LECTURE 4

Pandas, Part II

More Advanced Pandas (Grouping, Aggregation, Pivot Tables Merging)

Data 100/Data 200, Spring 2022 @ UC Berkeley

Josh Hug and Lisa Yan

New Syntax / Concept Summary

Today we'll cover:

- Sorting with a custom key (by googling how to do this).
- Creating and dropping columns.
- Groupby: Output of .groupby("Name") is a DataFrameGroupBy object. Condense back into a DataFrame or Series with:
  - groupby.agg
  - groupby.size
  - groupby.filter
  - and more...
- Pivot tables: An alternate way to group by exactly two columns.
- Joining tables using pd.merge.

Googling Custom Sorts

Lecture 04, Data 100 Spring 2022

- **Googling Custom Sorts**
- Adding, Modifying, and Removing Columns
- Groupby.agg
- Some groupby.agg Puzzles
- One more groupby Puzzle
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- A Quick Look at Joining Tables

Manipulating String Data

Last time, we saw how we could find, for example, the most popular male names in California in the year 2020:

```
babynames.query('Sex == "M" and Year == 2020')
         .sort_values("Count", ascending = False)
```

	State	Sex	Year	Name	Count
391409	CA	M	2020	Noah	2608
391410	CA	M	2020	Liam	2406
391411	CA	M	2020	Mateo	2057
391412	CA	M	2020	Sebastian	1981
391413	CA	M	2020	Julian	1679
...	...	...	...	...	...
393938	CA	M	2020	Gavino	5
393937	CA	M	2020	Gaspar	5
393936	CA	M	2020	Gannon	5
393935	CA	M	2020	Galen	5
394178	CA	M	2020	Zymir	5

Manipulating String Data

What if we wanted to find the longest names in California?

- Just sorting by name won't work!

```
babynames.query('Sex == "M" and Year == 2020')
          .sort_values("Name", ascending = False)
```

	State	Sex	Year	Name	Count
393226	CA	M	2020	Zyon	9
394178	CA	M	2020	Zymir	5
393352	CA	M	2020	Zyan	8
392118	CA	M	2020	Zyaire	37
392838	CA	M	2020	Zyair	13
...	...	...	...	...	...
393106	CA	M	2020	Aamir	9
393822	CA	M	2020	Aalam	5
393354	CA	M	2020	Aaditya	7
393353	CA	M	2020	Aadi	7
392994	CA	M	2020	Aaden	10

Before summer 2020, this would not have been straightforward.

- But these days it is! Let's figure it out by Googling.

Manipulating String Data

What if we wanted to find the longest names in California?

```
babynames.query('Sex == "M" and Year == 2020')
          .sort_values("Name", key = lambda x: x.str.len(),
                       ascending = False)
```

	State	Sex	Year	Name	Count
393478	CA	M	2020	Michaelangelo	7
393079	CA	M	2020	Michelangelo	10
392964	CA	M	2020	Maximilliano	11
394047	CA	M	2020	Maxemiliano	5
392610	CA	M	2020	Maximillian	16
...	...	...	...	...	...
393110	CA	M	2020	Aj	9
392856	CA	M	2020	Cy	12
393558	CA	M	2020	An	6
393981	CA	M	2020	Jj	5
392750	CA	M	2020	Om	14

Adding, Modifying, and Removing Columns

Lecture 04, Data 100 Spring 2022

Sorting By Length

As motivation, let's try to solve the sorting problem using a pre-2020 technique:

- We will create a temporary column, then sort on it.

Approach 1: Create a Temporary Column

Intuition: Create a column equal to the length. Sort by that column.

	State	Sex	Year	Name	Count	name_lengths
312731	CA	M	1993	Ryanchristopher	5	15
322558	CA	M	1997	Franciscojavier	5	15
297806	CA	M	1987	Franciscojavier	5	15
307174	CA	M	1991	Franciscojavier	6	15
302145	CA	M	1989	Franciscojavier	6	15

Syntax for Column Addition

Adding a column is easy:

```
#create a new series of only the lengths
babyname_lengths = babynames["Name"].str.len()

#add that series to the dataframe as a column
babynames["name_lengths"] = babyname_lengths
```

Can also do both steps on one line of code

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

Syntax for Column Addition

Sorting a table is as usual:

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

	State	Sex	Year	Name	Count	name_lengths
312731	CA	M	1993	Ryanchristopher	5	15
322558	CA	M	1997	Franciscojavier	5	15
297806	CA	M	1987	Franciscojavier	5	15
307174	CA	M	1991	Franciscojavier	6	15
302145	CA	M	1989	Franciscojavier	6	15

Syntax for Dropping a Column (or Row)

After sorting, we can drop the temporary column.

- The Drop method assumes you're dropping a row by default. Use axis = 'columns' to drop a column instead.

```
babynames = babynames.drop("name_lengths", axis = 'columns')
```

	State	Sex	Year	Name	Count	name_lengths
312731	CA	M	1993	Ryanchristopher	5	15
322558	CA	M	1997	Franciscojavier	5	15
297806	CA	M	1987	Franciscojavier	5	15
307174	CA	M	1991	Franciscojavier	6	15
302145	CA	M	1989	Franciscojavier	6	15

→

	State	Sex	Year	Name	Count
312731	CA	M	1993	Ryanchristopher	5
322558	CA	M	1997	Franciscojavier	5
297806	CA	M	1987	Franciscojavier	5
307174	CA	M	1991	Franciscojavier	6
302145	CA	M	1989	Franciscojavier	6

Sorting by Arbitrary Functions

Suppose we want to sort by the number of occurrences of “dr” + number of occurrences of “ea”.

- Use the Series `.map` method.

```
def dr_ea_count(string):
    return string.count('dr') + string.count('ea')

babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count)
babynames = babynames.sort_values(by = "dr_ea_count", ascending=False)
```

	State	Sex	Year	Name	Count	dr_ea_count
108712	CA	F	1988	Deandrea	5	3
293396	CA	M	1985	Deandrea	6	3
101958	CA	F	1986	Deandrea	6	3
115935	CA	F	1990	Deandrea	5	3
131003	CA	F	1994	Leandrea	5	3

Groupby.agg

Lecture 04, Data 100 Spring 2022

Goal

Goal: Find the female baby name whose popularity has fallen the most.

Number of Jennifers Born in California Per Year

Goal

Goal: Find the female baby name whose popularity has fallen the most.

Let's start by defining what we mean by changed popularity.

- In lecture, let's define the "ratio to peak" or RTP as the ratio of Jennifers born today to the maximum number born in a single year.

Example for "Jennifer":

- In 1972, we hit peak Jennifer. 6,064 Jennifers were born.
- In 2020, there were only 141 Jennifers.
- RTP is 141 / 6064 = 0.0233.

Let's spend some time in our notebook. The following N slides are for reference only and will be skipped during live lecture.

Calculating RTP

```
max_jennifers = max(babynames.query("Name == 'Jennifer' and Sex == 'F'")["Count"])
```

```
6064
```

```
current_jennifers = babynames.query("Name == 'Jennifer' and Sex == 'F'")["Count"].iloc[-1]
```

```
141
```

```
rtp = current_jennifers / max_jennifers
```

```
0.023251978891820582
```

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

```
jennifer_counts_series = babynames.query("Name == 'Jennifer' and Sex == 'F'")["Count"]
ratio_to_peak(jennifer_counts_series)
0.023251978891820582
```

Approach 1: Getting RTP for Every Name The Hard Way

Approach 1: Hack something together using our existing Python knowledge.

```
#build dictionary where each entry is the rtp for a given name
#e.g. rtps["jennifer"] should be 0.0231
rtps = {}
for name in ??:
    counts_of_current_name = female_babynames[??]["Count"]
    rtps[name] = ratio_to_peak(counts_of_current_name)

#convert to series
rtps = pd.Series(rtps)
```

Challenge: Try to fill in the code above by filling in the **??**.

Approach 1: Getting RTP for Every Name The Hard Way

Approach 1: Hack something together using our existing Python knowledge.

```
#build dictionary where each entry is the rtp for a given name
#e.g. rtps["jennifer"] should be 0.0231
rtps = {}
for name in babynames["Name"].unique():
    counts_of_current_name = female_babynames[female_babynames["Name"] == name]["Count"]
    rtps[name] = ratio_to_peak(counts_of_current_name)

#convert to series
rtps = pd.Series(rtps)
```

The code above is extremely slow, and also way more complicated than the better approach coming next.

Approach 2: Using Groupby and Agg

The code below is the more idiomatic way of computing what we want.

- Much simpler, much faster, much more versatile.

```
female_babynames.groupby("Name").agg(ratio_to_peak)
```

Name	Year	Count
Aadhira	1.0	0.600000
Aadhya	1.0	0.720000
Aadya	1.0	0.862069
Aahana	1.0	0.384615
Aahna	1.0	1.000000
...	...	...

Nothing new here! This is just the group function that you saw in data8, where collect = ratio_to_peak.

```
female_babynames.group("Name", ratio_to_peak)
```

Comparing the Two Approaches

As a reminder you should almost never be writing code in this class that includes loops or list comprehensions on Pandas series.

- Use the pandas API as intended!

