Principles and Techniques of Data Science

Data 100

Bella Crouch Yash Dave Ian Dong Kanu Grover
Ishani Gupta Minh Phan Nikhil Reddy Milad Shafaie
Matthew Shen Lillian Weng Prabhleen Kaur Xiaorui Liu

Table of contents

Welcome

About the Course Notes

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

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

If you spot any typos or would like to suggest any changes, please email us at **data100.instructors@berkeley.edu**.

1 Introduction

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

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

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

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

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

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

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

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

Course Goals

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

Some Topics We'll Cover

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

Prerequisites

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

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

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

1.1 Data Science Lifecycle

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

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

1.1.1 Ask a Question

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

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

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

1.1.2 Obtain Data

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

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

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

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

1.1.3 Understand the Data

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

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

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

1.1.4 Understand the World

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

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

- What does the data say about the world?

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

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

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

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

1.2 Conclusion

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

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

2 Pandas I

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

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

2.1 Tabular Data

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

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

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

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

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

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

Using `pandas`, we can

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

2.2 `Series`, `DataFrames`, and Indices

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

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

There are three fundamental data structures in `pandas`:

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

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

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

2.2.1 Series

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

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

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

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

```
0     welcome
1          to
2    data 100
dtype: object
```

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

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

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

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

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

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

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

```
s.index
```

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

Indices can also be changed after initialization.

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

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

```
s.index
```

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

2.2.1.1 Selection in `Series`

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

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

To demonstrate this, let's define a new Series `s`.

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

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

2.2.1.1.1 A Single Label

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

```
4
```

2.2.1.1.2 A List of Labels

```
# We return a Series of the values stored at the index labels "a" and "c"
s[["a", "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 by using a filtering condition.

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

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

```
s[s > 0]
```

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

2.2.2 `DataFrames`

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

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

2.2.2.1 Creating a `DataFrame`

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

1. From a CSV file.
2. Using a list and column name(s).

3. From a dictionary.
4. From a `Series`.

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

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

2.2.2.1.1 From a CSV file

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

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

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

	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

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

2.2.2.1.2 Using a List and Column Name(s)

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

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

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

	Numbers
0	1
1	2
2	3

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

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

	Number	Description
0	1	one
1	2	two

2.2.2.1.3 From a Dictionary

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

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

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

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

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

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

2.2.2.1.4 From a `Series`

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

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

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

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

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

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

	0
r1	a1
r2	a2
r3	a3

```
s_b.to_frame()
```

	0
r1	b1
r2	b2
r3	b3

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

```
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

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.

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

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

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

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

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

Party	Candidate	Year	Popular vote	Result	%
Democratic-Republican	Andrew Jackson	1824	151271	loss	57.210122
Democratic-Republican	John Quincy Adams	1824	113142	win	42.789878
Democratic	Andrew Jackson	1828	642806	win	56.203927
National Republican	John Quincy Adams	1828	500897	loss	43.796073
Democratic	Andrew Jackson	1832	702735	win	54.574789
...	...	...	...	...	...
Green	Jill Stein	2016	1457226	loss	1.073699
Democratic	Joseph Biden	2020	81268924	win	51.311515
Republican	Donald Trump	2020	74216154	loss	46.858542
Libertarian	Jo Jorgensen	2020	1865724	loss	1.177979
Green	Howard Hawkins	2020	405035	loss	0.255731

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

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

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

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

Here we see unique and numeric index values.

However, here the index values are not unique.

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

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

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

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

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

For column labels, use `DataFrame.columns`:

```
elections.columns
```

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

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

```
elections.shape
```

```
(182, 6)
```

2.4 Slicing in `DataFrames`

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

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

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

Common ways we may want to extract data are grabbing:

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

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

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

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

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

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

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

```
# Extract the first 5 rows of the DataFrame
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

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)
```

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

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

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

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

Arguments to `.loc` can be:

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

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

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

```
'Andrew Jackson'
```

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

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

```
87     15761254
25       848019
179    74216154
Name: Popular vote, dtype: int64
```

Note that if we pass `"Popular vote"` as a list, the output will be a `DataFrame`.

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

	Popular vote
87	15761254
25	848019
179	74216154

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

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

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

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

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

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122

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

We can use the same shorthand to extract all rows.

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

	Year	Candidate	Result
0	1824	Andrew Jackson	loss
1	1824	John Quincy Adams	win
2	1828	Andrew Jackson	win
3	1828	John Quincy Adams	loss
4	1832	Andrew Jackson	win
...	...	...	...
177	2016	Jill Stein	loss
178	2020	Joseph Biden	win
179	2020	Donald Trump	loss
180	2020	Jo Jorgensen	loss
181	2020	Howard Hawkins	loss

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

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

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

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

Lastly, we can interchange list and slicing notation.

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

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

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

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

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

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

```
'Andrew Jackson'
```

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

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

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

```
1    John Quincy Adams
2       Andrew Jackson
3    John Quincy Adams
Name: Candidate, dtype: object
```

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]
```

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

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

List behavior works just as expected.

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

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

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

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

	Year	Candidate	Party
0	1824	Andrew Jackson	Democratic-Republican
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican
4	1832	Andrew Jackson	Democratic
...	...	...	...
177	2016	Jill Stein	Green
178	2020	Joseph Biden	Democratic

	Year	Candidate	Party
179	2020	Donald Trump	Republican
180	2020	Jo Jorgensen	Libertarian
181	2020	Howard Hawkins	Green

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

Overall, it is important to remember that:

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

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

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

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

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

2.4.4.1 A slice of row numbers

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

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

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

2.4.4.2 A list of column labels

Suppose we now want the first four columns.

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

	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

2.4.4.3 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`, especially since it is far more concise.

2.5 Parting Note

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

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

With that, we will move on to Pandas II!

3 Pandas II

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

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

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

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

```python
# 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 own
import pandas as pd
import numpy as np
import urllib.request
import os.path
import zipfile

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

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

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

```
babynames.head()
```

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

3.1 Conditional Selection

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

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

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

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

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

	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`.

```python
babynames_first_10_rows.loc[[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

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

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

```python
# 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 407428.

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

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

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

```
print("The 0th item in this 'logical_operator' is: {}".format(logical_operator.iloc[0]))
print("The 239536th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2395
print("The 239537th item in this 'logical_operator' is: {}".format(logical_operator.iloc[2395
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

	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 multipl
(
    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 results to the `DataFrame` above with far more concise code.

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

```
0    False
1    False
2    False
3    False
4    False
Name: Name, dtype: bool
```

```
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.

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

```
0    False
1    False
2    False
3    False
4    False
Name: Name, dtype: bool
```

```
# Extracting 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.
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

	State	Sex	Year	Name	Count	name_lengths
4	CA	F	1910	Frances	134	7

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

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

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

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

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

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

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

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

In other words, if we simply call:

```python
# 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 Useful Utility Functions

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

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

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

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

3.3.1 NumPy

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

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

```
331824     8
334114     9
336390    11
338773    12
341387    10
Name: Count, dtype: int64
```

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

```
17.142857142857142
```

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

```
29
```

3.3.2 `.shape` and `.size`

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

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

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

```
(407428, 5)
```

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

```
2037140
```

3.3.3 `.describe()`

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

```
babynames.describe()
```

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

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

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

```
count     407428
unique         2
top            F
freq      239537
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()` (documentation) lets us quickly select random entries (a row if called from a `DataFrame`, or a value if called from a `Series`).

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

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

	State	Sex	Year	Name	Count
229638	CA	F	2020	Mayra	26

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

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

	Year	Name	Count
189843	2010	Skylar	104
201612	2012	Yu	5
192014	2010	Sofiya	9
266117	1955	Chris	252
94028	1984	Wendy	469

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

	Year	Name	Count
344275	2000	Kenan	7
152244	2000	Sherrie	6
343016	2000	Darrell	40
150175	2000	Iman	25

3.3.5 `.value_counts()`

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

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

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

```
Name
Jean         223
Francis      221
Guadalupe    218
Jessie       217
Marion       214
Name: count, dtype: int64
```

3.3.6 `.unique()`

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

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

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

3.3.7 `.sort_values()`

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

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

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

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

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

```
366001      Aadan
384005      Aadan
369120      Aadan
398211    Aadarsh
370306      Aaden
Name: Name, dtype: object
```

3.4 Parting Note

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

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

4 Pandas III

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

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

4.1 Custom Sorts

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

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

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

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

```
        f.write(resp.read())

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

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

babynames.tail(10)
```

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

4.1.1 Approach 1: Create a Temporary Column

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

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

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

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	4
1	CA	F	1910	Helen	239	5

	State	Sex	Year	Name	Count	name_lengths
2	CA	F	1910	Dorothy	220	7
3	CA	F	1910	Margaret	163	8
4	CA	F	1910	Frances	134	7

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

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

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

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

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

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

4.1.2 Approach 2: Sorting using the `key` Argument

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

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

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

4.1.3 Approach 3: Sorting using the `map` Function

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

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

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

# Sort the DataFrame by the new "dr_ea_count" column so we can see our handiwork
babynames = babynames.sort_values(by="dr_ea_count", ascending=False)
babynames.head()
```

