LECTURE 4

# Pandas, Part II

More on Pandas (Utility Functions, Grouping, Aggregation, Merge)

# Recap: DataFrames of Series!

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`**.

```
0      1824
1      1824
2      1828
3      1828
4      1832
       ...
177    2016
178    2020
179    2020
180    2020
181    2020
Name: Year,
```

The Series **`"Year"`**

```
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,
```

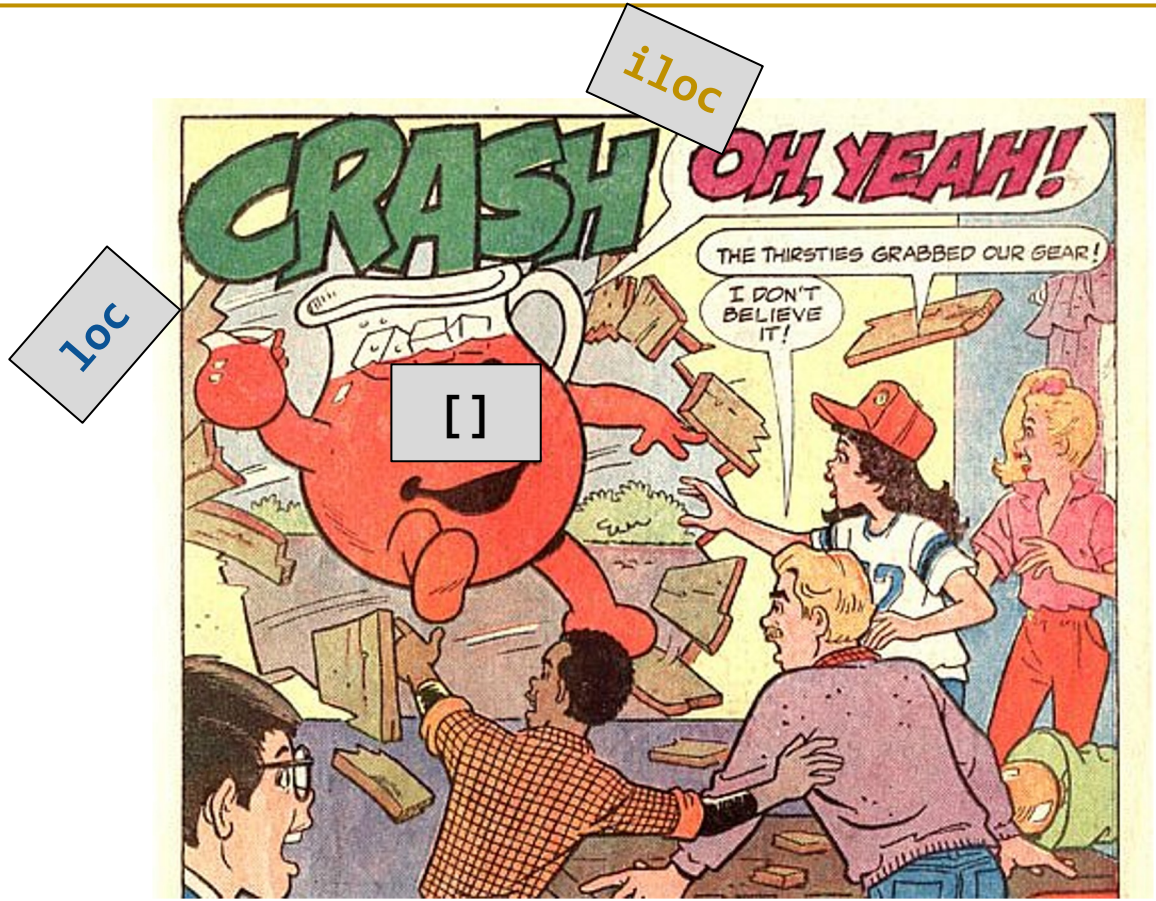
The Series **`"Candidate"`**

[...]

| | 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 |

The DataFrame **`elections`**

# Recap: .loc, .iloc and [ ]



# New Syntax / Concept Summary

Today we'll cover:

- Conditional selection.
- Handy utility functions.
- 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.

# More on Conditional Selection

- **Conditional Selection**
- Adding, Modifying, and Removing Columns
- Handy Utility Functions
- Custom Sort
- Groupby.agg
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- Joining Tables

# Boolean Array Input

Yet another input type supported by `loc` and `[]` is the boolean array.

```
babynames_first_10_rows = babynames.loc[:9, :]
```

| | 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 |
| 5 | CA | F | 1910 | Ruth | 128 |
| 6 | CA | F | 1910 | Evelyn | 126 |
| 7 | CA | F | 1910 | Alice | 118 |
| 8 | CA | F | 1910 | Virginia | 101 |
| 9 | CA | F | 1910 | Elizabeth | 93 |

```
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 |

# Boolean Array Input

We can perform the same operation using `loc`.

```
babynames_first_10_rows = babynames.loc[:9, :]
```

| | 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 |
| 5 | CA | F | 1910 | Ruth | 128 |
| 6 | CA | F | 1910 | Evelyn | 126 |
| 7 | CA | F | 1910 | Alice | 118 |
| 8 | CA | F | 1910 | Virginia | 101 |
| 9 | CA | F | 1910 | Elizabeth | 93 |

```
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 |

# Boolean Array Input

Useful because boolean arrays can be generated by using logical operators on Series.

Length 400761 Series where every entry is either “True” or “False”, where “True” occurs for every babyname with "Sex" = "F".

```
logical_operator = (babynames["Sex"] == "F")
0          True
1          True
2          True
3          True
4          True
          ...
400757    False
400758    False
400759    False
400760    False
400761    False
Name: Sex, Length: 400762, dtype: bool
```

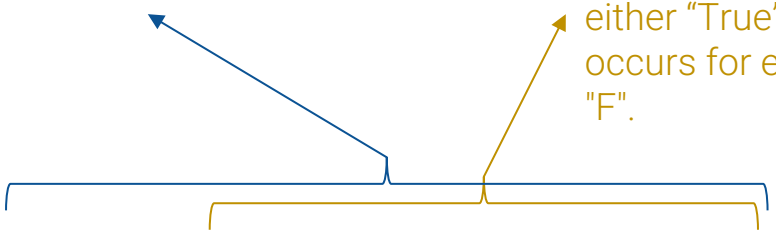
True in rows 0, 1, 2, …

# Boolean Array Input

Useful because boolean arrays can be generated by using logical operators on `Series`.

Length 235791 Series where every entry belongs to a babyname with "Sex" = "F"

Length 400761 Series where every entry is either “True” or “False”, where “True” occurs for every babyname with "Sex" = "F".



```
babynames[babynames["Sex"] == "F"]
```

```
0          True
1          True
2          True
3          True
4          True
          ...
400757    False
400758    False
400759    False
400760    False
400761    False
Name: Sex, Length: 400762, dtype: bool
```

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **235786** | CA | F | 2021 | Zarahi | 5 |
| **235787** | CA | F | 2021 | Zelia | 5 |
| **235788** | CA | F | 2021 | Zenobia | 5 |
| **235789** | CA | F | 2021 | Zeppelin | 5 |
| **235790** | CA | F | 2021 | Zoraya | 5 |

235791 rows × 5 columns

# Boolean Array Input

Can also use **.loc**.

Length 235791 Series where every entry belongs to a babyname with "Sex" = "F"

Length 400761 Series where every entry is either “True” or “False”, where “True” occurs for every babyname with "Sex" = "F".

```
babynames.loc[babynames["Sex"] == "F"]
0          True
1          True
2          True
3          True
4          True
          ...
400757    False
400758    False
400759    False
400760    False
400761    False
Name: Sex, Length: 400762, dtype: bool
```

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **235786** | CA | F | 2021 | Zarahi | 5 |
| **235787** | CA | F | 2021 | Zelia | 5 |
| **235788** | CA | F | 2021 | Zenobia | 5 |
| **235789** | CA | F | 2021 | Zeppelin | 5 |
| **235790** | CA | F | 2021 | Zoraya | 5 |

235791 rows × 5 columns

# Boolean Array Input

Boolean **`Series`** can be combined using various operators, allowing filtering of results by multiple criteria.

- Example: The **&** operator.

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

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **149044** | CA | F | 1999 | Zareen | 5 |
| **149045** | CA | F | 1999 | Zeinab | 5 |
| **149046** | CA | F | 1999 | Zhane | 5 |
| **149047** | CA | F | 1999 | Zoha | 5 |
| **149048** | CA | F | 1999 | Zoila | 5 |

149049 rows × 5 columns

## Exercise:

Which of the following pandas statements returns a `DataFrame` of the first 3 baby names with `Count > 250`.

```
babynames.iloc[[0, 233, 484], [3, 4]]
babynames.loc[[0, 233, 484]]
babynames.loc[babynames["Count"] > 250, ["Name", "Count"]].head(3)
babynames.loc[babynames["Count"] > 250, ["Name", "Count"]].iloc[0:2, :]
```

| | 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 |
| 5 | CA | F | 1910 | Ruth | 128 |
| 6 | CA | F | 1910 | Evelyn | 126 |

| | Name | Count |
|---|---|---|
| 0 | Mary | 295 |
| 233 | Mary | 390 |
| 484 | Mary | 534 |

# Answer

Which of the following pandas statements returns a `DataFrame` of the first 3 baby names with `Count > 250`.

```
babynames.iloc[[0, 233, 484], [3, 4]]
```

~~`babynames.loc[[0, 233, 484]]`~~

```
babynames.loc[babynames["Count"] > 250, ["Name", "Count"]].head(3)
```

~~`babynames.loc[babynames["Count"] > 250, ["Name", "Count"]].iloc[0:2, :]`~~

| | 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 |
| 5 | CA | F | 1910 | Ruth | 128 |
| 6 | CA | F | 1910 | Evelyn | 126 |

| | Name | Count |
|---|---|---|
| 0 | Mary | 295 |
| 233 | Mary | 390 |
| 484 | Mary | 534 |

