```
In [1]: from dsc80_utils import *
```

Lecture 4 – Simpson's Paradox, Joining, and Transforming

DSC 80, Fall 2025

Announcements 📣

- Project 1 checkpoint due **this Thursday**.
- Lab 2 is due on **Monday**.
- Project 1 is due **next Thursday**.

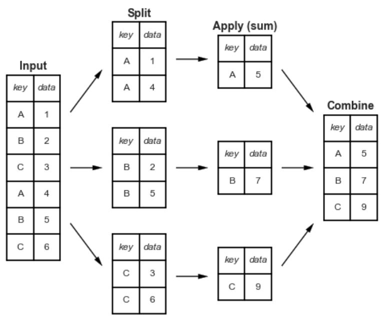
Agenda

- Transforming and filtration
- Distributions.
- Simpson's paradox.
- Merging.
  - Many-to-one & many-to-many joins.
- Transforming.
  - The price of `apply`.
- Other data representations.

Other `DataFrameGroupBy` methods

Split-apply-combine, revisited

When we introduced the split-apply-combine pattern, the "apply" step involved **aggregation** – our final DataFrame had one row for each group.



Instead of aggregating during the apply step, we could instead perform a:

- **Transformation**, in which we perform operations to every value within each group.
- **Filtration**, in which we keep only the groups that satisfy some condition.

Transformations

Suppose we want to convert the `'body_mass_g'` column to to z-scores (i.e. standard units):

$$z(x_i) = \frac{x_i - \text{mean of } x}{\text{SD of } x}$$

```
In [2]: import seaborn as sns
        penguins = sns.load_dataset('penguins').dropna()
```

```
In [3]: def z_score(x):
            return (x - x.mean()) / x.std(ddof=0)
```

```
In [4]: z_score(penguins['body_mass_g'])
```

```
Out[4]: 0     -0.57
        1     -0.51
        2     -1.19
               ...
        341    1.92
        342    1.23
        343    1.48
        Name: body_mass_g, Length: 333, dtype: float64
```

Transformations within groups

- Now, what if we wanted the z-score within each group?
- To do so, we can use the `transform` method on a `DataFrameGroupBy` object. The `transform` method takes in a function, which itself takes in a Series and returns a new Series.
- A transformation produces a DataFrame or Series of the same size – it is **not** an aggregation!

```
In [5]: z_mass = (penguins
                  .groupby('species')
                  ['body_mass_g']
                  .transform(z_score))
        z_mass
```

```
Out[5]: 0      0.10
        1      0.21
        2     -1.00
               ...
        341    1.32
        342    0.22
        343    0.62
        Name: body_mass_g, Length: 333, dtype: float64
```

```
In [6]: penguins.assign(z_mass=z_mass)
```

Out[6]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_ma
0	Adelie	Torgersen	39.1	18.7	181.0	3
1	Adelie	Torgersen	39.5	17.4	186.0	3
2	Adelie	Torgersen	40.3	18.0	195.0	3
...	...	...	...	...	...	
341	Gentoo	Biscoe	50.4	15.7	222.0	5
342	Gentoo	Biscoe	45.2	14.8	212.0	5
343	Gentoo	Biscoe	49.9	16.1	213.0	5

333 rows × 8 columns

```
In [7]: display_df(penguins.assign(z_mass=z_mass), rows=8)
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mas
0	Adelie	Torgersen	39.1	18.7	181.0	375
1	Adelie	Torgersen	39.5	17.4	186.0	380
2	Adelie	Torgersen	40.3	18.0	195.0	325
4	Adelie	Torgersen	36.7	19.3	193.0	345
...	...	...	...	...	...	
340	Gentoo	Biscoe	46.8	14.3	215.0	485
341	Gentoo	Biscoe	50.4	15.7	222.0	575
342	Gentoo	Biscoe	45.2	14.8	212.0	520
343	Gentoo	Biscoe	49.9	16.1	213.0	540

333 rows × 8 columns

Note that above, penguin 340 has a larger `'body_mass_g'` than penguin 0, but a lower `'z_mass'`.