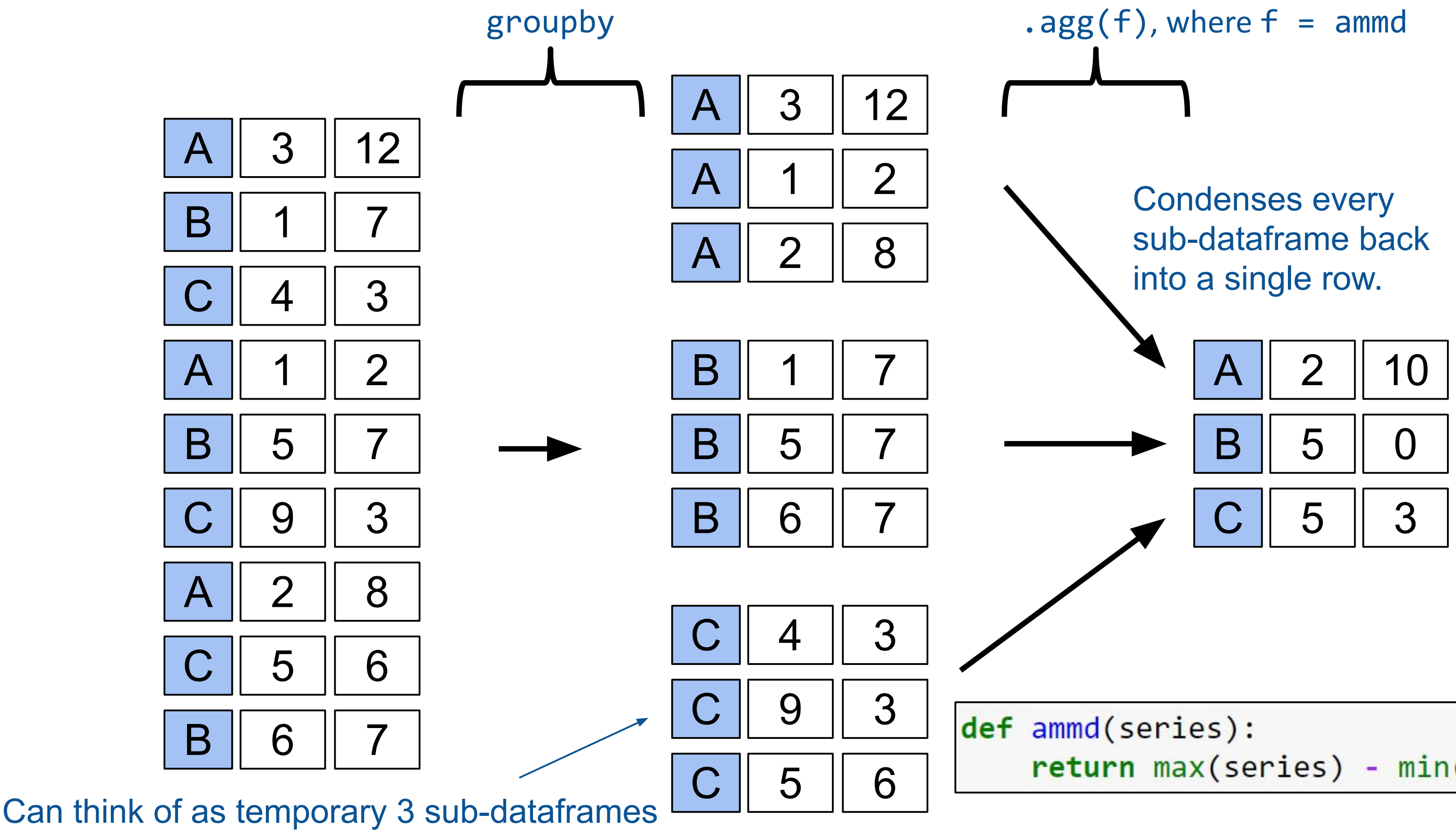
Approach 1: [BAD!!]

```
rtps = {}
for name in babynames["Name"].unique():
    counts_of_current_name = female_babynames[female_babynames["Name"] == name]["Count"]
    rtps[name] = ratio_to_peak(counts_of_current_name)
rtps = pd.Series(rtps)
```

Approach 2:

```
female_babynames.groupby("Name").agg(ratio_to_peak)
```

Visual Review of Data 8: Grouping and Collection




```python
def ammd(series):
    return max(series) - min(series)
```

Attendance Question: Check Your groupBy Understanding

Approach 2 generated two columns, Year and Count. www.yellkey.com/perhaps

In the five rows shown, note the Year is 1.0 for every value.

Are there any rows for which Year is **not** 1.0?

A. Yes, names that appeared for the first time in 2020.
B. Yes, names that did not appear in 2020.
C. Yes, names whose peak Count was in 2020.
D. No, every row has a Year value of 1.0.

```
fbn.groupby("Name").agg(ratio_to_peak)
```

Name	Year	Count
Aadhira	1.0	0.600000
Aadhya	1.0	0.720000
Aadya	1.0	0.862069
Aahana	1.0	0.384615
Aahna	1.0	1.000000
...	...	...

Attendance Question: Check Your groupBy Understanding

Approach 2 generated two columns, Year and Count. www.yellkey.com/perhaps

In the five rows shown, note the Year is 1.0 for every value.

Are there any rows for which Year is **not** 1.0?

A. Yes, names that appeared for the first time in 2020.
B. Yes, names that did not appear in 2020.
C. Yes, names whose peak Count was in 2020.
D. No, every row has a Year value of 1.0.

Note: This is a hard question! I originally tricked myself and thought the answer was B.

```
fbn.groupby("Name").agg(ratio_to_peak)
```

Name	Year	Count
Aadhira	1.0	0.600000
Aadhya	1.0	0.720000
Aadya	1.0	0.862069
Aahana	1.0	0.384615
Aahna	1.0	1.000000
...	...	...

Note on Nuisance Columns

At least as of the time of this slide creation (January 2022), executing our agg call results in:

```
female_babynames.groupby("Name").agg(ratio_to_peak)
```

```
/opt/conda/lib/python3.9/site-packages/pandas/core/groupby/generic.py:303: FutureWarning:

Dropping invalid columns in SeriesGroupBy.agg is deprecated. In a future version, a TypeError will be raised. Before calling .agg, select only columns which should be valid for the aggregating function.
```

	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
...	...	...	...	...	...
232097	CA	F	2020	Ziana	5
232098	CA	F	2020	Zoha	5
232099	CA	F	2020	Zuleika	5
232100	CA	F	2020	Zuriel	5
232101	CA	F	2020	Zyrah	5

→

	Year	Count
Name		
Aadhira	1.0	0.600000
Aadhya	1.0	0.720000
Aadya	1.0	0.862069
Aahana	1.0	0.384615
Aahna	1.0	1.000000
...	...	...

For more details, see Pandas 1.3 release notes.

Note on Nuisance Columns

At least as of the time of this slide creation (January 2022), executing our agg call results in:

```
female_babynames.groupby("Name").agg(ratio_to_peak)
```

```
/opt/conda/lib/python3.9/site-packages/pandas/core/groupby/generic.py:303: FutureWarning:

Dropping invalid columns in SeriesGroupBy.agg is deprecated. In a future version, a TypeError will be
raised. Before calling .agg, select only columns which should be valid for the aggregating function.
```

At some point in the future (maybe when you try running the notebook sometime in late 2022 or later), this code will simply crash!

- Presumably, the designers of pandas felt like automatically dropping nuisance columns leads to bad coding practices.
- And in line with the Zen of Python: “Explicit is better than implicit.”

Note on Nuisance Columns

Below, we explicitly select the columns **BEFORE** calling `agg` to avoid the warning.

```
rtp_table = female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
```

	Count
Name	
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

Renaming Columns

The code below renames the Count column to “Count RTP”.

```
rtp_table = female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"})
```

Name	Count
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

→

Name	Count RTP
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

This syntax will be covered in lab 2 in slightly more detail.

Some Data Science Payoff

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

```
rtp_table.sort_values("Count RTP")
```

	Count RTP
Name	
Debra	0.001260
Debbie	0.002817
Susan	0.004320
Kim	0.004405
Tammy	0.004551
...	...

