

LECTURE 5

# Pandas, Part III

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

Data Science, Fall 2023 @ Knowledge Stream

Sana Jabbar

# Today's Roadmap

---

Lecture 5

- Pandas, Part III
  - Groupby Review
  - More on Groupby
  - Pivot Tables
  - Joining Tables
- EDA, Part I
  - Structure: Tabular Data
  - Granularity
  - Structure: Variable Types

# Groupby Review

---

Lecture 5

- **Pandas, Part III**
  - **Groupby Review**
  - More on Groupby
  - Pivot Tables
  - Joining Tables
- EDA, Part I
  - Structure: Tabular Data
  - Granularity
  - Structure: Variable Types

## Revisiting groupby.agg

`dataframe.groupby(column_name).agg(aggregation_function)`

`babynames.groupby("Year")[["Count"]].agg(sum)` computes the total number of babies born in each year.

CA	F	1910	Mary	295
CA	M	2005	Zain	20
CA	F	2015	Luisa	40
CA	M	2005	Alijah	37
CA	M	2015	Jorge	460
CA	F	1910	Ann	47

.groupby("Year")  
→

CA	F	1910	Mary	295
CA	F	1910	Ann	47
CA	M	2005	Zain	20
CA	M	2005	Alijah	37
CA	F	2015	Luisa	40
CA	M	2015	Jorge	460

.agg(sum)  
→

1910	342
2005	57
2015	500

Original DataFrame

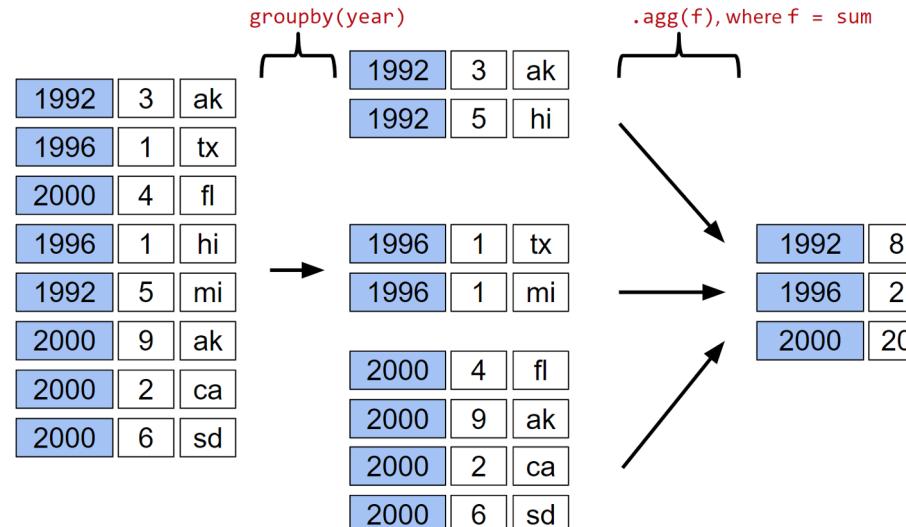
GroupBy Object

Output DataFrame

## Revisiting groupby .agg

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

- So far, we've seen that `df.groupby("year").agg(sum)`:
  - **Split** df into sub-DataFrames based on `year`.
  - **Apply** the `sum` function to each column of each sub-DataFrame.
  - **Combine** the results of `sum` into a single DataFrame, indexed by `year`.



## Aggregation Functions

---

What goes inside of `.agg( )`?

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

In-Built Python  
Functions

`.agg(sum)`  
`.agg(max)`  
`.agg(min)`

NumPy  
Functions

`.agg(np.sum)`  
`.agg(np.max)`  
`.agg(np.min)`  
`.agg(np.mean)`

In-Built pandas  
functions

`.agg("sum")`  
`.agg("max")`  
`.agg("min")`  
`.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()
```

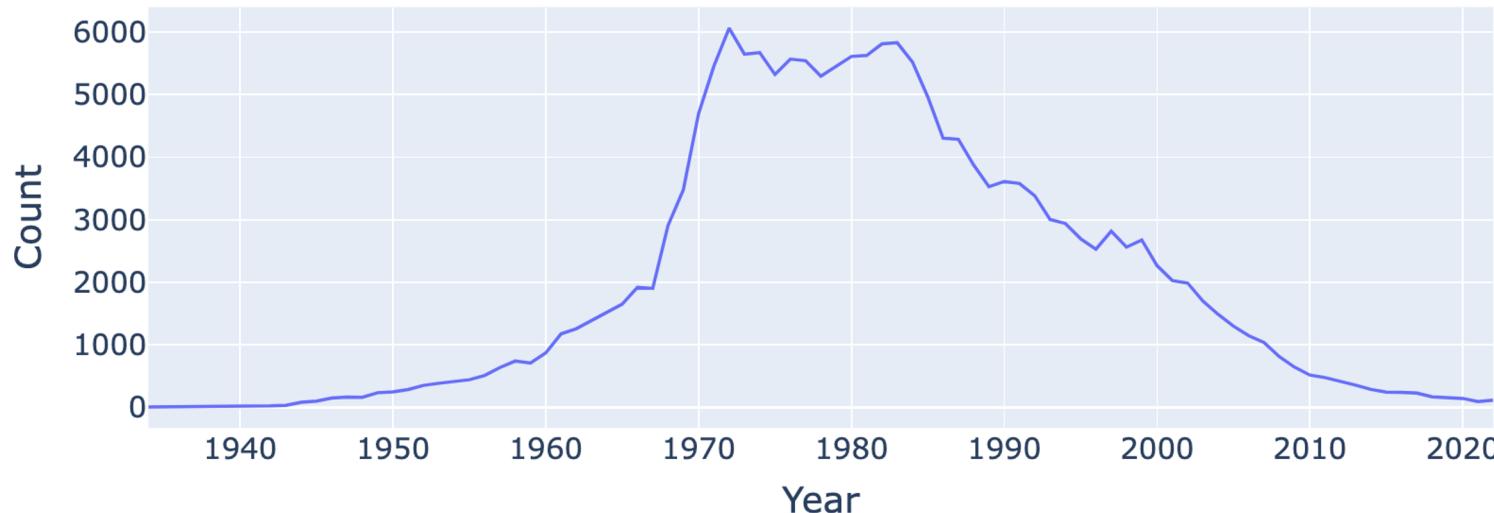
## Putting Things Into Practice

---

Goal: Find the baby name with sex "F" that has fallen in popularity the most in California.

