistical and computational building blocks of data science.

1.1 Data Science Lifecycle

The data science lifecycle is a *high-level overview* of the data science workflow. It's a cycle of stages that a data scientist should explore as they conduct a thorough analysis of a data-driven problem.

There are many variations of the key ideas present in the data science lifecycle. In Data 100, we visualize the stages of the lifecycle using a flow diagram. Notice how there are two entry points.

1.1.1 Ask a Question

Whether by curiosity or necessity, data scientists constantly ask questions. For example, in the business world, data scientists may be interested in predicting the profit generated by a certain investment. In the field of medicine, they may ask whether some patients are more likely than others to benefit from a treatment.

Posing questions is one of the primary ways the data science lifecycle begins. It helps to fully define the question. Here are some things you should ask yourself before framing a question.

- What do we want to know?
  - A question that is too ambiguous may lead to confusion.
- What problems are we trying to solve?
  - The goal of asking a question should be clear in order to justify your efforts to stakeholders.
- What are the hypotheses we want to test?
  - This gives a clear perspective from which to analyze final results.
- What are the metrics for our success?
  - This establishes a clear point to know when to conclude the project.

1.1.2 Obtain Data

The second entry point to the lifecycle is by obtaining data. A careful analysis of any problem requires the use of data. Data may be readily available to us, or we may have to embark on a process to collect it. When doing so, it is crucial to ask the following:

- What data do we have, and what data do we need?
  - Define the units of the data (people, cities, points in time, etc.) and what features to measure.
- How will we sample more data?
  - Scrape the web, collect manually, run experiments, etc.

- Is our data representative of the population we want to study?
  - If our data is not representative of our population of interest, then we can come to incorrect conclusions.

Key procedures: *data acquisition*, *data cleaning*

1.1.3 Understand the Data

Raw data itself is not inherently useful. It's impossible to discern all the patterns and relationships between variables without carefully investigating them. Therefore, translating pure data into actionable insights is a key job of a data scientist. For example, we may choose to ask:

- How is our data organized, and what does it contain?
  - Knowing what the data says about the world helps us better understand the world.
- Do we have relevant data?
  - If the data we have collected is not useful to the question at hand, then we must collect more data.
- What are the biases, anomalies, or other issues with the data?
  - These can lead to many false conclusions if ignored, so data scientists must always be aware of these issues.
- How do we transform the data to enable effective analysis?
  - Data is not always easy to interpret at first glance, so a data scientist should strive to reveal the hidden insights.

Key procedures: *exploratory data analysis*, *data visualization.*

1.1.4 Understand the World

After observing the patterns in our data, we can begin answering our questions. This may require that we predict a quantity (machine learning) or measure the effect of some treatment (inference).

From here, we may choose to report our results, or possibly conduct more analysis. We may not be satisfied with our findings, or our initial exploration may have brought up new questions that require new data.

- What does the data say about the world?

- Given our models, the data will lead us to certain conclusions about the real world.

- Does it answer our questions or accurately solve the problem?
  - If our model and data can not accomplish our goals, then we must reform our question, model, or both.

- How robust are our conclusions and can we trust the predictions?
  - Inaccurate models can lead to false conclusions.

Key procedures: *model creation*, *prediction*, *inference*.

1.2 Conclusion

The data science lifecycle is meant to be a set of general guidelines rather than a hard set of requirements. In our journey exploring the lifecycle, we'll cover both the underlying theory and technologies used in data science. By the end of the course, we hope that you start to see yourself as a data scientist.

With that, we'll begin by introducing one of the most important tools in exploratory data analysis: `pandas`.

2 Pandas I

ℹ Learning Outcomes

- Build familiarity with `pandas` and `pandas` syntax.
- Learn key data structures: `DataFrame`, `Series`, and `Index`.
- Understand methods for extracting data: `.loc`, `.iloc`, and `[]`.

In this sequence of lectures, we will dive right into things by having you explore and manipulate real-world data. We'll first introduce `pandas`, a popular Python library for interacting with **tabular data**.

2.1 Tabular Data

Data scientists work with data stored in a variety of formats. This class focuses primarily on *tabular data* — data that is stored in a table.

Tabular data is one of the most common systems that data scientists use to organize data. This is in large part due to the simplicity and flexibility of tables. Tables allow us to represent each **observation**, or instance of collecting data from an individual, as its own *row*. We can record each observation's distinct characteristics, or **features**, in separate *columns*.

To see this in action, we'll explore the `elections` dataset, which stores information about political candidates who ran for president of the United States in previous years.

In the `elections` dataset, each row (blue box) represents one instance of a candidate running for president in a particular year. For example, the first row represents Andrew Jackson running for president in the year 1824. Each column (yellow box) represents one characteristic piece of information about each presidential candidate. For example, the column named "Result" stores whether or not the candidate won the election.

Your work in Data 8 helped you grow very familiar with using and interpreting data stored in a tabular format. Back then, you used the `Table` class of the `datascience` library, a special programming library created specifically for Data 8 students.

In Data 100, we will be working with the programming library `pandas`, which is generally accepted in the data science community as the industry- and academia-standard tool for manipulating tabular data (as well as the inspiration for Petey, our panda bear mascot).

Using `pandas`, we can

- Arrange data in a tabular format.
- Extract useful information filtered by specific conditions.
- Operate on data to gain new insights.
- Apply `NumPy` functions to our data (our friends from Data 8).
- Perform vectorized computations to speed up our analysis (Lab 1).

2.2 `Series`, `DataFrames`, and Indices

To begin our work in `pandas`, we must first import the library into our Python environment. This will allow us to use `pandas` data structures and methods in our code.

```
# `pd` is the conventional alias for Pandas, as `np` is for NumPy
import pandas as pd
```

There are three fundamental data structures in `pandas`:

1. **`Series`**: 1D labeled array data; best thought of as columnar data.
2. **`DataFrame`**: 2D tabular data with rows and columns.
3. **`Index`**: A sequence of row/column labels.

`DataFrames`, `Series`, and Indices can be represented visually in the following diagram, which considers the first few rows of the `elections` dataset.

Notice how the **DataFrame** is a two-dimensional object — it contains both rows and columns. The **Series** above is a singular column of this `DataFrame`, namely the `Result` column. Both contain an **Index**, or a shared list of row labels (the integers from 0 to 4, inclusive).

2.2.1 Series

A `Series` represents a column of a `DataFrame`; more generally, it can be any 1-dimensional array-like object. It contains both:

- A sequence of **values** of the same type.
- A sequence of data labels called the **index**.

In the cell below, we create a `Series` named `s`.

```
s = pd.Series(["welcome", "to", "data 100"])
s
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	0
0	welcome
1	to
2	data 100

```python
# Accessing data values within the Series
s.values
```

```
array(['welcome', 'to', 'data 100'], dtype=object)
```

```python
# Accessing the Index of the Series
s.index
```

```
RangeIndex(start=0, stop=3, step=1)
```

By default, the `index` of a `Series` is a sequential list of integers beginning from 0. Optionally, a manually specified list of desired indices can be passed to the `index` argument.

```python
s = pd.Series([-1, 10, 2], index = ["a", "b", "c"])
s
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	0
a	-1
b	10
c	2

```python
s.index
```

```
Index(['a', 'b', 'c'], dtype='object')
```

Indices can also be changed after initialization.

```python
s.index = ["first", "second", "third"]
s
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	0
first	-1
second	10
third	2

```python
s.index
```

```
Index(['first', 'second', 'third'], dtype='object')
```

2.2.1.1 Selection in `Series`

Much like when working with `NumPy` arrays, we can select a single value or a set of values from a `Series`. To do so, there are three primary methods:

1. A single label.
2. A list of labels.
3. A filtering condition.

To demonstrate this, let's define the Series `ser`.

```python
ser = pd.Series([4, -2, 0, 6], index = ["a", "b", "c", "d"])
ser
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	0
a	4
b	-2
c	0
d	6

2.2.1.1.1 A Single Label

```python
# We return the value stored at the index label "a"
ser["a"]
```

4

2.2.1.1.2 A List of Labels

```python
# We return a Series of the values stored at the index labels "a" and "c"
ser[["a", "c"]]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	0
a	4
c	0

2.2.1.1.3 A Filtering Condition

Perhaps the most interesting (and useful) method of selecting data from a `Series` is by using a filtering condition.

First, we apply a boolean operation to the `Series`. This creates **a new `Series` of boolean values**.

```python
# Filter condition: select all elements greater than 0
ser > 0
```

	0
a	True
b	False
c	False
d	True

We then use this boolean condition to index into our original `Series`. `pandas` will select only the entries in the original `Series` that satisfy the condition.

```
ser[ser > 0]
```

	0
a	4
d	6

2.2.2 `DataFrames`

Typically, we will work with `Series` using the perspective that they are columns in a `DataFrame`. We can think of a **`DataFrame`** as a collection of **`Series`** that all share the same **`Index`**.

In Data 8, you encountered the `Table` class of the `datascience` library, which represented tabular data. In Data 100, we'll be using the `DataFrame` class of the `pandas` library.

2.2.2.1 Creating a `DataFrame`

There are many ways to create a `DataFrame`. Here, we will cover the most popular approaches:

1. From a CSV file.
2. Using a list and column name(s).
3. From a dictionary.
4. From a `Series`.

More generally, the syntax for creating a `DataFrame` is:

```
pandas.DataFrame(data, index, columns)
```

2.2.2.1.1 From a CSV file

In Data 100, our data are typically stored in a CSV (comma-separated values) file format. We can import a CSV file into a `DataFrame` by passing the data path as an argument to the following `pandas` function. `pd.read_csv("filename.csv")`

With our new understanding of `pandas` in hand, let's return to the `elections` dataset from before. Now, we can recognize that it is represented as a `pandas DataFrame`.