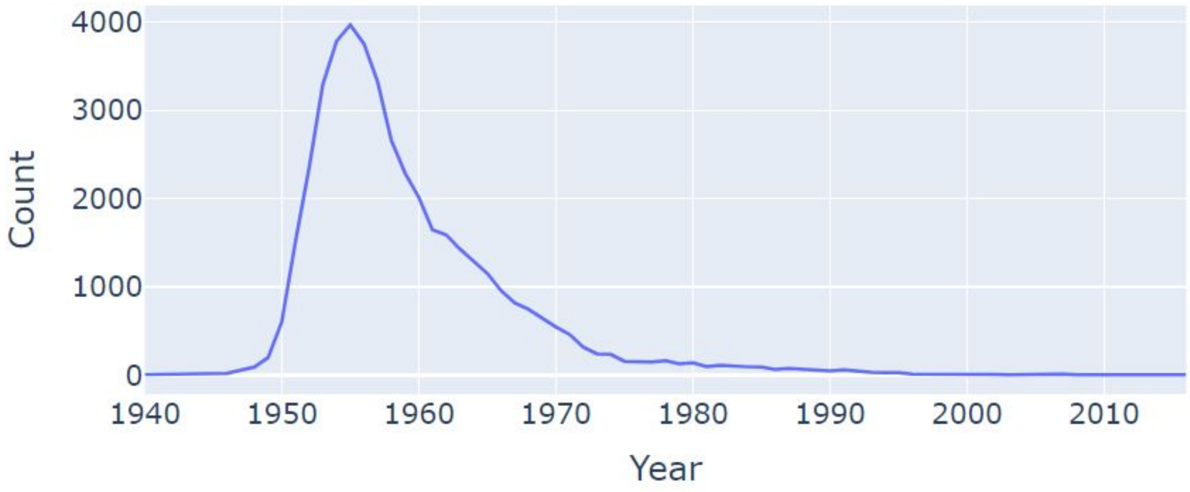
Some Data Science Payoff

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

```
rtp_table.sort_values("Count RTP")
```

	Count RTP
Name	
Debra	0.001260
Debbie	0.002817
Susan	0.004320
Kim	0.004405
Tammy	0.004551
...	...

```
px.line(babynames.query("Name == 'Debra' and Sex == 'F'"),
                                      x = "Year", y = "Count")
```



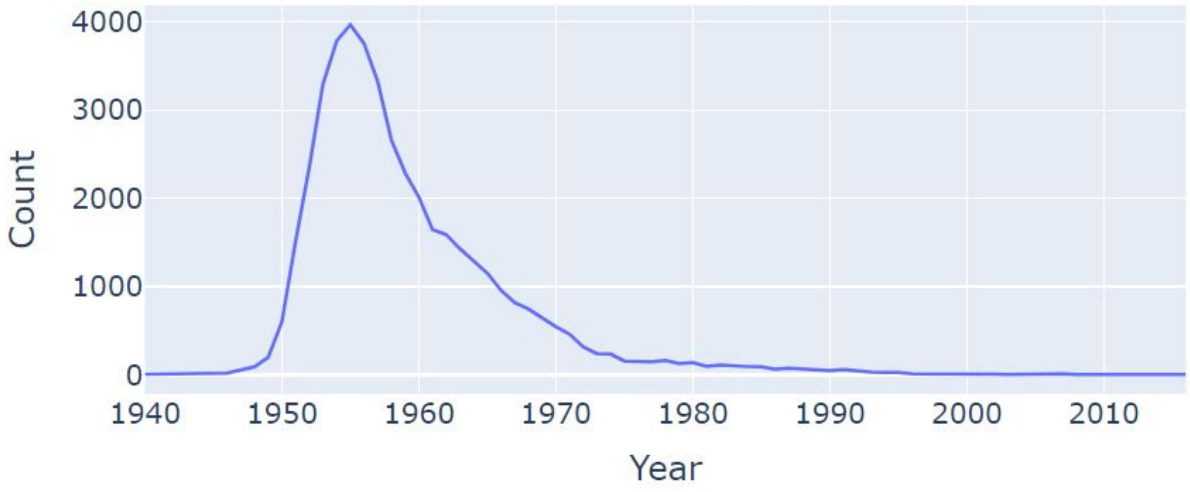
Some Data Science Payoff

With some fancier code we can plot the ten female names with the lowest Count RTP.

```
rtp_table.sort_values("Count RTP")
```

Name	Count RTP
Debra	0.001260
Debbie	0.002817
Susan	0.004320
Kim	0.004405
Tammy	0.004551
...	...

```
px.line(babynames.query("Name == 'Debra' and Sex == 'F'"),
                                          x = "Year", y = "Count")
```

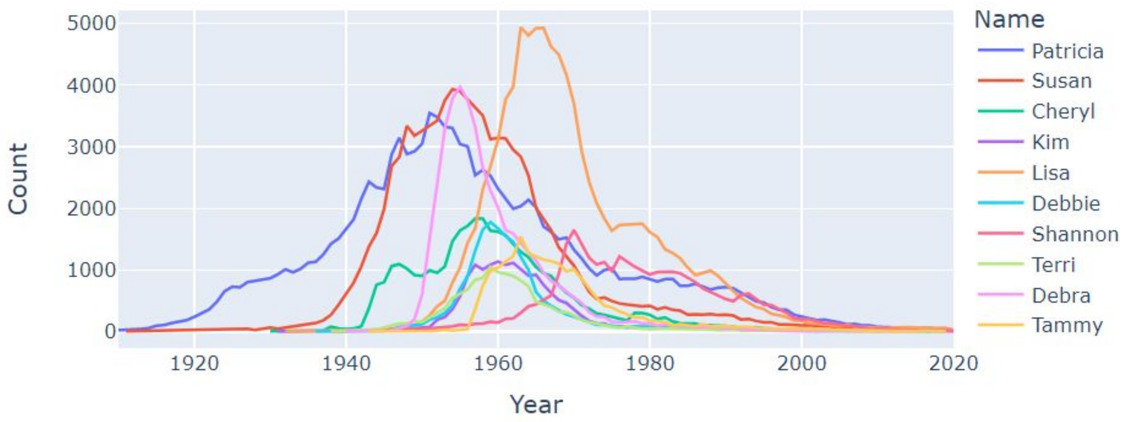


We can get the list of the top 10 names and then plot popularity with::

```
top10 = rtp_table.sort_values("Count RTP").head(10).index
```

```
Index(['Debra', 'Debbie', 'Susan', 'Kim', 'Tammy', 'Terri', 'Shannon',
       'Cheryl', 'Lisa', 'Patricia'],
      dtype='object', name='Name')
```

```
px.line(babynames.query("Name in @top10 and Sex == 'F'"),
                          x = "Year", y = "Count", color = "Name")
```



Some groupby.agg Puzzles

Lecture 04, Data 100 Spring 2022

- Googling Custom Sorts
- Adding, Modifying, and Removing Columns
- Groupby.agg
- **Some groupby.agg Puzzles**
- One more groupby Puzzle
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- A Quick Look at Joining Tables

Groupby Puzzle #1

Before we saw that the code below generates the Count RTP for all female names.

```
female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
```

	Count
Name	
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

Groupby Puzzle #1

Before we saw that the code below generates the Count RTP for all female names.

```
female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
```

Name	Count
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

Write a **`groupby.agg`** call that returns the total number of babies with each name.

Name	Count
Aadhira	22
Aadhya	368
Aadya	230
Aahana	129
Aahna	7
...	...

Groupby Puzzle #2

Before we saw that the code below generates the Count RTP for all female names.

```
female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
```

Name	Count
Aadhira	0.600000
Aadhya	0.720000
Aadya	0.862069
Aahana	0.384615
Aahna	1.000000
...	...

Write a **`groupby.agg`** call that returns the total number of babies with each name.

Name	Count
Aadhira	22
Aadhya	368
Aadya	230
Aahana	129
Aahna	7
...	...

```
female_babynames.groupby("Name")[["Count"]].agg(sum)
```

Groupby Puzzle #2

Before we saw that the code below generates the total number of babies with each name.

```
female_babynames.groupby("Name")[["Count"]].agg(sum)
```

	Count
Name	
Aadhira	22
Aadhya	368
Aadya	230
Aahana	129
Aahna	7
...	...

Write a `groupby.agg` call that returns the total babies born in every year:

	Count
Year	
1910	5950
1911	6602
1912	9804
1913	11860
1914	13815
...	...

Groupby Puzzle #2

Before we saw that the code below generates the total number of babies with each name.

```
female_babynames.groupby("Name")[["Count"]].agg(sum)
```

	Count
Name	
Aadhira	22
Aadhya	368
Aadya	230
Aahana	129
Aahna	7
...	...

Write a **`groupby.agg`** call that returns the total babies born in every year:

	Count
Year	
1910	5950
1911	6602
1912	9804
1913	11860
1914	13815
...	...

```
female_babynames.groupby("Year")[["Count"]].agg(sum)
```

Shorthand groupby Methods

Pandas also provides a number of shorthand functions that you can use in place of `agg`.

```
female_babynames.groupby("Name")[["Count"]].agg(sum)
```

	Count
Year	
1910	5950
1911	6602
1912	9804
1913	11860
1914	13815
...	...