# Alternatives to Boolean Array Selection

Boolean array selection is a useful tool, but can lead to overly verbose code for complex conditions.

```
babynames[(babynames["Name"] == "Bella") |
          (babynames["Name"] == "Alex") |
          (babynames["Name"] == "Ani") |
          (babynames["Name"] == "Lisa")]
```

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **386576** | CA | M | 2017 | Alex | 482 |
| **389498** | CA | M | 2018 | Alex | 494 |
| **392360** | CA | M | 2019 | Alex | 436 |
| **395230** | CA | M | 2020 | Alex | 378 |
| **398031** | CA | M | 2021 | Alex | 331 |

359 rows × 5 columns

Pandas provides **many** alternatives, for example:

- `.isin`
- `.str.startswith`
- `.groupby.filter`

# Alternatives to Boolean Array Selection

Pandas provides **many** alternatives, for example:

- `.isin`
- `.str.startswith`
- `.groupby.filter`

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

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **386576** | CA | M | 2017 | Alex | 482 |
| **389498** | CA | M | 2018 | Alex | 494 |
| **392360** | CA | M | 2019 | Alex | 436 |
| **395230** | CA | M | 2020 | Alex | 378 |
| **398031** | CA | M | 2021 | Alex | 331 |

359 rows × 5 columns

# Alternatives to Boolean Array Selection

Pandas provides **many** alternatives, for example:

- `.isin`
- **`.str.startswith`**
- `.groupby.filter`

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

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **400648** | CA | M | 2021 | Nirvan | 5 |
| **400649** | CA | M | 2021 | Nivin | 5 |
| **400650** | CA | M | 2021 | Nolen | 5 |
| **400651** | CA | M | 2021 | Nomar | 5 |
| **400652** | CA | M | 2021 | Nyles | 5 |

11994 rows × 5 columns

# Adding, removing, and modifying columns

- Conditional Selection
- **Adding, removing, and modifying columns**
- Handy Utility Functions
- Custom Sort
- Groupby.agg
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- Joining Tables

# Syntax for Adding a Column

Adding a column is easy:

1. Use `[ ]` to reference the desired new column.
2. Assign this column to a `Series` or array of the appropriate length.

```
# 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
```

| | 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 |
| **...** | ... | ... | ... | ... | ... | ... |
| **407423** | CA | M | 2022 | Zayvier | 5 | 7 |
| **407424** | CA | M | 2022 | Zia | 5 | 3 |
| **407425** | CA | M | 2022 | Zora | 5 | 4 |
| **407426** | CA | M | 2022 | Zuriel | 5 | 6 |
| **407427** | CA | M | 2022 | Zylo | 5 | 4 |

407428 rows × 6 columns

# Syntax for Modifying a Column

Modifying a column is very similar to adding a column.

1. Use `[ ]` to reference the existing column.
2. Assign this column 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
```

| | 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 |
| **...** | ... | ... | ... | ... | ... | ... |
| **407423** | CA | M | 2022 | Zayvier | 5 | 6 |
| **407424** | CA | M | 2022 | Zia | 5 | 2 |
| **407425** | CA | M | 2022 | Zora | 5 | 3 |
| **407426** | CA | M | 2022 | Zuriel | 5 | 5 |
| **407427** | CA | M | 2022 | Zylo | 5 | 3 |

407428 rows × 6 columns

# Syntax for Renaming a Column

Rename a column using the (creatively named) `.rename()` method.

- `.rename()` takes in a **dictionary** that maps old column names to their new ones.

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

| | 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 |
| **...** | ... | ... | ... | ... | ... | ... |
| **407423** | CA | M | 2022 | Zayvier | 5 | 6 |
| **407424** | CA | M | 2022 | Zia | 5 | 2 |
| **407425** | CA | M | 2022 | Zora | 5 | 3 |
| **407426** | CA | M | 2022 | Zuriel | 5 | 5 |
| **407427** | CA | M | 2022 | Zylo | 5 | 3 |

407428 rows × 6 columns

# Syntax for Dropping a Column (or Row)

Remove columns using the (also creatively named) `.drop` method.

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

```
babynames = babynames.drop("Length", axis = "columns")
```

| | 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 |
| **...** | ... | ... | ... | ... | ... | ... |
| **407423** | CA | M | 2022 | Zayvier | 5 | 6 |
| **407424** | CA | M | 2022 | Zia | 5 | 2 |
| **407425** | CA | M | 2022 | Zora | 5 | 3 |
| **407426** | CA | M | 2022 | Zuriel | 5 | 5 |
| **407427** | CA | M | 2022 | Zylo | 5 | 3 |

407428 rows × 6 columns

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **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 |

407428 rows × 5 columns

# An Important Note: `DataFrame` Copies

Notice that we *re-assigned* **`babynames`** to an updated value on the previous slide.

```
babynames = babynames.drop("Length", axis = "columns")
```

By default, **`pandas`** methods create a **copy** of the **`DataFrame`**, without changing the original **`DataFrame`** at all. To apply our changes, we must update our **`DataFrame`** to this new, modified copy.

```
babynames.drop("Length", axis = "columns")
babynames
```

| | 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 |
| **...** | ... | ... | ... | ... | ... | ... |

Our change was not applied!

# Handy Utility Functions

- Conditional Selection
- Adding, Modifying, and Removing Columns
- **Handy Utility Functions**
- Custom Sorts
- Groupby.agg
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- Joining Tables

# Numpy

Pandas **`Series`** and **`DataFrames`** support a large number of operations, including mathematical operations, so long as the data is numerical.

```
bella_counts = babynames[babynames["Name"] ==
"Bella"]["Count"]
 np.mean(bella_counts)
 270.1860465116279


 max(bella_counts)
 902
```

```
6289        5
7512        8
35477       5
54487       7
58451       6
68845       6
73387       5
93601       5
96397       5
108054      7
111276      8
114677     10
117991     14
121524     17
125545     13
128946     18
132163     31
136362     15
139366     28
142917     27
146251     39
149607     65
153241     97
156955    122
160707    191
164586    213
168557    310
172646    334
176836    384
181090    439
185287    699
189455    902
193562    777
197554    761
201650    807
205629    704
209653    645
213592    643
217451    719
221207    747
224905    720
228576    566
232200    494
Name: Count, dtype: int64
```

# Pandas

In addition to its rich syntax for indexing and support for other libraries (numpy, built-in functions), Pandas provides an enormous number of useful utility functions. Today, we'll discuss:

- `size/shape`
- `describe`
- `sample`
- `value_counts`
- `uniques`
- `sort_values`

# shape/size

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **400757** | CA | M | 2021 | Zyan | 5 |
| **400758** | CA | M | 2021 | Zyion | 5 |
| **400759** | CA | M | 2021 | Zyire | 5 |
| **400760** | CA | M | 2021 | Zylo | 5 |
| **400761** | CA | M | 2021 | Zyrus | 5 |

400762 rows × 5 columns

```
babynames.shape
(400762, 5)
```

```
babynames.size
2003810
```

# describe()

| | 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 |
| **...** | ... | ... | ... | ... | ... |
| **400757** | CA | M | 2021 | Zyan | 5 |
| **400758** | CA | M | 2021 | Zyion | 5 |
| **400759** | CA | M | 2021 | Zyire | 5 |
| **400760** | CA | M | 2021 | Zylo | 5 |
| **400761** | CA | M | 2021 | Zyrus | 5 |

400762 rows × 5 columns

```
babynames.describe()
```

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

# describe()

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

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

```
count     400762
unique         2
top            F
freq      235791
Name: Sex, dtype: object
```

# sample()

If you want a `DataFrame` with a random selection of rows, you can use the `sample()` method.

- By default, ***it is without replacement***. Use `replace=True` for **replacement**.
- Naturally, can be chained with other methods and operators (`iloc`, etc).

```
babynames.sample()
```

| | State | Sex | Year | Name | Count |
|---|---|---|---|---|---|
| **108418** | CA | F | 1988 | Janielle | 6 |

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

| | Year | Name | Count |
|---|---|---|---|
| **169346** | 2005 | Greta | 36 |
| **231690** | 2020 | Yui | 6 |
| **203404** | 2013 | Libby | 14 |
| **385359** | 2016 | Cael | 9 |
| **386609** | 2017 | Everett | 353 |

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

| | Year | Name | Count |
|---|---|---|---|
| **340297** | 2000 | Emmet | 8 |
| **339662** | 2000 | Fred | 17 |
| **150463** | 2000 | Rosalia | 18 |
| **152732** | 2000 | Shanae | 5 |

# value_counts()

The `Series.value_counts` method counts the number of occurrences of a each unique value in a `Series`.

- Return value is also a `Series`.

