

LECTURE 4

Pandas, Part 3

More Pandas (Utility Functions, Custom Sorts, Grouping, Aggregation)

CSCI 3022

Maribeth Oscamou

Announcements

- HW 2 Due Tomorrow (11:59pm MT on Canvas)
- Quiz 2 Friday (Scope: HW 1) 10 min.
- See Updated Office hours on Canvas

Alternatives to Boolean Selection

- **Alternatives to Boolean Selection**
- Adding, removing, and modifying columns
- Useful utility functions
- Custom sorts
- Grouping

Alternatives to Boolean Array Selection

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

```
elections[(elections["Party"] == "Anti-Masonic") |  
           (elections["Party"] == "American") |  
           (elections["Party"] == "Anti-Monopoly") |  
           (elections["Party"] == "American Independent")]
```

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

- `.isin`
- `.str.startswith`
- `.query`
- `.groupby.filter` (see lecture 5)

	Year	Candidate	Party	Popular vote	Result	%
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
22	1856	Millard Fillmore	American	873053	loss	21.554001
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
115	1968	George Wallace	American Independent	9901118	loss	13.571218
119	1972	John G. Schmitz	American Independent	1100868	loss	1.421524
124	1976	Lester Maddox	American Independent	170274	loss	0.209640
126	1976	Thomas J. Anderson	American	158271	loss	0.194862

Alternatives to Boolean Array Selection

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

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

```
a_parties = ["Anti-Masonic", "American", "Anti-Monopoly", "American Independent"]  
elections[elections["Party"].isin(a_parties)]
```

	Year	Candidate	Party	Popular vote	Result	%
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
22	1856	Millard Fillmore	American	873053	loss	21.554001
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
115	1968	George Wallace	American Independent	9901118	loss	13.571218
119	1972	John G. Schmitz	American Independent	1100868	loss	1.421524
124	1976	Lester Maddox	American Independent	170274	loss	0.209640
126	1976	Thomas J. Anderson	American	158271	loss	0.194862

Alternatives to Boolean Array Selection

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

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

```
elections[elections["Party"].str.startswith("A")]
```

	Year	Candidate	Party	Popular vote	Result	%
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
22	1856	Millard Fillmore	American	873053	loss	21.554001
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
115	1968	George Wallace	American Independent	9901118	loss	13.571218
119	1972	John G. Schmitz	American Independent	1100868	loss	1.421524
124	1976	Lester Maddox	American Independent	170274	loss	0.209640
126	1976	Thomas J. Anderson	American	158271	loss	0.194862

Alternatives to Boolean Array Selection

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

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

```
elections.query('Year >= 2000 and Result == "win"')
```

	Year	Candidate	Party	Popular vote	Result	%
152	2000	George W. Bush	Republican	50456002	win	47.974666
157	2004	George W. Bush	Republican	62040610	win	50.771824
162	2008	Barack Obama	Democratic	69498516	win	53.023510
168	2012	Barack Obama	Democratic	65915795	win	51.258484
173	2016	Donald Trump	Republican	62984828	win	46.407862
178	2020	Joseph Biden	Democratic	81268924	win	51.311515

One More Query Example

Query has a rich syntax.

- Can access Python variables with the special @ character.
- We won't cover **query** syntax in detail in our class, but you're welcome to use it.

```
parties = ["Republican", "Democratic"]
```

```
elections.query('Result == "win" and Party not in @parties')
```

	Year	Candidate	Party	Popular vote	Result	%
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
11	1840	William Henry Harrison	Whig	1275583	win	53.051213
16	1848	Zachary Taylor	Whig	1360235	win	47.309296
27	1864	Abraham Lincoln	National Union	2211317	win	54.951512

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

- `.isin`
- `.str.startswith`
- `.query`
- `.groupby.filter` (next lecture)

Adding, Removing, and Modifying Columns

- **Adding, removing, and modifying columns**
 - Useful utility functions
 - Custom sorts
 - Grouping

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
babynames["name_lengths"] = babynames["Name"].str.len()

# Add a column named "name_lengths" that
# includes the length of each name
babynames["name_lengths"] = babynames["name_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 x 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 x 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



	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 x 6 columns

407428 rows x 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!



Useful Utility Functions

- Adding, removing, and modifying columns
- **Useful utility functions**
- Custom sorts
- Grouping

Pandas **Series** and **DataFrames** support a large number of operations, including mathematical operations, so long as the data is numerical. [NumPy reference](#).

```
yash_count = babynames[babynames["Name"] == "Yash"]["Count"]
```

```
np.mean(yash_count)
```

```
17.142857142857142
```

```
np.max(yash_count)
```

```
29
```

```
331824    8
334114    9
336390   11
338773   12
341387   10
343571   14
345767   24
348230   29
350889   24
353445   29
356221   25
358978   27
361831   29
364905   24
367867   23
370945   18
374055   14
376756   18
379660   18
383338    9
385903   12
388529   17
391485   16
394906   10
397874    9
400171   15
403092   13
406006   13
```

```
Name: Count, dtype: int64
```

In addition to its rich syntax for indexing and support for other libraries (**NumPy**, native Python functions), **pandas** provides an enormous number of useful utility functions. Today, we'll discuss just a few:

- `size/shape`
- `sample`
- `value_counts`
- `unique`
- `sort_values`

The **pandas** library is rich in utility functions (we could spend the entire summer talking about them)! We encourage you to explore as you complete your assignments by Googling and reading [documentation](#), just as data scientists do.