```
f_babynames = babynames[babynames["Sex"] == "F"]  
f_babynames = f_babynames.sort_values(["Year"])  
jenn_counts_series = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"]
```

Number of Jennifers Born in California Per Year.



## What Is "Popularity"?

---

Goal: Find the baby name with sex "F" that has fallen in popularity the most in California.

How do we define "fallen in popularity"?

- Let's create a metric: "Ratio to Peak" (RTP).
- The RTP is the ratio of babies born with a given name in 2022 to the *maximum* number of babies born with that name in *any* year.

Example for "Jennifer":

- In 1972, we hit peak Jennifer. 6,065 Jennifers were born.
- In 2022, there were only 114 Jennifers.
- RTP is  $114 / 6065 = 0.018796372629843364$ .

## Calculating RTP

---

```
max_jenn = max(f_babynames[f_babynames["Name"] == "Jennifer"]["Count"])
6065
```

```
curr_jenn = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"].iloc[-1]
114
```

```
rtp = curr_jenn / max_jenn
0.018796372629843364
```

Remember: `f_babynames` is sorted by year.  
`.iloc[-1]` means “grab the latest year”



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

```
jenn_counts_ser = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"]
ratio_to_peak(jenn_counts_ser)
0.018796372629843364
```

## Renaming Columns After Grouping

By default, `.groupby` will not rename any aggregated columns (the column is still named "Count", even though it now represents the RTP).

For better readability, we may wish to rename "Count" to "Count RTP"

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

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

Count		Count RTP	
Name		Name	
Aadhini	1.000000	Aadhini	1.000000
Aadhira	0.500000	Aadhira	0.500000
Aadhyा	0.660000	Aadhyा	0.660000
Aadya	0.586207	Aadya	0.586207
Aahana	0.269231	Aahana	0.269231
...	...	...	...

## 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	
<b>Debra</b>	0.001260
<b>Debbie</b>	0.002815
<b>Carol</b>	0.003180
<b>Tammy</b>	0.003249
<b>Susan</b>	0.003305
...	...
<b>Fidelia</b>	1.000000
<b>Naveyah</b>	1.000000
<b>Finlee</b>	1.000000
<b>Roseline</b>	1.000000
<b>Aadhini</b>	1.000000

13782 rows x 1 columns

# Some Data Science Payoff

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

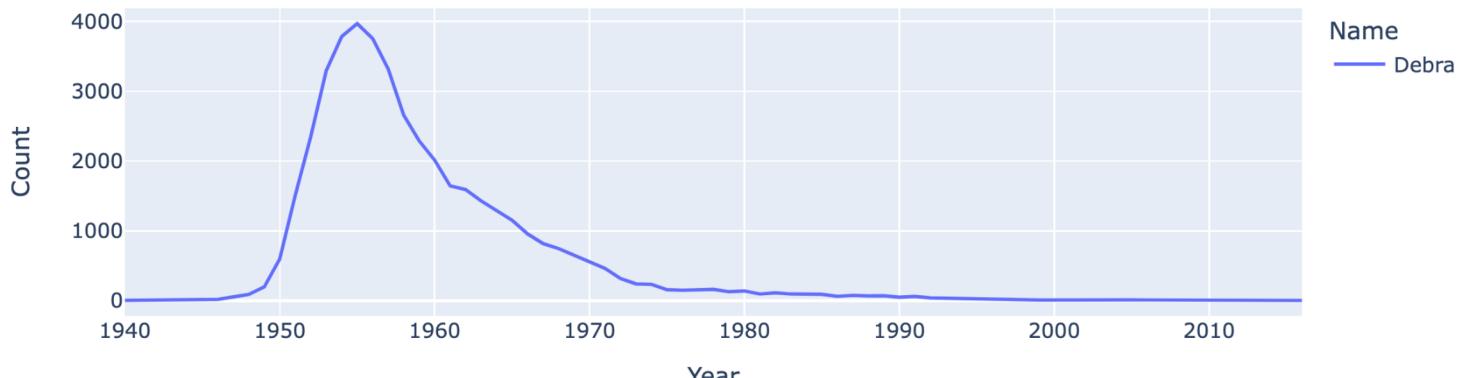
```
rtp_table.sort_values("Count RTP")
```

	Count RTP
Name	
<b>Debra</b>	0.001260
<b>Debbie</b>	0.002815
<b>Carol</b>	0.003180
<b>Tammy</b>	0.003249
<b>Susan</b>	0.003305
...	...
<b>Fidelia</b>	1.000000
<b>Naveyah</b>	1.000000
<b>Finlee</b>	1.000000
<b>Roseline</b>	1.000000
<b>Aadhini</b>	1.000000

13782 rows x 1 columns

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

Popularity for: ('Debra',)

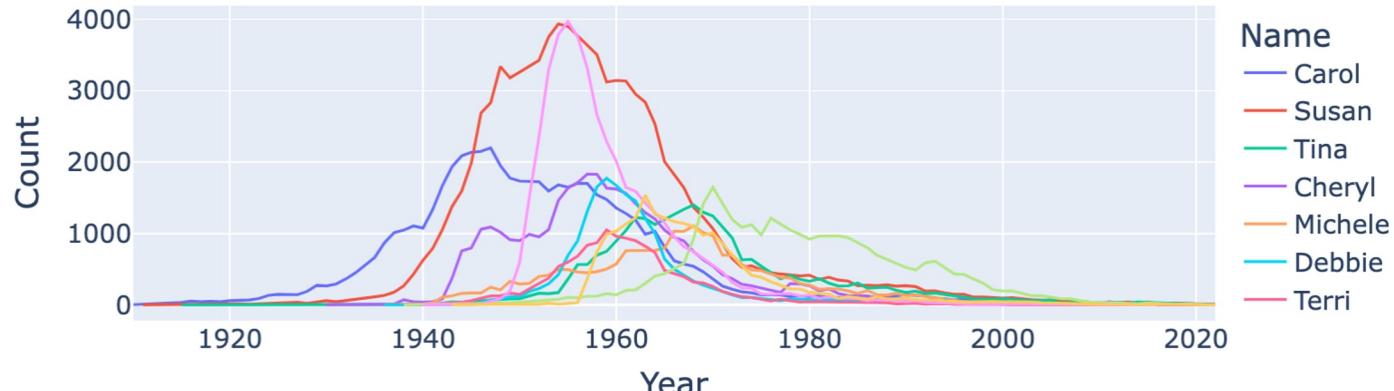


We'll learn about plotting in week 4.

## 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', 'Debbie', 'Carol', 'Tammy', 'Susan', 'Cheryl', 'Shannon',  
       'Tina', 'Michele', 'Terri'],  
      dtype='object', name='Name')  
  
px.line(f_babynames[f_babynames["Name"].isin(top10)],  
        x = "Year", y = "Count", color = "Name")
```



## Answer

---

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

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

We use similar logic to compute the summed counts of all baby names.