```
babyname["Name"].value_counts()
```

```
Jean         221
Francis      219
Guadalupe    216
Jessie       215
Marion       213
            ...
Janin          1
Jilliann       1
Jomayra        1
Karess         1
Zyrus          1
Name: Name, Length: 20239, dtype: int64
```

# unique()

The `Series.unique` method returns an array of every unique value in a `Series`.

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

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

# sort_values()

The `DataFrame.sort_values` and `Series.sort_values` methods sort a `DataFrame` (or `Series`).

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

```
380256      Aadan
362255      Aadan
365374      Aadan
394460    Aadarsh
366561      Aaden
           ...
232144      Zyrah
217415      Zyrah
197519      Zyrah
220674      Zyrah
400761      Zyrus
Name: Name, Length: 400762, dtype: object
```

# sort_values

The `DataFrame.sort_values` and `Series.sort_values` methods sort a `DataFrame` (or `Series`).

- The `DataFrame` version requires an argument specifying the column on which to sort.

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

| | State | Sex | Year | Name | Count |
|---|---|---|---|---|---|
| **263272** | CA | M | 1956 | Michael | 8262 |
| **264297** | CA | M | 1957 | Michael | 8250 |
| **313644** | CA | M | 1990 | Michael | 8247 |
| **278109** | CA | M | 1969 | Michael | 8244 |
| **279405** | CA | M | 1970 | Michael | 8197 |
| **...** | ... | ... | ... | ... | ... |
| **159967** | CA | F | 2002 | Arista | 5 |
| **159966** | CA | F | 2002 | Arisbeth | 5 |
| **159965** | CA | F | 2002 | Arisa | 5 |
| **159964** | CA | F | 2002 | Arionna | 5 |
| **400761** | CA | M | 2021 | Zyrus | 5 |

400762 rows × 5 columns

# Manipulating String Data

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

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

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

# Custom Sorts

- More on Conditional Selection
- Adding, Modifying, and Removing Columns
- Handy Utility Functions
- **Custom Sorts**
- Groupby.agg
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- Joining Tables

# Sorting By Length

Let’s try to solve the sorting problem with different approaches:

- 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 |
|---|---|---|---|---|---|---|
| **313143** | CA | M | 1989 | Franciscojavier | 6 | 15 |
| **333732** | CA | M | 1997 | Ryanchristopher | 5 | 15 |
| **330421** | CA | M | 1996 | Franciscojavier | 8 | 15 |
| **323615** | CA | M | 1993 | Johnchristopher | 5 | 15 |
| **310235** | CA | M | 1988 | Franciscojavier | 10 | 15 |

# Syntax for Column Addition

Adding a column is easy:

```
# 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
```

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 |
|---|---|---|---|---|---|---|
| **313143** | CA | M | 1989 | Franciscojavier | 6 | 15 |
| **333732** | CA | M | 1997 | Ryanchristopher | 5 | 15 |
| **330421** | CA | M | 1996 | Franciscojavier | 8 | 15 |
| **323615** | CA | M | 1993 | Johnchristopher | 5 | 15 |
| **310235** | CA | M | 1988 | Franciscojavier | 10 | 15 |

# 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 |
|---|---|---|---|---|---|---|
| **313143** | CA | M | 1989 | Franciscojavier | 6 | 15 |
| **340695** | CA | M | 2000 | Franciscojavier | 6 | 15 |
| **333732** | CA | M | 1997 | Ryanchristopher | 5 | 15 |
| **318049** | CA | M | 1991 | Ryanchristopher | 7 | 15 |
| **333556** | CA | M | 1997 | Franciscojavier | 5 | 15 |

| | State | Sex | Year | Name | Count |
|---|---|---|---|---|---|
| **313143** | CA | M | 1989 | Franciscojavier | 6 |
| **340695** | CA | M | 2000 | Franciscojavier | 6 |
| **333732** | CA | M | 1997 | Ryanchristopher | 5 |
| **318049** | CA | M | 1991 | Ryanchristopher | 7 |
| **333556** | CA | M | 1997 | Franciscojavier | 5 |

# 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.

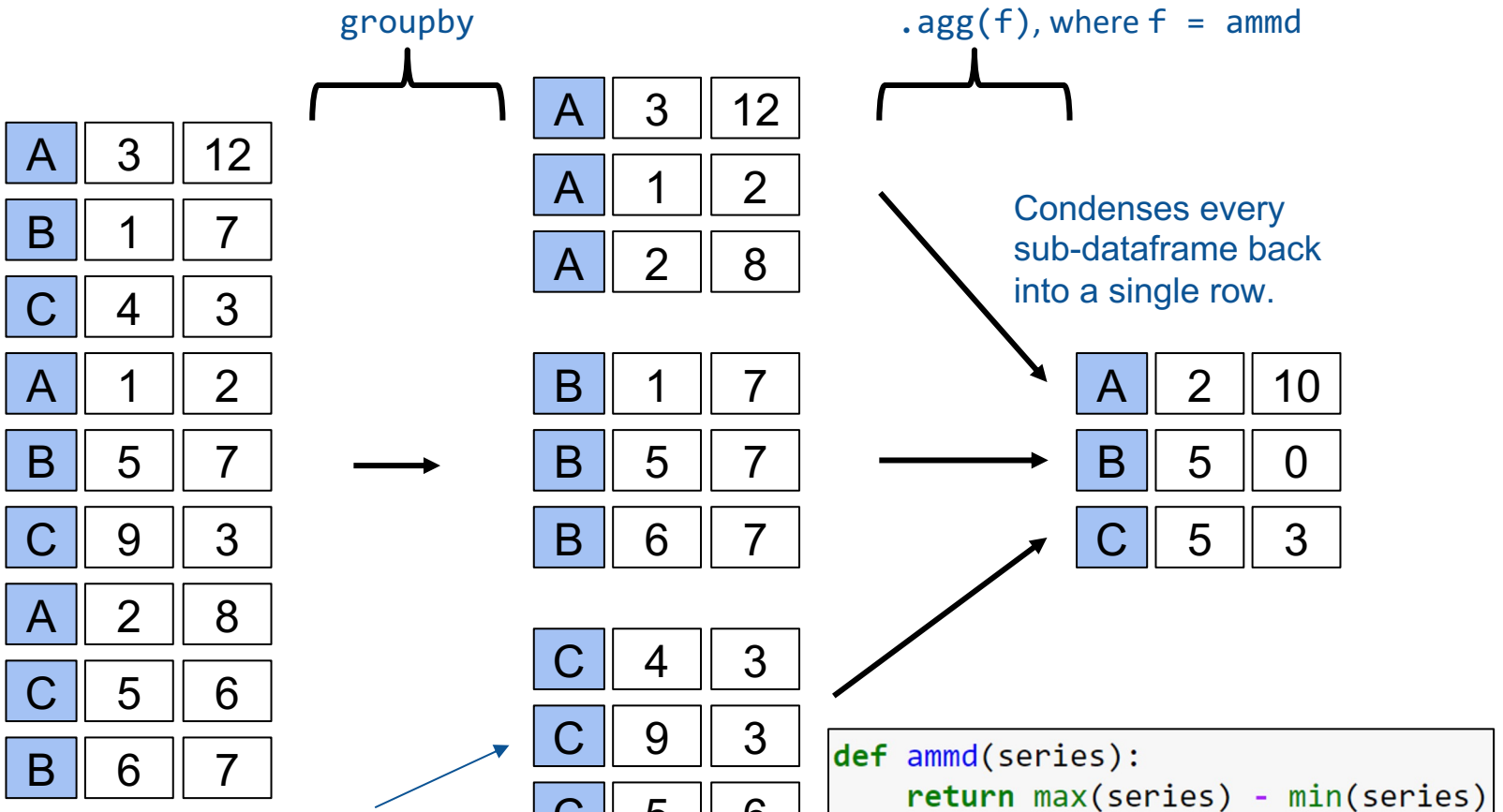
```
def dr_ea_count(string):
    return string.count('dr') + string.count('ea')


# Use `map` to apply `dr_ea_count` to each name in the "Name" column
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 |
|---|---|---|---|---|---|---|
| **304390** | CA | M | 1985 | Deandrea | 6 | 3 |
| **131022** | CA | F | 1994 | Leandrea | 5 | 3 |
| **101969** | CA | F | 1986 | Deandrea | 6 | 3 |
| **108723** | CA | F | 1988 | Deandrea | 5 | 3 |
| **115950** | CA | F | 1990 | Deandrea | 5 | 3 |

# Groupby.agg

- More on Conditional Selection
- Adding, Modifying, and Removing Columns
- Handy Utility Functions
- Custom Sorts
- **Groupby.agg**
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- Joining Tables

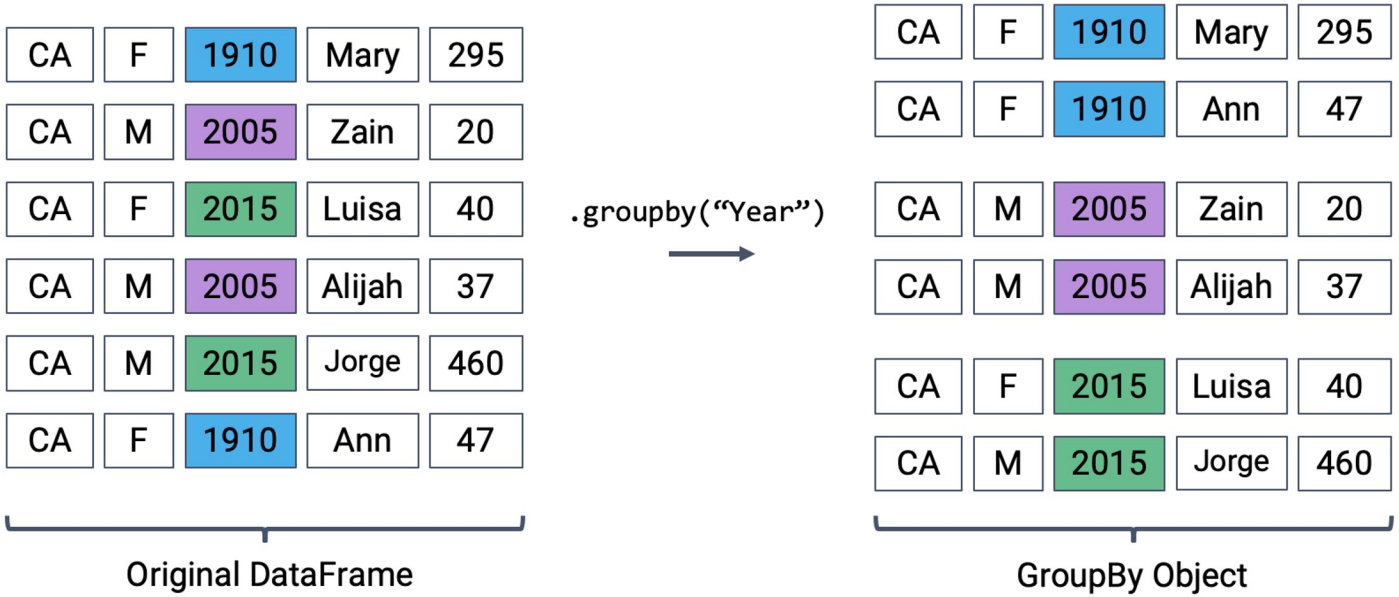
# Visual Review: Grouping and Collection



# groupby()

A `groupby` operation involves some combination of **splitting the object, applying a function**, and **combining the results**.