```
elections = pd.read_csv("data/elections.csv")
elections
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789
5	1832	Henry Clay	National Republican	484205	loss	37.603628
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
7	1836	Hugh Lawson White	Whig	146109	loss	10.005985
8	1836	Martin Van Buren	Democratic	763291	win	52.272472
9	1836	William Henry Harrison	Whig	550816	loss	37.721543
10	1840	Martin Van Buren	Democratic	1128854	loss	46.948787
11	1840	William Henry Harrison	Whig	1275583	win	53.051213
12	1844	Henry Clay	Whig	1300004	loss	49.250523
13	1844	James Polk	Democratic	1339570	win	50.749477
14	1848	Lewis Cass	Democratic	1223460	loss	42.552229
15	1848	Martin Van Buren	Free Soil	291501	loss	10.138474
16	1848	Zachary Taylor	Whig	1360235	win	47.309296
17	1852	Franklin Pierce	Democratic	1605943	win	51.013168
18	1852	John P. Hale	Free Soil	155210	loss	4.930283
19	1852	Winfield Scott	Whig	1386942	loss	44.056548
20	1856	James Buchanan	Democratic	1835140	win	45.306080
21	1856	John C. Frémont	Republican	1342345	loss	33.139919
22	1856	Millard Fillmore	American	873053	loss	21.554001
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
27	1864	Abraham Lincoln	National Union	2211317	win	54.951512
28	1864	George B. McClellan	Democratic	1812807	loss	45.048488
29	1868	Horatio Seymour	Democratic	2708744	loss	47.334695
30	1868	Ulysses Grant	Republican	3013790	win	52.665305
31	1872	Horace Greeley	Liberal Republican	2834761	loss	44.071406
32	1872	Ulysses Grant	Republican	3597439	win	55.928594
33	1876	Rutherford Hayes	Republican	4034142	win	48.471624
34	1876	Samuel J. Tilden	Democratic	4288546	loss	51.528376
35	1880	James B. Weaver	Greenback	308649	loss	3.352344
36	1880	James Garfield	Republican	4453337	win	48.369234
37	1880	Winfield Scott Hancock	Democratic	4444976	loss	48.278422
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
39	1884	Grover Cleveland	Democratic	4914482	win	48.884933
40	1884	James G. Blaine	Republican	4856905	loss	48.312208
41	1884	John St. John	Prohibition	147482	loss	1.467021
42	1888	Alson Streeter	Union Labor	146602	loss	1.288861
43	1888	Benjamin Harrison	Republican	5443633	win	47.858041
44	1888	Clinton B. Fisk	Prohibition	249819	loss	2.196299
45	1888	Grover Cleveland	Democratic	5534488	loss	48.656799
46	1892	Benjamin Harrison	Republican	5176108	loss	42.984101
47	1892	Grover Cleveland	Democratic	5553898	win	46.121393
48	1892	James B. Weaver	Populist	1041028	loss	8.645038
49	1892	John Bidwell	Prohibition	270879	loss	2.249468
50	1896	John M. Palmer	National Democratic	134645	loss	0.969566
51	1896	Joshua Levering	Prohibition	131312	loss	0.945565

This code stores our DataFrame object in the elections variable. Upon inspection, our elections DataFrame has 182 rows and 6 columns (Year, Candidate, Party, Popular Vote, Result, %). Each row represents a single record — in our example, a presidential candidate from some particular year. Each column represents a single attribute or feature of the record.

2.2.2.1.2 Using a List and Column Name(s)

We'll now explore creating a DataFrame with data of our own.

Consider the following examples. The first code cell creates a DataFrame with a single column Numbers.

```
df_list = pd.DataFrame([1, 2, 3], columns=["Numbers"])
df_list
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Numbers
0	1
1	2
2	3

The second creates a DataFrame with the columns Numbers and Description. Notice how a 2D list of values is required to initialize the second DataFrame — each nested list represents a single row of data.

```
df_list = pd.DataFrame([[1, "one"], [2, "two"]], columns = ["Number", "Description"])
df_list
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Number	Description
0	1	one
1	2	two

2.2.2.1.3 From a Dictionary

A third (and more common) way to create a `DataFrame` is with a dictionary. The dictionary keys represent the column names, and the dictionary values represent the column values.

Below are two ways of implementing this approach. The first is based on specifying the columns of the `DataFrame`, whereas the second is based on specifying the rows of the `DataFrame`.

```
df_dict = pd.DataFrame({
    "Fruit": ["Strawberry", "Orange"],
    "Price": [5.49, 3.99]
})
df_dict
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

```
df_dict = pd.DataFrame(
    [
        {"Fruit":"Strawberry", "Price":5.49},
        {"Fruit": "Orange", "Price":3.99}
    ]
)
df_dict
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

2.2.2.1.4 From a Series

Earlier, we explained how a `Series` was synonymous to a column in a `DataFrame`. It follows, then, that a `DataFrame` is equivalent to a collection of `Series`, which all share the same `Index`.

In fact, we can initialize a `DataFrame` by merging two or more `Series`. Consider the `Series` `s_a` and `s_b`.

```
# Notice how our indices, or row labels, are the same

s_a = pd.Series(["a1", "a2", "a3"], index = ["r1", "r2", "r3"])
s_b = pd.Series(["b1", "b2", "b3"], index = ["r1", "r2", "r3"])
```

We can turn individual `Series` into a `DataFrame` using two common methods (shown below):

```
pd.DataFrame(s_a)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	0
r1	a1
r2	a2
r3	a3

```
s_b.to_frame()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	0
r1	b1
r2	b2
r3	b3

To merge the two `Series` and specify their column names, we use the following syntax:

```python
pd.DataFrame({
    "A-column": s_a,
    "B-column": s_b
})
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	A-column	B-column
r1	a1	b1
r2	a2	b2
r3	a3	b3

2.2.3 Indices

On a more technical note, an index doesn't have to be an integer, nor does it have to be unique. For example, we can set the index of the `elections` `DataFrame` to be the name of presidential candidates.

```python
# Creating a DataFrame from a CSV file and specifying the index column
elections = pd.read_csv("data/elections.csv", index_col = "Candidate")
elections
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

Candidate	Year	Party	Popular vote	Result	%
Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
Andrew Jackson	1828	Democratic	642806	win	56.203927
John Quincy Adams	1828	National Republican	500897	loss	43.796073
Andrew Jackson	1832	Democratic	702735	win	54.574789
Henry Clay	1832	National Republican	484205	loss	37.603628
William Wirt	1832	Anti-Masonic	100715	loss	7.821583
Hugh Lawson White	1836	Whig	146109	loss	10.005985
Martin Van Buren	1836	Democratic	763291	win	52.272472
William Henry Harrison	1836	Whig	550816	loss	37.721543
Martin Van Buren	1840	Democratic	1128854	loss	46.948787
William Henry Harrison	1840	Whig	1275583	win	53.051213
Henry Clay	1844	Whig	1300004	loss	49.250523
James Polk	1844	Democratic	1339570	win	50.749477
Lewis Cass	1848	Democratic	1223460	loss	42.552229
Martin Van Buren	1848	Free Soil	291501	loss	10.138474
Zachary Taylor	1848	Whig	1360235	win	47.309296
Franklin Pierce	1852	Democratic	1605943	win	51.013168
John P. Hale	1852	Free Soil	155210	loss	4.930283
Winfield Scott	1852	Whig	1386942	loss	44.056548
James Buchanan	1856	Democratic	1835140	win	45.306080
John C. Frémont	1856	Republican	1342345	loss	33.139919
Millard Fillmore	1856	American	873053	loss	21.554001
Abraham Lincoln	1860	Republican	1855993	win	39.699408
John Bell	1860	Constitutional Union	590901	loss	12.639283
John C. Breckinridge	1860	Southern Democratic	848019	loss	18.138998
Stephen A. Douglas	1860	Northern Democratic	1380202	loss	29.522311
Abraham Lincoln	1864	National Union	2211317	win	54.951512
George B. McClellan	1864	Democratic	1812807	loss	45.048488
Horatio Seymour	1868	Democratic	2708744	loss	47.334695
Ulysses Grant	1868	Republican	3013790	win	52.665305
Horace Greeley	1872	Liberal Republican	2834761	loss	44.071406
Ulysses Grant	1872	Republican	3597439	win	55.928594
Rutherford Hayes	1876	Republican	4034142	win	48.471624
Samuel J. Tilden	1876	Democratic	4288546	loss	51.528376
James B. Weaver	1880	Greenback	308649	loss	3.352344
James Garfield	1880	Republican	4453337	win	48.369234
Winfield Scott Hancock	1880	Democratic	4444976	loss	48.278422
Benjamin Butler	1884	Anti-Monopoly	134294	loss	1.335838
Grover Cleveland	1884	Democratic	4914482	win	48.884933
James G. Blaine	1884	Republican	4856905	loss	48.312208
John St. John	1884	Prohibition	147482	loss	1.467021
Alson Streeter	1888	Union Labor	146602	loss	1.288861
Benjamin Harrison	1888	Republican	5443633	win	47.858041
Clinton B. Fisk	1888	Prohibition	249819	loss	2.196299
Grover Cleveland	1888	Democratic	5534488	loss	48.656799
Benjamin Harrison	1892	Republican	5176108	loss	42.984101
Grover Cleveland	1892	Democratic	5553898	win	46.121393
James B. Weaver	1892	Populist	1041028	loss	8.645038
John Bidwell	1892	Prohibition	270879	loss	2.249468
John M. Palmer	1896	National Democratic	134645	loss	0.969566

