Principles and Techniques of Data Science

Data 100

Kanu Grover Bella Crouch

Table of contents

Welcome

About the Course Notes

This text was developed for the Spring 2023 Edition of the UC Berkeley course Data 100: Principles and Techniques of Data Science.

As this project is in development during the Spring 2023 semester, the course notes may be in flux. We appreciate your understanding. If you spot any errors or would like to suggest any changes, please email us. **Email**: data100.instructors@berkeley.edu

1 Introduction

> **Note**
>
> - Understand the stages of the data science lifecycle.

Data science is an interdisciplinary field with a variety of applications. The field is rapidly evolving; many of the key technical underpinnings in modern-day data science have been popularized during the early 21st century.

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. To do so, 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 will 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 gives a clear point to know when to finish 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, its 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, 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 to 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 collected 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 reveal these 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 question. 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 by our findings, or our initial exploration may have brought up new questions that require a 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 untrue 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 list of requirements. In our journey exploring the lifecycle, we'll cover both the underlying theory and technologies used in data science, and we hope you'll build an appreciation for the field.

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

2 Pandas I

> **Note**
>
> - Build familiarity with basic `pandas` syntax
> - Learn the methods of selecting and filtering data from a DataFrame.
> - Understand the differences between DataFrames and Series

Data scientists work with data stored in a variety of formats. The primary focus of this class is in understanding tabular data – one of the most widely used formats in data science. This note introduces DataFrames, which are among the most popular representations of tabular data. We'll also introduce `pandas`, the standard Python package for manipulating data in DataFrames.

2.1 Introduction to Exploratory Data Analysis

Imagine you collected, or have been given a box of data. What do you do next?

The first step is to clean your data. **Data cleaning** often corrects issues in the structure and formatting of data, including missing values and unit conversions.

Data scientists have coined the term **exploratory data analysis (EDA)** to describe the process of transforming raw data to insightful observations. EDA is an *open-ended* analysis of transforming, visualizing, and summarizing patterns in data. In order to conduct EDA, we first need to familiarize ourselves with `pandas` – an important programming tool.

2.2 Introduction to Pandas

`pandas` is a data analysis library to make data cleaning and analysis fast and convenient in Python.

The `pandas` library adopts many coding idioms from `NumPy`. The biggest difference is that `pandas` is designed for working with tabular data, one of the most common data formats (and the focus of Data 100).

Before writing any code, we must import `pandas` into our Python environment.

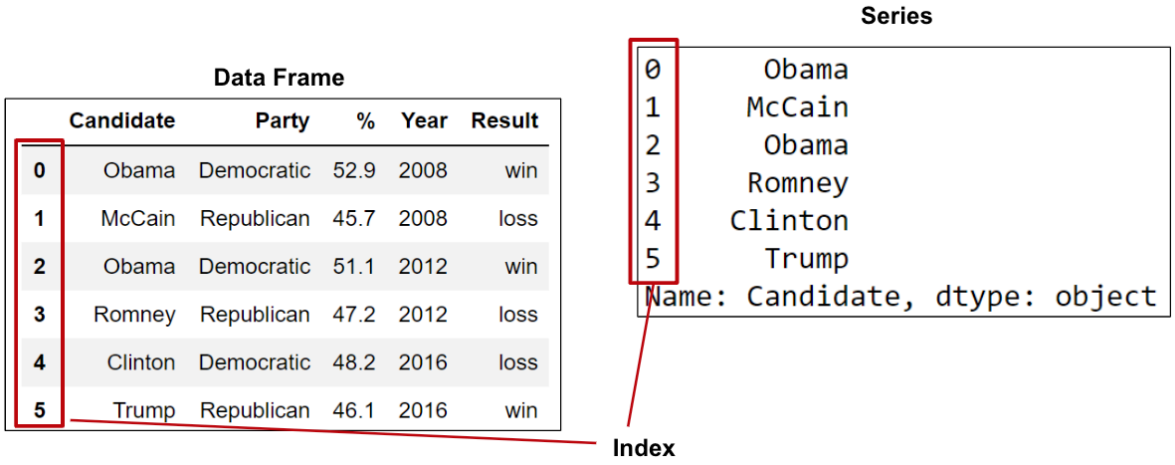
```
# `pd` is the conventional alias for Pandas, as `np` is for NumPy
import pandas as pd
```

2.3 Series, DataFrames, and Indices

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.



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 `Candidate` column. Both contain an **Index**, or a shared list of row labels (the integers from 0 to 5, inclusive).

2.3.1 Series

A Series represents a column of a DataFrame; more generally, it can be any 1-dimensional array-like object containing values of the same type with associated data labels, called its index.

```
import pandas as pd

s = pd.Series([-1, 10, 2])
```

```
print(s)
```

```
0    -1
1    10
2     2
dtype: int64
```

```
s.array # Data contained within the Series
```

```
<PandasArray>
[-1, 10, 2]
Length: 3, dtype: int64
```

```
s.index # The Index of the Series
```

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

By default, row indices in `pandas` are a sequential list of integers beginning from 0. Optionally, a list of desired indices can be passed to the `index` argument.

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

```
a    -1
b    10
c     2
dtype: int64
```

Indices can also be changed after initialization.

```
s.index = ["first", "second", "third"]
print(s)
```

```
first     -1
second    10
third      2
dtype: int64
```

2.3.1.1 Selection in Series

Similar to an array, we can select a single value or a set of values from a Series. There are 3 primary methods of selecting data.

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

Let's define the following Series `ser`.

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

```
a    4
b   -2
c    0
d    6
dtype: int64
```

2.3.1.1.1 A Single Index Label

```
print(ser["a"]) # Notice how the return value is a single array element
```

```
4
```

2.3.1.1.2 A List of Index Labels

```
ser[["a", "c"]] # Notice how the return value is another Series
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	0
a	4
c	0

2.3.1.1.3 A Filtering Condition

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

We first must apply a vectorized boolean operation to our Series that encodes the filter conditon.

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

	0
a	True
b	False
c	False
d	True

Upon "indexing" in our Series with this condition, `pandas` selects only the rows with `True` values.

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

	0
a	4
d	6

2.3.2 DataFrames

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.

Here is an example of a DataFrame that contains election data.

```
import pandas as pd

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

Let's dissect the code above.

1. We first import the `pandas` library into our Python environment, using the alias `pd`. `import pandas as pd`
2. There are a number of ways to read data into a DataFrame. In Data 100, our data are typically stored in a CSV (comma-seperated 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("elections.csv")`

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 presedential candidate from some particular year. Each column represents a single attribute, or feature of the record.

In the example above, we constructed a DataFrame object using data from a CSV file. As we'll explore in the next section, we can create a DataFrame with data of our own.

2.3.2.1 Creating a DataFrame

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

1. Using a list and column names
2. From a dictionary
3. From a Series

2.3.2.1.1 Using a List and Column Names

Consider the following examples.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Numbers
0	1
1	2
2	3

The first code cell creates a DataFrame with a single column `Numbers`, while the second creates a DataFrame with an additional column `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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Number	Description
0	1	one
1	2	two

2.3.2.1.2 From a Dictionary

A second (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.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

2.3.2.1.3 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.

```
# 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"])

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

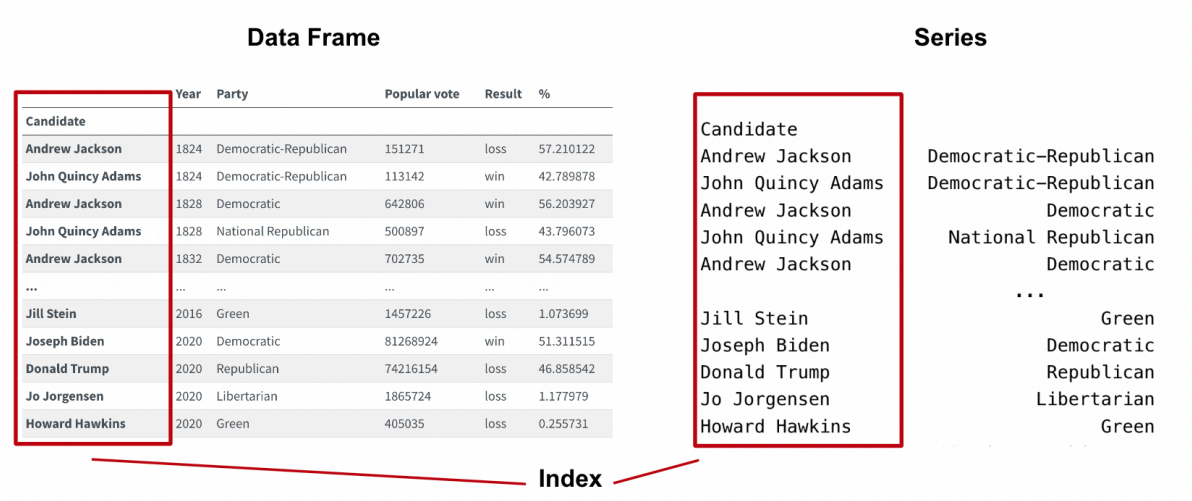
	A-column	B-column
r1	a1	b1
r2	a2	b2
r3	a3	b3

2.3.3 Indices

The major takeaway: we can think of a **DataFrame** as a collection of **Series** that all share the same **Index**.

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 presedential candidates. Selecting a new Series from this modified DataFrame yields the following.

```
# This sets the index to the "Candidate" column
elections.set_index("Candidate", inplace=True)
```



To retrieve the indices of a DataFrame, simply use the `.index` attribute of the DataFrame class.

```
elections.index
```

```
Index(['Andrew Jackson', 'John Quincy Adams', 'Andrew Jackson',
       'John Quincy Adams', 'Andrew Jackson', 'Henry Clay', 'William Wirt',
       'Hugh Lawson White', 'Martin Van Buren', 'William Henry Harrison',
       ...
       'Darrell Castle', 'Donald Trump', 'Evan McMullin', 'Gary Johnson',
       'Hillary Clinton', 'Jill Stein', 'Joseph Biden', 'Donald Trump',
       'Jo Jorgensen', 'Howard Hawkins'],
      dtype='object', name='Candidate', length=182)
```

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

2.4 Slicing in DataFrames

Now that we've learned how to create DataFrames, 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**. We will do so with three primary methods of the DataFrame class:

1. `.loc`
2. `.iloc`
3. `[]`

2.4.1 Indexing with .loc

The `.loc` operator selects rows and columns in a DataFrame by their row and column label(s), respectively. The **row 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.

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

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

```
'Andrew Jackson'
```

To select *multiple* rows and columns, we can use Python slice notation. Here, we select both the first four rows and columns.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Party	Popular vote
0	1824	Democratic-Republican	151271
1	1824	Democratic-Republican	113142
2	1828	Democratic	642806
3	1828	National Republican	500897

Suppose that instead, we wanted *every* column value for the first four rows in the `elections` DataFrame. The shorthand `:` is useful for this.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

There are a couple of things we should note. 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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

2.4.2 Indexing with .iloc

Slicing with `.iloc` works similarily to `.loc`, although `.iloc` uses the integer positions of rows and columns rather the labels. 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 for the first presedential candidate in our `elections` DataFrame:

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

```
1824
```

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} (or first) position of the `elections` DataFrame. Generally, this is true of any DataFrame where the row labels are incremented in ascending order from 0.

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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

Slicing is no longer inclusive in `.iloc` - it's *exclusive*. This is one of Pandas syntatical subtleties; you'll get used to 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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

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.

2.4.3 Indexing with []

The `[]` selection operator is the most baffling of all, yet the 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.

2.4.3.1 A slice of row numbers

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

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

2.4.3.2 A list of column labels

Suppose we now want the first four columns.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

2.4.3.3 A single column label

Lastly, if we only want 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 looks like a Series! In this course, we'll become very comfortable with `[]`, especially for selecting columns. In practice, `[]` is much more common than `.loc`.

2.5 Parting Note

The `pandas` library is enormous and contains many useful functions. Here is a link to documentation.

The introductory `pandas` lectures will cover important data structures and methods you should be fluent in. However, we want you to get familiar with the real world programming practice of …Googling! Answers to your questions can be found in documentation, Stack Overflow, etc.

With that, let's move on to Pandas II.

3 Pandas II

> **Note**
>
> - Build familiarity with advanced `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.

```
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 = "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 = '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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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 if they follow 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 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 in the position of a corresponding True value in the array.

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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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`s and `False`s 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/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

Here, `logical_operator` evaluates to a Series of boolean values with length 400762.

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

```
There are a total of 400762 values in 'logical_operator'
```

Rows starting at row 0 and ending at row 235790 evaluate to `True` and are thus returned in the DataFrame.

```
print("The 0th item in this 'logical_operator' is: {}".format(logical_operator.iloc[0]))
print("The 235790th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2
print("The 235791th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2
```

```
The 0th item in this 'logical_operator' is: True
The 235790th item in this 'logical_operator' is: True
The 235791th item in this 'logical_operator' is: False
```

Passing a Series as an argument to `babynames[]` has the same affect 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 thoughout the course.

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

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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 operators that allow us to filter results by multiple conditions. Some examples include the `&` (and) operator and the `|` (or) operator.

Note: When combining multiple conditions with logical operators, be sure to surround each condition with a set of parenthesis `()`. If you forget, your code will throw an error.

For example, if we want to return data on all females born before the 21st century, we can write:

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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. `Pandas` provide many alternatives:

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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 `.isin` function can be used to filter dataframes. The method helps in selecting rows with having a particular (or multiple) value in a particular column.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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.

```
babynames[babynames["Name"].str.startswith("N")].head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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 Handy 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.

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

3.2.1 Numpy

```
bella_counts = babynames[babynames["Name"] == "Bella"]["Count"]
```

```
# Average number of babies named Bella each year
np.mean(bella_counts)
```

```
270.1860465116279
```

```
# Max number of babies named Bella born on a given year
max(bella_counts)
```

```
902
```

3.2.2 `.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.

```
babynames.shape
```

```
(400762, 5)
```

```
babynames.size
```

```
2003810
```

3.2.3 `.describe()`

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

```
babynames.describe()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Count
count	400762.000000	400762.000000
mean	1985.131287	79.953781
std	26.821004	295.414618
min	1910.000000	5.000000
25%	1968.000000	7.000000
50%	1991.000000	13.000000
75%	2007.000000	38.000000
max	2021.000000	8262.000000

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

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Sex
count	400762
unique	2
top	F
freq	235791

3.2.4 `.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()` lets us quickly select random entries (a row if called from a DataFrame, or a value if called from a Series).

```
babynames.sample()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count
49083	CA	F	1963	Loria	6

```
babynames.sample(5).iloc[:, 2:]
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Name	Count
50906	1964	Dahlia	6
316731	1991	Tom	44
320245	1992	Renato	9
290560	1977	Isidoro	5
255301	1947	Leslie	145

```
babynames[babynames["Year"] == 2000].sample(4, replace = True).iloc[:, 2:]
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Name	Count
340167	2000	Imran	9
149643	2000	Tabitha	60
340850	2000	Yaseen	6
339407	2000	Elisha	28

3.2.5 `.value_counts()`

When we want to know the distribution of the items in a Series (for example, what items are most/least common), we use `.value-counts()` to get a breakdown of the unique *values* and their *counts*. 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`.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Name
Jean	221
Francis	219
Guadalupe	216
Jessie	215
Marion	213

3.2.6 `.unique()`

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

Exercise: what function can we call on the Series below to get the number of unique names?

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

```
array(['Mary', 'Helen', 'Dorothy', ..., 'Zyire', 'Zylo', 'Zyrus'],
      dtype=object)
```

3.2.7 `.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` allows us to order a DataFrame or Series by a specified rule. For DataFrames, we must specify the column by which we want to compare the rows and the function will return such rows. We can choose to either receive the rows in `ascending` order (default) or `descending` order.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count
263272	CA	M	1956	Michael	8262
264297	CA	M	1957	Michael	8250
313644	CA	M	1990	Michael	8247
278109	CA	M	1969	Michael	8244
279405	CA	M	1970	Michael	8197

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.

```
babynames["Name"].sort_values(ascending=True).head()
```

	Name
380256	Aadan
362255	Aadan
365374	Aadan
394460	Aadarsh
366561	Aaden

3.2.7.1 Sorting With a Custom Key

Using `.sort_values` can be useful in many situations, but it many not cover all use cases. This is because `pandas` automatically sorts values in order according to numeric value (for number data) or alphabetical order (for string data). The following code finds the top 5 most popular names in California in 2021.

```
# Sort names by count in year 2021
# Recall that `.head(5)` displays the first five rows in the DataFrame
babynames[babynames["Year"] == 2021].sort_values("Count", ascending=False).head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count
397909	CA	M	2021	Noah	2591
397910	CA	M	2021	Liam	2469
232145	CA	F	2021	Olivia	2395
232146	CA	F	2021	Emma	2171
397911	CA	M	2021	Mateo	2108

This offers us a lot of functionality, but what if we need to sort by some other metric? For example, what if we wanted to find the longest names in the DataFrame?

We can do this by specifying the `key` parameter of `.sort_values`. The `key` parameter is assigned to a function of our choice. This function is then applied to each value in the specified column. `pandas` will, finally, sort the DataFrame by the values outputted by the function.

```
# Here, a lambda function is applied to find the length of each value, `x`, in the "Name"
babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False).head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

3.3 Adding and Removing Columns

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 `dataframe["new_column"]`, then assign this to a Series or Array containing the values that will populate this column.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count	name_lengths
313143	CA	M	1989	Franciscojavier	6	15
333732	CA	M	1997	Ryanchristopher	5	15
330421	CA	M	1996	Franciscojavier	8	15
323615	CA	M	1993	Johnchristopher	5	15
310235	CA	M	1988	Franciscojavier	10	15

In the example above, we made use of an in-built function given to us by the `str` accessor for getting the length of names. Then we used `name_length` column to sort the dataframe. What if we had wanted to generate the values in our new column using a function of our own making?

We can do this using the Series `.map` method. `.map` takes in a function as input, and will apply this function to each value of a Series.

For example, say we wanted to find the number of occurrences of the sequence "dr" or "ea" in each name.

```
# 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
babynames.sort_values(by = "dr_ea_count", ascending = False).head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count	name_lengths	dr_ea_count
101969	CA	F	1986	Deandrea	6	8	3
304390	CA	M	1985	Deandrea	6	8	3
131022	CA	F	1994	Leandrea	5	8	3
115950	CA	F	1990	Deandrea	5	8	3
108723	CA	F	1988	Deandrea	5	8	3

If we want to remove a column or row of a DataFrame, we can call the `.drop` 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 "dr_ea_count" and "length" columns from the DataFrame
babynames = babynames.drop(["dr_ea_count", "name_lengths"], axis="columns")
babynames.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

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

In other words, if we simply call:

```
# This creates a copy of `babynames` and removes the row with label 3...
babynames.drop(3, axis="rows")

# ...but the original `babynames` is unchanged!
# Notice that the row with label 3 is still present
babynames.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

3.4 Aggregating Data with GroupBy

Up until this point, we have been working with individual rows of DataFrames. 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.

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 0x11112f790>
```

What does this strange output mean? Calling `.groupby` has generated a `GroupBy` object. You can imagine this as a set of "mini" sub-DataFrames, 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" DataFrames, 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.groupby("Year").agg(sum).head(5)
```

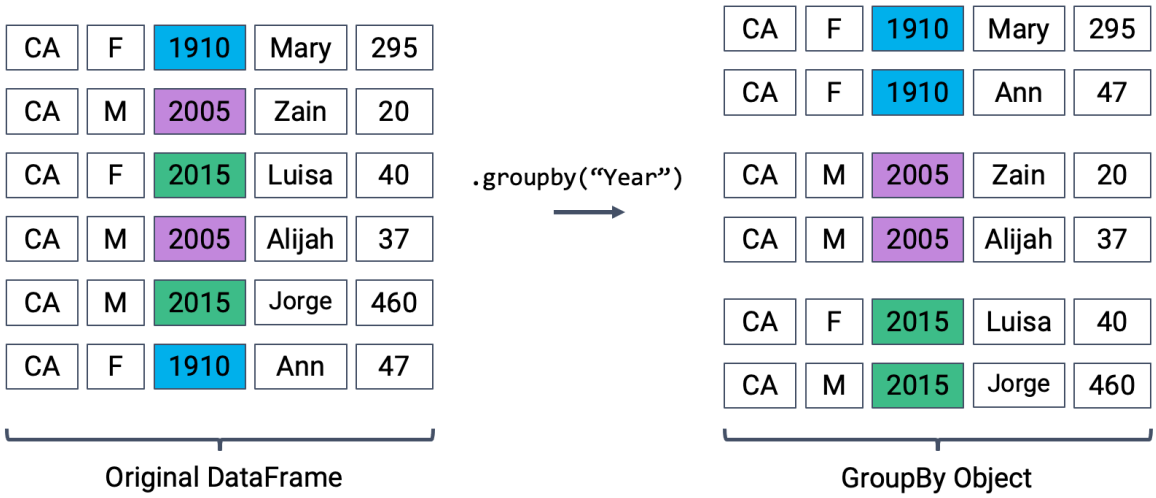


Figure 3.1: Creating a GroupBy object

```
/var/folders/7x/09tm3ct91s14kpwnpd0400k00000gn/T/ipykernel_5807/1490078280.py:1: FutureWarnin
  babynames.groupby("Year").agg(sum).head(5)
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatter
  return method()
```

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

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.

You may be wondering: where did the `"State"`, `"Sex"`, and `"Name"` columns go? Logically, it doesn't make sense to `sum` the string data in these columns (how would we add "Mary" + "Ann"?). Because of this, `pandas` will simply omit these columns when it performs the aggregation on the DataFrame. Since this happens implicitly, without the user specifying that these columns should be ignored, it's easy to run into troubling situations where columns

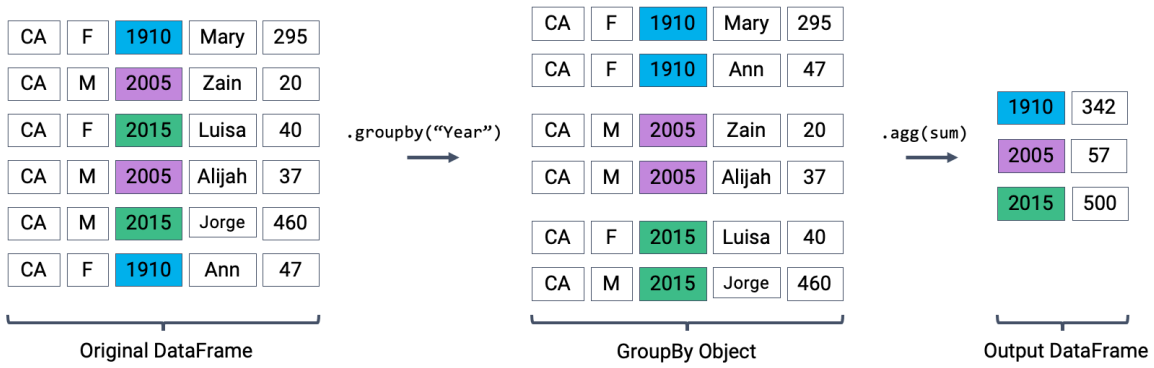


Figure 3.2: Performing an aggregation

are removed without the programmer noticing. It is better coding practice to select *only* the columns we care about before performing the aggregation.

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

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

3.4.1 Parting note

Manipulating `DataFrames` is a skill that is not mastered in just one day. Due to the flexibility of `pandas`, there are many different ways to get from a point A to a 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

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

4.1 More on `agg()` Function

Last time, we introduced the concept of aggregating data – we familiarized ourselves with `GroupBy` objects and used them as tools to consolidate and summarize a DataFrame. In this lecture, we will explore 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.

4.2 `GroupBy()`, Continued

As we learned last lecture, 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:

- Organizes all rows with the same year into a subframe for that year.
- Creates a new DataFrame with one row representing each subframe year.
- Combines all integer rows in each subframe using the `sum` function.

4.2.1 Aggregation with `lambda` Functions

Throughout this note, we'll work with the `elections` DataFrame.

```
import pandas as pd

elections = pd.read_csv("data/elections.csv")
elections.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

What if we wish to aggregate our DataFrame using a non-standard function – for example, a function of our own design? We can do so by combining `.agg` with `lambda` expressions.

Let's first consider a puzzle to jog our memory. We will attempt to find the `Candidate` from each `Party` with the highest `%` of votes.

A naive approach may be to group by the `Party` column and aggregate by the maximum.

```
elections.groupby("Party").agg(max).head(10)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Party	Year	Candidate	Popular vote	Result	%
American	1976	Thomas J. Anderson	873053	loss	21.554001
American Independent	1976	Lester Maddox	9901118	loss	13.571218
Anti-Masonic	1832	William Wirt	100715	loss	7.821583
Anti-Monopoly	1884	Benjamin Butler	134294	loss	1.335838
Citizens	1980	Barry Commoner	233052	loss	0.270182
Communist	1932	William Z. Foster	103307	loss	0.261069
Constitution	2016	Michael Peroutka	203091	loss	0.152398
Constitutional Union	1860	John Bell	590901	loss	12.639283
Democratic	2020	Woodrow Wilson	81268924	win	61.344703
Democratic-Republican	1824	John Quincy Adams	151271	win	57.210122

This approach is clearly wrong – the DataFrame claims that Woodrow Wilson won the presidency in 2020.

Why is this happening? Here, the `max` aggregation function is taken over every column *independently*. Among Democrats, `max` is computing:

- The most recent `Year` a Democratic candidate ran for president (2020)
- The `Candidate` with the alphabetically "largest" name ("Woodrow Wilson")
- The `Result` with the alphabetically "largest" outcome ("win")

Instead, let's try a different approach. We will:

1. Sort the DataFrame so that rows are in descending order of `%`
2. Group by `Party` and select the first row of each groupby object

While it may seem unintuitive, sorting `elections` by descending order of `%` is extremely helpful. If we then group by `Party`, the first row of each groupby object will contain information about the `Candidate` with the highest voter `%`.