.shape and .size

- **.shape** returns the shape of a **DataFrame** or **Series** in the form (number of rows, number of columns)
- **.size** returns the total number of entries in a **DataFrame** or **Series** (number of rows times number of columns)

babynames

	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

babynames.shape
(407428, 5)

babynames.size
2037140

407428 rows × 5 columns

.sample()

To sample a random selection of rows from a **DataFrame**, we 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
121141	CA	F	1992	Shanelle	28

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

	Year	Name	Count
44448	1961	Karyn	36
260410	1948	Carol	7
397541	2019	Arya	11
4767	1921	Sumiko	16
104369	1987	Thomas	11

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

	Year	Name	Count
151749	2000	Iridian	7
343560	2000	Maverick	14
149491	2000	Stacy	91
149212	2000	Angel	307

`.value_counts()`

The `Series.value_counts` method counts the number of occurrences of each unique value in a `Series` (it *counts* the number of times each *value* appears).

- Return value is also a `Series`.

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

```
Name
Jean      223
Francis   221
Guadalupe 218
Jessie    217
Marion    214
...
Renesme   1
Purity    1
Olanna    1
Nohea     1
Zayvier   1
Name: count, Length: 20437, dtype: int64
```

.unique()

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

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

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

`.sort_values()`

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

- **`Series.sort_values()` will automatically sort all values in the `Series`.**
- `DataFrame.sort_values(column_name)` must specify the name of the column to be used for sorting.

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

```
366001      Aadan
```

```
384005      Aadan
```

```
369120      Aadan
```

```
398211    Aadarsh
```

```
370306      Aaden
```

```
...
```

```
220691      Zyrah
```

```
197529      Zyrah
```

```
217429      Zyrah
```

```
232167      Zyrah
```

```
404544      Zyrus
```


```
Name: Name, Length: 407428, dtype: object
```

`.sort_values()`

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

- `Series.sort_values()` will automatically sort all values in the `Series`.
- `DataFrame.sort_values(column_name)` must specify the name of the column to be used for sorting.

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



	State	Sex	Year	Name	Count
268041	CA	M	1957	Michael	8260
267017	CA	M	1956	Michael	8258
317387	CA	M	1990	Michael	8246
281850	CA	M	1969	Michael	8245
283146	CA	M	1970	Michael	8196
...
317292	CA	M	1989	Olegario	5
317291	CA	M	1989	Norbert	5
317290	CA	M	1989	Niles	5
317289	CA	M	1989	Nikola	5
407427	CA	M	2022	Zylo	5

By default, rows are sorted in *ascending* order.

.sort_values()

You can sort by multiple values

```
DataFrame.sort_values(by=[col1, col2])
```

```
babynames.sort_values(by=["Count", "Name"])
```

	State	Sex	Year	Name	Count
268041	CA	M	1957	Michael	8260
267017	CA	M	1956	Michael	8258
317387	CA	M	1990	Michael	8246
281850	CA	M	1969	Michael	8245
283146	CA	M	1970	Michael	8196
...
317292	CA	M	1989	Olegario	5
317291	CA	M	1989	Norbert	5
317290	CA	M	1989	Niles	5
317289	CA	M	1989	Nikola	5
407427	CA	M	2022	Zylo	5

By default, rows are sorted in *ascending* order.

407428 rows x 5 columns

Custom Sorts

- Adding, removing, and modifying columns
- Useful utility functions
- **Custom sorts**
- Grouping

Sorting By Length:

Suppose we wanted to find the longest names our dataframe?

Just sorting by name won't work!

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

	State	Sex	Year	Name	Count
400698	CA	M	2020	Zyon	9
401664	CA	M	2020	Zymir	5
400829	CA	M	2020	Zyan	8
399585	CA	M	2020	Zyaire	37
400307	CA	M	2020	Zyair	13
...
400579	CA	M	2020	Aamir	9
401308	CA	M	2020	Aalam	5
400831	CA	M	2020	Aaditya	7
400830	CA	M	2020	Aadi	7
400466	CA	M	2020	Aaden	10

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 and Sort Based on the New Column

Suppose we wanted to find the longest names our dataframe?

```
# Create a Series of the length of each name
```

```
babynames["name_lengths"] = babynames["Name"].str.len()
```

```
# Add a column named "name_lengths" that includes the length of each name
```

```
babynames["name_lengths"] = babynames["name_lengths"]
```

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

```
babynames.head(5)
```

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

Approach 2: Sorting Using the key Argument

Define your own function:

```
def yourfunctionname(x):  
    return x.str.len()
```

```
(  
    babynames.sort_values("Name", key=yourfunctionname, ascending=False)  
        .head()  
)
```

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

Approach 2: Sorting Using the key Argument

Define your own function:

```
def yourfunctionname(x):  
    return x.str.len()
```

```
(  
babynames.sort_values("Name", key=yourfunctionname, ascending=False)  
    .head()  
)
```

SHORTCUT: USE LAMBDA NOTATION

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

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

Approach 3: Sorting Using the map Function

Suppose we want to sort by the number of occurrences of "dr" and "ea"s.