We can also select a new column and set it as the index of the `DataFrame`. For example, we can set the index of the `elections` `DataFrame` to represent the candidate's party.

```
elections.reset_index(inplace = True) # Resetting the index so we can set it again
# This sets the index to the "Party" column
elections.set_index("Party")
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

Party	Candidate	Year	Popular vote	Result	%
Democratic-Republican	Andrew Jackson	1824	151271	loss	57.210122
Democratic-Republican	John Quincy Adams	1824	113142	win	42.789878
Democratic	Andrew Jackson	1828	642806	win	56.203927
National Republican	John Quincy Adams	1828	500897	loss	43.796073
Democratic	Andrew Jackson	1832	702735	win	54.574789
National Republican	Henry Clay	1832	484205	loss	37.603628
Anti-Masonic	William Wirt	1832	100715	loss	7.821583
Whig	Hugh Lawson White	1836	146109	loss	10.005985
Democratic	Martin Van Buren	1836	763291	win	52.272472
Whig	William Henry Harrison	1836	550816	loss	37.721543
Democratic	Martin Van Buren	1840	1128854	loss	46.948787
Whig	William Henry Harrison	1840	1275583	win	53.051213
Whig	Henry Clay	1844	1300004	loss	49.250523
Democratic	James Polk	1844	1339570	win	50.749477
Democratic	Lewis Cass	1848	1223460	loss	42.552229
Free Soil	Martin Van Buren	1848	291501	loss	10.138474
Whig	Zachary Taylor	1848	1360235	win	47.309296
Democratic	Franklin Pierce	1852	1605943	win	51.013168
Free Soil	John P. Hale	1852	155210	loss	4.930283
Whig	Winfield Scott	1852	1386942	loss	44.056548
Democratic	James Buchanan	1856	1835140	win	45.306080
Republican	John C. Frémont	1856	1342345	loss	33.139919
American	Millard Fillmore	1856	873053	loss	21.554001
Republican	Abraham Lincoln	1860	1855993	win	39.699408
Constitutional Union	John Bell	1860	590901	loss	12.639283
Southern Democratic	John C. Breckinridge	1860	848019	loss	18.138998
Northern Democratic	Stephen A. Douglas	1860	1380202	loss	29.522311
National Union	Abraham Lincoln	1864	2211317	win	54.951512
Democratic	George B. McClellan	1864	1812807	loss	45.048488
Democratic	Horatio Seymour	1868	2708744	loss	47.334695
Republican	Ulysses Grant	1868	3013790	win	52.665305
Liberal Republican	Horace Greeley	1872	2834761	loss	44.071406
Republican	Ulysses Grant	1872	3597439	win	55.928594
Republican	Rutherford Hayes	1876	4034142	win	48.471624
Democratic	Samuel J. Tilden	1876	4288546	loss	51.528376
Greenback	James B. Weaver	1880	308649	loss	3.352344
Republican	James Garfield	1880	4453337	win	48.369234
Democratic	Winfield Scott Hancock	1880	4444976	loss	48.278422
Anti-Monopoly	Benjamin Butler	1884	134294	loss	1.335838
Democratic	Grover Cleveland	1884	4914482	win	48.884933
Republican	James G. Blaine	1884	4856905	loss	48.312208
Prohibition	John St. John	1884	147482	loss	1.467021
Union Labor	Alson Streeter	1888	146602	loss	1.288861
Republican	Benjamin Harrison	1888	5443633	win	47.858041
Prohibition	Clinton B. Fisk	1888	249819	loss	2.196299
Democratic	Grover Cleveland	1888	5534488	loss	48.656799
Republican	Benjamin Harrison	1892	5176108	loss	42.984101
Democratic	Grover Cleveland	1892	5553898	win	46.121393
Populist	James B. Weaver	1892	1041028	loss	8.645038
Prohibition	John Bidwell	1892	270879	loss	2.249468
National Democratic	John M. Palmer	1896	134645	loss	0.969566

And, if we'd like, we can revert the index back to the default list of integers.

```
# This resets the index to be the default list of integer
elections.reset_index(inplace=True)
elections.index
```

```
RangeIndex(start=0, stop=182, step=1)
```

It is also important to note that the row labels that constitute an index don't have to be unique. While index values can be unique and numeric, acting as a row number, they can also be named and non-unique.

Here we see unique and numeric index values.

However, here the index values are not unique.

2.3 `DataFrame` Attributes: Index, Columns, and Shape

On the other hand, column names in a `DataFrame` are almost always unique. Looking back to the `elections` dataset, it wouldn't make sense to have two columns named `"Candidate"`. Sometimes, you'll want to extract these different values, in particular, the list of row and column labels.

For index/row labels, use `DataFrame.index`:

```
elections.set_index("Party", inplace = True)
elections.index
```

```
Index(['Democratic-Republican', 'Democratic-Republican', 'Democratic',
       'National Republican', 'Democratic', 'National Republican',
       'Anti-Masonic', 'Whig', 'Democratic', 'Whig',
       ...
       'Constitution', 'Republican', 'Independent', 'Libertarian',
       'Democratic', 'Green', 'Democratic', 'Republican', 'Libertarian',
       'Green'],
      dtype='object', name='Party', length=182)
```

For column labels, use `DataFrame.columns`:

```
elections.columns
```

```
Index(['index', 'Candidate', 'Year', 'Popular vote', 'Result', '%'], dtype='object')
```

And for the shape of the `DataFrame`, we can use `DataFrame.shape` to get the number of rows followed by the number of columns:

```
elections.shape
```

```
(182, 6)
```

2.4 Slicing in DataFrames

Now that we've learned more about `DataFrame`s, let's dive deeper into their capabilities.

The API (Application Programming Interface) for the `DataFrame` class is enormous. In this section, we'll discuss several methods of the `DataFrame` API that allow us to extract subsets of data.

The simplest way to manipulate a `DataFrame` is to extract a subset of rows and columns, known as **slicing**.

Common ways we may want to extract data are grabbing:

- The first or last `n` rows in the `DataFrame`.
- Data with a certain label.
- Data at a certain position.

We will do so with four primary methods of the `DataFrame` class:

1. `.head` and `.tail`
2. `.loc`
3. `.iloc`
4. `[]`

2.4.1 Extracting data with .head and .tail

The simplest scenario in which we want to extract data is when we simply want to select the first or last few rows of the `DataFrame`.

To extract the first `n` rows of a `DataFrame` `df`, we use the syntax `df.head(n)`.

```
elections = pd.read_csv("data/elections.csv")
```

```python
# Extract the first 5 rows of the DataFrame
elections.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789

Similarly, calling `df.tail(n)` allows us to extract the last `n` rows of the `DataFrame`.

```python
# Extract the last 5 rows of the DataFrame
elections.tail(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Year	Candidate	Party	Popular vote	Result	%
177	2016	Jill Stein	Green	1457226	loss	1.073699
178	2020	Joseph Biden	Democratic	81268924	win	51.311515
179	2020	Donald Trump	Republican	74216154	loss	46.858542
180	2020	Jo Jorgensen	Libertarian	1865724	loss	1.177979
181	2020	Howard Hawkins	Green	405035	loss	0.255731

2.4.2 Label-based Extraction: Indexing with `.loc`

For the more complex task of extracting data with specific column or index labels, we can use `.loc`. The `.loc` accessor allows us to specify the ***labels*** of rows and columns we wish to extract. The **labels** (commonly referred to as the **indices**) are the bold text on the far *left* of a `DataFrame`, while the **column labels** are the column names found at the *top* of a `DataFrame`.

To grab data with `.loc`, we must specify the row and column label(s) where the data exists. The row labels are the first argument to the `.loc` function; the column labels are the second.

Arguments to `.loc` can be:

- A single value.
- A slice.
- A list.

For example, to select a single value, we can select the row labeled `0` and the column labeled `Candidate` from the `elections` `DataFrame`.

```
elections.loc[0, 'Candidate']
```

```
'Andrew Jackson'
```

Keep in mind that passing in just one argument as a single value will produce a `Series`. Below, we've extracted a subset of the `"Popular vote"` column as a `Series`.

```
elections.loc[[87, 25, 179], "Popular vote"]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Popular vote
87	15761254
25	848019
179	74216154

To select *multiple* rows and columns, we can use Python slice notation. Here, we select the rows from labels `0` to `3` and the columns from labels `"Year"` to `"Popular vote"`. Notice that unlike Python slicing, `.loc` is *inclusive* of the right upper bound.

```
elections.loc[0:3, 'Year':'Popular vote']
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Suppose that instead, we want to extract *all* column values for the first four rows in the `elections` `DataFrame`. The shorthand `:` is useful for this.