```
elections_sorted_by_percent = elections.sort_values("%", ascending=False)
elections_sorted_by_percent.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Candidate	Party	Popular vote	Result	%
114	1964	Lyndon Johnson	Democratic	43127041	win	61.344703
91	1936	Franklin Roosevelt	Democratic	27752648	win	60.978107
120	1972	Richard Nixon	Republican	47168710	win	60.907806
79	1920	Warren Harding	Republican	16144093	win	60.574501
133	1984	Ronald Reagan	Republican	54455472	win	59.023326

```
elections_sorted_by_percent.groupby("Party").agg(lambda x : x.iloc[0]).head(10)

# Equivalent to the below code
# elections_sorted_by_percent.groupby("Party").agg('first').head(10)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Party	Year	Candidate	Popular vote	Result	%
American	1856	Millard Fillmore	873053	loss	21.554001
American Independent	1968	George Wallace	9901118	loss	13.571218
Anti-Masonic	1832	William Wirt	100715	loss	7.821583
Anti-Monopoly	1884	Benjamin Butler	134294	loss	1.335838
Citizens	1980	Barry Commoner	233052	loss	0.270182
Communist	1932	William Z. Foster	103307	loss	0.261069
Constitution	2008	Chuck Baldwin	199750	loss	0.152398
Constitutional Union	1860	John Bell	590901	loss	12.639283
Democratic	1964	Lyndon Johnson	43127041	win	61.344703
Democratic-Republican	1824	Andrew Jackson	151271	loss	57.210122

Notice how our code correctly determines that Lyndon Johnson from the Democratic Party has the highest voter `%`.

More generally, `lambda` functions are used to design custom aggregation functions that aren't pre-defined by Python. The input parameter `x` to the `lambda` function is a `GroupBy` object. Therefore, it should make sense why `lambda x : x.iloc[0]` selects the first row in each groupby object.

In fact, there's a few different ways to approach this problem. Each approach has different tradeoffs in terms of readability, performance, memory consumption, complexity, etc. We've given a few examples below.

Note: Understanding these alternative solutions is not required. They are given to demonstrate the vast number of problem-solving approaches in `pandas`.

```
# Using the idxmax function
best_per_party = elections.loc[elections.groupby('Party')['%'].idxmax()]
best_per_party.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Candidate	Party	Popular vote	Result	%
22	1856	Millard Fillmore	American	873053	loss	21.554001
115	1968	George Wallace	American Independent	9901118	loss	13.571218
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
127	1980	Barry Commoner	Citizens	233052	loss	0.270182

```
# Using the .drop_duplicates function
best_per_party2 = elections.sort_values('%').drop_duplicates(['Party'], keep='last')
best_per_party2.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Candidate	Party	Popular vote	Result	%
148	1996	John Hagelin	Natural Law	113670	loss	0.118219
164	2008	Chuck Baldwin	Constitution	199750	loss	0.152398
110	1956	T. Coleman Andrews	States' Rights	107929	loss	0.174883
147	1996	Howard Phillips	Taxpayers	184656	loss	0.192045
136	1988	Lenora Fulani	New Alliance	217221	loss	0.237804

4.2.2 Other `GroupBy` Features

There are many aggregation methods we can use with `.agg`. Some useful options are:

- `.max`: creates a new DataFrame with the maximum value of each group
- `.mean`: creates a new DataFrame with the mean value of each group
- `.size`: creates a new Series with the number of entries in each group

In fact, these (and other) aggregation functions are so common that `pandas` allows for writing shorthand. Instead of explicitly stating the use of `.agg`, we can call the function directly on the `GroupBy` object.

For example, the following are equivalent:

- `elections.groupby("Candidate").agg(mean)`
- `elections.groupby("Candidate").mean()`

4.2.3 The `groupby.filter()` function

Another common use for `GroupBy` objects is to filter data by group.

`groupby.filter` takes an argument f, where f is a function that:

- Takes a `GroupBy` object as input
- Returns a single `True` or `False` for the entire subframe

`GroupBy` objects that correspond to `True` are returned in the final result, whereas those with a `False` value are not. Importantly, `groupby.filter` is different from `groupby.agg` in that the *entire* subframe is returned in the final DataFrame, not just a single row.

To illustrate how this happens, consider the following `.filter` function applied on some arbitrary data. Say we want to identify "tight" election years – that is, we want to find all rows that correspond to elections years where all candidates in that year won a similar portion of the total vote. Specifically, let's find all rows corresponding to a year where no candidate won more than 45% of the total vote.

An equivalent way of framing this goal is to say:

- Find the years where the maximum `%` in that year is less than 45%
- Return all DataFrame rows that correspond to these years

For each year, we need to find the maximum `%` among *all* rows for that year. If this maximum `%` is lower than 45%, we will tell `pandas` to keep all rows corresponding to that year.

```
elections.groupby("Year").filter(lambda sf: sf["%"].max() < 45).head(9)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year	Candidate	Party	Popular vote	Result	%
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
66	1912	Eugene V. Debs	Socialist	901551	loss	6.004354
67	1912	Eugene W. Chafin	Prohibition	208156	loss	1.386325
68	1912	Theodore Roosevelt	Progressive	4122721	loss	27.457433
69	1912	William Taft	Republican	3486242	loss	23.218466
70	1912	Woodrow Wilson	Democratic	6296284	win	41.933422

What's going on here? In this example, we've defined our filtering function, f, to be `lambda sf: sf["%"].max() < 45`. This filtering function will find the maximum `"%"` value among all entries in the grouped subframe, which we call `sf`. If the maximum value is less than 45, then the filter function will return `True` and all rows in that grouped subframe will appear in the final output DataFrame.

Examine the DataFrame above. Notice how, in this preview of the first 9 rows, all entries from the years 1860 and 1912 appear. This means that in 1860 and 1912, no candidate in that year won more than 45% of the total vote.

You may ask: how is the `groupby.filter` procedure different to the boolean filtering we've seen previously? Boolean filtering considers *individual* rows when applying a boolean condition. For example, the code `elections[elections["%"] < 45]` will check the `"%"` value of every single row in `elections`; if it is less than 45, then that row will be kept in the output. `groupby.filter`, in contrast, applies a boolean condition *across* all rows in a group. If not all rows in that group satisfy the condition specified by the filter, the entire group will be discarded in the output.

4.3 Aggregating Data with Pivot Tables

We know now that `.groupby` gives us the ability to group and aggregate data across our DataFrame. The examples above formed groups using just one column in the DataFrame. It's possible to group by multiple columns at once by passing in a list of columns names to `.groupby`.

Let's consider the `babynames` dataset from last lecture. In this problem, we will find the total number of baby names associated with each sex for each year. To do this, we'll group by *both* the `"Year"` and `"Sex"` columns.

```
import urllib.request
import os.path

# Download data from the web directly
data_url = "https://www.ssa.gov/oact/babynames/names.zip"
local_filename = "data/babynames.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())


# Load data without unzipping the file
import zipfile
babynames = []
with zipfile.ZipFile(local_filename, "r") as zf:
    data_files = [f for f in zf.filelist if f.filename[-3:] == "txt"]
    def extract_year_from_filename(fn):
        return int(fn[3:7])
    for f in data_files:
        year = extract_year_from_filename(f.filename)
        with zf.open(f) as fp:
            df = pd.read_csv(fp, names=["Name", "Sex", "Count"])
```

```
            df["Year"] = year
            babynames.append(df)
babynames = pd.concat(babynames)
```

```
babynames.head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Name	Sex	Count	Year
0	Mary	F	7065	1880
1	Anna	F	2604	1880
2	Emma	F	2003	1880
3	Elizabeth	F	1939	1880
4	Minnie	F	1746	1880

```
# Find the total number of baby names associated with each sex for each year in the data
babynames.groupby(["Year", "Sex"])[["Count"]].agg(sum).head(6)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

		Count
Year	Sex	
1880	F	90994
	M	110490
1881	F	91953
	M	100737
1882	F	107847
	M	113686

Notice that both `"Year"` and `"Sex"` serve as the index of the DataFrame (they are both rendered in bold). We've created a *multindex* where two different index values, the year and sex, are used to uniquely identify each row.

This isn't the most intuitive way of representing this data – and, because multindexes have multiple dimensions in their index, they can often be difficult to use.

Another strategy to aggregate across two columns is to create a pivot table. You saw these back in Data 8. One set of values is used to create the index of the table; another set is used

to define the column names. The values contained in each cell of the table correspond to the aggregated data for each index-column pair.

The best way to understand pivot tables is to see one in action. Let's return to our original goal of summing the total number of names associated with each combination of year and sex. We'll call the `pandas` `.pivot_table` method to create a new table.

```
# The `pivot_table` method is used to generate a Pandas pivot table
import numpy as np
babynames.pivot_table(index = "Year", columns = "Sex", values = "Count", aggfunc = np.sum)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Sex Year	F	M
1880	90994	110490
1881	91953	100737
1882	107847	113686
1883	112319	104625
1884	129019	114442

Looks a lot better! Now, our DataFrame is structured with clear index-column combinations. Each entry in the pivot table represents the summed count of names for a given combination of `"Year"` and `"Sex"`.

Let's take a closer look at the code implemented above.

- `index = "Year"` specifies the column name in the original DataFrame that should be used as the index of the pivot table
- `columns = "Sex"` specifies the column name in the original DataFrame that should be used to generate the columns of the pivot table
- `values = "Count"` indicates what values from the original DataFrame should be used to populate the entry for each index-column combination
- `aggfunc = np.sum` tells `pandas` what function to use when aggregating the data specified by `values`. Here, we are `sum`ming the name counts for each pair of `"Year"` and `"Sex"`

We can even include multiple values in the index or columns of our pivot tables.

```
babynames_pivot = babynames.pivot_table(
    index="Year",     # the rows (turned into index)
    columns="Sex",    # the column values
    values=["Count", "Name"],
```

```
    aggfunc=max,   # group operation
)
babynames_pivot.head(6)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Count		Name	
Sex	F	M	F	M
Year				
1880	7065	9655	Zula	Zeke
1881	6919	8769	Zula	Zeb
1882	8148	9557	Zula	Zed
1883	8012	8894	Zula	Zeno
1884	9217	9388	Zula	Zollie
1885	9128	8756	Zula	Zollie

4.4 Joining Tables

When working on data science projects, we're unlikely to have absolutely all the data we want contained in a single DataFrame – a real-world data scientist needs to grapple with data coming from multiple sources. If we have access to multiple datasets with related information, we can join two or more tables into a single DataFrame.

To put this into practice, we'll revisit the `elections` dataset.

```
elections.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

Say we want to understand the 2020 popularity of the names of each presidential candidate. To do this, we'll need the combined data of `babynames` *and* `elections`.

We'll start by creating a new column containing the first name of each presidential candidate. This will help us join each name in `elections` to the corresponding name data in `babynames`.

```
# This `str` operation splits each candidate's full name at each
# blank space, then takes just the candidiate's first name
elections["First Name"] = elections["Candidate"].str.split().str[0]
elections.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

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

```
# Here, we'll only consider `babynames` data from 2020
babynames_2020 = babynames[babynames["Year"]==2020]
babynames_2020.head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Name	Sex	Count	Year
0	Olivia	F	17641	2020
1	Emma	F	15656	2020
2	Ava	F	13160	2020
3	Charlotte	F	13065	2020
4	Sophia	F	13036	2020

Now, we're ready to join the two tables. `pd.merge` is the `pandas` method used to join DataFrames together. The `left` and `right` parameters are used to specify the DataFrames to be joined. The `left_on` and `right_on` parameters are assigned to the string names of the columns to be used when performing the join. These two `on` parameters tell `pandas` what values should act as pairing keys to determine which rows to merge across the DataFrames. We'll talk more about this idea of a pairing key next lecture.

```
merged = pd.merge(left = elections, right = babynames_2020, \
                  left_on = "First Name", right_on = "Name")
merged.head()
# Notice that pandas automatically specifies `Year_x` and `Year_y`
# when both merged DataFrames have the same column name to avoid confusion
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Year_x	Candidate	Party	Popular vote	Result	%	First Name	N
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	A
1	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	A
2	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	A
3	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	A
4	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew	A

5 Data Cleaning and EDA

> **Note**
>
> - Recognize common file formats
> - Categorize data by its variable type
> - Build awareness of issues with data faithfulness and develop targeted solutions

In the past few lectures, we've learned that `pandas` is a toolkit to restructure, modify, and explore a dataset. What we haven't yet touched on is *how* to make these data transformation decisions. When we receive a new set of data from the "real world," how do we know what processing we should do to convert this data into a usable form?

Data cleaning, also called **data wrangling**, is the process of transforming raw data to facilitate subsequent analysis. It is often used to address issues like:

- Unclear structure or formatting
- Missing or corrupted values
- Unit conversions
- …and so on

Exploratory Data Analysis (EDA) is the process of understanding a new dataset. It is an open-ended, informal analysis that involves familiarizing ourselves with the variables present in the data, discovering potential hypotheses, and identifying potential issues with the data. This last point can often motivate further data cleaning to address any problems with the dataset's format; because of this, EDA and data cleaning are often thought of as an "infinite loop," with each process driving the other.

In this lecture, we will consider the key properties of data to consider when performing data cleaning and EDA. In doing so, we'll develop a "checklist" of sorts for you to consider when approaching a new dataset. Throughout this process, we'll build a deeper understanding of this early (but very important!) stage of the data science lifecycle.

5.1 Structure

5.1.1 File Format

In the past two `pandas` lectures, we briefly touched on the idea of file format: the way data is encoded in a file for storage. Specifically, our `elections` and `babynames` datasets were stored and loaded as CSVs:

```
import pandas as pd
pd.read_csv("data/elections.csv").head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	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

CSVs, which stand for **Comma-Separated Values**, are a common tabular data format. To better understand the properties of a CSV, let's take a look at the first few rows of the raw data file to see what it looks like before being loaded into a DataFrame.

```
Year,Candidate,Party,Popular vote,Result,%

1824,Andrew Jackson,Democratic-Republican,151271,loss,57.21012204

1824,John Quincy Adams,Democratic-Republican,113142,win,42.78987796

1828,Andrew Jackson,Democratic,642806,win,56.20392707
```

Each row, or **record**, in the data is delimited by a newline. Each column, or **field**, in the data is delimited by a comma (hence, comma-separated!).

Another common file type is the **TSV (Tab-Separated Values)**. In a TSV, records are still delimited by a newline, while fields are delimited by `\t` tab character. A TSV can be loaded into `pandas` using `pd.read_csv()` with the `delimiter` parameter: `pd.read_csv("file_name.tsv", delimiter="\t")`. A raw TSV file is shown below.

```
Year   Candidate   Party   Popular vote    Result  %

1824    Andrew Jackson  Democratic-Republican   151271  loss    57.21012204

1824    John Quincy Adams   Democratic-Republican   113142  win 42.78987796

1828    Andrew Jackson  Democratic  642806  win 56.20392707
```

JSON (JavaScript Object Notation) files behave similarly to Python dictionaries. They can be loaded into `pandas` using `pd.read_json`. A raw JSON is shown below.

```
[

 {

   "Year": 1824,

   "Candidate": "Andrew Jackson",

   "Party": "Democratic-Republican",

   "Popular vote": 151271,

   "Result": "loss",

   "%": 57.21012204

 },
```

5.1.2 Variable Types

After loading data into a file, it's a good idea to take the time to understand what pieces of information are encoded in the dataset. In particular, we want to identify what variable types are present in our data. Broadly speaking, we can categorize variables into one of two overarching types.

Quantitative variables describe some numeric quantity or amount. We can sub-divide quantitative data into:

- **Continuous quantitative variables**: numeric data that can be measured on a continuous scale to arbitrary precision. Continuous variables do not have a strict set of possible values – they can be recorded to any number of decimal places. For example, weights, GPA, or CO2 concentrations

- **Discrete quantitative variables**: numeric data that can only take on a finite set of possible values. For example, someone's age or number of siblings.

Qualitative variables, also known as **categorical variables**, describe data that isn't measuring some quantity or amount. The sub-categories of categorical data are:

- **Ordinal qualitative variables**: categories with ordered levels. Specifically, ordinal variables are those where the difference between levels has no consistent, quantifiable meaning. For example, a Yelp rating or set of income brackets.
- **Nominal qualitative variables**: categories with no specific order. For example, someone's political affiliation or Cal ID number.

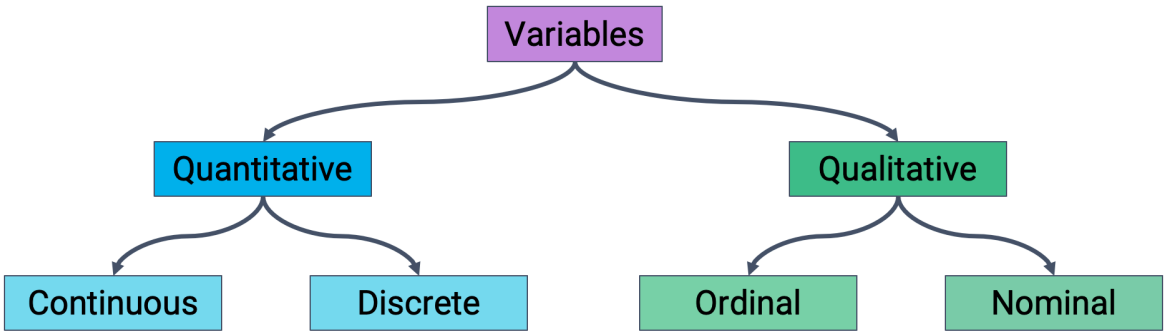


Figure 5.1: Classification of variable types

5.1.3 Primary and Foreign Keys

Last time, we introduced `.merge` as the `pandas` method for joining multiple DataFrames together. In our discussion of joins, we touched on the idea of using a "key" to determine what rows should be merged from each table. Let's take a moment to examine this idea more closely.

The **primary key** is the column or set of columns in a table that determine the values of the remaining columns. It can be thought of as the unique identifier for each individual row in the table. For example, a table of Data 100 students might use each student's Cal ID as the primary key.

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Cal ID	Name	Major
0	3034619471	Oski	Data Science
1	3035619472	Ollie	Computer Science
2	3025619473	Orrie	Data Science
3	3046789372	Ollie	Economics

The **foreign key** is the column or set of columns in a table that reference primary keys in other tables. Knowing a dataset's foreign keys can be useful when assigning the `left_on` and `right_on` parameters of `.merge`. In the table of office hour tickets below, `"Cal ID"` is a foreign key referencing the previous table.

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	OH Request	Cal ID	Question
0	1	3034619471	HW 2 Q1
1	2	3035619472	HW 2 Q3
2	3	3025619473	Lab 3 Q4
3	4	3035619472	HW 2 Q7

5.2 Granularity, Scope, and Temporality

After understanding the structure of the dataset, the next task is to determine what exactly the data represents. We'll do so by considering the data's granularity, scope, and temporality.

The **granularity** of a dataset is the level of detail included in the data. To determine the data's granularity, ask: what does each row in the dataset represent? Fine-grained data contains a high level of detail, with a single row representing a small individual unit. For example, each record may represent one person. Coarse-grained data is encoded such that a single row represents a large individual unit – for example, each record may represent a group of people.

The **scope** of a dataset is the subset of the population covered by the data. If we were investigating student performance in Data Science courses, a dataset with narrow scope might encompass all students enrolled in Data 100; a dataset with expansive scope might encompass all students in California.

The **temporality** of a dataset describes the time period over which the data was collected. To fully understand the temporality of the data, it may be necessary to standardize timezones or inspect recurring time-based trends in the data (Do patterns recur in 24-hour patterns? Over the course of a month? Seasonally?).

5.3 Faithfulness

At this stage in our data cleaning and EDA workflow, we've achieved quite a lot: we've identified how our data is structured, come to terms with what information it encodes, and gained insight as to how it was generated. Throughout this process, we should always recall the original intent of our work in Data Science – to use data to better understand and model the real world. To achieve this goal, we need to ensure that the data we use is faithful to reality; that is, that our data accurately captures the "real world."

Data used in research or industry is often "messy" – there may be errors or inaccuracies that impact the faithfulness of the dataset. Signs that data may not be faithful include:

- Unrealistic or "incorrect" values, such as negative counts, locations that don't exist, or dates set in the future
- Violations of obvious dependencies, like an age that does not match a birthday
- Clear signs that data was entered by hand, which can lead to spelling errors or fields that are incorrectly shifted
- Signs of data falsification, such as fake email addresses or repeated use of the same names
- Duplicated records or fields containing the same information

A common issue encountered with real-world datasets is that of missing data. One strategy to resolve this is to simply drop any records with missing values from the dataset. This does, however, introduce the risk of inducing biases – it is possible that the missing or corrupt records may be systemically related to some feature of interest in the data.

Another method to address missing data is to perform **imputation**: infer the missing values using other data available in the dataset. There is a wide variety of imputation techniques that can be implemented; some of the most common are listed below.

- Average imputation: replace missing values with the average value for that field
- Hot deck imputation: replace missing values with some random value
- Regression imputation: develop a model to predict missing values
- Multiple imputation: replace missing values with multiple random values

Regardless of the strategy used to deal with missing data, we should think carefully about *why* particular records or fields may be missing – this can help inform whether or not the absence of these values is signficant in some meaningful way.

6 EDA Demo: Tuberculosis in the United States

Now, let's follow this data-cleaning and EDA workflow to see what can we say about the presence of Tuberculosis in the United States!

We will examine the data included in the original CDC article published in 2021.

6.1 CSVs and Field Names

Suppose Table 1 was saved as a CSV file located in `data/cdc_tuberculosis.csv`.

We can then explore the CSV (which is a text file, and does not contain binary-encoded data) in many ways: 1. Using a text editor like emacs, vim, VSCode, etc. 2. Opening the CSV directly in DataHub (read-only), Excel, Google Sheets, etc. 3. The Python file object 4. pandas, using `pd.read_csv()`

1, 2. Let's start with the first two so we really solidify the idea of a CSV as **rectangular data (i.e., tabular data) stored as comma-separated values**.

3. Next, let's try using the Python file object. Let's check out the first three lines:

```
with open("data/cdc_tuberculosis.csv", "r") as f:
    i = 0
    for row in f:
        print(row)
        i += 1
        if i > 3:
            break
```

```
,No. of TB cases,,,TB incidence,,

U.S. jurisdiction,2019,2020,2021,2019,2020,2021

Total,"8,900","7,173","7,860",2.71,2.16,2.37
```

```
Alabama,87,72,92,1.77,1.43,1.83
```

Whoa, why are there blank lines interspaced between the lines of the CSV?

You may recall that all line breaks in text files are encoded as the special newline character `\n`. Python's `print()` prints each string (including the newline), and an additional newline on top of that.

If you're curious, we can use the `repr()` function to return the raw string with all special characters:

```
with open("data/cdc_tuberculosis.csv", "r") as f:
    i = 0
    for row in f:
        print(repr(row)) # print raw strings
        i += 1
        if i > 3:
            break
```

```
',No. of TB cases,,,TB incidence,,\n'
'U.S. jurisdiction,2019,2020,2021,2019,2020,2021\n'
'Total,"8,900","7,173","7,860",2.71,2.16,2.37\n'
'Alabama,87,72,92,1.77,1.43,1.83\n'
```

4. Finally, let's see the tried-and-true Data 100 approach: pandas.

```
tb_df = pd.read_csv("data/cdc_tuberculosis.csv")
tb_df.head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Unnamed: 0	No. of TB cases	Unnamed: 2	Unnamed: 3	TB incidence	Unnamed: 5	Unname
0	U.S. jurisdiction	2019	2020	2021	2019.00	2020.00	202
1	Total	8,900	7,173	7,860	2.71	2.16	
2	Alabama	87	72	92	1.77	1.43	
3	Alaska	58	58	58	7.91	7.92	
4	Arizona	183	136	129	2.51	1.89	

Wait, what's up with the "Unnamed" column names? And the first row, for that matter?

Congratulations – you're ready to wrangle your data. Because of how things are stored, we'll need to clean the data a bit to name our columns better.

A reasonable first step is to identify the row with the right header. The `pd.read_csv()` function (documentation) has the convenient `header` parameter:

```
tb_df = pd.read_csv("data/cdc_tuberculosis.csv", header=1) # row index
tb_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	2019	2020	2021	2019.1	2020.1	2021.1
0	Total	8,900	7,173	7,860	2.71	2.16	2.37
1	Alabama	87	72	92	1.77	1.43	1.83
2	Alaska	58	58	58	7.91	7.92	7.92
3	Arizona	183	136	129	2.51	1.89	1.77
4	Arkansas	64	59	69	2.12	1.96	2.28

Wait…but now we can't differentiate betwen the "Number of TB cases" and "TB incidence" year columns. pandas has tried to make our lives easier by automatically adding ".1" to the latter columns, but this doesn't help us as humans understand the data.

We can do this manually with `df.rename()` (documentation):

```
rename_dict = {'2019': 'TB cases 2019',
               '2020': 'TB cases 2020',
               '2021': 'TB cases 2021',
               '2019.1': 'TB incidence 2019',
               '2020.1': 'TB incidence 2020',
               '2021.1': 'TB incidence 2021'}
tb_df = tb_df.rename(columns=rename_dict)
tb_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Total	8,900	7,173	7,860	2.71	2
1	Alabama	87	72	92	1.77	1
2	Alaska	58	58	58	7.91	7
3	Arizona	183	136	129	2.51	1
4	Arkansas	64	59	69	2.12	1

6.2 Record Granularity

You might already be wondering: What's up with that first record?

Row 0 is what we call a **rollup record**, or summary record. It's often useful when displaying tables to humans. The **granularity** of record 0 (Totals) vs the rest of the records (States) is different.

Okay, EDA step two. How was the rollup record aggregated?

Let's check if Total TB cases is the sum of all state TB cases. If we sum over all rows, we should get **2x** the total cases in each of our TB cases by year (why?).

```
tb_df.sum(axis=0)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	0
U.S. jurisdiction	TotalAlabamaAlaskaArizonaArkansasCaliforniaCol...
TB cases 2019	8,9008758183642,11166671824558302997326108523 7...
TB cases 2020	7,1737258136591,70652541719412221928216923937 6...
TB cases 2021	7,8609258129691,75058544319499228106425512749 4...
TB incidence 2019	109.94
TB incidence 2020	93.09
TB incidence 2021	102.94

Whoa, what's going on? Check out the column types:

```
tb_df.dtypes
```

	0
U.S. jurisdiction	object
TB cases 2019	object
TB cases 2020	object
TB cases 2021	object
TB incidence 2019	float64
TB incidence 2020	float64
TB incidence 2021	float64

Looks like those commas are causing all TB cases to be read as the `object` datatype, or **storage type** (close to the Python string datatype), so pandas is concatenating strings instead of adding integers.

Fortunately `read_csv` also has a `thousands` parameter (https://pandas.pydata.org/docs/reference/api/pandas.r

```
# improve readability: chaining method calls with outer parentheses/line breaks
tb_df = (
    pd.read_csv("data/cdc_tuberculosis.csv", header=1, thousands=',')
    .rename(columns=rename_dict)
)
tb_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Total	8900	7173	7860	2.71	2
1	Alabama	87	72	92	1.77	1
2	Alaska	58	58	58	7.91	7
3	Arizona	183	136	129	2.51	1
4	Arkansas	64	59	69	2.12	1

```
tb_df.sum()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	0
U.S. jurisdiction	TotalAlabamaAlaskaArizonaArkansasCaliforniaCol...
TB cases 2019	17800
TB cases 2020	14346
TB cases 2021	15720
TB incidence 2019	109.94
TB incidence 2020	93.09
TB incidence 2021	102.94

The Total TB cases look right. Phew!

Let's just look at the records with **state-level granularity**:

```
state_tb_df = tb_df[1:]
state_tb_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2
1	Alabama	87	72	92	1.77	1
2	Alaska	58	58	58	7.91	7
3	Arizona	183	136	129	2.51	1
4	Arkansas	64	59	69	2.12	1
5	California	2111	1706	1750	5.35	4

6.3 Gather More Data: Census

U.S. Census population estimates source (2019), source (2020-2021).