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

or

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

Name	Count
Aadan	18
Aadarsh	6
Aaden	647
Aadhav	27
Aadhini	6
...	...
Zymir	5
Zyon	133
Zyra	103
Zyrah	21
Zyrus	5

20437 rows × 1 columns

## Answer

---

Now, we create groups for each year.

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

or

```
babynames.groupby("Year")[[ "Count"]].sum()
```

or

```
babynames.groupby("Year").sum(numeric_only=True)
```

Year	Count
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926
...	...
2018	395436
2019	386996
2020	362882
2021	362582
2022	360023

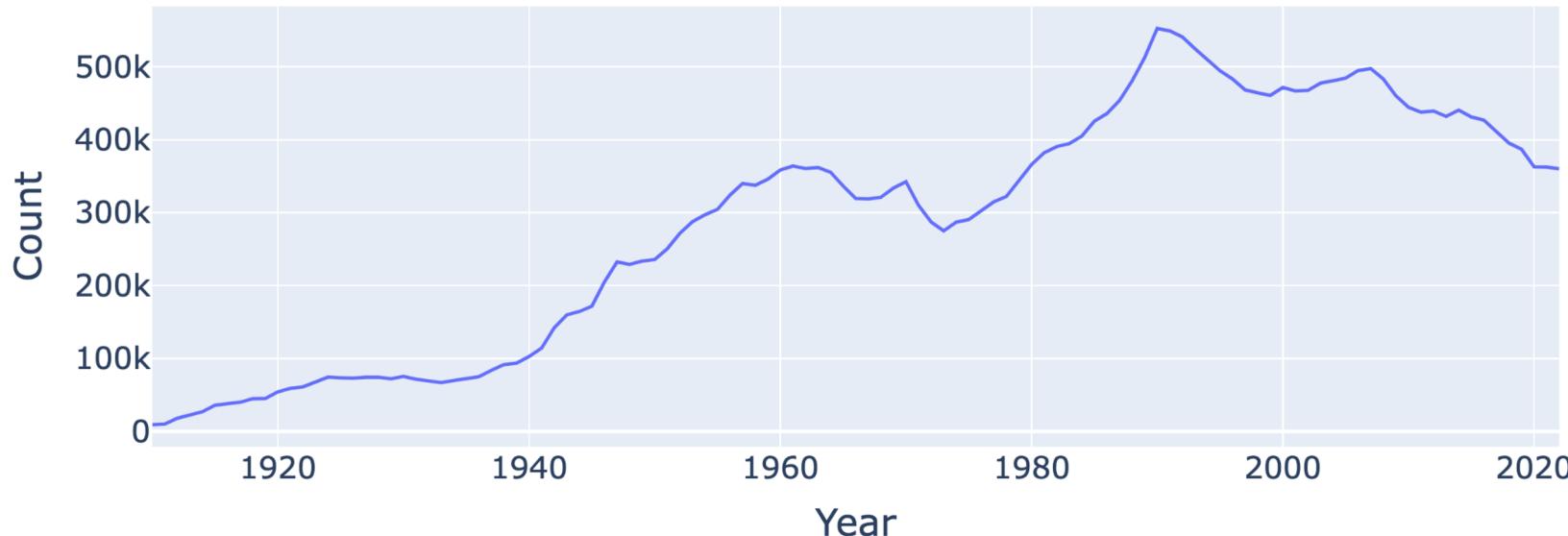
113 rows × 1 columns

## Plotting Birth Counts

---

Plotting the `DataFrame` we just generated tells an interesting story.