```
elections.loc[0:3, :]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073

We can use the same shorthand to extract all rows.

```
elections.loc[:, ["Year", "Candidate", "Result"]]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Result
0	1824	Andrew Jackson	loss
1	1824	John Quincy Adams	win
2	1828	Andrew Jackson	win
3	1828	John Quincy Adams	loss
4	1832	Andrew Jackson	win
5	1832	Henry Clay	loss
6	1832	William Wirt	loss
7	1836	Hugh Lawson White	loss
8	1836	Martin Van Buren	win
9	1836	William Henry Harrison	loss
10	1840	Martin Van Buren	loss
11	1840	William Henry Harrison	win
12	1844	Henry Clay	loss
13	1844	James Polk	win
14	1848	Lewis Cass	loss
15	1848	Martin Van Buren	loss
16	1848	Zachary Taylor	win
17	1852	Franklin Pierce	win
18	1852	John P. Hale	loss
19	1852	Winfield Scott	loss
20	1856	James Buchanan	win
21	1856	John C. Frémont	loss
22	1856	Millard Fillmore	loss
23	1860	Abraham Lincoln	win
24	1860	John Bell	loss
25	1860	John C. Breckinridge	loss
26	1860	Stephen A. Douglas	loss
27	1864	Abraham Lincoln	win
28	1864	George B. McClellan	loss
29	1868	Horatio Seymour	loss
30	1868	Ulysses Grant	win
31	1872	Horace Greeley	loss
32	1872	Ulysses Grant	win
33	1876	Rutherford Hayes	win
34	1876	Samuel J. Tilden	loss
35	1880	James B. Weaver	loss
36	1880	James Garfield	win
37	1880	Winfield Scott Hancock	loss
38	1884	Benjamin Butler	loss
39	1884	Grover Cleveland	win
40	1884	James G. Blaine	loss
41	1884	John St. John	loss
42	1888	Alson Streeter	loss
43	1888	Benjamin Harrison	win
44	1888	Clinton B. Fisk	loss
45	1888	Grover Cleveland	loss
46	1892	Benjamin Harrison	loss
47	1892	Grover Cleveland	win
48	1892	James B. Weaver	loss
49	1892	John Bidwell	loss
50	1896	John M. Palmer	loss
51	1896	Joshua Levering	loss

There are a couple of things we should note. Firstly, unlike conventional Python, `pandas` allows us to slice string values (in our example, the column labels). Secondly, slicing with `.loc` is *inclusive*. Notice how our resulting `DataFrame` includes every row and column between and including the slice labels we specified.

Equivalently, we can use a list to obtain multiple rows and columns in our `elections` `DataFrame`.

```
elections.loc[[0, 1, 2, 3], ['Year', 'Candidate', 'Party', 'Popular vote']]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Lastly, we can interchange list and slicing notation.

```
elections.loc[[0, 1, 2, 3], :]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073

2.4.3 Integer-based Extraction: Indexing with `.iloc`

Slicing with `.iloc` works similarly to `.loc`. However, `.iloc` uses the *index positions* of rows and columns rather than the labels (think to yourself: **l**oc uses **l**ables; **i**loc uses **i**ndices). The arguments to the `.iloc` function also behave similarly — single values, lists, indices, and any combination of these are permitted.

Let's begin reproducing our results from above. We'll begin by selecting the first presidential candidate in our `elections` `DataFrame`:

```
# elections.loc[0, "Candidate"] - Previous approach
elections.iloc[0, 1]
```

```
'Andrew Jackson'
```

Notice how the first argument to both `.loc` and `.iloc` are the same. This is because the row with a label of `0` is conveniently in the 0^{th} (equivalently, the first position) of the `elections` `DataFrame`. Generally, this is true of any `DataFrame` where the row labels are incremented in ascending order from 0.

And, as before, if we were to pass in only one single value argument, our result would be a `Series`.

```
elections.iloc[[1,2,3],1]
```

	Candidate
1	John Quincy Adams
2	Andrew Jackson
3	John Quincy Adams

However, when we select the first four rows and columns using `.iloc`, we notice something.

```
# elections.loc[0:3, 'Year':'Popular vote'] - Previous approach
elections.iloc[0:4, 0:4]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Slicing is no longer inclusive in `.iloc` — it's *exclusive*. In other words, the right end of a slice is not included when using `.iloc`. This is one of the subtleties of `pandas` syntax; you will get used to it with practice.

List behavior works just as expected.

```
#elections.loc[[0, 1, 2, 3], ['Year', 'Candidate', 'Party', 'Popular vote']] - Previous Ap
elections.iloc[[0, 1, 2, 3], [0, 1, 2, 3]]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

And just like with `.loc`, we can use a colon with `.iloc` to extract all rows or columns.

```
elections.iloc[:, 0:3]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party
0	1824	Andrew Jackson	Democratic-Republican
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican
4	1832	Andrew Jackson	Democratic
5	1832	Henry Clay	National Republican
6	1832	William Wirt	Anti-Masonic
7	1836	Hugh Lawson White	Whig
8	1836	Martin Van Buren	Democratic
9	1836	William Henry Harrison	Whig
10	1840	Martin Van Buren	Democratic
11	1840	William Henry Harrison	Whig
12	1844	Henry Clay	Whig
13	1844	James Polk	Democratic
14	1848	Lewis Cass	Democratic
15	1848	Martin Van Buren	Free Soil
16	1848	Zachary Taylor	Whig
17	1852	Franklin Pierce	Democratic
18	1852	John P. Hale	Free Soil
19	1852	Winfield Scott	Whig
20	1856	James Buchanan	Democratic
21	1856	John C. Frémont	Republican
22	1856	Millard Fillmore	American
23	1860	Abraham Lincoln	Republican
24	1860	John Bell	Constitutional Union
25	1860	John C. Breckinridge	Southern Democratic
26	1860	Stephen A. Douglas	Northern Democratic
27	1864	Abraham Lincoln	National Union
28	1864	George B. McClellan	Democratic
29	1868	Horatio Seymour	Democratic
30	1868	Ulysses Grant	Republican
31	1872	Horace Greeley	Liberal Republican
32	1872	Ulysses Grant	Republican
33	1876	Rutherford Hayes	Republican
34	1876	Samuel J. Tilden	Democratic
35	1880	James B. Weaver	Greenback
36	1880	James Garfield	Republican
37	1880	Winfield Scott Hancock	Democratic
38	1884	Benjamin Butler	Anti-Monopoly
39	1884	Grover Cleveland	Democratic
40	1884	James G. Blaine	Republican
41	1884	John St. John	Prohibition
42	1888	Alson Streeter	Union Labor
43	1888	Benjamin Harrison	Republican
44	1888	Clinton B. Fisk	Prohibition
45	1888	Grover Cleveland	Democratic
46	1892	Benjamin Harrison	Republican
47	1892	Grover Cleveland	Democratic
48	1892	James B. Weaver	Populist
49	1892	John Bidwell	Prohibition
50	1896	John M. Palmer	National Democratic
51	1896	Joshua Levering	Prohibition

This discussion begs the question: when should we use `.loc` vs. `.iloc`? In most cases, `.loc` is generally safer to use. You can imagine `.iloc` may return incorrect values when applied to a dataset where the ordering of data can change. However, `.iloc` can still be useful — for example, if you are looking at a `DataFrame` of sorted movie earnings and want to get the median earnings for a given year, you can use `.iloc` to index into the middle.

Overall, it is important to remember that:

- `.loc` performances **l**abel-based extraction.
- `.iloc` performs **i**nteger-based extraction.

2.4.4 Context-dependent Extraction: Indexing with `[]`

The `[]` selection operator is the most baffling of all, yet the most commonly used. It only takes a single argument, which may be one of the following:

1. A slice of row numbers.
2. A list of column labels.
3. A single-column label.

That is, `[]` is *context-dependent*. Let's see some examples.

2.4.4.1 A slice of row numbers

Say we wanted the first four rows of our `elections` `DataFrame`.

```
elections[0:4]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073

2.4.4.2 A list of column labels

Suppose we now want the first four columns.

```
elections[["Year", "Candidate", "Party", "Popular vote"]]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897
4	1832	Andrew Jackson	Democratic	702735
5	1832	Henry Clay	National Republican	484205
6	1832	William Wirt	Anti-Masonic	100715
7	1836	Hugh Lawson White	Whig	146109
8	1836	Martin Van Buren	Democratic	763291
9	1836	William Henry Harrison	Whig	550816
10	1840	Martin Van Buren	Democratic	1128854
11	1840	William Henry Harrison	Whig	1275583
12	1844	Henry Clay	Whig	1300004
13	1844	James Polk	Democratic	1339570
14	1848	Lewis Cass	Democratic	1223460
15	1848	Martin Van Buren	Free Soil	291501
16	1848	Zachary Taylor	Whig	1360235
17	1852	Franklin Pierce	Democratic	1605943
18	1852	John P. Hale	Free Soil	155210
19	1852	Winfield Scott	Whig	1386942
20	1856	James Buchanan	Democratic	1835140
21	1856	John C. Frémont	Republican	1342345
22	1856	Millard Fillmore	American	873053
23	1860	Abraham Lincoln	Republican	1855993
24	1860	John Bell	Constitutional Union	590901
25	1860	John C. Breckinridge	Southern Democratic	848019
26	1860	Stephen A. Douglas	Northern Democratic	1380202
27	1864	Abraham Lincoln	National Union	2211317
28	1864	George B. McClellan	Democratic	1812807
29	1868	Horatio Seymour	Democratic	2708744
30	1868	Ulysses Grant	Republican	3013790
31	1872	Horace Greeley	Liberal Republican	2834761
32	1872	Ulysses Grant	Republican	3597439
33	1876	Rutherford Hayes	Republican	4034142
34	1876	Samuel J. Tilden	Democratic	4288546
35	1880	James B. Weaver	Greenback	308649
36	1880	James Garfield	Republican	4453337
37	1880	Winfield Scott Hancock	Democratic	4444976
38	1884	Benjamin Butler	Anti-Monopoly	134294
39	1884	Grover Cleveland	Democratic	4914482
40	1884	James G. Blaine	Republican	4856905
41	1884	John St. John	Prohibition	147482
42	1888	Alson Streeter	Union Labor	146602
43	1888	Benjamin Harrison	Republican	5443633
44	1888	Clinton B. Fisk	Prohibition	249819
45	1888	Grover Cleveland	Democratic	5534488
46	1892	Benjamin Harrison	Republican	5176108
47	1892	Grover Cleveland	Democratic	5553898
48	1892	James B. Weaver	Populist	1041028
49	1892	John Bidwell	Prohibition	270879
50	1896	John M. Palmer	National Democratic	134645
51	1896	Joshua Levering	Prohibition	131312