- 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)  
babynames.head()
```

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

Grouping

- Adding, removing, and modifying columns
- Useful utility functions
- Custom sorts
- **Grouping**

Why Group?

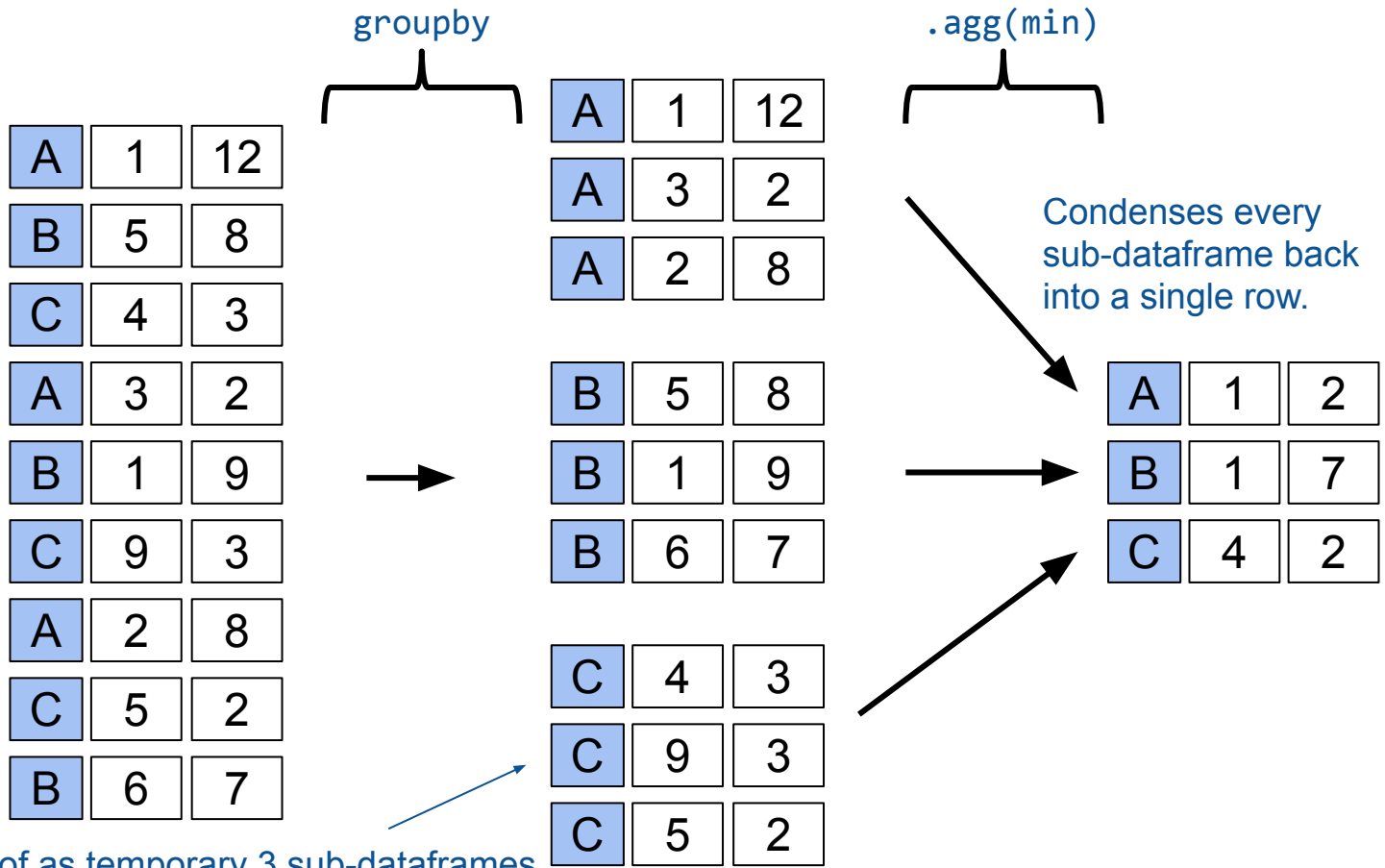
Our goal:

- Group together rows that fall under the same category.
 - For example, group together all rows from the same year.
- Perform an operation that *aggregates* across all rows in the category.
 - For example, sum up the total number of babies born in that year.

Grouping is a powerful tool to

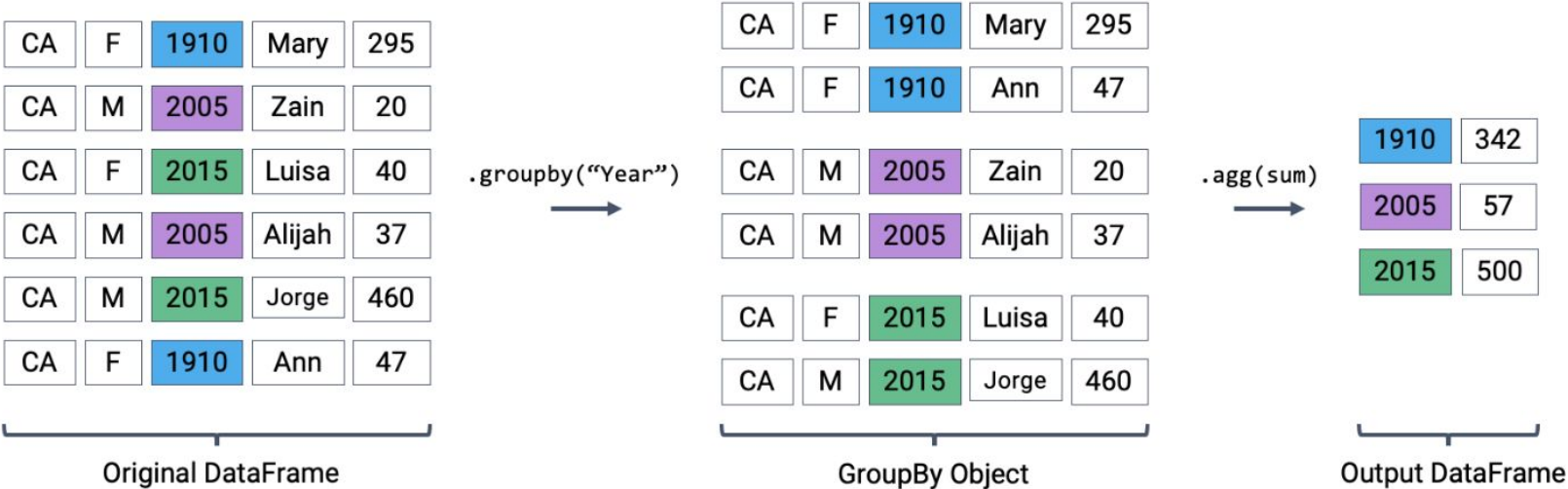
- 1) perform large operations, all at once and
- 2) summarize trends in a dataset.

Visual Overview of Grouping and Collection



Can think of as temporary 3 sub-dataframes

Visual Overview: 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 the same group (here, a particular 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

