Principles and Techniques of Data Science

Data 100

Kanu Grover Bella Crouch Dominic Liu Minh Phan
Matthew Shen Milad Shafaie

Table of contents

Welcome

This text offers supplementary resources to accompany lectures presented in the Summer 2023 iteration of the UC Berkeley course Data 100: Principles and Techniques of Data Science, taught by Bella Crouch and Dominic Liu.

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

If you spot any typos or would like to suggest changes, let us know! **Email**: data100.instructors@berkeley.edu

1 Introduction

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

Data science is an interdisciplinary field with a variety of applications. The field is rapidly evolving; many of the key technical underpinnings in modern-day data science were only popularized during the 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: the lifecycle starts either when we want to ask a question, or when we get a dataset.

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 obtaining data. A careful analysis of any problem requires the use of data. Sometimes, data may be readily available to us; other times, we may have to embark on a process to collect it. When doing so, it is crucial to ask the following:

- What data do we have and what data do we need?
  - Define the units of the data (people, cities, points in time, etc.) and what features to measure.
- How will we sample more data?
  - Scrape the web, collect manually, 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 cannot 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 in Data 100, we'll cover the underlying theory and technologies used in data science. It is our hope that, by the end of the course, you start to see yourself as a data scientist.

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

2 Pandas I

> **Learning Outcomes**
>
> - Build familiarity with basic `pandas` syntax
> - Learn key data structures: DataFrames, Series, and Indices
> - Understand methods for extracting data: `.loc`, `.iloc`, and `[ ]`

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

2.1 Tabular Data

Data scientists work with data stored in a variety of formats. The primary focus of this class is in understanding *tabular data* —- data that is stored in a table.

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

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

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

	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
...	...	...	...	...	...	...

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

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

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

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

2.2 DataFrames, Series, and Indices

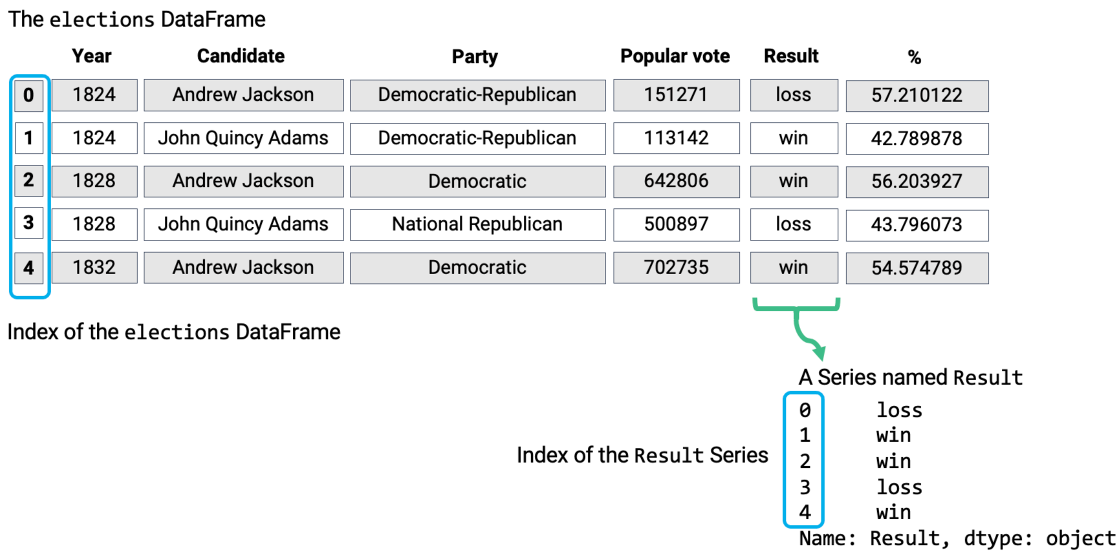
To begin our studies in `pandas`, we must first import the library into our Python environment. This will allow us to use `pandas` data structures and methods in our code.

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

There are three fundamental data structures in `pandas`:

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

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



	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

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

2.2.1 Series

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

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

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

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

```
array([-1, 10,  2])
```

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

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

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

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

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

Indices can also be changed after initialization.

