4 Pandas III

Code

Learning Outcomes

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

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

First, let's load `babynames` dataset.

▶ Code

	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

4.1 Aggregating Data with `.groupby`

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

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

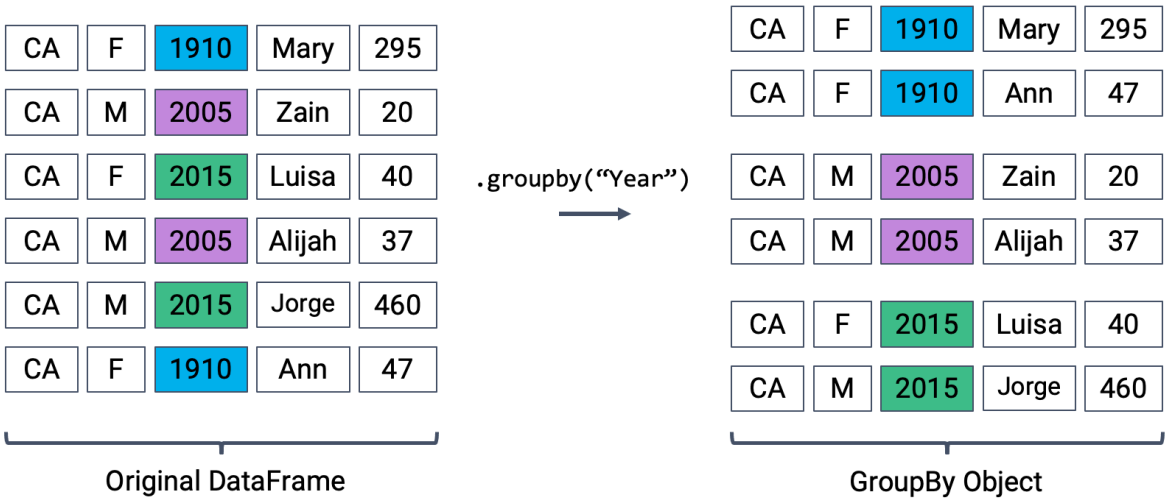
```
babynames.groupby("Year")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x10428bf50>
```

What does this strange output mean? Calling `.groupby` (documentation) has generated a `GroupBy` object. You can imagine this as a set of "mini", grouped sub-`DataFrame`s, where each group/sub-`DataFrame` contains all of

the rows from `babynames` that correspond to a particular year.

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



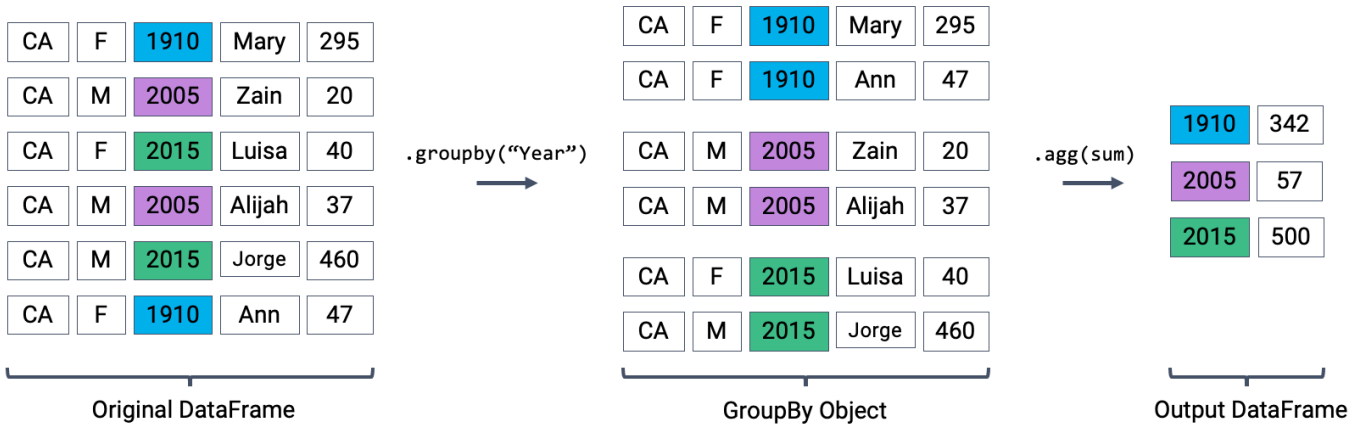
We can't work with a `GroupBy` object directly – that is why you saw that strange output earlier rather than a standard view of a `DataFrame`. To actually manipulate values within these groups/sub-`DataFrame`s, we'll need to call an *aggregation method*. This is a method that tells `pandas` how to aggregate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

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

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

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

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



Performing an aggregation

Calling `.agg` has condensed each subframe (i.e. group/sub-`DataFrame`) back into a single row. This gives us our final output: a `DataFrame` that is now indexed by `"Year"`, with a single row for each unique year in the original `babynames` DataFrame.

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

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

	Count
Year	
1910	5
1911	5
1912	5
1913	5
1914	5

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

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

```
# Same result, but now we explicitly tell pandas to only consider the "Count" column when
```

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

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

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

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

4.1.1 Aggregation Functions

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

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

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

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

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

	Count
Name	
Aadan	7
Aadarsh	6
Aaden	158

	Count
Name	
Aadhav	8
Aadhini	6

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

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

	Count
Name	
Aadan	6.000000
Aadarsh	6.000000
Aaden	46.214286
Aadhav	6.750000
Aadhini	6.000000