```
In [8]: penguins.assign(z_mass=z_mass).loc[[0, 340], ['body_mass_g', 'z_mass', 'spec
```

Out[8]:

	body_mass_g	z_mass	species
0	3750.0	0.10	Adelie
340	4850.0	-0.49	Gentoo

- Penguin 0 has an above average `'body_mass_g'` among `'Adelie'` penguins.
- Penguin 340 has a below average `'body_mass_g'` among `'Gentoo'` penguins. Remember from earlier that the average `'body_mass_g'` of `'Gentoo'` penguins is much higher than for other species.

In [9]:
```
penguins.groupby('species')['body_mass_g'].mean()
```

Out[9]:
```
species
Adelie       3706.16
Chinstrap    3733.09
Gentoo       5092.44
Name: body_mass_g, dtype: float64
```

Filtering groups

- To keep only the groups that satisfy a particular condition, use the `filter` method on a `DataFrameGroupBy` object.
- The `filter` method takes in a function, which itself takes in a DataFrame/Series and return a single Boolean. The result is a new DataFrame/Series with only the groups for which the filter function returned `True`.

For example, suppose we want only the `'species'` whose average `'bill_length_mm'` is above 39.

In [10]:
```
(penguins
 .groupby('species')
 .filter(lambda df: df['bill_length_mm'].mean() > 39)
)
```

Out[10]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mas
152	Chinstrap	Dream	46.5	17.9	192.0	35(
153	Chinstrap	Dream	50.0	19.5	196.0	39(
154	Chinstrap	Dream	51.3	19.2	193.0	36!
...	...	...	...	...	...	
341	Gentoo	Biscoe	50.4	15.7	222.0	57!
342	Gentoo	Biscoe	45.2	14.8	212.0	52(
343	Gentoo	Biscoe	49.9	16.1	213.0	54(

187 rows × 7 columns

No more `'Adelie'` s!

Or, as another example, suppose we only want `'species'` with at least 100 penguins:

In [11]:
```
(penguins
 .groupby('species')
 .filter(lambda df: df.shape[0] > 100)
)
```

Out[11]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_ma
0	Adelie	Torgersen	39.1	18.7	181.0	3
1	Adelie	Torgersen	39.5	17.4	186.0	3
2	Adelie	Torgersen	40.3	18.0	195.0	3
...	...	...	...	...	...	
341	Gentoo	Biscoe	50.4	15.7	222.0	5
342	Gentoo	Biscoe	45.2	14.8	212.0	5
343	Gentoo	Biscoe	49.9	16.1	213.0	5

265 rows × 7 columns

No more `'Chinstrap'` s!

Grouping with multiple columns

When we group with multiple columns, one group is created for **every unique combination** of elements in the specified columns.

In [12]:
```
penguins
```

Out[12]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_ma
0	Adelie	Torgersen	39.1	18.7	181.0	3
1	Adelie	Torgersen	39.5	17.4	186.0	3
2	Adelie	Torgersen	40.3	18.0	195.0	3
...	...	...	...	...	...	
341	Gentoo	Biscoe	50.4	15.7	222.0	5
342	Gentoo	Biscoe	45.2	14.8	212.0	5
343	Gentoo	Biscoe	49.9	16.1	213.0	5

333 rows × 7 columns

In [13]:
```python
species_and_island = (
    penguins
    .groupby(['species', 'island'])
    [['bill_length_mm', 'body_mass_g']]
    .mean()
)
species_and_island
```

Out[13]:

		bill_length_mm	body_mass_g
species	**island**		
Adelie	**Biscoe**	38.98	3709.66
	Dream	38.52	3701.36
	Torgersen	39.04	3708.51
Chinstrap	**Dream**	48.83	3733.09
Gentoo	**Biscoe**	47.57	5092.44

Grouping and indexes

- The `groupby` method creates an index based on the specified columns.
- When grouping by multiple columns, the resulting DataFrame has a `MultiIndex`.
- Advice: When working with a `MultiIndex`, use `reset_index` or set `as_index=False` in `groupby`.

In [14]:
```python
species_and_island
```

Out[14]:

species	island	bill_length_mm	body_mass_g
Adelie	Biscoe	38.98	3709.66
	Dream	38.52	3701.36
	Torgersen	39.04	3708.51
Chinstrap	Dream	48.83	3733.09
Gentoo	Biscoe	47.57	5092.44

In [15]:
```
species_and_island['body_mass_g']
```

Out[15]:
```
species    island
Adelie     Biscoe       3709.66
           Dream        3701.36
           Torgersen    3708.51
Chinstrap  Dream        3733.09
Gentoo     Biscoe       5092.44
Name: body_mass_g, dtype: float64
```

In [16]:
```
species_and_island.loc['Adelie']
```

Out[16]:

island	bill_length_mm	body_mass_g
Biscoe	38.98	3709.66
Dream	38.52	3701.36
Torgersen	39.04	3708.51

In [17]:
```
species_and_island.loc[('Adelie', 'Torgersen')]
```

Out[17]:
```
bill_length_mm      39.04
body_mass_g       3708.51
Name: (Adelie, Torgersen), dtype: float64
```

In [18]:
```
species_and_island.reset_index()
```

Out[18]:

	species	island	bill_length_mm	body_mass_g
0	Adelie	Biscoe	38.98	3709.66
1	Adelie	Dream	38.52	3701.36
2	Adelie	Torgersen	39.04	3708.51
3	Chinstrap	Dream	48.83	3733.09
4	Gentoo	Biscoe	47.57	5092.44

In [19]:
```
(penguins
 .groupby(['species', 'island'], as_index=False)
 [['bill_length_mm', 'body_mass_g']]
 .mean()
)
```

Out[19]:

	species	island	bill_length_mm	body_mass_g
0	Adelie	Biscoe	38.98	3709.66
1	Adelie	Dream	38.52	3701.36
2	Adelie	Torgersen	39.04	3708.51
3	Chinstrap	Dream	48.83	3733.09
4	Gentoo	Biscoe	47.57	5092.44

Question 🤔

Find the most popular `Male` and `Female` baby `Name` for each `Year` in `baby`. **Exclude** `Year`s where there were fewer than 1 million births recorded.

In [20]:
```
baby_path = Path('data') / 'baby.csv'
baby = pd.read_csv(baby_path)
baby
```

Out[20]:

	Name	Sex	Count	Year
0	Liam	M	20456	2022
1	Noah	M	18621	2022
2	Olivia	F	16573	2022
...	...	...	...	...
2085155	Wright	M	5	1880
2085156	York	M	5	1880
2085157	Zachariah	M	5	1880

2085158 rows × 4 columns

In [21]:
```
# your code here...
```

Pivot tables using the `pivot_table` method

Pivot tables: an extension of grouping

Pivot tables are a compact way to display tables for humans to read:

Sex	F	M
Year		
2018	1698373	1813377
2019	1675139	1790682
2020	1612393	1721588
2021	1635800	1743913
2022	1628730	1733166

- Notice that each value in the table is a sum over the counts, split by year and sex.
- **You can think of pivot tables as grouping using two columns, then "pivoting" one of the group labels into columns.**

`pivot_table`

The `pivot_table` (not `pivot`!) DataFrame method aggregates a DataFrame using two columns. To use it:

```
df.pivot_table(index=index_col,
               columns=columns_col,
               values=values_col,
               aggfunc=func)
```

The resulting DataFrame will have:

- One row for every unique value in `index_col`.
- One column for every unique value in `columns_col`.
- Values determined by applying `func` on values in `values_col`.

In [22]:
```
last_5_years = baby.query('Year >= 2018')
last_5_years
```

Out[22]:

	Name	Sex	Count	Year
0	Liam	M	20456	2022
1	Noah	M	18621	2022
2	Olivia	F	16573	2022
...	...	...	...	...
159444	Zyrie	M	5	2018
159445	Zyron	M	5	2018
159446	Zzyzx	M	5	2018

159447 rows × 4 columns

In [23]:
```
last_5_years.pivot_table(
    index='Year',
    columns='Sex',
    values='Count',
    aggfunc='sum',
)
```

Out[23]:

Sex	**F**	**M**
Year		
2018	1698373	1813377
2019	1675139	1790682
2020	1612393	1721588
2021	1635800	1743913
2022	1628730	1733166

In [24]:
```
# compare to the top-left element of the table
last_5_years[
    (last_5_years['Year'] == 2018)
    &
    (last_5_years['Sex'] == 'F')
]['Count'].sum()
```

Out[24]:
```
np.int64(1698373)
```

In [25]:
```
# compare to the top-right element of the table
last_5_years[
    (last_5_years['Year'] == 2018)
    &
    (last_5_years['Sex'] == 'M')
]['Count'].sum()
```

Out[25]:
```
np.int64(1813377)
```

In [26]:
```
# Look at the similarity to the snippet above!
(last_5_years
 .groupby(['Year', 'Sex'])
 [['Count']]
 .sum()
)
```

Out[26]:

		Count
Year	**Sex**	
2018	**F**	1698373
	M	1813377
2019	**F**	1675139
...	**...**	...
2021	**M**	1743913
2022	**F**	1628730
	M	1733166

10 rows × 1 columns

Example

Find the number of penguins per `'island'` and `'species'`.

In [27]:
```
penguins
```

Out[27]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_ma
0	Adelie	Torgersen	39.1	18.7	181.0	3
1	Adelie	Torgersen	39.5	17.4	186.0	3
2	Adelie	Torgersen	40.3	18.0	195.0	3
...	...	...	...	...	...	
341	Gentoo	Biscoe	50.4	15.7	222.0	5
342	Gentoo	Biscoe	45.2	14.8	212.0	5
343	Gentoo	Biscoe	49.9	16.1	213.0	5

333 rows × 7 columns

In [28]:
```
penguins.pivot_table(
    index='species',
    columns='island',
    values='bill_length_mm', # Choice of column here doesn't actually matter
    aggfunc='count',
)
```

Out[28]:

island	Biscoe	Dream	Torgersen
species			
Adelie	44.0	55.0	47.0
Chinstrap	NaN	68.0	NaN
Gentoo	119.0	NaN	NaN

Note that there is a `NaN` at the intersection of `'Biscoe'` and `'Chinstrap'`, because there were no Chinstrap penguins on Biscoe Island.

We can either use the `fillna` method afterwards or the `fill_value` argument to fill in `NaN` s.

In [29]:

```
penguins.pivot_table(
    index='species',
    columns='island',
    values='bill_length_mm',
    aggfunc='count',
    fill_value=0,
)
```

Out[29]:

island	Biscoe	Dream	Torgersen
species			
Adelie	44	55	47
Chinstrap	0	68	0
Gentoo	119	0	0

Granularity, revisited

Take another look at the pivot table from the previous slide. Each row of the original `penguins` DataFrame represented a single penguin, and each column represented features of the penguins.

What is the granularity of the DataFrame below?

In [30]:

```
penguins.pivot_table(
    index='species',
    columns='island',
    values='bill_length_mm',
    aggfunc='count',
    fill_value=0,
)
```

Out[30]:

island	Biscoe	Dream	Torgersen
species			
Adelie	44	55	47
Chinstrap	0	68	0
Gentoo	119	0	0

Reshaping

- `pivot_table` reshapes DataFrames from "long" to "wide".
- Other DataFrame reshaping methods:
  - `melt` : Un-pivots a DataFrame. Very useful in data cleaning.
  - `pivot` : Like `pivot_table` , but doesn't do aggregation.
  - `stack` : Pivots multi-level columns to multi-indices.
  - `unstack` : Pivots multi-indices to columns.
  - Google and the documentation are your friends!

Distributions

Let's compute probabilities using an easier way.

We'll start by using the `pivot_table` method to recreate the DataFrame shown below.

sex	Female	Male
species		
Adelie	73	73
Chinstrap	34	34
Gentoo	58	61

Joint distribution

When using `aggfunc='count'` , a pivot table describes the **joint distribution** of two categorical variables. This is also called a **contingency table**.

In [31]:
```
counts = penguins.pivot_table(
    index='species',
    columns='sex',
    values='body_mass_g',
    aggfunc='count',
    fill_value=0,
)
counts
```

Out[31]:

sex	Female	Male
species		
Adelie	73	73
Chinstrap	34	34
Gentoo	58	61

We can normalize the DataFrame by dividing by the total number of penguins. The resulting numbers can be interpreted as **probabilities** that a randomly selected penguin from the dataset belongs to a given combination of species and sex.

In [32]:
```
joint = counts / counts.sum().sum()
joint
```

Out[32]:

sex	Female	Male
species		
Adelie	0.22	0.22
Chinstrap	0.10	0.10
Gentoo	0.17	0.18

Marginal probabilities

If we sum over one of the axes, we can compute **marginal probabilities**, i.e. unconditional probabilities.

In [33]:
```
joint
```

Out[33]:

sex	Female	Male
species		
Adelie	0.22	0.22
Chinstrap	0.10	0.10
Gentoo	0.17	0.18

In [34]:
```
# Recall, joint.sum(axis=0) sums across the rows,
# which computes the sum of the **columns**.
joint.sum(axis=0)
```

Out[34]:
```
sex
Female    0.5
Male      0.5
dtype: float64
```

```
In [35]: joint.sum(axis=1)
```

```
Out[35]: species
         Adelie       0.44
         Chinstrap    0.20
         Gentoo       0.36
         dtype: float64
```

For instance, the second Series tells us that a randomly selected penguin has a 0.36 chance of being of species `'Gentoo'`.

Conditional probabilities

Using `counts`, how might we compute conditional probabilities like

$$P(\text{species } = \text{"Adelie"} \mid \text{sex } = \text{"Female"})?$$