Running the below cells cleans the data. There are a few new methods here: * `df.convert_dtypes()` (documentation) conveniently converts all float dtypes into ints and is out of scope for the class. * `df.drop_na()` (documentation) will be explained in more detail next time.

```
# 2010s census data
census_2010s_df = pd.read_csv("data/nst-est2019-01.csv", header=3, thousands=",")
census_2010s_df = (
    census_2010s_df
    .reset_index()
```

```
    .drop(columns=["index", "Census", "Estimates Base"])
    .rename(columns={"Unnamed: 0": "Geographic Area"})
    .convert_dtypes()                # "smart" converting of columns, use at your own ris
    .dropna()                        # we'll introduce this next time
)
census_2010s_df['Geographic Area'] = census_2010s_df['Geographic Area'].str.strip('.')

# with pd.option_context('display.min_rows', 30): # shows more rows
#     display(census_2010s_df)

census_2010s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Geographic Area	2010	2011	2012	2013	2014	2015	2016
0	United States	309321666	311556874	313830990	315993715	318301008	320635163	32294131
1	Northeast	55380134	55604223	55775216	55901806	56006011	56034684	5604233
2	Midwest	66974416	67157800	67336743	67560379	67745167	67860583	6798754
3	South	114866680	116006522	117241208	118364400	119624037	120997341	12235176
4	West	72100436	72788329	73477823	74167130	74925793	75742555	7655968

Occasionally, you will want to modify code that you have imported. To reimport those modifications you can either use the python importlib library:

```
from importlib import reload
reload(utils)
```

or use iPython magic which will intelligently import code when files change:

```
%load_ext autoreload
%autoreload 2
```

```
# census 2020s data
census_2020s_df = pd.read_csv("data/NST-EST2022-POP.csv", header=3, thousands=",")
census_2020s_df = (
    census_2020s_df
    .reset_index()
    .drop(columns=["index", "Unnamed: 1"])
    .rename(columns={"Unnamed: 0": "Geographic Area"})
    .convert_dtypes()                # "smart" converting of columns, use at your own ris
```

```
    .dropna()                           # we'll introduce this next time
)
census_2020s_df['Geographic Area'] = census_2020s_df['Geographic Area'].str.strip('.')

census_2020s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Geographic Area	2020	2021	2022
0	United States	331511512	332031554	333287557
1	Northeast	57448898	57259257	57040406
2	Midwest	68961043	68836505	68787595
3	South	126450613	127346029	128716192
4	West	78650958	78589763	78743364

6.4 Joining Data on Primary Keys

Time to `merge`! Here we use the DataFrame method `df1.merge(right=df2, ...)` on DataFrame `df1` (documentation). Contrast this with the function `pd.merge(left=df1, right=df2, ...)` (documentation). Feel free to use either.

```
# merge TB dataframe with two US census dataframes
tb_census_df = (
    tb_df
    .merge(right=census_2010s_df,
           left_on="U.S. jurisdiction", right_on="Geographic Area")
    .merge(right=census_2020s_df,
           left_on="U.S. jurisdiction", right_on="Geographic Area")
)
tb_census_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

This is a little unwieldy. We could either drop the unneeded columns now, or just merge on smaller census DataFrames. Let's do the latter.

```
# try merging again, but cleaner this time
tb_census_df = (
    tb_df
    .merge(right=census_2010s_df[["Geographic Area", "2019"]],
           left_on="U.S. jurisdiction", right_on="Geographic Area")
    .drop(columns="Geographic Area")
    .merge(right=census_2020s_df[["Geographic Area", "2020", "2021"]],
           left_on="U.S. jurisdiction", right_on="Geographic Area")
    .drop(columns="Geographic Area")
)
tb_census_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

6.5 Reproducing Data: Compute Incidence

Let's recompute incidence to make sure we know where the original CDC numbers came from.

From the CDC report: TB incidence is computed as "Cases per 100,000 persons using mid-year population estimates from the U.S. Census Bureau."

If we define a group as 100,000 people, then we can compute the TB incidence for a given state population as

$$\text{TB incidence} = \frac{\text{TB cases in population}}{\text{groups in population}} = \frac{\text{TB cases in population}}{\text{population}/100000}$$

$$= \frac{\text{TB cases in population}}{\text{population}} \times 100000$$

Let's try this for 2019:

```
tb_census_df["recompute incidence 2019"] = tb_census_df["TB cases 2019"]/tb_census_df["201
tb_census_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

Awesome!!!

Let's use a for-loop and Python format strings to compute TB incidence for all years. Python f-strings are just used for the purposes of this demo, but they're handy to know when you explore data beyond this course (Python documentation).

```
# recompute incidence for all years
for year in [2019, 2020, 2021]:
    tb_census_df[f"recompute incidence {year}"] = tb_census_df[f"TB cases {year}"]/tb_cens
tb_census_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 20
0	Alabama	87	72	92	1.77	1
1	Alaska	58	58	58	7.91	7
2	Arizona	183	136	129	2.51	1
3	Arkansas	64	59	69	2.12	1
4	California	2111	1706	1750	5.35	4

These numbers look pretty close!!! There are a few errors in the hundredths place, particularly in 2021. It may be useful to further explore reasons behind this discrepancy.

```
tb_census_df.describe()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB incid
count	51.000000	51.000000	51.000000	51.000000	51.000000	
mean	174.509804	140.647059	154.117647	2.102549	1.782941	
std	341.738752	271.055775	286.781007	1.498745	1.337414	
min	1.000000	0.000000	2.000000	0.170000	0.000000	
25%	25.500000	29.000000	23.000000	1.295000	1.210000	
50%	70.000000	67.000000	69.000000	1.800000	1.520000	
75%	180.500000	139.000000	150.000000	2.575000	1.990000	
max	2111.000000	1706.000000	1750.000000	7.910000	7.920000	

6.6 Bonus EDA: Reproducing the reported statistic

How do we reproduce that reported statistic in the original CDC report?

> Reported TB incidence (cases per 100,000 persons) increased **9.4%**, from **2.2** during 2020 to **2.4** during 2021 but was lower than incidence during 2019 (2.7). Increases occurred among both U.S.-born and non–U.S.-born persons.

This is TB incidence computed across the entire U.S. population! How do we reproduce this * We need to reproduce the "Total" TB incidences in our rolled record. * But our current `tb_census_df` only has 51 entries (50 states plus Washington, D.C.). There is no rolled record. * What happened…?

Let's get exploring!

Before we keep exploring, we'll set all indexes to more meaningful values, instead of just numbers that pertained to some row at some point. This will make our cleaning slightly easier.

```
tb_df = tb_df.set_index("U.S. jurisdiction")
tb_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
Total	8900	7173	7860	2.71	2.16
Alabama	87	72	92	1.77	1.43
Alaska	58	58	58	7.91	7.92
Arizona	183	136	129	2.51	1.89
Arkansas	64	59	69	2.12	1.96

```
census_2010s_df = census_2010s_df.set_index("Geographic Area")
census_2010s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	
United States	309321666	311556874	313830990	315993715	318301008	320635163	322941311	3
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	1
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	

```
census_2020s_df = census_2020s_df.set_index("Geographic Area")
census_2020s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Geographic Area	2020	2021	2022
United States	331511512	332031554	333287557
Northeast	57448898	57259257	57040406
Midwest	68961043	68836505	68787595
South	126450613	127346029	128716192
West	78650958	78589763	78743364

It turns out that our merge above only kept state records, even though our original `tb_df` had the "Total" rolled record:

```
tb_df.head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
Total	8900	7173	7860	2.71	2.16
Alabama	87	72	92	1.77	1.43
Alaska	58	58	58	7.91	7.92
Arizona	183	136	129	2.51	1.89
Arkansas	64	59	69	2.12	1.96

Recall that merge by default does an **inner** merge by default, meaning that it only preserves keys that are present in **both** DataFrames.

The rolled records in our census dataframes have different `Geographic Area` fields, which was the key we merged on:

```
census_2010s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	
United States	309321666	311556874	313830990	315993715	318301008	320635163	322941311	3
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	1
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	

The Census DataFrame has several rolled records. The aggregate record we are looking for actually has the Geographic Area named "United States".

One straightforward way to get the right merge is to rename the value itself. Because we now have the Geographic Area index, we'll use `df.rename()` (documentation):

```
# rename rolled record for 2010s
census_2010s_df.rename(index={'United States':'Total'}, inplace=True)
census_2010s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	
Total	309321666	311556874	313830990	315993715	318301008	320635163	322941311	3
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	1
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	

```
# same, but for 2020s rename rolled record
census_2020s_df.rename(index={'United States':'Total'}, inplace=True)
census_2020s_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

Geographic Area	2020	2021	2022
Total	331511512	332031554	333287557
Northeast	57448898	57259257	57040406
Midwest	68961043	68836505	68787595
South	126450613	127346029	128716192
West	78650958	78589763	78743364

Next let's rerun our merge. Note the different chaining, because we are now merging on indexes (`df.merge()` documentation).

```
tb_census_df = (
    tb_df
    .merge(right=census_2010s_df[["2019"]],
           left_index=True, right_index=True)
    .merge(right=census_2020s_df[["2020", "2021"]],
           left_index=True, right_index=True)
)
tb_census_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB in
Total	8900	7173	7860	2.71	2.16	
Alabama	87	72	92	1.77	1.43	
Alaska	58	58	58	7.91	7.92	
Arizona	183	136	129	2.51	1.89	
Arkansas	64	59	69	2.12	1.96	

Finally, let's recompute our incidences:

```
# recompute incidence for all years
for year in [2019, 2020, 2021]:
    tb_census_df[f"recompute incidence {year}"] = tb_census_df[f"TB cases {year}"]/tb_cens
tb_census_df.head(5)
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB in
Total	8900	7173	7860	2.71	2.16	
Alabama	87	72	92	1.77	1.43	
Alaska	58	58	58	7.91	7.92	
Arizona	183	136	129	2.51	1.89	
Arkansas	64	59	69	2.12	1.96	

We reproduced the total U.S. incidences correctly!

We're almost there. Let's revisit the quote:

> Reported TB incidence (cases per 100,000 persons) increased **9.4%**, from **2.2** during 2020 to **2.4** during 2021 but was lower than incidence during 2019 (2.7). Increases occurred among both U.S.-born and non–U.S.-born persons.

Recall that percent change from A to B is computed as percent change $= \frac{B-A}{A} \times 100$.

```
incidence_2020 = tb_census_df.loc['Total', 'recompute incidence 2020']
incidence_2020
```

```
2.1637257652759883
```

```
incidence_2021 = tb_census_df.loc['Total', 'recompute incidence 2021']
incidence_2021
```

```
2.3672448914298068
```

```
difference = (incidence_2021 - incidence_2020)/incidence_2020 * 100
difference
```

```
9.405957511804143
```

7 Regular Expressions

> **Note**
>
> - Understand Python string manipulation, Pandas Series methods
> - Parse and create regex, with a reference table
> - Use vocabulary (closure, metacharater, groups, etc.) to describe regex metacharacters

7.1 Why Work with Text?

Last lecture, we learned of the difference between quantitative and qualitative variable types. The latter includes string data - the primary focus of today's lecture. In this note, we'll discuss the necessary tools to manipulate text: Python string manipulation and regular expressions.

There are two main reasons for working with text.

1. Canonicalization: Convert data that has multiple formats into a standard form.
    - By manipulating text, we can join tables with mismatched string labels
2. Extract information into a new feature.
    - For example, we can extract date and time features from text

7.2 Python String Methods

First, we'll introduce a few methods useful for string manipulation. The following table includes a number of string operations supported by Python and `pandas`. The Python functions operate on a single string, while their equivalent in `pandas` are **vectorized** - they operate on a Series of string data.

Operation	Python	Pandas (Series)
Transformation	• `s.lower(_)` • `s.upper(_)`	• `ser.str.lower(_)` • `ser.str.upper(_)`

Operation	Python	Pandas (Series)
Replacement + Deletion	• `s.replace(_)`	• `ser.str.replace(_)`
Split	• `s.split(_)`	• `ser.str.split(_)`
Substring	• `s[1:4]`	• `ser.str[1:4]`
Membership	• `'_' in s`	• `ser.str.contains(_)`
Length	• `len(s)`	• `ser.str.len()`

We'll discuss the differences between Python string functions and `pandas` Series methods in the following section on canonicalization.

7.2.1 Canonicalization

Assume we want to merge the given tables.

```
import pandas as pd

with open('data/county_and_state.csv') as f:
    county_and_state = pd.read_csv(f)

with open('data/county_and_population.csv') as f:
    county_and_pop = pd.read_csv(f)
```

```
display(county_and_state), display(county_and_pop);
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LS

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044

Last time, we used a **primary key** and **foreign key** to join two tables. While neither of these keys exist in our DataFrames, the `County` columns look similar enough. Can we convert these columns into one standard, canonical form to merge the two tables?

7.2.1.1 Canonicalization with Python String Manipulation

The following function uses Python string manipulation to convert a single county name into canonical form. It does so by eliminating whitespace, punctuation, and unnecessary text.

```
def canonicalize_county(county_name):
    return (
        county_name
            .lower()
            .replace(' ', '')
            .replace('&', 'and')
            .replace('.', '')
            .replace('county', '')
            .replace('parish', '')
    )

canonicalize_county("St. John the Baptist")
```

```
'stjohnthebaptist'
```

We will use the `pandas map` function to apply the `canonicalize_county` function to every row in both DataFrames. In doing so, we'll create a new column in each called `clean_county_python` with the canonical form.

```
county_and_pop['clean_county_python'] = county_and_pop['County'].map(canonicalize_county)
county_and_state['clean_county_python'] = county_and_state['County'].map(canonicalize_coun
```

```
display(county_and_state), display(county_and_pop);
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	County	State	clean_county_python
0	De Witt County	IL	dewitt
1	Lac qui Parle County	MN	lacquiparle
2	Lewis and Clark County	MT	lewisandclark
3	St John the Baptist Parish	LS	stjohnthebaptist

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	County	Population	clean_county_python
0	DeWitt	16798	dewitt
1	Lac Qui Parle	8067	lacquiparle
2	Lewis & Clark	55716	lewisandclark
3	St. John the Baptist	43044	stjohnthebaptist

7.2.1.2 Canonicalization with Pandas Series Methods

Alternatively, we can use `pandas` Series methods to create this standardized column. To do so, we must call the `.str` attribute of our Series object prior to calling any methods, like `.lower` and `.replace`. Notice how these method names match their equivalent built-in Python string functions.

Chaining multiple Series methods in this manner eliminates the need to use the `map` function (as this code is vectorized).

```
def canonicalize_county_series(county_series):
    return (
        county_series
            .str.lower()
            .str.replace(' ', '')
            .str.replace('&', 'and')
            .str.replace('.', '')
            .str.replace('county', '')
            .str.replace('parish', '')
    )

county_and_pop['clean_county_pandas'] = canonicalize_county_series(county_and_pop['County'
```

```
county_and_state['clean_county_pandas'] = canonicalize_county_series(county_and_state['Cou
```

```
/var/folders/7x/09tm3ct91s14kpwnpd0400k00000gn/T/ipykernel_5892/837021704.py:7: FutureWarning
  .str.replace('.', '')
/var/folders/7x/09tm3ct91s14kpwnpd0400k00000gn/T/ipykernel_5892/837021704.py:7: FutureWarning
  .str.replace('.', '')
```

```
display(county_and_pop), display(county_and_state);
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatter
  return method()
```

	County	Population	clean_county_python	clean_county_pandas
0	DeWitt	16798	dewitt	dewitt
1	Lac Qui Parle	8067	lacquiparle	lacquiparle
2	Lewis & Clark	55716	lewisandclark	lewisandclark
3	St. John the Baptist	43044	stjohnthebaptist	stjohnthebaptist

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatter
  return method()
```

	County	State	clean_county_python	clean_county_pandas
0	De Witt County	IL	dewitt	dewitt
1	Lac qui Parle County	MN	lacquiparle	lacquiparle
2	Lewis and Clark County	MT	lewisandclark	lewisandclark
3	St John the Baptist Parish	LS	stjohnthebaptist	stjohnthebaptist

7.2.2 Extraction

Extraction explores the idea of obtaining useful information from text data. This will be particularily important in model building, which we'll study in a few weeks.

Say we want to read some data from a `.txt` file.

```
with open('data/log.txt', 'r') as f:
    log_lines = f.readlines()
```

```
log_lines
```

```
['169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585
 '193.205.203.3 - - [2/Feb/2005:17:23:6 -0800] "GET /stat141/Notes/dim.html HTTP/1.0" 404 302
 '169.237.46.240 - "" [3/Feb/2006:10:18:37 -0800] "GET /stat141/homework/Solutions/hw1Sol.pdf
```

Suppose we want to extract the day, month, year, hour, minutes, seconds, and timezone. Unfortunately, these items are not in a fixed position from the beginning of the string, so slicing by some fixed offset won't work.

Instead, we can use some clever thinking. Notice how the relevant information is contained within a set of brackets, further seperated by `/` and `:`. We can hone in on this region of text, and split the data on these characters. Python's built-in `.split` function makes this easy.

```
first = log_lines[0] # Only considering the first row of data

pertinent = first.split("[")[1].split(']')[0]
day, month, rest = pertinent.split('/')
year, hour, minute, rest = rest.split(':')
seconds, time_zone = rest.split(' ')
day, month, year, hour, minute, seconds, time_zone
```

```
('26', 'Jan', '2014', '10', '47', '58', '-0800')
```

There are two problems with this code:

1. Python's built-in functions limit us to extract data one record at a time
    - This can be resolved using a map function or Pandas Series methods.
2. The code is quite verbose
    - This is a larger issue that is trickier to solve

In the next section, we'll introduce regular expressions - a tool that solves problem 2.

7.3 Regex Basics

A **regular expression ("regex")** is a sequence of characters that specifies a search pattern. They are written to extract specific information from text. Regular expressions are essentially

part of a smaller programming language embedded in Python, made available through the `re` module. As such, they have a stand-alone syntax and methods for various capabilities.

Regular expressions are useful in many applications beyond data science. For example, Social Security Numbers (SSNs) are often validated with regular expresions.

```
r"[0-9]{3}-[0-9]{2}-[0-9]{4}" # Regular Expression Syntax

# 3 of any digit, then a dash,
# then 2 of any digit, then a dash,
# then 4 of any digit
```

```
'[0-9]{3}-[0-9]{2}-[0-9]{4}'
```

There are a ton of resources to learn and experiment with regular expressions. A few are provided below:

- Official Regex Guide
- Data 100 Reference Sheet
- Regex101.com
  - Be sure to check `Python` under the category on the left.

7.3.1 Basics Regex Syntax

There are four basic operations with regular expressions.

Operation	Order	Syntax Example	Matches	Doesn't Match
`Or: |`	4	AA|BAAB	AA BAAB	every other string
`Concatenation`	3	AABAAB	AABAAB	every other string
`Closure: *` (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
`Group: ()` (parenthesis)	1	A(A|B)AAB (AB)*A	AAAAB ABAAB A ABABABABA	every other string AA ABBA

Notice how these metacharacter operations are ordered. Rather than being literal characters, these **metacharacters** manipulate adjacent characters. `()` takes precedence, followed by `*`,

and finally `|`. This allows us to differentiate between very different regex commands like `AB*` and `(AB)*`. The former reads "`A` then zero or more copies of `B`", while the latter specifies "zero or more copies of `AB`".

7.3.1.1 Examples

Question 1: Give a regular expression that matches `moon`, `moooon`, etc. Your expression should match any even number of `o`s except zero (i.e. don't match `mn`).

Answer 1: `moo(oo)*n`

- Hardcoding `oo` before the capture group ensures that `mn` is not matched.
- A capture group of `(oo)*` ensures the number of `o`'s is even.

Question 2: Using only basic operations, formulate a regex that matches `muun`, `muuuun`, `moon`, `moooon`, etc. Your expression should match any even number of `u`s or `o`s except zero (i.e. don't match `mn`).

Answer 2: `m(uu(uu)*|oo(oo)*)n`

- The leading `m` and trailing `n` ensures that only strings beginning with `m` and ending with `n` are matched.
- Notice how the outer capture group surrounds the `|`.
  - Consider the regex `m(uu(uu)*)|(oo(oo)*)n`. This incorrectly matches `muu` and `oooon`.
    - Each OR clause is everything to the left and right of `|`. The incorrect solution matches only half of the string, and ignores either the beginning `m` or trailing `n`.
    - A set of paranthesis must surround `|`. That way, each OR clause is everything to the left and right of `|` **within** the group. This ensures both the beginning `m` *and* trailing `n` are matched.

7.4 Regex Expanded

Provided below are more complex regular expression functions.

Operation	Syntax Example	Matches	Doesn't Match
`Any Character`: `.` (except newline)	.U.U.U.	CUMULUS JUGULUM	SUCCUBUS TUMUL- TUOUS

Operation	Syntax Example	Matches	Doesn't Match
`Character Class: []` (match one character in `[]`)	[A-Za-z][a-z]*	word Capitalized	camelCase 4illegal
`Repeated "a" Times: {a}`	j[aeiou]{3}hn	jaoehn jooohn	jhn jaeiouhn
`Repeated "from a to b" Times: {a, b}`	j[0u]{1,2}hn	john juohn	jhn jooohn
`At Least One: +`	jo+hn	john jooooooohn	jhn jjohn
`Zero or One: ?`	joh?n	jon john	any other string

A character class matches a single character in it's class. These characters can be hardcoded – in the case of `[aeiou]` – or shorthand can be specified to mean a range of characters. Examples include:

1. `[A-Z]`: Any capitalized letter
2. `[a-z]`: Any lowercase letter
3. `[0-9]`: Any single digit
4. `[A-Za-z]`: Any capitalized of lowercase letter
5. `[A-Za-z0-9]`: Any capitalized or lowercase letter or single digit

7.4.0.1 Examples

Let's analyze a few examples of complex regular expressions.

Matches	Does Not Match
1. `.*SPB.*` RASPBERRY SPBOO	SUBSPACE SUBSPECIES
2. `[0-9]{3}-[0-9]{2}-[0-9]{4}` 231-41-5121 573-57-1821	231415121 57-3571821
3. `[a-z]+@([a-z]+\.)+(edu|com)` horse@pizza.com horse@pizza.food.com	frank_99@yahoo.com hug@cs

Explanations

1. `.*SPB.*` only matches strings that contain the substring `SPB`.

- The `.*` metacharacter matches any amount of non-negative characters. Newlines do not count.

2. This regular expression matches 3 of any digit, then a dash, then 2 of any digit, then a dash, then 4 of any digit
    - You'll recognize this as the familiar Social Security Number regular expression
3. Matches any email with a `com` or `edu` domain, where all characters of the email are letters.
    - At least one `.` must preceed the domain name. Including a backslash `\` before any metacharacter (in this case, the `.`) tells regex to match that character exactly.

7.5 Convenient Regex

Here are a few more convenient regular expressions.

Operation	Syntax Example	Matches	Doesn't Match
`built in character class`	`\w+ \d+ \s+`	Fawef_03 231123 `whitespace`	this person 423 people `non-whitespace`
`character class negation: [^]` (everything except the given characters)	[^a-z]+.	PEPPERS3982 17211!↑å	porch CLAmS
`escape character: \` (match the literal next character)	cow\.com	cow.com	cowscom
`beginning of line: ^`	^ark	ark two ark o ark	dark
`end of line: $`	ark$	dark ark o ark	ark two
`lazy version of zero or more : *?`	5.*?5	5005 55	5005005

7.5.0.1 Examples

Let's revist our earlier problem of extracting date/time data from the given `.txt` files. Here is how the data looked.

```
log_lines[0]
```

```
'169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 '
```

Question: Give a regular expression that matches everything contained within and including the brackets - the day, month, year, hour, minutes, seconds, and timezone.

Answer: `\[.*\]`

- Notice how matching the literal `[` and `]` is necessary. Therefore, an escape character `\` is required before both `[` and `]` - otherwise these metacharacters will match character classes.
- We need to match a particular format between `[` and `]`. For this example, `.*` will suffice.

Alternative Solution: `\[\w+/\w+/\w+:\w+:\w+:\w+\s-\w+\]`

- This solution is much safer.
  - Imagine the data between `[` and `]` was garbage - `.*` will still match that.
  - The alternate solution will only match data that follows the correct format.

7.6 Regex in Python and Pandas (Regex Groups)

7.6.1 Canonicalization

7.6.1.1 Canonicalization with Regex

Earlier in this note, we examined the process of canonicalization using Python string manipulation and `pandas` Series methods. However, we mentioned this approach had a major flaw: our code was unnecessarily verbose. Equipped with our knowledge of regular expressions, let's fix this.

To do so, we need to understand a few functions in the `re` module. The first of these is the substitute function: `re.sub(pattern, rep1, text)`. It behaves similarily to Python's built-in `.replace` function, and returns text with all instances of `pattern` replaced by `rep1`.

The regular expression here removes text surrounded by <> (also known as HTML tags).

In order, the pattern matches … 1. a single < 2. any character that is not a > : div, td valign…, /td, /div 3. a single >

Any substring in `text` that fulfills all three conditions will be replaced by `''`.

```
import re

text = "<div><td valign='top'>Moo</td></div>"
pattern = r"<[^>]+>"
```

```
re.sub(pattern, '', text)
```

```
'Moo'
```

Notice the `r` preceeding the regular expression pattern; this specifies the regular expression is a raw string. Raw strings do not recognize escape sequences (ie the Python newline metacharacter `\n`). This makes them useful for regular expressions, which often contain literal `\` characters.

In other words, don't forget to tag your regex with a `r`.

7.6.1.2 Canonicalization with Pandas

We can also use regular expressions with Pandas Series methods. This gives us the benefit of operating on an entire column of data as opposed to a single value. The code is simple: `ser.str.replace(pattern, repl, regex=True)`.

Consider the following DataFrame `html_data` with a single column.

```
data = {"HTML": ["<div><td valign='top'>Moo</td></div>", \
                 "<a href='http://ds100.org'>Link</a>", \
                 "<b>Bold text</b>"]}
html_data = pd.DataFrame(data)
```

```
html_data
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	HTML
0	<td valign='top'>Moo</td>
1	Link
2	Bold text

```
pattern = r"<[^>]+>"
html_data['HTML'].str.replace(pattern, '', regex=True)
```

	HTML
0	Moo
1	Link
2	Bold text

7.6.2 Extraction

7.6.2.1 Extraction with Regex

Just like with canonicalization, the `re` module provides capability to extract relevant text from a string: `re.findall(pattern, text)`. This function returns a list of all matches to `pattern`.