Original DataFrame

`.groupby("Year")`

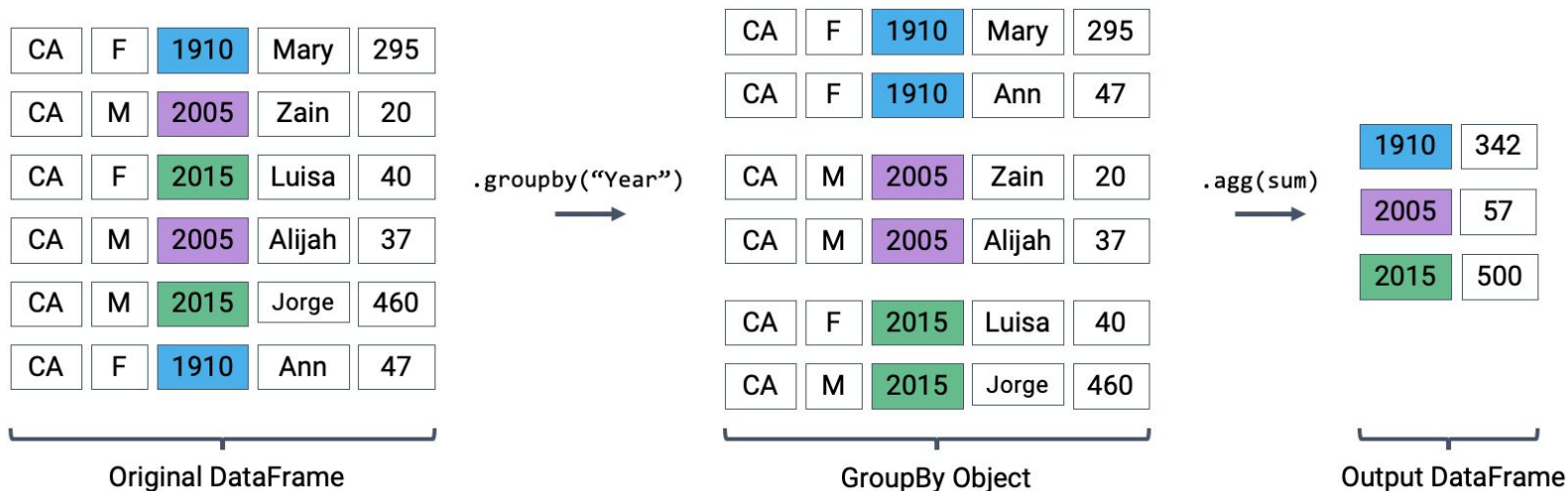


CA	F	1910	Mary	295
CA	F	1910	Ann	47
CA	M	2005	Zain	20
CA	M	2005	Alijah	37
CA	F	2015	Luisa	40
CA	M	2015	Jorge	460

GroupBy Object

`.groupby().agg()`

- We cannot work directly with **DataFrameGroupBy** objects! The diagram below is to help understand what goes on conceptually – in reality, we can't "see" the result of calling `.groupby()`.
- Instead, we transform a **DataFrameGroupBy** object back into a DataFrame using `.agg`
 - `.agg` is how we apply an aggregation operation to the data.

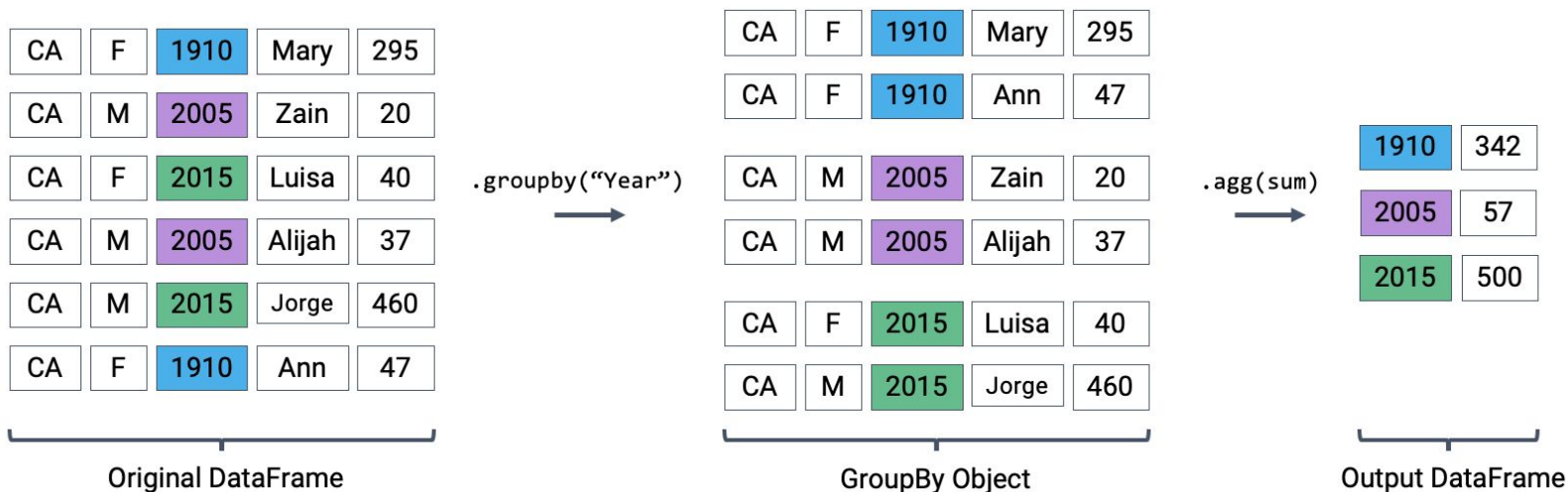


Where did the non-numeric columns go? We'll talk about this in a bit.

Putting It All Together

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

- `babynames.groupby("Year").agg(sum)` computes the total number of babies born in each 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()
```