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

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

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

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

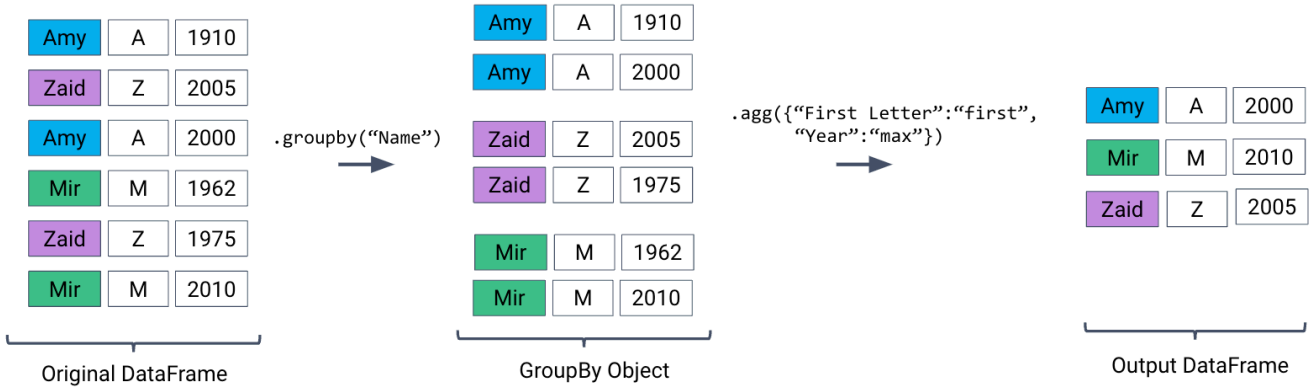
```
# Imagine we had an additional column, "First Letter". We'll explain this code next week
babynames["First Letter"] = babynames["Name"].str[0]

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

	Name	First Letter	Year
0	Mary	M	1910
1	Helen	H	1910
2	Dorothy	D	1910
3	Margaret	M	1910
4	Frances	F	1910

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

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



Aggregating using "first"

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

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

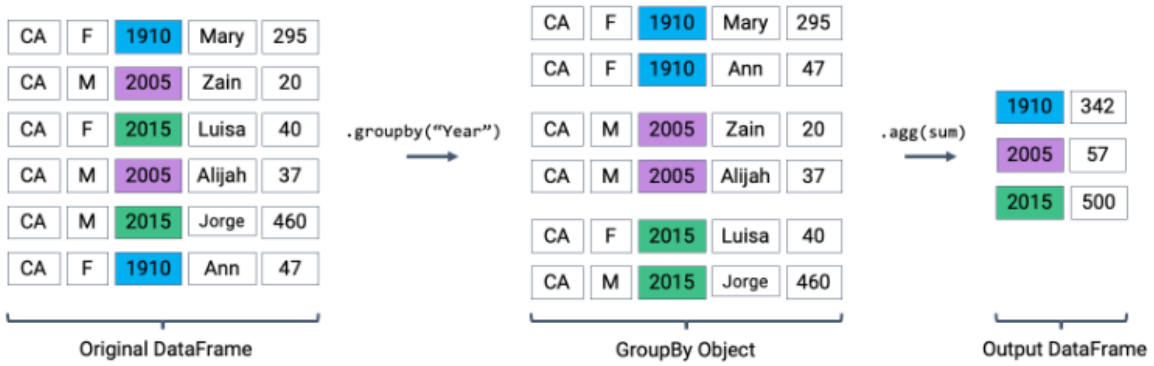
4.1.2 Plotting Birth Counts

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

▶ Code

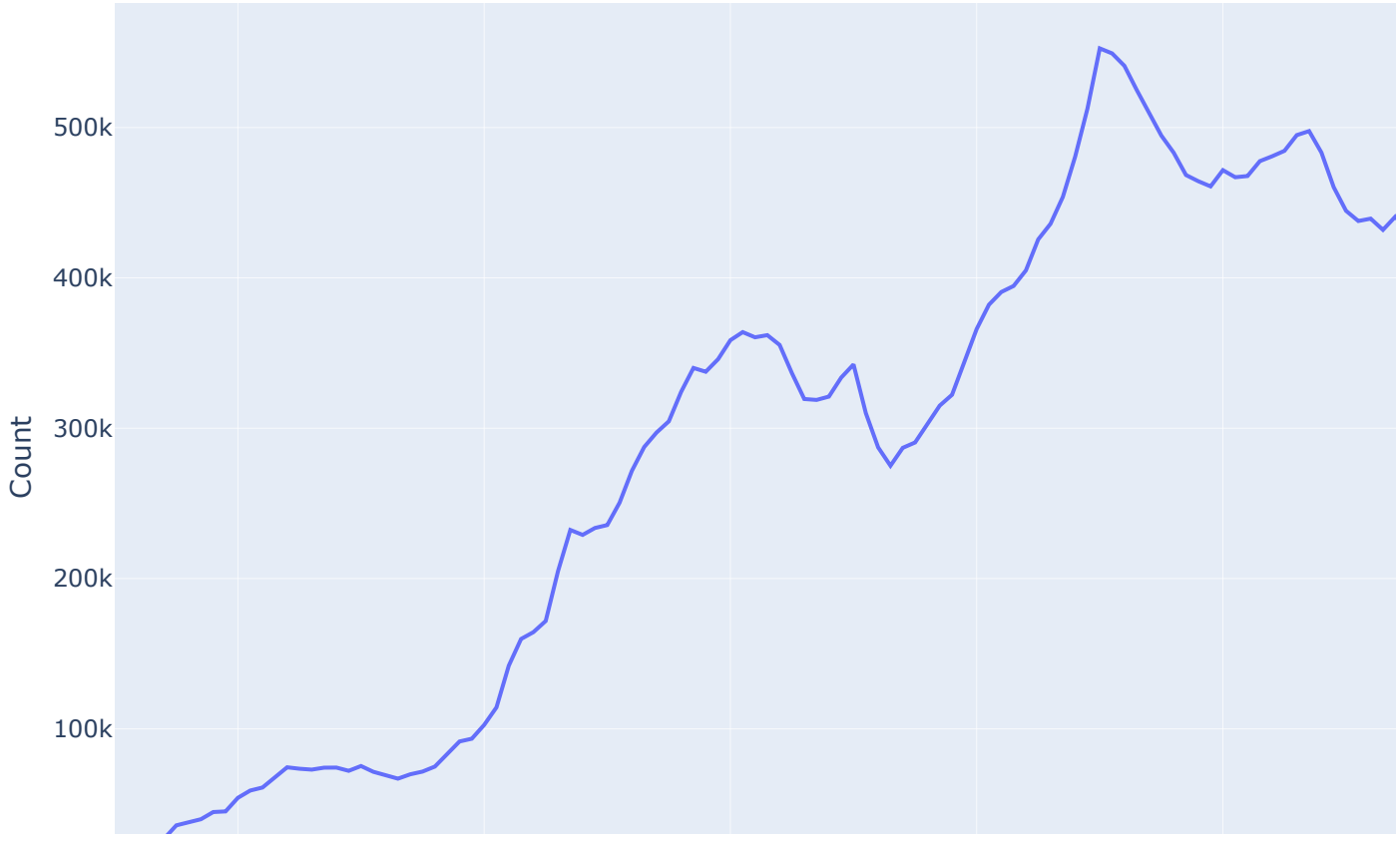
	Count
Year	
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

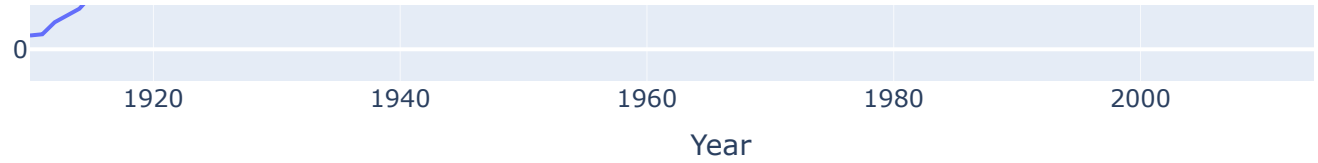
Here's an illustration of the process:



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