Instead, we could have simply written:

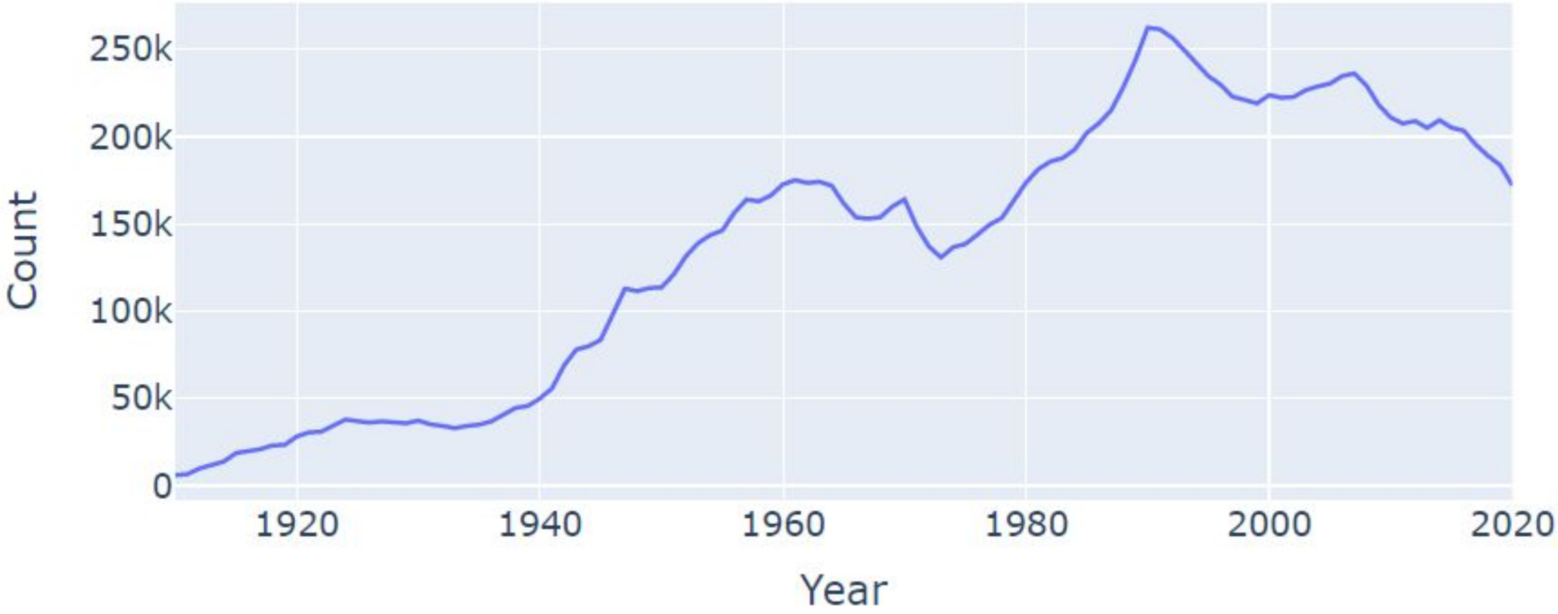
```
female_babynames.groupby("Name")[["Count"]].sum()
```

For more examples (first, last, mean, median, etc.) see the left sidebar on:
https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.GroupBy.sum.html

Plotting Birth Counts

Plotting the DataFrame we just generated tells an interesting story.

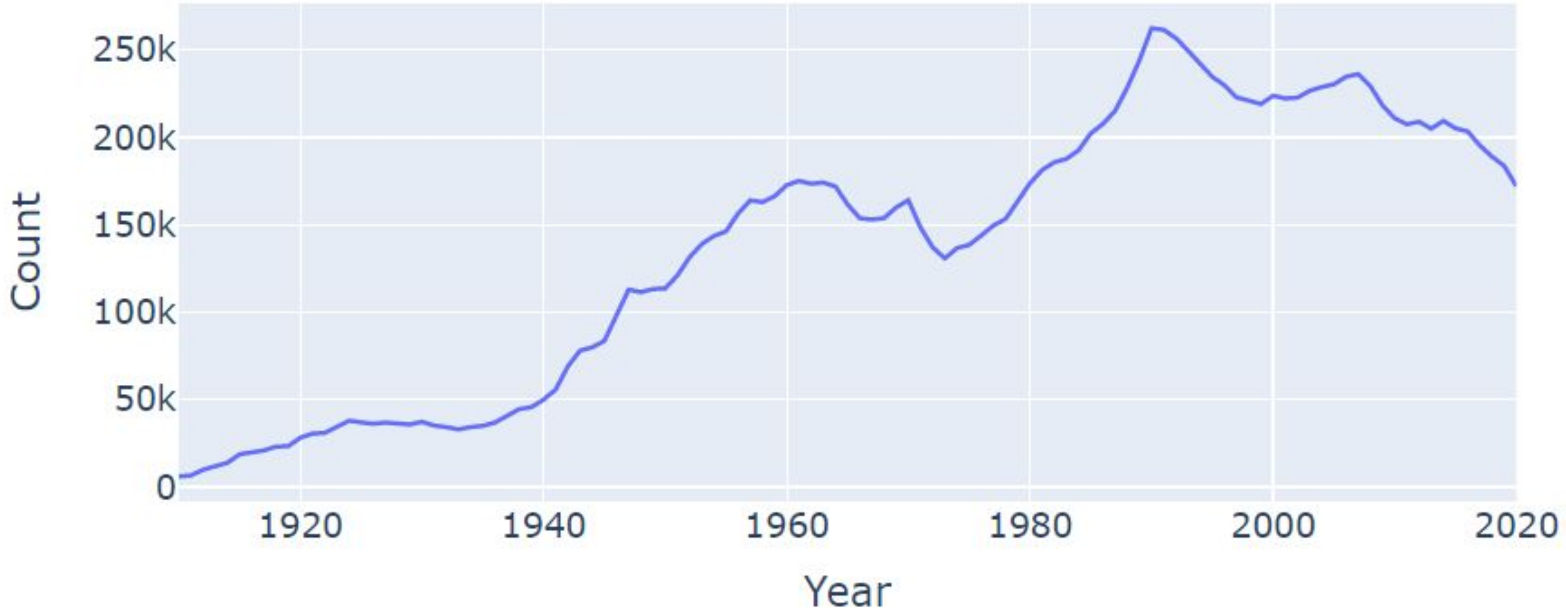
```
puzzle2 = female_babynames.groupby("Year")[["Count"]].agg(sum)
px.line(puzzle2, y = "Count")
```



A Word of Warning!

We made an enormous assumption when we decided to use this dataset to estimate the birth rate.

- According to https://lao.ca.gov/LAOEconTax/Article/Detail/691, the true number of babies born in California in 2020 was more than 400,000.
- What happened?

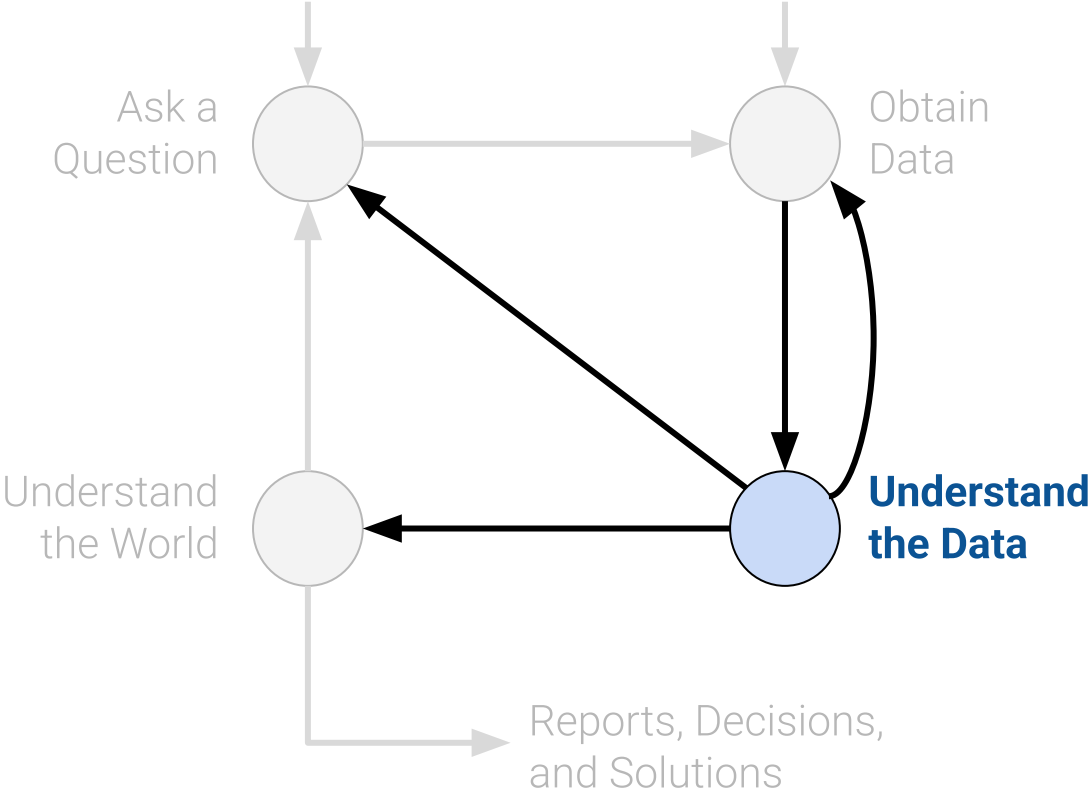


From Lecture 1: Exploratory Data Analysis and Visualization

- How is our data organized and what does it contain?
- Do we already have relevant data?
- What are the biases, anomalies, or other issues with the data?
- How do we transform the data to enable effective analysis?