2.4.4.3 A single-column label

Lastly, `[]` allows us to extract only the `"Candidate"` column.

```
elections["Candidate"]
```

	Candidate
0	Andrew Jackson
1	John Quincy Adams
2	Andrew Jackson
3	John Quincy Adams
4	Andrew Jackson
5	Henry Clay
6	William Wirt
7	Hugh Lawson White
8	Martin Van Buren
9	William Henry Harrison
10	Martin Van Buren
11	William Henry Harrison
12	Henry Clay
13	James Polk
14	Lewis Cass
15	Martin Van Buren
16	Zachary Taylor
17	Franklin Pierce
18	John P. Hale
19	Winfield Scott
20	James Buchanan
21	John C. Frémont
22	Millard Fillmore
23	Abraham Lincoln
24	John Bell
25	John C. Breckinridge
26	Stephen A. Douglas
27	Abraham Lincoln
28	George B. McClellan
29	Horatio Seymour
30	Ulysses Grant
31	Horace Greeley
32	Ulysses Grant
33	Rutherford Hayes
34	Samuel J. Tilden
35	James B. Weaver
36	James Garfield
37	Winfield Scott Hancock
38	Benjamin Butler
39	Grover Cleveland
40	James G. Blaine
41	John St. John
42	Alson Streeter
43	Benjamin Harrison
44	Clinton B. Fisk
45	Grover Cleveland
46	Benjamin Harrison
47	Grover Cleveland
48	James B. Weaver
49	John Bidwell
50	John M. Palmer
51	Joshua Levering

The output is a `Series`! In this course, we'll become very comfortable with `[]`, especially for selecting columns. In practice, `[]` is much more common than `.loc`, especially since it is far more concise.

2.5 Parting Note

The `pandas` library is enormous and contains many useful functions. Here is a link to its documentation. We certainly don't expect you to memorize each and every method of the library, and we will give you a reference sheet for exams.

The introductory Data 100 `pandas` lectures will provide a high-level view of the key data structures and methods that will form the foundation of your `pandas` knowledge. A goal of this course is to help you build your familiarity with the real-world programming practice of … Googling! Answers to your questions can be found in documentation, Stack Overflow, etc. Being able to search for, read, and implement documentation is an important life skill for any data scientist.

With that, we will move on to Pandas II!

3 Pandas II

> **Learning Outcomes**
>
> - Continue building familiarity with `pandas` syntax.
> - Extract data from a `DataFrame` using conditional selection.
> - Recognize situations where aggregation is useful and identify the correct technique for performing an aggregation.

Last time, we introduced the `pandas` library as a toolkit for processing data. We learned the `DataFrame` and `Series` data structures, familiarized ourselves with the basic syntax for manipulating tabular data, and began writing our first lines of `pandas` code.

In this lecture, we'll start to dive into some advanced `pandas` syntax. You may find it helpful to follow along with a notebook of your own as we walk through these new pieces of code.

We'll start by loading the `babynames` dataset.

```
# This code pulls census data and loads it into a DataFrame
# We won't cover it explicitly in this class, but you are welcome to explore it on your ow
import pandas as pd
import numpy as np
import urllib.request
import os.path
import zipfile

data_url = "https://www.ssa.gov/oact/babynames/state/namesbystate.zip"
local_filename = "data/babynamesbystate.zip"
if not os.path.exists(local_filename): # If the data exists don't download again
    with urllib.request.urlopen(data_url) as resp, open(local_filename, 'wb') as f:
        f.write(resp.read())

zf = zipfile.ZipFile(local_filename, 'r')

ca_name = 'STATE.CA.TXT'
field_names = ['State', 'Sex', 'Year', 'Name', 'Count']
with zf.open(ca_name) as fh:
```

```
    babynames = pd.read_csv(fh, header=None, names=field_names)

babynames.head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

3.1 Conditional Selection

Conditional selection allows us to select a subset of rows in a `DataFrame` that satisfy some specified condition.

To understand how to use conditional selection, we must look at another possible input of the `.loc` and `[]` methods – a boolean array, which is simply an array or `Series` where each element is either `True` or `False`. This boolean array must have a length equal to the number of rows in the `DataFrame`. It will return all rows that correspond to a value of `True` in the array. We used a very similar technique when performing conditional extraction from a `Series` in the last lecture.

To see this in action, let's select all even-indexed rows in the first 10 rows of our `DataFrame`.

```
# Ask yourself: why is :9 is the correct slice to select the first 10 rows?
babynames_first_10_rows = babynames.loc[:9, :]

# Notice how we have exactly 10 elements in our boolean array argument
babynames_first_10_rows[[True, False, True, False, True, False, True, False, True, False]]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
2	CA	F	1910	Dorothy	220
4	CA	F	1910	Frances	134
6	CA	F	1910	Evelyn	126
8	CA	F	1910	Virginia	101

We can perform a similar operation using `.loc`.

```
babynames_first_10_rows.loc[[True, False, True, False, True, False, True, False, True, Fal
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
2	CA	F	1910	Dorothy	220
4	CA	F	1910	Frances	134
6	CA	F	1910	Evelyn	126
8	CA	F	1910	Virginia	101

These techniques worked well in this example, but you can imagine how tedious it might be to list out `True` and `False`for every row in a larger `DataFrame`. To make things easier, we can instead provide a logical condition as an input to `.loc` or `[]` that returns a boolean array with the necessary length.

For example, to return all names associated with `F` sex:

```
# First, use a logical condition to generate a boolean array
logical_operator = (babynames["Sex"] == "F")

# Then, use this boolean array to filter the DataFrame
babynames[logical_operator].head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Recall from the previous lecture that `.head()` will return only the first few rows in the `DataFrame`. In reality, `babynames[logical operator]` contains as many rows as there are entries in the original `babynames DataFrame` with sex `"F"`.

Here, `logical_operator` evaluates to a `Series` of boolean values with length 407428.

```
print("There are a total of {} values in 'logical_operator'".format(len(logical_operator))
```

```
There are a total of 407428 values in 'logical_operator'
```

Rows starting at row 0 and ending at row 239536 evaluate to `True` and are thus returned in the `DataFrame`. Rows from 239537 onwards evaluate to `False` and are omitted from the output.

```
print("The 0th item in this 'logical_operator' is: {}".format(logical_operator.iloc[0]))
print("The 239536th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2
print("The 239537th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2
```

```
The 0th item in this 'logical_operator' is: True
The 239536th item in this 'logical_operator' is: True
The 239537th item in this 'logical_operator' is: False
```

Passing a `Series` as an argument to `babynames[]` has the same effect as using a boolean array. In fact, the `[]` selection operator can take a boolean `Series`, array, and list as arguments. These three are used interchangeably throughout the course.

We can also use `.loc` to achieve similar results.

```
babynames.loc[babynames["Sex"] == "F"].head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Boolean conditions can be combined using various bitwise operators, allowing us to filter results by multiple conditions. In the table below, p and q are boolean arrays or `Series`.

Symbol	Usage	Meaning
~	~p	Returns negation of p
|	p | q	p OR q
&	p & q	p AND q
^	p ^ q	p XOR q (exclusive or)

When combining multiple conditions with logical operators, we surround each individual condition with a set of parenthesis `()`. This imposes an order of operations on `pandas` evaluating your logic and can avoid code erroring.

For example, if we want to return data on all names with sex `"F"` born before the year 2000, we can write:

```
babynames[(babynames["Sex"] == "F") & (babynames["Year"] < 2000)].head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Note that we're working with `Series`, so using `and` in place of `&`, or `or` in place `|` will error.

```
# This line of code will raise a ValueError
# babynames[(babynames["Sex"] == "F") and (babynames["Year"] < 2000)].head()
```

If we want to return data on all names with sex "F" *or* all born before the year 2000, we can write:

```
babynames[(babynames["Sex"] == "F") | (babynames["Year"] < 2000)].head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Boolean array selection is a useful tool, but can lead to overly verbose code for complex conditions. In the example below, our boolean condition is long enough to extend for several lines of code.

```
# Note: The parentheses surrounding the code make it possible to break the code on to mult
(
    babynames[(babynames["Name"] == "Bella") |
              (babynames["Name"] == "Alex") |
              (babynames["Name"] == "Ani") |
              (babynames["Name"] == "Lisa")]
).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
6289	CA	F	1923	Bella	5
7512	CA	F	1925	Bella	8
12368	CA	F	1932	Lisa	5
14741	CA	F	1936	Lisa	8
17084	CA	F	1939	Lisa	5

Fortunately, `pandas` provides many alternative methods for constructing boolean filters.