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

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

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

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

4.2 Aggregating Data with `.groupby`

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

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

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

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x167cf1d50>
```

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

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

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

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

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

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

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

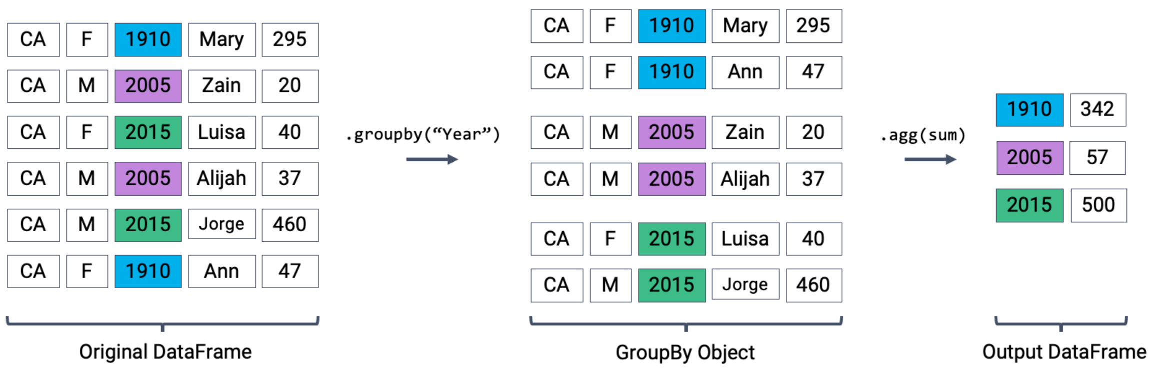


Figure 4.1: Performing an aggregation

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.

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

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

	Count
Year	
1910	5

Year	Count
1911	5
1912	5
1913	5
1914	5

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

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

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

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

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

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

4.2.1 Aggregation Functions

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

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

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

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

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

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

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

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

Name	Count
Aadan	6.000000
Aadarsh	6.000000
Aaden	46.214286
Aadhav	6.750000

Name	Count
Aadhini	6.000000

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

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

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

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

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

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

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

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

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

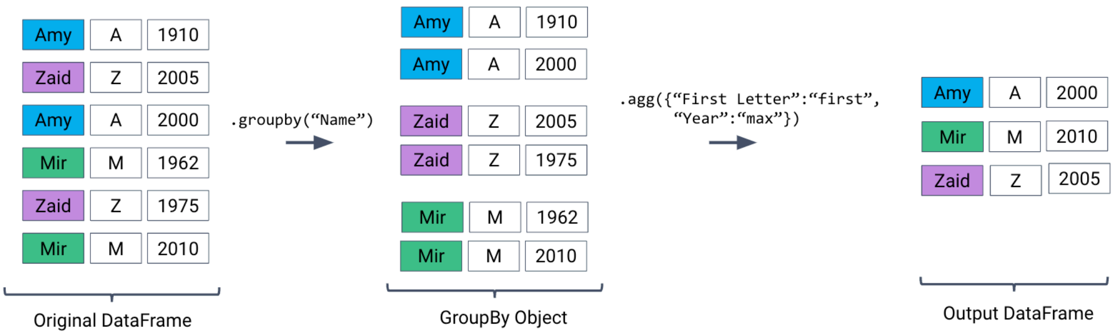


Figure 4.2: Aggregating using "first"

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

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

4.2.2 Plotting Birth Counts

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

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

```
/var/folders/gr/vb80r2qs5td4rqbnv4dn2klh0000gn/T/ipykernel_19239/390646742.py:1: FutureWarni

The provided callable <built-in function sum> is currently using DataFrameGroupBy.sum. In a 
```

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

Here's an illustration of the process:

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

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

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

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

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

4.2.3 Summary of the `.groupby()` Function

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

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

- **Splits** the `DataFrame` into sub-`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`.

4.2.4 Revisiting the `.agg()` Function

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

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

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

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

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

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

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

```
0.018796372629843364
```

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

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

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

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

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

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

4.2.5 Nuisance Columns

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

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

4.2.6 Renaming Columns After Grouping

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

```
rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"})
rtp_table
```

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

Name	Year	Count RTP
Zyanya	1.0	0.466667
Zyla	1.0	1.000000
Zylah	1.0	1.000000
Zyra	1.0	1.000000
Zyrah	1.0	0.833333

4.2.7 Some Data Science Payoff

By sorting `rtp_table`, we can see the names whose popularity has decreased the most.

```
rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"})
rtp_table.sort_values("Count RTP").head()
```

Name	Year	Count RTP
Debra	1.0	0.001260
Debbie	1.0	0.002815
Carol	1.0	0.003180
Tammy	1.0	0.003249
Susan	1.0	0.003305

To visualize the above `DataFrame`, let's look at the line plot below:

```
import plotly.express as px
px.line(f_babynames[f_babynames["Name"] == "Debra"], x = "Year", y = "Count")
```

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

We can get the list of the top 10 names and then plot popularity with the following code:

```
top10 = rtp_table.sort_values("Count RTP").head(10).index
px.line(
    f_babynames[f_babynames["Name"].isin(top10)],
    x = "Year",
    y = "Count",
    color = "Name"
)
```

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

As a quick exercise, consider what code would compute the total number of babies with each name.

```python
babynames.groupby("Name")[["Count"]].agg("sum").head()
# alternative solution:
# babynames.groupby("Name")[["Count"]].sum()
```

	Count
Name	
Aadan	18
Aadarsh	6
Aaden	647
Aadhav	27
Aadhini	6

4.3 `.groupby()`, Continued

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

```python
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
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

4.3.1 Raw `GroupBy` Objects

The result of `groupby` applied to a `DataFrame` is a `DataFrameGroupBy` object, **not** a `DataFrame`.

```
grouped_by_year = elections.groupby("Year")
type(grouped_by_year)
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

There are several ways to look into `DataFrameGroupBy` objects:

```
grouped_by_party = elections.groupby("Party")
grouped_by_party.groups
```

```
{'American': [22, 126], 'American Independent': [115, 119, 124], 'Anti-Masonic': [6], 'Anti-I
Labor': [78], 'Free Soil': [15, 18], 'Green': [149, 155, 156, 165, 170, 177, 181], 'Greenbac
```

```
grouped_by_party.get_group("Socialist")
```

	Year	Candidate	Party	Popular vote	Result	%
58	1904	Eugene V. Debs	Socialist	402810	loss	2.985897
62	1908	Eugene V. Debs	Socialist	420852	loss	2.850866
66	1912	Eugene V. Debs	Socialist	901551	loss	6.004354
71	1916	Allan L. Benson	Socialist	590524	loss	3.194193
76	1920	Eugene V. Debs	Socialist	913693	loss	3.428282
85	1928	Norman Thomas	Socialist	267478	loss	0.728623
88	1932	Norman Thomas	Socialist	884885	loss	2.236211
92	1936	Norman Thomas	Socialist	187910	loss	0.412876
95	1940	Norman Thomas	Socialist	116599	loss	0.234237
102	1948	Norman Thomas	Socialist	139569	loss	0.286312

4.3.2 Other `GroupBy` Methods

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.

Let's illustrate some examples by creating a `DataFrame` called `df`.

```
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

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*.

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

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

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

	num	state
letter		
A	2	1

	num	state
letter		
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](#).

4.3.3 Filtering by Group

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

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

- Takes a `DataFrame` object as input
- Returns a single `True` or `False`.

`groupby.filter` applies `func` to each group/sub-`DataFrame`:

- If `func` returns `True` for a group, then all rows belonging to the group are preserved.
- If `func` returns `False` for a group, then all rows belonging to that group are filtered out.

In other words, sub-`DataFrame`s 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 and the column we grouped on does **NOT** become the index!

To illustrate how this happens, let's go back to the `elections` dataset. Say we want to identify "tight" election years – that is, we want to find all rows that correspond to election 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, `func`, 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.

4.3.4 Aggregation with `lambda` Functions

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)
```

```
/var/folders/gr/vb80r2qs5td4rqbnv4dn2klh0000gn/T/ipykernel_19239/4278286395.py:1: FutureWarn

The provided callable <built-in function max> is currently using DataFrameGroupBy.max. In a
```

	Year	Candidate	Popular vote	Result	%
Party					
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

4.4 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 again. 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.

```
babynames.head()
```

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

```
# 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)
```

```
/var/folders/gr/vb80r2qs5td4rqbnv4dn2klh0000gn/T/ipykernel_19239/3186035650.py:3: FutureWarn

The provided callable <built-in function sum> is currently using DataFrameGroupBy.sum. In a
```

Year	Sex	Count
1910	F	5950
	M	3213
1911	F	6602
	M	3381
1912	F	9804
	M	8142

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.

Here's an illustration of the process:

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

```
# The `pivot_table` method is used to generate a Pandas pivot table
import numpy as np
babynames.pivot_table(
    index = "Year",
    columns = "Sex",
    values = "Count",
    aggfunc = "sum",
).head(5)
```

Sex	F	M
Year		
1910	5950	3213
1911	6602	3381
1912	9804	8142
1913	11860	10234
1914	13815	13111

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				
1910	295	237	Yvonne	William
1911	390	214	Zelma	Willis
1912	534	501	Yvonne	Woodrow
1913	584	614	Zelma	Yoshio
1914	773	769	Zelma	Yoshio
1915	998	1033	Zita	Yukio

Note that each row provides the number of girls and number of boys having that year's most common name, and also lists the alphabetically largest girl name and boy name. The counts for number of girls/boys in the resulting `DataFrame` do not correspond to the names listed. For example, in 1910, the most popular girl name is given to 295 girls, but that name was likely not Yvonne.

4.5 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 2022. 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 candidate'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 2022
babynames_2022 = babynames[babynames["Year"]==2022]
babynames_2022.head()
```

	State	Sex	Year	Name	Count	First Letter
237964	CA	F	2022	Leandra	10	L
404916	CA	M	2022	Leandro	99	L
405892	CA	M	2022	Andreas	14	A
235927	CA	F	2022	Andrea	322	A
405695	CA	M	2022	Deandre	18	D

Now, we're ready to join the two tables. `pd.merge` is the **pandas** method used to join `DataFrame`s together.

```
merged = pd.merge(left = elections, right = babynames_2022, \
                  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

# Second option
# merged = elections.merge(right = babynames_2022, \
    # left_on = "First Name", right_on = "Name")
```

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

Let's take a closer look at the parameters:

- `left` and `right` parameters are used to specify the `DataFrame`s 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 `DataFrame`s. We'll talk more about this idea of a pairing key next lecture.

4.6 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 chances 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

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns
#%matplotlib inline
plt.rcParams['figure.figsize'] = (12, 9)

sns.set()
sns.set_context('talk')
np.set_printoptions(threshold=20, precision=2, suppress=True)
pd.set_option('display.max_rows', 30)
pd.set_option('display.max_columns', None)
pd.set_option('display.precision', 2)
# This option stops scientific notation for pandas
pd.set_option('display.float_format', '{:.2f}'.format)

# Silence some spurious seaborn warnings
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

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 possible 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

We often prefer rectangular data for data analysis. Rectangular structures are easy to manipulate and analyze. A key element of data cleaning is about transforming data to be more rectangular.

There are two kinds of rectangular data: tables and matrices. Tables have named columns with different data types and are manipulated using data transformation languages. Matrices contain numeric data of the same type and are manipulated using linear algebra.

5.1.1 File Formats

There are many file types for storing structured data: TSV, JSON, XML, ASCII, SAS, etc. We'll only cover CSV, TSV, and JSON in lecture, but you'll likely encounter other formats as you work with different datasets. Reading documentation is your best bet for understanding how to process the multitude of different file types.

5.1.1.1 CSV

CSVs, which stand for **Comma-Separated Values**, are a common tabular data 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:

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

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.21
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.79
2	1828	Andrew Jackson	Democratic	642806	win	56.20
3	1828	John Quincy Adams	National Republican	500897	loss	43.80
4	1832	Andrew Jackson	Democratic	702735	win	54.57

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`. We'll use the `repr()` function to return the raw string with its special characters:

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

```
'Year,Candidate,Party,Popular vote,Result,%\n'
'1824,Andrew Jackson,Democratic-Republican,151271,loss,57.21012204\n'
'1824,John Quincy Adams,Democratic-Republican,113142,win,42.78987796\n'
'1828,Andrew Jackson,Democratic,642806,win,56.20392707\n'
```

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

5.1.1.2 TSV

Another common file type is **TSV (Tab-Separated Values)**. In a TSV, records are still delimited by a newline `\n`, while fields are delimited by `\t` tab character.