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



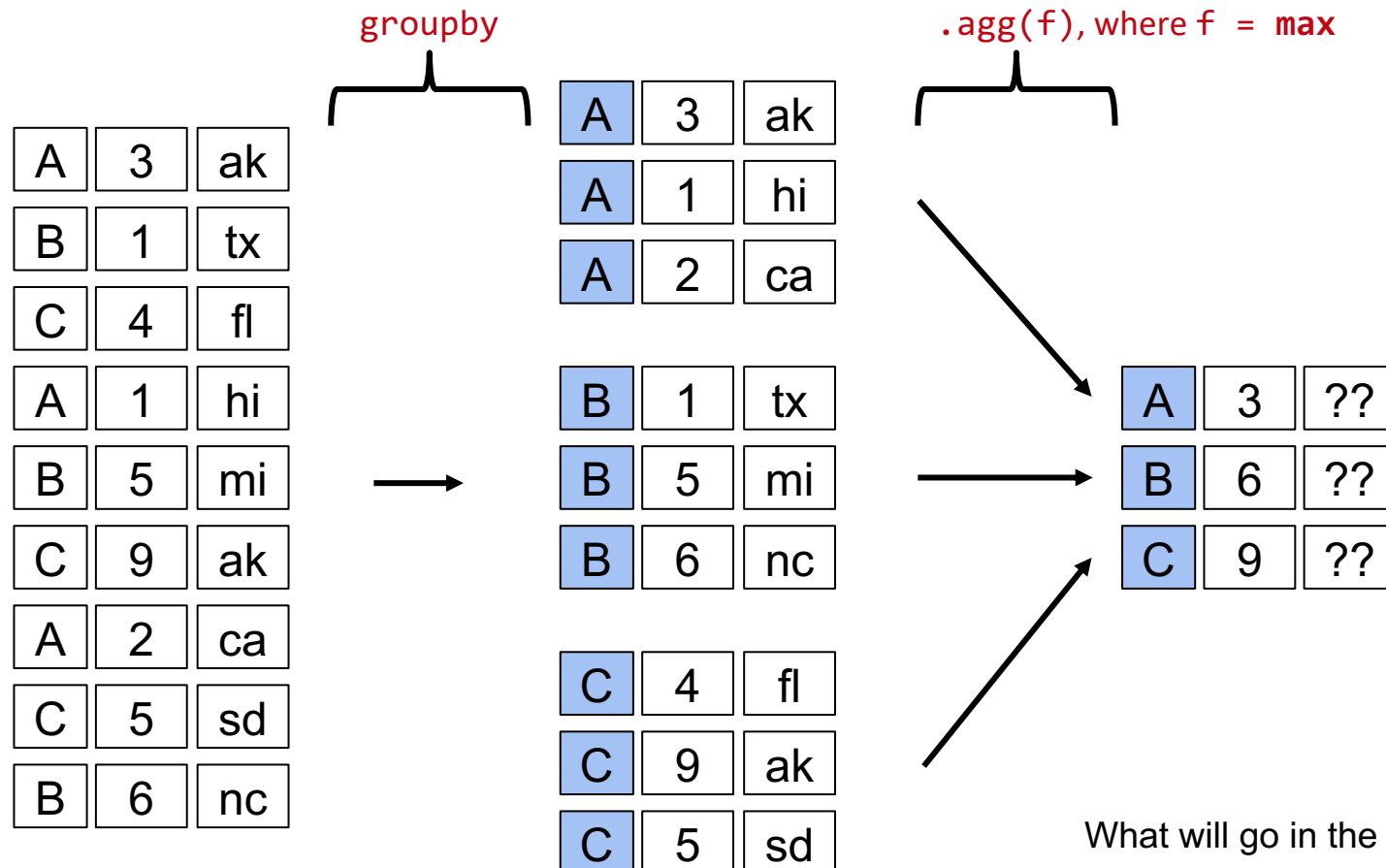
# More on Groupby

---

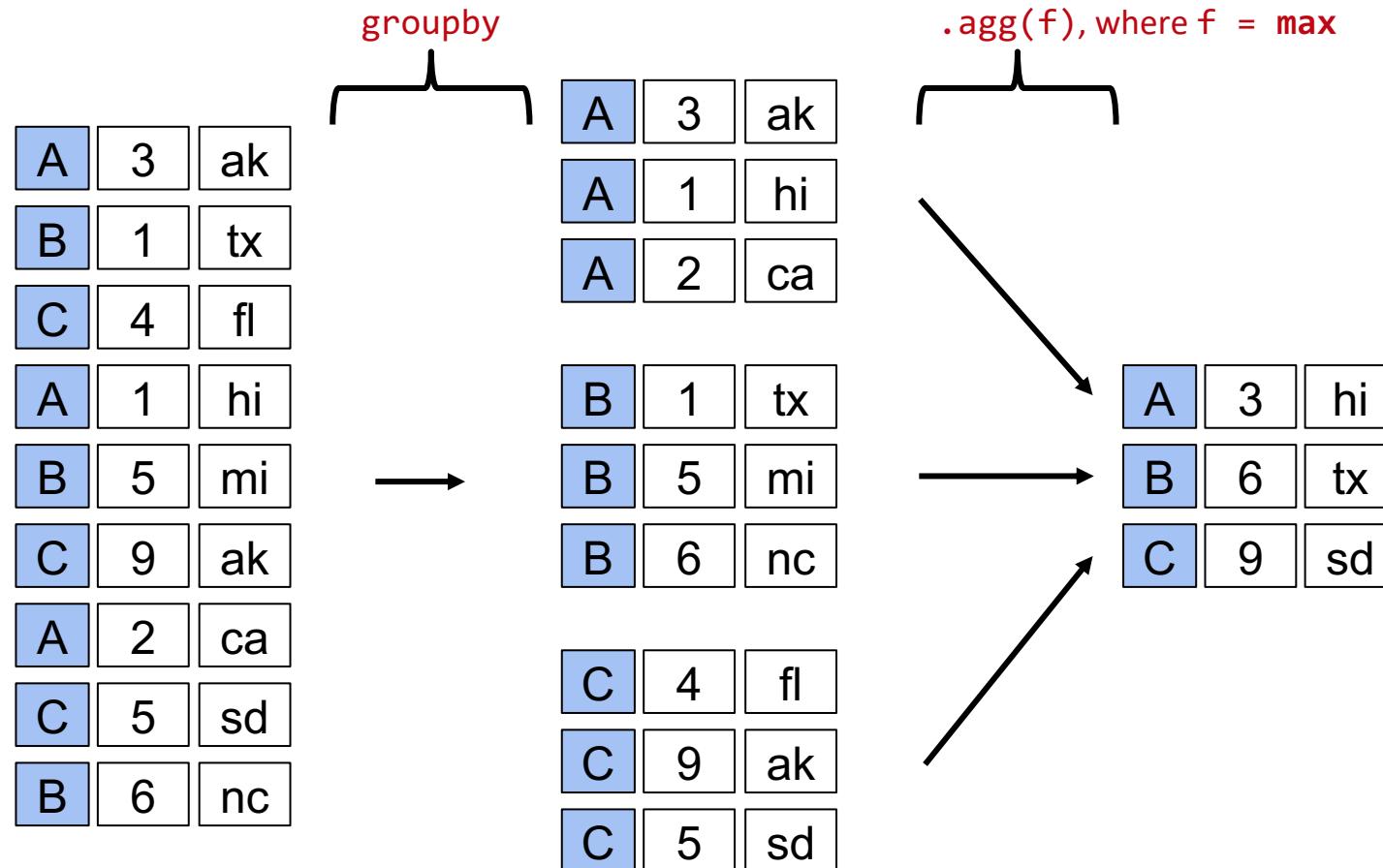
Lecture 5

- **Pandas, Part III**
  - Groupby Review
  - **More on Groupby**
  - Pivot Tables
  - Joining Tables
- EDA, Part I
  - Structure: Tabular Data
  - Granularity
  - Structure: Variable Types

## Groupby Review Question



## Answer



## Raw GroupBy Objects and Other 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).first()
```

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

```
df.groupby(col).sum()
```

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