```
In [36]: counts
```

Out[36]:

sex	Female	Male
species		
Adelie	73	73
Chinstrap	34	34
Gentoo	58	61

$$P(\text{species} = c \mid \text{sex} = x) = \frac{\#\,(\text{species} = c \text{ and sex} = x)}{\#\,(\text{sex} = x)}$$

▶ ➡️ Click **here** to see more of a derivation.

Answer: To find conditional probabilities of `'species'` **given** `'sex'`, divide by **column sums**. To find conditional probabilities of `'sex'` **given** `'species'`, divide by **row sums**.

Conditional probabilities

To find conditional probabilities of `'species'` **given** `'sex'`, divide by **column sums**.
To find conditional probabilities of `'sex'` **given** `'species'`, divide by **row sums**.

```
In [37]: counts
```

Out[37]:

sex	Female	Male
species		
Adelie	73	73
Chinstrap	34	34
Gentoo	58	61

In [38]:
```
counts.sum(axis=0)
```

Out[38]:
```
sex
Female    165
Male      168
dtype: int64
```

The conditional distribution of `'species'` **given** `'sex'` is below. Note that in this new DataFrame, the `'Female'` and `'Male'` columns each sum to 1.

In [39]:
```
counts / counts.sum(axis=0)
```

Out[39]:

sex	Female	Male
species		
Adelie	0.44	0.43
Chinstrap	0.21	0.20
Gentoo	0.35	0.36

For instance, the above DataFrame tells us that the probability that a randomly selected penguin is of `'species'` `'Adelie'` **given** that they are of `'sex'` `'Female'` is 0.442424.

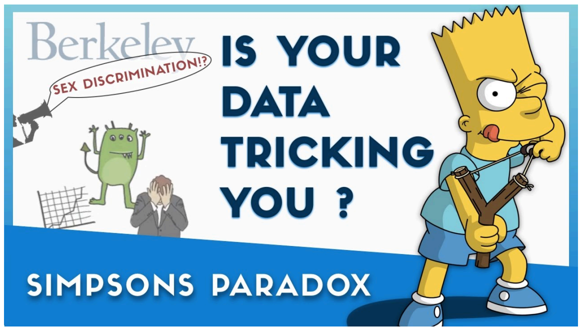
Question 🤔

Find the conditional distribution of `'sex'` given `'species'`.

Hint: Use `.T`.

In [40]:
```
# Your code goes here.
```

Simpson's paradox



Example: Grades

- Two students, Lisa and Bart, just finished their first year at UCSD. They both took a different number of classes in Fall, Winter, and Spring.
- Each quarter, Lisa had a higher GPA than Bart.
- But Bart has a higher overall GPA.
- How is this possible? 🤔

Run this cell to create DataFrames that contain each students' grades.

In [41]:
```
lisa = pd.DataFrame([[20, 46], [18, 54], [5, 20]],
    columns=['Units', 'Grade Points Earned'],
    index=['Fall', 'Winter', 'Spring'],
)
lisa.columns.name = 'Lisa' # This allows us to see the name "Lisa" in the to

bart = pd.DataFrame([[5, 10], [5, 13.5], [22, 81.4]],
    columns=['Units', 'Grade Points Earned'],
    index=['Fall', 'Winter', 'Spring'],
)
bart.columns.name = 'Bart'
```

Quarter-specific vs. overall GPAs

Note: The number of "grade points" earned for a course is

$$\text{number of units} \cdot \text{grade (out of 4)}$$

For instance, an A- in a 4 unit course earns $3.7 \cdot 4 = 14.8$ grade points.

In [42]:
```
dfs_side_by_side(lisa, bart)
```

Lisa	Units	Grade Points Earned
Fall	20	46
Winter	18	54
Spring	5	20

Bart	Units	Grade Points Earned
Fall	5	10.0
Winter	5	13.5
Spring	22	81.4

Lisa had a higher GPA in all three quarters.

In [43]:
```
quarterly_gpas = pd.DataFrame({
    "Lisa's Quarter GPA": lisa['Grade Points Earned'] / lisa['Units'],
    "Bart's Quarter GPA": bart['Grade Points Earned'] / bart['Units'],
})