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


Quick Subpuzzle

- a). 'Ak', 'tx', 'fl'
- b). 'Hi', 'tx', 'ak'
- c). 'Hi', 'tx', 'sd'
- d). 'Hi', 'nc', 'fl'
- e). None of these

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

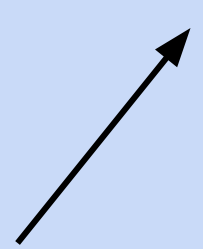
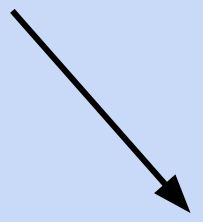


A	3	ak
A	1	hi
A	2	ca

B	1	tx
B	5	mi
B	6	nc

C	4	fl
C	9	ak
C	5	sd

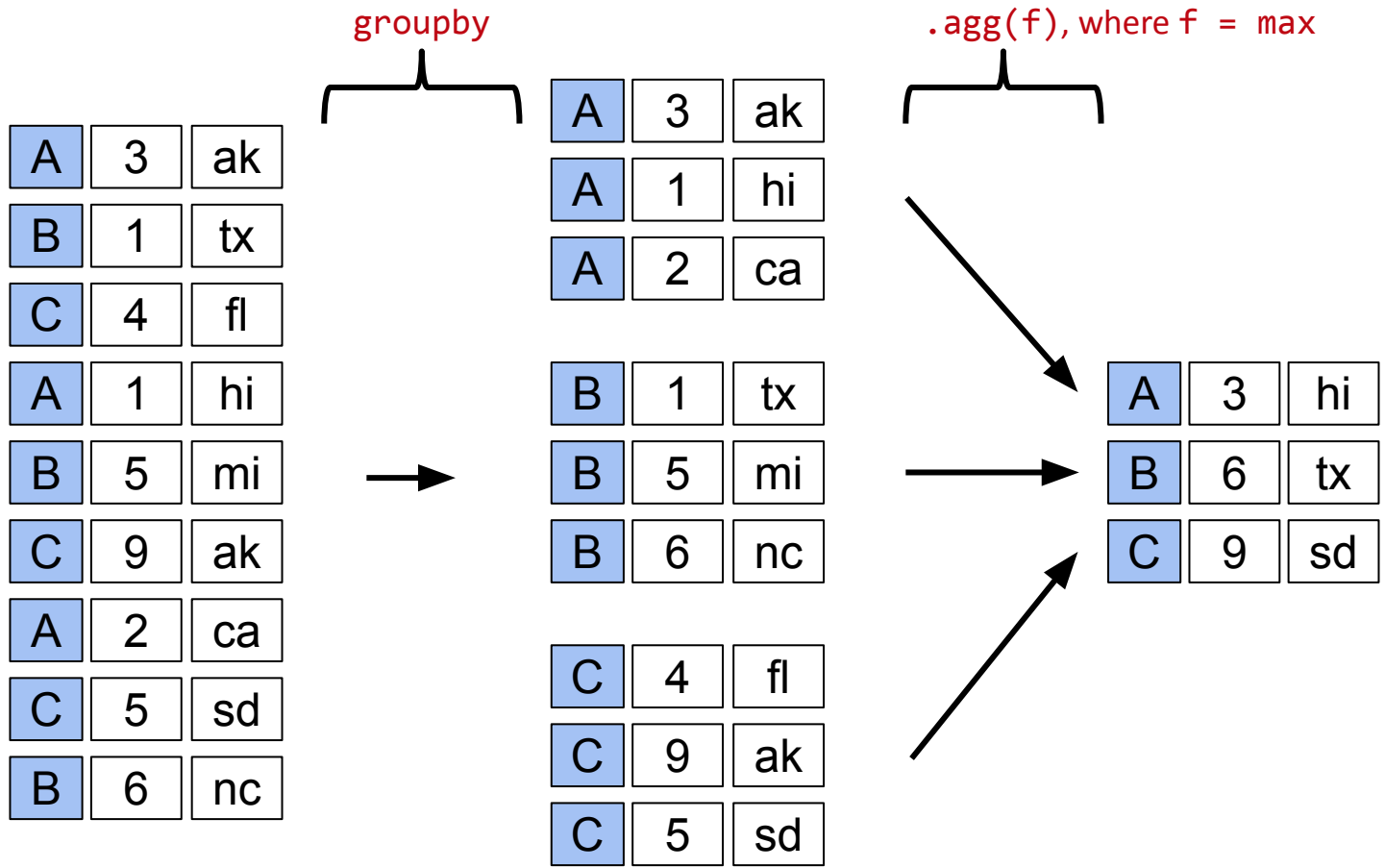
.agg(f), where f = max



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

What will go in the ??