```
df.groupby(col).min()
```

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

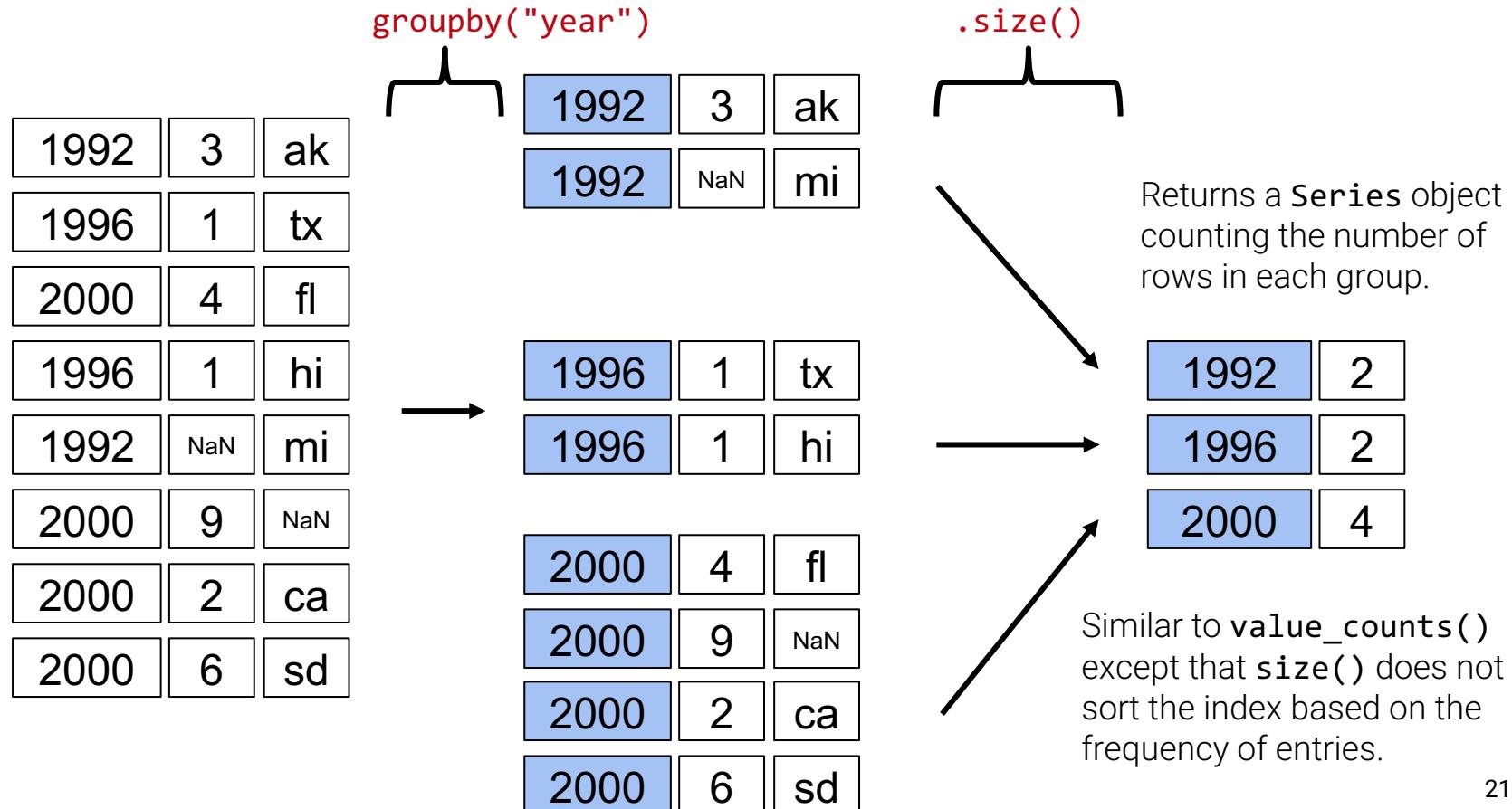
```
df.groupby(col).max()
```

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

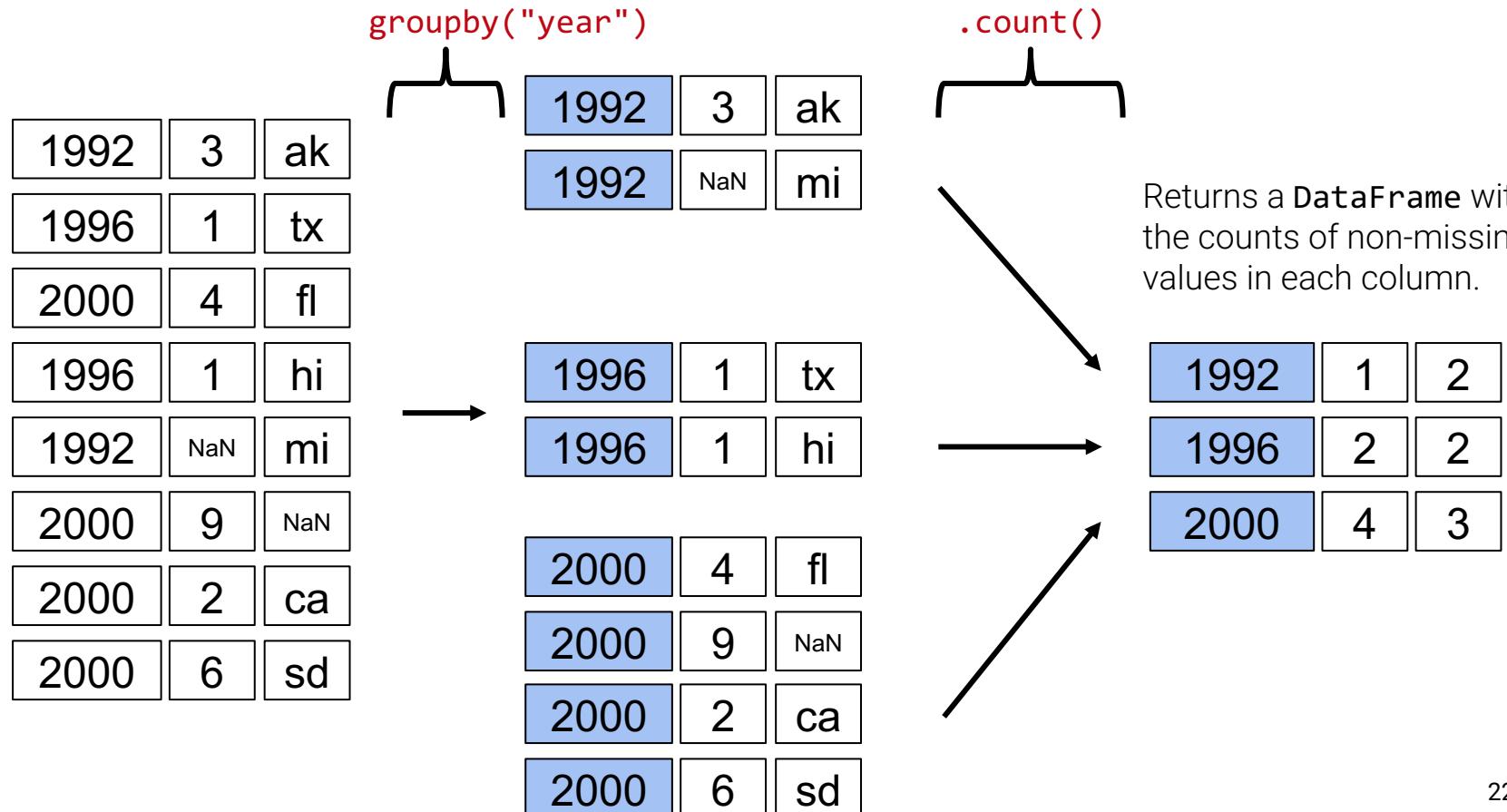
🧐 What's the difference?

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()



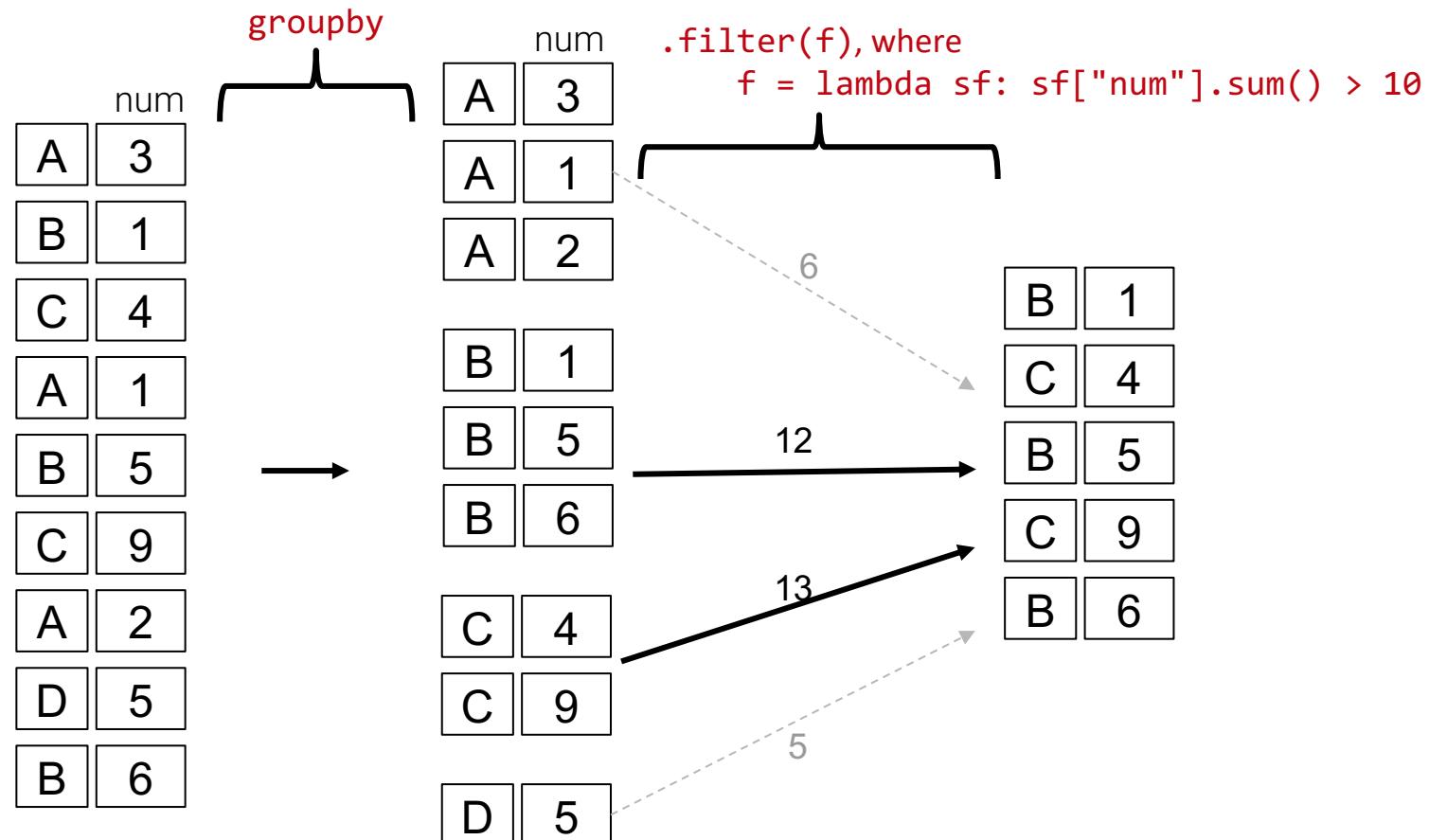
## Filtering by Group

---

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

## Attempt #1

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

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

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

## Problem with Attempt #1

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

