LECTURE 5

Pandas, Part II

Advanced Pandas syntax, aggregation, and joining

Data 100/Data 200, Fall 2021 @ UC Berkeley
Fernando Pérez and Alvin Wan
(content by Josh Hug, F. Pérez)

Announcements

Quick note: In this class, I use the terms "method" and "function" interchangeably.

For example:

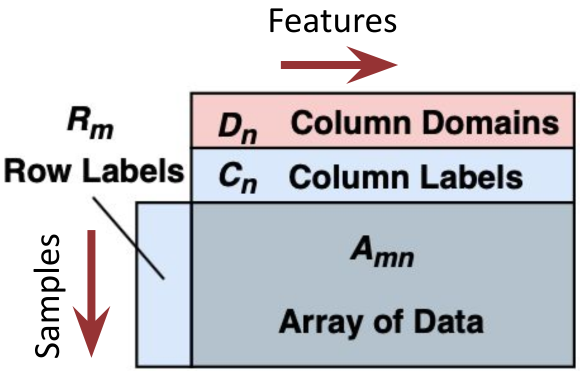
- `df.value_counts()` is a method. It is also a function.

Quick review from last lecture

The world, a statistician's view (I'm NOT a statistician 😀)

A (statistical) population
from which we draw
samples.
Each sample has certain
features.

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win



A generic DataFrame
(from https://arxiv.org/abs/2001.00888)

The Relationship Between Data Frames, Series, and Indices

We can think of a Data Frame as a collection of Series that all share the same Index.

- Candidate, Party, %, Year, and Result Series all share an index from 0 to 5.

Candidate Series   Party Series   % Series   Year Series   Result Series

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Non-native English speaker note: The plural of “series” is “series”. Sorry.

DataFrame access: [], loc, iloc

[]: flexible, confusing?

Name → [ ] → Series

Single Column Selection

List → [ ] → DataFrame

Multiple Column Selection

Numeric Slice → [ ] → DataFrame

(Multiple) Row Selection

loc: Labels

- Strings, integers - row/column labels
- Lists - similar, but always return dataframes
- Slices of labels: **end-point inclusive**!
- Boolean arrays: “mask” selection.

iloc: integer/positional

- Always 0-based, for rows and columns.
- Slices as usual, end-point exclusive.
- Use carefully (error prone).

head, size, shape, and describe

- Handy utilities to summarize a DF.

This Lecture: New Syntax / Concept Summary

- Operations on String series, e.g. babynames[“Name”].str.startswith()
- Creating and dropping columns.
  - Creating temporary columns is often convenient for sorting.
- Passing an index as an argument to loc.
  - Useful as an alternate way to sort a dataframe.
- Groupby: Output of .groupby(“Name”) is a DataFrameGroupBy object. Condense back into a DataFrame or Series with:
  - groupby.agg
  - groupby.size
  - groupby.filter
  - and more...
- Pivot tables: An alternate way to group by exactly two columns.
- Merge: A method to join two dataframes

Structure For Today

Today we'll introduce additional syntax by trying to solve various practical problems on our baby names dataset.

- Goal 1: Find the most popular name in California in 2018 (done in lec 5).
- Goal 2: Find all names that start with J.
- Goal 3: Sort names by length.
- Goal 4: Find the name whose popularity has changed the most.
- Goal 5: Count the number of female and male babies born in each year.

We will also play around with our election dataset.

You'll get a chance to practice this syntax in next week's lab and homework.

`str`

Manipulating String Data

Goal 1: Find all rows where the Name starts with J.

One way using just the Python you learned in 61A / CS 88 would be to use a list comprehension to build a boolean array.

Manipulating String Data

Suppose we want to find all rows where the Name starts with J.

Approach 1: Use list comprehensions from 61A/CS88.

- Create a list of booleans where ith entry is True if ith name starts with J.
- Pass this list to [] or loc[].

	State	Sex	Year	Name	Count
221131	CA	F	2018	Emma	2722
378377	CA	M	2018	Noah	2555
221132	CA	F	2018	Mia	2484
221133	CA	F	2018	Olivia	2456
378378	CA	M	2018	Liam	2405

Manipulating String Data

Suppose we want to find all rows where the Name starts with J.

Approach 1: Use list comprehensions from 61A/CS88.

- Create a list of booleans where ith entry is True if ith name starts with J.
- Pass this list to [] or loc[].

Goal: Fill the list comprehension ??? so that it returns the desired list.

```
starts_with_j = [???]
babynames[starts_with_j].sample(5)
```

	State	Sex	Year	Name	Count
22345	CA	F	1945	Jannie	8
170806	CA	F	2005	Jolina	9
354075	CA	M	2009	Jaylan	8
124886	CA	F	1993	Joseph	26
357780	CA	M	2010	Jess	5

```
[x * 2 for x in [3, 4, 5]]
```

```
[6, 8, 10]
```

Example list comprehension

Manipulating String Data

Suppose we want to find all rows where the Name starts with J.

Approach 1: Use list comprehensions from 61A/CS88.

- Create a list of booleans where ith entry is True if ith name starts with J.
- Pass this list to [] or loc[].

Goal: Write a list comprehension that returns the desired list.

```
starts_with_j = [x.startswith('J') for x in babynames["Name"]]
babynames[starts_with_j].sample(5)
```

	State	Sex	Year	Name	Count
22345	CA	F	1945	Jannie	8
170806	CA	F	2005	Jolina	9
354075	CA	M	2009	Jaylan	8
124886	CA	F	1993	Joseph	26
357780	CA	M	2010	Jess	5

A More Advanced Approach

Approach 1: Use a list comprehensions.

```
j_names = babynames[ [x.startswith('J') for x in babynames["Name"]] ]
```

Approach 2: Use a `str` method from the `Series` class (more on this shortly).

```
j_names = babynames[babynames["Name"].str.startswith('J')]
```

Question: What's better about this second approach?

- **More readable!** Others can understand your code. ← the main great thing
- First one is likely to be less efficient.

Idiomatic Code

Approach 1: Use a list comprehensions.

```
j_names = babynames[ [x.startswith('J') for x in babynames["Name"]] ]
```

Approach 2: Use a `str` method from the `Series` class (more on this shortly).

```
j_names = babynames[babynames["Name"].str.startswith('J')]
```

Terminology note: We say that approach #1 is not **idiomatic**.