Let's check out the first few rows of the raw TSV file. Again, we'll use the `repr()` function so that `print` shows the special characters.

```python
with open("data/elections.txt", "r") as table:
    i = 0
    for row in table:
        print(repr(row))
        i += 1
        if i > 3:
            break
```

```
'\ufeffYear\tCandidate\tParty\tPopular vote\tResult\t%\n'
'1824\tAndrew Jackson\tDemocratic-Republican\t151271\tloss\t57.21012204\n'
'1824\tJohn Quincy Adams\tDemocratic-Republican\t113142\twin\t42.78987796\n'
'1828\tAndrew Jackson\tDemocratic\t642806\twin\t56.20392707\n'
```

TSVs can be loaded into `pandas` using `pd.read_csv`. We'll need to specify the **delimiter** with parameter`sep='\t'` (documentation).

```python
pd.read_csv("data/elections.txt", sep='\t').head(3)
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.21
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.79
2	1828	Andrew Jackson	Democratic	642806	win	56.20

An issue with CSVs and TSVs comes up whenever there are commas or tabs within the records. How does `pandas` differentiate between a comma delimiter vs. a comma within the field itself, for example `8,900`? To remedy this, check out the `quotechar` parameter.

5.1.1.3 JSON

JSON (JavaScript Object Notation) files behave similarly to Python dictionaries. A raw JSON is shown below.

```python
with open("data/elections.json", "r") as table:
    i = 0
    for row in table:
        print(row)
        i += 1
        if i > 8:
            break
```

```
[

 {

   "Year": 1824,

   "Candidate": "Andrew Jackson",

   "Party": "Democratic-Republican",

   "Popular vote": 151271,

   "Result": "loss",

   "%": 57.21012204

 },
```

JSON files can be loaded into `pandas` using `pd.read_json`.

```
pd.read_json('data/elections.json').head(3)
```

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.21
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.79
2	1828	Andrew Jackson	Democratic	642806	win	56.20

5.1.1.3.1 EDA with JSON: Berkeley COVID-19 Data

The City of Berkeley Open Data website has a dataset with COVID-19 Confirmed Cases among Berkeley residents by date. Let's download the file and save it as a JSON (note the source URL file type is also a JSON). In the interest of reproducible data science, we will download the data programatically. We have defined some helper functions in the `ds100_utils.py` file that we can reuse these helper functions in many different notebooks.

```
from ds100_utils import fetch_and_cache

covid_file = fetch_and_cache(
    "https://data.cityofberkeley.info/api/views/xn6j-b766/rows.json?accessType=DOWNLOAD",
    "confirmed-cases.json",
```

```
    force=False)
covid_file          # a file path wrapper object
```

```
Using cached version that was downloaded (UTC): Wed Aug 28 15:54:12 2024
```

```
PosixPath('data/confirmed-cases.json')
```

5.1.1.3.1.1 File Size

Let's start our analysis by getting a rough estimate of the size of the dataset to inform the tools we use to view the data. For relatively small datasets, we can use a text editor or spreadsheet. For larger datasets, more programmatic exploration or distributed computing tools may be more fitting. Here we will use `Python` tools to probe the file.

Since there seem to be text files, let's investigate the number of lines, which often corresponds to the number of records

```python
import os

print(covid_file, "is", os.path.getsize(covid_file) / 1e6, "MB")

with open(covid_file, "r") as f:
    print(covid_file, "is", sum(1 for l in f), "lines.")
```

```
data/confirmed-cases.json is 0.116367 MB
data/confirmed-cases.json is 1110 lines.
```

5.1.1.3.1.2 Unix Commands

As part of the EDA workflow, Unix commands can come in very handy. In fact, there's an entire book called "Data Science at the Command Line" that explores this idea in depth! In Jupyter/IPython, you can prefix lines with `!` to execute arbitrary Unix commands, and within those lines, you can refer to Python variables and expressions with the syntax `{expr}`.

Here, we use the `ls` command to list files, using the `-lh` flags, which request "long format with information in human-readable form." We also use the `wc` command for "word count," but with the `-l` flag, which asks for line counts instead of words.

These two give us the same information as the code above, albeit in a slightly different form:

```
!ls -lh {covid_file}
!wc -l {covid_file}
```

```
-rw-r--r--  1 xiaoruiliu  staff   114K Aug 28 15:54 data/confirmed-cases.json
    1109 data/confirmed-cases.json
```

5.1.1.3.1.3 File Contents

Let's explore the data format using `Python`.

```
with open(covid_file, "r") as f:
    for i, row in enumerate(f):
        print(repr(row)) # print raw strings
        if i >= 4: break
```

```
'{\n'
'  "meta" : {\n'
'    "view" : {\n'
'      "id" : "xn6j-b766",\n'
'      "name" : "COVID-19 Confirmed Cases",\n'
```

We can use the `head` Unix command (which is where `pandas`' `head` method comes from!) to see the first few lines of the file:

```
!head -5 {covid_file}
```

```
{
  "meta" : {
    "view" : {
      "id" : "xn6j-b766",
      "name" : "COVID-19 Confirmed Cases",
```

In order to load the JSON file into `pandas`, Let's first do some EDA with Oython's `json` package to understand the particular structure of this JSON file so that we can decide what (if anything) to load into `pandas`. Python has relatively good support for JSON data since it closely matches the internal python object model. In the following cell we import the entire JSON datafile into a python dictionary using the `json` package.

```python
import json

with open(covid_file, "rb") as f:
    covid_json = json.load(f)
```

The `covid_json` variable is now a dictionary encoding the data in the file:

```python
type(covid_json)
```

```
dict
```

We can examine what keys are in the top level JSON object by listing out the keys.

```python
covid_json.keys()
```

```
dict_keys(['meta', 'data'])
```

Observation: The JSON dictionary contains a `meta` key which likely refers to metadata (data about the data). Metadata is often maintained with the data and can be a good source of additional information.

We can investigate the metadata further by examining the keys associated with the metadata.

```python
covid_json['meta'].keys()
```

```
dict_keys(['view'])
```

The `meta` key contains another dictionary called `view`. This likely refers to metadata about a particular "view" of some underlying database. We will learn more about views when we study SQL later in the class.

```python
covid_json['meta']['view'].keys()
```

```
dict_keys(['id', 'name', 'assetType', 'attribution', 'averageRating', 'category', 'createdAt
```

Notice that this a nested/recursive data structure. As we dig deeper we reveal more and more keys and the corresponding data:

```
meta
|-> data
    | ... (haven't explored yet)
|-> view
    | -> id
    | -> name
    | -> attribution
    ...
    | -> description
    ...
    | -> columns
    ...
```

There is a key called description in the view sub dictionary. This likely contains a description of the data:

```python
print(covid_json['meta']['view']['description'])
```

```
Counts of confirmed COVID-19 cases among Berkeley residents by date.
```

5.1.1.3.1.4 Examining the Data Field for Records

We can look at a few entries in the `data` field. This is what we'll load into `pandas`.

```python
for i in range(3):
    print(f"{i:03} | {covid_json['data'][i]}")
```

```
000 | ['row-kzbg.v7my-c3y2', '00000000-0000-0000-0405-CB14DE51DAA7', 0, 1643733903, None, 164
001 | ['row-jkyx_9u4r-h2yw', '00000000-0000-0000-F806-86D0DBE0E17F', 0, 1643733903, None, 164
002 | ['row-qifg_4aug-y3ym', '00000000-0000-0000-2DCE-4D1872F9B216', 0, 1643733903, None, 164
```

Observations: * These look like equal-length records, so maybe `data` is a table! * But what do each of values in the record mean? Where can we find column headers?

For that, we'll need the `columns` key in the metadata dictionary. This returns a list:

```python
type(covid_json['meta']['view']['columns'])
```

```
list
```

5.1.1.3.1.5 Summary of exploring the JSON file

1. The above **metadata** tells us a lot about the columns in the data including column names, potential data anomalies, and a basic statistic.
2. Because of its non-tabular structure, JSON makes it easier (than CSV) to create **self-documenting data**, meaning that information about the data is stored in the same file as the data.
3. Self-documenting data can be helpful since it maintains its own description and these descriptions are more likely to be updated as data changes.

5.1.1.3.1.6 Loading COVID Data into `pandas`

Finally, let's load the data (not the metadata) into a `pandas DataFrame`. In the following block of code we:

1. Translate the JSON records into a `DataFrame`:

   - fields: `covid_json['meta']['view']['columns']`
   - records: `covid_json['data']`

2. Remove columns that have no metadata description. This would be a bad idea in general, but here we remove these columns since the above analysis suggests they are unlikely to contain useful information.

3. Examine the `tail` of the table.

```python
# Load the data from JSON and assign column titles
covid = pd.DataFrame(
    covid_json['data'],
    columns=[c['name'] for c in covid_json['meta']['view']['columns']])