▶ Code





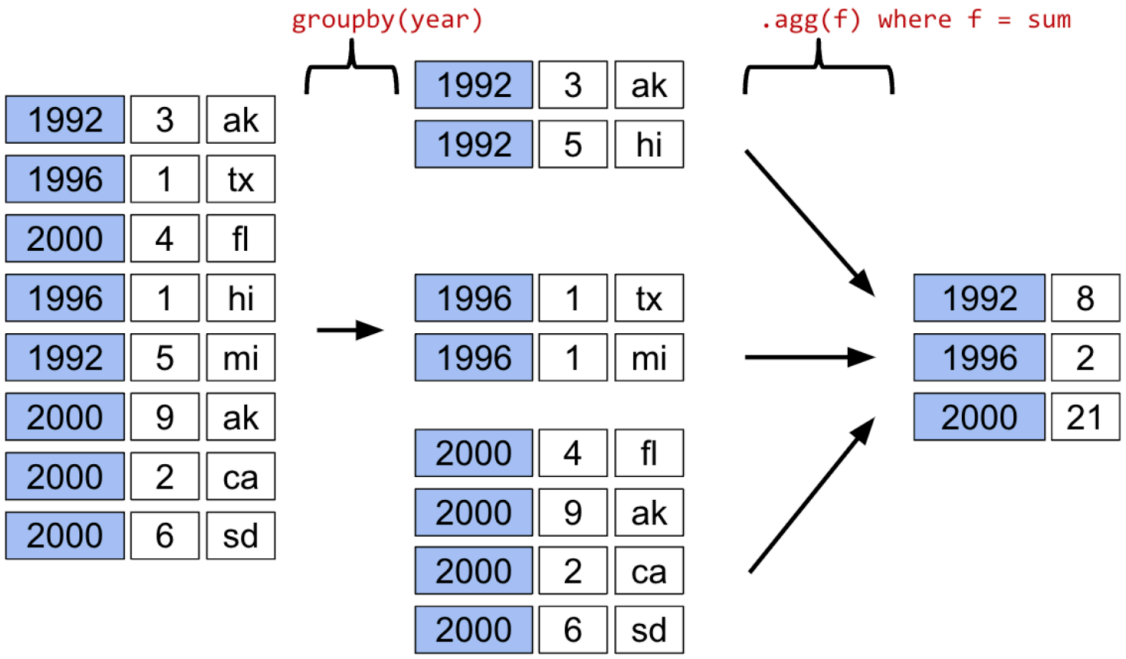
A word of warning: we made an enormous assumption when we decided to use this dataset to estimate birth rate. According to this article from the Legislative Analyst Office, the true number of babies born in California in 2020 was 421,275. However, our plot shows 362,882 babies —— what happened?

4.1.3 Summary of the `.groupby()` Function

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

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

- **Splits** the `DataFrame` into sub-`DataFrame`s with rows belonging to the same year.
- **Applies** the `sum` function to each column of each sub-`DataFrame`.
- **Combines** the results of `sum` into a single `DataFrame`, indexed by `year`.



4.1.4 Revisiting the `.agg()` Function

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

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

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

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

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

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

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