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

Every column is calculated independently! Among Democrats:

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

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

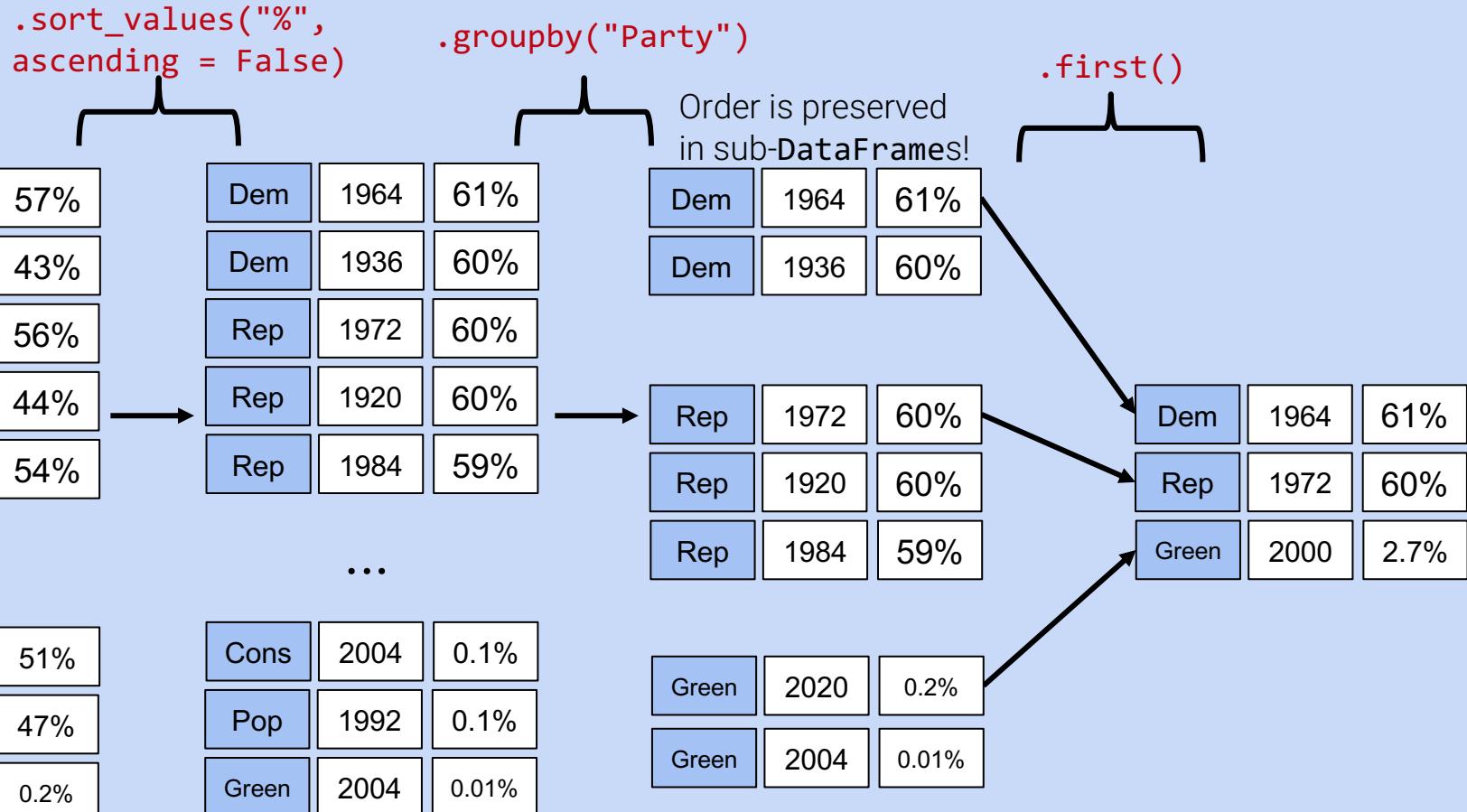
## Attempt #2: Motivation

---

- We want to preserve entire rows, so we need an aggregate function that does that.

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

## Attempt #2: Solution



## Attempt #2: Solution

- First sort the **DataFrame** so that rows are in descending order of %.
- Then group by Party and take the first item of each sub-**DataFrame**.
- 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").first()
```