covid.tail()
```

	sid	id	position	created_at	created_meta	updated_at	updated_meta	meta	Date	New Cases
699	row-49b6__x8zv.gyum	00000000-0000-0000-A18C-9174A6D05774	0	1643733903	None					
700	row-gs55-p5em.y4v9	00000000-0000-0000-F41D-5724AEABB4D6	0	1643733903	None					
701	row-3pyj.tf95-qu67	00000000-0000-0000-BEE3-B0188D2518BD	0	1643733903	None					
702	row-cgnd.8syv.jvjn	00000000-0000-0000-C318-63CF75F7F740	0	1643733903	None					
703	row-qywv__24x6-237y	00000000-0000-0000-FE92-9789FED3AA20	0	1643733903	None					

5.1.2 Primary and Foreign Keys

Last time, we introduced `.merge` as the `pandas` method for joining multiple `DataFrame`s 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 *uniquely* 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

5.1.3 Variable Types

Variables are columns. A variable is a measurement of a particular concept. Variables have two common properties: data type/storage type and variable type/feature type. The data type of a variable indicates how each variable value is stored in memory (integer, floating point, boolean, etc.) and affects which `pandas` functions are used. The variable type is a conceptualized measurement of information (and therefore indicates what values a variable can take on). Variable type is identified through expert knowledge, exploring the data itself, or consulting the data codebook. The variable type affects how one visualizes and inteprets the data. In this class, "variable types" are conceptual.

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 divide quantitative data further 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 the number of siblings they have.

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. Some examples include levels of education (high school, undergrad, grad, etc.), income bracket (low, medium, high), or Yelp rating.
- **Nominal qualitative variables**: categories with no specific order. For example, someone's political affiliation or Cal ID number.

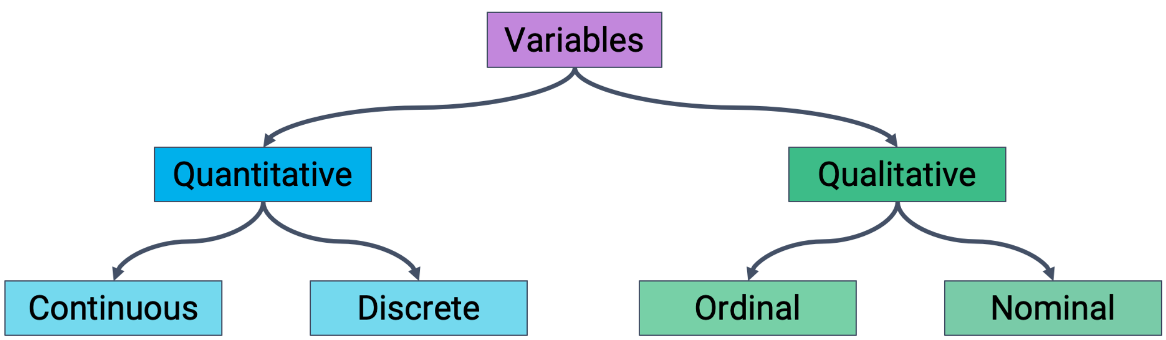


Figure 5.1: Classification of variable types

Note that many variables don't sit neatly in just one of these categories. Qualitative variables could have numeric levels, and conversely, quantitative variables could be stored as strings.

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.

5.2.1 Granularity

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.

5.2.2 Scope

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 a narrow scope might encompass all students enrolled in Data 100 whereas a dataset with an expansive scope might encompass all students in California.

5.2.3 Temporality

The **temporality** of a dataset describes the periodicity over which the data was collected as well as when the data was most recently collected or updated.

Time and date fields of a dataset could represent a few things:

1. when the "event" happened
2. when the data was collected, or when it was entered into the system
3. when the data was copied into the database

To fully understand the temporality of the data, it also may be necessary to standardize time zones or inspect recurring time-based trends in the data (do patterns recur in 24-hour periods? Over the course of a month? Seasonally?). The convention for standardizing time is the Coordinated Universal Time (UTC), an international time standard measured at 0 degrees latitude that stays consistent throughout the year (no daylight savings). We can represent Berkeley's time zone, Pacific Standard Time (PST), as UTC-7 (with daylight savings).

5.2.3.1 Temporality with `pandas`' `dt` accessors

Let's briefly look at how we can use `pandas`' `dt` accessors to work with dates/times in a dataset using the dataset you'll see in Lab 3: the Berkeley PD Calls for Service dataset.

```
calls = pd.read_csv("data/Berkeley_PD_-_Calls_for_Service.csv")
calls.head()
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	CVLEGEND	CVDOW	InDbDate	Block_Locatio
0	21014296	THEFT MISD. (UNDER $950)	04/01/2021 12:00:00 AM	10:58	LARCENY			
1	21014391	THEFT MISD. (UNDER $950)	04/01/2021 12:00:00 AM	10:38	LARCENY			
2	21090494	THEFT MISD. (UNDER $950)	04/19/2021 12:00:00 AM	12:15	LARCENY			
3	21090204	THEFT FELONY (OVER $950)	02/13/2021 12:00:00 AM	17:00	LARCENY			
4	21090179	BURGLARY AUTO	02/08/2021 12:00:00 AM	6:20	BURGLARY - VI			

Looks like there are three columns with dates/times: `EVENTDT`, `EVENTTM`, and `InDbDate`.

Most likely, `EVENTDT` stands for the date when the event took place, `EVENTTM` stands for the time of day the event took place (in 24-hr format), and `InDbDate` is the date this call is recorded onto the database.

If we check the data type of these columns, we will see they are stored as strings. We can convert them to `datetime` objects using pandas `to_datetime` function.

```
calls["EVENTDT"] = pd.to_datetime(calls["EVENTDT"])
calls.head()
```

```
/var/folders/gr/vb80r2qs5td4rqbnv4dn2klh0000gn/T/ipykernel_19298/874729699.py:1: UserWarning

Could not infer format, so each element will be parsed individually, falling back to `dateut
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	CVLEGEND	CVDOW	InDbDate	Block_Locatio
0	21014296	THEFT MISD. (UNDER $950)	2021-04-01	10:58	LARCENY	4		
1	21014391	THEFT MISD. (UNDER $950)	2021-04-01	10:38	LARCENY	4		
2	21090494	THEFT MISD. (UNDER $950)	2021-04-19	12:15	LARCENY	1		
3	21090204	THEFT FELONY (OVER $950)	2021-02-13	17:00	LARCENY	6		
4	21090179	BURGLARY AUTO	2021-02-08	6:20	BURGLARY - VEHICLE	1		

Now, we can use the `dt` accessor on this column.

We can get the month:

```
calls["EVENTDT"].dt.month.head()
```

```
0    4
1    4
2    4
3    2
4    2
Name: EVENTDT, dtype: int32
```

Which day of the week the date is on:

```
calls["EVENTDT"].dt.dayofweek.head()
```

```
0    3
1    3
2    0
3    5
4    0
Name: EVENTDT, dtype: int32
```

Check the mimimum values to see if there are any suspicious-looking, 70s dates:

```
calls.sort_values("EVENTDT").head()
```

	CASENO	OFFENSE	EVENTDT	EVENTTM	CVLEGEND	CVDOW	InDbDate	Block_Locatio
2513	20057398	BURGLARY COMMERCIAL	2020-12-17	16:05	BURGLARY - COMMERCI			
624	20057207	ASSAULT/BATTERY MISD.	2020-12-17	16:50	ASSAULT			
154	20092214	THEFT FROM AUTO	2020-12-17	18:30	LARCENY - FROM VEHIC			
659	20057324	THEFT MISD. (UNDER $950)	2020-12-17	15:44	LARCENY			
993	20057573	BURGLARY RESIDENTIAL	2020-12-17	22:15	BURGLARY - RESIDENTIA			

Doesn't look like it! We are good!

We can also do many things with the `dt` accessor like switching time zones and converting time back to UNIX/POSIX time. Check out the documentation on `.dt` accessor and time series/date functionality.

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
- Truncated data, e.g. Microsoft Excel would limit the number of rows to 655536 and the number of columns to 255

We often solve some of these more common issues in the following ways:

- Spelling errors: apply corrections or drop records that aren't in a dictionary
- Time zone inconsistencies: convert to a common time zone (e.g. UTC)
- Duplicated records or fields: identify and eliminate duplicates (using primary keys)
- Unspecified or inconsistent units: infer the units and check that values are in reasonable ranges in the data

5.3.1 Missing Values

Another 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 solution is to keep the data as `NaN` values.

A third 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 and replace with the predicted value from the model.
- 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 significant or meaningful.

5.4 EDA Demo 1: Tuberculosis in the United States

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

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

5.4.1 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()`

To try out options 1 and 2, you can view or download the Tuberculosis from the lecture demo notebook under the `data` folder in the left hand menu. Notice how the CSV file is a type of **rectangular data (i.e., tabular data) stored as comma-separated values**.

Next, let's try out option 3 using the `Python` file object. We'll look at the first four lines:

```python
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'
```

Finally, let's try option 4 and use 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: 6
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

You may notice some strange things about this table: what's up with the "Unnamed" column names and the first row?

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 that we can set to use the elements in row 1 as the appropriate columns:

```
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 2020
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

5.4.2 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 do you think this is?).

```
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 with the TB cases in 2019, 2020, and 2021? 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
```

Since there are commas in the values for TB cases, the numbers are read as the `object` datatype, or **storage type** (close to the `Python` string datatype), so `pandas` is concatenating strings instead of adding integers (recall that Python can "sum", or concatenate, strings together: `"data" + "100"` evaluates to `"data100"`).

Fortunately `read_csv` also has a `thousands` parameter (documentation):

```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 2020
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 2020
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

5.4.3 Gather Census Data

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 risk
    .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	2017	2018	2019
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 `python`'s `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 risk
    .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

5.4.4 Joining Data (Merging `DataFrames`)

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 2020
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

Having all of these columns is a little unwieldy. We could either drop the unneeded columns now, or just merge on smaller census `DataFrame`s. 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 2020
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

5.4.5 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["2019"]
tb_census_df.head(5)
```

	U.S. jurisdiction	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020
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 (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_census_
tb_census_df.head(5)
```

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

```python
tb_census_df.describe()
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB incidence 20
count	51.00	51.00	51.00	51.00	51.00	51.00
mean	174.51	140.65	154.12	2.10	1.78	1.97
std	341.74	271.06	286.78	1.50	1.34	1.48
min	1.00	0.00	2.00	0.17	0.00	0.21
25%	25.50	29.00	23.00	1.29	1.21	1.23
50%	70.00	67.00	69.00	1.80	1.52	1.70
75%	180.50	139.00	150.00	2.58	1.99	2.22
max	2111.00	1706.00	1750.00	7.91	7.92	7.92

5.4.6 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 pertain to some row at some point. This will make our cleaning slightly easier.

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

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

```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	2017	2018	2019
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
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 `DataFrame` have different `Geographic Area` fields, which was the key we merged on:

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

Geographic Area	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
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):

```python
# 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	2017	2018	2019
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).

```
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 incidence 20
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:

```
# 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_census_
tb_census_df.head(5)
```

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB incidence 20
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

	TB cases 2019	TB cases 2020	TB cases 2021	TB incidence 2019	TB incidence 2020	TB incidence 20
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
```

5.5 EDA Demo 2: Mauna Loa CO2 Data – A Lesson in Data Faithfulness

Mauna Loa Observatory has been monitoring CO2 concentrations since 1958.

```
co2_file = "data/co2_mm_mlo.txt"
```

Let's do some **EDA**!!

5.5.1 Reading this file into `Pandas`?

Let's instead check out this `.txt` file. Some questions to keep in mind: Do we trust this file extension? What structure is it?

Lines 71-78 (inclusive) are shown below:

```
line number |                            file contents

71          |   #            decimal     average   interpolated    trend    #days
72          |   #             date                             (season corr)
73          |   1958   3    1958.208      315.71      315.71      314.62     -1
74          |   1958   4    1958.292      317.45      317.45      315.29     -1
75          |   1958   5    1958.375      317.50      317.50      314.71     -1
76          |   1958   6    1958.458      -99.99      317.10      314.85     -1
77          |   1958   7    1958.542      315.86      315.86      314.98     -1
78          |   1958   8    1958.625      314.93      314.93      315.94     -1
```

Notice how:

- The values are separated by white space, possibly tabs.
- The data line up down the rows. For example, the month appears in 7th to 8th position of each line.
- The 71st and 72nd lines in the file contain column headings split over two lines.