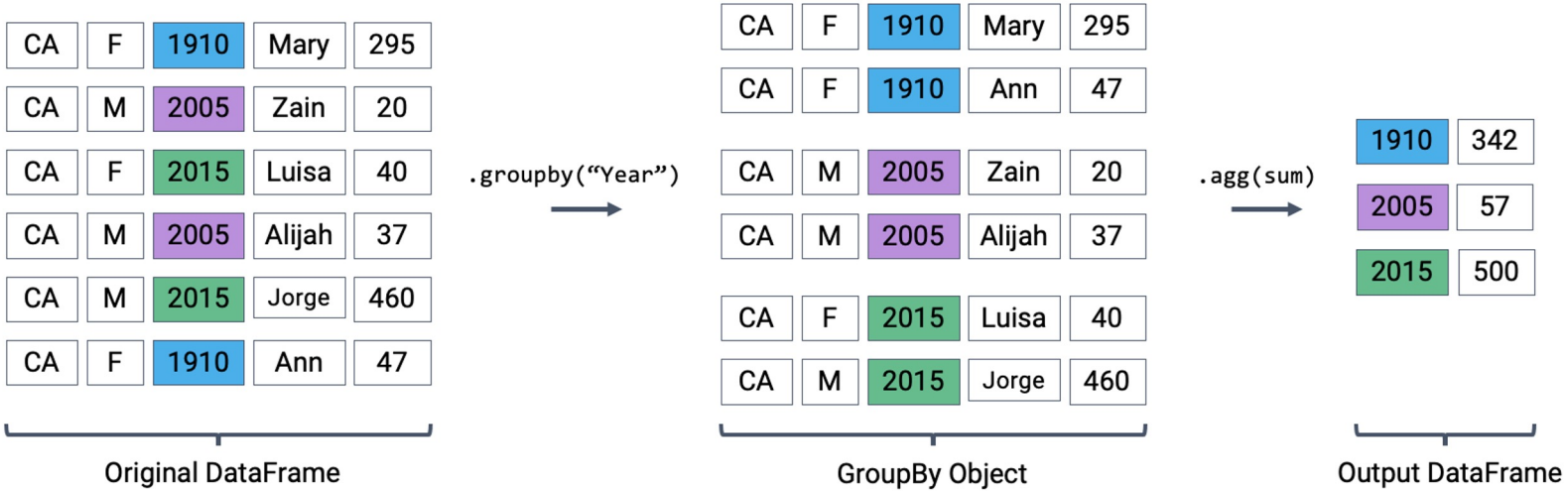
- Calling `.groupby()` generates `DataFrameGroupBy` objects → "mini" sub-DataFrames
- Each subframe contains all rows that correspond to a particular year



# groupby.agg

A `groupby` operation involves some combination of **splitting the object, applying a function**, and **combining the results**.

- Calling `.groupby()` generates `DataFrameGroupBy` objects → "mini" sub-DataFrames
- Each subframe contains all rows that correspond to a particular year
- Since we can't work directly with `DataFrameGroupBy` objects, we will use aggregation methods to summarize each `DataFrameGroupBy` object into one aggregated row per subframe.



# Aggregation Functions

What goes inside of `.agg( )`?

- Any function that aggregates several values into one summary value
- Common examples:

| In-Built Python Functions | NumPy Functions | In-Built `pandas` functions |
|---|---|---|
| `.agg(sum)` | `.agg(np.sum)` | `.agg("sum")` |
| `.agg(max)` | `.agg(np.max)` | `.agg("max")` |
| `.agg(min)` | `.agg(np.min)` | `.agg("min")` |
| | `.agg(np.mean)` | `.agg("mean")` |
| | | `.agg("first")` |
| | | `.agg("last")` |

Some commonly-used aggregation functions can even be called directly, without the explicit use of `.agg( )`

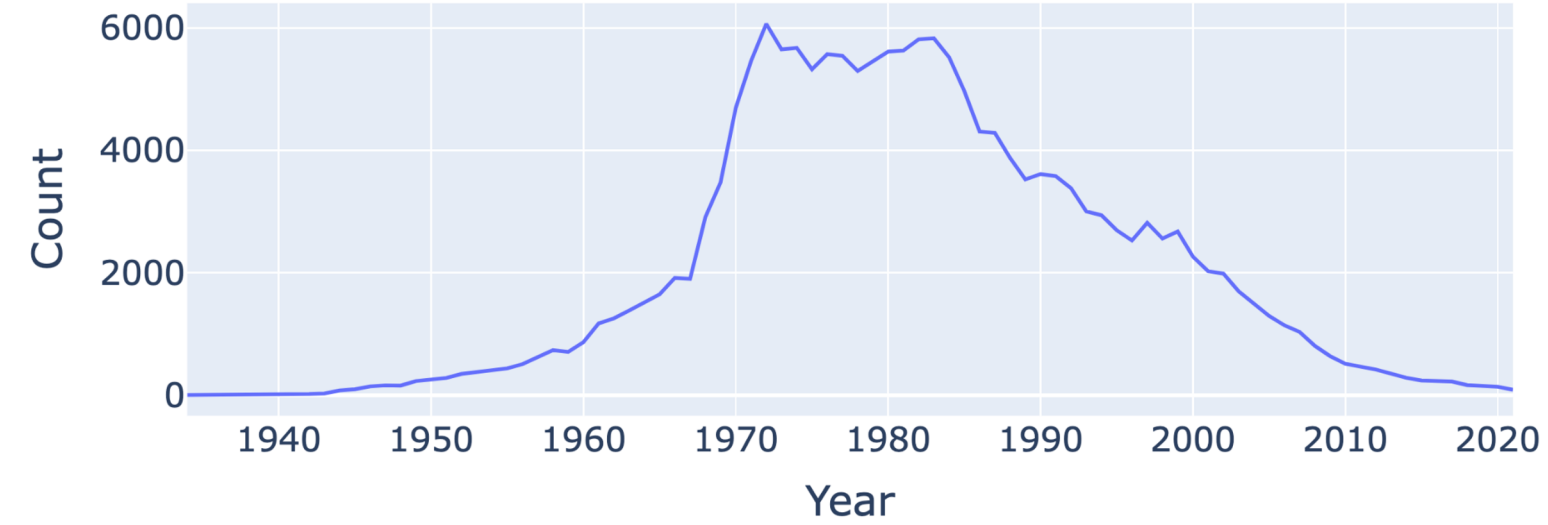
```
babynames.groupby("Year").mean()
```

# Goal

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

```
female_babynames = babynames[babynames["Sex"] == "F"]
female_babynames = female_babynames.sort_values(["Year", "Count"])
jenn_counts_ser = female_babynames[female_babynames["Name"] == "Jennifer"]["Count"]
```

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 babies born with a given name today to the maximum number of the name born in a single year.

Example for "Jennifer":

- In 1972, we hit peak Jennifer. 6,065 Jennifers were born.
- In 2021, there were only 91 Jennifers.
- RTP is 91 / 6065 = 0.015004.

# Calculating RTP

```
max_jenn = max(female_babynames[female_babynames["Name"] == "Jennifer"]["Count"])
```

```
6065
```

```
curr_jenn = female_babynames[female_babynames["Name"] == "Jennifer"]["Count"].iloc[-1]
```

```
91
```

```
rtp = curr_jenn / max_jenn
```

```
0.015004122011541632
```

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

```
jenn_counts_ser = female_babynames[female_babynames["Name"] == "Jennifer"]["Count"]
ratio_to_peak(jenn_counts_ser)
```

```
0.015004122011541632
```

# 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 0.01500412201154163 2
rtps = {}
for name in female_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.

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

| Name | Year | Count |
|---|---|---|
| Aadhira | 1.0 | 0.700000 |
| Aadhya | 1.0 | 0.580000 |
| Aadya | 1.0 | 0.724138 |
| Aahana | 1.0 | 0.192308 |
| Aahna | 1.0 | 1.000000 |
| ... | ... | ... |
| Zyanya | 1.0 | 0.857143 |
| Zyla | 1.0 | 1.000000 |
| Zylah | 1.0 | 1.000000 |
| Zyra | 1.0 | 1.000000 |

# 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!!]

```
# Build dictionary where each entry is the rtp for a given name
# e.g. rtps["jennifer"] should be 0.015004122011541632
rtps = {}
for name in female_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)
```

Approach 2:

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

# Question

Approach 2 generated two columns, Year and Count.

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

| Name | Year | Count |
|---|---|---|
| Aadhira | 1.0 | 0.700000 |
| Aadhya | 1.0 | 0.580000 |
| Aadya | 1.0 | 0.724138 |
| Aahana | 1.0 | 0.192308 |
| Aahna | 1.0 | 1.000000 |
| ... | ... | ... |
| Zyanya | 1.0 | 0.857143 |
| Zyla | 1.0 | 1.000000 |
| Zylah | 1.0 | 1.000000 |
| Zyra | 1.0 | 1.000000 |

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 2021.
B. Yes, names that did not appear in 2021.
C. Yes, names whose peak Count was in 2021.
D. No, every row has a Year value of 1.0.

# Answer

Approach 2 generated two columns, Year and Count.