```
0.018796372629843364
```

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

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

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

	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

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

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

4.1.5 Nuisance Columns

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

```
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'
```

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

4.1.6 Renaming Columns After Grouping

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

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

	Year	Count RTP
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

4.1.7 Some Data Science Payoff

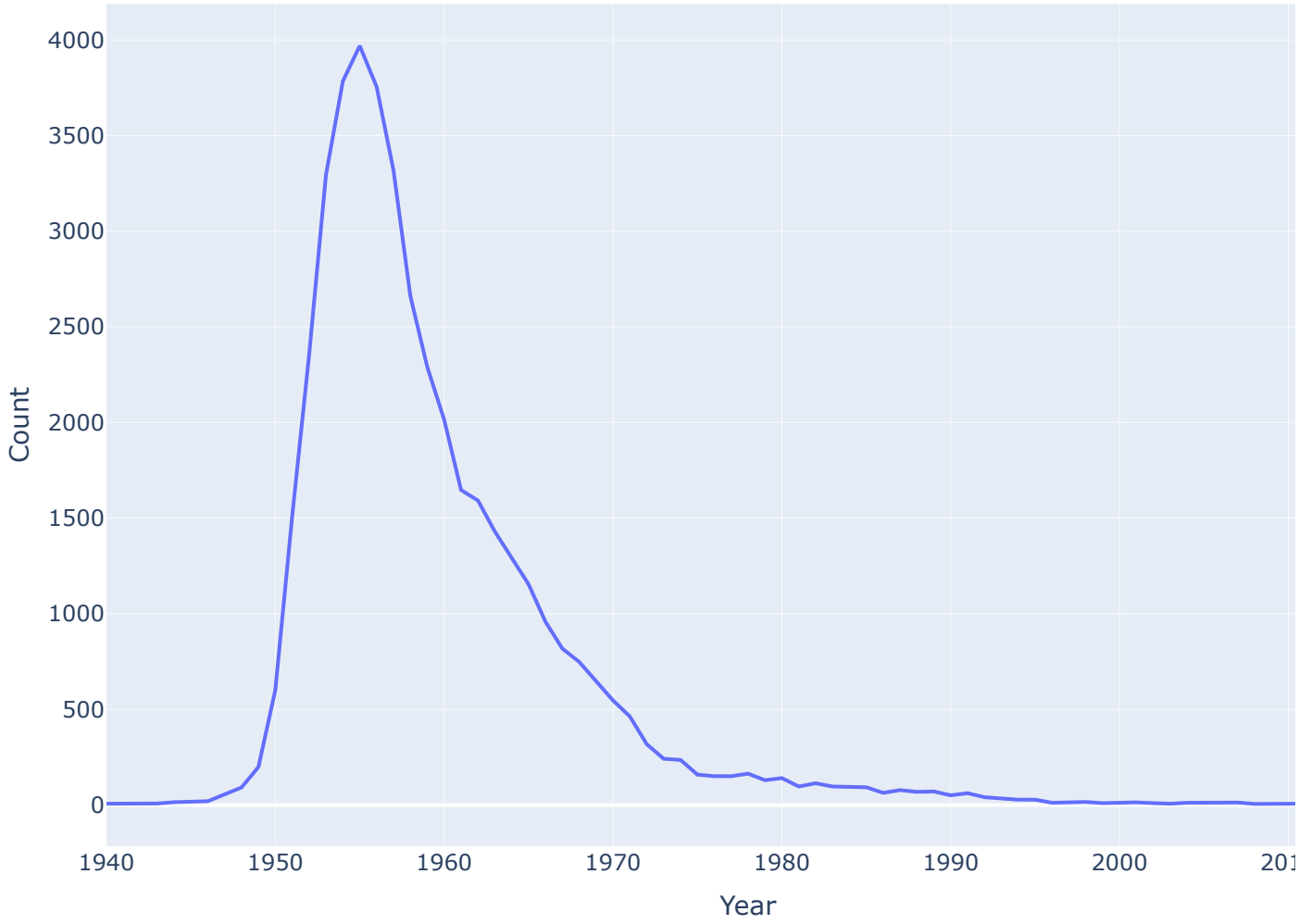
By sorting `rtp_table`, we can see the names whose popularity has decreased the most.

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

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

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

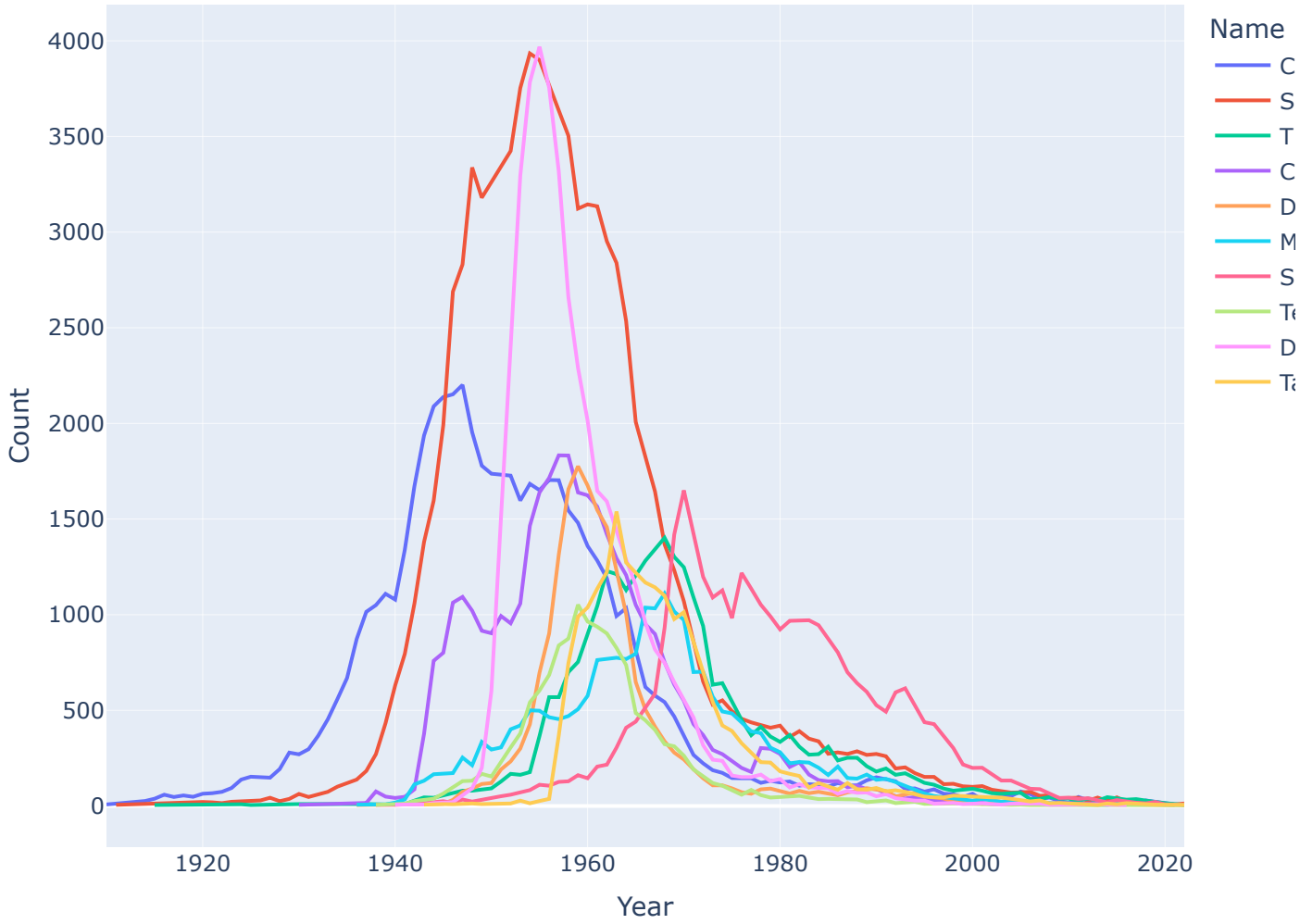
▶ Code



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

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