We can use `read_csv` to read the data into a `pandas DataFrame`, and we provide several arguments to specify that the separators are white space, there is no header (**we will set our own column names**), and to skip the first 72 rows of the file.

```
co2 = pd.read_csv(
    co2_file, header = None, skiprows = 72,
    sep = r'\s+'       #delimiter for continuous whitespace (stay tuned for regex next lectu
)
co2.head()
```

	0	1	2	3	4	5	6
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1
3	1958	6	1958.46	-99.99	317.10	314.85	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1

Congratulations! You've wrangled the data!

…But our columns aren't named. **We need to do more EDA.**

5.5.2 Exploring Variable Feature Types

The NOAA webpage might have some useful tidbits (in this case it doesn't).

Using this information, we'll rerun `pd.read_csv`, but this time with some **custom column names.**

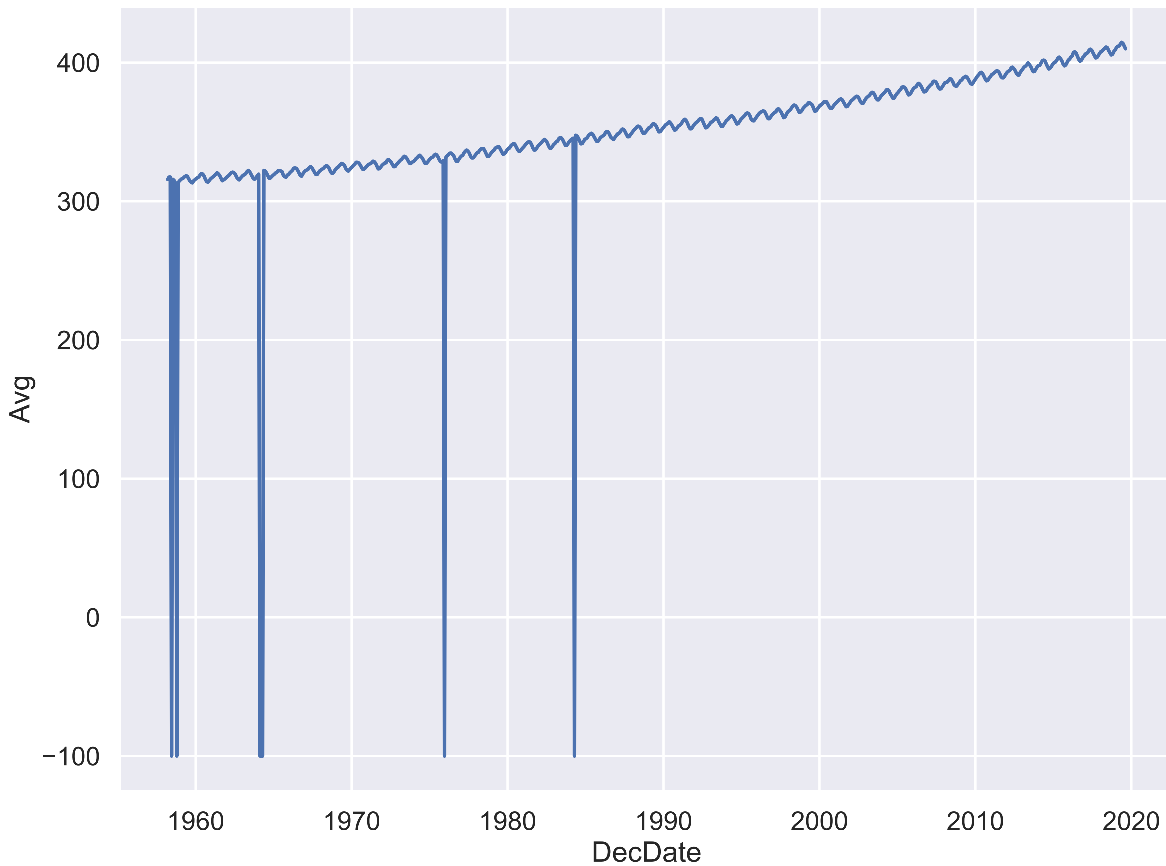
```
co2 = pd.read_csv(
    co2_file, header = None, skiprows = 72,
    sep = '\s+', #regex for continuous whitespace (next lecture)
    names = ['Yr', 'Mo', 'DecDate', 'Avg', 'Int', 'Trend', 'Days']
)
co2.head()
```

	Yr	Mo	DecDate	Avg	Int	Trend	Days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1
3	1958	6	1958.46	-99.99	317.10	314.85	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1

5.5.3 Visualizing CO2

Scientific studies tend to have very clean data, right…? Let's jump right in and make a time series plot of CO2 monthly averages.

```
sns.lineplot(x='DecDate', y='Avg', data=co2);
```



The code above uses the `seaborn` plotting library (abbreviated `sns`). We will cover this in the Visualization lecture, but now you don't need to worry about how it works!

Yikes! Plotting the data uncovered a problem. The sharp vertical lines suggest that we have some **missing values**. What happened here?

```
co2.head()
```

	Yr	Mo	DecDate	Avg	Int	Trend	Days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1
3	1958	6	1958.46	-99.99	317.10	314.85	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1

```
co2.tail()
```

	Yr	Mo	DecDate	Avg	Int	Trend	Days
733	2019	4	2019.29	413.32	413.32	410.49	26
734	2019	5	2019.38	414.66	414.66	411.20	28
735	2019	6	2019.46	413.92	413.92	411.58	27
736	2019	7	2019.54	411.77	411.77	411.43	23
737	2019	8	2019.62	409.95	409.95	411.84	29

Some data have unusual values like -1 and -99.99.

Let's check the description at the top of the file again.

- -1 signifies a missing value for the number of days `Days` the equipment was in operation that month.
- -99.99 denotes a missing monthly average `Avg`

How can we fix this? First, let's explore other aspects of our data. Understanding our data will help us decide what to do with the missing values.

5.5.4 Sanity Checks: Reasoning about the data

First, we consider the shape of the data. How many rows should we have?

- If chronological order, we should have one record per month.
- Data from March 1958 to August 2019.
- We should have $ 12 \times(2019-1957) - 2 - 4 = 738 $ records.

```
co2.shape
```

```
(738, 7)
```

Nice!! The number of rows (i.e. records) match our expectations.

Let's now check the quality of each feature.

5.5.5 Understanding Missing Value 1: `Days`

`Days` is a time field, so let's analyze other time fields to see if there is an explanation for missing values of days of operation.

Let's start with **months**, `Mo`.

Are we missing any records? The number of months should have 62 or 61 instances (March 1957-August 2019).

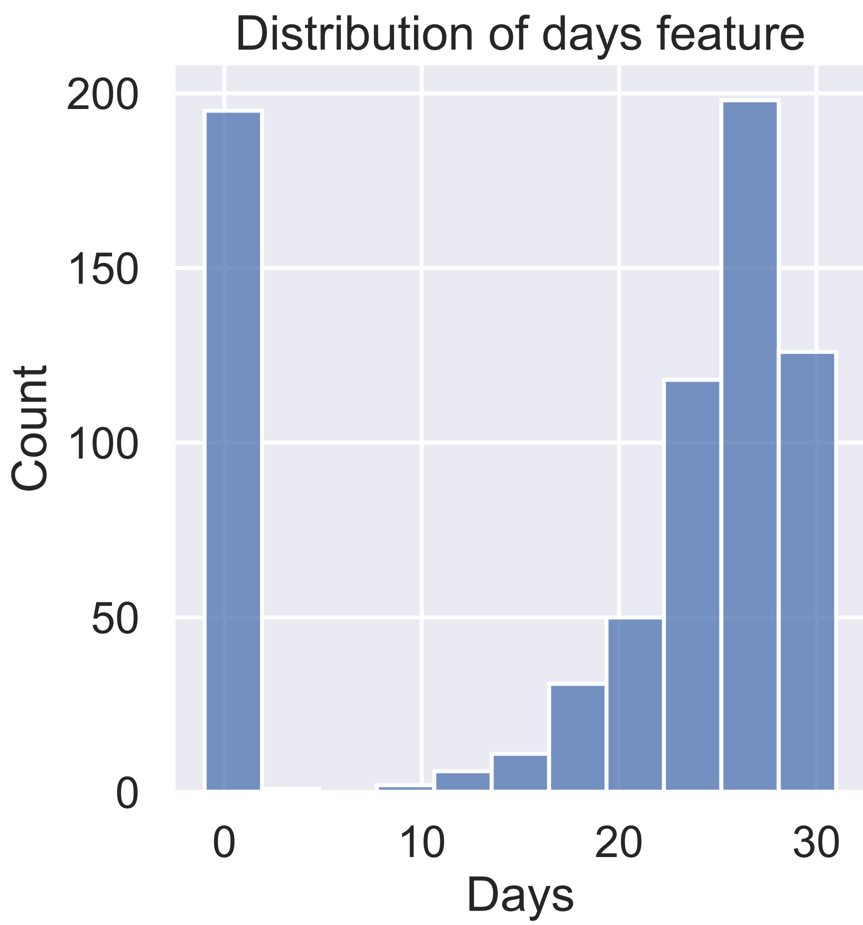
```
co2["Mo"].value_counts().sort_index()
```

```
Mo
1     61
2     61
3     62
4     62
5     62
6     62
7     62
8     62
9     61
10    61
11    61
12    61
Name: count, dtype: int64
```

As expected Jan, Feb, Sep, Oct, Nov, and Dec have 61 occurrences and the rest 62.