quarterly_gpas
```

Out[43]:

	Lisa's Quarter GPA	Bart's Quarter GPA
Fall	2.3	2.0
Winter	3.0	2.7
Spring	4.0	3.7

Question 🤔

Use the DataFrame `lisa` to compute Lisa's overall GPA, and use the DataFrame `bart` to compute Bart's overall GPA.

In [44]:
```
# Helper function to show lisa and bart side-by-side to save screen space
dfs_side_by_side(lisa, bart)
```

Lisa	Units	Grade Points Earned
Fall	20	46
Winter	18	54
Spring	5	20

Bart	Units	Grade Points Earned
Fall	5	10.0
Winter	5	13.5
Spring	22	81.4

In [45]:
```
# Your code goes here.
```

What happened?

In [46]:
```
(quarterly_gpas
 .assign(Lisa_Units=lisa['Units'],
         Bart_Units=bart['Units'])
 .iloc[:, [0, 2, 1, 3]]
)
```

Out[46]:

	Lisa's Quarter GPA	Lisa_Units	Bart's Quarter GPA	Bart_Units
Fall	2.3	20	2.0	5
Winter	3.0	18	2.7	5
Spring	4.0	5	3.7	22

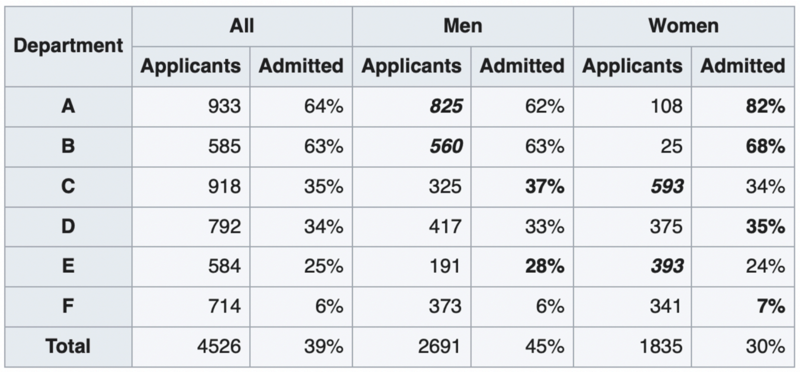
- When Lisa and Bart both performed poorly, Lisa took more units than Bart. **This brought down 📉 Lisa's overall average.**
- When Lisa and Bart both performed well, Bart took more units than Lisa. **This brought up 📈 Bart's overall average.**

Simpson's paradox

- Simpson's paradox occurs when **grouped data and ungrouped data show opposing trends**.
  - It is named after Edward H. Simpson, not Lisa or Bart Simpson.
- It often happens because there is a hidden factor (i.e. a **confounder**) within the data that influences results.
- **Question**: What is the "correct" way to summarize your data? What if you had to act on these results?

Example: How Berkeley was *almost* sued for gender discrimination (1973)

What do you notice?



Department	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
A	933	64%	***825***	62%	108	**82%**
B	585	63%	***560***	63%	25	**68%**
C	918	35%	325	**37%**	***593***	34%
D	792	34%	417	33%	375	**35%**
E	584	25%	191	**28%**	***393***	24%
F	714	6%	373	6%	341	**7%**
Total	4526	39%	2691	45%	1835	30%

What happened?

- The overall acceptance rate for women (30%) was lower than it was for men (45%).
- However, most departments (A, B, D, F) had a higher acceptance rate for women.
- Department A had a 62% acceptance rate for men and an 82% acceptance rate for women!
  - 31% of men applied to Department A.
  - 6% of women applied to Department A.
- Department F had a 6% acceptance rate for men and a 7% acceptance rate for women!
  - 14% of men applied to Department F.
  - 19% of women applied to Department F.
- **Conclusion**: Women tended to apply to departments with a lower acceptance rate; the data don't support the hypothesis that there was major gender discrimination against women.

Example: changes over time

- In 2020, you surveyed CS and DSC students: "Do you use VSCode?". 70% said yes.
- In 2025, you repeat the survey and find that 66% said yes.
- Should you conclude that VSCode is declining in popularity?
- Not necessarily! It can be true that VSCode has increased in popularity within each group.
  - Suppose VSCode is not popular among DSC majors.
  - More people are DSC majors now than in 2020.

In [47]:
```
vscoders = pd.DataFrame({
    "DSC": [15, 175],
    "CSE": [300, 350]
}, index=[2020, 2025])
```

In [48]:
```
n_majors = pd.DataFrame({
    "DSC": [50, 400],
    "CSE": [400, 400]
}, index=[2020, 2025])
```

In [49]:
```
vscoders
```

Out[49]:

	DSC	CSE
2020	15	300
2025	175	350

In [50]:
```
n_majors
```

Out [50]:

	DSC	CSE
2020	50	400
2025	400	400