Using the familiar regular expression for Social Security Numbers:

```
text = "My social security number is 123-45-6789 bro, or maybe it's 321-45-6789."
pattern = r"[0-9]{3}-[0-9]{2}-[0-9]{4}"
re.findall(pattern, text)
```

```
['123-45-6789', '321-45-6789']
```

7.6.2.2 Extraction with Pandas

Pandas similarily provides extraction functionality on a Series of data: `ser.str.findall(pattern)`

Consider the following DataFrame `ssn_data`.

```
data = {"SSN": ["987-65-4321", "forty", \
                "123-45-6789 bro or 321-45-6789",
               "999-99-9999"]}
ssn_data = pd.DataFrame(data)
```

```
ssn_data
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	SSN
0	987-65-4321
1	forty
2	123-45-6789 bro or 321-45-6789
3	999-99-9999

```
ssn_data["SSN"].str.findall(pattern)
```

	SSN
0	[987-65-4321]
1	[]
2	[123-45-6789, 321-45-6789]
3	[999-99-9999]

This function returns a list for every row containing the pattern matches in a given string.

7.6.3 Regular Expression Capture Groups

Earlier we used parentheses `( )` to specify the highest order of operation in regular expressions. However, they have another meaning; paranthesis are often used to represent **capture groups**. Capture groups are essentially, a set of smaller regular expressions that match multiple substrings in text data.

Let's take a look at an example.

7.6.3.1 Example 1

```
text = "Observations: 03:04:53 - Horse awakens. \
        03:05:14 - Horse goes back to sleep."
```

Say we want to capture all occurences of time data (hour, minute, and second) as *seperate entities*.

```
pattern_1 = r"(\d\d):(\d\d):(\d\d)"
re.findall(pattern_1, text)
```

```
[('03', '04', '53'), ('03', '05', '14')]
```

Notice how the given pattern has 3 capture groups, each specified by the regular expression `(\d\d)`. We then use `re.findall` to return these capture groups, each as tuples containing 3 matches.

These regular expression capture groups can be different. We can use the `(\d{2})` shorthand to extract the same data.

```
pattern_2 = r"(\d\d):(\d\d):(\d{2})"
re.findall(pattern_2, text)
```

```
[('03', '04', '53'), ('03', '05', '14')]
```

7.6.3.2 Example 2

With the notion of capture groups, convince yourself how the following regular expression works.

```
first = log_lines[0]
first
```

```
'169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 '
```

```
pattern = r'\[(\d+)\/(\w+)\/(\d+):(\d+):(\d+):(\d+) (.+)\]'
day, month, year, hour, minute, second, time_zone = re.findall(pattern, first)[0]
print(day, month, year, hour, minute, second, time_zone)
```

```
26 Jan 2014 10 47 58 -0800
```

7.7 Limitations of Regular Expressions

Today, we explored the capabilities of regular expressions in data wrangling with text data. However, there are a few things to be wary of.

Writing regular expressions is like writing a program.

- Need to know the syntax well.
- Can be easier to write than to read.
- Can be difficult to debug.

Regular expressions are terrible at certain types of problems:

- For parsing a hierarchical structure, such as JSON, use the `json.load()` parser, not regex!
- Complex features (e.g. valid email address).
- Counting (same number of instances of a and b). (impossible)
- Complex properties (palindromes, balanced parentheses). (impossible)

8 Visualization I

> **Note**
>
> - Use `matplotlib`, `seaborn`, and `plotly` to create data visualization.
> - Analyze histogram and identify outliers, mode, and skewness.
> - Using `boxplot` and `violinplot` to compare two distributions.

In our journey of the data science lifecycle, we have begun to explore the vast world of exploratory data analysis. More recently, we learned how to pre-process data using various data manipulation techniques. As we work towards understanding our data, there is one key component missing in our arsenal - the ability to visualize and discern relationships in existing data.

These next two lectures will introduce you to various examples of data visualizations and their underlying theory. In doing so, we'll motivate their importace in real-world examples with the use of plotting libraries.

8.1 Visualizations in Data 8 and Data 100 (so far)

You've likely encountered several forms of data visualizations in your studies. You may remember two such examples from Data 8: line charts and histograms. Each of these served a unique purpose. For example, line charts displayed how numerical quantities changed over time, while histograms were useful in understanding a variable's distribution.

Line Chart

Histogram

8.2 Goals of Visualization

Visualizations are useful for a number of reasons. In Data 100, we consider two areas in particular:

1. To broaden your understanding of the data

- Key part in exploratory data analysis.
- Useful in investigating relationships between variables.

2. To communicate results/conclusions to others
    - Visualization theory is especially important here.

One of the most common applications of visualizations is in understanding a distribution of data.

8.3 An Overview of Distributions

A distribution describes the frequency of unique values in a variable. Distributions must satisfy two properties:

1. Each data point must belong to only one category.
2. The total frequency of all categories must sum to 100%. In other words, their total count should equal the number of values in consideration.

Not a Valid Distribution

Valid Distribution

Left Diagram: This is not a valid distribution since individuals can be associated to more than one category and the bar values demonstrate values in minutes and not probability

Right Diagram: This example satisfies the two properties of distributions, so it is a valid distribution.

8.4 Bar Plots

As we saw above, a **bar plot** is one of the most common ways of displaying the distribution of a **qualitative** (categorical) variable. The length of a bar plot encodes the frequency of a category; the width encodes no useful information.

Let's contextualize this in an example. We will use the familiar `births` dataset from Data 8 in our analysis.

```
import pandas as pd

births = pd.read_csv("data/baby.csv")
births.head(5)
```

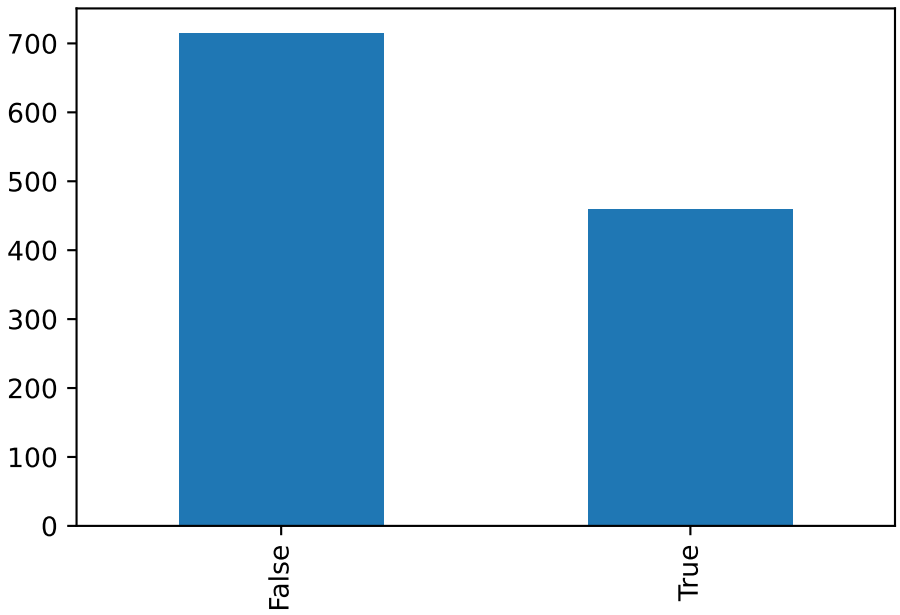
```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Mate
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
3	108	282	23	67	125	True
4	136	286	25	62	93	False

We can visualize the distribution of the `Maternal Smoker` column using a bar plot. There are a few ways to do this.

8.4.1 Plotting in Pandas

```
births['Maternal Smoker'].value_counts().plot(kind = 'bar');
```



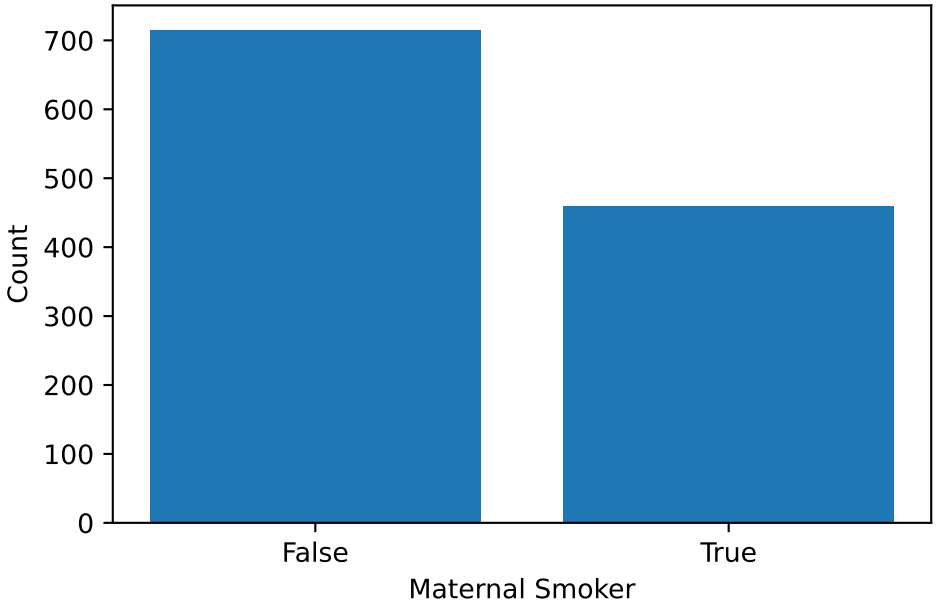
Recall that `.value_counts()` returns a `Series` with the total count of each unique value. We call `.plot(kind = 'bar')` on this result to visualize these counts as a bar plot.

Plotting methods in `pandas` are the least preferred and not supported in Data 100, as their functionality is limited. Instead, future examples will focus on other libaries built specifically for visualizing data. The most well-known library here is `matplotlib`.

8.4.2 Plotting in Matplotlib

```
import matplotlib.pyplot as plt

ms = births['Maternal Smoker'].value_counts()
plt.bar(ms.index.astype('string'), ms)
plt.xlabel('Maternal Smoker')
plt.ylabel('Count');
```

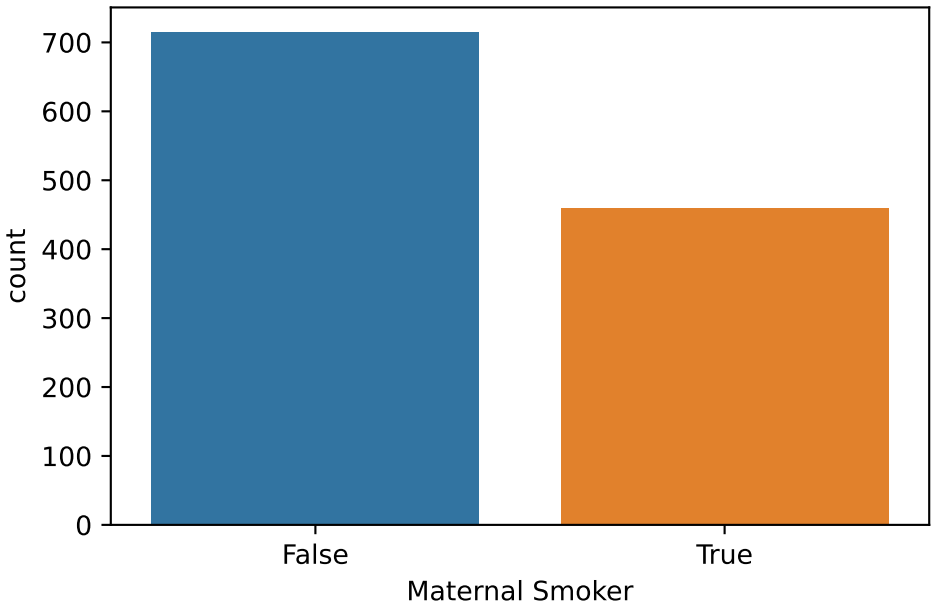


While more code is required to achieve the same result, `matplotlib` is often used over `pandas` for its ability to plot more complex visualizations, some of which are discussed shortly.

However, notice how we need to explicitly specify the type of the value for the x-axis to `string`. In absence of conversion, the x-axis will be a range of integers rather than the two categories, `True` and `False`. This is because `matplotlib` coerces `True` to a value of 1 and `False` to 0. Also, note how we needed to label the axes with `plt.xlabel` and `plt.ylabel` - `matplotlib` does not support automatic axis labeling. To get around these inconveniences, we can use a more effecient plotting library, `seaborn`.

8.4.3 Plotting in Seaborn

```
import seaborn as sns
sns.countplot(data = births, x = 'Maternal Smoker');
```



`seaborn.countplot` both counts and visualizes the number of unique values in a given column. This column is specified by the `x` argument to `sns.countplot`, while the `DataFrame` is specified by the `data` argument.

For the vast majority of visualizations, `seaborn` is far more concise and aesthetically pleasing than `matplotlib`. However, the color scheme of this particular bar plot is abritrary - it encodes no additional information about the categories themselves. This is not always true; color may signify meaningful detail in other visualizations. We'll explore this more in-depth during the next lecture.

8.4.4 Plotting in Plotly

`plotly` is one of the most versatile plottling libraries and widely used in industry. However, `plotly` has various dependencies that make it difficult to support in Data 100. Therfore, we have intentionally excluded the code to generate the plot above.

By now, you'll have noticed that each of these plotting libraries have a very different syntax. As with `pandas`, we'll teach you the important methods in `matplotlib` and `seaborn`, but you'll learn more through documentation.

1. Matplotlib Documentation
2. Seaborn Documentation

Example Questions:

- What colors should we use?
- How wide should the bars be?
- Should the legend exist?
- Should the bars and axes have dark borders?

To accomplish goal 2, here are some ways we can improve plot:

- Introducing different colors for each bar
- Including a legend
- Including a title
- Labeling the y-axis
- Using color-blind friendly palettes
- Re-orienting the labels
- Increase the font size

8.5 Histograms

Histograms are a natural extension to bar plots; they visualize the distribution of **quantitative** (numerical) data.

Revisiting our example with the `births` DataFrame, let's plot the distribution of the `Maternal Pregnancy Weight` column.

```
births.head(5)
```

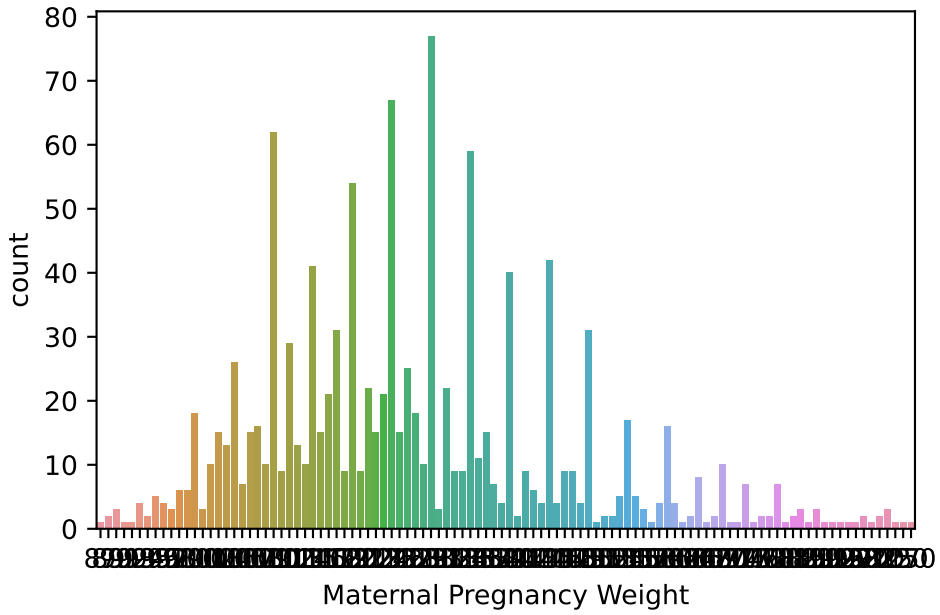
```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	Birth Weight	Gestational Days	Maternal Age	Maternal Height	Maternal Pregnancy Weight	Mate
0	120	284	27	62	100	False
1	113	282	33	64	135	False
2	128	279	28	64	115	True
3	108	282	23	67	125	True
4	136	286	25	62	93	False

How should we define our categories for this variable? In the previous example, these were the unique values of the `Maternal Smoker` column: `True` and `False`. If we use similar logic here, our categories are the different numerical weights contained in the `Maternal Pregnancy Weight` column.

Under this assumption, let's plot this distribution using the `seaborn.countplot` function.

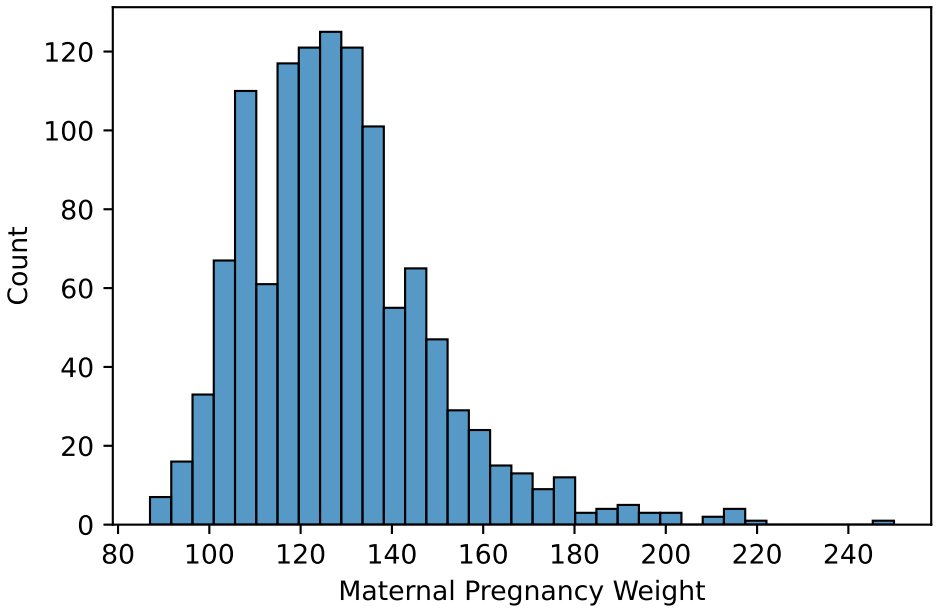
```
sns.countplot(data = births, x = 'Maternal Pregnancy Weight');
```



This histogram clearly suffers from **overplotting**. This is somewhat expected for `Maternal Pregnancy Weight` - it is a quantitative variable that takes on a wide range of values.

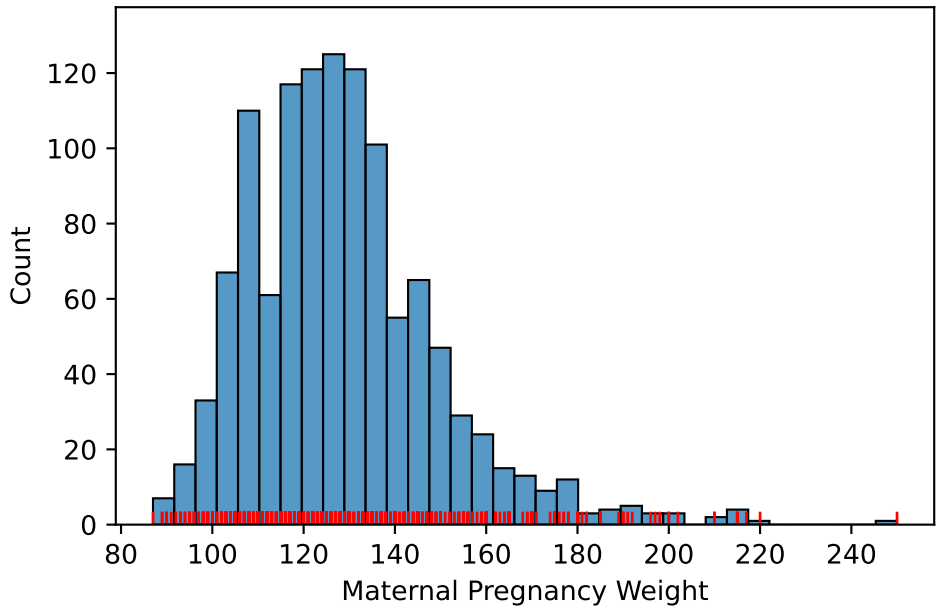
To combat this problem, statisticians use bins to categorize numerical data. Luckily, `seaborn` provides a helpful plotting function that automatically bins our data.

```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
```



This diagram is known as a histogram. While it looks more reasonable, notice how we lose fine-grain information on the distribution of data contained within each bin. We can introduce rug plots to minimize this information loss. An overlaid "rug plot" displays the within-bin distribution of our data, as denoted by the thickness of the colored line on the x-axis.

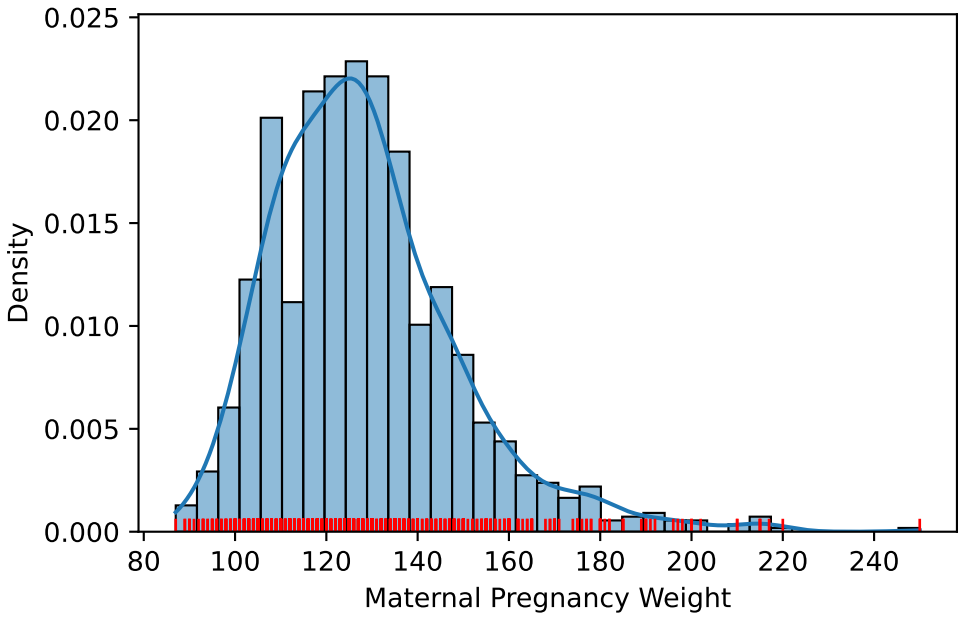
```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
sns.rugplot(data = births, x = 'Maternal Pregnancy Weight', color = 'red');
```



You may have seen histograms drawn differently - perhaps with an overlaid **density curve** and normalized y-axis. We can display both with a few tweaks to our code.

To visualize a density curve, we can set the the `kde = True` argument of the `sns.histplot`. Setting the argument `stat = 'density'` normalizes our histogram and displays densities, instead of counts, on the y-axis. You'll notice that the area under the density curve is 1.

```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight', kde = True,
             stat = 'density')
sns.rugplot(data = births, x = 'Maternal Pregnancy Weight', color = 'red');
```



8.6 Evaluating Histograms

Histograms allow us to assess a distribution by their shape. There are a few properties of histograms we can analyze:

1. Skewness and Tails
    - Skewed left vs skewed right
    - Left tail vs right tail
2. Outliers
    - Defined arbitrarily for now
3. Modes
    - Most commonly occuring data

8.6.1 Skewness and Tails

If a distribution has a long right tail (such as `Maternal Pregancy Weight`), it is **skewed right**. In a right-skewed distribution, the few large outliers "pull" the mean to the **right** of the median.

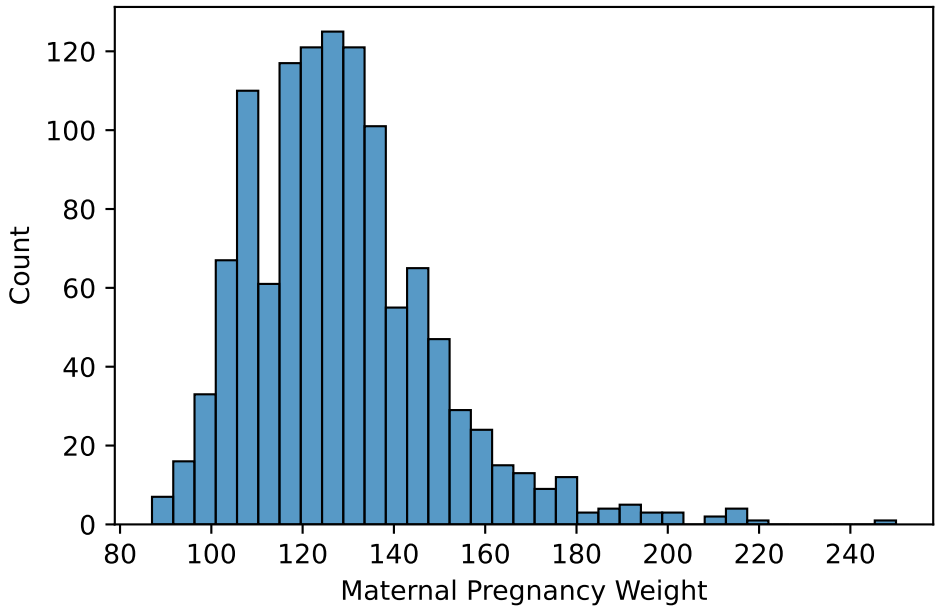
If a distribution has a long left tail, it is **skewed left**. In a left-skewed distribution, the few small outliers "pull" the mean to the **left** of the median.

In the case where a distribution has equal-sized right and left tails, it is **symmetric**. The mean is approximately **equal** to the median. Think of mean as the balancing point of the distribution

```
import numpy as np

sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
df_mean = np.mean(births['Maternal Pregnancy Weight'])
df_median = np.median(births['Maternal Pregnancy Weight'])
print("The mean is: {} and the median is {}".format(df_mean,df_median))
```

```
The mean is: 128.4787052810903 and the median is 125.0
```



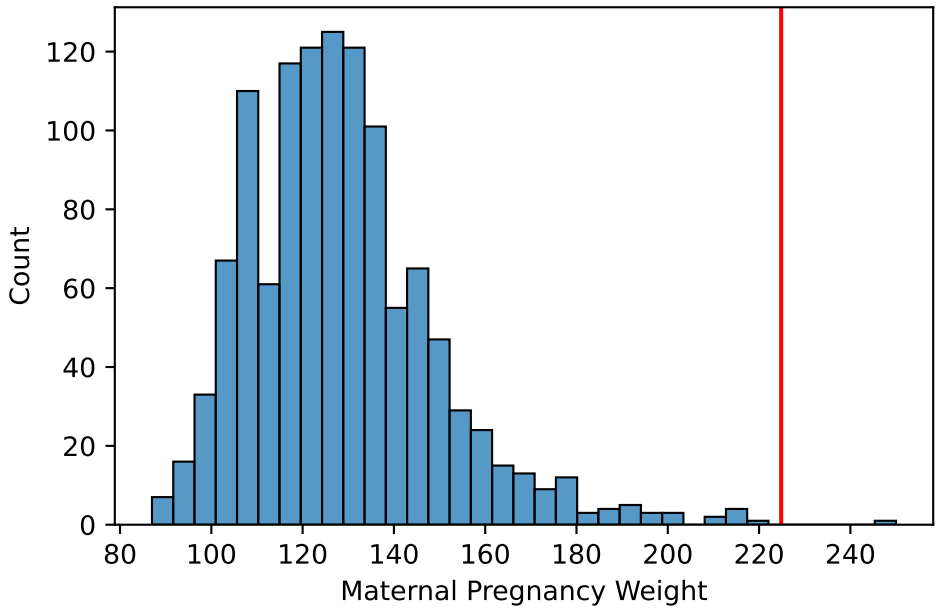
8.6.2 Outliers

Loosely speaking, an **outlier** is defined as a data point that lies an abnormally large distance away from other values. We'll define the statistical measure for this shortly.

Outliers disproportionately influce the mean because their magnitude is directly involved in computing the average. However, the median is largely unaffected - the magnitude of an outlier

is irrelevant; we only care that it is some non-zero distance away from the midpoint of the data.