```
    color = "Name"
)
```



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

▶ Code

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

4.2 `.groupby()`, Continued

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

▶ Code

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

4.2.1 Raw `GroupBy` Objects

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

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

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

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

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

```
{'American': [22, 126], 'American Independent': [115, 119, 124], 'Anti-Masonic': [6],
'Anti-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, 183],
'Democratic-Republican': [0, 1], 'Dixiecrat': [103], 'Farmer-Labor': [78], 'Free Soil':
[15, 18], 'Green': [149, 155, 156, 165, 170, 177, 181, 184], 'Greenback': [35],
'Independent': [121, 130, 143, 161, 167, 174, 185], 'Liberal Republican': [31],
'Libertarian': [125, 128, 132, 138, 139, 146, 153, 159, 163, 169, 175, 180], 'Libertarian
Party': [186], '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, 182],
'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
76	1920	Eugene V. Debs	Socialist	913693	loss	3.428282
85	1928	Norman Thomas	Socialist	267478	loss	0.728623
88	1932	Norman Thomas	Socialist	884885	loss	2.236211
92	1936	Norman Thomas	Socialist	187910	loss	0.412876
95	1940	Norman Thomas	Socialist	116599	loss	0.234237
102	1948	Norman Thomas	Socialist	139569	loss	0.286312

4.2.2 Other `GroupBy` Methods

There are many aggregation methods we can use with `.agg`. Some useful options are:

- `.mean()`: creates a new `DataFrame` with the mean value of each group
- `.sum()`: creates a new `DataFrame` with the sum of each group
- `.max()` and `.min()`: creates a new `DataFrame` with the maximum/minimum value of each group
- `.first()` and `.last()`: creates a new `DataFrame` with the first/last row in each group
- `.head(n)` and `.tail(n)`: creates a new `DataFrame` with the first/last `n` rows in each group
- `.size()`: creates a new `Series` with the number of entries in each group
- `.count()`: creates a new `DataFrame` with the number of entries, excluding missing values.

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

```
df = pd.DataFrame({'letter':['A','A','B','C','C','C'],
                   'num':[1,2,3,4,np.nan,4],
                   'state':[np.nan, 'tx', 'fl', 'hi', np.nan, 'ak']})
df
```

	letter	num	state
0	A	1.0	NaN
1	A	2.0	tx
2	B	3.0	fl
3	C	4.0	hi
4	C	NaN	NaN
5	C	4.0	ak

Note the slight difference between `.size()` and `.count()`: while `.size()` returns a `Series` and counts the number of entries including the missing values, `.count()` returns a `DataFrame` and counts the number of

entries in each column *excluding missing values*.

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

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

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

	num	state
letter		
A	2	1
B	1	1
C	2	2

You might recall that the `value_counts()` function in the previous note does something similar. It turns out `value_counts()` and `groupby.size()` are the same, except `value_counts()` sorts the resulting `Series` in descending order automatically.

```
df["letter"].value_counts()
```

```
letter
C    3
A    2
B    1
Name: count, dtype: int64
```

These (and other) aggregation functions are so common that `pandas` allows for writing shorthand. Instead of explicitly stating the use of `.agg`, we can call the function directly on the `GroupBy` object.

For example, the following are equivalent:

- `elections.groupby("Candidate").agg(mean)`
- `elections.groupby("Candidate").mean()`

There are many other methods that `pandas` supports. You can check them out on the `pandas` documentation.

4.2.3 Filtering by Group

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

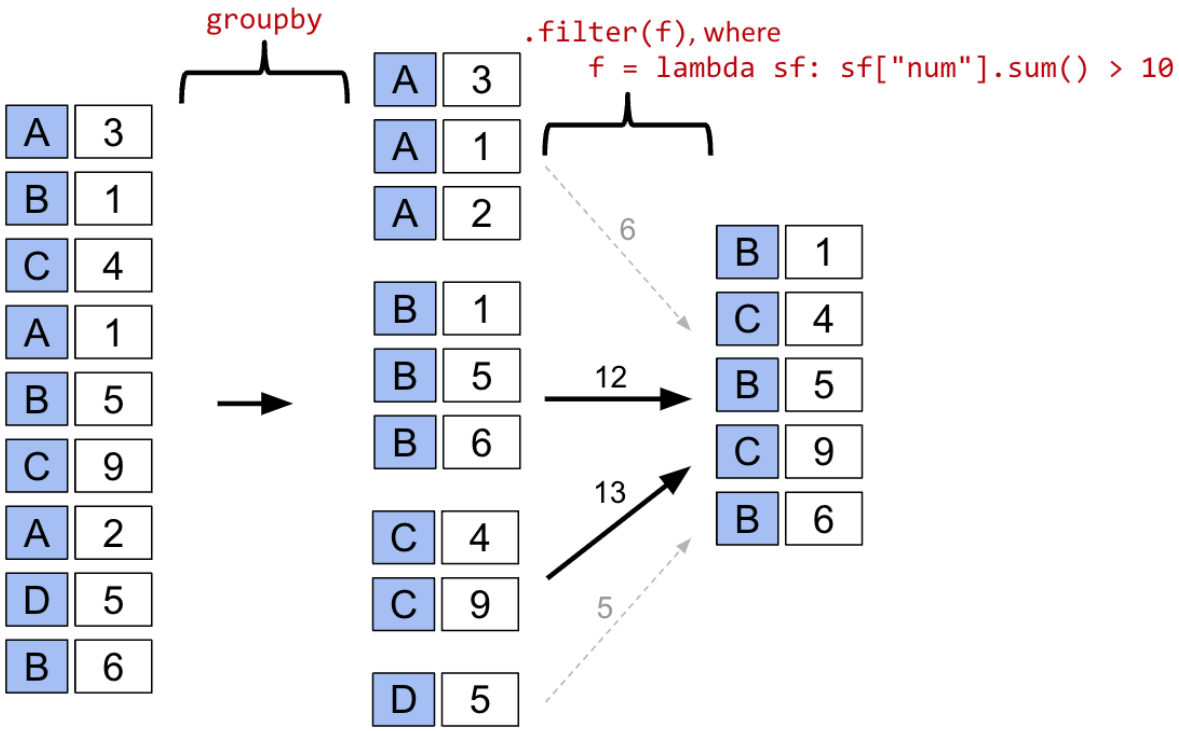
`groupby.filter` takes an argument `func`, where `func` is a function that:

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

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

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

In other words, sub-`DataFrame`s that correspond to `True` are returned in the final result, whereas those with a `False` value are not. Importantly, `groupby.filter` is different from `groupby.agg` in that an *entire* sub-`DataFrame` is returned in the final `DataFrame`, not just a single row. As a result, `groupby.filter` preserves the original indices and the column we grouped on does **NOT** become the index!



`sf` refers to subframe or sub-`DataFrame` which are the "mini", grouped sub-`DataFrame`s.

To illustrate how this happens, let's go back to the `elections` dataset. Say we want to identify "tight" election years – that is, we want to find all rows that correspond to election years where all candidates in that year won a similar portion of the total vote. Specifically, let's find all rows corresponding to a year where no candidate won more than 45% of the total vote.

In other words, we want to:

- Find the years where the maximum `%` in that year is less than 45%
- Return all `DataFrame` rows that correspond to these years

For each year, we need to find the maximum `%` among *all* rows for that year. If this maximum `%` is lower than 45%, we will tell `pandas` to keep all rows corresponding to that year.

```
elections.groupby("Year").filter(lambda sf: sf["%"].max() < 45).head(9)
```

	Year	Candidate	Party	Popular vote	Result	%
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
66	1912	Eugene V. Debs	Socialist	901551	loss	6.004354
67	1912	Eugene W. Chafin	Prohibition	208156	loss	1.386325
68	1912	Theodore Roosevelt	Progressive	4122721	loss	27.457433
69	1912	William Taft	Republican	3486242	loss	23.218466
70	1912	Woodrow Wilson	Democratic	6296284	win	41.933422

What's going on here? In this example, we've defined our filtering function, `func`, to be `lambda sf: sf["%"].max() < 45`. This filtering function will find the maximum `"%"` value among all entries in the grouped sub-`DataFrame`, which we call `sf`. If the maximum value is less than 45, then the filter function will return `True` and all rows in that grouped sub-`DataFrame` will appear in the final output `DataFrame`.

Examine the `DataFrame` above. Notice how, in this preview of the first 9 rows, all entries from the years 1860 and 1912 appear. This means that in 1860 and 1912, no candidate in that year won more than 45% of the total vote.

You may ask: how is the `groupby.filter` procedure different to the boolean filtering we've seen previously? Boolean filtering considers *individual* rows when applying a boolean condition. For example, the code `elections[elections["%"] < 45]` will check the `"%"` value of every single row in `elections`; if it is less than 45, then that row will be kept in the output. `groupby.filter`, in contrast, applies a boolean condition *across* all rows in a group. If not all rows in that group satisfy the condition specified by the filter, the entire group will be discarded in the output.

4.2.4 Aggregation with `lambda` Functions

What if we wish to aggregate our `DataFrame` using a non-standard function – for example, a function of our own design? We can do so by combining `.agg` with `lambda` expressions.

Let's first consider a puzzle to jog our memory. We will attempt to find the `Candidate` from each `Party` with the highest `%` of votes.

A naive approach may be to group by the `Party` column and aggregate by the maximum.

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

	Year	Candidate	Popular vote	Result	%
Party					
American	1976	Thomas J. Anderson	873053	loss	21.554001
American Independent	1976	Lester Maddox	9901118	loss	13.571218

Party	Year	Candidate	Popular vote	Result	%
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	2024	Woodrow Wilson	81268924	win	61.344703
Democratic-Republican	1824	John Quincy Adams	151271	win	57.210122

This approach is clearly wrong – the `DataFrame` claims that Woodrow Wilson won the presidency in 2020.

Why is this happening? Here, the `max` aggregation function is taken over every column *independently*. Among Democrats, `max` is computing:

- The most recent `Year` a Democratic candidate ran for president (2020)
- The `Candidate` with the alphabetically "largest" name ("Woodrow Wilson")
- The `Result` with the alphabetically "largest" outcome ("win")

Instead, let's try a different approach. We will:

1. Sort the `DataFrame` so that rows are in descending order of `%`
2. Group by `Party` and select the first row of each sub-`DataFrame`

While it may seem unintuitive, sorting `elections` by descending order of `%` is extremely helpful. If we then group by `Party`, the first row of each `GroupBy` object will contain information about the `Candidate` with the highest voter `%`.

```
elections_sorted_by_percent = elections.sort_values("%", ascending=False)
elections_sorted_by_percent.head(5)
```