In [51]:
```
overall_popularity = vscoders.sum(axis=1) / n_majors.sum(axis=1)
overall_popularity
```

Out [51]:
```
2020    0.70
2025    0.66
dtype: float64
```

In [52]:
```
within_major_popularity = vscoders / n_majors
within_major_popularity
```

Out [52]:

	DSC	CSE
2020	0.30	0.75
2025	0.44	0.88

Takeaways

Be skeptical of...

- Aggregate statistics.
- People misusing statistics to "prove" that discrimination doesn't exist.
- Drawing conclusions from individual publications (p-hacking, publication bias, narrow focus, etc.).
- Everything!

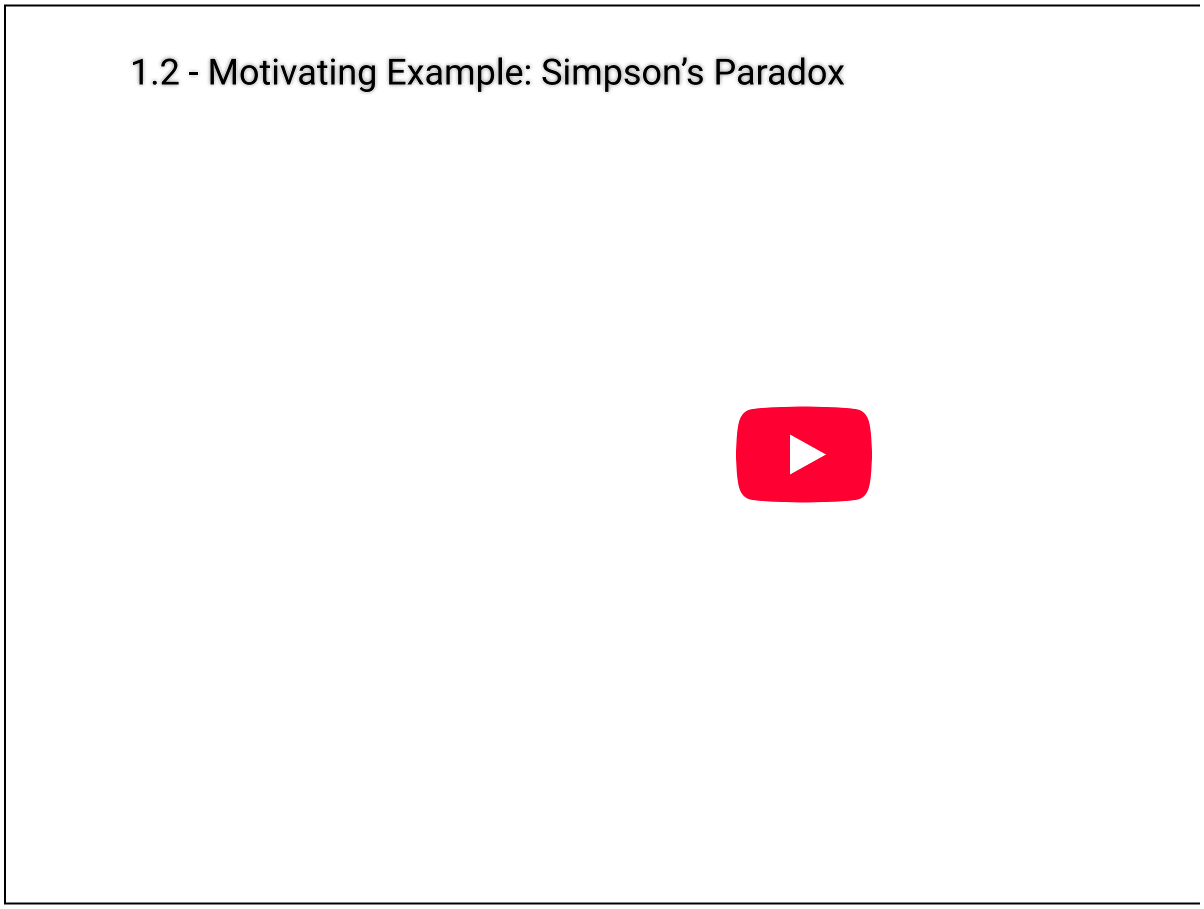
We need to apply domain knowledge and human judgement calls to decide what to do when Simpson's paradox is present.

Really?

To handle Simpson's paradox with rigor, we need some ideas from causal inference which we don't have time to cover in DSC 80. This video has a good example of how to approach Simpson's paradox using a minimal amount of causal inference, if you're curious (not required for DSC 80).

In [53]:
```
IFrame('https://www.youtube-nocookie.com/embed/zeuW1Z2EtLs?si=l2Dl7P-5RCq3OD
       width=800, height=450)
```

Out[53]:



Further reading

- Gender Bias in Admission Statistics?
  - Contains a **great** visualization, but seems to be paywalled now.
- What is Simpson's Paradox?
- Understanding Simpson's Paradox
  - Requires more statistics background, but gives a rigorous understanding of when to use aggregated vs. unaggregated data.

Merging

Example: Name categories

The New York Times article from Lecture 1 claims that certain categories of names are becoming more popular. For example:

- Forbidden names like Lucifer, Lilith, Kali, and Danger.
- Evangelical names like Amen, Savior, Canaan, and Creed.
- Mythological names.

- It also claims that baby boomer names are becoming less popular.

Let's see if we can verify these claims using data!

Loading in the data

Our first DataFrame, `baby`, is the same as we saw in Lecture 1. It has one row for every combination of `'Name'`, `'Sex'`, and `'Year'`.

In [54]:
```
baby_path = Path('data') / 'baby.csv'
baby = pd.read_csv(baby_path)
baby
```

Out[54]:

	Name	Sex	Count	Year
0	Liam	M	20456	2022
1	Noah	M	18621	2022
2	Olivia	F	16573	2022
...	...	...	...	...
2085155	Wright	M	5	1880
2085156	York	M	5	1880
2085157	Zachariah	M	5	1880

2085158 rows × 4 columns

Our second DataFrame, `nyt`, contains the New York Times' categorization of each of several names, based on the aforementioned article.

In [55]:
```
nyt_path = Path('data') / 'nyt_names.csv'
nyt = pd.read_csv(nyt_path)
nyt
```

Out[55]:

	nyt_name	category
0	Lucifer	forbidden
1	Lilith	forbidden
2	Danger	forbidden
...	...	...
20	Venus	celestial
21	Celestia	celestial
22	Skye	celestial

23 rows × 2 columns

Issue: To find the number of babies born with (for example) forbidden names each year, we need to combine information from both `baby` and `nyt`.

Merging

- We want to link rows from `baby` and `nyt` together whenever the names match up.
- This is a **merge** (`pandas` term), i.e. a **join** (SQL term).
- A merge is appropriate when we have two sources of information **about the same individuals** that is **linked by a common column(s)**.
- The common column(s) are called the **join key**.

Example merge

Let's demonstrate on a small subset of `baby` and `nyt`.

In [56]:
```
nyt_small = nyt.iloc[[11, 12, 14]].reset_index(drop=True)