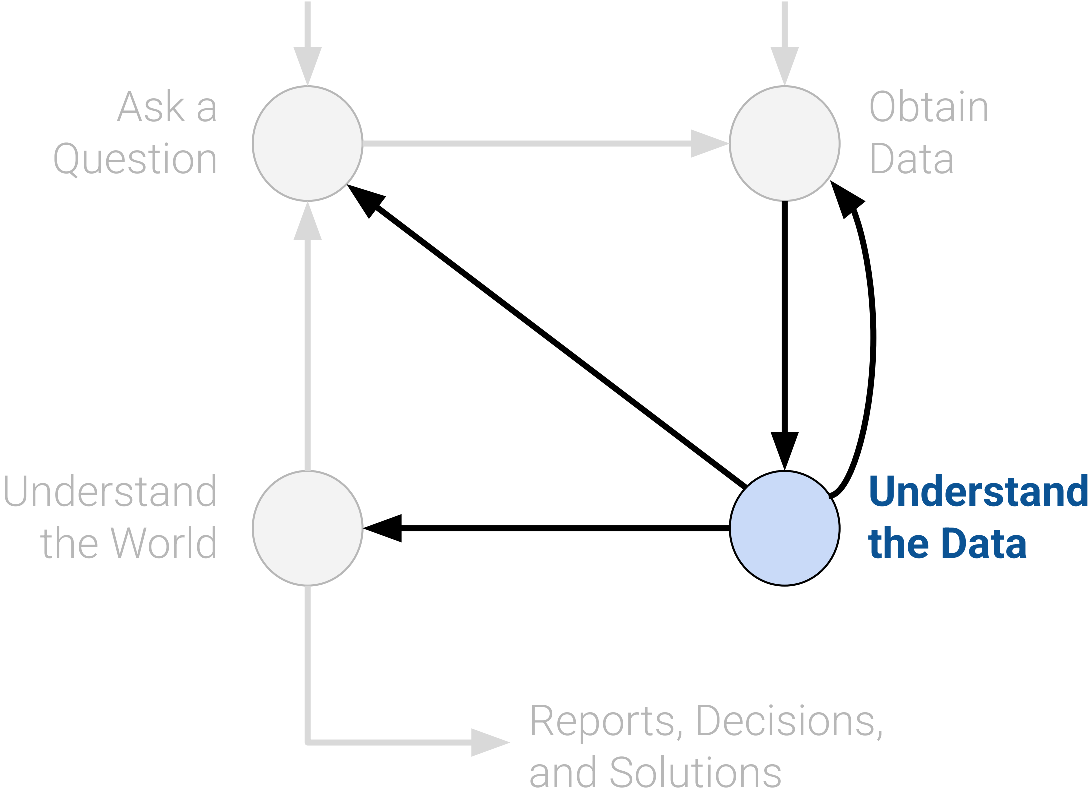
Bottom line: Blindly using tools is dangerous!

Lisa will cover EDA next week.



What are the biases, anomalies, or other issues with the data?

- **We only used names for babies who are female at birth.**
- Not all babies register for social security.
- The database does not include names of popularity less than 5 per year (for example both of my kids).



One more groupby Puzzle

Lecture 04, Data 100 Spring 2022

Groupby Puzzle #4

Why does the table seem to claim that Woodrow Wilson won the presidency in 2020?

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

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

groupby Puzzle #4

Why does the table seem to claim that Woodrow Wilson won the presidency in 2020?

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

Every column is calculated independently! Among Democrats:

- Last year they ran: 2020
- Alphabetically latest candidate name: Woodrow Wilson
- Highest % of vote: 61.34

	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

Quick Subpuzzle

A	3	ak
B	1	tx
C	4	fl
A	1	hi
B	5	mi
C	9	ak
A	2	ca
C	5	sd
B	6	nc

`groupby`

A	3	ak
A	1	hi
A	2	ca

B	1	tx
B	5	mi
B	6	nc

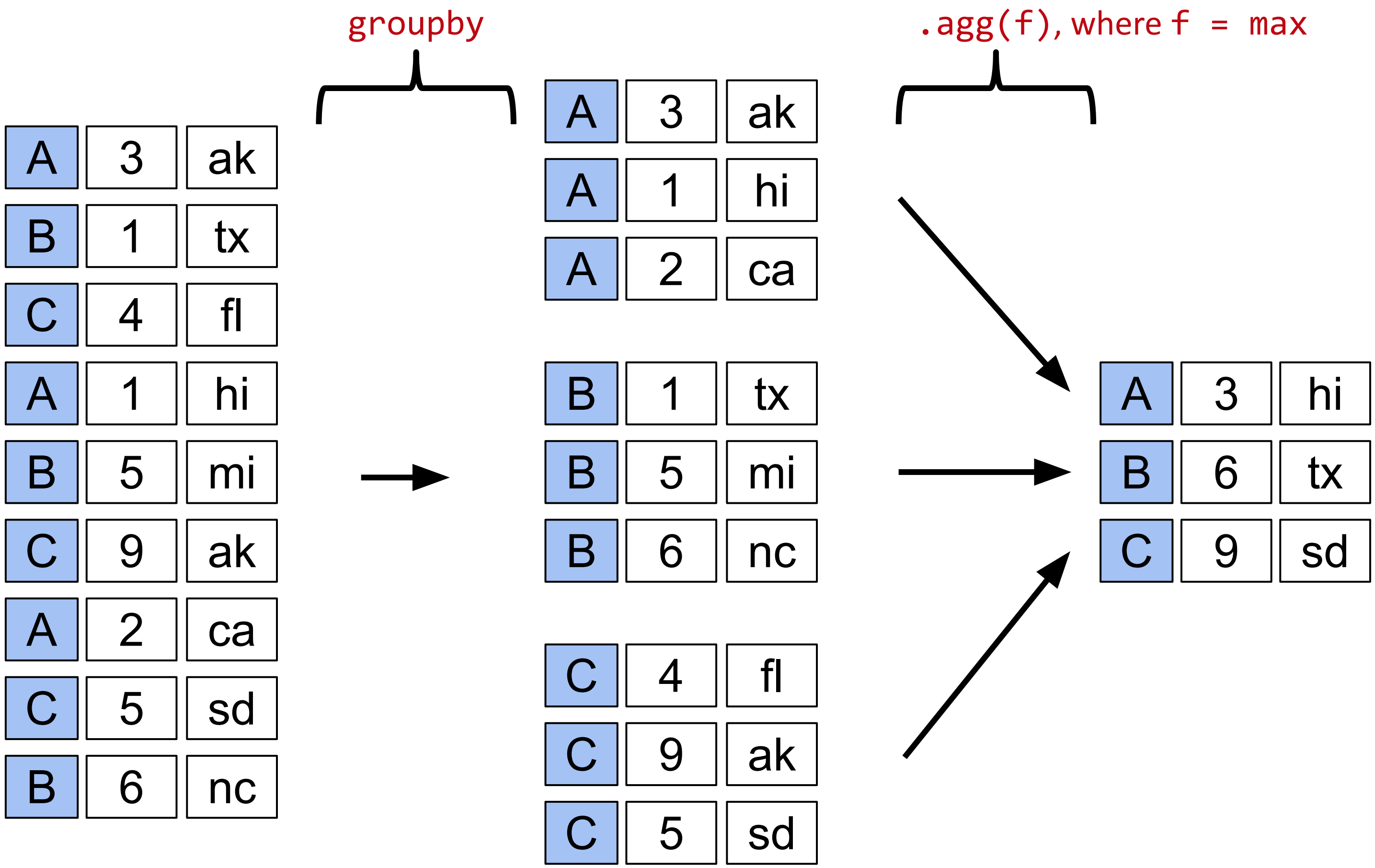
C	4	fl
C	9	ak
C	5	sd

`.agg(f)`, where `f = max`

A	3	??
B	6	??
C	9	??

What will go in the ??

Quick Subpuzzle



Puzzle #4

Very hard puzzle: Try to write code that returns the table below.

- Each row shows the best result (in %) by each party.
    - For example: Best Democratic result ever was Johnson's 1964 win.

	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

Puzzle #4

Very hard puzzle: Try to write code that returns the table below.

- Hint, first do: `elections_sorted_by_percent = elections.sort_values("%", ascending=False)`
- Each row shows the best result (in %) by each party.

	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

Puzzle #4

Very hard puzzle: Try to write code that returns the table below.

- First sort the DataFrame so that rows are in ascending order of %.
- Then group by Party and take the first item of each series.
- Note: Lab will give you a chance to try this out if you didn't quite follow during lecture.

```
elections_sorted_by_percent = elections.sort_values("%", ascending=False)
elections_sorted_by_percent.groupby("Party").agg(lambda x : x.iloc[0])
```

	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`

→

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

There's More Than One Way to Find the Best Result by Party

In Pandas, there's more than one way to get to the same answer.

- Each approach has different tradeoffs in terms of readability, performance, memory consumption, complexity, etc.
- Takes a very long time to understand these tradeoffs!
- If you find your current solution to be particularly convoluted or hard to read, maybe try finding another way!

Alternate Approaches

```
elections_sorted_by_percent = elections.sort_values("%", ascending=False)
elections_sorted_by_percent.groupby("Party").agg(lambda x : x.iloc[0])
```

Some examples that use syntax we haven't discussed in class:

```
best_per_party = elections.loc[elections.groupby('Party')['%'].idxmax()]
```

```
best_per_party2 = elections.sort_values('%').drop_duplicates(['Party'], keep='last')
```

See today's lecture notebook if you want to explore **`idxmax`** and **`drop_duplicates`**.

- We won't cover these formally in the course.