	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

## groupby Puzzle - Alternate 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], 'Citizens': [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, 108, 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, 62, 66, 71, 76, 85, 88, 92, 95, 102], 'Southern Democratic': [25], 'States' Rights': [110], 'Taxpayers': [147], 'Union': [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

# Pivot Tables

---

Lecture 05

- **Pandas, Part III**
  - Groupby Review
  - More on Groupby
  - **Pivot Tables**
  - Joining Tables
- EDA, Part I
  - Structure: Tabular Data
  - Granularity
  - Structure: Variable Types

## Grouping by Multiple Columns

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

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

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

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.

groupby output

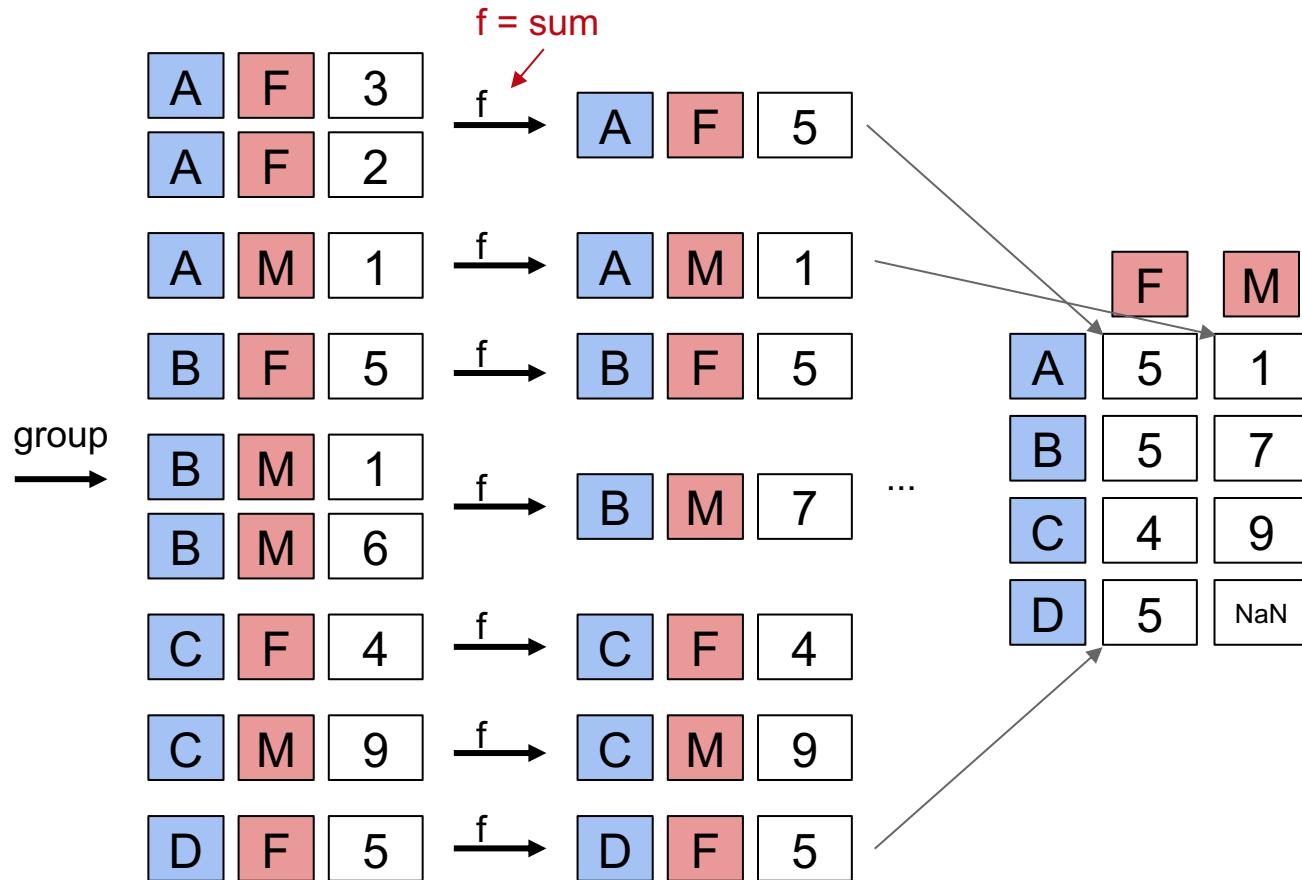
Count		
Year	Sex	
1910	F	5950
	M	3213
1911	F	6602
	M	3381
1912	F	9804
	M	8142