The `.isin` function is one such example. This method evaluates if the values in a `Series` are contained in a different sequence (list, array, or `Series`) of values. In the cell below, we achieve equivalent results to the `DataFrame` above with far more concise code.

```
names = ["Bella", "Alex", "Narges", "Lisa"]
babynames["Name"].isin(names).head()
```

	Name
0	False
1	False
2	False
3	False
4	False

```
babynames[babynames["Name"].isin(names)].head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
6289	CA	F	1923	Bella	5
7512	CA	F	1925	Bella	8
12368	CA	F	1932	Lisa	5
14741	CA	F	1936	Lisa	8
17084	CA	F	1939	Lisa	5

The function `str.startswith` can be used to define a filter based on string values in a `Series` object. It checks to see if string values in a `Series` start with a particular character.

```
# Identify whether names begin with the letter "N"
babynames["Name"].str.startswith("N").head()
```

	Name
0	False
1	False
2	False
3	False
4	False

```
# Extracting names that begin with the letter "N"
babynames[babynames["Name"].str.startswith("N")].head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
76	CA	F	1910	Norma	23
83	CA	F	1910	Nellie	20
127	CA	F	1910	Nina	11
198	CA	F	1910	Nora	6
310	CA	F	1911	Nellie	23

3.2 Adding, Removing, and Modifying Columns

In many data science tasks, we may need to change the columns contained in our `DataFrame` in some way. Fortunately, the syntax to do so is fairly straightforward.

To add a new column to a `DataFrame`, we use a syntax similar to that used when accessing an existing column. Specify the name of the new column by writing `df["column"]`, then assign this to a `Series` or array containing the values that will populate this column.

```
# Create a Series of the length of each name.
babyname_lengths = babynames["Name"].str.len()

# Add a column named "name_lengths" that includes the length of each name
babynames["name_lengths"] = babyname_lengths
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	4
1	CA	F	1910	Helen	239	5
2	CA	F	1910	Dorothy	220	7
3	CA	F	1910	Margaret	163	8
4	CA	F	1910	Frances	134	7

If we need to later modify an existing column, we can do so by referencing this column again with the syntax `df["column"]`, then re-assigning it to a new `Series` or array of the appropriate length.

```
# Modify the "name_lengths" column to be one less than its original value
babynames["name_lengths"] = babynames["name_lengths"] - 1
babynames.head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	3
1	CA	F	1910	Helen	239	4
2	CA	F	1910	Dorothy	220	6
3	CA	F	1910	Margaret	163	7
4	CA	F	1910	Frances	134	6

We can rename a column using the `.rename()` method. It takes in a dictionary that maps old column names to their new ones.

```
# Rename "name_lengths" to "Length"
babynames = babynames.rename(columns={"name_lengths":"Length"})
babynames.head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	State	Sex	Year	Name	Count	Length
0	CA	F	1910	Mary	295	3
1	CA	F	1910	Helen	239	4
2	CA	F	1910	Dorothy	220	6
3	CA	F	1910	Margaret	163	7
4	CA	F	1910	Frances	134	6

If we want to remove a column or row of a `DataFrame`, we can call the `.drop` (documentation) method. Use the `axis` parameter to specify whether a column or row should be dropped. Unless otherwise specified, `pandas` will assume that we are dropping a row by default.

```
# Drop our new "Length" column from the DataFrame
babynames = babynames.drop("Length", axis="columns")
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

Notice that we *re-assigned* `babynames` to the result of `babynames.drop(...)`. This is a subtle but important point: `pandas` table operations **do not occur in-place**. Calling `df.drop(...)` will output a *copy* of `df` with the row/column of interest removed without modifying the original `df` table.

In other words, if we simply call:

```
# This creates a copy of `babynames` and removes the column "Name"...
babynames.drop("Name", axis="columns")

# ...but the original `babynames` is unchanged!
# Notice that the "Name" column is still present
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134

3.3 Useful Utility Functions

`pandas` contains an extensive library of functions that can help shorten the process of setting and getting information from its data structures. In the following section, we will give overviews of each of the main utility functions that will help us in Data 100.

Discussing all functionality offered by `pandas` could take an entire semester! We will walk you through the most commonly-used functions and encourage you to explore and experiment on your own.

- `NumPy` and built-in function support
- `.shape`
- `.size`
- `.describe()`
- `.sample()`
- `.value_counts()`
- `.unique()`
- `.sort_values()`

The `pandas` documentation will be a valuable resource in Data 100 and beyond.

3.3.1 `NumPy`

`pandas` is designed to work well with `NumPy`, the framework for array computations you encountered in Data 8. Just about any `NumPy` function can be applied to `pandas` `DataFrames` and `Series`.

```
# Pull out the number of babies named Yash each year
yash_count = babynames[babynames["Name"] == "Yash"]["Count"]
```

```python
yash_count.head()
```

	Count
331824	8
334114	9
336390	11
338773	12
341387	10

```python
# Average number of babies named Yash each year
np.mean(yash_count)
```

```
17.142857142857142
```

```python
# Max number of babies named Yash born in any one year
np.max(yash_count)
```

```
29
```

3.3.2 `.shape` and `.size`

`.shape` and `.size` are attributes of `Series` and `DataFrames` that measure the "amount" of data stored in the structure. Calling `.shape` returns a tuple containing the number of rows and columns present in the `DataFrame` or `Series`. `.size` is used to find the total number of elements in a structure, equivalent to the number of rows times the number of columns.

Many functions strictly require the dimensions of the arguments along certain axes to match. Calling these dimension-finding functions is much faster than counting all of the items by hand.

```python
# Return the shape of the DataFrame, in the format (num_rows, num_columns)
babynames.shape
```

```
(407428, 5)
```

```python
# Return the size of the DataFrame, equal to num_rows * num_columns
babynames.size
```

```
2037140
```

3.3.3 .describe()

If many statistics are required from a `DataFrame` (minimum value, maximum value, mean value, etc.), then `.describe()` (documentation) can be used to compute all of them at once.

```
babynames.describe()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Count
count	407428.000000	407428.000000
mean	1985.733609	79.543456
std	27.007660	293.698654
min	1910.000000	5.000000
25%	1969.000000	7.000000
50%	1992.000000	13.000000
75%	2008.000000	38.000000
max	2022.000000	8260.000000

A different set of statistics will be reported if `.describe()` is called on a `Series`.

```
babynames["Sex"].describe()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Sex
count	407428
unique	2
top	F
freq	239537

3.3.4 `.sample()`

As we will see later in the semester, random processes are at the heart of many data science techniques (for example, train-test splits, bootstrapping, and cross-validation). `.sample()` (documentation) lets us quickly select random entries (a row if called from a `DataFrame`, or a value if called from a `Series`).

By default, `.sample()` selects entries *without* replacement. Pass in the argument `replace=True` to sample with replacement.

```
# Sample a single row
babynames.sample()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
262157	CA	M	1950	Wiley	7

Naturally, this can be chained with other methods and operators (`iloc`, etc.).

```
# Sample 5 random rows, and select all columns after column 2
babynames.sample(5).iloc[:, 2:]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Name	Count
174657	2006	Annalie	11
304329	1983	Heber	6
113901	1990	Steffanie	21
315293	1989	Allan	111
278239	1965	Llewellyn	5

```
# Randomly sample 4 names from the year 2000, with replacement, and select all columns aft
babynames[babynames["Year"] == 2000].sample(4, replace = True).iloc[:, 2:]
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Year	Name	Count
150083	2000	Drew	28
151984	2000	Breahna	6
151353	2000	Marleny	9
150228	2000	Ashlie	23

3.3.5 `.value_counts()`

The `Series.value_counts()` (documentation) method counts the number of occurrence of each unique value in a `Series`. In other words, it *counts* the number of times each unique *value* appears. This is often useful for determining the most or least common entries in a `Series`.

In the example below, we can determine the name with the most years in which at least one person has taken that name by counting the number of times each name appears in the `"Name"` column of `babynames`. Note that the return value is also a `Series`.

```
babynames["Name"].value_counts().head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Name
Jean	223
Francis	221
Guadalupe	218
Jessie	217
Marion	214

3.3.6 `.unique()`

If we have a `Series` with many repeated values, then `.unique()` (documentation) can be used to identify only the *unique* values. Here we return an array of all the names in `babynames`.

```
babynames["Name"].unique()
```

```
array(['Mary', 'Helen', 'Dorothy', ..., 'Zae', 'Zai', 'Zayvier'],
      dtype=object)
```

3.3.7 `.sort_values()`

Ordering a `DataFrame` can be useful for isolating extreme values. For example, the first 5 entries of a row sorted in descending order (that is, from highest to lowest) are the largest 5 values. `.sort_values` (documentation) allows us to order a `DataFrame` or `Series` by a specified column. We can choose to either receive the rows in `ascending` order (default) or `descending` order.