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

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

2.2.1.1 Selection in Series

Much like when working with NumPy arrays, we can select a single value or a set of values from a Series. There are 3 primary methods of selecting data.

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

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

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

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

2.2.1.1.1 A Single Index Label

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

```
4
```

2.2.1.1.2 A List of Index Labels

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

```
a    4
c    0
dtype: int64
```

2.2.1.1.3 A Filtering Condition

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

First, we apply a boolean condition to the Series. This create **a new Series of boolean values**.

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

```
a     True
b    False
c    False
d     True
dtype: bool
```

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

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

```
a    4
d    6
dtype: int64
```

2.2.2 DataFrames

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

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

```
import pandas as pd

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

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

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 datasets 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("data/elections.csv")`

This code stores our DataFrame object in the `elections` variable. We see that 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 also create a DataFrame with data of our own.

2.2.2.1 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

2.2.2.1.1 Using a List and Column Names

Consider the following examples. The first code cell creates a DataFrame with a single column `Numbers`. The second creates a DataFrame with the columns `Numbers` and `Description`. Notice how a 2D list of values is required to initialize the second DataFrame – each nested list represents a single row of data.

```
df_list_1 = pd.DataFrame([1, 2, 3], columns=["Numbers"])
df_list_1
```

	Numbers
0	1
1	2
2	3

```
df_list_2 = pd.DataFrame([[1, "one"], [2, "two"]], columns = ["Number", "Description"])
df_list_2
```

	Number	Description
0	1	one
1	2	two

2.2.2.1.2 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
```

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

2.2.2.1.3 From a Series

Earlier, we noted that a Series is usually thought of as 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})
```

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

2.2.3 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 presidential candidates.

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

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

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

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

2.3 Slicing in DataFrames

Now that we've learned how to create DataFrames, let's dive more deeply 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 four primary methods of the DataFrame class:

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

2.3.1 Extracting data with `.head` and `.tail`

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

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

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

	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
4	Andrew Jackson	1832	Democratic	702735	win	54.574789

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

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

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

2.3.2 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 the rows from labels `0` to `3` and the columns from labels `"Year"` to `"Popular vote"`.

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

	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, :]
```

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

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

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

	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.

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

	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

2.3.3 Indexing with .iloc

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

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

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

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

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

	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*. In other words, the right-end of a slice is not included when using `.iloc`. This is one of the subtleties of `pandas` syntax; you will get used to it with practice.

List behavior works just as expected.

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

	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.

2.3.4 Indexing with []

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

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

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

2.3.4.1 A slice of row numbers

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

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

	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

	Candidate	Year	Party	Popular vote	Result	%
2	Andrew Jackson	1828	Democratic	642806	win	56.203927
3	John Quincy Adams	1828	National Republican	500897	loss	43.796073

2.3.4.2 A list of column labels

Suppose we now want the first four columns.

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

	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
...	...	...	...	...
177	2016	Jill Stein	Green	1457226
178	2020	Joseph Biden	Democratic	81268924
179	2020	Donald Trump	Republican	74216154
180	2020	Jo Jorgensen	Libertarian	1865724
181	2020	Howard Hawkins	Green	405035

2.3.4.3 A single column label

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

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

```
0          Andrew Jackson
1       John Quincy Adams
2          Andrew Jackson
3       John Quincy Adams
4          Andrew Jackson
              ...
177            Jill Stein
178          Joseph Biden
179          Donald Trump
```

```
180         Jo Jorgensen
181       Howard Hawkins
Name: Candidate, Length: 182, dtype: object
```

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

2.4 Parting Note

The `pandas` library is enormous and contains many useful functions. Here is a link to documentation. We certainly don't expect you to memorize each and every method of the library.

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

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

3 Pandas II

> **Learning Outcomes**
>
> - 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.

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

data_url = "https://www.ssa.gov/oact/babynames/state/namesbystate.zip"
local_filename = "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:
```

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

babynames.head()
```

	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 or `Series` where each element is either `True` or `False`. This boolean array must have a length equal to the number of rows in the DataFrame. It will return all rows that correspond to a value of `True` in the array. We used a very similar technique when performing conditional extraction from a `Series` in the last lecture.