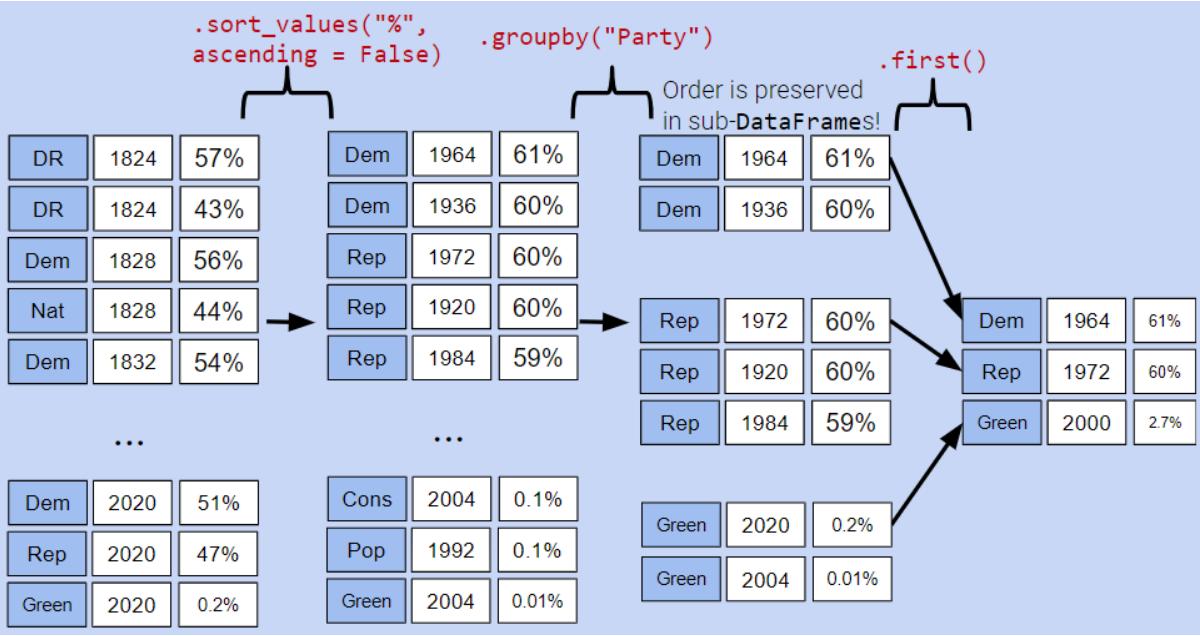
	Year	Candidate	Party	Popular vote	Result	%
114	1964	Lyndon Johnson	Democratic	43127041	win	61.344703
91	1936	Franklin Roosevelt	Democratic	27752648	win	60.978107
120	1972	Richard Nixon	Republican	47168710	win	60.907806
79	1920	Warren Harding	Republican	16144093	win	60.574501
133	1984	Ronald Reagan	Republican	54455472	win	59.023326

```
elections_sorted_by_percent.groupby("Party").agg(lambda x : x.iloc[0]).head(10)
```

```
# Equivalent to the below code
# elections_sorted_by_percent.groupby("Party").agg('first').head(10)
```

	Year	Candidate	Popular vote	Result	%
Party					
American	1856	Millard Fillmore	873053	loss	21.554001
American Independent	1968	George Wallace	9901118	loss	13.571218
Anti-Masonic	1832	William Wirt	100715	loss	7.821583
Anti-Monopoly	1884	Benjamin Butler	134294	loss	1.335838
Citizens	1980	Barry Commoner	233052	loss	0.270182
Communist	1932	William Z. Foster	103307	loss	0.261069
Constitution	2008	Chuck Baldwin	199750	loss	0.152398
Constitutional Union	1860	John Bell	590901	loss	12.639283
Democratic	1964	Lyndon Johnson	43127041	win	61.344703
Democratic-Republican	1824	Andrew Jackson	151271	loss	57.210122

Here's an illustration of the process:



Notice how our code correctly determines that Lyndon Johnson from the Democratic Party has the highest voter `%`.

More generally, `lambda` functions are used to design custom aggregation functions that aren't pre-defined by Python. The input parameter `x` to the `lambda` function is a `GroupBy` object. Therefore, it should make sense why `lambda x : x.iloc[0]` selects the first row in each groupby object.

In fact, there's a few different ways to approach this problem. Each approach has different tradeoffs in terms of readability, performance, memory consumption, complexity, etc. We've given a few examples below.

Note: Understanding these alternative solutions is not required. They are given to demonstrate the vast number of problem-solving approaches in `pandas`.

```
# Using the idxmax function
best_per_party = elections.loc[elections.groupby('Party')['%'].idxmax()]
best_per_party.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
22	1856	Millard Fillmore	American	873053	loss	21.554001
115	1968	George Wallace	American Independent	9901118	loss	13.571218
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
127	1980	Barry Commoner	Citizens	233052	loss	0.270182

```
# Using the .drop_duplicates function
best_per_party2 = elections.sort_values('%').drop_duplicates(['Party'], keep='last')
best_per_party2.head(5)
```

	Year	Candidate	Party	Popular vote	Result	%
148	1996	John Hagelin	Natural Law	113670	loss	0.118219
164	2008	Chuck Baldwin	Constitution	199750	loss	0.152398
110	1956	T. Coleman Andrews	States' Rights	107929	loss	0.174883
147	1996	Howard Phillips	Taxpayers	184656	loss	0.192045
136	1988	Lenora Fulani	New Alliance	217221	loss	0.237804

4.3 Aggregating Data with Pivot Tables

We know now that `.groupby` gives us the ability to group and aggregate data across our `DataFrame`. The examples above formed groups using just one column in the `DataFrame`. It's possible to group by multiple columns at once by passing in a list of column names to `.groupby`.

Let's consider the `babynames` dataset again. In this problem, we will find the total number of baby names associated with each sex for each year. To do this, we'll group by *both* the `"Year"` and `"Sex"` columns.

```
babynames.head()
```

	State	Sex	Year	Name	Count	First Letter
0	CA	F	1910	Mary	295	M
1	CA	F	1910	Helen	239	H

	State	Sex	Year	Name	Count	First Letter
2	CA	F	1910	Dorothy	220	D
3	CA	F	1910	Margaret	163	M
4	CA	F	1910	Frances	134	F