- Idiom: "the language peculiar to a people or to a district, community, or class."
- In other words, people from the broader pandas community won't like reading your code if it looks like approach 1.

Str Methods

The `str` methods from the `Series` class have pretty intuitive behavior.

- Won't define formally. Full list at bottom of [this link].

Example: `str.startswith`

```
babynames[babynames["Name"].str.startswith('J')].sample(5)
```

	State	Sex	Year	Name	Count
32151	CA	F	1953	Jewel	11
316051	CA	M	1995	Jarrett	32
344242	CA	M	2006	Josemanuel	26
343769	CA	M	2006	Junior	96
172550	CA	F	2006	Jazmin	582

Str Methods

The `str` methods from the `Series` class have pretty intuitive behavior.

- Won't define formally. Full list at bottom of [this link].

Example: `str.contains`

```
babynames[babynames["Name"].str.contains('ad')].sample(5)
```

	State	Sex	Year	Name	Count
221233	CA	F	2018	Madelyn	336
221518	CA	F	2018	Guadalupe	98
290499	CA	M	1984	Bradford	32
152534	CA	F	2000	Khadija	5
132159	CA	F	1995	Soledad	31

Str Methods

The `str` methods from the `Series` class have pretty intuitive behavior.

- Won't define formally. Full list at bottom of [this link].

Example: `str.split`

```
babynames["Name"].str.split('a').to_frame().head(5)
```

	Name
0	[M, ry]
1	[Helen]
2	[Dorothy]
3	[M, rg, ret]
4	[Fr, nces]

Challenge

Write a line of code that creates a list (or Series or array) of all names that end with "ert".

- Your list should have only one instance of each name!

```
babynames[babynames["Name"].str.startswith('J')].sample(5)
```

	State	Sex	Year	Name	Count
32151	CA	F	1953	Jewel	11
316051	CA	M	1995	Jarrett	32
344242	CA	M	2006	Josemanuel	26
343769	CA	M	2006	Junior	96
172550	CA	F	2006	Jazmin	582

Challenge

Write a line of code that creates a list (or Series or array) of all names that end with "ert".

- Your list should have only one instance of each name!

```
babynames[babynames["Name"].str.endswith("ert")]["Name"].unique()
```

Adding, Modifying, and Removing Columns

Sorting By Length

Goal 3: Sort our baby names by length.

The `sort_values` function does not provide the ability to pass a custom comparison function.

Lots of weird ways to do this, e.g. from last year's Spring 19 lecture:

```
babynames.iloc[[i for i, m in sorted(enumerate(babynames['Name']), key=lambda x: -len(x[1]))]].head(5)
```

Let's see two different ways of doing this that are much nicer.

- Approach 1: Creating a temporary column, then sort on it.
- Approach 2: Creating a sorted index and using loc.

Approach 1: Create a Temporary Column

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

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

Syntax for Column Addition

Adding a column is easy:

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

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

Can also do both steps on one line of code

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

Syntax for Dropping a Column (or Row)

After sorting, we can drop the temporary column.

- The Drop method assumes you're dropping a row by default. Use axis = 1 to drop a column instead.

```
babynames = babynames.drop("name_lengths", axis = 1)
```

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

→

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

Sorting by Arbitrary Functions

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

- Use the Series `.map` method.

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

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

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

A Little More .loc

Sorting By Length

Goal 3: Sort our baby names by length.

The `sort_values` function does not provide the ability to pass a custom comparison function.

Let's see two different ways of doing this that are much nicer.

- Approach 1: Creating a temporary column, then sort on it.
- **Approach 2: Creating a sorted index and using loc.**

Approach 2: Create a Sorted Index and Pass to .loc

Another approach is to take advantage of another feature of .loc.

- `df.loc[idx]` returns the DataFrame in the same order as the given index.
- Only works if the index exactly matches the DataFrame.

Let’s see this approach in action.

Approach 2: Create a Sorted Index and Pass to .loc

Step 1: Create Series of only the lengths of the names.

- This Series will have the same index as the original DataFrame.

```
name_lengths = babynames["Name"].str.len()
name_lengths.head(5)
```

	State	Sex	Year	Name	Count
340748	CA	M	2005	Pedro	442
294382	CA	M	1986	Royce	32
241809	CA	M	1943	Les	7
52043	CA	F	1965	Cristine	18
308476	CA	M	1992	Reyes	24

babynames

```
340748    5
294382    5
241809    3
52043     8
308476    5
Name: Name, dtype: int64
```

name_lengths

Approach 2: Create a Sorted Index and Pass to .loc

Step 2: Sort the series of only name lengths.

- This Series will have an index which is reordered relative to the original dataframe.

```
name_lengths_sorted_by_length = name_lengths.sort_values()
name_lengths_sorted_by_length.head(5)
```

```
340748    5
294382    5
241809    3
52043     8
308476    5
Name: Name, dtype: int64
```

name_lengths

```
111450    2
165876    2
57212     2
307201    2
329408    2
Name: Name, dtype: int64
```

name_lengths_sorted_by_length

Approach 2: Create a Sorted Index and Pass to .loc

Step 3: Pass the sorted index as an argument of .loc to the original dataframe.

```
index_sorted_by_length = name_lengths_sorted_by_length.index
babynames.loc[index_sorted_by_length].head(5)
```

```
111450    2
165876    2
57212     2
307201    2
329408    2
Name: Name, dtype: int64
```

name_lengths_sorted_by_length

	State	Sex	Year	Name	Count
111450	CA	F	1989	Vy	8
165876	CA	F	2004	An	17
57212	CA	F	1968	Jo	80
307201	CA	M	1991	Jc	6
329408	CA	M	2000	Al	7

babynames.loc[index_sorted_by_length]

```
groupby.agg
```

Sorting By Length

Goal 4: Find the names that have changed the most in popularity.

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

- In lecture, let's stay simple and use the AMMD (absolute max/min difference): max(count) - min(count).

Note: This is not a common term. I just made it up.