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

```python
# 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]]
```

	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
```

	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()
```

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

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

Here, `logical_operator` evaluates to a `Series` of boolean values with length 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`. Rows from 235791 onwards evaluate to `False` and are omitted from the output.

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

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

Boolean conditions can be combined using various bitwise operators that allow us to filter results by multiple conditions.

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

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

For example, if we want to return data on all names with sex "F" born before the 21st century, we can write:

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Adding, Removing, and Modifying Columns

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

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

```
# Create a Series of the length of each name. We'll discuss `str` methods next week.
babyname_lengths = babynames["Name"].str.len()

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

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

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

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

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

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

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

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

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

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

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

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

In other words, if we simply call:

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

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

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

3.3 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.

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

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

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

3.3.1 `NumPy`

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

```
# Pull out the number of babies named Bella each year
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 in any one year
np.max(bella_counts)
```

```
902
```

3.3.2 `.shape` and `.size`

`.shape` and `.size` are attributes of `Series` and `DataFrame`s 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.

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

```
(400762, 5)
```

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

```
2003810
```

3.3.3 `.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()
```

	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()
```

```
count     400762
unique         2
```

```
top             F
freq       235791
Name: Sex, dtype: object
```

3.3.4 `.sample()`

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

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

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

	State	Sex	Year	Name	Count
325642	CA	M	1994	Kiel	7

```python
# Sample 5 random rows
babynames.sample(5)
```

	State	Sex	Year	Name	Count
48802	CA	F	1963	Christiana	8
167533	CA	F	2004	Emalie	6
46651	CA	F	1962	Lonnie	14
362440	CA	M	2008	Shia	7
54138	CA	F	1966	Cary	11

```python
# Randomly sample 4 names from the year 2000, with replacement
babynames[babynames["Year"] == 2000].sample(4, replace = True)
```

	State	Sex	Year	Name	Count
151959	CA	F	2000	Aubry	6
151930	CA	F	2000	Amulya	6
150128	CA	F	2000	Arleth	26

	State	Sex	Year	Name	Count
339796	CA	M	2000	Augustin	14

3.3.5 `.value_counts()`

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

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

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

```
Name
Jean         221
Francis      219
Guadalupe    216
Jessie       215
Marion       213
Name: count, dtype: int64
```

3.3.6 `.unique()`

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

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

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

3.3.7 `.sort_values()`

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

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

	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.

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

```
380256      Aadan
362255      Aadan
365374      Aadan
394460    Aadarsh
366561      Aaden
Name: Name, dtype: object
```

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 0x104b78410>
```

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.