```
sns.histplot(data = births, x = 'Maternal Pregnancy Weight');
## Where do we draw the line of outlier?
plt.axvline(df_mean*1.75, color = 'red');
```

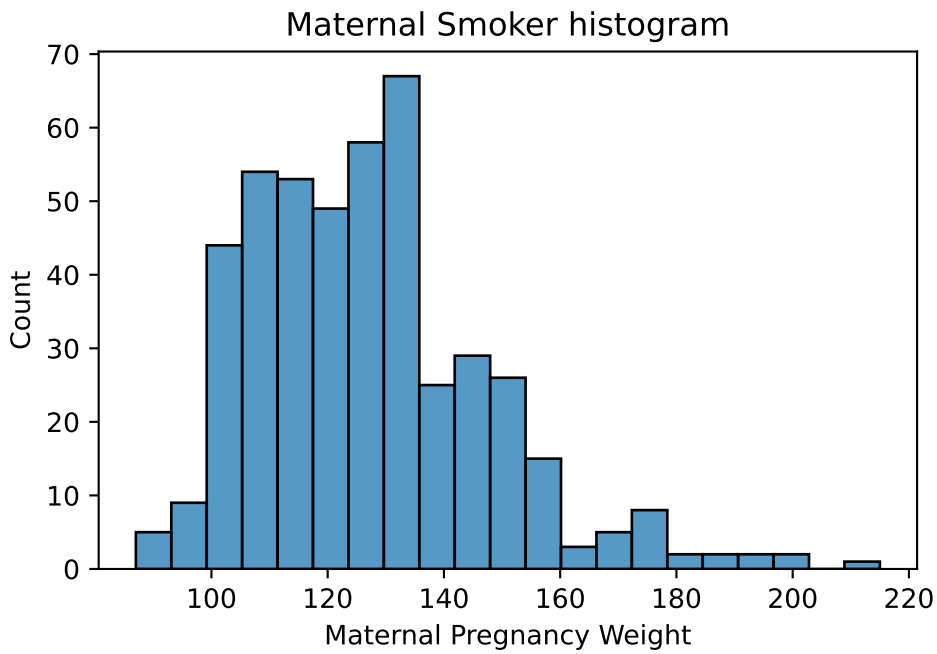


8.6.3 Modes

A **mode** of a distribution is a local or global maximum. A distribution with a single clear maximum is **unimodal**, distributions with two modes are **bimodal**, and those with 3 or more are **multimodal**. You need to distinguish between **modes** and *random noise*.

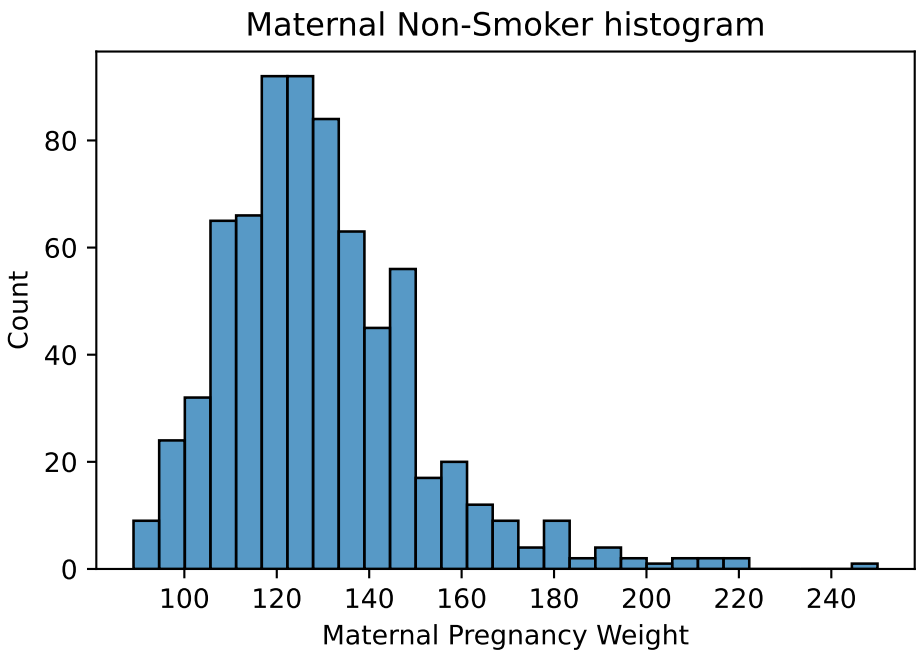
For example, the distribution of birth weights for maternal smokers is (weakly) multimodal.

```
births_maternal_smoker = births[births['Maternal Smoker'] == True]
sns.histplot(data = births_maternal_smoker, x = 'Maternal Pregnancy Weight')\
            .set(title = 'Maternal Smoker histogram');
```



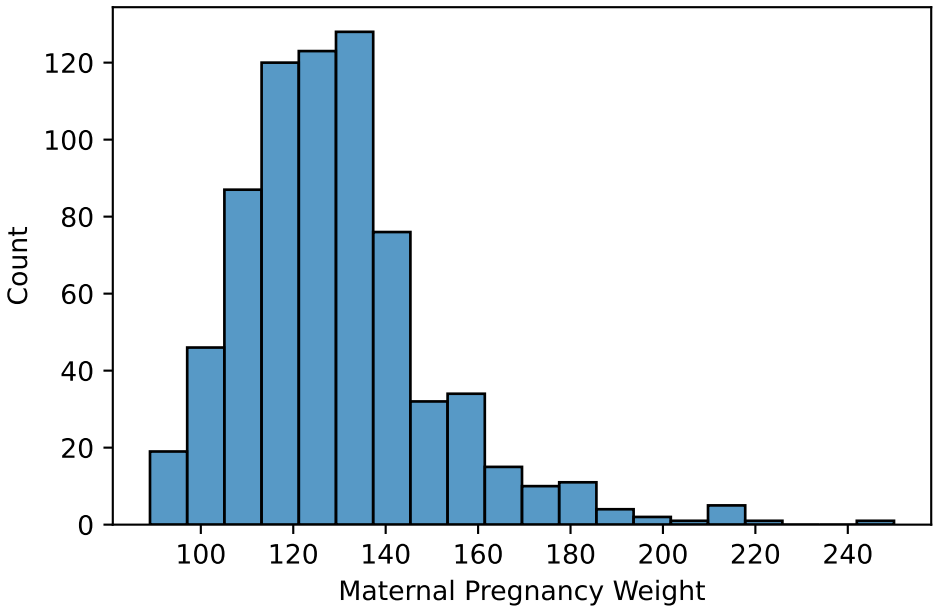
On the other hand, the distribution of birth weights for maternal non-smokers is weakly bimodal.

```
births_maternal_non_smoker = births[births['Maternal Smoker'] == False]
sns.histplot(data = births_maternal_non_smoker, x = 'Maternal Pregnancy Weight')\
            .set(title = 'Maternal Non-Smoker histogram');
```



However, changing the bins reveals that the data is not bi-modal.

```
sns.histplot(data = births_maternal_non_smoker, x = 'Maternal Pregnancy Weight',\
             bins = 20);
```



8.7 Density Curves

Instead of a discrete histogram, we can visualize what a continuous distribution corresponding to that same data could look like using a curve. - The smooth curve drawn on top of the histogram here is called a density curve.

In lecture 8, we will study how exactly to compute these density curves (using a technique is called Kernel Density Estimation).

If we plot `birth weights` of babies of *smoking mothers*, we get a histogram that appears bimodal.

- Density curve reinforces belief in this bimodality.

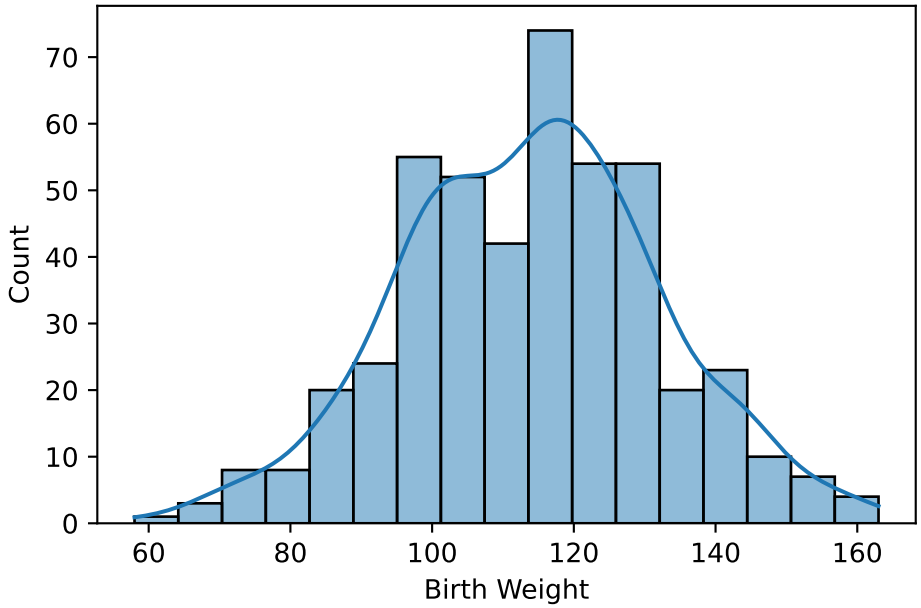
However, if we plot `birth weights` of babies of *non-smoking mothers*, we get a histogram that appears unimodal.

From a goal 1 perspective, this is EDA which tells us there may be something interesting here worth pursuing.

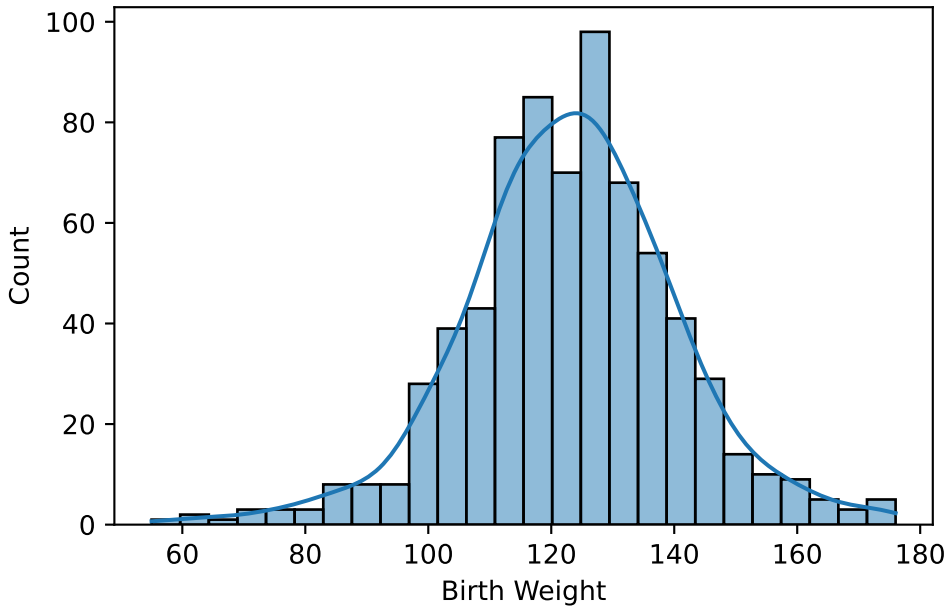
- Deeper analysis necessary!
- If we found something truly interesting, we'd have to cautiously write up an argument and create goal 2 level visualizations.

```
births_non_maternal_smoker = births[births['Maternal Smoker'] == False]
births_maternal_smoker = births[births['Maternal Smoker'] == True]

sns.histplot(data = births_maternal_smoker , x = 'Birth Weight',\
             kde = True);
```




```
sns.histplot(data = births_non_maternal_smoker , x = 'Birth Weight',\
             kde = True);
```



8.7.1 Histograms and Density

Rather than labeling by counts, we can instead plot the density, as shown below. Density gives us a measure that is invariant to the total number of observed units. The numerical values on the Y-axis for a sample of 100 units would be the same for when we observe a sample of 10000 units instead. We can still calculate the absolute number of observed units using density.

Example: There are 1174 observations total. - Total area of this bin should be: 120/1174 = ~10% - Density of this bin is therefore: 10% / (115 - 110) = 0.02

8.8 Box Plots and Violin Plots

8.8.1 Boxplots

Boxplots are an alternative to histograms that visualize numerical distributions. They are especially useful in graphicaly summarizing several characteristics of a distribution. These include:

1. Lower Quartile (1^{st} Quartile)
2. Median (2^{nd} Quartile)
3. Upper Quartile (3^{rd} Quartile)
4. Interquartile Range (IQR)

5. Whiskers
6. Outliers

The **lower quartile**, **median**, and **uper quartile** are the 25^{th}, 50^{th}, and 75^{th} percentiles of data, respectively. The **interquartile range** measures the spread of the middle 50% of the distribution, calculated as the (3^{rd} Quartile $-$ 1^{st} Quartile).

The **whiskers** of a box-plot are the two points that lie at the [1^{st} Quartile $-$ (1.5$\times$ IQR)], and the [3^{rd} Quartile + (1.5$\times$ IQR)]. They are the lower and upper ranges of "normal" data (the points excluding outliers). Subsequently, the **outliers** are the data points that fall beyond the whiskers, or further than (1.5$\times$ IQR) from the extreme quartiles.

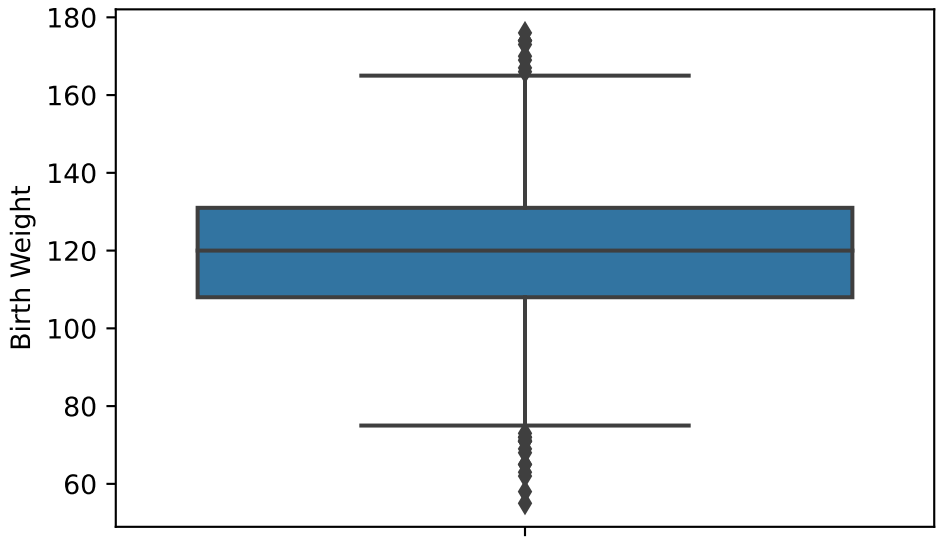
Let's visualize a box-plot of the `Birth Weight` column.

```
sns.boxplot(data = births, y = 'Birth Weight');

bweights = births['Birth Weight']
q1 = np.percentile(bweights, 25)
q2 = np.percentile(bweights, 50)
q3 = np.percentile(bweights, 75)
iqr = q3 - q1
whisk1 = q1 - (1.5 * iqr)
whisk2 = q3 + (1.5 * iqr)

print("The first quartile is {}".format(q1))
print("The second quartile is {}".format(q2))
print("The third quartile is {}".format(q3))
print("The interquartile range is {}".format(iqr))
print("The whiskers are {} and {}".format(whisk1, whisk2))
```

```
The first quartile is 108.0
The second quartile is 120.0
The third quartile is 131.0
The interquartile range is 23.0
The whiskers are 73.5 and 165.5
```

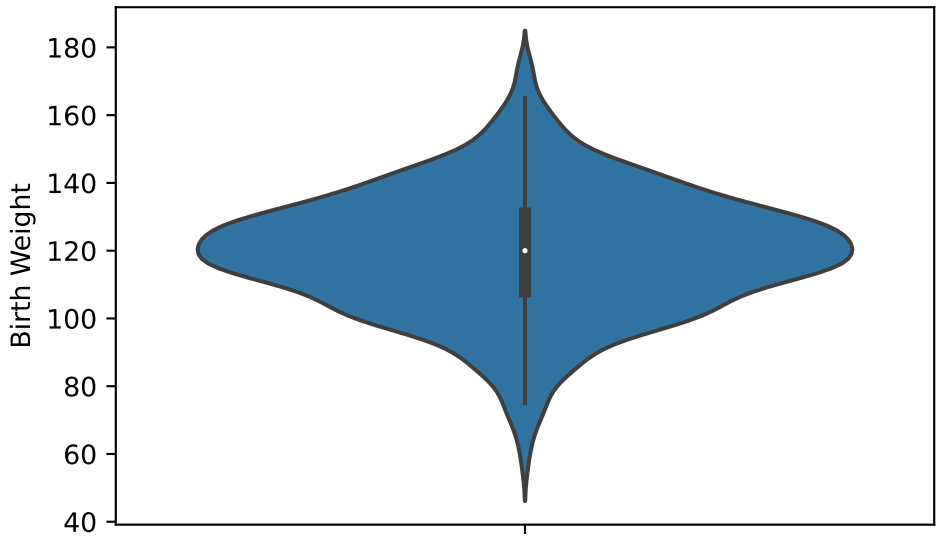


Here is a helpful visual that summarizes our discussion above.

8.8.2 Violin Plots

Another diagram that is useful in visualizing a variable's distribution is the violin plot. A **violin plot** supplements a box-plot with a smoothed density curve on either side of the plot. These density curves highlight the relative frequency of variable's possible values. If you look closely, you'll be able to discern the quartiles, whiskers, and other hallmark features of the box-plot.

```
sns.violinplot(data = births, y = 'Birth Weight');
```



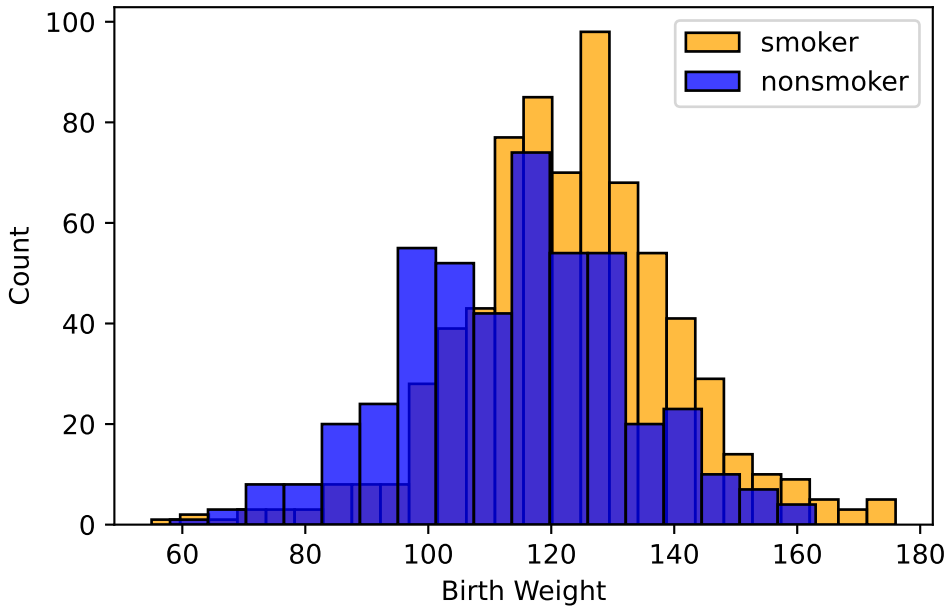
8.9 Comparing Quantitative Distributions

Earlier in our discussion of the mode, we visualized two histograms that described the distribution of birth weights for maternal smokers and non-smokers. However, comparing these histograms was difficult because they were displayed on seperate plots. Can we overlay the two to tell a more compelling story?

In `seaborn`, multiple calls to a plotting library in the same code cell will overlay the plots. For example:

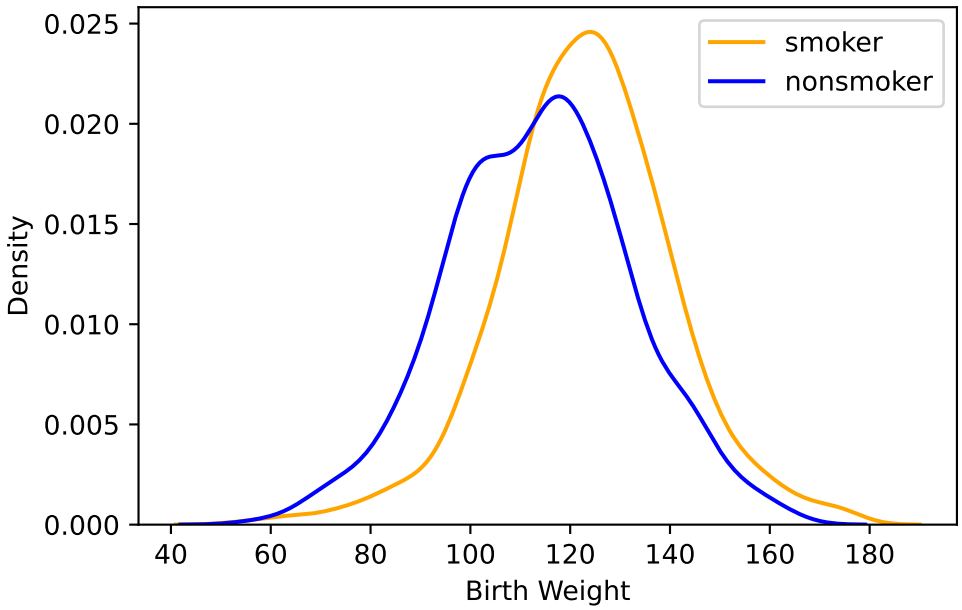
```
births_maternal_smoker = births[births['Maternal Smoker'] == False]
births_non_maternal_smoker = births[births['Maternal Smoker'] == True]

sns.histplot(data = births_maternal_smoker, x = 'Birth Weight',
             color = 'orange', label = 'smoker')
sns.histplot(data = births_non_maternal_smoker, x = 'Birth Weight',
             color = 'blue', label = 'nonsmoker')
plt.legend();
```



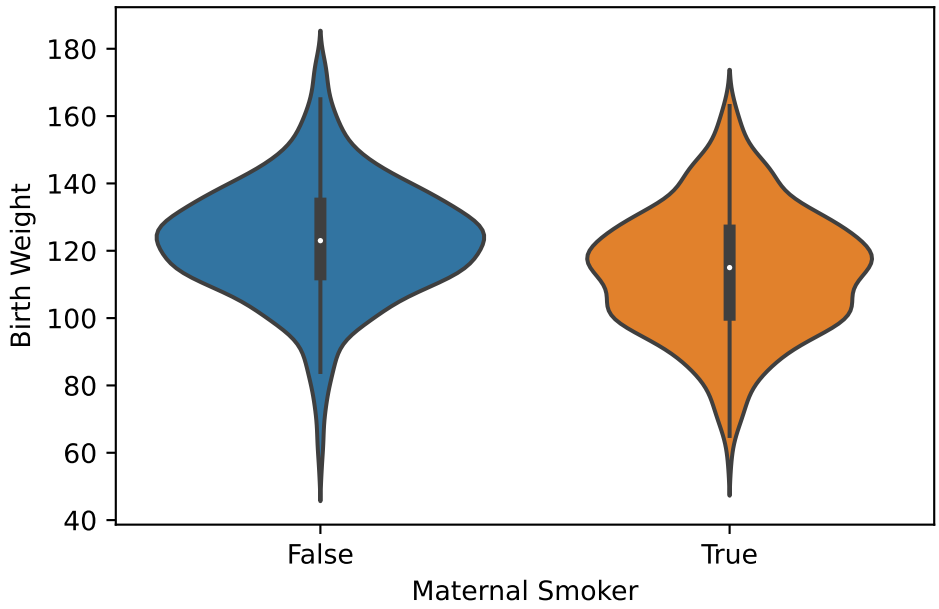
However, notice how this diagram suffers from overplotting. We can fix this with a call to `sns.kdeplot`. This will remove the bins and overlay the histogram with a density curve that better summarizes the distribution.