Next let's explore **days** `Days` itself, which is the number of days that the measurement equipment worked.

```
sns.displot(co2['Days']);
plt.title("Distribution of days feature"); # suppresses unneeded plotting output
```

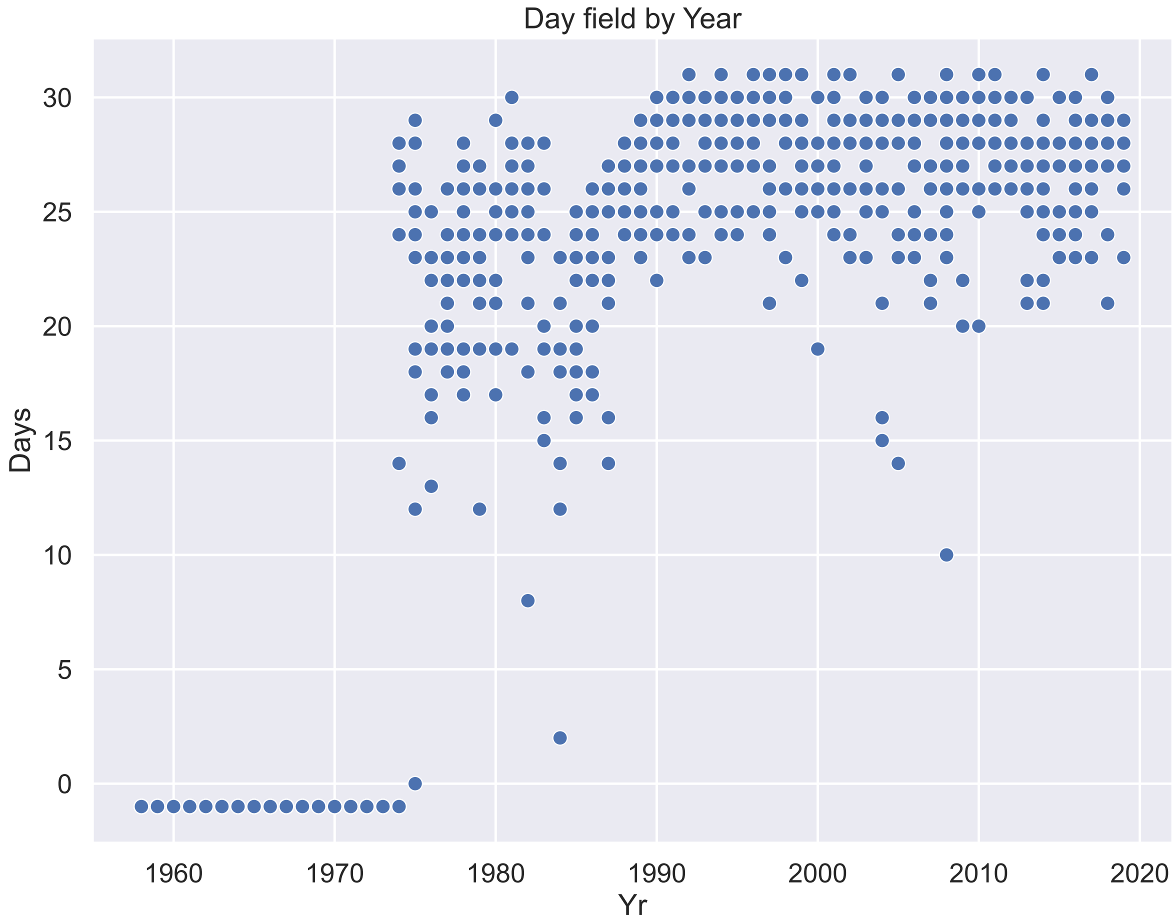


In terms of data quality, a handful of months have averages based on measurements taken on fewer than half the days. In addition, there are nearly 200 missing values–**that's about 27% of the data**!

Finally, let's check the last time feature, **year** `Yr`.

Let's check to see if there is any connection between missing-ness and the year of the recording.

```
sns.scatterplot(x="Yr", y="Days", data=co2);
plt.title("Day field by Year"); # the ; suppresses output
```



Observations:

- All of the missing data are in the early years of operation.
- It appears there may have been problems with equipment in the mid to late 80s.

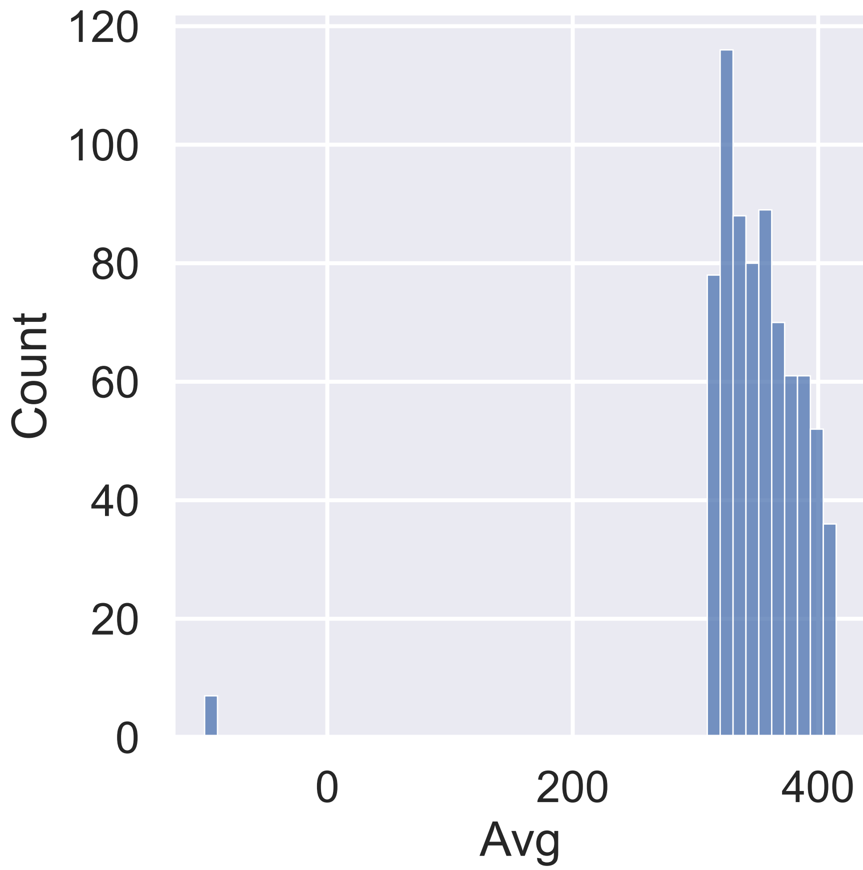
Potential Next Steps:

- Confirm these explanations through documentation about the historical readings.
- Maybe drop the earliest recordings? However, we would want to delay such action until after we have examined the time trends and assess whether there are any potential problems.

5.5.6 Understanding Missing Value 2: `Avg`

Next, let's return to the -99.99 values in `Avg` to analyze the overall quality of the CO2 measurements. We'll plot a histogram of the average CO2 measurements

```python
# Histograms of average CO2 measurements
sns.displot(co2['Avg']);
```



The non-missing values are in the 300-400 range (a regular range of CO2 levels).

We also see that there are only a few missing `Avg` values (**<1% of values**). Let's examine all of them:

```python
co2[co2["Avg"] < 0]
```

	Yr	Mo	DecDate	Avg	Int	Trend	Days
3	1958	6	1958.46	-99.99	317.10	314.85	-1
7	1958	10	1958.79	-99.99	312.66	315.61	-1
71	1964	2	1964.12	-99.99	320.07	319.61	-1
72	1964	3	1964.21	-99.99	320.73	319.55	-1

	Yr	Mo	DecDate	Avg	Int	Trend	Days
73	1964	4	1964.29	-99.99	321.77	319.48	-1
213	1975	12	1975.96	-99.99	330.59	331.60	0
313	1984	4	1984.29	-99.99	346.84	344.27	2

There doesn't seem to be a pattern to these values, other than that most records also were missing `Days` data.

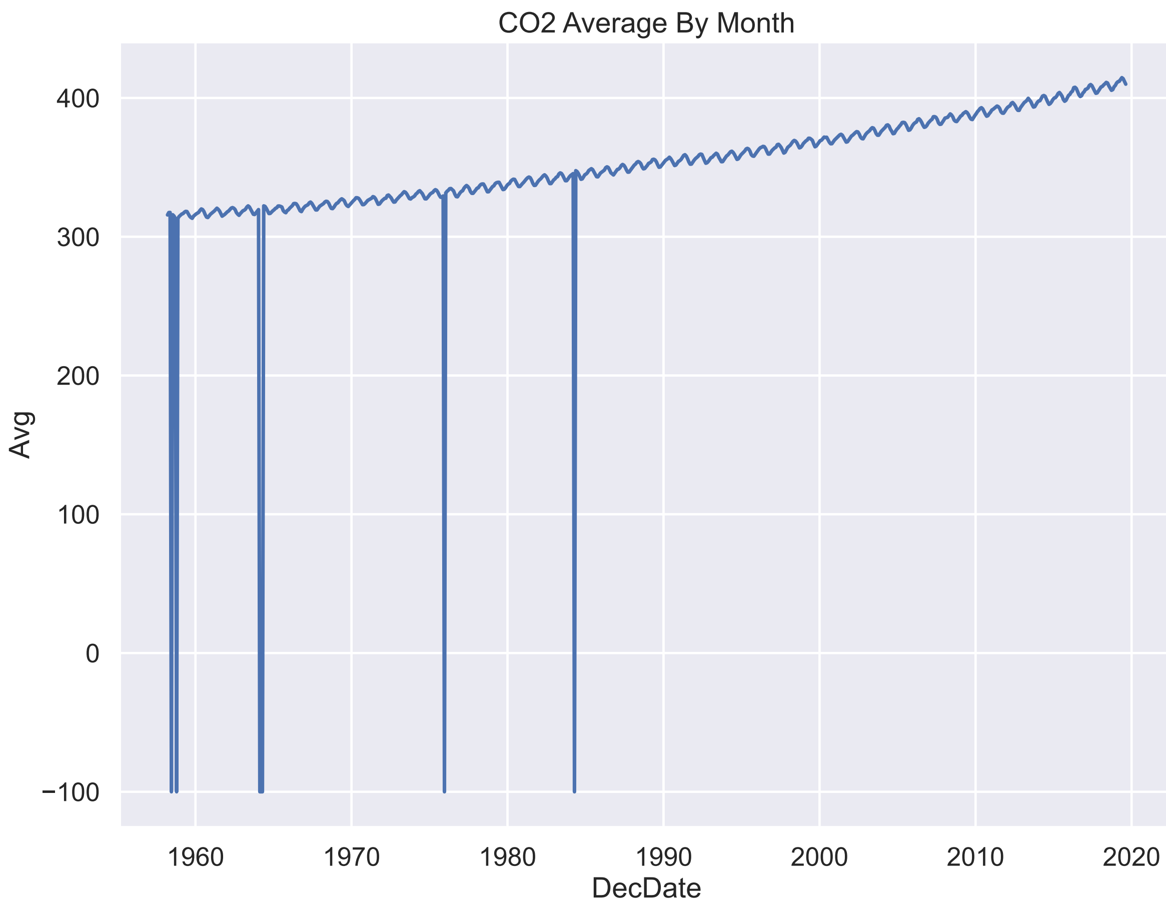
5.5.7 Drop, `NaN`, or Impute Missing `Avg` Data?

How should we address the invalid `Avg` data?

1. Drop records
2. Set to NaN
3. Impute using some strategy

Remember we want to fix the following plot:

```
sns.lineplot(x='DecDate', y='Avg', data=co2)
plt.title("CO2 Average By Month");
```



Since we are plotting `Avg` vs `DecDate`, we should just focus on dealing with missing values for `Avg`.