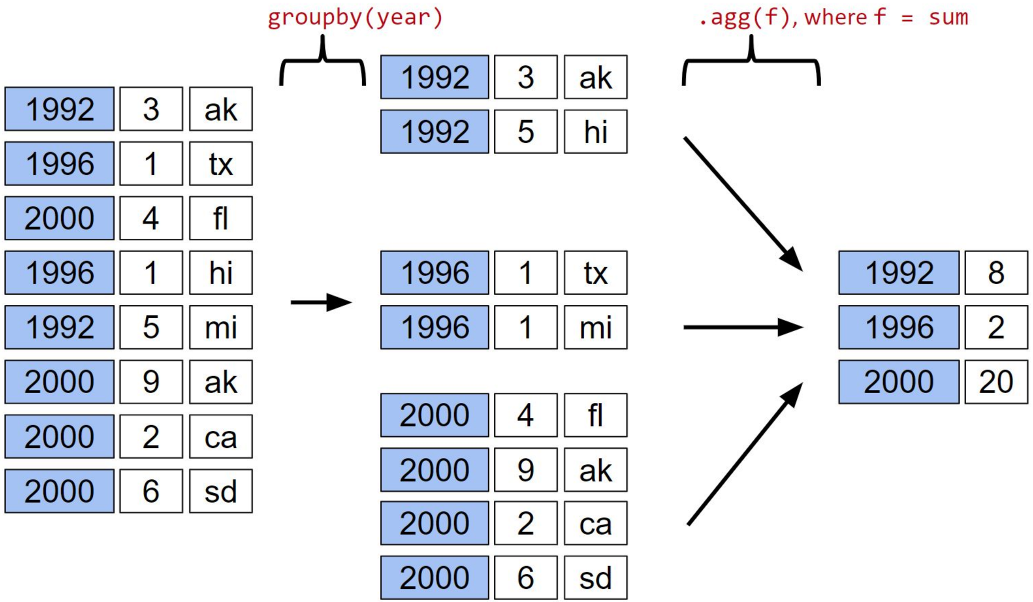
Other DataFrameGroupBy Features

Lecture 04, Data 100 Spring 2022

Revisiting `groupby.agg`

So far, we've seen that `df.groupby("year").agg(sum)`:

- Organizes all rows with the same year into a subframe for that year.
- Creates a new dataframe with one row representing each subframe year.
  - All rows in each subframe are combined using the sum function.



Raw groupby Objects

The result of a groupby operation applied to a DataFrame is a `DataFrameGroupBy` object.

- It is not a DataFrame!

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

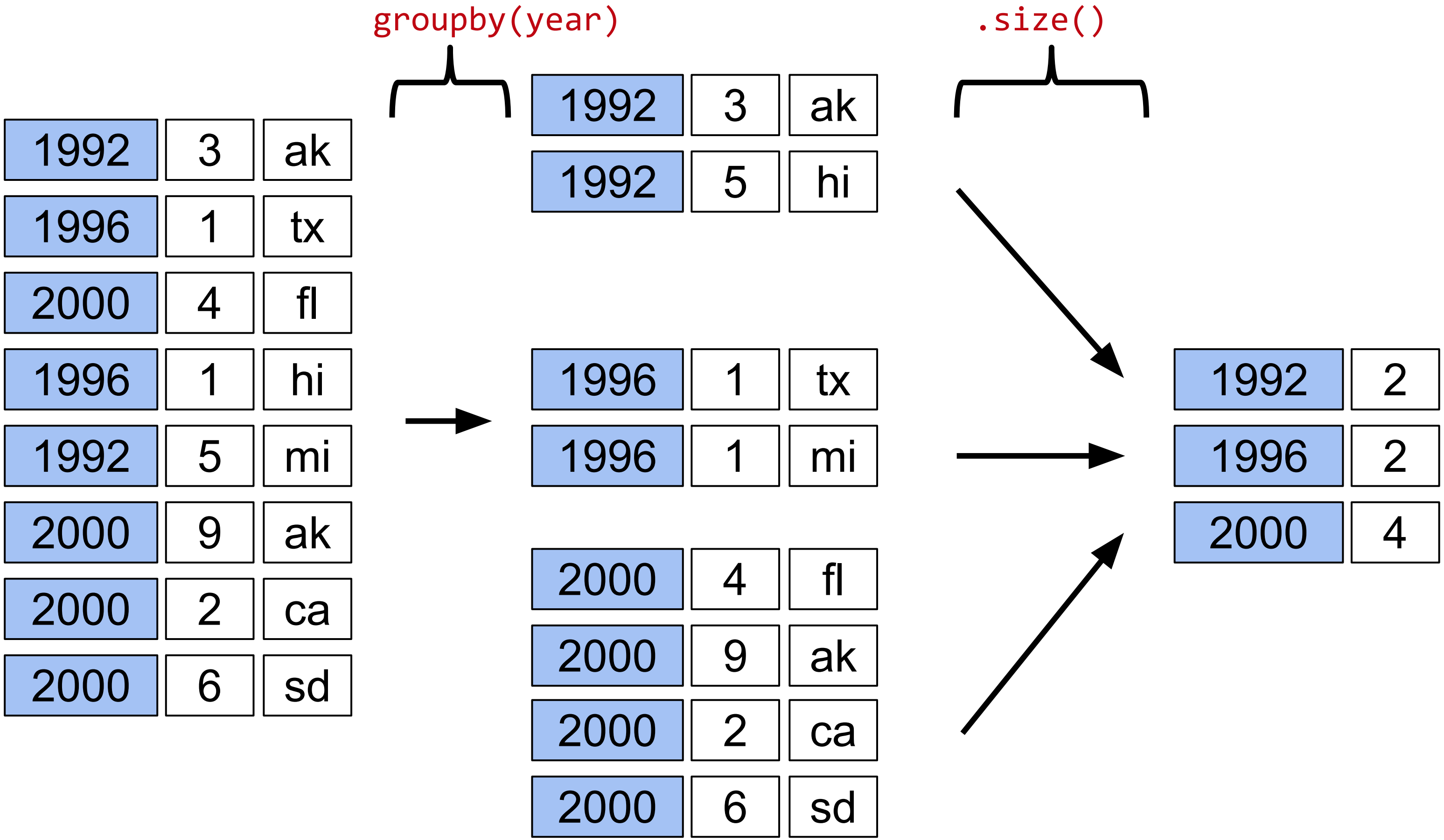
```
pandas.core.groupby.generic.DataFrameGroupBy
```

Given a DataFrameGroupBy object, can use various functions to generate DataFrames (or Series). `agg` is only one choice:

- `agg`: Creates a new DataFrame with one aggregated row per subframe.
- `max`: Creates a new DataFrame aggregated using the max function.
- `size`: Creates a new Series with the size of each subframe.
- `filter`: Creates a copy of the original DataFrame, but keeping only rows from subframes that obey the provided condition.

See https://pandas.pydata.org/docs/reference/groupby.html for a list of DataFrameGroupBy methods.

groupby.size()