Quick Subpuzzle



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

```
elections.groupby("Party").agg(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



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

Another GroupBy Puzzle

Try to write code that returns the table below.

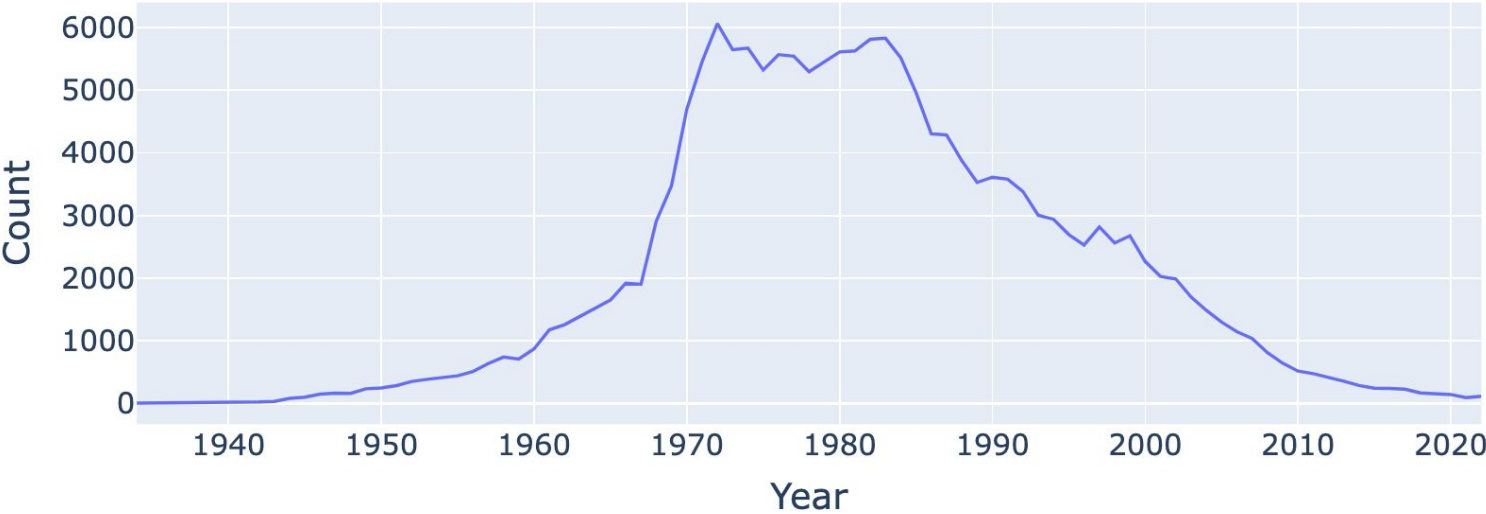
- Each row shows the best result (in %) by each party.
 - For example: Best Democratic result ever was Johnson's 1964 win.

	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

Putting Things Into Practice

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

Number of Jennifers Born in California Per Year

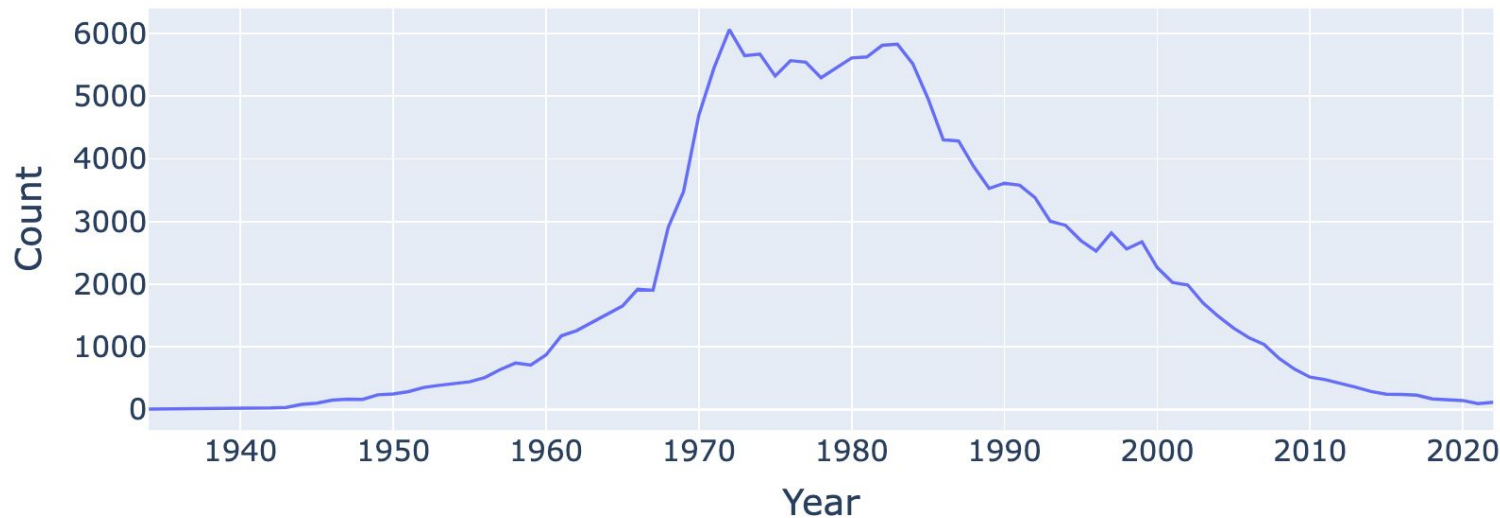


Putting Things Into Practice

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

```
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.