```
# Sort the "Count" column from highest to lowest
babynames.sort_values(by="Count", ascending=False).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
268041	CA	M	1957	Michael	8260
267017	CA	M	1956	Michael	8258
317387	CA	M	1990	Michael	8246
281850	CA	M	1969	Michael	8245
283146	CA	M	1970	Michael	8196

Unlike when calling `.value_counts()` on a `DataFrame`, we do not need to explicitly specify the column used for sorting when calling `.value_counts()` on a `Series`. We can still specify the ordering paradigm – that is, whether values are sorted in ascending or descending order.

```
# Sort the "Name" Series alphabetically
babynames["Name"].sort_values(ascending=True).head()
```

	Name
366001	Aadan
384005	Aadan
369120	Aadan
398211	Aadarsh
370306	Aaden

3.4 Parting Note

Manipulating `DataFrames` is not a skill that is mastered in just one day. Due to the flexibility of `pandas`, there are many different ways to get from point A to point B. We recommend trying multiple different ways to solve the same problem to gain even more practice and reach that point of mastery sooner.

Next, we will start digging deeper into the mechanics behind grouping data.

4 Pandas III

> **Learning Outcomes**
>
> - Perform advanced aggregation using `.groupby()`
> - Use the `pd.pivot_table` method to construct a pivot table
> - Perform simple merges between DataFrames using `pd.merge()`

We will introduce the concept of aggregating data – we will familiarize ourselves with `GroupBy` objects and used them as tools to consolidate and summarize a`DataFrame`. In this lecture, we will explore working with the different aggregation functions and dive into some advanced `.groupby` methods to show just how powerful of a resource they can be for understanding our data. We will also introduce other techniques for data aggregation to provide flexibility in how we manipulate our tables.

4.1 Custom Sorts

First, let's finish our discussion about sorting. Let's try to solve a sorting problem using different approaches. Assume we want to find the longest baby names and sort our data accordingly.

We'll start by loading the `babynames` dataset. Note that this dataset is filtered to only contain data from California.

```
# This code pulls census data and loads it into a DataFrame
# We won't cover it explicitly in this class, but you are welcome to explore it on your ow
import pandas as pd
import numpy as np
import urllib.request
import os.path
import zipfile

data_url = "https://www.ssa.gov/oact/babynames/state/namesbystate.zip"
local_filename = "data/babynames by state.zip"
if not os.path.exists(local_filename): # If the data exists don't download again
```

```
    with urllib.request.urlopen(data_url) as resp, open(local_filename, 'wb') as f:
        f.write(resp.read())

zf = zipfile.ZipFile(local_filename, 'r')

ca_name = 'STATE.CA.TXT'
field_names = ['State', 'Sex', 'Year', 'Name', 'Count']
with zf.open(ca_name) as fh:
    babynames = pd.read_csv(fh, header=None, names=field_names)

babynames.tail(10)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	State	Sex	Year	Name	Count
407418	CA	M	2022	Zach	5
407419	CA	M	2022	Zadkiel	5
407420	CA	M	2022	Zae	5
407421	CA	M	2022	Zai	5
407422	CA	M	2022	Zay	5
407423	CA	M	2022	Zayvier	5
407424	CA	M	2022	Zia	5
407425	CA	M	2022	Zora	5
407426	CA	M	2022	Zuriel	5
407427	CA	M	2022	Zylo	5

4.1.1 Approach 1: Create a Temporary Column

One method to do this is to first start by creating a column that contains the lengths of the names.

```
# Create a Series of the length of each name
babyname_lengths = babynames["Name"].str.len()

# Add a column named "name_lengths" that includes the length of each name
babynames["name_lengths"] = babyname_lengths
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	4
1	CA	F	1910	Helen	239	5
2	CA	F	1910	Dorothy	220	7
3	CA	F	1910	Margaret	163	8
4	CA	F	1910	Frances	134	7

We can then sort the `DataFrame` by that column using `.sort_values()`:

```python
# Sort by the temporary column
babynames = babynames.sort_values(by="name_lengths", ascending=False)
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count	name_lengths
334166	CA	M	1996	Franciscojavier	8	15
337301	CA	M	1997	Franciscojavier	5	15
339472	CA	M	1998	Franciscojavier	6	15
321792	CA	M	1991	Ryanchristopher	7	15
327358	CA	M	1993	Johnchristopher	5	15

Finally, we can drop the `name_length` column from `babynames` to prevent our table from getting cluttered.

```python
# Drop the 'name_length' column
babynames = babynames.drop("name_lengths", axis='columns')
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
334166	CA	M	1996	Franciscojavier	8
337301	CA	M	1997	Franciscojavier	5
339472	CA	M	1998	Franciscojavier	6
321792	CA	M	1991	Ryanchristopher	7
327358	CA	M	1993	Johnchristopher	5

4.1.2 Approach 2: Sorting using the `key` Argument

Another way to approach this is to use the `key` argument of `.sort_values()`. Here we can specify that we want to sort `"Name"` values by their length.

```
babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	State	Sex	Year	Name	Count
334166	CA	M	1996	Franciscojavier	8
327472	CA	M	1993	Ryanchristopher	5
337301	CA	M	1997	Franciscojavier	5
337477	CA	M	1997	Ryanchristopher	5
312543	CA	M	1987	Franciscojavier	5

4.1.3 Approach 3: Sorting using the `map` Function

We can also use the `map` function on a `Series` to solve this. Say we want to sort the `babynames` table by the number of `"dr"`'s and `"ea"`'s in each `"Name"`. We'll define the function `dr_ea_count` to help us out.

```
# First, define a function to count the number of times "dr" or "ea" appear in each name
def dr_ea_count(string):
    return string.count('dr') + string.count('ea')

# Then, use `map` to apply `dr_ea_count` to each name in the "Name" column
babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count)

# Sort the DataFrame by the new "dr_ea_count" column so we can see our handiwork
```

```python
babynames = babynames.sort_values(by="dr_ea_count", ascending=False)
babynames.head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	State	Sex	Year	Name	Count	dr_ea_count
115957	CA	F	1990	Deandrea	5	3
101976	CA	F	1986	Deandrea	6	3
131029	CA	F	1994	Leandrea	5	3
108731	CA	F	1988	Deandrea	5	3
308131	CA	M	1985	Deandrea	6	3

We can drop the `dr_ea_count` once we're done using it to maintain a neat table.

```python
# Drop the `dr_ea_count` column
babynames = babynames.drop("dr_ea_count", axis = 'columns')
babynames.head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	State	Sex	Year	Name	Count
115957	CA	F	1990	Deandrea	5
101976	CA	F	1986	Deandrea	6
131029	CA	F	1994	Leandrea	5
108731	CA	F	1988	Deandrea	5
308131	CA	M	1985	Deandrea	6

4.2 Aggregating Data with `.groupby`

Up until this point, we have been working with individual rows of `DataFrame`s. As data scientists, we often wish to investigate trends across a larger *subset* of our data. For example, we may want to compute some summary statistic (the mean, median, sum, etc.) for a group of rows in our `DataFrame`. To do this, we'll use `pandas` `GroupBy` objects. Our goal is to group together rows that fall under the same category and perform an operation that aggregates across all rows in the category.

Let's say we wanted to aggregate all rows in `babynames` for a given year.

```
babynames.groupby("Year")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x107d15040>
```

What does this strange output mean? Calling `.groupby` (documentation) has generated a `GroupBy` object. You can imagine this as a set of "mini" sub-`DataFrame`s, where each subframe contains all of the rows from `babynames` that correspond to a particular year.

The diagram below shows a simplified view of `babynames` to help illustrate this idea.

We can't work with a `GroupBy` object directly – that is why you saw that strange output earlier rather than a standard view of a `DataFrame`. To actually manipulate values within these "mini" `DataFrame`s, we'll need to call an *aggregation method*. This is a method that tells `pandas` how to aggregate the values within the `GroupBy` object. Once the aggregation is applied, `pandas` will return a normal (now grouped) `DataFrame`.

The first aggregation method we'll consider is `.agg`. The `.agg` method takes in a function as its argument; this function is then applied to each column of a "mini" grouped DataFrame. We end up with a new `DataFrame` with one aggregated row per subframe. Let's see this in action by finding the `sum` of all counts for each year in `babynames` – this is equivalent to finding the number of babies born in each year.

```
babynames[["Year", "Count"]].groupby("Year").agg(sum).head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

Year	Count
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

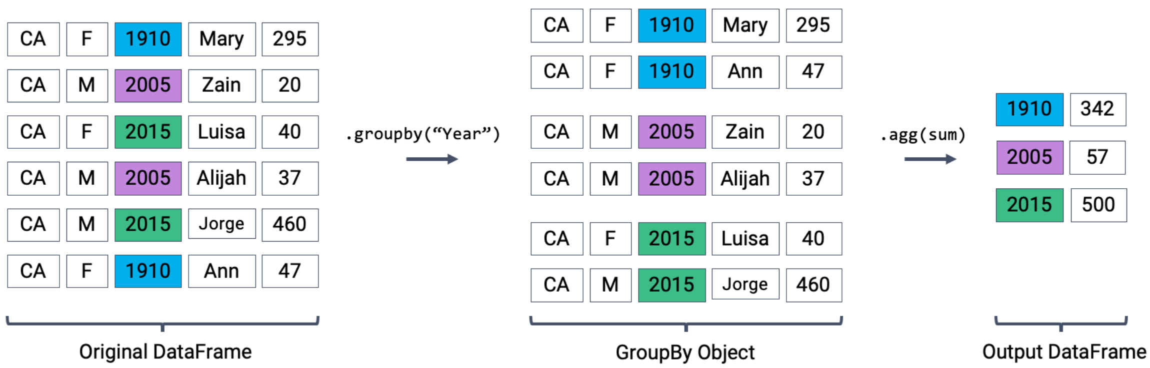

Figure 4.1: Performing an aggregation

We can relate this back to the diagram we used above. Remember that the diagram uses a simplified version of `babynames`, which is why we see smaller values for the summed counts.