Example for "Jennifer":

- In 1954, there were only 5.
- In 1972, we hit peak Jennifer. 6,066 Jennifers were born.
- AMMD is 6,066 - 5 = 6,061.

Example: Computing the AMMD for a Given Name

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

```
jennifer_counts = babynames.query("Name == 'Jennifer'")["Count"]
```

```
88492     5812
123809    3003
20807       80
19084       22
42180      868
Name: Count, dtype: int64
```

```
ammd(jennifer_counts)
```

```
6061
```

Approach 1: Getting AMMD for Every Name The Hard Way

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

```
#build dictionary where entry i is the ammd for the given name
#e.g. ammd["jennifer"] should be 6061
ammd_of_babyname_counts = {}
for name in ??:
    counts_of_current_name = babynames[??]["Count"]
    ammd_of_babyname_counts[name] = ammd(counts_of_current_name)

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

Challenge: Try to fill in the code above.

Approach 1: Getting AMMD for Every Name The Hard Way

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

```
#build dictionary where entry i is the ammd for the given name
#e.g. ammd["jennifer"] should be 6061
ammd_of_babyname_counts = {}
for name in sorted(babynames["Name"].unique()):
    counts_of_current_name = babynames[babynames["Name"] == name]["Count"]
    ammd_of_babyname_counts[name] = ammd(counts_of_current_name)

#convert to series
ammd_of_babyname_counts = pd.Series(ammd_of_babyname_counts)
ammd_of_babyname_counts.head(5)
```

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

Approach 2: Using Groupby and Agg

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

- Much simpler, much faster, much more versatile.

Approach 1

```
#build dictionary where entry i is the ammd for the given name
#e.g. ammd["jennifer"] should be 6061
ammd_of_babyname_counts = {}
for name in babynames["Name"].unique()[0:5]:
    counts_of_current_name = babynames[babynames["Name"] == name]["Count"]
    ammd_of_babyname_counts[name] = ammd(counts_of_current_name)

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

```
Aadan        2
Aaden      138
Aadhav       2
Aadhira      4
Aadhya      45
dtype: int64
```

Approach 2

```
babynames.groupby("Name").agg(ammd)
```

Name	Year	Count
Aadan	6	2
Aaden	11	138
Aadhav	3	2
Aadhira	1	4
Aadhya	11	45

Attendance Question: Check Your groupBy Understanding

Approach 2 generated two columns, Year and Count.

What do you think the Year column represents?

A. The number of years a name appeared.
B. The difference between the earliest and latest year a name appeared.
C. It has no meaning because our code was only designed to work with counts.
D. Not sure.

Approach 2

```
babynames.groupby("Name").agg(ammd)
```

Name	Year	Count
Aadan	6	2
Aaden	11	138
Aadhav	3	2
Aadhira	1	4
Aadhya	11	45

Attendance Question:

Approach 2 generated two columns, Year and Count.

What do you think the Year column represents?

A. The number of years a name appeared.

B. The difference between the earliest and latest year a name appeared.

C. It has no meaning because our code was only designed to work with counts.

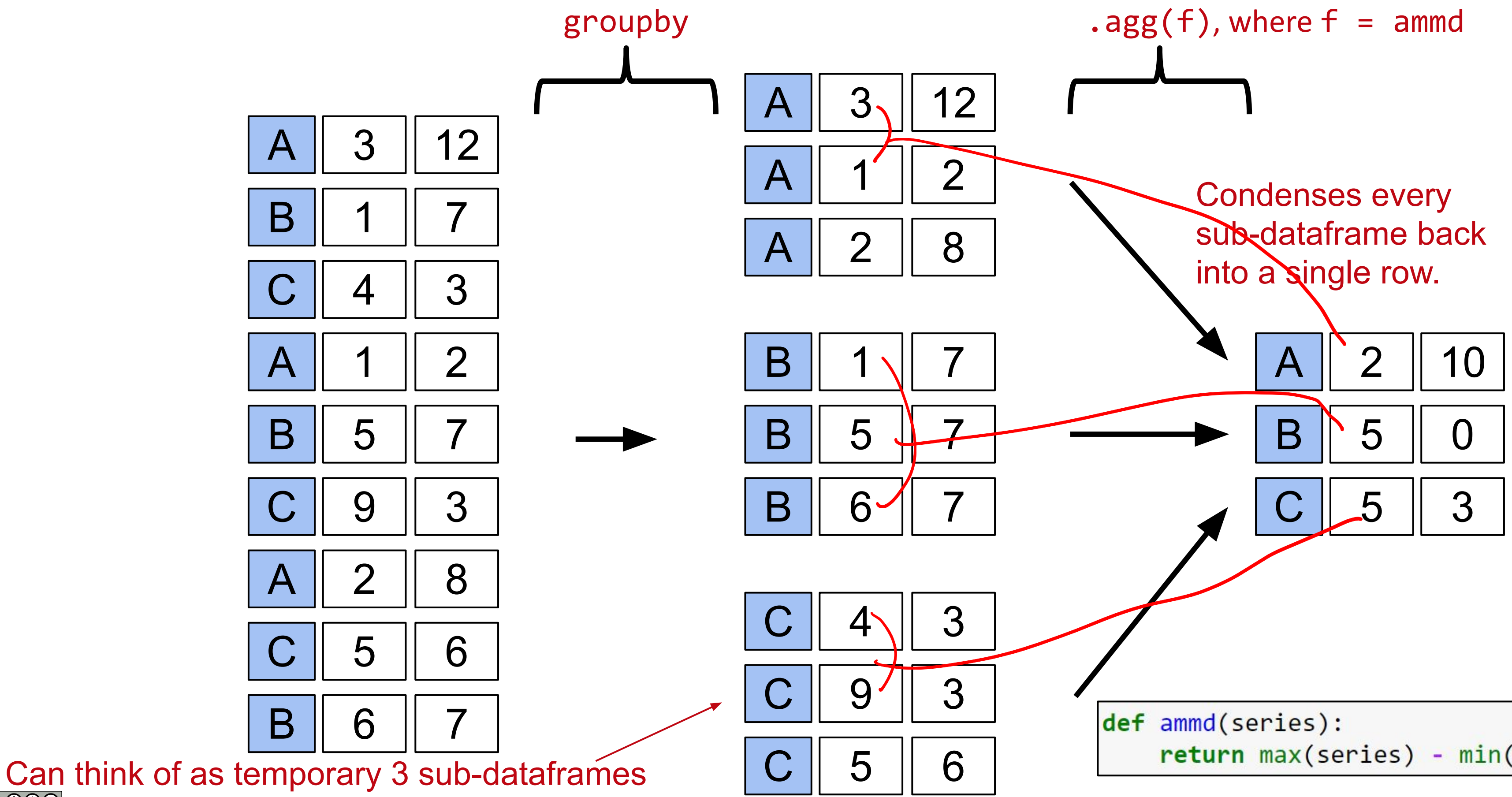
D. Not sure.