```
# Find the total number of baby names associated with each sex for each
# year in the data
babynames.groupby(["Year", "Sex"])[["Count"]].agg(sum).head(6)
```

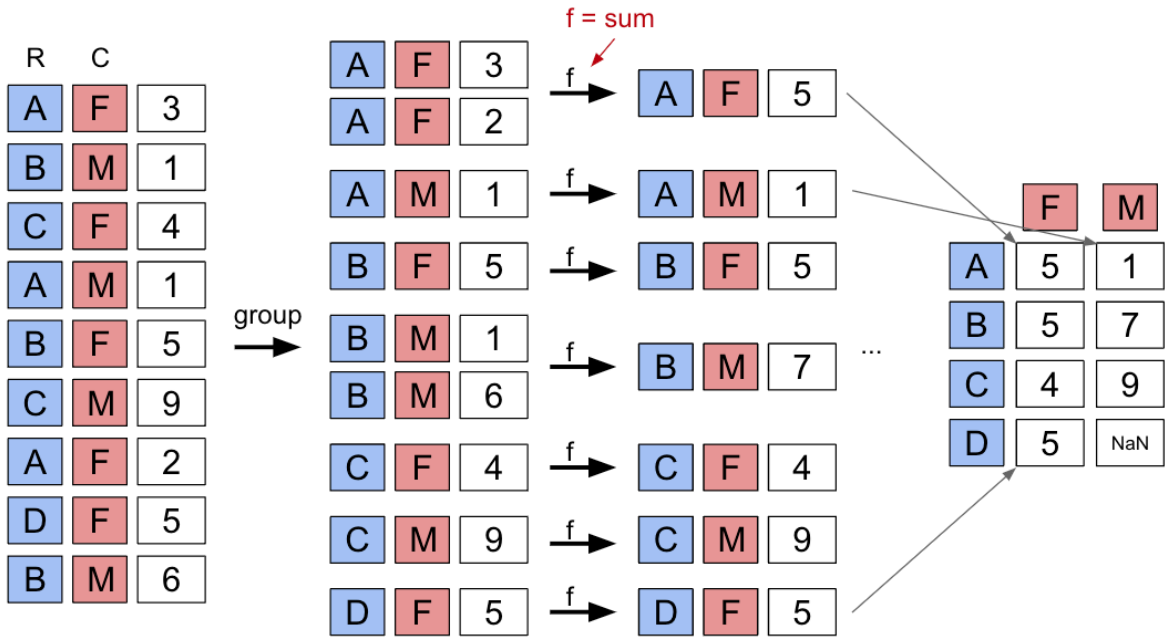
		Count
Year	**Sex**	
1910	**F**	5950
	M	3213
1911	**F**	6602
	M	3381
1912	**F**	9804
	M	8142

Notice that both `"Year"` and `"Sex"` serve as the index of the `DataFrame` (they are both rendered in bold). We've created a *multi-index* `DataFrame` where two different index values, the year and sex, are used to uniquely identify each row.

This isn't the most intuitive way of representing this data – and, because multi-indexed DataFrames have multiple dimensions in their index, they can often be difficult to use.

Another strategy to aggregate across two columns is to create a pivot table. You saw these back in Data 8. One set of values is used to create the index of the pivot table; another set is used to define the column names. The values contained in each cell of the table correspond to the aggregated data for each index-column pair.

Here's an illustration of the process:



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

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

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

Looks a lot better! Now, our `DataFrame` is structured with clear index-column combinations. Each entry in the pivot table represents the summed count of names for a given combination of `"Year"` and `"Sex"`.

Let's take a closer look at the code implemented above.

- `index = "Year"` specifies the column name in the original `DataFrame` that should be used as the index of the pivot table

- `columns = "Sex"` specifies the column name in the original `DataFrame` that should be used to generate the columns of the pivot table
- `values = "Count"` indicates what values from the original `DataFrame` should be used to populate the entry for each index-column combination
- `aggfunc = np.sum` tells `pandas` what function to use when aggregating the data specified by `values`. Here, we are summing the name counts for each pair of `"Year"` and `"Sex"`

We can even include multiple values in the index or columns of our pivot tables.

```
babynames_pivot = babynames.pivot_table(
    index="Year",     # the rows (turned into index)
    columns="Sex",    # the column values
    values=["Count", "Name"],
    aggfunc="max",      # group operation
)
babynames_pivot.head(6)
```

	Count		Name	
Sex	**F**	**M**	**F**	**M**
Year				
1910	295	237	Yvonne	William
1911	390	214	Zelma	Willis
1912	534	501	Yvonne	Woodrow
1913	584	614	Zelma	Yoshio
1914	773	769	Zelma	Yoshio
1915	998	1033	Zita	Yukio

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

4.4 Joining Tables

When working on data science projects, we're unlikely to have absolutely all the data we want contained in a single `DataFrame` – a real-world data scientist needs to grapple with data coming from multiple sources. If we have access to multiple datasets with related information, we can join two or more tables into a single `DataFrame`.

To put this into practice, we'll revisit the `elections` dataset.

```
elections.head(5)
```

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

Say we want to understand the popularity of the names of each presidential candidate in 2022. To do this, we'll need the combined data of `babynames` *and* `elections`.

We'll start by creating a new column containing the first name of each presidential candidate. This will help us join each name in `elections` to the corresponding name data in `babynames`.

```
# This `str` operation splits each candidate's full name at each
# blank space, then takes just the candidate's first name
elections["First Name"] = elections["Candidate"].str.split().str[0]
elections.head(5)
```

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

```
# Here, we'll only consider `babynames` data from 2022
babynames_2022 = babynames[babynames["Year"]==2022]
babynames_2022.head()
```

	State	Sex	Year	Name	Count	First Letter
235835	CA	F	2022	Olivia	2178	O
235836	CA	F	2022	Emma	2080	E
235837	CA	F	2022	Camila	2046	C
235838	CA	F	2022	Mia	1882	M
235839	CA	F	2022	Sophia	1762	S

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

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

```
# Notice that pandas automatically specifies `Year_x` and `Year_y`
# when both merged DataFrames have the same column name to avoid confusion

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

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

Let's take a closer look at the parameters:

- `left` and `right` parameters are used to specify the `DataFrame`s to be joined.
- `left_on` and `right_on` parameters are assigned to the string names of the columns to be used when performing the join. These two `on` parameters tell `pandas` what values should act as pairing keys to determine which rows to merge across the `DataFrame`s. We'll talk more about this idea of a pairing key next lecture.

4.5 Parting Note

Congratulations! We finally tackled `pandas`. Don't worry if you are still not feeling very comfortable with it—you will have plenty of chances to practice over the next few weeks.

Next, we will get our hands dirty with some real-world datasets and use our `pandas` knowledge to conduct some exploratory data analysis.