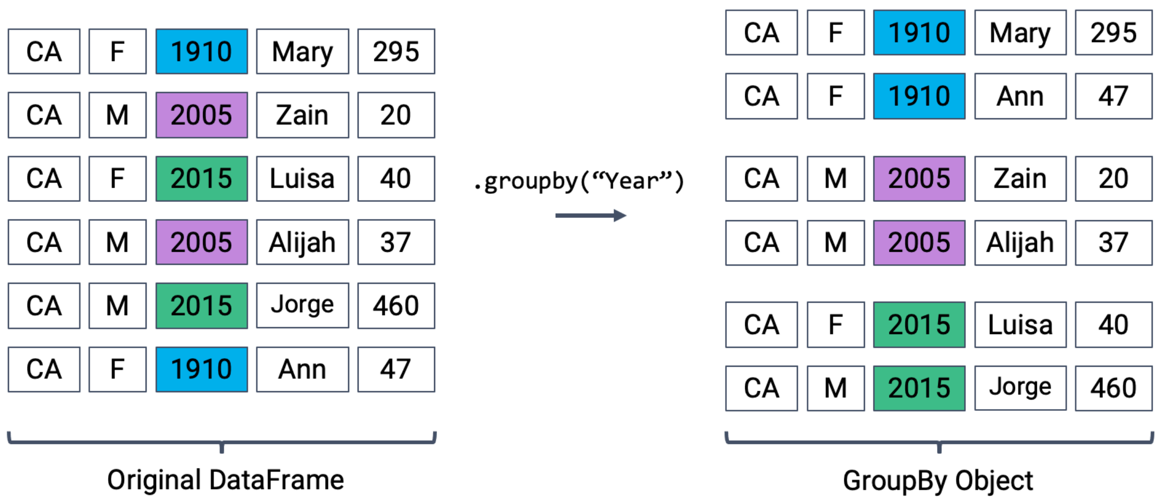


Figure 3.1: Creating a GroupBy object

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)
```

	State	Sex
Year		
1910	CACACACACACACACACACACACACACACACACACACACACACA...	FFFFFFFFFFFFFFF
1911	CACACACACACACACACACACACACACACACACACACACACACA...	FFFFFFFFFFFFFFF
1912	CACACACACACACACACACACACACACACACACACACACACACA...	FFFFFFFFFFFFFFF
1913	CACACACACACACACACACACACACACACACACACACACACACA...	FFFFFFFFFFFFFFF

	State	Sex
Year		
1914	CACACACACACACACACACACACACACACACACACACACACACACACA...	FFFFFFFFFFFFFFFF

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.

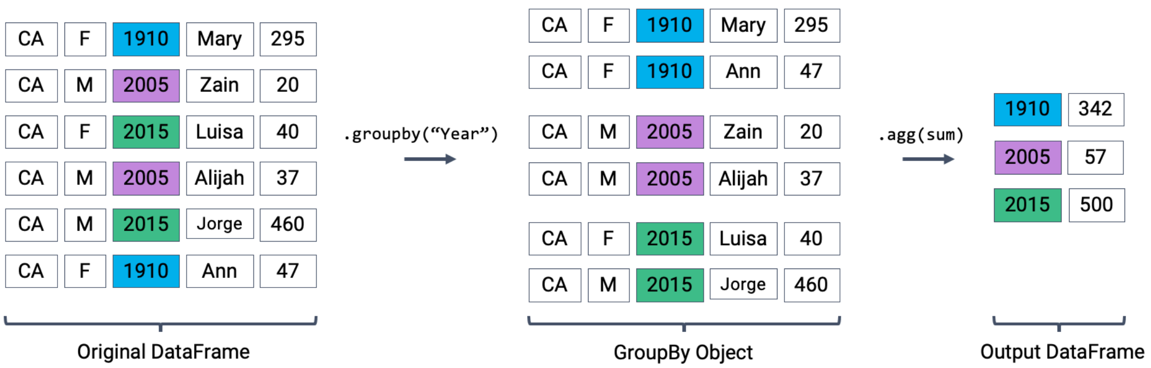


Figure 3.2: Performing an aggregation

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 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)
```

	Count
Year	
1910	9163
1911	9983
1912	17946

Year	Count
1913	22094
1914	26926

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

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

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

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

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

Name	Count
Aadan	7
Aadarsh	6
Aaden	158
Aadhav	8
Aadhira	10

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

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

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

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

	Count
Name	
Aadan	6.000000
Aadarsh	6.000000
Aaden	46.214286
Aadhav	6.750000
Aadhira	7.250000

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

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

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

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

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

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

	Name	First Letter	Year
0	Mary	M	1910
1	Helen	H	1910
2	Dorothy	D	1910
3	Margaret	M	1910
4	Frances	F	1910

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

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

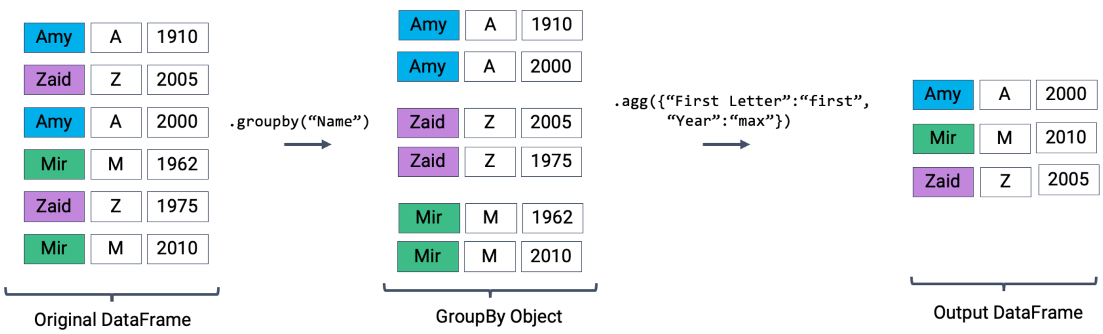


Figure 3.3: Aggregating using "first"