```
sns.kdeplot(data = births_maternal_smoker, x = 'Birth Weight', color = 'orange', label = '
sns.kdeplot(data = births_non_maternal_smoker, x = 'Birth Weight', color = 'blue', label =
plt.legend();
```

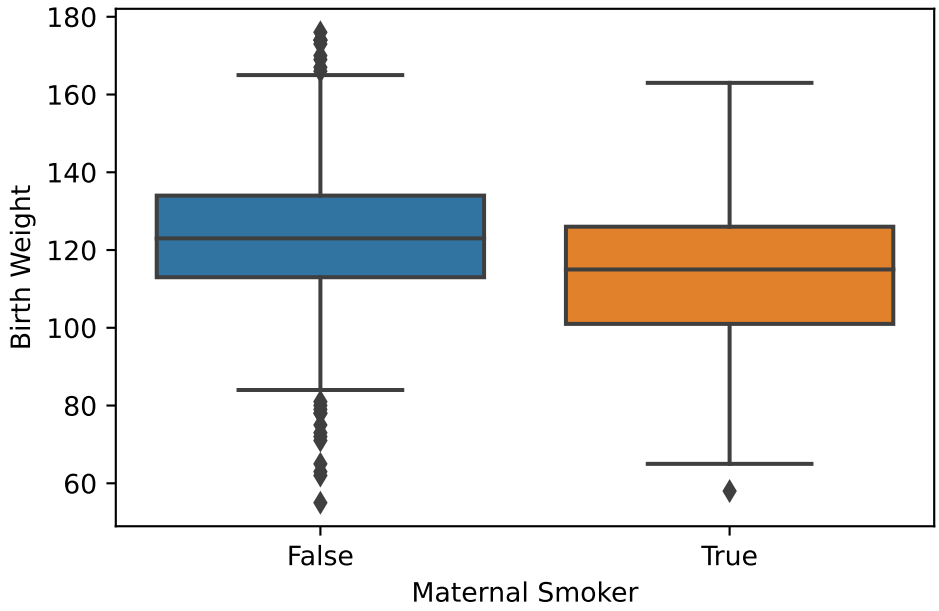


Unfortunately, we lose critical information in our distribution by removing small details. Therefore, we typically prefer to use box-plots and violin plots when comparing distributions. These are more concise and allow us to compare summary statistics across many distributions.

```
sns.violinplot(data = births, x = 'Maternal Smoker', y = 'Birth Weight');
```




```
sns.boxplot(data=births, x = 'Maternal Smoker', y = 'Birth Weight');
```



8.10 Ridge Plots

Ridge plots show many density curves offset from one another with minimal overlap. They are useful when the specific shape of each curve is important.

9 Visualization II

> **Note**
>
> - Use KDE for estimating density curve.
> - Using transformations to analyze the relationship between two variables.
> - Evalauting quality of a visualization based on visualization theory concepts.

9.1 Kernel Density Functions

9.1.1 KDE Mechanics

In the last lecture, we learned that density curves are smooth, continuous functions that represent a distribution of values. In this section, we'll learn how to construct density curves using Kernel Density Estimation (KDE).

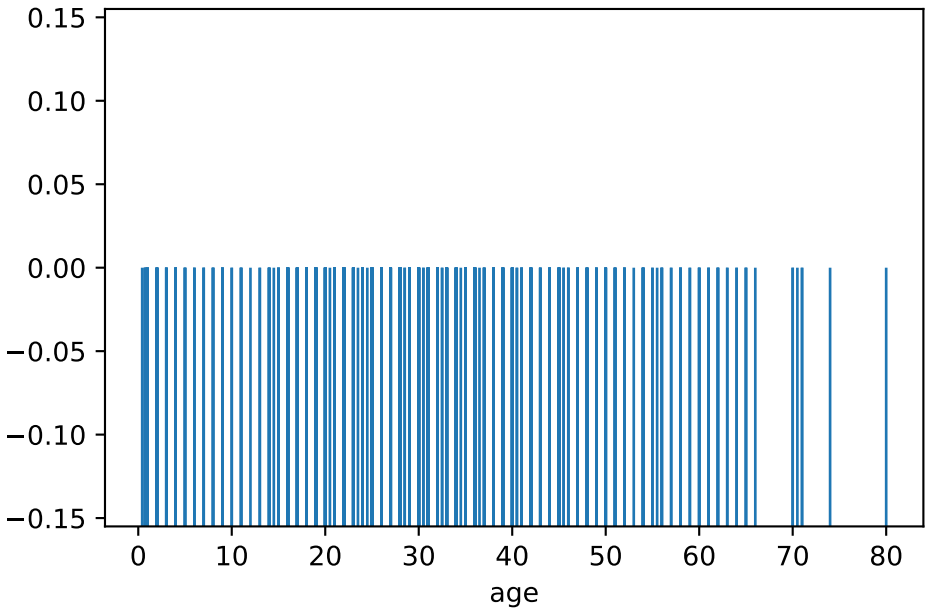
9.1.1.1 Smoothing

Kernel Density Estimation involves a technique called **smoothing** - a process applied to a distribution of values that allows us to analyze the more general structure of the dataset.

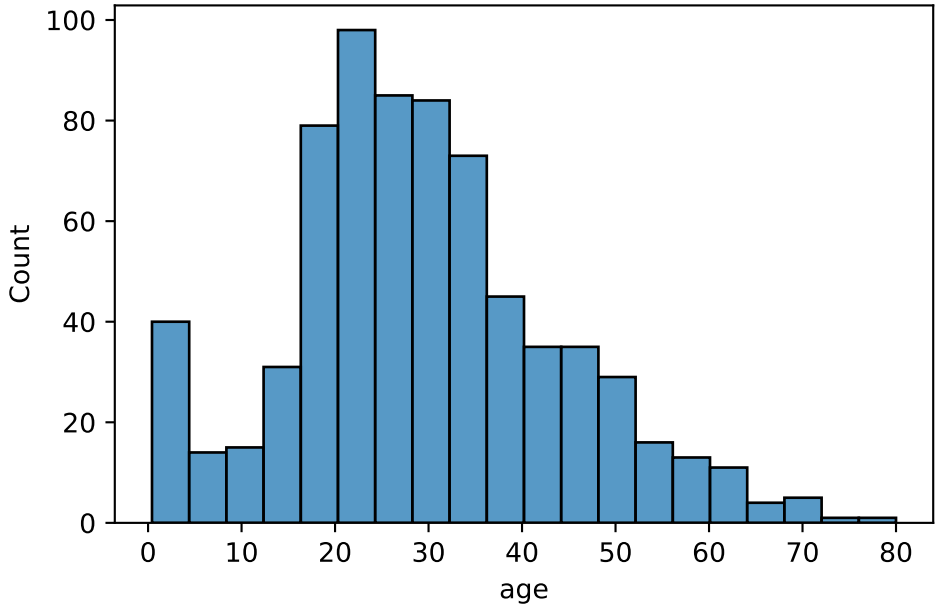
Many of the visualizations we learned during the last lecture are examples of smoothing. Histograms are smoothed versions of one-dimensional rug plots, and hex plots are smoother alternatives to two-dimensional scatter plots. They remove the detail from individual observations so we can visualize the larger properties of our distribution.

```
import seaborn as sns

titanic = sns.load_dataset('titanic')
sns.rugplot(titanic['age'],height = 0.5);
```




```
sns.histplot(titanic['age']);
```



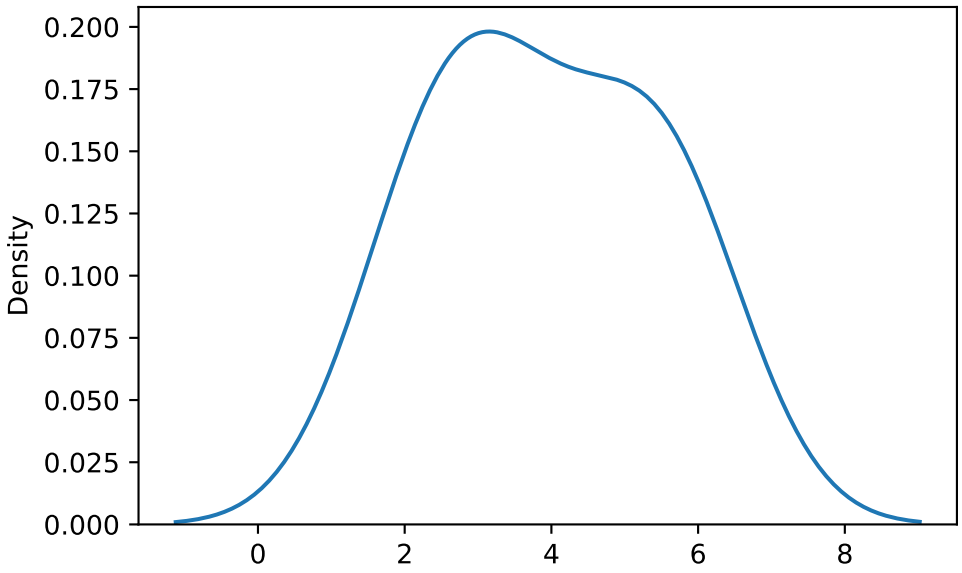
9.1.1.2 Kernel Density Estimation

Kernel Density Estimation is a smoothing technique that allows us to estimate a density curve (also known as a probability density function) from a set of observations. There are a few steps in this process:

1. Place a kernel at each data point
2. Normalize kernels to have total area of 1 (across all kernels)
3. Sum kernels together

Suppose we have 5 data points: $[2.2, 2.8, 3.7, 5.3, 5.7]$. We wish to recreate the following Kernel Density Estimate:

```
data = [2.2, 2.8, 3.7, 5.3, 5.7]
sns.kdeplot(data);
```



Let's walk through each step to construct this density curve.

9.1.1.2.1 Step 1 - Place a Kernel at Each Data Point

To begin generating a density curve, we need to choose a **kernel** and **bandwidth value** (α). What are these exactly? A **kernel** is a density curve itself, and the **bandwidth** (α) is a measure of the kernel's width. Recall that a valid density has an area of 1.

At each of our 5 points (depicted in the rug plot on the left), we've placed a Gaussian kernel with a bandwidth parameter of alpha = 1. We'll explore what these are in the next section.

Rugplot of Data

Kernelized Data

9.1.1.2.2 Step 2 - Normalize Kernels to Have Total Area of 1

Notice how these 5 kernels are density curves - meaning they each have an area of 1. In Step 3, we will be summing each these kernels, and we want the result to be a valid density that has an area of 1. Therefore, it makes sense to normalize our current set of kernels by multiplying each by $\frac{1}{5}$.

Kernelized Data

Normalized Kernels

9.1.1.2.3 Step 3 - Sum Kernels Together

Our kernel density estimate (KDE) is the sum of the normalized kernels along the x-axis. It is depicted below on the right.

Normalized Kernels

Kernel Density Estimate

9.1.2 Kernel Functions and Bandwidth

9.1.2.1 Kernels

A **kernel** (for our purposes) is a valid density function. This means it:

- Must be non-negative for all inputs.
- Must integrate to 1.

A general "KDE formula" function is given above.

1. $K_\alpha(x, xi)$ is the kernel centered on the observation `i`.
    - Each kernel individually has area 1.
    - x represents any number on the number line. It is the input to our function.
2. n is the number of observed data points that we have.
    - We multiply by $\frac{1}{n}$ so that the total area of the KDE is still 1.

3. Each $x_i \in \{x_1, x_2, \ldots, x_n\}$ represents an observed data point.
    - These are what we use to create our KDE by summing multiple shifted kernels centered at these points.

*α (alpha) is the bandwidth or smoothing parameter.

9.1.2.1.1 Gaussian Kernel

The most common kernel is the **Gaussian kernel**. The Gaussian kernel is equivalent to the Gaussian probability density function (the Normal distribution), centered at the observed value x_i with a standard deviation of α (this is known as the **bandwidth** parameter).

$K_a(x, x_i) = \frac{1}{\sqrt{2\pi\alpha^2}} e^{-\frac{(x-x_i)^2}{2\alpha^2}}$

```
import numpy as np
import matplotlib.pyplot as plt

def gaussian_kernel(alpha, x, z):
    return 1.0/np.sqrt(2. * np.pi * alpha**2) * np.exp(-(x - z) ** 2 / (2.0 * alpha**2))

xs = np.linspace(-5, 5, 200)
alpha = 1
kde_curve = [gaussian_kernel(alpha, x, 0) for x in xs]
plt.plot(xs, kde_curve);
```

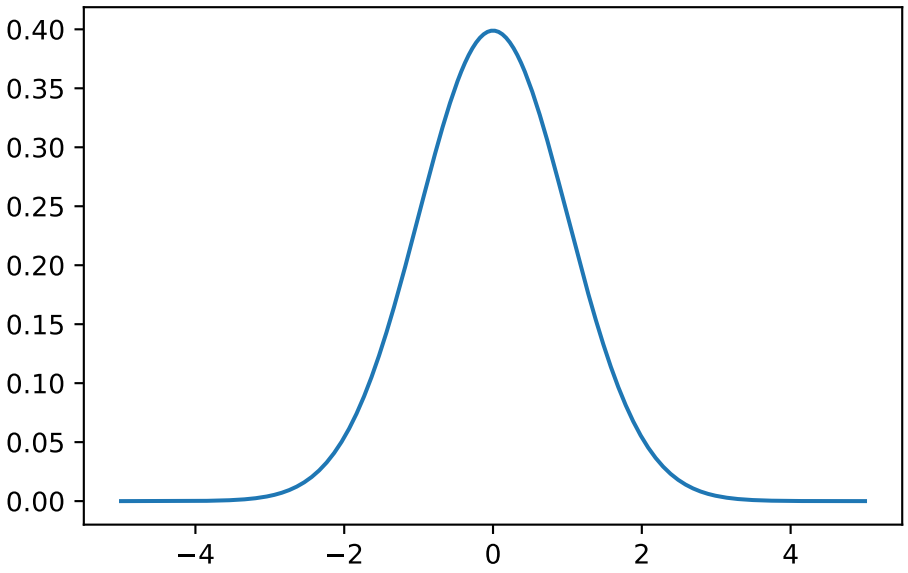


Figure 9.1: The Gaussian kernel centered at 0 with bandwidth $\alpha = 1$.

If you've taken a probability class, you'll recognize that the mean of this Gaussian kernel is x_i and the standard deviation is α. Increasing α - equivalently, the bandwidth - smoothens the density curve. Larger values of α are typically easier to understand; however, we begin to lose important distributional information.

Here is how adjusting α affects a distribution in some variable from an arbitrary dataset.

Gaussian Kernel, $\alpha = 0.1$

Gaussian Kernel, $\alpha = 1$

Gaussian Kernel, $\alpha = 2$

Gaussian Kernel, $\alpha = 10$

9.1.2.1.2 Boxcar Kernel

Another example of a kernel is the **Boxcar kernel**. The boxcar kernel assigns a uniform density to points within a "window" of the observation, and a density of 0 elsewhere. The equation below is a Boxcar kernel with the center at x_i and the bandwidth of α.

$$K_a(x, x_i) = \begin{cases} \frac{1}{\alpha}, & |x - x_i| \leq \frac{\alpha}{2} \\ 0, & \text{else} \end{cases}$$

```
def boxcar_kernel(alpha, x, z):
    return (((x-z)>=-alpha/2)&((x-z)<=alpha/2))/alpha

xs = np.linspace(-5, 5, 200)
alpha=1
kde_curve = [boxcar_kernel(alpha, x, 0) for x in xs]
plt.plot(xs, kde_curve);
```

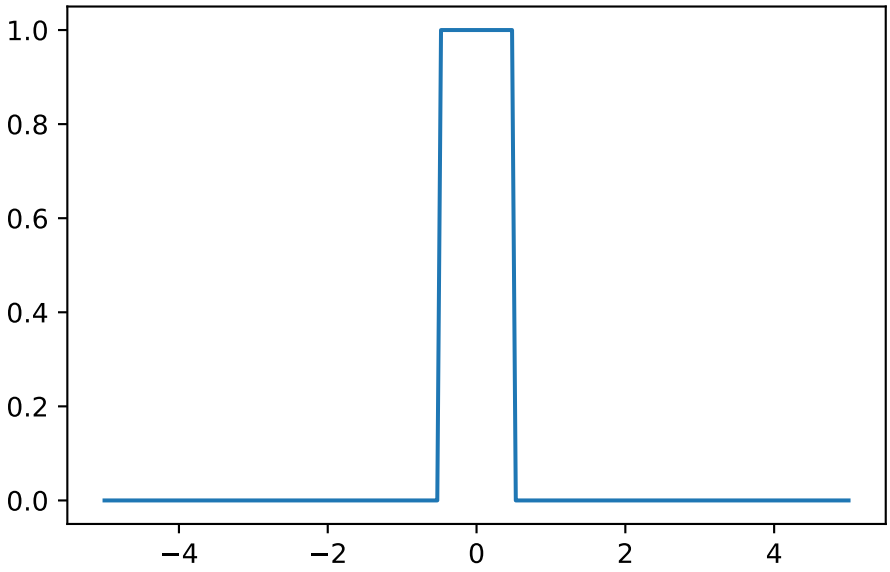


Figure 9.2: The Boxcar kernel centered at 0 with bandwidth $\alpha = 1$.

The diagram on the right is how the density curve for our 5 point dataset would have looked had we used the Boxcar kernel with bandwidth $\alpha = 1$.

9.1.3 Relationships Between Quantitative Variables

Up until now, we've discussed how to visualize single-variable distributions. Going beyond this, we want to understand the relationship between pairs of numerical variables.

9.1.3.1 Scatter Plots

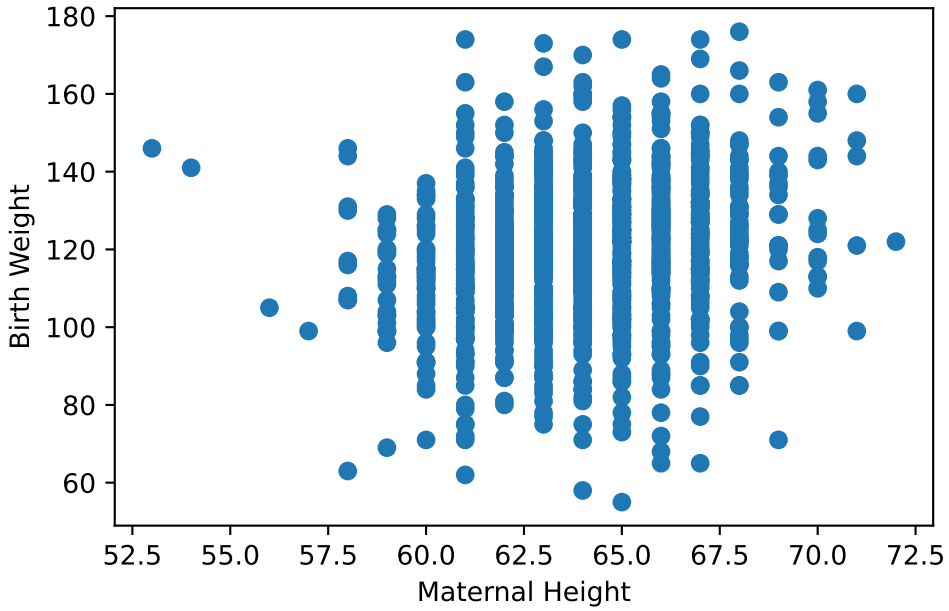
Scatter plots are one of the most useful tools in representing the relationship between two quantitative variables. They are particularly important in gauging the strength, or correla-

tion between variables. Knowledge of these relationships can then motivate decisions in our modeling process.

For example, let's plot a scatter plot comparing the `Maternal Height` and `Birth Weight` colums, using both `matplotlib` and `seaborn`.

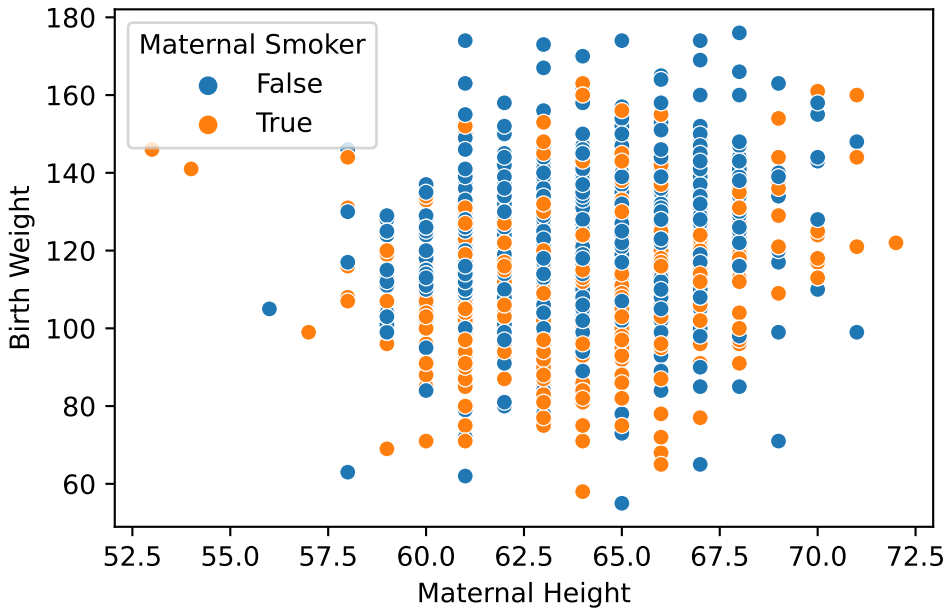
```
import pandas as pd
births = pd.read_csv("data/baby.csv")
births.head(5)

# Matplotlib Example
plt.scatter(births['Maternal Height'], births['Birth Weight'])
plt.xlabel('Maternal Height')
plt.ylabel('Birth Weight');
```



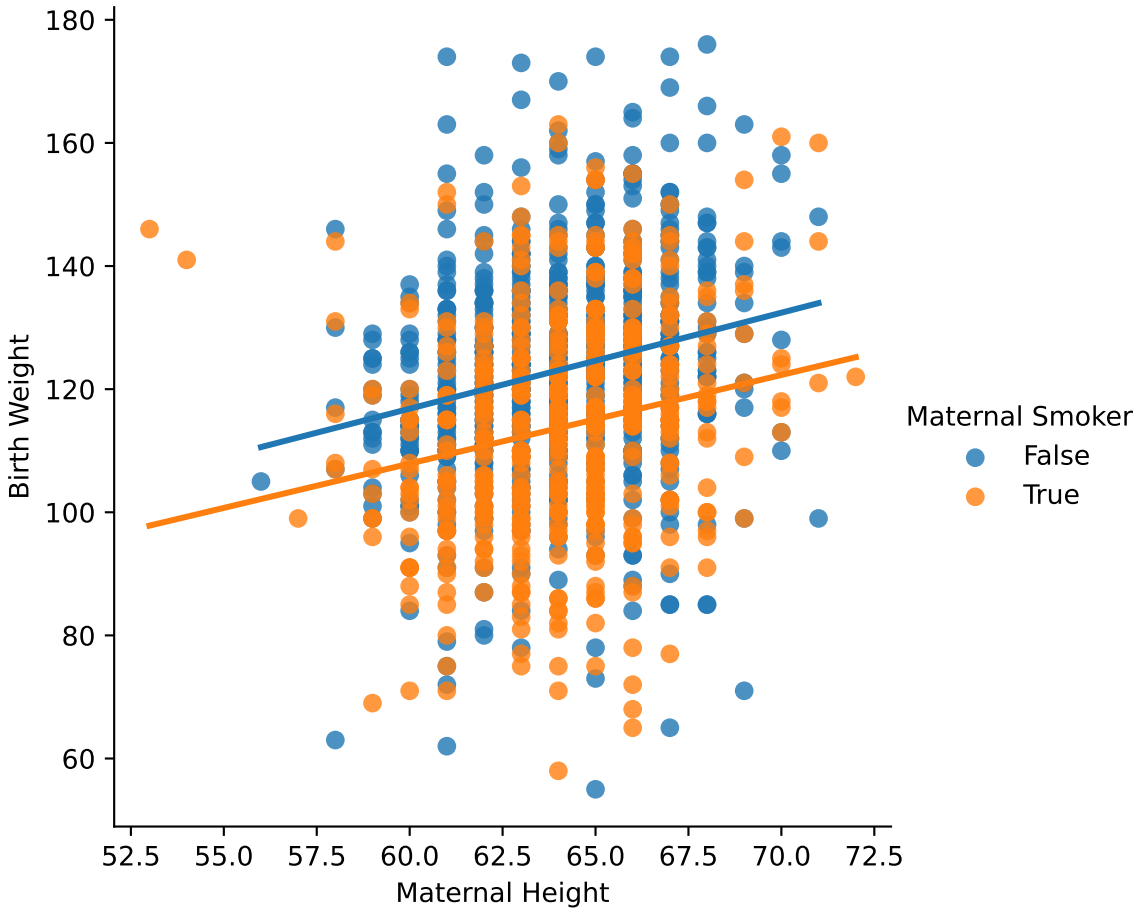

```
# Seaborn Example
sns.scatterplot(data = births, x = 'Maternal Height', y = 'Birth Weight',
                hue = 'Maternal Smoker')
```

```
<Axes: xlabel='Maternal Height', ylabel='Birth Weight'>
```



This is an example where color is used to add a third dimension to our plot. This is possible with the `hue` paramater in `seaborn`, which adds a categorical column encoding to an existing visualization. This way, we can look for relationships in `Maternal Height` and `Birth Weight` in both maternal smokers and non-smokers. If we wish to see the relationship's strength more clearly, we can use `sns.lmplot`.