Approach 2

```
babynames.groupby("Name").agg(ammd)
```

Name	Year	Count
Aadan	6	2
Aaden	11	138
Aadhav	3	2
Aadhira	1	4
Aadhya	11	45

DataFrame groupby.agg Visually



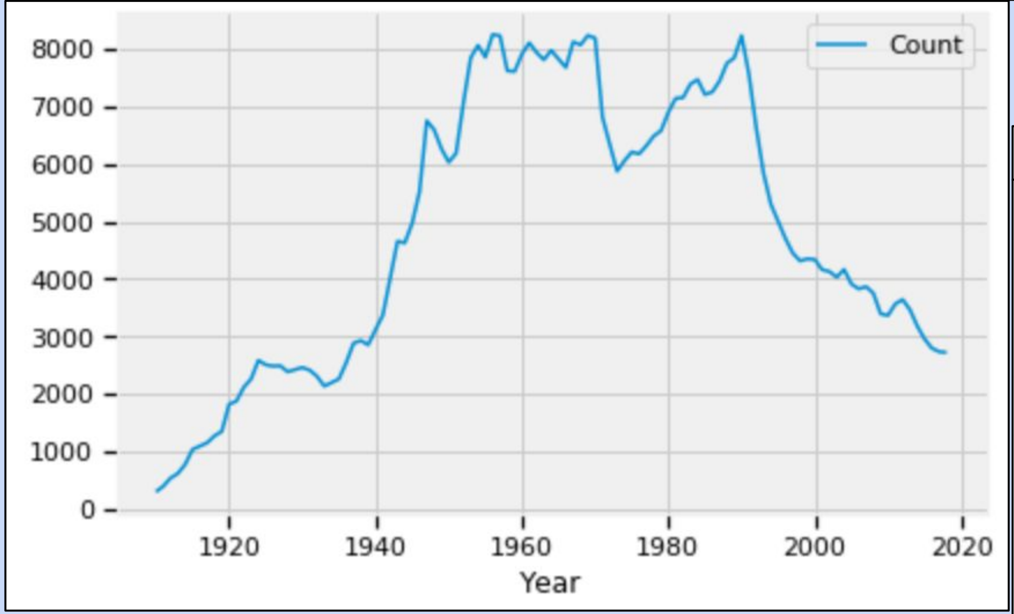

```python
def ammd(series):
    return max(series) - min(series)
```

Some groupby.agg puzzles

groupby Puzzle #1

Below, we show the result of the given code. What does it mean?

```
babynames.groupby("Year").agg(ammd).plot()
```



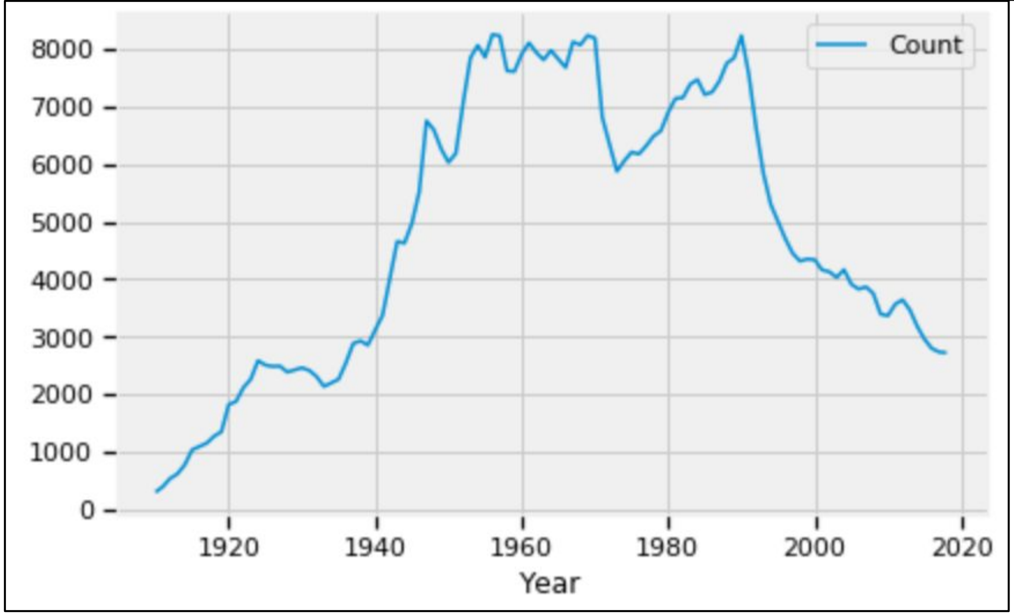

```
babynames.groupby("Year").agg(ammd).head(5)
```

Year	Count
1910	290
1911	385
1912	529
1913	609
1914	768

groupby Puzzle #1

Below, we show the result of the given code. What does it mean?

```
babynames.groupby("Year").agg(ammd).plot()
```



Your answer:

groupby Puzzle #2

Be careful when using groupby. Consider the results on our elections table:

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

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	2016	Woodrow Wilson	69498516	win	61.344703

groupby Puzzle #2

Why does the table seem to claim that Woodrow Wilson won the presidency in 2016?

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

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	2016	Woodrow Wilson	69498516	win	61.344703

groupby Puzzle #2

Why does the table seem to claim that Woodrow Wilson won the presidency in 2016?