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

| Name | Year | Count |
|---|---|---|
| Aadhira | 1.0 | 0.700000 |
| Aadhya | 1.0 | 0.580000 |
| Aadya | 1.0 | 0.724138 |
| Aahana | 1.0 | 0.192308 |
| Aahna | 1.0 | 1.000000 |
| ... | ... | ... |
| Zyanya | 1.0 | 0.857143 |
| Zyla | 1.0 | 1.000000 |
| Zylah | 1.0 | 1.000000 |
| Zyra | 1.0 | 1.000000 |

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 2021.
B. Yes, names that did not appear in 2021.
C. Yes, names whose peak Count was in 2021.
**D. No, every row has a Year value of 1.0.**

# Note on Nuisance Columns

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 |
|---|---|---|---|---|---|
| **1** | CA | F | 1910 | Helen | 239 |
| **83** | CA | F | 1910 | Nellie | 20 |
| **184** | CA | F | 1910 | Aileen | 6 |
| **186** | CA | F | 1910 | Astrid | 6 |
| **187** | CA | F | 1910 | Beulah | 6 |
| **...** | ... | ... | ... | ... | ... |
| **235340** | CA | F | 2021 | An | 5 |
| **232380** | CA | F | 2021 | Adelyn | 144 |
| **232496** | CA | F | 2021 | Ailani | 99 |
| **232501** | CA | F | 2021 | Dayana | 98 |
| **235759** | CA | F | 2021 | Vy | 5 |

235791 rows × 5 columns

| Name | Year | Count |
|---|---|---|
| **Aadhira** | 1.0 | 0.700000 |
| **Aadhya** | 1.0 | 0.580000 |
| **Aadya** | 1.0 | 0.724138 |
| **Aahana** | 1.0 | 0.192308 |
| **Aahna** | 1.0 | 1.000000 |
| **...** | ... | ... |
| **Zyanya** | 1.0 | 0.857143 |
| **Zyla** | 1.0 | 1.000000 |
| **Zylah** | 1.0 | 1.000000 |
| **Zyra** | 1.0 | 1.000000 |

For more details, see Pandas 1.3 release notes.

# Note on Nuisance Columns

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, 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)
```

| Name | Count |
|---|---|
| Aadhira | 0.700000 |
| Aadhya | 0.580000 |
| Aadya | 0.724138 |
| Aahana | 0.192308 |
| Aahna | 1.000000 |
| ... | ... |
| Zyanya | 0.857143 |
| Zyla | 1.000000 |
| Zylah | 1.000000 |
| Zyra | 1.000000 |
| Zyrah | 0.833333 |

13661 rows × 1 columns

# 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 | 0.700000 |
| Aadhya | 0.580000 |
| Aadya | 0.724138 |
| Aahana | 0.192308 |
| Aahna | 1.000000 |
| ... | ... |
| Zyanya | 0.857143 |
| Zyla | 1.000000 |
| Zylah | 1.000000 |
| Zyra | 1.000000 |
| Zyrah | 0.833333 |

13661 rows × 1 columns

| Name | Count RTP |
|---|---|
| Aadhira | 0.700000 |
| Aadhya | 0.580000 |
| Aadya | 0.724138 |
| Aahana | 0.192308 |
| Aahna | 1.000000 |
| ... | ... |
| Zyanya | 0.857143 |
| Zyla | 1.000000 |
| Zylah | 1.000000 |
| Zyra | 1.000000 |
| Zyrah | 0.833333 |

13661 rows × 1 columns

# 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")
```

| Name | Count RTP |
|---|---|
| Debra | 0.001260 |
| Susan | 0.002034 |
| Debbie | 0.002817 |
| Cheryl | 0.003273 |
| Carol | 0.003635 |
| ... | ... |
| Jovi | 1.000000 |
| Neta | 1.000000 |
| Doni | 1.000000 |
| Dondi | 1.000000 |
| Kela | 1.000000 |

13661 rows × 1 columns

# Some Data Science Payoff

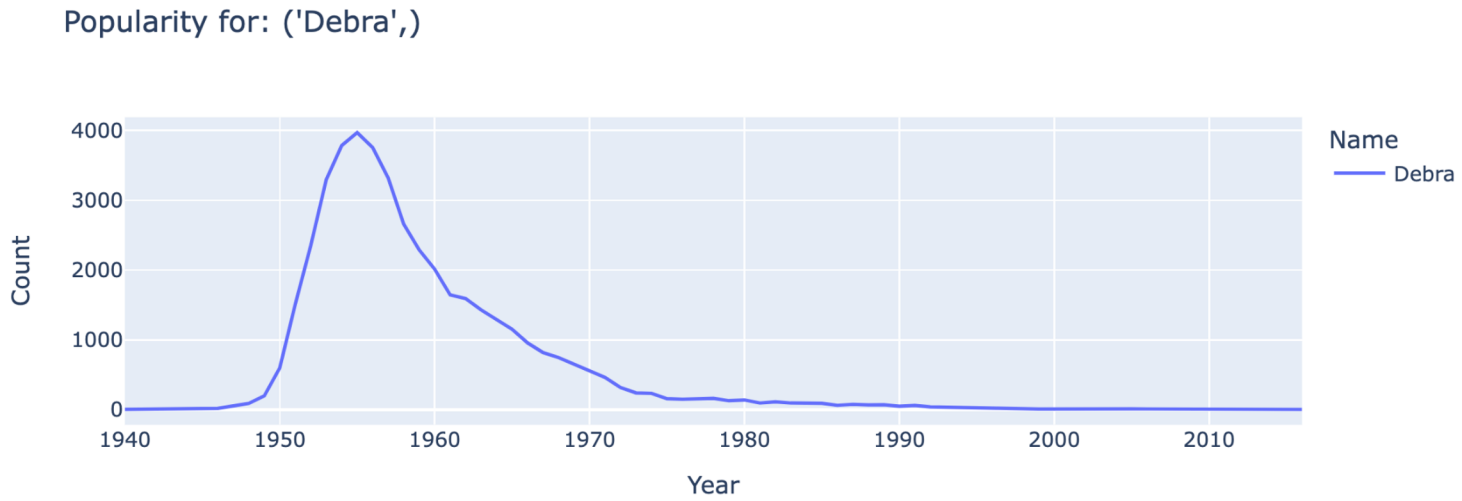
By sorting `rtp_table` we can see the names whose popularity has decreased the most.

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

| Name | Count RTP |
|---|---|
| Debra | 0.001260 |
| Susan | 0.002034 |
| Debbie | 0.002817 |
| Cheryl | 0.003273 |
| Carol | 0.003635 |
| ... | ... |
| Jovi | 1.000000 |
| Neta | 1.000000 |
| Doni | 1.000000 |
| Dondi | 1.000000 |
| Kela | 1.000000 |

13661 rows × 1 columns

```
px.line(female_babynames[female_babyname["Name"] == "Debra")],
                        x = "Year", y = "Count")
```



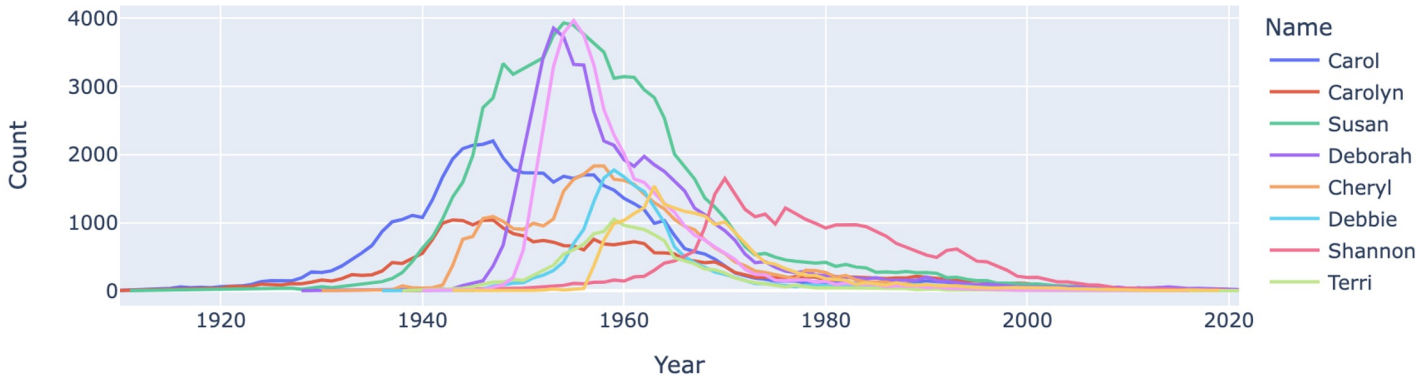
# Some Data Science Payoff

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', 'Susan', 'Debbie', 'Cheryl', 'Carol', 'Tammy', 'Terri',
       'Shannon', 'Deborah', 'Carolyn'],
      dtype='object', name='Name')
```

```
px.line(female_babynames[female_babyname["Name"].isin(top10)],
                         x = "Year", y = "Count", color = "Name")
```



# Some groupby.agg Puzzles

- Conditional Selection
- Handy Utility Functions
- Adding, Modifying, and Removing Columns
- Groupby.agg
- **Some groupby.agg Puzzles**
- 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)
```