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

	First Letter	Year
Name		
Aadan	A	2014
Aadarsh	A	2019
Aaden	A	2020
Aadhav	A	2019
Aadhira	A	2021

Some aggregation functions are common enough that `pandas` allows them to be called directly, without the explicit use of `.agg`.

```python
babynames.groupby("Name")[["Count"]].mean().head()
```

Name	Count
Aadan	6.000000
Aadarsh	6.000000
Aaden	46.214286
Aadhav	6.750000
Aadhira	7.250000

We can also define aggregation functions of our own! This can be done using either a `def` or `lambda` statement. Again, the condition for a custom aggregation function is that it must take in a `Series` and output a single scalar value.

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

babynames.groupby("Name")[["Year", "Count"]].agg(ratio_to_peak)
```

Name	Year	Count
Aadan	1.0	0.714286
Aadarsh	1.0	1.000000
Aaden	1.0	0.063291
Aadhav	1.0	0.750000
Aadhira	1.0	0.700000
...	...	...
Zymir	1.0	1.000000
Zyon	1.0	0.933333
Zyra	1.0	1.000000
Zyrah	1.0	0.833333
Zyrus	1.0	1.000000

```python
# Alternatively, using lambda
babynames.groupby("Name")[["Year", "Count"]].agg(lambda s: s.iloc[-1]/max(s))
```

	Year	Count
Name		
Aadan	1.0	0.714286
Aadarsh	1.0	1.000000
Aaden	1.0	0.063291
Aadhav	1.0	0.750000
Aadhira	1.0	0.700000
...	...	...
Zymir	1.0	1.000000
Zyon	1.0	0.933333
Zyra	1.0	1.000000
Zyrah	1.0	0.833333
Zyrus	1.0	1.000000

3.5 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

> **Learning Outcomes**
>
> - 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 `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:

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

4.1.1 Aggregation with `lambda` Functions

We'll work with the `elections` DataFrame again.

```
import pandas as pd
import numpy as np

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

	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

	Year	Candidate	Party	Popular vote	Result	%
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)
```

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 sub-DataFrame

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)
```

	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)
```

	Year	Candidate	Popular vote	Result	%
Party					
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

Here's an illustration of the process:

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)
```

	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)
```

	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

4.1.2 Other `GroupBy` Features

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

- `.mean`: creates a new DataFrame with the mean value of each group
- `.sum`: creates a new DataFrame with the sum of each group
- `.max` and `.min`: creates a new DataFrame with the maximum/minimum value of each group

- `.first` and `.last`: creates a new DataFrame with the first/last row in each group
- `.size`: creates a new **Series** with the number of entries in each group
- `.count`: creates a new **DataFrame** with the number of entries, excluding missing values.

Note the slight difference between `.size()` and `.count()`: while `.size()` returns a Series and counts the number of entries including the missing values, `.count()` returns a DataFrame and counts the number of entries in each column excluding missing values. Here's an example:

```
df = pd.DataFrame({'letter':['A','A','B','C','C','C'],
                   'num':[1,2,3,4,np.NaN,4],
                   'state':[np.NaN, 'tx', 'fl', 'hi', np.NaN, 'ak']})
df
```

	letter	num	state
0	A	1.0	NaN
1	A	2.0	tx
2	B	3.0	fl
3	C	4.0	hi
4	C	NaN	NaN
5	C	4.0	ak

```
df.groupby("letter").size()
```

```
letter
A    2
B    1
C    3
dtype: int64
```

```
df.groupby("letter").count()
```

	num	state
letter		
A	2	1
B	1	1
C	2	2

You might recall that the `value_counts()` function in the previous note does something similar. It turns out `value_counts()` and `groupby.size()` are the same, except `value_counts()` sorts the resulting Series in descending order automatically.

```
df["letter"].value_counts()
```

```
letter
C    3
A    2
B    1
Name: count, dtype: int64
```

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()`

There are many other methods that `pandas` supports. You can check them out on the [`pandas` documentation].

4.1.3 Filtering by Group

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 DataFrame object as input
- Returns a single `True` or `False` for the each sub-DataFrame

Sub-DataFrames 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 an *entire* sub-DataFrame is returned in the final DataFrame, not just a single row. As a result, `groupby.filter` preserves the original indices.

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.

In other words, we want to:

- 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)
```

	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 sub-DataFrame, 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 sub-DataFrame 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.