Every column is calculated independently! Among Democrats:

- Last year they ran: 2016
- Alphabetically latest candidate name: Woodrow Wilson
- Highest % of vote: 61.34

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

	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	2016	Woodrow Wilson	69498516	win	61.344703

groupby Puzzle #3

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

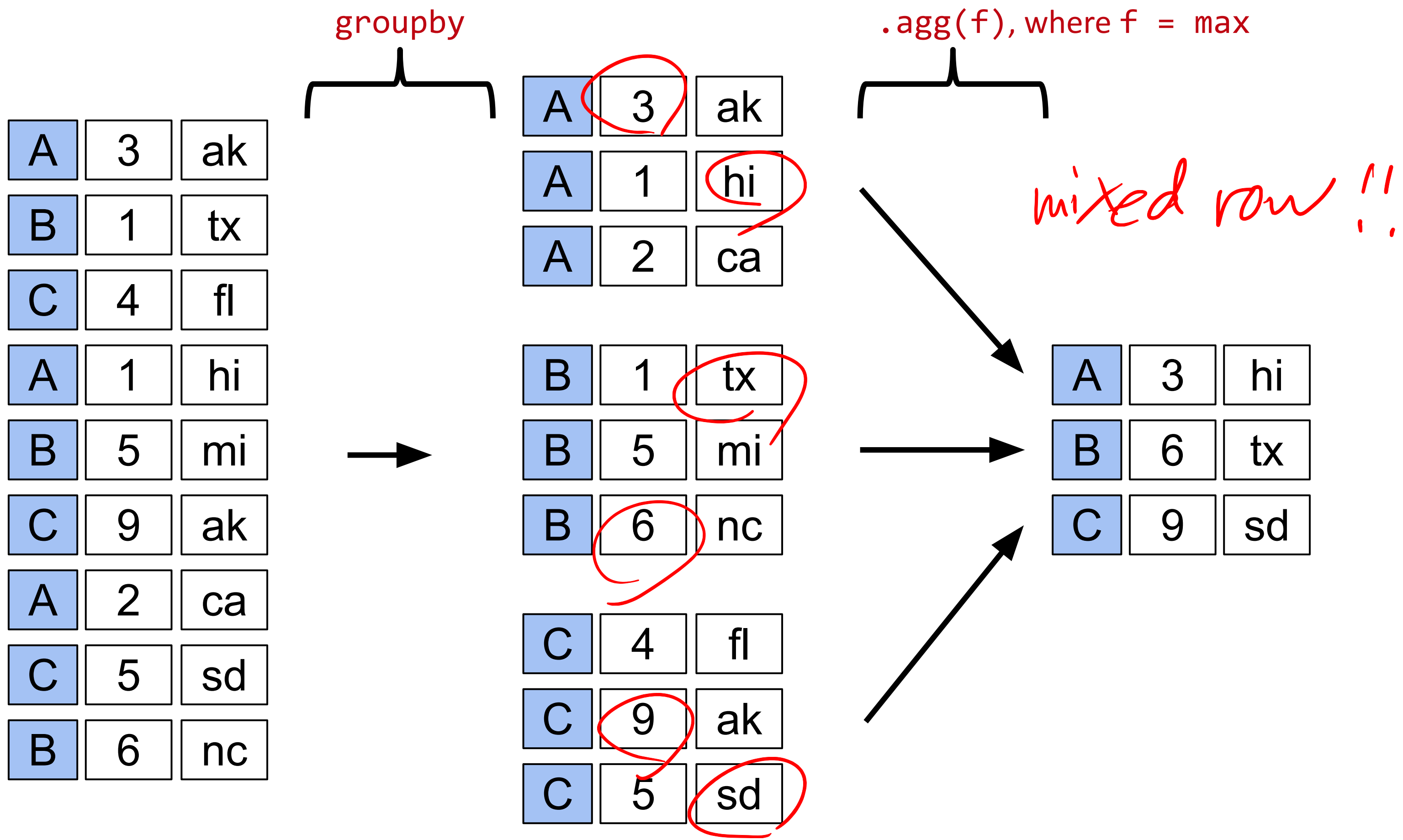
C	4	fl
C	9	ak
C	5	sd

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

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

What will go in the ??

groupby Puzzle #3



Puzzle #4

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

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

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

Puzzle #4

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

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

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

Puzzle #4

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

- First sort the DataFrame so that rows are in ascending order of %.

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

- Then group by Party and take the 0th member of each series.

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

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

Quick Note

If this type of programming seems scary, don't worry, you'll get used to it.

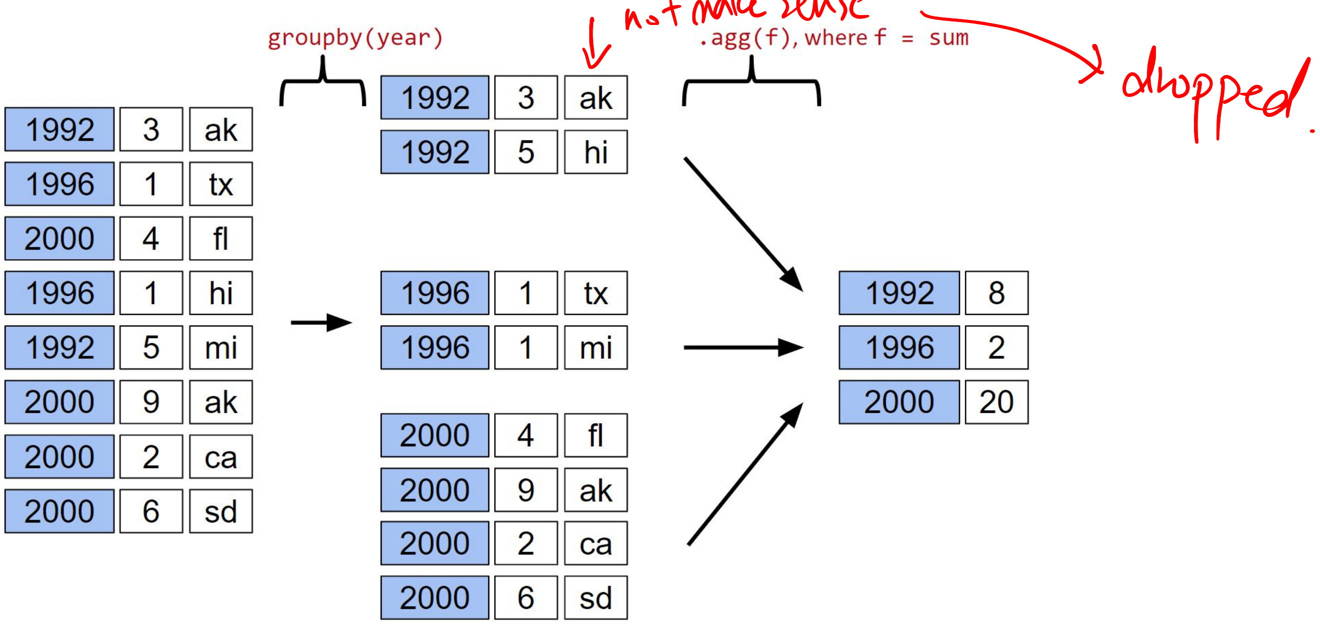
- Very different than the procedural style that you may be used to in Java, Matlab, Python, etc.
- Has a more declarative/SQL like feel.