```
sns.lmplot(data = births, x = 'Maternal Height', y = 'Birth Weight',
           hue = 'Maternal Smoker', ci = False);
```



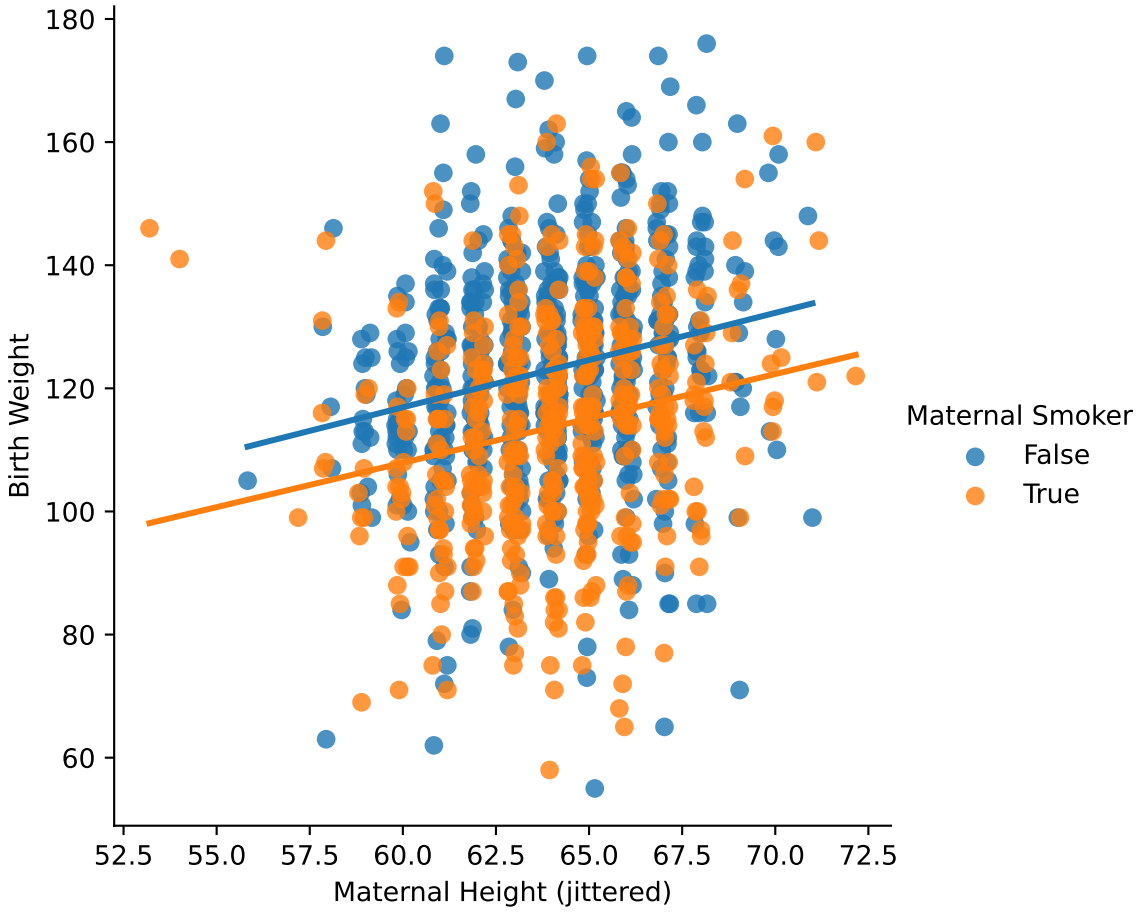
We can make out a weak, positive relationship in the mother's height and birth weight for both maternal smokers and non-smokers (the baseline is slightly lower in maternal smokers).

9.1.4 Overplotting

As you may have noticed, the scatterplots of `Maternal Height` vs. `Birth Weight` have many densely plotted areas. Many of the points are on top of one other! This makes it difficult to tell exactly how many babies are plotted in each the more densely populated regions of the graph. This can arise when the tools used for measuring data have low granularity, many different values are rounded to the same value, or if the ranges of the two variables differ greatly in scale.

We can overcome this by introducing a small amount of uniform random noise to our data. This is called *jittering*. Let's see what happens when we introduce noise to the `Maternal Height`.

```
births["Maternal Height (jittered)"] = births["Maternal Height"] + np.random.uniform(-0.2,
sns.lmplot(data = births, x = 'Maternal Height (jittered)', y = 'Birth Weight',
           hue = 'Maternal Smoker', ci = False);
```



This plot more clearly shows that most of the data is clustered tightly around the point (62.5,120) and gradually becomes more loose further away from the center. It is much easier for us and others to see how the data is distributed. In conclusion, *jittering* helps us better understand our own data (Goal 1) and communicate results to others (Goal 2).

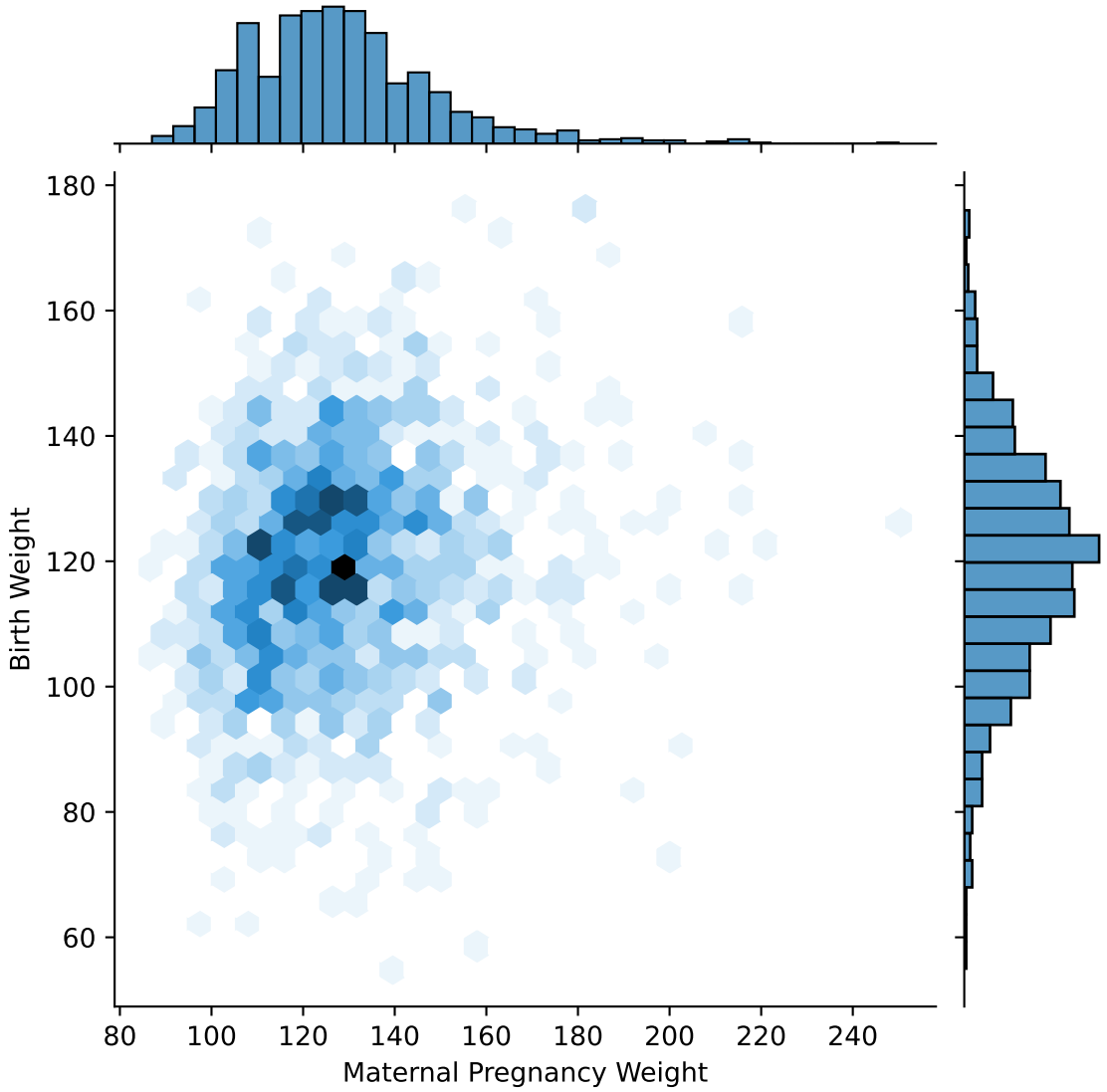
9.1.4.1 Hex Plots and Contour Plots

Unfortunately, our scatter plots above suffered from overplotting, which made them hard to interpret. And with a large number of points, jittering is unlikely to resolve the issue. Instead, we can look to hex plots and contour plots.

Hex Plots can be thought of as a two dimensional histogram that shows the joint distribution between two variables. This is particularly useful working with very dense data.

```
sns.jointplot(data = births, x = 'Maternal Pregnancy Weight',
              y = 'Birth Weight', kind = 'hex')
```

```
<seaborn.axisgrid.JointGrid at 0x126dd8650>
```



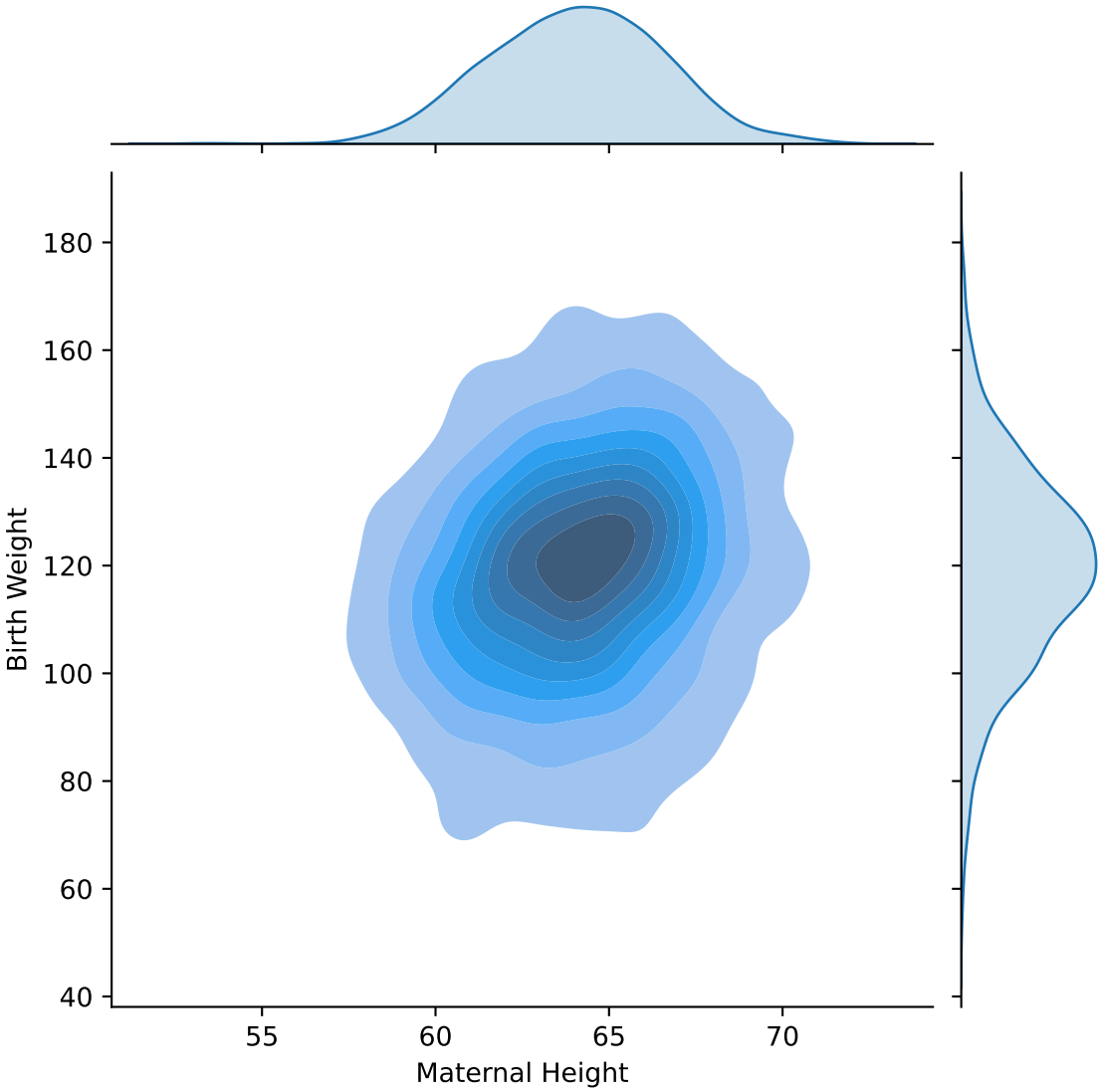
The axes are evidently binned into hexagons, which makes the linear relationship easier to

decipher. Darker regions generally indicate a higher density of points.

On the other hand, **contour plots** are two dimensional versions of density curves with marginal distributions of each variable on the axes. We've used very similar code here to generate our contour plots, with the addition of the `kind = 'kde'` and `fill = True` arguments.

```
sns.jointplot(data = births, x = 'Maternal Height', y = 'Birth Weight',\
              kind = 'kde', fill = True)
```

```
<seaborn.axisgrid.JointGrid at 0x126f3bc10>
```



9.2 Transformations

These last two lectures have covered visualizations in great depth. We looked at various forms of visualizations, plotting libraries, and high-level theory.

Much of this was done to uncover insights in data, which will prove necessary for the modeling process. A strong graphical correlation between two variables hinted an underlying relationship that has reason for further study. However, relying on visual relationships alone is limiting -

not all plots show association. The presence of outliers and other statistical anomalies make it hard to interpret data.

Transformations are the process of manipulating data to find significant relationships between variables. These are often found by applying mathematical functions to variables that "transform" their range of possible values and highlight some previously hidden associations between data.

9.2.0.1 Transforming a Distribution

When a distribution has a large dynamic range, it can be useful to take the logarithm of the data. For example, computing the logarithm of the ticket prices on the Titanic reduces skeweness and yields a distribution that is more "spread" across the x-axis. While it makes individual observations harder to interpret, the distribution is more favorable for subsequent analysis.

9.2.0.2 Linearizing a Relationship

Transformations are perhaps most useful to **linearize a relationship** between variables. If we find a transformation to make a scatter plot of two variables linear, we can "backtrack" to find the exact relationship between the variables. Linear relationships are particularly simple to interpret, and we'll be doing a lot of linear modeling in Data 100 - starting next week!

Say we want to understand the relationship between healthcare and life expectancy. Intuitively there should be a positive correlation, but upon plotting values from a dataset, we find a non-linear relationship that is somewhat hard to understand. However, applying a logarithmic transformation to both variables - healthcare and life expectancy - results in a scatter plot with a linear trend that we can interpret.

How can we find the relationship between the original variables? We know that taking a log of both axes gave us a linear relationship, so we can say (roughly) that

$$\log y = a \times \log x + b$$

Solving for y implies a **power** relationship in the original plot.

$$y = e^{a \times \log x + b}$$
$$y = Ce^{a \times \log x}$$
$$y = Cx^a$$

How did we know that taking the logarithm of both sides would result in a linear relationship? The **Tukey-Mosteller Bulge Diagram** is helpful here. We can use the direction of the buldge in our original data to find the appropriate transformations that will linearize the relationship. These transformations are found on axes that are nearest to the buldge. The buldge in our earlier example lay in Quadrant 2, so the transformations $\log x$, $\sqrt{x}$, y^2, or y^3 are possible contenders. It's important to note that this diagram is not perfect, and some transformations will work better than others. In our case, $\log x$ and $\log y$ (found in Quadrant 3) were the best.

9.2.0.3 Additional Remarks

Visualization requires a lot of thought! - There are many tools for visualizing distributions. - Distribution of a single variable: 1. rug plot 2. histogram 3. density plot 4. box plot 5. violin plot - Joint distribution of two quantitative variables: 1. scatter plot 2. hex plot 3. contour plot.

This class primarily uses `seaborn` and `matplotlib`, but `Pandas` also has basic built-in plotting methods. Many other visualization libraries exist, and `plotly` is one of them. - `plotly` creates very easily creates interactive plots. - `plotly` will occasionally appear in lecture code, labs, and assignments!

Next, we'll go deeper into the theory behind visualization.

9.3 Visualization Theory

This section marks a pivot to the second major topic of this lecture - visualization theory. We'll discuss the abstract nature of visualizations and analyze how they convey information.

Remember, we had two goals for visualizing data. This section is particularly important in:

1. Helping us understand the data and results
2. Communicating our results and conclusions with others

9.3.1 Information Channels

Visualizations are able to convey information through various encodings. In the remainder of this lecture, we'll look at the use of color, scale, and depth, to name a few.

9.3.1.1 Encodings in Rugplots

One detail that we may have overlooked in our earlier discussion of rugplots is the importance of encodings. Rugplots are effective visuals because they utilize line thickness to encode frequency. Consider the following diagram:

9.3.1.2 Multi-Dimensional Encodings

Encodings are also useful for representing multi-dimensional data. Notice how the following visual highlights four distinct "dimensions" of data:

- X-axis
- Y-axis
- Area
- Color

The human visual perception sytem is only capable of visualizing data in a three-dimensional plane, but as you've seen, we can encode many more channels of information.

9.3.2 Harnessing the Axes

9.3.2.1 Consider Scale of the Data

However, we should be careful to not misrepresent relationships in our data by manipulating the scale or axes. The visualization below improperly portrays two seemingly independent relationships on the same plot. The authors have clearly changed the scale of the y-axis to mislead their audience.

Notice how the downwards-facing line segment contains values in the millions, while the upwards-trending segment only contains values near three hundred thousand. These lines should not be intersecting.

When there is a large difference in the magnitude of the data, it's advised to analyze percentages instead of counts. The following diagrams correctly display the trends in cancer screening and abortion rates.

9.3.2.2 Reveal the Data

Great visualizations not only consider the scale of the data, but also utilize the axes in a way that best conveys information. For example, data scientists commonly set certain axes limits to highlight parts of the visualization they are most interested in.

The visualization on the right captures the trend in coronavirus cases during the month March in 2020. From only looking at the visualization on the left, a viewer may incorrectly believe that coronavirus began to skyrocket on March 4^{th}, 2020. However, the second illustration tells a different story - cases rose closer to March 21^{th}, 2020.

9.3.3 Harnessing Color

Color is another important feature in visualizations that does more than what meets the eye.

Last lecture, we used color to encode a categorical variable in our scatter plot. In this section, we will discuss uses of color in novel visualizations like colormaps and heatmaps.

5-8% of the world is red-green color blind, so we have to be very particular about our color scheme. We want to make these as accessible as possible. Choosing a set of colors which work together is evidently a challenging task!

9.3.3.1 Colormaps

Colormaps are mappings from pixel data to color values, and they're often used to highlight distinct parts of an image. Let's investigate a few properties of colormaps.

Jet Colormap

Viridis Colormap

The jet colormap is infamous for being misleading. While it seems more vibrant than viridis, the aggressive colors poorly encode numerical data. To understand why, let's analyze the following images.

The diagram on the left compares how a variety of colormaps represent pixel data that transitions from a high to low intensity. These include the jet colormap (row a) and grayscale (row b). Notice how the grayscale images do the best job in smoothly transitioning between pixel data. The jet colormap is the worst at this - the four images in row (a) look like a conglomeration of individual colors.

The difference is also evident in the images labeled (a) and (b) on the left side. The grayscale image is better at preserving finer detail in the vertical line strokes. Additionally, grayscale is preferred in x-ray scans for being more neutral. The intensity of dark red color in the jet colormap is frightening and indicates something is wrong.

Why is the jet colormap so much worse? The answer lies in how its color composition is percieved to the human eye.

Jet Colormap Perception

Viridis Colormap Perception

The jet colormap is largely misleading because it is not perceptually uniform. **Perceptually uniform colormaps** have the property that if the pixel data goes from 0.1 to 0.2, the perceptual change is the same as when the data goes from 0.8 to 0.9.

Notice how the said uniformity is present within the linear trend displayed in the viridis colormap. On the other hand, the jet colormap is largely non-linear - this is precisely why it's considered a worse colormap.

9.3.4 Harnessing Markings

In our earlier discussion of multi-dimensional encodings, we analyzed a scatter plot with four pseudo-dimensions: the two axes, area, and color. Were these appropriate to use? The following diagram analyzes how well the human eye can distinguish between these "markings".

There are a few key takeaways from this diagram

- Lengths are easy to discern. Don't use plots with jiggled baselines - keep everything axis-aligned.
- Avoid pie charts! Angle judgements are inaccurate.
- Areas and volumes are hard to distinguish (area charts, word clouds, etc)

9.3.5 Harnessing Conditioning

Conditioning is the process of comparing data that belong to seperate groups. We've seen this before in overlayed distributions, side-by-side box-plots, and scatter plots with categorical encodings. Here, we'll introduce terminology that formalizes these examples.

Consider an example where we want to analyze income earnings for male and females with varying levels of education. There are multiple ways to compare this data.

The barplot is an example of **juxtaposition**: placing multiple plots side by side, with the same scale. The scatter plot is an example of **superposition**: placing multiple density curves, scatter plots on top of each other.

Which is better depends on the problem at hand. Here, superposition makes the precise wage difference very clear from a quick glance. But many sophisticated plots convey information that favors the use of juxtaposition. Below is one example.

9.3.6 Harnessing Context

The last component to a great visualization is perhaps the most critical - the use of context. Adding informative titles, axis labels, and descriptive captions are all best practices that we've heard repeatedly in Data 8.

A publication-ready plot (and every Data 100 plot) needs:

- Informative title (takeaway, not description)
- Axis labels
- Reference lines, markers, etc
- Legends, if appropriate
- Captions that describe data

Captions should be:

- Comprehensive and self-contained
- Describe what has been graphed
- Draw attention to important features
- Describe conclusions drawn from graphs

10 Sampling

> **Note**
>
> - Understand how to appropriately collect data to help answer a question.

In Data Science, understanding characteristics of a population starts with having quality data to investigate. While it is often impossible to collect all the data describing a population, we can overcome this by properly sampling from the population. In this note, we will discuss appropriate techniques for sampling from populations.

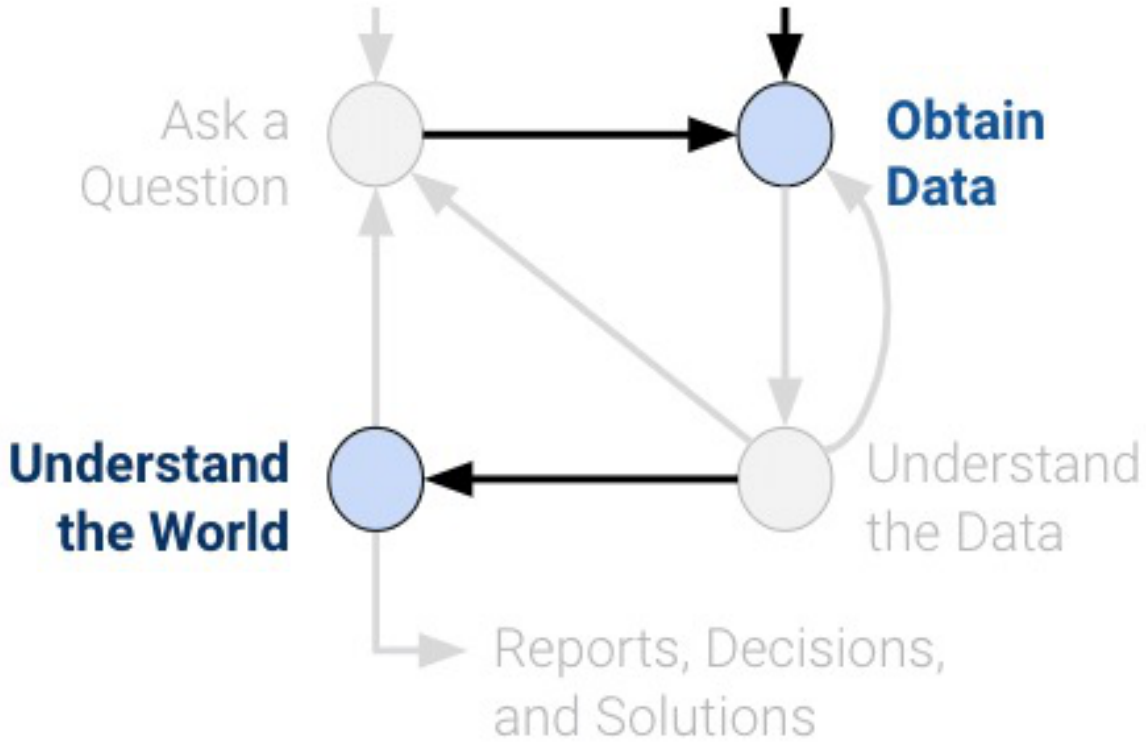


Figure 10.1: Lifecycle diagram

10.1 Censuses and Surveys

In general: a **census** is "an official count or survey of a population, typically recording various details of individuals."

- Example: The U.S. Decennial Census was held in April 2020, and it counts **every person** living in all 50 states, DC, and US territories. (Not just citizens.) Participation is required by law (it is mandated by the U.S. Constitution). Important uses include the allocation of Federal funds, congressional representation, and drawing congressional and state legislative districts. The census is composed of a **survey** mailed to different housing addresses in the United States.
- **Individuals** in a population are not always people. Other populations include: bacteria in your gut (sampled using DNA sequencing); trees of a certain species; small businesses receiving a microloan; or published results in an academic journal / field.

A **survey** is a set of questions. An example is workers sampling individuals and households. What is asked, and how it is asked, can affect how the respondent answers, or even whether the respondent answers in the first place.

While censuses are great, it is often difficult and expensive to survey everyone in a population. Thus, we usually survey a subset of the population instead.

A **sample** is often used to make inferences about the population. That being said, how the sample is drawn will affect the reliability of such inferences. Two common source of error in sampling are **chance error**, where random samples can vary from what is expected, in any direction; and **bias**, which is a a systematic error in one direction.

Because of how surveys and samples are drawn, it turns out that samples are usually—but not always—a subset of the population: * **Population**: The group that you want to learn something about. * **Sampling Frame**: The list from which the sample is drawn. For example, if sampling people, then the sampling frame is the set of all people that could possibly end up in your sample. * **Sample**: Who you actually end up sampling. The sample is therefore a subset of your *sampling frame*.

While ideally these three sets would be exactly the same, in practice they usually aren't. For example, there may be individuals in your sampling frame (and hence, your sample) that are not in your population. And generally, sample sizes are much smaller than population sizes.

10.2 Bias: A Case Study

The following case study is adapted from *Statistics* by Freedman, Pisani, and Purves, W.W. Norton NY, 1978.

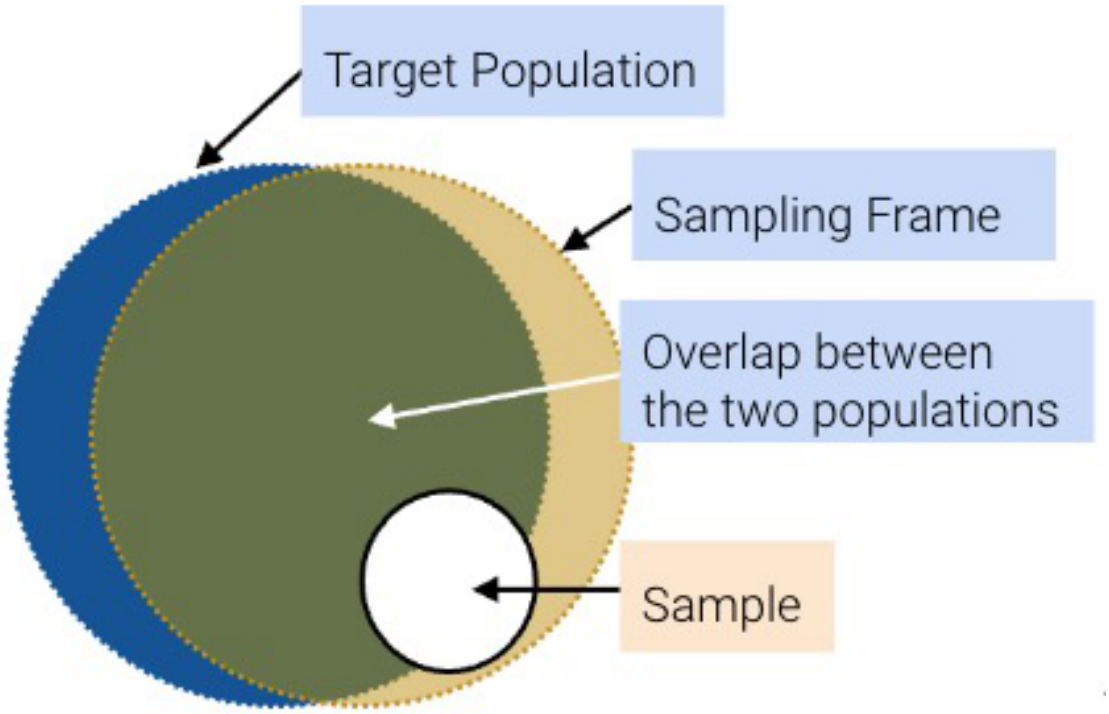


Figure 10.2: Sampling_Frames

In 1936, President Franklin D. Roosevelt (D) went up for re-election against Alf Landon (R) . As is usual, **polls** were conducted in the months leading up to the election to try and predict the outcome. The *Literary Digest* was a magazine that had successfully predicted the outcome of 5 general elections coming into 1936. In their polling for the 1936 election, they sent out their survey to 10 million individuals, who they found from phone books, lists of magazine subscribers, and lists of country club members. Of the roughly 2.4 million people who filled out the survey, only 43% reported they would vote for Roosevelt; thus the *Digest* predicted that Landon would win.

On election day, Roosevelt won in a landslide, winning 61% of the popular vote of about 45 million voters. How could the *Digest* have been so wrong with their polling?

It turns out that the *Literary Digest* sample was not representative of the population. Their sampling frame inherently skewed towards more affluent voters, who tended to vote Republican, and they completely overlooked the lion's share of voters who were still suffering through the Great Depression. Furthermore, they had a dismal response rate (about 24%); who knows how the other non-respondents would have polled? The *Digest* folded just 18 months after this disaster.

At the same time, George Gallup, a rising statistician, also made predictions about the 1936 elections. His estimate (56% Roosevelt) was much closer despite having a smaller sample size of "only" 50,000 (still more than necessary; more when we cover the Central Limit Theorem). Gallup also predicted the *Digest*'s prediction within 1%, with a sample size of only 3000 people. He did so by anticipating the *Digest*'s affluent sampling frame and subsampled those individuals. The **Gallup Poll** today is one of the leading polls for election results.

So what's the moral of the story? Samples, while convenient, are subject to chance error and **bias**. Election polling, in particular, can involve many sources of bias. To name a few: * **Selection bias** systematically excludes (or favors) particular groups. * **Response bias** occurs because people don't always respond truthfully. Survey designers pay special detail to the nature and wording of questions to avoid this type of bias. * **Non-response bias** occurs because people don't always respond to survey requests, which can skew responses. For example, the Gallup poll is conducted through landline phone calls, but many different populations in the U.S. do not pay for a landline, and still more do not always answer the phone. Surveyers address this bias by staying persistent and keeping surveys short.

10.3 Probability Samples

When sampling, it is essential to focus on the quality of the sample rather than the quantity of the sample. A huge sample size does not fix a bad sampling method. Our main goal is to gather a sample that is representative of the population it came from. The most common way to accomplish this is by randomly sampling from the population.

- A **convenience sample** is whatever you can get ahold of. Note that haphazard sampling is not necessarily random sampling; there are many potential sources of bias.
- In a **probability sample**, we know the chance any given set of individuals will be in the sample.
  - Probability samples allow us to estimate the bias and chance error, which helps us quantify uncertainty (more in a future lecture).
  - Note that this does not imply that all individuals in the population need have the same chance of being selected (see: stratified random samples).
  - Further note that the real world is usually more complicated. For example, we do not generally know the probability that a given bacterium is in a microbiome sample, or whether people will answer when Gallup calls landlines. That being said, we try to model probability sampling where possible if the sampling or measurement process is not fully under our control.

A few common random sampling schemes: * A **random sample with replacement** is a sample drawn **uniformly** at random **with** replacement. * Random doesn't always mean "uniformly at random," but in this specific context, it does. * Some individuals in the population might get picked more than once

- A **simple random sample (SRS)** is a sample drawn uniformly at random without replacement.
  - Every individual (and subset of individuals) has the same chance of being selected.
  - Every pair has the same chance as every other pair.
  - Every triple has the same chance as every other triple.
  - And so on.
- A **stratified random sample**, where random sampling is performed on strata (specific groups), and the groups together compose a sample.

10.3.1 Example: Stratified random sample

Suppose that we are trying to run a poll to predict the mayoral election in Bearkeley City (an imaginary city that neighbors Berkeley). Suppose we try a **stratified random sample** to select 100 voters as follows: 1. First, we take a simple random sample and obtain 50 voters that are above the median city income ("above-median-income"), i.e., in the upper 50-th percentile of income in the city. 2. We then take a simple random sample of the other 50 from voters that are below the median city income.

This is a **probability sample**: For any group of 100 people, if there are not exactly 50 "above-median-income" voters, then that group has zero probability of being chosen. For any other group (which has exactly 50 "above-median-income" voters), then the chance of it being chosen is 1/ # of such groups.

Note that even if we replace the group counts with 80/20 (80 "above-median-income" voters, 20 others), then it is still a probability sample, because we can compute the precise probability of each group being chosen. However, the sampling scheme (and thus the modeling of voter preferences) becomes biased towards voters with income above the median.

10.4 Approximating Simple Random Sampling

The following is a very common situation in data science: - We have an enormous population. - We can only afford to sample a relatively small number of individuals. If the population is huge compared to the sample, then random sampling with and without replacement are pretty much the same.

Example : Suppose there are 10,000 people in a population. Exactly 7,500 of them like Snack 1; the other 2,500 like Snack 2. What is the probability that in a random sample of 20, all people like Snack 1?

- Method 1: SRS (Random Sample Without Replacement): $\prod_{k=0}^{19} \frac{7500-k}{10000-k} \approx 0.003151$
- Method 2: Random Sample with Replacement: $(0.75)^{20} \approx 0.003171$

As seen here, when the population size is large, probabilities of sampling with replacement are much easier to compute and lead to a reasonable approximation.

10.4.1 Multinomial Probabilities

The approximation discussed above suggests the convenience of **multinomial probabilities**, which arise from sampling a categorical distribution at random **with replacement*.

Suppose that we have a bag of marbles with the following distribution: 60% are blue, 30% are green, and 10% are red. If we then proceed to draw 100 marbles from this bag, at random with replacement, then the resulting 100-size sample is modeled as a multinomial distribution using `np.random.multinomial`:

```
import numpy as np
np.random.multinomial(100, [0.60, 0.30, 0.10])
```

```
array([64, 30,  6])
```

This method allows us to generate, say, 10 samples of size 100 using the `size` parameter:

```
np.random.multinomial(100, [0.60, 0.30, 0.10], size=10)
```

```
array([[57, 39,  4],
       [55, 33, 12],
       [56, 36,  8],
       [69, 24,  7],
       [59, 35,  6],
       [61, 29, 10],
       [67, 29,  4],
       [60, 27, 13],
       [56, 34, 10],
       [61, 30,  9]])
```

10.5 Comparing Convenience Sample and SRS

Suppose that we are trying to run a poll to predict the mayoral election in Bearkeley City (an imaginary city that neighbors Berkeley). Suppose we took a sample to predict the election outcome by polling all retirees. Even if they answer truthfully, we have a **convenience sample**. How biased would this sample be in predicting the results? While we will not numerically quantify the bias, in this demo we'll visually show that because of the voter population distribution, any error in our prediction from a retiree sample cannot be simply due to chance:

First, let's grab a data set that has every single voter in the Bearkeley (again, this is a fake dataset) and how they **actually** voted in the election. For the purposes of this example, assume: * "high income" indicates a voter is above the median household income, which is $97,834 (actual Berkeley number). * There are only two mayoral candidates: one Democrat and one Republican. * Every registered voter votes in the election for the candidate under their registered party (Dem or Rep).

```
import pandas as pd
import numpy as np
bearkeley = pd.read_csv("data/bearkeley.csv")

# create a 1/0 int that indicates democratic vote
bearkeley['vote.dem'] = (bearkeley['vote'] == 'Dem').astype(int)
bearkeley.head()
```

```
/Users/matthewshen/.pyenv/versions/3.11.2/lib/python3.11/site-packages/IPython/core/formatte
  return method()
```

	age	high_income	vote	vote.dem
0	35	False	Dem	1
1	42	True	Rep	0
2	55	False	Dem	1
3	77	True	Rep	0
4	31	False	Dem	1

```
bearkeley.shape
```

```
(1300000, 4)
```

```
actual_vote = np.mean(bearkeley["vote.dem"])
actual_vote
```

```
0.5302792307692308
```

This is the **actual outcome** of the election. Based on this result, the Democratic candidate would win. However, if we were to only consider retiree voters (a retired person is anyone age 65 and up):

```
convenience_sample = bearkeley[bearkeley['age'] >= 65]
np.mean(convenience_sample["vote.dem"])
```

```
0.3744755089093924
```

Based on this result, we would have predicted that the Republican candidate would win! This error is not due to the sample being too small to yield accurate predictions, because there are 359,396 retirees (about 27% of the 1.3 million Bearkeley voters). Instead, there seems to be something larger happening. Let's visualize the voter preferences of the entire population to see how retirees trend:

Let us aggregate all voters by age and visualize the fraction of Democratic voters, split by income.

```
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

sns.set_theme(style='darkgrid', font_scale = 1.5,
```

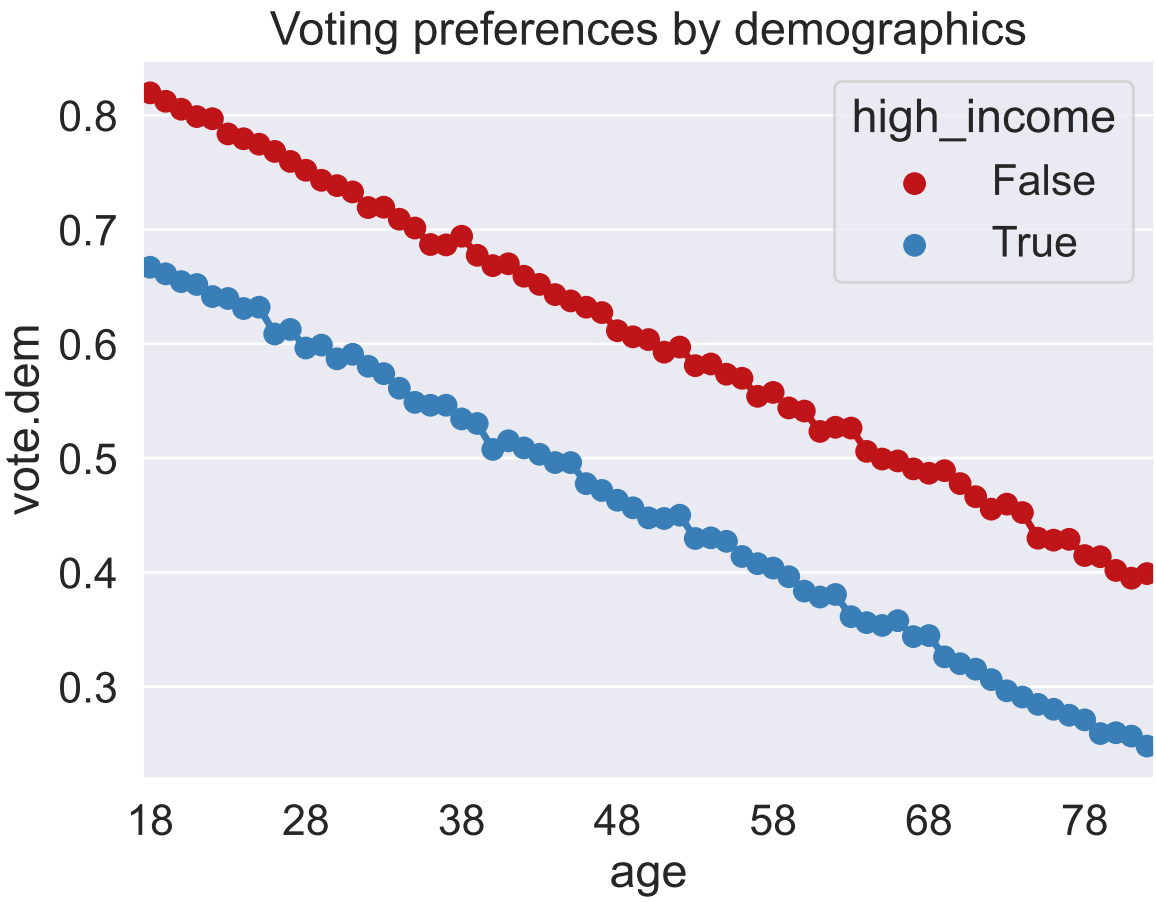
```
              rc={'figure.figsize':(7,5)})

# aggregate all voters by age
votes_by_demo = bearkeley.groupby(["age","high_income"]).agg("mean").reset_index()

fig = plt.figure();
red_blue = ["#bf1518", "#397eb7"]
with sns.color_palette(sns.color_palette(red_blue)):
    ax = sns.pointplot(data=votes_by_demo, x = "age", y = "vote.dem", hue = "high_income")

ax.set_title("Voting preferences by demographics")
fig.canvas.draw()
new_ticks = [i.get_text() for i in ax.get_xticklabels()];
plt.xticks(range(0, len(new_ticks), 10), new_ticks[::10]);
```

```
/var/folders/7x/09tm3ct91s14kpwnpd0400k00000gn/T/ipykernel_6050/4190445166.py:9: FutureWarni
  votes_by_demo = bearkeley.groupby(["age","high_income"]).agg("mean").reset_index()
```



From the plot above, we see that retirees in the imaginary city of Bearkeley tend to vote less Democrat, which skewed our predictions from our sample. We also note that high-income voters tend to vote less Democrat (and more Republican).

Let's compare our biased convenience sample to a simple random sample. Supposing we took an SRS the same size as our retiree sample, we see that we get a result very close to the actual vote:

```
## By default, replace = False
n = len(convenience_sample)
random_sample = bearkeley.sample(n, replace = False)

np.mean(random_sample["vote.dem"])
```

```
0.5308461975091543
```

This is very close to the actual vote!

We could even get pretty close with a *much smaller sample size*, say 800:

It turns out that we are pretty close, **much smaller sample size**, say, 800 (we'll learn how to choose this number when we introduce the Central Limit Theorem):

```
n = 800
random_sample = bearkeley.sample(n, replace = False)
np.mean(random_sample["vote.dem"])
```

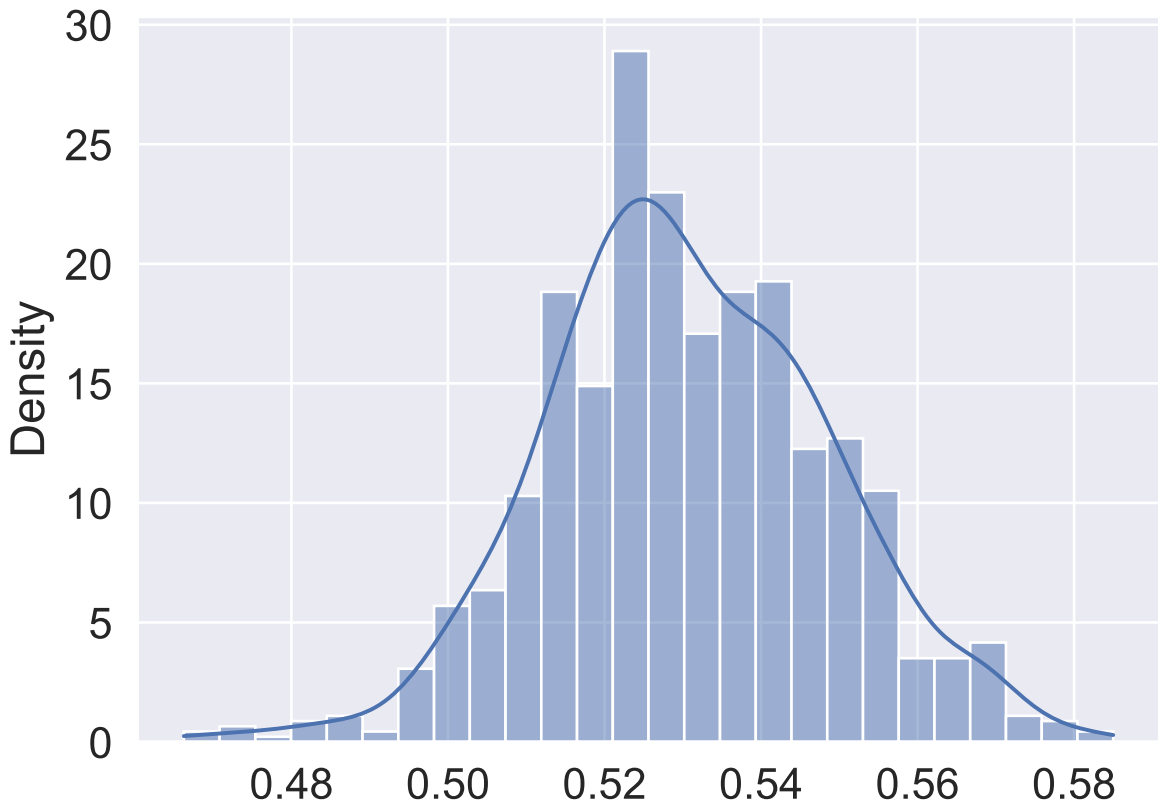
```
0.52375
```

To visualize the chance error in an SRS, let's simulate 1000 samples of the 800-size Simple Random Sample:

```
poll_result = []
nrep = 1000   # number of simulations
n = 800       # size of our sample
for i in range(0,nrep):
    random_sample = bearkeley.sample(n, replace = False)
    poll_result.append(np.mean(random_sample["vote.dem"]))
sns.histplot(poll_result, stat='density', kde=True)

# What fraction of these simulated samples would have predicted Democrat?
poll_result = pd.Series(poll_result)
np.sum(poll_result >= 0.5)/1000
```

```
0.964
```



A few observations: First, the KDE looks roughly Gaussian. Second, supposing that we predicted a Democratic winner if 50% of our sample voted Democrat, then just about 4% of our simulated samples would have predicted the election result incorrectly. This visualization further justifies why our convenience sample had error that was not entirely just due to chance. We'll revisit this notion later in the course.

10.6 Summary

Understanding the sampling process is what lets us go from describing the data to understanding the world. Without knowing / assuming something about how the data were collected, there is no connection between the sample and the population. Ultimately, the dataset doesn't tell us about the world behind the data.

11 Introduction to Modeling

> **Note**
>
> - Understand what models are and how to carry out the four-step modeling process
> - Define the concept of loss and gain familiarity with L1 and L2 loss
> - Fit a model using minimization techniques

Up until this point in the semester, we've focused on analyzing datasets. We've looked into the early stages of the data science lifecycle, focusing on the programming tools, visualization techniques, and data cleaning methods needed for data analysis.

This lecture marks a shift in focus. We will move away from examining datasets to actually *using* our data to better understand the world. Specifically, the next sequence of lectures will explore predictive modeling: generating models to make some prediction about the world around us. In this lecture, we'll introduce the conceptual framework for setting up a modeling task. In the next few lectures, we'll put this framework into practice by implementing several kinds of models.

11.1 What is a Model?

A model is an **idealized representation** of a system. A system is a set of principles or procedures according to which something functions. We live in a world full of systems: the procedure of turning on a light happens according to a specific set of rules dictating the flow of electricity. The truth behind how any event occurs are usually complex, and many times the specifics are unknown. The workings of the world can be viewed is its own giant procedure. Models seek to simplify the world and distill them it into workable pieces.

Example: We model the fall of an object on Earth as subject to a constant acceleration of $9.81 \frac{m}{s^2}$ due to gravity.

- While this describes the behavior of our system, it is merely an approximation.
- It doesn't account for the effects of air resistance, local variations in gravity, etc.
- In practice, it's accurate enough to be useful!

11.1.1 Reasons for building models

Often times, (1) we care about creating models that are simple and interpretable, allowing us to understand what the relationships between our variables are. Other times, (2) we care more about making extremely accurate predictions, at the cost of having an uninterpretable model. These are sometimes called black-box models, and are common in fields like deep learning.

1. To understand complex phenomena occurring in the world we live in.
    - What factors play a role in the growth of COVID-19?
    - How do an object's velocity and acceleration impact how far it travels? (Physics: $d = d_0 + vt + \frac{1}{2}at^2$)
2. To make accurate predictions about unseen data.
    - Can we predict if an email is spam or not?
    - Can we generate a one-sentence summary of this 10-page long article?

11.1.2 Common Types of Models

In general, models can be split into two categories:

Note: These specific models are not in the scope of Data 100 and exist to serve as motivation.

1. Deterministic physical (mechanistic) models: Laws that govern how the world works.
    - Kepler's Third Law of Planetary Motion (1619): The ratio of the square of an object's orbital period with the cube of the semi-major axis of its orbit is the same for all objects orbiting the same primary.
        - $T^2 \propto R^3$
    - Newton's Laws: motion and gravitation (1687): Newton's second law of motion models the relationship between the mass of an object and the force required to accelerate it.
        - $F = ma$
        - $F_g = G\frac{m_1 m_2}{r^2}$
2. Probabilistic models: models that attempt to understand how random processes evolve. These are more general and can be used describe many phenomena in the real world. These models commonly make simplifying assumption about the nature of the world.
    - Poisson Process models: Used to model random events that can happen with some probability at any point in time and are strictly increasing in count, such as the arrival of customers at a store.

11.2 Simple Linear Regression

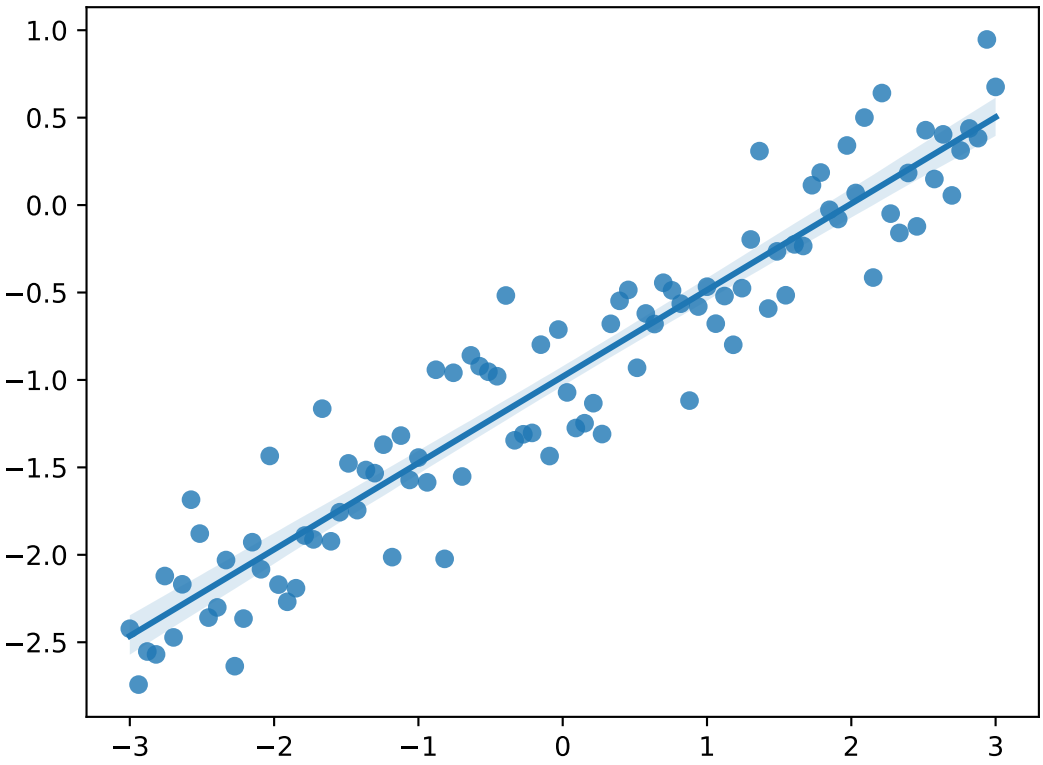
The **regression line** is the unique straight line that minimizes the **mean squared error** of estimation among all straight lines. As with any straight line, it can be defined by a slope and a y-intercept:

- slope: $r \cdot \frac{\text{Standard Deviation of y}}{\text{Standard Deviation of x}}$
- y-intercept: average of y $-$ slope $\cdot$ average of x

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# Set random seed for consistency
np.random.seed(43)
plt.style.use('default')

#Generate random noise for plotting
x = np.linspace(-3, 3, 100)
y = x * 0.5 - 1 + np.random.randn(100) * 0.3

#plot regression line
sns.regplot(x=x,y=y);
```



11.2.1 Definitions

For a random variable x:

- Mean: $\bar{x}$
- Standard Deviation: σ_x
- Predicted value: $\hat{x}$

11.2.1.1 Standard Units

A random variable is represented in standard units if the following are true:

1. 0 in standard units is the mean ($\bar{x}$) in the original variable's units.
2. An increase of 1 standard unit is an increase of 1 standard deviation(σ_x) in the original variable's units

11.2.1.2 Correlation

The correlation (r) is the average of the product of x and y, both measured in *standard units*. Correlation measures the strength of a linear association between two variables.

1. $r = \frac{1}{n}\sum_1^n(\frac{x_i-\bar{x}}{\sigma_x})(\frac{y_i-\bar{y}}{\sigma_y})$
2. Correlations are between -1 and 1: $|r| < 1$

```
def plot_and_get_corr(ax, x, y, title):
    ax.set_xlim(-3, 3)
    ax.set_ylim(-3, 3)
    ax.set_xticks([])
    ax.set_yticks([])
    ax.scatter(x, y, alpha = 0.73)
    r = np.corrcoef(x, y)[0, 1]
    ax.set_title(title + " (corr: {})".format(r.round(2)))
    return r

fig, axs = plt.subplots(2, 2, figsize = (10, 10))

# Just noise
x1, y1 = np.random.randn(2, 100)
corr1 = plot_and_get_corr(axs[0, 0], x1, y1, title = "noise")

# Strong linear
x2 = np.linspace(-3, 3, 100)
y2 = x2 * 0.5 - 1 + np.random.randn(100) * 0.3
corr2 = plot_and_get_corr(axs[0, 1], x2, y2, title = "strong linear")

# Unequal spread
x3 = np.linspace(-3, 3, 100)
y3 = - x3/3 + np.random.randn(100)*(x3)/2.5
corr3 = plot_and_get_corr(axs[1, 0], x3, y3, title = "strong linear")
extent = axs[1, 0].get_window_extent().transformed(fig.dpi_scale_trans.inverted())

# Strong non-linear
x4 = np.linspace(-3, 3, 100)
y4 = 2*np.sin(x3 - 1.5) + np.random.randn(100) * 0.3
corr4 = plot_and_get_corr(axs[1, 1], x4, y4, title = "strong non-linear")

plt.show()
```