4.2 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 column names to `.groupby`.

Let's consider the `babynames` dataset. 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()
```

	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

```python
# 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)
```

Year	Sex	Count
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 *multi-index* DataFrame 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 multi-indexed DataFrames 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 pivot 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.

```python
# 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).head(5)
```

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

Sex Year	F	M
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 summing 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)
```

	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

4.3 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)
```

	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 popularity of the names of each presidential candidate in 2020. 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)
```

	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()
```

	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.

```
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
```

	Year_x	Candidate	Party	Popular vote	Result	%	First Name	Na
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

Let's take a closer look at the parameters:

- `left` and `right` parameters are used to specify the DataFrames to be joined.
- `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.

4.4 Parting Note

Congratulations! We finally tackled `pandas`. Don't worry if you are still not feeling very comfortable with it—you will have plenty of chance to practice over the next few weeks.

Next, we will get our hands dirty with some real-world datasets and use our `pandas` knowledge to conduct some exploratory data analysis.

5 Data Cleaning and EDA

> **ℹ Learning Outcomes**
>
> - 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)
```

	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.

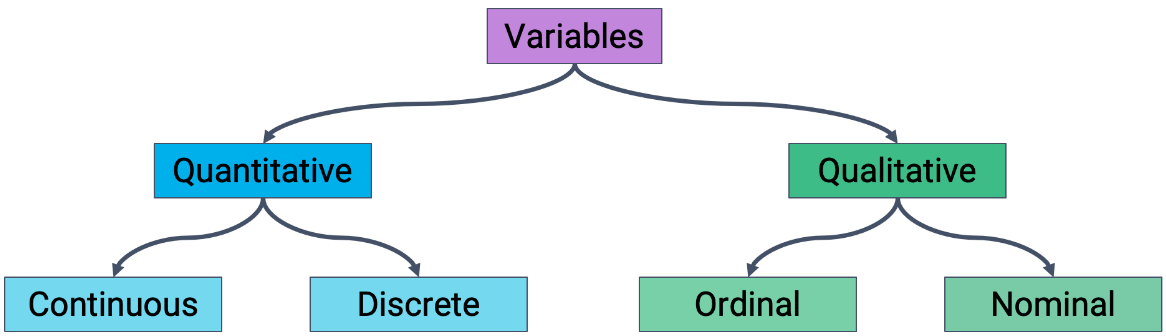


Figure 5.1: Classification of variable types

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.

	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.

	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 what a single row represents. You can also think of it as 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:

```python
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.

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

	Unnamed: 0	No. of TB cases	Unnamed: 2	Unnamed: 3	TB incidence	Unnamed: 5	Unnamed
0	U.S. jurisdiction	2019	2020	2021	2019.00	2020.00	2021.00
1	Total	8,900	7,173	7,860	2.71	2.16	2.37
2	Alabama	87	72	92	1.77	1.43	1.83
3	Alaska	58	58	58	7.91	7.92	7.92
4	Arizona	183	136	129	2.51	1.89	1.77

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)
```

	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)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
0	Total	8,900	7,173	7,860	2.71	2.16
1	Alabama	87	72	92	1.77	1.43
2	Alaska	58	58	58	7.91	7.92
3	Arizona	183	136	129	2.51	1.89
4	Arkansas	64	59	69	2.12	1.96

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)
```

```
U.S. jurisdiction    TotalAlabamaAlaskaArizonaArkansasCaliforniaCol...
TB cases 2019        8,9008758183642,111666718245583029973261085237...
TB cases 2020        7,1737258136591,706525417194122219282169239376...
TB cases 2021        7,8609258129691,750585443194992281064255127494...
TB incidence 2019                                               109.94
TB incidence 2020                                                93.09
TB incidence 2021                                               102.94
dtype: object
```

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

```
tb_df.dtypes
```

```
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
dtype: object
```

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:

```python
# 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)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
0	Total	8900	7173	7860	2.71	2.16
1	Alabama	87	72	92	1.77	1.43
2	Alaska	58	58	58	7.91	7.92
3	Arizona	183	136	129	2.51	1.89
4	Arkansas	64	59	69	2.12	1.96

```python
tb_df.sum()
```

```
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
dtype: object
```

The Total TB cases look right. Phew!