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 2021 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
```

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



```
rtp = curr_jenn / max_jenn
```

```
0.018796372629843364
```

```
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
```

Calculating RTP Using `.groupby()`

`.groupby()` makes it easy to compute the RTP for all names at once!

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

	Year	Count
Name		
Aadhini	1.0	1.000000
Aadhira	1.0	0.500000
Aadhya	1.0	0.660000
Aadya	1.0	0.586207
Aahana	1.0	0.269231
...
Zyanya	1.0	0.466667
Zyla	1.0	1.000000
Zylah	1.0	1.000000
Zyra	1.0	1.000000
Zyrah	1.0	0.833333

13782 rows × 2 columns

A Note on Nuisance Columns

At least as of the time of this slide creation (August 2023), executing our agg call results in a `TypeError`.

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

```
Cell In[110], line 5, in ratio_to_peak(series)
      1 def ratio_to_peak(series):
      2     """
      3     Compute the RTP for a Series containing the counts per year for a single name
      4     """
----> 5     return series.iloc[-1] / np.max(series)

TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

A Note on Nuisance Columns

Below, we explicitly select the column(s) we want to apply our aggregation function to **BEFORE** calling **agg**. This avoids the warning (and can prevent unintentional loss of data).

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

Count	
Name	
Aadhini	1.000000
Aadhira	0.500000
Aadhya	0.660000
Aadya	0.586207
Aahana	0.269231
...	...
Zyanya	0.466667
Zyla	1.000000
Zylah	1.000000
Zyra	1.000000
Zyrah	0.833333

13782 rows × 1 columns

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 = female_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
Aadhya	0.660000	Aadhya	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	
Debra	0.001260
Debbie	0.002815
Carol	0.003180
Tammy	0.003249
Susan	0.003305
...	...
Fidelia	1.000000
Naveyah	1.000000
Finlee	1.000000
Roseline	1.000000
Aadhini	1.000000

13782 rows x 1 columns



Some Data Science Payoff

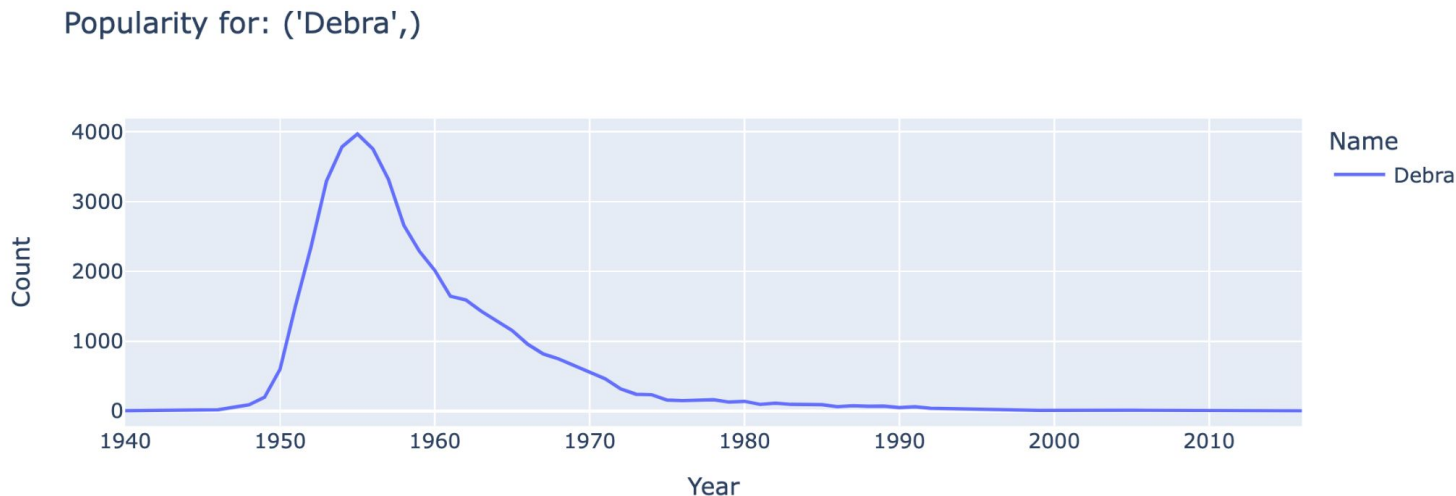
By sorting `rtp_table` we can see the names whose popularity has decreased the most.

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

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

13782 rows x 1 columns

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



We'll learn about plotting in week 3.

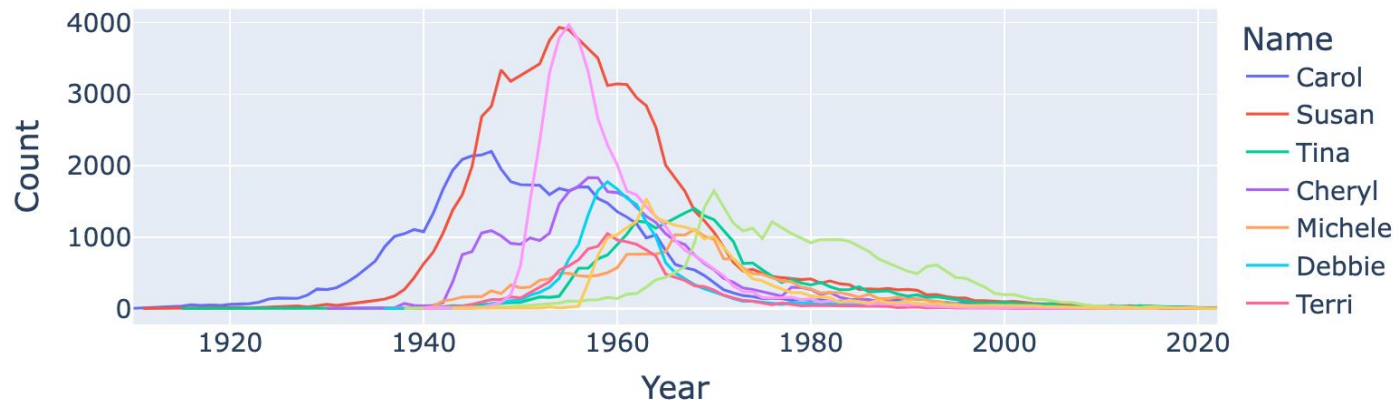
Some Data Science Payoff

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
```

```
ndex(['Debra', 'Debbie', 'Carol', 'Tammy', 'Susan', 'Cheryl', 'Shannon',  
      'Tina', 'Michele', 'Terri'],  
      dtype='object', name='Name')
```

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



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!