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

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

| Year | Count |
|---|---|
| 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)
```

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

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

| Year | Count |
|---|---|
| 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:

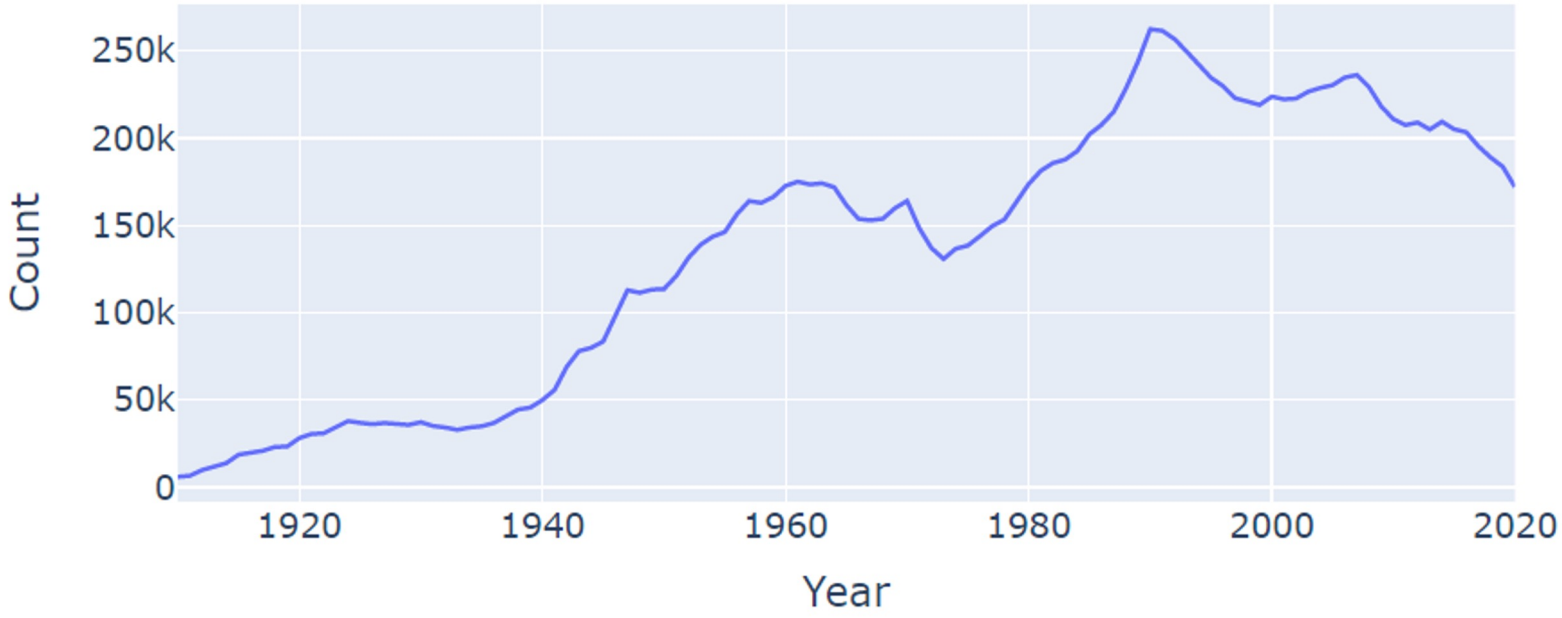
```
female_babynames.groupby("Name")[["Count"]].sum()
```

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

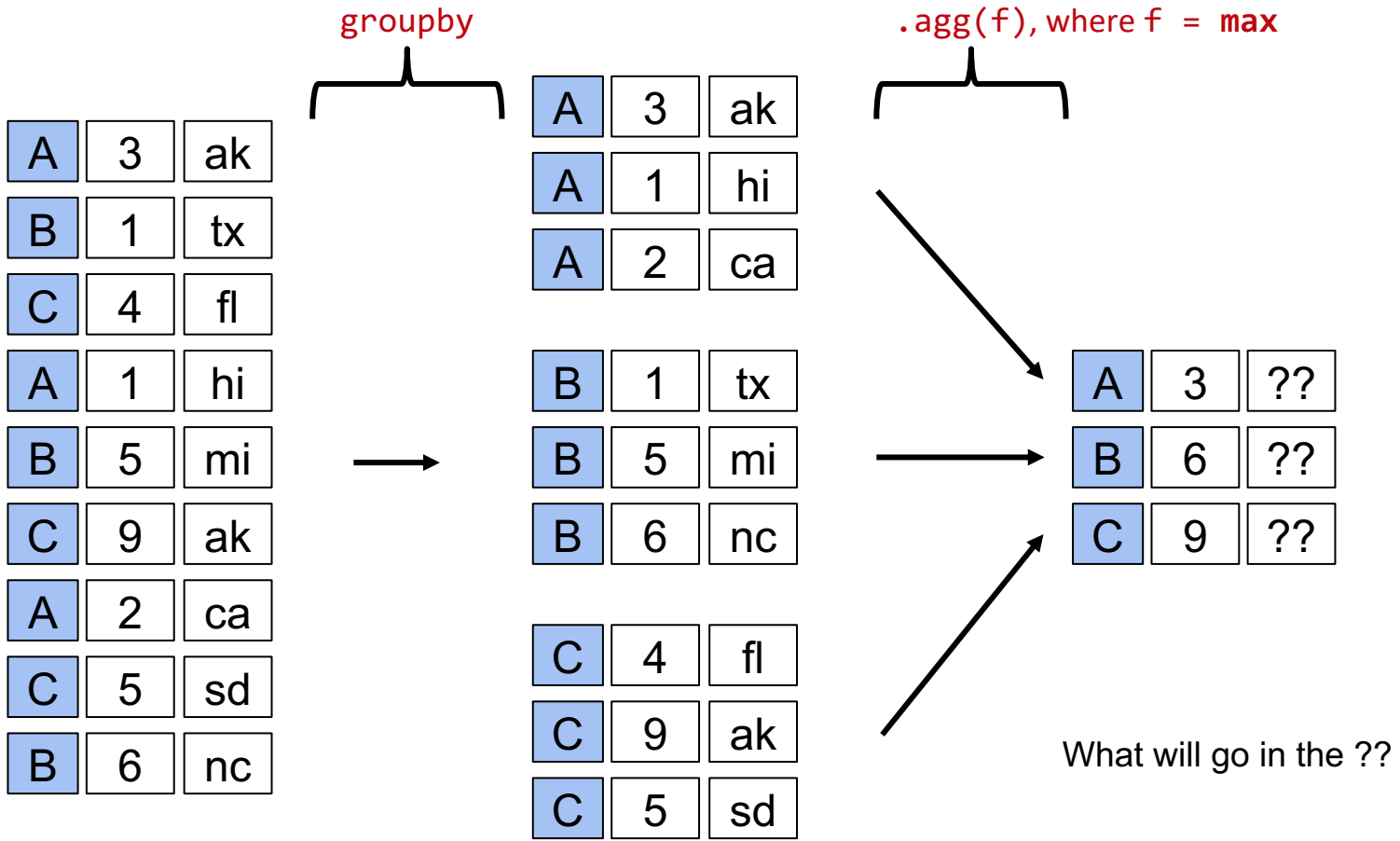
# Plotting Birth Counts

Plotting the DataFrame we just generated tells an interesting story.

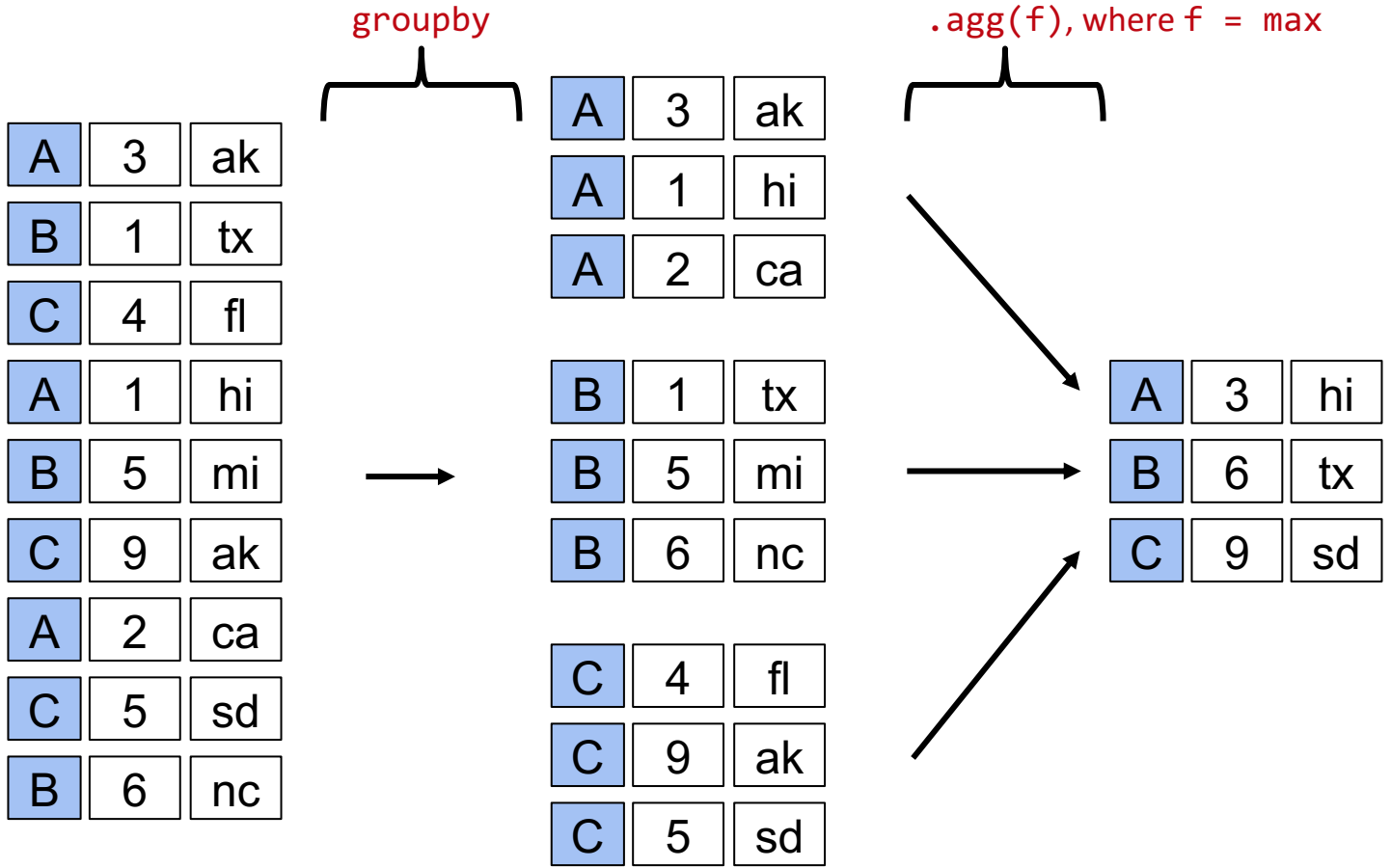
```
puzzle2 = female_babynames.groupby("Year")[["Count"]].agg(sum)
px.line(puzzle2, y = "Count")
```



# Puzzle #3


groupby
.agg(f), where f = max
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
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
A 3 ??
B 6 ??
C 9 ??
What will go in the ??

# Puzzle #3


groupby
.agg(f), where f = max
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
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
A 3 hi
B 6 tx
C 9 sd

# Puzzle #4

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.

| 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 |

# Puzzle #4

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.

| 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 |

# Puzzle #4

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.