Other groupby Features

Revisiting groupby.agg

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

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



Raw groupby Objects

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

- It is not a DataFrame!

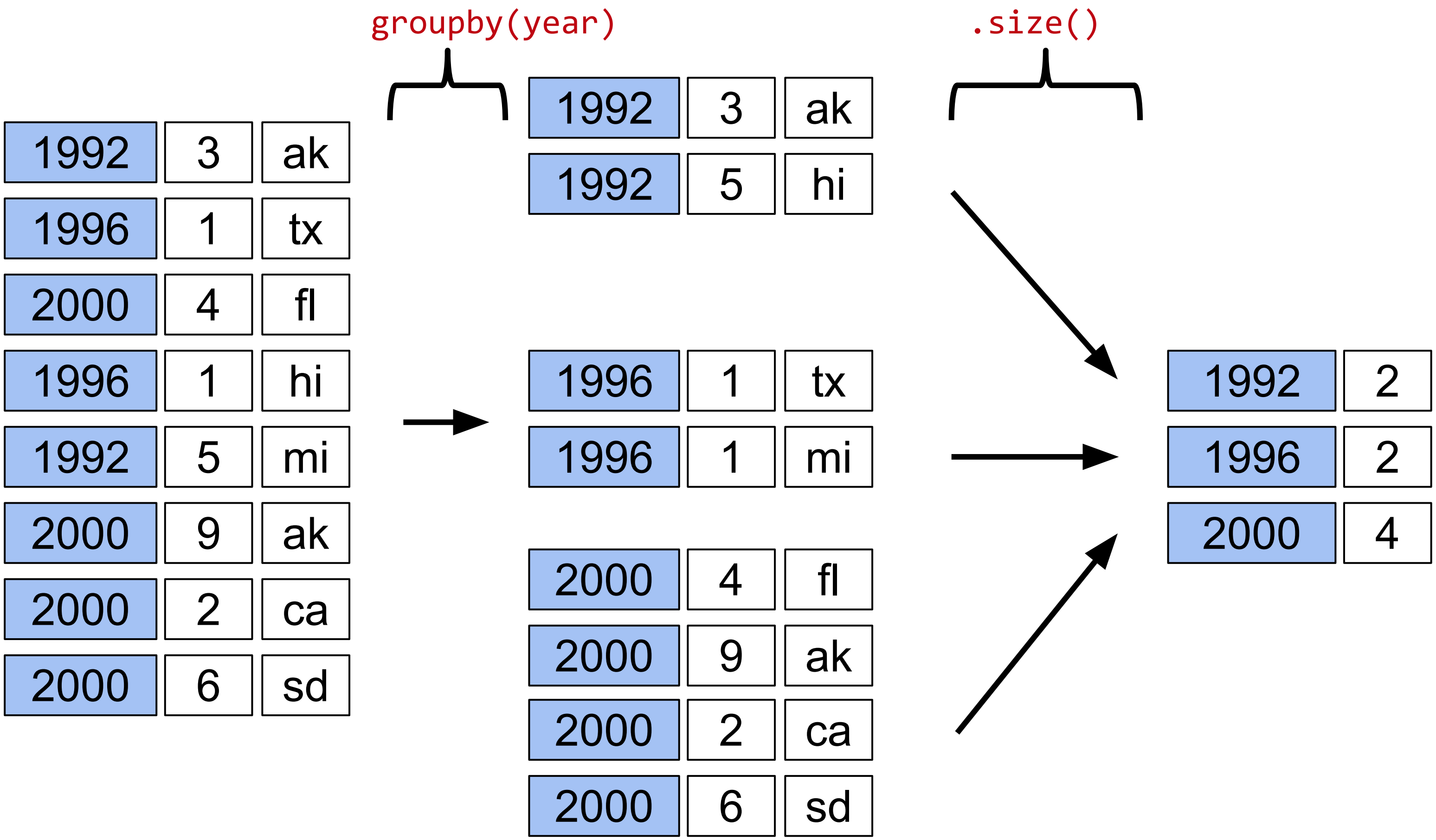
```
grouped_by_year = elections_sorted_by_percent.query("Year > 1950").groupby("Year")
type(grouped_by_year)
```

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

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

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

groupby.size()