Calling `.agg` has condensed each subframe back into a single row. This gives us our final output: a `DataFrame` that is now indexed by `"Year"`, with a single row for each unique year in the original `babynames` DataFrame.

There are many different aggregation functions we can use, all of which are useful in different applications.

```
babynames[["Year", "Count"]].groupby("Year").agg(min).head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Count
Year	
1910	5
1911	5
1912	5
1913	5
1914	5

```
babynames[["Year", "Count"]].groupby("Year").agg(max).head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa
```

```
In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Count
Year	
1910	295
1911	390
1912	534
1913	614
1914	773

```python
# Same result, but now we explicitly tell pandas to only consider the "Count" column when
babynames.groupby("Year")[["Count"]].agg(sum).head(5)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Count
Year	
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

There are many different aggregations that can be applied to the grouped data. The primary requirement is that an aggregation function must:

- Take in a `Series` of data (a single column of the grouped subframe).
- Return a single value that aggregates this `Series`.

4.2.1 Aggregation Functions

Because of this fairly broad requirement, `pandas` offers many ways of computing an aggregation.

In-built Python operations – such as `sum`, `max`, and `min` – are automatically recognized by `pandas`.

```
# What is the minimum count for each name in any year?
babynames.groupby("Name")[["Count"]].agg(min).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Count
Name	
Aadan	5
Aadarsh	6
Aaden	10
Aadhav	6
Aadhini	6

```
# What is the largest single-year count of each name?
babynames.groupby("Name")[["Count"]].agg(max).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	Count
Name	
Aadan	7
Aadarsh	6
Aaden	158
Aadhav	8
Aadhini	6

As mentioned previously, functions from the `NumPy` library, such as `np.mean`, `np.max`, `np.min`, and `np.sum`, are also fair game in `pandas`.

```
# What is the average count for each name across all years?
babynames.groupby("Name")[["Count"]].agg(np.mean).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

Name	Count
Aadan	6.000000
Aadarsh	6.000000
Aaden	46.214286
Aadhav	6.750000
Aadhini	6.000000

`pandas` also offers a number of in-built functions. Functions that are native to `pandas` can be referenced using their string name within a call to `.agg`. Some examples include:

- `.agg("sum")`
- `.agg("max")`
- `.agg("min")`
- `.agg("mean")`
- `.agg("first")`
- `.agg("last")`

The latter two entries in this list – `"first"` and `"last"` – are unique to `pandas`. They return the first or last entry in a subframe column. Why might this be useful? Consider a case where *multiple* columns in a group share identical information. To represent this information in the grouped output, we can simply grab the first or last entry, which we know will be identical to all other entries.

Let's illustrate this with an example. Say we add a new column to `babynames` that contains the first letter of each name.

```
# Imagine we had an additional column, "First Letter". We'll explain this code next week
babynames["First Letter"] = babynames["Name"].str[0]

# We construct a simplified DataFrame containing just a subset of columns
babynames_new = babynames[["Name", "First Letter", "Year"]]
babynames_new.head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Name	First Letter	Year
115957	Deandrea	D	1990
101976	Deandrea	D	1986
131029	Leandrea	L	1994
108731	Deandrea	D	1988
308131	Deandrea	D	1985

If we form groups for each name in the dataset, `"First Letter"` will be the same for all members of the group. This means that if we simply select the first entry for `"First Letter"` in the group, we'll represent all data in that group.

We can use a dictionary to apply different aggregation functions to each column during grouping.

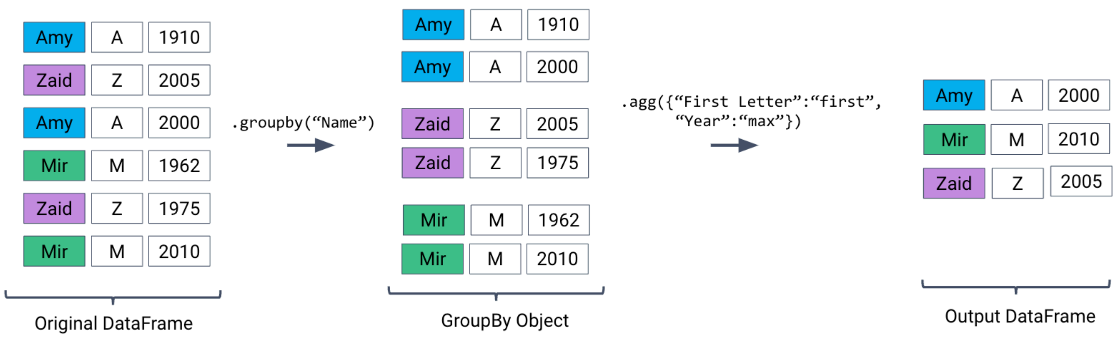

Figure 4.2: Aggregating using "first"

```
babynames_new.groupby("Name").agg({"First Letter":"first", "Year":"max"}).head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

Name	First Letter	Year
Aadan	A	2014
Aadarsh	A	2019
Aaden	A	2020
Aadhav	A	2019
Aadhini	A	2022

4.2.2 Plotting Birth Counts

Let's use `.agg` to find the total number of babies born in each year. Recall that using `.agg` with `.groupby()` follows the format: `df.groupby(column_name).agg(aggregation_function)`. The line of code below gives us the total number of babies born in each year.

```
babynames.groupby("Year")[["Count"]].agg(sum).head(5)
# Alternative 1
# babynames.groupby("Year")[["Count"]].sum()
# Alternative 2
# babynames.groupby("Year").sum(numeric_only=True)
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

	Count
Year	
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

Here's an illustration of the process:

Plotting the `Dataframe` we obtain tells an interesting story.

```
import plotly.express as px
puzzle2 = babynames.groupby("Year")[["Count"]].agg(sum)
px.line(puzzle2, y = "Count")
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): text/html
```

A word of warning: we made an enormous assumption when we decided to use this dataset to estimate birth rate. According to this article from the Legistlative Analyst Office, the true number of babies born in California in 2020 was 421,275. However, our plot shows 362,882 babies —— what happened?

4.2.3 Summary of the .groupby() Function

A `groupby` operation involves some combination of **splitting a `DataFrame` into grouped subframes**, **applying a function**, and **combining the results**.

For some arbitrary `DataFrame` `df` below, the code `df.groupby("year").agg(sum)` does the following:

- **Splits** the `DataFrame` into sub-`DataFrame`s with rows belonging to the same year.
- **Applies** the `sum` function to each column of each sub-`DataFrame`.
- **Combines** the results of `sum` into a single `DataFrame`, indexed by `year`.

4.2.4 Revisiting the .agg() Function

`.agg()` can take in any function that aggregates several values into one summary value. Some commonly-used aggregation functions can even be called directly, without explicit use of `.agg()`. For example, we can call `.mean()` on `.groupby()`:

```
babynames.groupby("Year").mean().head()
```

We can now put this all into practice. Say we want to find the baby name with sex "F" that has fallen in popularity the most in California. To calculate this, we can first create a metric: "Ratio to Peak" (RTP). The RTP is the ratio of babies born with a given name in 2022 to the *maximum* number of babies born with the name in *any* year.

Let's start with calculating this for one baby, "Jennifer".

```
# We filter by babies with sex "F" and sort by "Year"
f_babynames = babynames[babynames["Sex"] == "F"]
f_babynames = f_babynames.sort_values(["Year"])

# Determine how many Jennifers were born in CA per year
jenn_counts_series = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"]

# Determine the max number of Jennifers born in a year and the number born in 2022
# to calculate RTP
max_jenn = max(f_babynames[f_babynames["Name"] == "Jennifer"]["Count"])
curr_jenn = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"].iloc[-1]
rtp = curr_jenn / max_jenn
rtp
```

```
0.018796372629843364
```

By creating a function to calculate RTP and applying it to our `DataFrame` by using `.groupby()`, we can easily compute the RTP for all names at once!

```
def ratio_to_peak(series):
    return series.iloc[-1] / max(series)

#Using .groupby() to apply the function
rtp_table = f_babynames.groupby("Name")[["Year", "Count"]].agg(ratio_to_peak)
rtp_table.head()
```

```
/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWa

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S
```

Name	Year	Count
Aadhini	1.0	1.000000
Aadhira	1.0	0.500000
Aadhya	1.0	0.660000
Aadya	1.0	0.586207
Aahana	1.0	0.269231

In the rows shown above, we can see that every row shown has a `Year` value of `1.0`.

This is the "**`pandas`**-ification" of logic you saw in Data 8. Much of the logic you've learned in Data 8 will serve you well in Data 100.

4.2.5 Nuisance Columns

Note that you must be careful with which columns you apply the `.agg()` function to. If we were to apply our function to the table as a whole by doing `f_babynames.groupby("Name").agg(ratio_to_peak)`, executing our `.agg()` call would result in a `TypeError`.

We can avoid this issue (and prevent unintentional loss of data) by explicitly selecting column(s) we want to apply our aggregation function to **BEFORE** calling `.agg()`,

4.2.6 Renaming Columns After Grouping

By default, `.groupby` will not rename any aggregated columns. As we can see in the table above, the aggregated column is still named `Count` even though it now represents the RTP. For better readability, we can rename `Count` to `Count RTP`