pivot\_table output

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

R	C	
A	F	3
B	M	1
C	F	4
A	M	1
B	F	5
C	M	9
A	F	2
D	F	5
B	M	6



## Pivot Tables with Multiple Values

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)
```

Year	Count		Name	
	Sex		F	M
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

# Join Tables

---

Lecture 05

- **Pandas, Part III**
  - Groupby Review
  - More on Groupby
  - Pivot Tables
  - **Joining Tables**
- EDA, Part I
  - Structure: Tabular Data
  - Granularity
  - Structure: Variable Types

## Joining Tables

---

Suppose want to know the popularity of presidential candidate's names in 2022.

- Example: Dwight Eisenhower's name Dwight is not popular today, with only 5 babies born with this name in California in 2022.

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

## Creating Table 1: Babynames in 2022

---

Let's set aside names in California from 2022 first:

```
babynames_2022 = babynames[babynames["Year"] == 2022]
```

```
babynames_2022
```

	State	Sex	Year	Name	Count
235835	CA	F	2022	Olivia	2178
235836	CA	F	2022	Emma	2080
235837	CA	F	2022	Camila	2046
235838	CA	F	2022	Mia	1882
235839	CA	F	2022	Sophia	1762
235840	CA	F	2022	Isabella	1733
235841	CA	F	2022	Luna	1516
235842	CA	F	2022	Sofia	1307
235843	CA	F	2022	Amelia	1289
235844	CA	F	2022	Gianna	1107

## 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
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878	John
2	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073	John
4	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew
...	...	...	...	...	...	...	...
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

182 rows × 7 columns

## Joining Our Tables

```
merged = pd.merge(left = elections, right = babynames_2022,  
                  left_on = "First Name", right_on = "Name")
```

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

	Year_x	Candidate	Party	Popular vote	Result	%	First Name	State	Sex	Year_y	Name	Count
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122	Andrew	CA	M	2022	Andrew	741
1	1828	Andrew Jackson	Democratic	642806	win	56.203927	Andrew	CA	M	2022	Andrew	741
2	1832	Andrew Jackson	Democratic	702735	win	54.574789	Andrew	CA	M	2022	Andrew	741
3	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878	John	CA	M	2022	John	490
4	1828	John Quincy Adams	National Republican	500897	loss	43.796073	John	CA	M	2022	John	490
...	...	...	...	...	...	...	...	...	...	...	...	...

## Joining Our Tables: Four Options

---

1. `inner_merged_df = pd.merge(left_df, right_df, on='key')`
2. `outer_merged_df = pd.merge(left_df, right_df, on='key', how='outer')`
3. `left_merged_df = pd.merge(left_df, right_df, on='key', how='left')`
4. `right_merged_df = pd.merge(left_df, right_df, on='key', how='right')`

**DataFrame 1**

	ID	Age
0	2	25
1	3	30
2	4	28

**DataFrame 2**

	ID	Name
0	0	Bob
1	1	Alice
2	2	John

## Joining Our Tables: Inner Merge

1. `inner_merged_df = pd.merge(left_df, right_df, on='ID')`

**DataFrame 1**

	ID	Name
0	1	Bob
1	2	Alice
2	3	John

**DataFrame 2**

	ID	Age
0	2	25
1	3	30
2	4	28

**Inner merge**

	ID	Name	Age
0	2	Alice	25
1	3	John	30

## Joining Our Tables: Outer Merge

1. outer\_merged\_df = pd.merge(left\_df, right\_df, on='ID', how='outer')

**DataFrame 1**

	ID	Name
0	1	Bob
1	2	Alice
2	3	John

**DataFrame 2**

	ID	Age
0	2	25
1	3	30
2	4	28

**outer merge**

	ID	Name	Age
0	1	Bob	NaN
1	2	Alice	25
2	3	John	30
4	4	NaN	28

## Joining Our Tables: right Merge

---

1. `right_merged_df = pd.merge(left_df, right_df=, on='ID', how='right')`

**DataFrame 1**

	ID	Name
0	1	Bob
1	2	Alice
2	3	John

**DataFrame 2**

	ID	Age
0	2	25
1	3	30
2	4	28

**right merge**

	ID	Name	Age
0	2	Alice	25
1	3	John	30
2	4	NaN	28

## Joining Our Tables: left Merge

---

1. `left_merged_df = pd.merge(left_df, right_df=, on='ID', how='left')`

**DataFrame 1**

	ID	Name
0	1	Bob
1	2	Alice
2	3	John

**DataFrame 2**

	ID	Age
0	2	25
1	3	30
2	4	28

**left merge**

	ID	Name	Age
0	1	Bob	NaN
1	2	Alice	25
2	3	John	30