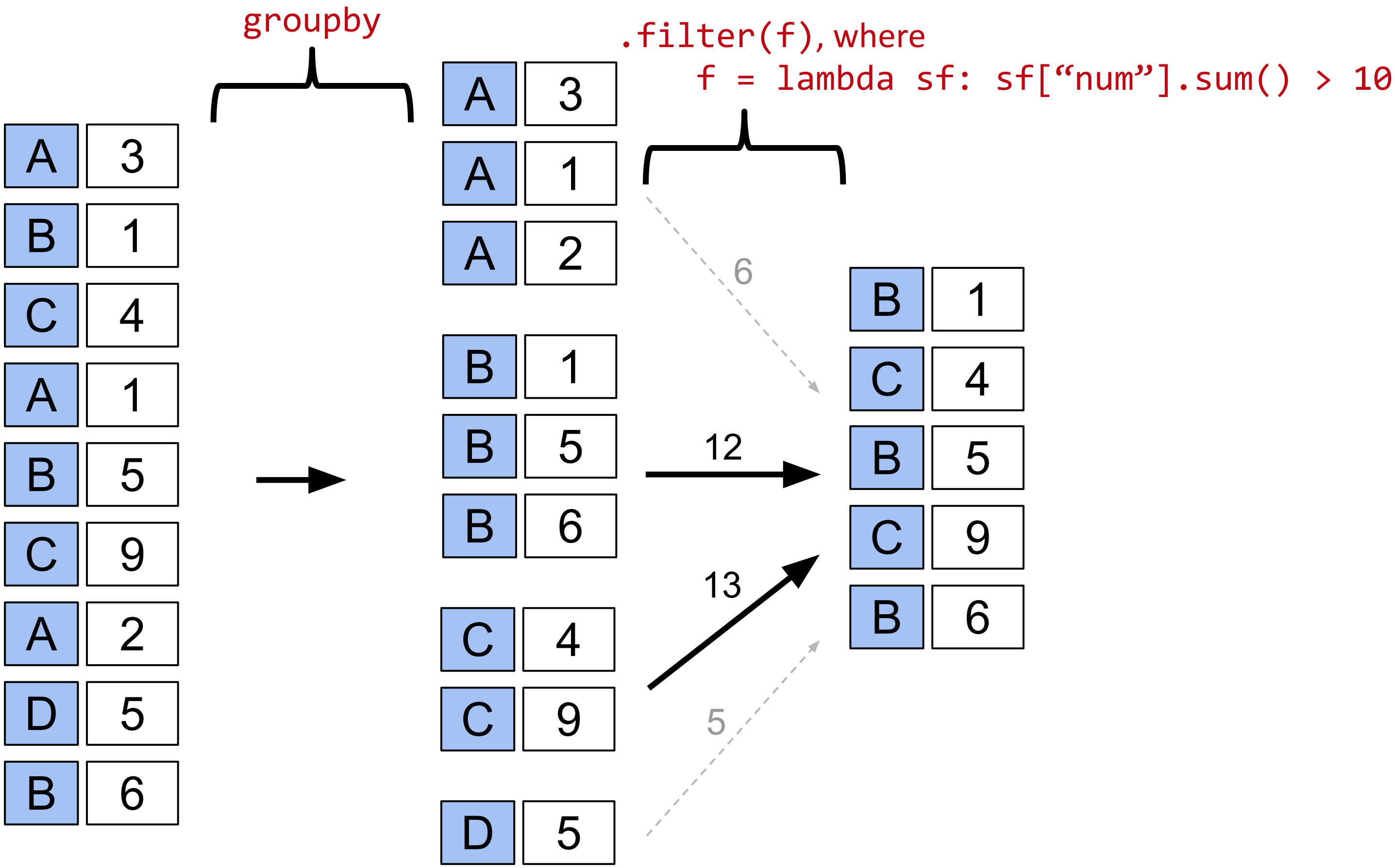


Filtering by Group

Another common use for groups is to filter data.

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

groupby.filter



groupby.sum(), groupby.mean(), groupby.max(), etc...

For common operations, rather than saying e.g. groupby.agg(sum), we can instead do groupby.sum():

```
elections.groupby("Year").agg(sum).head()
```

```
elections.groupby("Year").sum().head()
```

```
elections.groupby("Year").agg(max).head()
```

```
elections.groupby("Year").max().head()
```

groupby([]) and Pivot Tables

Grouping by Multiple Columns

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

Example:

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

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

Note: Resulting DataFrame is multi-indexed. That is, its index has multiple dimensions. Will explore next week outside of lecture.

Pivot Tables

A more natural approach is to use a pivot table (like we saw in data 8).

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

Sex	F	M
Year		
1910	295	237
1911	390	214
1912	534	501
1913	584	614
1914	773	769
1915	998	1033

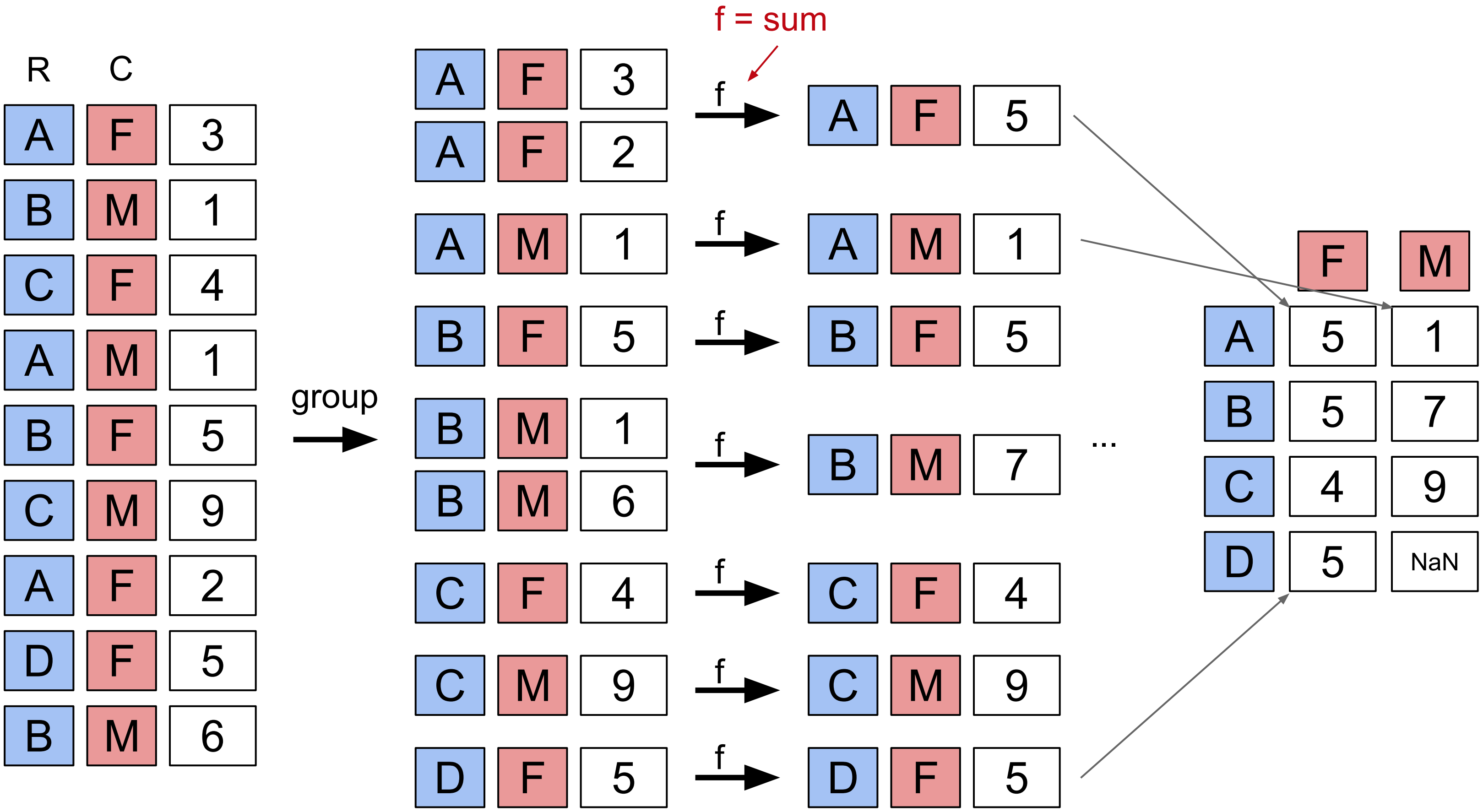
groupby(["Year", "Sex"]) vs. pivot_table