```
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 |

# 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')
```

# 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!

# Other DataFrameGroupBy Features

- Conditional Selection
- Handy Utility Functions
- Adding, Modifying, and Removing Columns
- Groupby.agg
- Some groupby.agg Puzzles
- **Other DataFrameGroupBy Features**
- Groupby and PivotTables
- A Quick Look at Joining Tables

# Revisiting Raw `GroupBy` Objects and Aggregation Methods

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

- It is not a **`DataFrame`**!

```
grouped_by_year = elections.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:

```
df.groupby(col).mean()
df.groupby(col).sum()
df.groupby(col).min()
df.groupby(col).max()
```

```
df.groupby(col).first()
df.groupby(col).last()
```

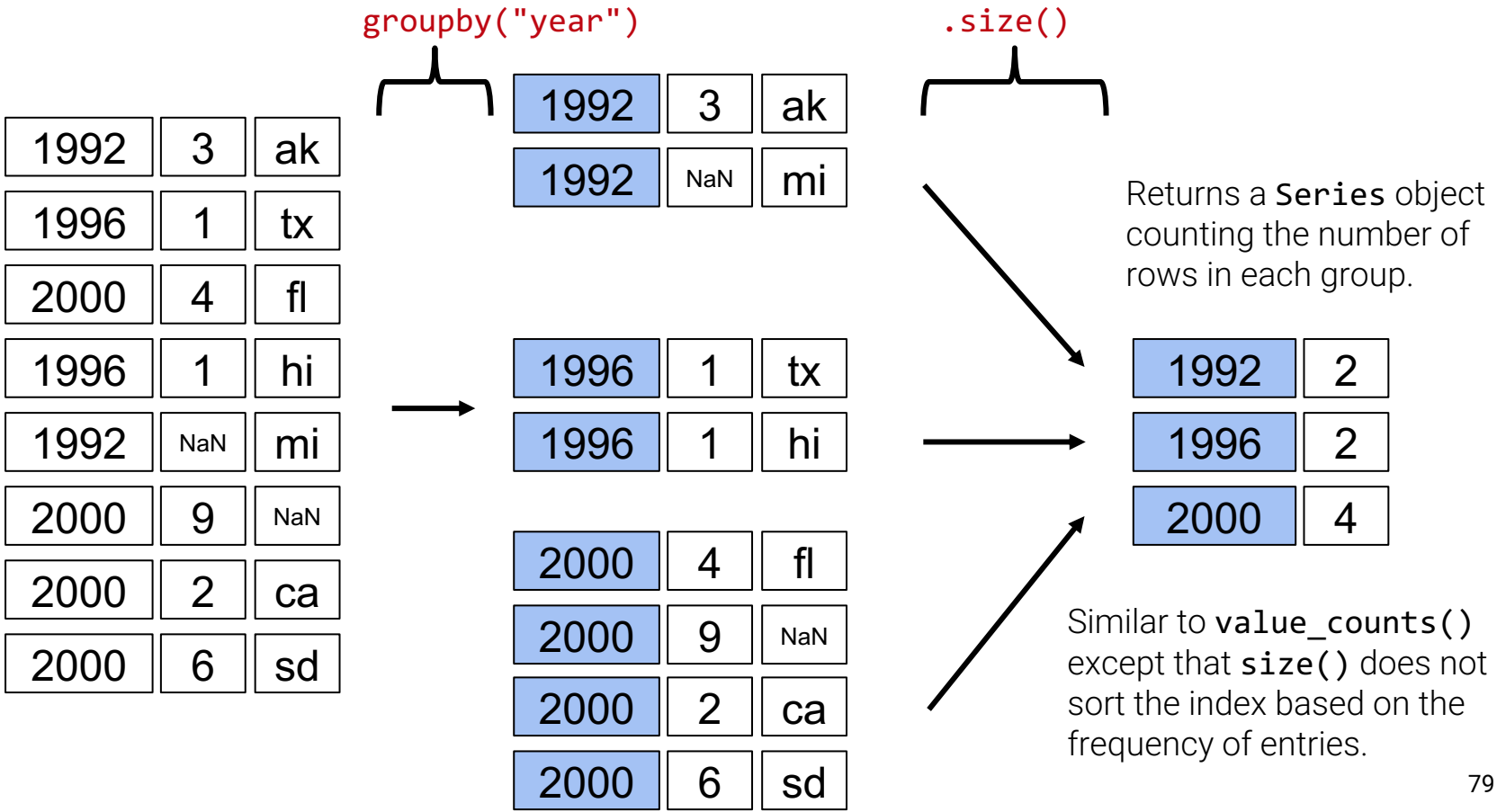
```
df.groupby(col).size()
df.groupby(col).count()
```

```
df.groupby(col).filter()
```

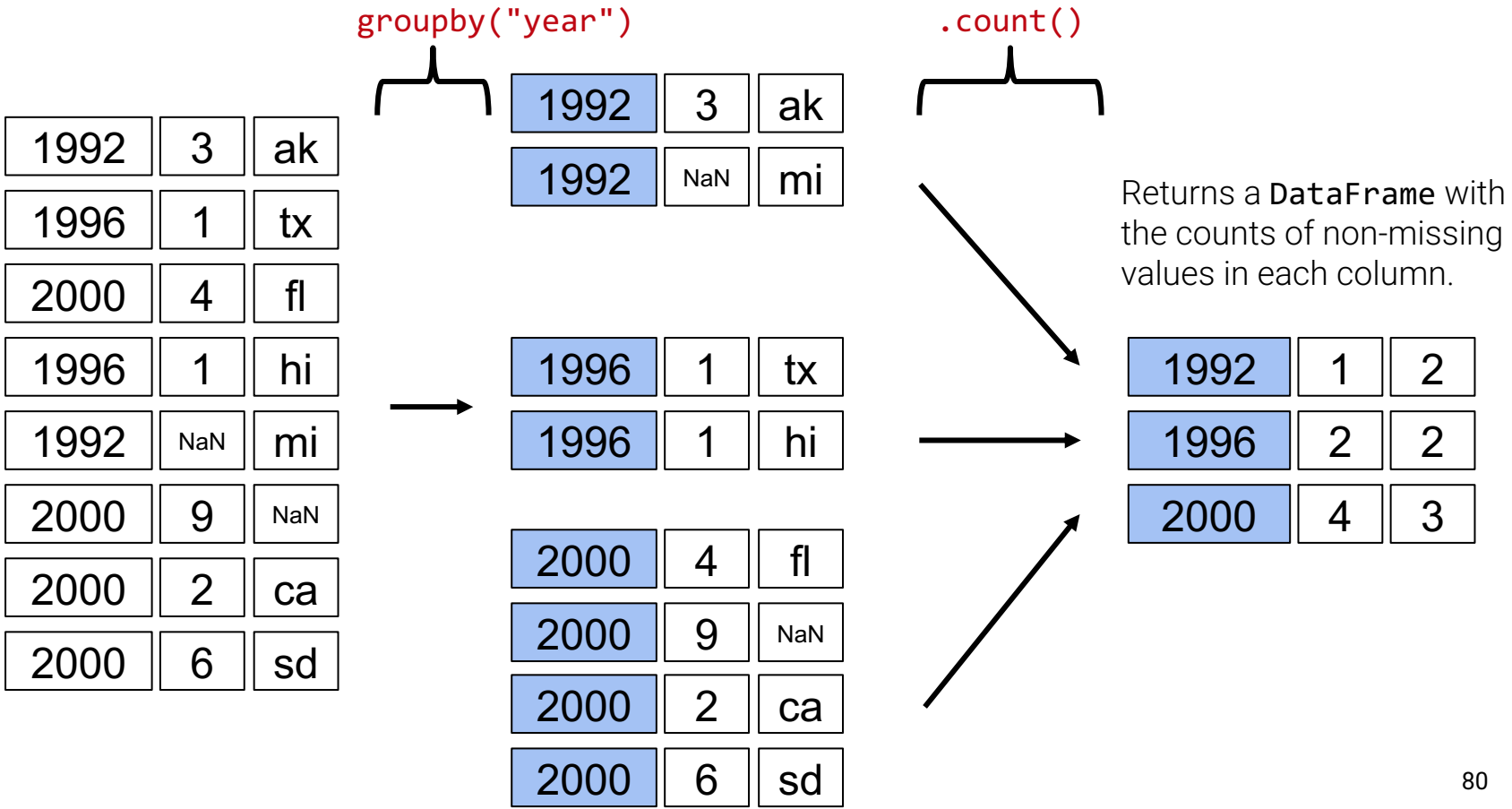
😐What's the difference?