Let's consider a few options: 1. Drop those records 2. Replace -99.99 with NaN 3. Substitute it with a likely value for the average CO2?

What do you think are the pros and cons of each possible action?

Let's examine each of these three options.

```
# 1. Drop missing values
co2_drop = co2[co2['Avg'] > 0]
co2_drop.head()
```

	Yr	Mo	DecDate	Avg	Int	Trend	Days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1

	Yr	Mo	DecDate	Avg	Int	Trend	Days
2	1958	5	1958.38	317.50	317.50	314.71	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1
5	1958	8	1958.62	314.93	314.93	315.94	-1

```
# 2. Replace NaN with -99.99
co2_NA = co2.replace(-99.99, np.nan)
co2_NA.head()
```

	Yr	Mo	DecDate	Avg	Int	Trend	Days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1
3	1958	6	1958.46	NaN	317.10	314.85	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1

We'll also use a third version of the data.

First, we note that the dataset already comes with a **substitute value** for the -99.99.

From the file description:

> The `interpolated` column includes average values from the preceding column (`average`) and **interpolated values** where data are missing. Interpolated values are computed in two steps…

The `Int` feature has values that exactly match those in `Avg`, except when `Avg` is -99.99, and then a **reasonable** estimate is used instead.

So, the third version of our data will use the `Int` feature instead of `Avg`.

```
# 3. Use interpolated column which estimates missing Avg values
co2_impute = co2.copy()
co2_impute['Avg'] = co2['Int']
co2_impute.head()
```

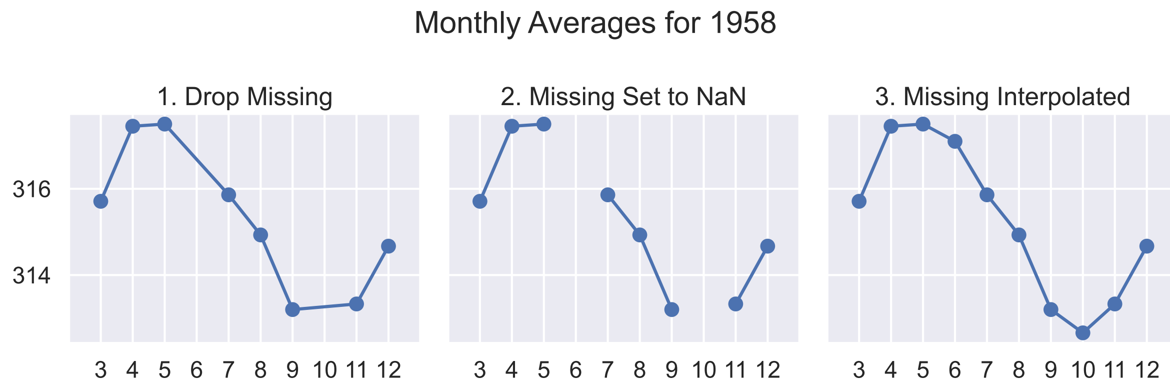
	Yr	Mo	DecDate	Avg	Int	Trend	Days
0	1958	3	1958.21	315.71	315.71	314.62	-1
1	1958	4	1958.29	317.45	317.45	315.29	-1
2	1958	5	1958.38	317.50	317.50	314.71	-1

	Yr	Mo	DecDate	Avg	Int	Trend	Days
3	1958	6	1958.46	317.10	317.10	314.85	-1
4	1958	7	1958.54	315.86	315.86	314.98	-1

What's a **reasonable** estimate?

To answer this question, let's zoom in on a short time period, say the measurements in 1958 (where we know we have two missing values).

```
# results of plotting data in 1958

def line_and_points(data, ax, title):
    # assumes single year, hence Mo
    ax.plot('Mo', 'Avg', data=data)
    ax.scatter('Mo', 'Avg', data=data)
    ax.set_xlim(2, 13)
    ax.set_title(title)
    ax.set_xticks(np.arange(3, 13))

def data_year(data, year):
    return data[data["Yr"] == 1958]

# uses matplotlib subplots
# you may see more next week; focus on output for now
fig, axes = plt.subplots(ncols = 3, figsize=(12, 4), sharey=True)

year = 1958
line_and_points(data_year(co2_drop, year), axes[0], title="1. Drop Missing")
line_and_points(data_year(co2_NA, year), axes[1], title="2. Missing Set to NaN")
line_and_points(data_year(co2_impute, year), axes[2], title="3. Missing Interpolated")

fig.suptitle(f"Monthly Averages for {year}")
plt.tight_layout()
```



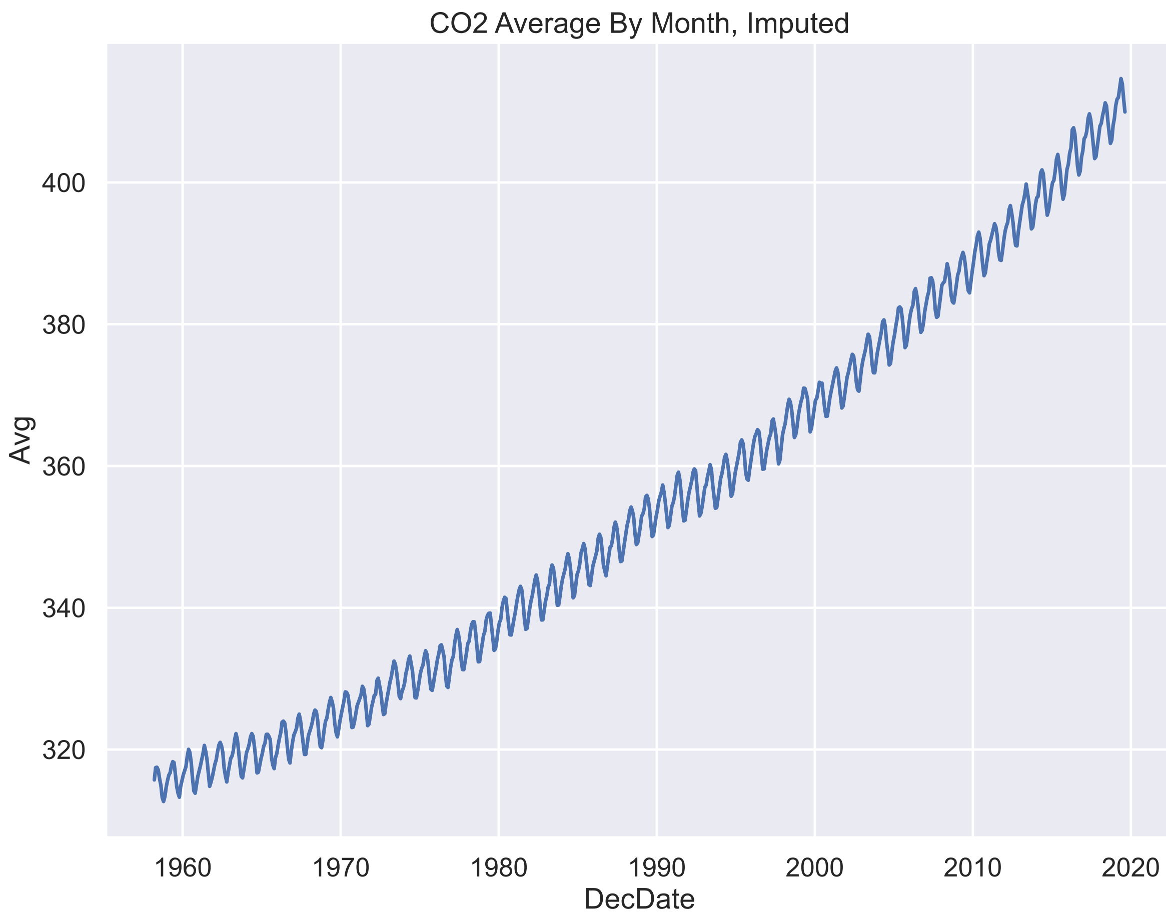
In the big picture since there are only 7 `Avg` values missing (**<1%** of 738 months), any of these approaches would work.

However there is some appeal to **option C, Imputing**:

- Shows seasonal trends for CO2
- We are plotting all months in our data as a line plot

Let's replot our original figure with option 3:

```
sns.lineplot(x='DecDate', y='Avg', data=co2_impute)
plt.title("CO2 Average By Month, Imputed");
```



Looks pretty close to what we see on the NOAA website!

5.5.8 Presenting the Data: A Discussion on Data Granularity

From the description:

- Monthly measurements are averages of average day measurements.
- The NOAA GML website has datasets for daily/hourly measurements too.

The data you present depends on your research question.

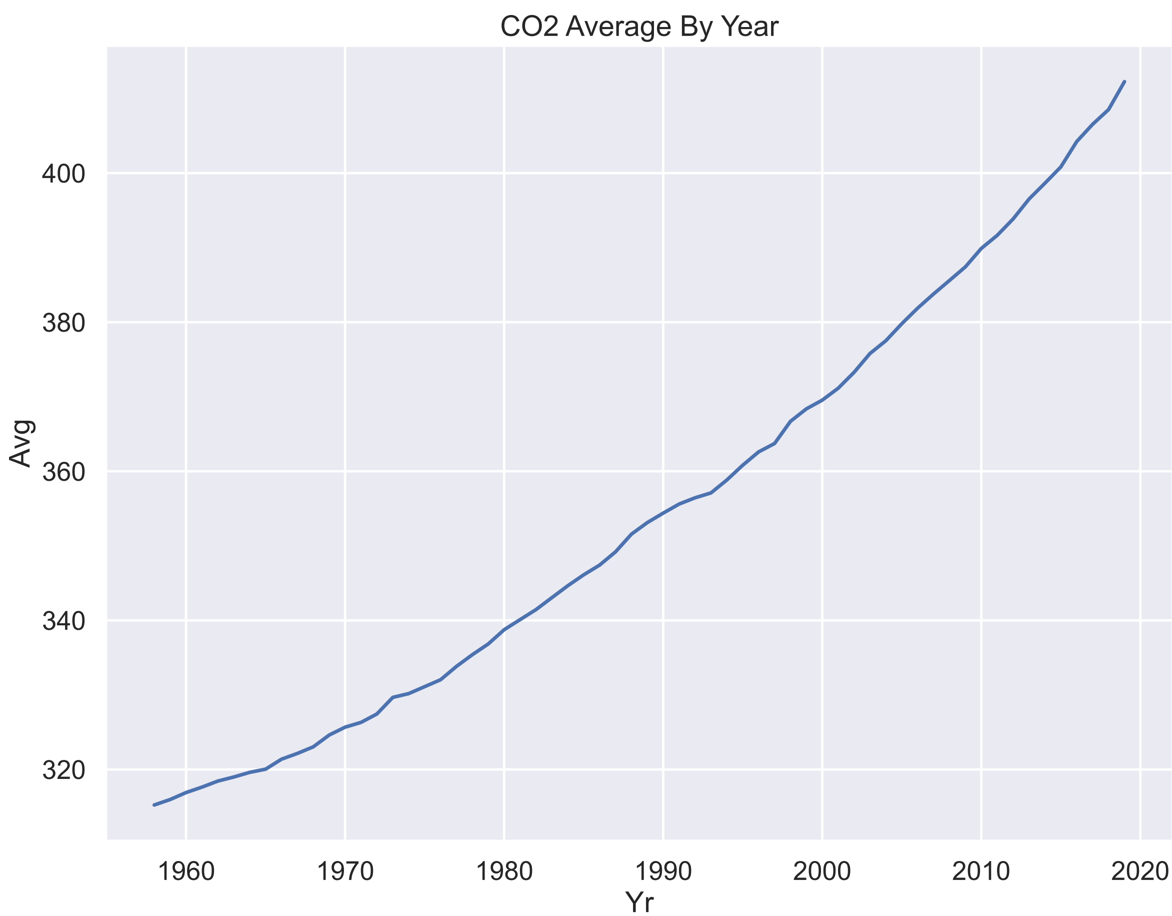
How do CO2 levels vary by season?