The pivot table more naturally represents our data.

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

Sex	**F**	**M**
Year		
1910	295	237
1911	390	214
1912	534	501
1913	584	614
1914	773	769
1915	998	1033

Pivot Tables



Joins

merge

- Basic syntax for joining two dataframes `df` and `df2`
  - `df.merge(df2)`
- Output is another dataframe
- Pandas also has a method called `join`
  - limited version of `merge`
- We will only use `merge`

Types of Joins

- As in SQL:
  - inner, outer, left, and right joins
- Typical usage

```
df.merge(df2,
         how = “inner”,
         left_on = “column_label_in_df”,
         right_on = “column_label_in_df2”)
```

- “inner” can be replaced by “outer” or “left” or “right”

New Syntax / Concept Summary

- Operations on String series, e.g. babynames[“Name”].str.startswith()
- Creating and dropping columns.
  - Creating temporary columns is often convenient for sorting.
- Passing an index as an argument to loc.
  - Useful as an alternate way to sort a dataframe.
- Groupby: Output of .groupby(“Name”) is a DataFrameGroupBy object. Condense back into a DataFrame or Series with:
  - groupby.agg
  - groupby.size
  - groupby.filter
  - and more...
- Pivot tables: An alternate way to group by exactly two columns.
- Merge: A method to join two dataframes

Extra Slides from Fa18

groupby Key Concepts

If we call `groupby` on a Series:

- The resulting output is a `SeriesGroupBy` object.
- The Series that are passed as arguments to `groupby` must share an index with the calling Series.

```
percent_grouped_by_party = df['%'].groupby(df['Party'])
percent_grouped_by_party.groups
```

```
{'Democratic': Int64Index([1, 4, 6, 7, 10, 13, 15, 17, 19, 21], dtype='int64'),
 'Independent': Int64Index([2, 9, 12], dtype='int64'),
 'Republican': Int64Index([0, 3, 5, 8, 11, 14, 16, 18, 20, 22], dtype='int64')}
```

`SeriesGroupBy` objects can then be aggregated back into a `Series` using an aggregation method.

```
percent_grouped_by_party.mean()
```

→

```
Party
Democratic      46.53
Independent     11.30
Republican      47.86
Name: %, dtype: float64
```

groupby Key Concepts

If we call groupby on a DataFrame:

- The resulting output is a DataFrameGroupBy object.

DataFrameGroupBy objects can then be aggregated back into a DataFrame or a Series using an aggregation method.

```
everything_grouped_by_party = df.groupby('Party')
```

```
{'Democratic': Int64Index([1, 4, 6, 7, 10, 13, 15, 17, 19, 21], dtype='int64'),
 'Independent': Int64Index([2, 9, 12], dtype='int64'),
 'Republican': Int64Index([0, 3, 5, 8, 11, 14, 16, 18, 20, 22], dtype='int64')}
```

```
everything_grouped_by_party.mean()
```

→

	%	Year
Party		
Democratic	46.53	1998.000000
Independent	11.30	1989.333333
Republican	47.86	1998.000000

groupby and agg

Most of the built-in handy aggregation methods are just shorthand for a universal aggregation method called agg.

- Example, `.mean()` is just `.agg(np.mean)`.

```
everything_grouped_by_party = df.groupby('Party')
```

```
{'Democratic': Int64Index([1, 4, 6, 7, 10, 13, 15, 17, 19, 21], dtype='int64'),
 'Independent': Int64Index([2, 9, 12], dtype='int64'),
 'Republican': Int64Index([0, 3, 5, 8, 11, 14, 16, 18, 20, 22], dtype='int64')}
```

```
everything_grouped_by_party.mean()
```

```
everything_grouped_by_party.agg(np.mean)
```

	%	Year
Party		
Democratic	46.53	1998.000000
Independent	11.30	1989.333333
Republican	47.86	1998.000000

The `MultiIndex`

If we group a Series (or DataFrame) by multiple Series and then perform an aggregation operation, the resulting Series (or Dataframe) will have a `MultiIndex`.

```
everything_grouped_by_party_and_result = df.groupby([df['Party'], df['Result']])
everything_grouped_by_party_and_result.mean()
```

The resulting DataFrame has:

- Two columns “%” and “Year”
- A MultiIndex, where results of aggregate function are indexed by Party first, then Result.

		%	Year
Party	**Result**		
Democratic	**loss**	44.850000	1995.333333
	win	49.050000	2002.000000
Independent	**loss**	11.300000	1989.333333
Republican	**loss**	42.750000	2002.000000
	win	51.266667	1995.333333