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

# groupby.size() and groupby.count()



# groupby.size() and groupby.count()

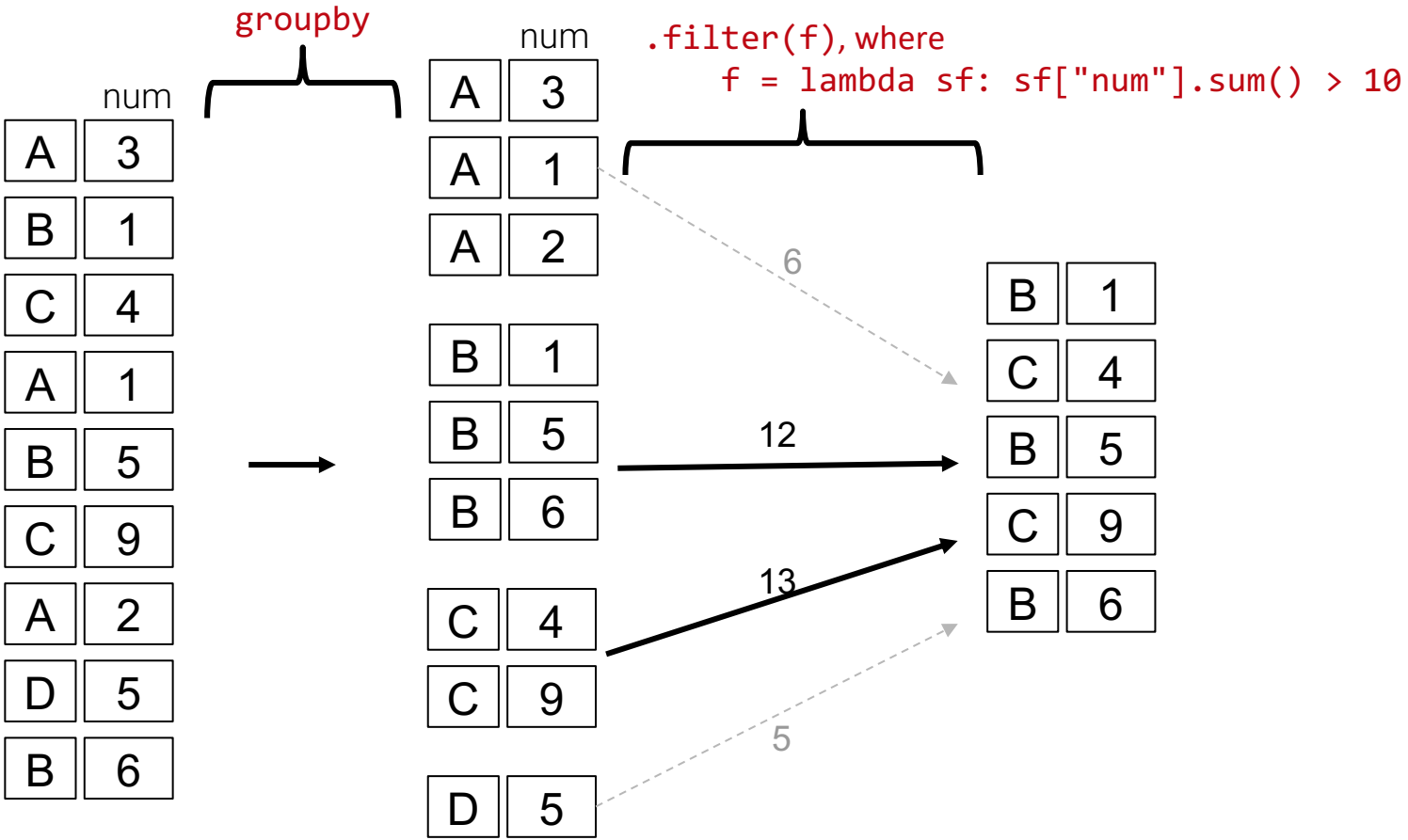


# Filtering by Group

Another common use for groups is to filter data.

- `groupby.filter` takes an argument `func`.
- `func` is a function that:
  - Takes a `DataFrame` as input.
  - Returns either `True` or `False`.
- `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**.
- Notes:
  - Filtering is done per group, not per row. Different from boolean filtering.
  - Unlike `agg()`, the column we grouped on does NOT become the index!

# groupby.filter()



# Filtering Elections Dataset

Going back to the `elections` dataset.

Let's keep only election year results where the max '%' is less than 45%.

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

| | 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 |
| **115** | 1968 | George Wallace | American Independent | 9901118 | loss | 13.571218 |

**Puzzle**: We want to know the **best election by each party**.

- Best election: The election with the highest % of votes.
- For example, Democrat's best election was in 1964, with candidate Lyndon Johnson winning 61.3% of votes.

| 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 |

# groupby Puzzle - Possible Approaches

Using a `lambda` function

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

Using `idxmax` function

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

Using `drop_duplicates` function

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

# 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!

# More on `DataFrameGroupby` Object

We can look into **`DataFrameGroupby`** objects in following ways:

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

```
{'American': [22, 126], 'American Independent': [115, 119, 124], 'Anti-Masonic': [6], 'Anti-Monopoly': [38], 'Citiz
ens': [127], 'Communist': [89], 'Constitution': [160, 164, 172], 'Constitutional Union': [24], 'Democratic': [2, 4,
8, 10, 13, 14, 17, 20, 28, 29, 34, 37, 39, 45, 47, 52, 55, 57, 64, 70, 74, 77, 81, 83, 86, 91, 94, 97, 100, 105, 10
8, 111, 114, 116, 118, 123, 129, 134, 137, 140, 144, 151, 158, 162, 168, 176, 178], 'Democratic-Republican': [0,
1], 'Dixiecrat': [103], 'Farmer-Labor': [78], 'Free Soil': [15, 18], 'Green': [149, 155, 156, 165, 170, 177, 181],
'Greenback': [35], 'Independent': [121, 130, 143, 161, 167, 174], 'Liberal Republican': [31], 'Libertarian': [125,
128, 132, 138, 139, 146, 153, 159, 163, 169, 175, 180], 'National Democratic': [50], 'National Republican': [3, 5],
'National Union': [27], 'Natural Law': [148], 'New Alliance': [136], 'Northern Democratic': [26], 'Populist': [48,
61, 141], 'Progressive': [68, 82, 101, 107], 'Prohibition': [41, 44, 49, 51, 54, 59, 63, 67, 73, 75, 99], 'Reform':
[150, 154], 'Republican': [21, 23, 30, 32, 33, 36, 40, 43, 46, 53, 56, 60, 65, 69, 72, 79, 80, 84, 87, 90, 96, 98,
104, 106, 109, 112, 113, 117, 120, 122, 131, 133, 135, 142, 145, 152, 157, 166, 171, 173, 179], 'Socialist': [58, 6
2, 66, 71, 76, 85, 88, 92, 95, 102], 'Southern Democratic': [25], 'States' Rights': [110], 'Taxpayers': [147], 'Uni
on': [93], 'Union Labor': [42], 'Whig': [7, 9, 11, 12, 16, 19]}
```

```
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 |

# Groupby and PivotTables

- Conditional Selection
- Handy Utility Functions
- Adding, Modifying, and Removing Columns
- Groupby.agg
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- **Groupby and PivotTables**
- A Quick Look at Joining Tables

# 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)
```

| | | Count |
|---|---|---|
| **Year** | **Sex** | |
| **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 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 |

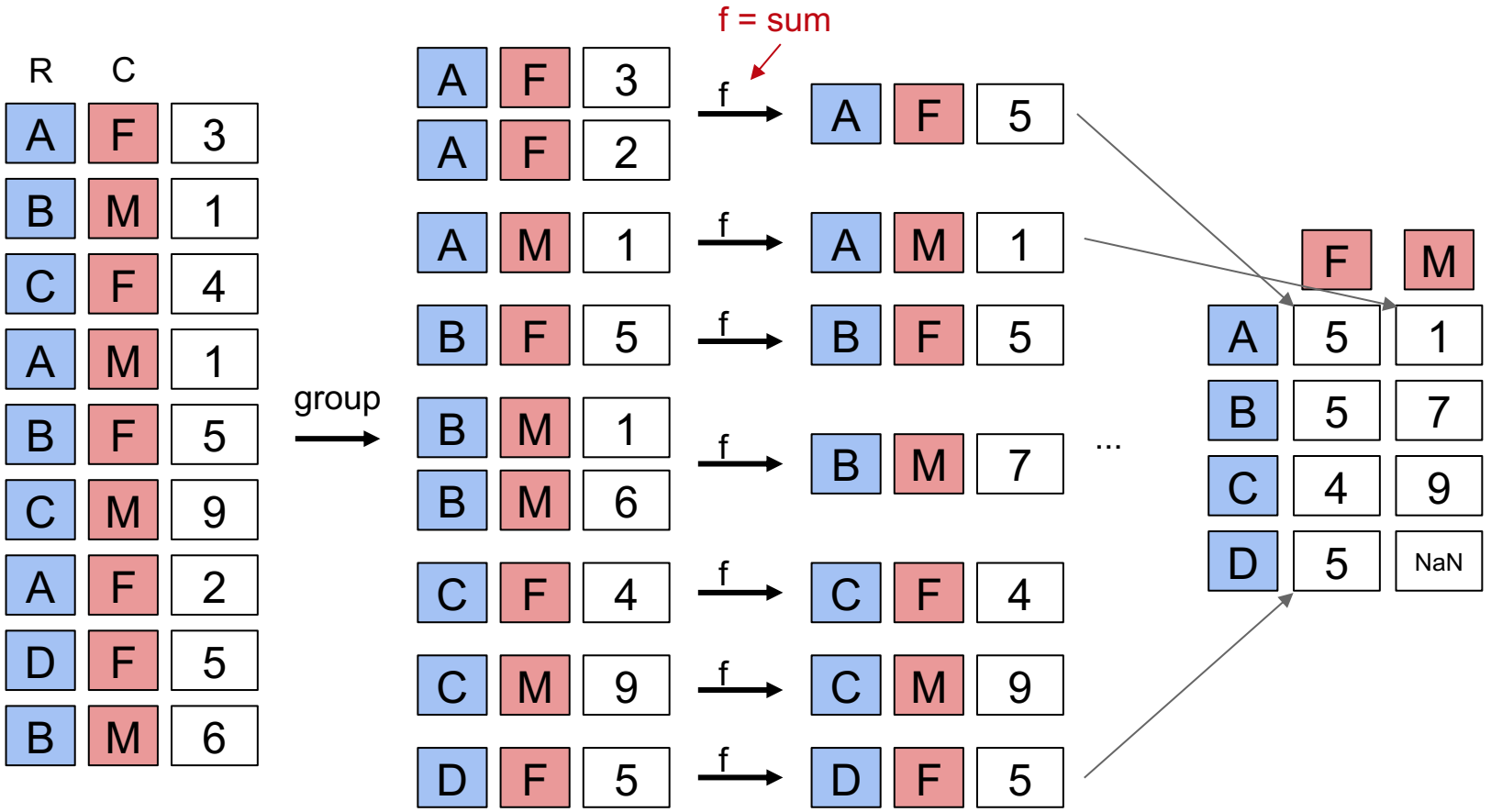
# groupby(["Year", "Sex"]) vs. pivot_table

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

- Conditional Selection
- Handy Utility Functions
- Adding, Modifying, and Removing Columns
- Groupby.agg
- Some groupby.agg Puzzles
- Other DataFrameGroupBy Features
- Groupby and PivotTables
- **A Quick Look at Joining Tables**

# 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.

```
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 |

# 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.