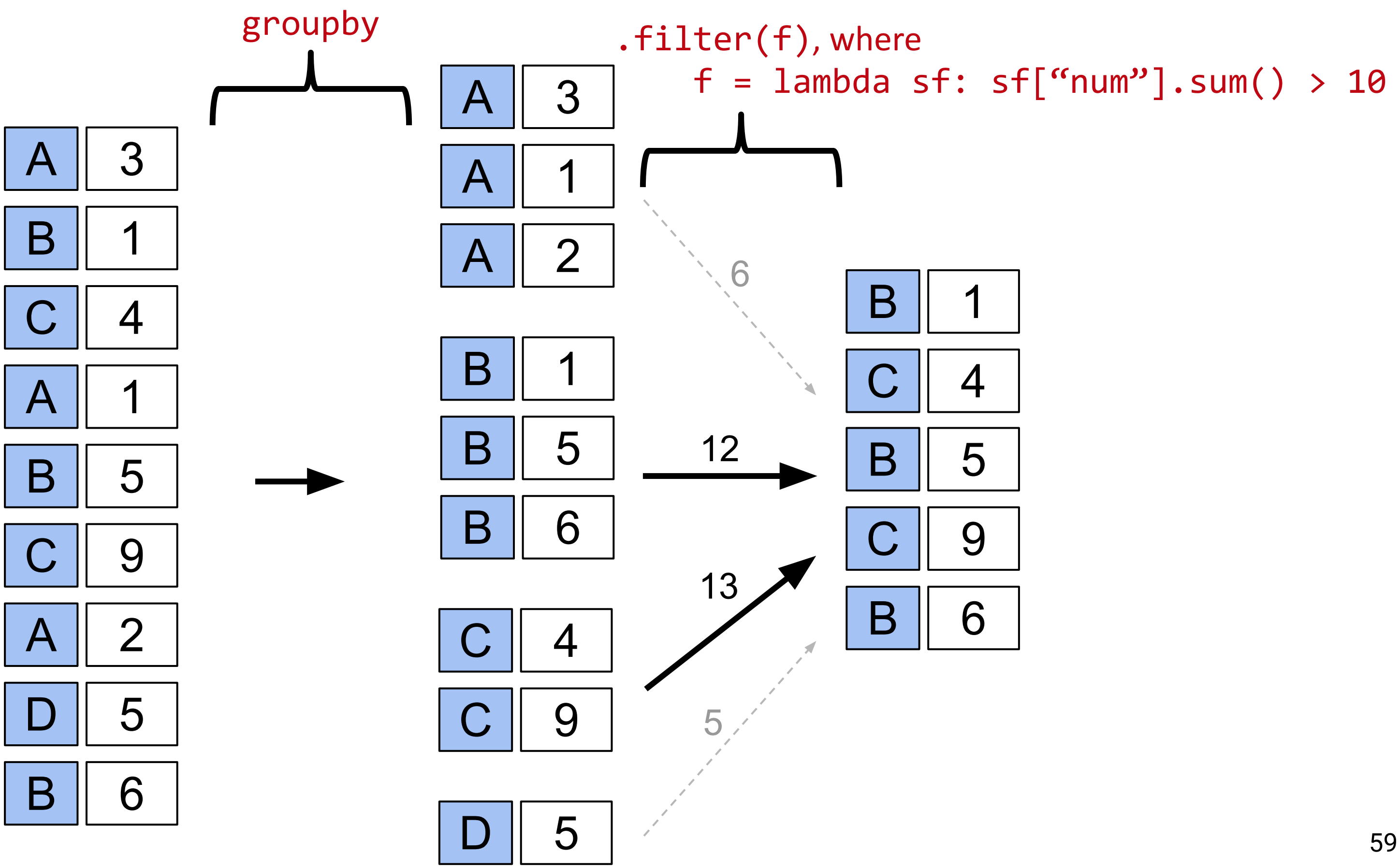


Filtering by Group

Another common use for groups is to filter data.

- `groupby.filter` takes an argument `f`.
- `f` is a function that:
  - Takes a DataFrame as input.
  - Returns either true or false.
- For each group g, f is applied to the subframe comprised of the rows from the original dataframe corresponding to that group.

groupby.filter



Groupby and PivotTables

Lecture 04, Data 100 Spring 2022

Grouping by Multiple Columns

Suppose we want to build a table showing the total number of babies born of each sex in each year. One way is to *groupby using both columns* of interest:

Example:
```
babynames.groupby(["Year", "Sex"]).agg(sum).head(6)
```

Year	Sex	Count
1910	F	5950
	M	3213
1911	F	6602
	M	3381
1912	F	9803
	M	8142

Note: Resulting DataFrame is multi-indexed. That is, its index has multiple dimensions. Will explore in a later lecture.

Pivot Tables

A more natural approach is to use our Data 8 brains and create a pivot table.

```
babynames_pivot = babynames.pivot_table(
    index='Year',       # rows (turned into index)
    columns='Sex',      # column values
    values=['Count'], # field(s) to process in each group
    aggfunc=np.sum,     # group operation
)
babynames_pivot.head(6)
```

	Count	
Sex	F	M
Year		
1910	5950	3213
1911	6602	3381
1912	9804	8142
1913	11860	10234
1914	13815	13111
1915	18643	17192

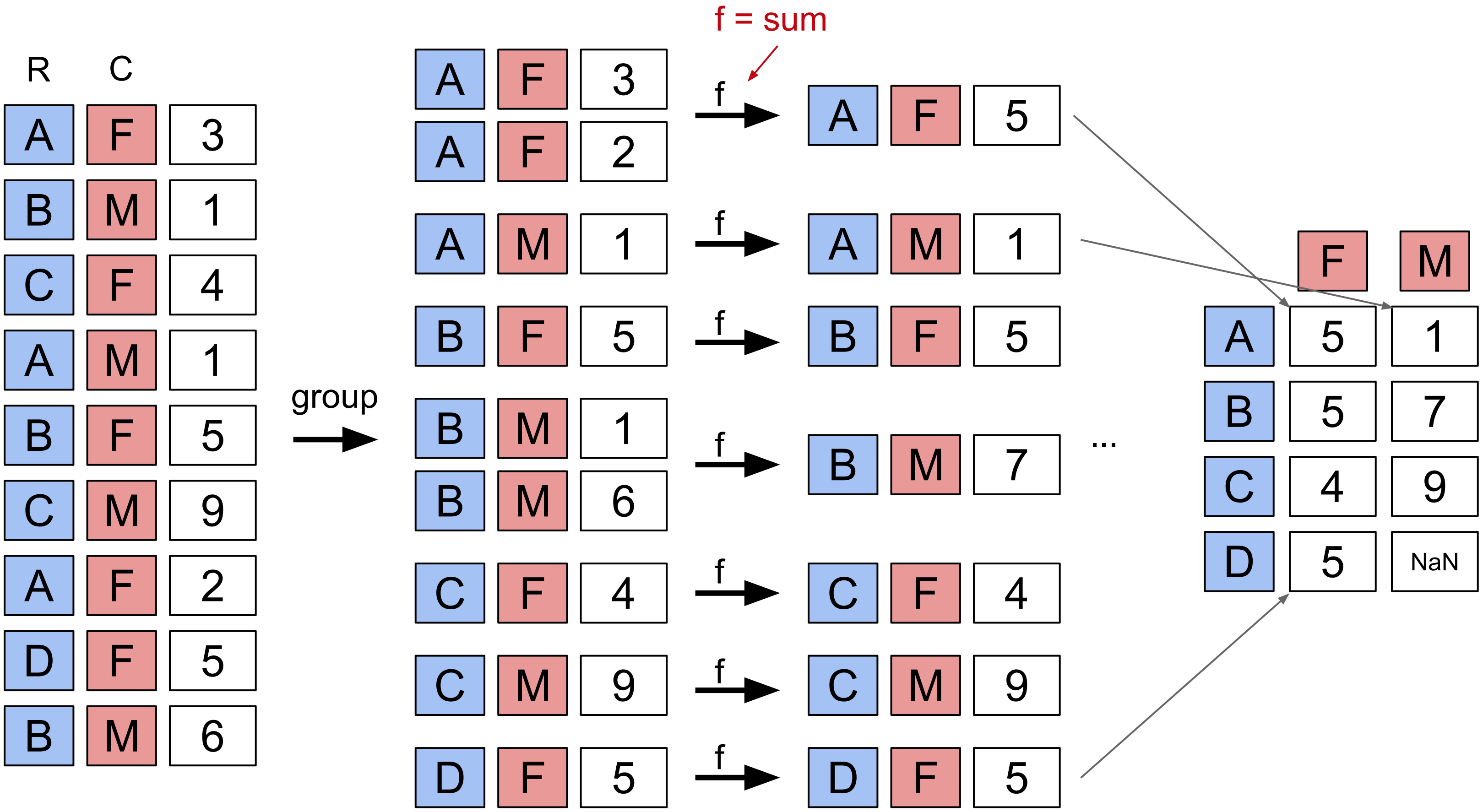
groupby([“Year”, “Sex”]) vs. pivot_table

The pivot table more naturally represents our data.

		Count
Year	**Sex**	
1910	**F**	5950
	M	3213
1911	**F**	6602
	M	3381
1912	**F**	9803
	M	8142

	Count	
Sex	**F**	**M**
Year		
1910	5950	3213
1911	6602	3381
1912	9804	8142
1913	11860	10234
1914	13815	13111
1915	18643	17192

Pivot Table Mechanics



Pivot Tables

We can include multiple values in our pivot tables.

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

A Quick Look at Joining Tables

Lecture 04, Data 100 Spring 2022

- Googling Custom Sorts
- Adding, Modifying, and Removing Columns
- Groupby.agg
- Some groupby.agg Puzzles
- One more groupby Puzzle
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- **A Quick Look at Joining Tables**

Joining Tables

Suppose want to know the 2020 male popularity of presidential candidate's names.

- Example: Dwight Eisenhower's name Dwight is not popular today, with only 5 babies born with this name in California in 2020.

To solve this problem, we'll have to join tables.

- This will be almost exactly like Table.join from data 8 (http://data8.org/datascience/_autosummary/datascience.tables.Table.join.html)

Creating Table 1: Male Babynames

Let's set aside only male names from 2020 first:

```
male_2020_babynames = babynames.query('Sex == "M" and Year == 2020')
male_2020_babynames
```

	State	Sex	Year	Name	Count
392447	CA	M	2020	Deandre	19
394024	CA	M	2020	Leandre	5
392438	CA	M	2020	Andreas	19
391863	CA	M	2020	Leandro	72
392562	CA	M	2020	Rudra	17
...	...	...	...	...	...

Creating Table 2: Presidents with First Names

To join our table, we'll also need to set aside the first names of each candidate.

- You'll have a chance to write this code again on lab, so don't worry about the details too much.

```
elections["First Name"] = elections["Candidate"].str.split().str[0]
```

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

Joining Our Tables

```
merged = pd.merge(left = elections, right = male_2020_babynames,
                  left_on = "First Name", right_on = "Name")
```

	Year_x	Candidate	Party	Popular vote	Result	%	First Name	State	Sex	Year_y	Name	Count
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	CA	M	2020	Andrew	867
1	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	CA	M	2020	Andrew	867
2	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew	CA	M	2020	Andrew	867
3	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878	John	CA	M	2020	John	617
4	1828	John Quincy Adams	National Republican	500897	loss	43.796073	John	CA	M	2020	John	617

Lab

We'll talk more about joining tables in a future lecture.