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

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

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
1	Alabama	87	72	92	1.77	1.43
2	Alaska	58	58	58	7.91	7.92
3	Arizona	183	136	129	2.51	1.89
4	Arkansas	64	59	69	2.12	1.96
5	California	2111	1706	1750	5.35	4.32

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)
```

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

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)
```

	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)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
0	Alabama	87	72	92	1.77	1.43
1	Alaska	58	58	58	7.91	7.92
2	Arizona	183	136	129	2.51	1.89
3	Arkansas	64	59	69	2.12	1.96
4	California	2111	1706	1750	5.35	4.32

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)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
0	Alabama	87	72	92	1.77	1.43
1	Alaska	58	58	58	7.91	7.92
2	Arizona	183	136	129	2.51	1.89
3	Arkansas	64	59	69	2.12	1.96
4	California	2111	1706	1750	5.35	4.32

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)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
0	Alabama	87	72	92	1.77	1.43
1	Alaska	58	58	58	7.91	7.92
2	Arizona	183	136	129	2.51	1.89
3	Arkansas	64	59	69	2.12	1.96
4	California	2111	1706	1750	5.35	4.32

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)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 202
0	Alabama	87	72	92	1.77	1.43
1	Alaska	58	58	58	7.91	7.92
2	Arizona	183	136	129	2.51	1.89
3	Arkansas	64	59	69	2.12	1.96
4	California	2111	1706	1750	5.35	4.32

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()
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB incide
count	51.000000	51.000000	51.000000	51.000000	51.000000	51.000000
mean	174.509804	140.647059	154.117647	2.102549	1.782941	1.971961
std	341.738752	271.055775	286.781007	1.498745	1.337414	1.478468
min	1.000000	0.000000	2.000000	0.170000	0.000000	0.210000
25%	25.500000	29.000000	23.000000	1.295000	1.210000	1.235000
50%	70.000000	67.000000	69.000000	1.800000	1.520000	1.700000
75%	180.500000	139.000000	150.000000	2.575000	1.990000	2.220000
max	2111.000000	1706.000000	1750.000000	7.910000	7.920000	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)
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
U.S. jurisdiction					
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

U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
Arkansas	64	59	69	2.12	1.96

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

Geographic Area	2010	2011	2012	2013	2014	2015	2016	20
United States	309321666	311556874	313830990	315993715	318301008	320635163	322941311	32
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	56
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	68
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	12
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	77

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

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:

```python
tb_df.head()
```

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

U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
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)
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	20
United States	309321666	311556874	313830990	315993715	318301008	320635163	322941311	32
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	56
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	68
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	12
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	77

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)
```

Geographic Area	2010	2011	2012	2013	2014	2015	2016	20
Total	309321666	311556874	313830990	315993715	318301008	320635163	322941311	32
Northeast	55380134	55604223	55775216	55901806	56006011	56034684	56042330	56
Midwest	66974416	67157800	67336743	67560379	67745167	67860583	67987540	68
South	114866680	116006522	117241208	118364400	119624037	120997341	122351760	12
West	72100436	72788329	73477823	74167130	74925793	75742555	76559681	77

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

	2020	2021	2022
Geographic Area			
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).

```python
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)
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB inc
Total	8900	7173	7860	2.71	2.16	2.37
Alabama	87	72	92	1.77	1.43	1.83
Alaska	58	58	58	7.91	7.92	7.92
Arizona	183	136	129	2.51	1.89	1.77
Arkansas	64	59	69	2.12	1.96	2.28

Finally, let's recompute our incidences:

```python
# 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)
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB inc
Total	8900	7173	7860	2.71	2.16	2.37
Alabama	87	72	92	1.77	1.43	1.83
Alaska	58	58	58	7.91	7.92	7.92
Arizona	183	136	129	2.51	1.89	1.77
Arkansas	64	59	69	2.12	1.96	2.28

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
```