- You might want to keep average monthly data.

Are CO2 levels rising over the past 50+ years, consistent with global warming predictions?

- You might be happier with a **coarser granularity** of average year data!

```
co2_year = co2_impute.groupby('Yr').mean()
sns.lineplot(x='Yr', y='Avg', data=co2_year)
plt.title("CO2 Average By Year");
```



Indeed, we see a rise by nearly 100 ppm of CO2 since Mauna Loa began recording in 1958.

5.6 Summary

We went over a lot of content this lecture; let's summarize the most important points:

5.6.1 Dealing with Missing Values

There are a few options we can take to deal with missing data:

- Drop missing records
- Keep `NaN` missing values
- Impute using an interpolated column

5.6.2 EDA and Data Wrangling

There are several ways to approach EDA and Data Wrangling:

- Examine the **data and metadata**: what is the date, size, organization, and structure of the data?
- Examine each **field/attribute/dimension** individually.
- Examine pairs of related dimensions (e.g. breaking down grades by major).
- Along the way, we can:
  - **Visualize** or summarize the data.
  - **Validate assumptions** about data and its collection process. Pay particular attention to when the data was collected.
  - Identify and **address anomalies**.
  - Apply data transformations and corrections (we'll cover this in the upcoming lecture).
  - **Record everything you do!** Developing in Jupyter Notebook promotes *reproducibility* of your own work!

6 Regular Expressions

Learning Outcomes

- Understand Python string manipulation, `pandas` `Series` methods
- Parse and create regex, with a reference table
- Use vocabulary (closure, metacharacters, groups, etc.) to describe regex metacharacters

6.1 Why Work with Text?

Last lecture, we learned of the difference between quantitative and qualitative variable types. The latter includes string data — the primary focus of lecture 6. In this note, we'll discuss the necessary tools to manipulate text: Python string manipulation and regular expressions.

There are two main reasons for working with text.

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

6.2 Python String Methods

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

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

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

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

6.2.1 Canonicalization

Assume we want to merge the given tables.

```
import pandas as pd

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

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

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

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

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

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

6.2.1.1 Canonicalization with Python String Manipulation

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

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

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

```
'stjohnthebaptist'
```

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

```
county_and_pop['clean_county_python'] = county_and_pop['County'].map(canonicalize_county)
county_and_state['clean_county_python'] = county_and_state['County'].map(canonicalize_county)
display(county_and_state), display(county_and_pop);
```

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

	County	Population	clean_county_python
0	DeWitt	16798	dewitt
1	Lac Qui Parle	8067	lacquiparle
2	Lewis & Clark	55716	lewisandclark

	County	Population	clean_county_python
3	St. John the Baptist	43044	stjohnthebaptist

6.2.1.2 Canonicalization with Pandas Series Methods

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

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

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

county_and_pop['clean_county_pandas'] = canonicalize_county_series(county_and_pop['County'])
county_and_state['clean_county_pandas'] = canonicalize_county_series(county_and_state['County
display(county_and_pop), display(county_and_state);
```

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

	County	State	clean_county_python	clean_county_pandas
0	De Witt County	IL	dewitt	dewitt
1	Lac qui Parle County	MN	lacquiparle	lacquiparle
2	Lewis and Clark County	MT	lewisandclark	lewisandclark

	County	State	clean_county_python	clean_county_pandas
3	St John the Baptist Parish	LS	stjohnthebaptist	stjohnthebaptist

6.2.2 Extraction

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

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

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

log_lines
```

```
['169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585
 '193.205.203.3 - - [2/Feb/2005:17:23:6 -0800] "GET /stat141/Notes/dim.html HTTP/1.0" 404 30
 '169.237.46.240 - "" [3/Feb/2006:10:18:37 -0800] "GET /stat141/homework/Solutions/hw1Sol.pd
```

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

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

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

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

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

There are two problems with this code:

1. Python's built-in functions limit us to extract data one record at a time,

- This can be resolved using the `map` function or `pandas Series` methods.

2. The code is quite verbose.

    - This is a larger issue that is trickier to solve

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

6.3 RegEx Basics

A **regular expression ("RegEx")** is a sequence of characters that specifies a search pattern. They are written to extract specific information from text. Regular expressions are essentially part of a smaller programming language embedded in Python, made available through the `re` module. As such, they have a stand-alone syntax and methods for various capabilities.

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

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

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

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

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

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

6.3.1 Basics RegEx Syntax

There are four basic operations with regular expressions.

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

Notice how these metacharacter operations are ordered. Rather than being literal characters, these **metacharacters** manipulate adjacent characters. `()` takes precedence, followed by `*`, and finally `|`. This allows us to differentiate between very different regex commands like `AB*` and `(AB)*`. The former reads "`A` then zero or more copies of `B`", while the latter specifies "zero or more copies of `AB`".

6.3.1.1 Examples

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

Answer 1: `moo(oo)*n`

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

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

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

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

* A set of parenthesis must surround `|`. That way, each OR clause is everything to the left and right of `|` **within** the group. This ensures both the beginning `m` *and* trailing `n` are matched.

6.4 RegEx Expanded

Provided below are more complex regular expression functions.

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

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

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

6.4.0.1 Examples

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

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

Explanations

1. `.*SPB.*` only matches strings that contain the substring `SPB`.
    - The `.*` metacharacter matches any amount of non-negative characters. Newlines do not count.
2. This regular expression matches 3 of any digit, then a dash, then 2 of any digit, then a dash, then 4 of any digit.
    - You'll recognize this as the familiar Social Security Number regular expression.
3. Matches any email with a `com` or `edu` domain, where all characters of the email are letters.
    - At least one `.` must precede the domain name. Including a backslash `\` before any metacharacter (in this case, the `.`) tells RegEx to match that character exactly.

6.5 Convenient RegEx

Here are a few more convenient regular expressions.

Operation	Syntax Example	Matches	Doesn't Match
`built in character class`	`\w+ \d+ \s+`	Fawef_03 231123 `whitespace`	this person 423 people `non-whitespace`
`character class negation`: `[^]` (everything except the given characters)	[^a-z]+.	PEPPERS3982 17211!↑å	porch CLAmS
`escape character`: `\` (match the literal next character)	cow\.com	cow.com	cowscom

Operation	Syntax Example	Matches	Doesn't Match
`beginning of line`: `^`	^ark	ark two ark o ark	dark
`end of line`: `$`	ark$	dark ark o ark	ark two
`lazy version of zero or more` : `*?`	5.*?5	5005 55	5005005

6.5.1 Greediness

In order to fully understand the last operation in the table, we have to discuss greediness. RegEx is greedy – it will look for the longest possible match in a string. To motivate this with an example, consider the pattern `<div>.*</div>`. In the sentence below, we would hope that the bolded portions would be matched:

"This is a **<div>example</div>** of greediness **<div>in</div>** regular expressions."

However, in reality, RegEx captures far more of the sentence. The way RegEx processes the text given that pattern is as follows:

1. "Look for the exact string <÷>"
2. Then, "look for any character 0 or more times"
3. Then, "look for the exact string </div>"

The result would be all the characters starting from the leftmost <div> and the rightmost </div> (inclusive):

"This is a **<div>example</div> of greediness <div>in</div>** regular expressions."

We can fix this by making our pattern non-greedy, `<div>.*?</div>`. You can read up more in the documentation here.

6.5.2 Examples

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

```
log_lines[0]
```

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

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

Answer: `\[.*\]`

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

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

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

6.6 Regex in Python and Pandas (RegEx Groups)

6.6.1 Canonicalization

6.6.1.1 Canonicalization with RegEx

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

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

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

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

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

```
import re

text = "<div><td valign='top'>Moo</td></div>"
pattern = r"<[^>]+>"
re.sub(pattern, '', text)
```

```
'Moo'
```

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

In other words, don't forget to tag your RegEx with an `r`.

6.6.1.2 Canonicalization with `pandas`

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

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

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

```
html_data
```

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

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

```
0          Moo
1         Link
2    Bold text
Name: HTML, dtype: object
```

6.6.2 Extraction

6.6.2.1 Extraction with RegEx

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

Using the familiar regular expression for Social Security Numbers:

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

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

6.6.2.2 Extraction with `pandas`

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

Consider the following `DataFrame` `ssn_data`.

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

```
ssn_data
```

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

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

```
0                 [987-65-4321]
1                            []
2    [123-45-6789, 321-45-6789]
3                 [999-99-9999]
Name: SSN, dtype: object
```

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

As you may expect, there are similar **pandas** equivalents for other **re** functions as well. `Series.str.extract` takes in a pattern and returns a `DataFrame` of each capture group's first match in the string. In contrast, `Series.str.extractall` returns a multi-indexed `DataFrame` of all matches for each capture group. You can see the difference in the outputs below:

```python
pattern_cg = r"([0-9]{3})-([0-9]{2})-([0-9]{4})"
ssn_data["SSN"].str.extract(pattern_cg)
```

	0	1	2
0	987	65	4321
1	NaN	NaN	NaN
2	123	45	6789
3	999	99	9999

```python
ssn_data["SSN"].str.extractall(pattern_cg)
```

	match	0	1	2
0	0	987		
2	0	123		
	1	321		
3	0	999		

6.6.3 Regular Expression Capture Groups

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

Let's take a look at an example.

6.6.3.1 Example 1

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

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

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

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

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

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

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

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

6.6.3.2 Example 2

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

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

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

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

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