```
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.

Extra Content: View and Copies in Pandas

When we subset a numpy array, the result is not always a new array; sometimes what NumPy returns is a view of the data in the original array.

Since pandas Series and DataFrames are backed by numpy arrays, it will probably come as no surprise that something similar sometimes happens in pandas. Unfortunately, while this behavior is relatively straightforward in numpy, in pandas there's just no getting around the fact that it's a hot mess.

The View/Copy Headache in pandas

In `numpy`, the rules for when you get views and when you don't are a little complicated, but they are consistent: certain behaviors (like simple indexing) will *always* return a view, and others (fancy indexing) will *never* return a view.

But in `pandas`, whether you get a view or not—and whether changes made to a view will propagate back to the original DataFrame—depends on the structure and data types in the original DataFrame.

https://www.practicaldatascience.org/html/views_and_copies_in_pandas.html

The Good News

To help address this issue, pandas has a built-in alert system that will **sometimes** warning you when you're in a situation that may cause problems, called the `SettingWithCopyWarning`, which you can see here:

```
[11]: df = pd.DataFrame({"a": np.arange(4), "b": ["w", "x", "y", "z"]})
      my_slice = df["a"]
      my_slice
```

```
[11]: 0    0
      1    1
      2    2
      3    3
      Name: a, dtype: int64
```

```
[12]: my_slice.iloc[1] = 2
```

```
/var/folders/fs/h_8_rwsn5hvg9mhp0txgc_s9v6191b/T/ipykernel_41268/1176285234.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#
my_slice.iloc[1] = 2
```

Any time you see a `SettingWithCopyWarning`, go up to where the possible view was created (in this case, `my_slice = df["a"]`) and add a `.copy()`:

```
[13]: my_slice = df["a"].copy()
      my_slice.iloc[1] = 2
```

The Bad News

The bad news is that the `SettingWithCopyWarning` will only flag one pattern where the copy-view problem crops up. Indeed, if you follow the link provided in the warning, you'll see it wasn't designed to address the copy-view problem *writ large*, but rather a more narrow behavior where the user tries to change a subset of a DataFrame incorrectly (we'll talk more about that in our coming readings). Indeed, you'll notice we didn't get a single `SettingWithCopyWarning` until the section where we started talking about that warning in particular (and I created an example designed to set it off).

So: if you see a `SettingWithCopyWarning` **do not** ignore it—find where you may have created a view or may have created a copy and add a `.copy()` so the error goes away. **But just because you don't see that warning doesn't mean you're in the clear!**

Which leads me to what I will admit is an infuriating piece of advice to have to offer: **if you take a subset for any purpose other than immediately analyzing, you should add `.copy()` to that subsetting.** Seriously. Just when in doubt, `.copy()`.

Best Practice 1: Avoid Chained Indexing

Chained indexing occurs when you use multiple indexing operations consecutively. For example, if you select a column and then select a row, this is chained indexing. It's important to avoid chained indexing because it can create views instead of copies, which can lead to unexpected behavior.

Here's an example of chained indexing:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Chained indexing (avoid this!)
df_view = df['A'][0:2]

# This will raise a warning
df_view['A'] = [100, 200]
```

Output:

```
C:\Users\user\AppData\Local\Programs\Python\Python38\lib\site-packages\ipykernel_launcher.py:
A value is trying to be set on a copy of a slice from a DataFrame
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide
import sys
```

In this example, we use chained indexing to select the first two rows of the 'A' column. This creates a view of the DataFrame, which we then try to modify. However, we get a warning telling us that we are trying to modify a copy of the original DataFrame.

To avoid this, it's best to use `.loc` or `.iloc` to select rows and columns in a single operation.

Best Practice 2: Use `.copy()` When Needed

If you need to modify a DataFrame without affecting the original DataFrame, it's important to create a copy using the `.copy()` method. This will ensure that any changes you make to the copy will not affect the original DataFrame.

Here's an example:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Create a copy of the DataFrame
df_copy = df.copy()

# Modify the copy
df_copy['A'] = [100, 200, 300]

# The original DataFrame is not affected
print(df)
print(df_copy)
```

Output:

```
   A  B
0  1  4
1  2  5
2  3  6

   A  B
0 100  4
1 200  5
2 300  6
```

In this example, we create a copy of the original DataFrame using the `.copy()` method. We then modify the copy by setting the 'A' column to a new value. The original DataFrame is not affected by this operation.

Best Practice 3: Use .iloc for Slice Assignment

If you need to modify a slice of a DataFrame, it's important to use `.iloc` to ensure that a copy is created. This will ensure that any changes you make to the slice will not affect the original DataFrame.

Here's an example:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Modify a slice of the DataFrame
df.iloc[0:2]['A'] = [100, 200]

# The original DataFrame is not affected
print(df)
```

Output:

```
   A  B
0 100 4
1 200 5
2   3  6
```

In this example, we modify a slice of the DataFrame using `.iloc` to select the first two rows and the 'A' column. We then set the values of the 'A' column to a new value. However, the original DataFrame is not affected by this operation because a copy was created.