- Note: Your code in the lab will also be capable of finding the popularity of female presidential candidate names!

New Syntax / Concept Summary

Today we covered:

- Sorting with a custom key.
- Creating and dropping columns.
- Groupby: Output of .groupby(“Name”) is a DataFrameGroupBy object. Condense back into a DataFrame or Series with:
  - groupby.agg
  - groupby.size
  - groupby.filter
  - and more...
- Pivot tables: An alternate way to group by exactly two columns.
- Joining tables using pd.merge.

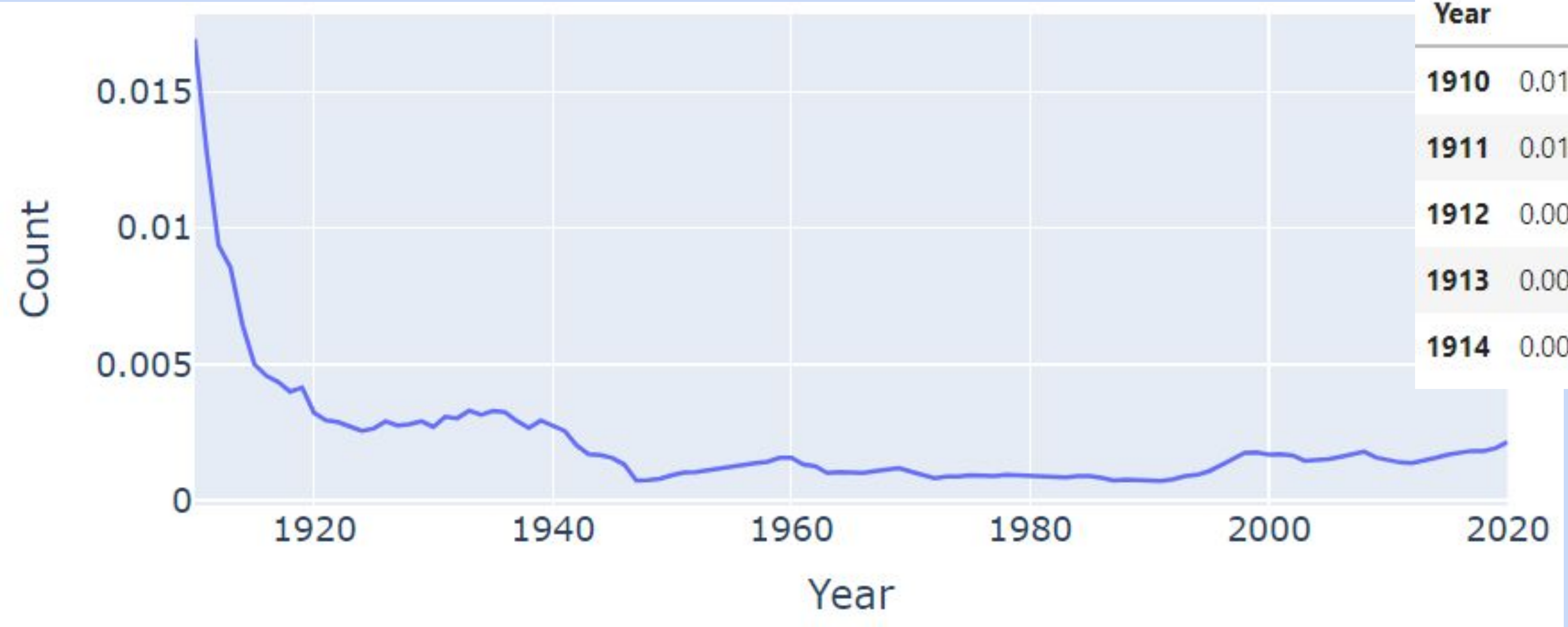
Groupby Puzzle #4 (Tricky!)

You might expect that the code below gives the relative birth rate for each year.

```
female_babynames.groupby("Year")[["Count"]].agg(ratio_to_peak)
```

However, this code actually yields the result shown. What is this?

Year	Count
1910	0.016949
1911	0.012821
1912	0.009363
1913	0.008562
1914	0.006468



Groupby Puzzle #3 (Tricky!)

You might expect that the code below gives the ratio_to_peak for the birth rate for each year.

```
female_babynames.groupby("Year")[["Count"]].agg(ratio_to_peak)
```

However, this code actually yields the result shown. What is this?

Consider the 1914 names:

	State	Sex	Year	Name	Count
1120	CA	F	1914	Mary	773
1121	CA	F	1914	Dorothy	564
1122	CA	F	1914	Helen	471
1123	CA	F	1914	Margaret	394
1124	CA	F	1914	Ruth	294
...	...	...	...	...	...
1483	CA	F	1914	Rhea	5
1484	CA	F	1914	Rosamond	5
1485	CA	F	1914	Rosina	5
1486	CA	F	1914	Sophia	5
1487	CA	F	1914	Tomiko	5

`max(series)`

`series.iloc[-1]`

	Count
Year	
1910	0.016949
1911	0.012821
1912	0.009363
1913	0.008562
1914	0.006468

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

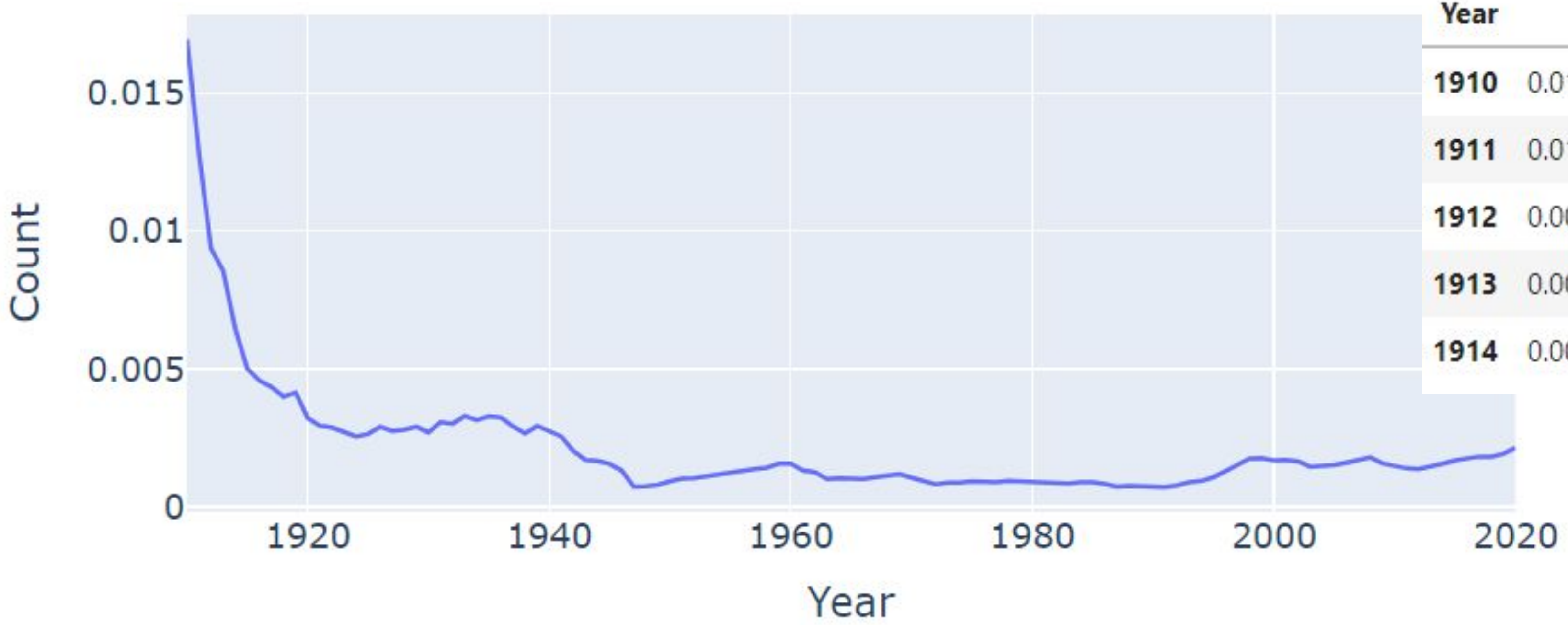
5 / 773 = **0.006468**

Groupby Puzzle #3 (Tricky!)

You might expect that the code below gives the ratio_to_peak for the birth rate for each year.

```
female_babynames.groupby("Year")[["Count"]].agg(ratio_to_peak)
```

Instead it plots 5 / max(count) for each year.



	Count
Year	
1910	0.016949
1911	0.012821
1912	0.009363
1913	0.008562
1914	0.006468

Computing Relative Birth Counts

In puzzle two we wrote:

```
puzzle2 = female_babynames.groupby("Year")[["Count"]].agg(sum)
```

To compute relative birth counts, we'd just divide this DataFrame by its max value.

Year	Count
1910	5950
1911	6602
1912	9804
1913	11860
1914	13815
...	...

```
puzzle2 / max(puzzle2["Count"])
```

Year	Count
1910	0.022677
1911	0.025162
1912	0.037366
1913	0.045202
1914	0.052653
...	...