names_to_keep = ['Julius', 'Karen', 'Noah']
baby_small = (baby
 .query("Year == 2020 and Name in @names_to_keep")
 .reset_index(drop=True)
)

dfs_side_by_side(baby_small, nyt_small)
```

	Name	Sex	Count	Year
0	Noah	M	18407	2020
1	Julius	M	966	2020
2	Karen	F	330	2020
3	Noah	F	306	2020
4	Karen	M	6	2020

	nyt_name	category
0	Karen	boomer
1	Julius	mythology
2	Freya	mythology

In [57]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name')
```

Out[57]:

	Name	Sex	Count	Year	nyt_name	category
0	Julius	M	966	2020	Julius	mythology
1	Karen	F	330	2020	Karen	boomer
2	Karen	M	6	2020	Karen	boomer

The `merge` method

- The `merge` DataFrame method joins two DataFrames by columns or indexes.
  - As mentioned before, "merge" is just the `pandas` word for "join."
- When using the `merge` method, the DataFrame before `merge` is the "left" DataFrame, and the DataFrame passed into `merge` is the "right" DataFrame.
  - In `baby_small.merge(nyt_small)`, `baby_small` is considered the "left" DataFrame and `nyt_small` is the "right" DataFrame; the columns from the left DataFrame appear to the left of the columns from right DataFrame.
- By default:
  - If join keys are not specified, all shared columns between the two DataFrames are used.
  - The "type" of join performed is an inner join. **This is the only type of join you saw in DSC 10, but there are more, as we'll now see!**

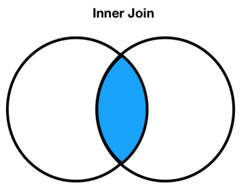
Join types: inner joins

In [58]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name')
```

Out[58]:

	Name	Sex	Count	Year	nyt_name	category
0	Julius	M	966	2020	Julius	mythology
1	Karen	F	330	2020	Karen	boomer
2	Karen	M	6	2020	Karen	boomer

- Note that `'Noah'` and `'Freya'` do not appear in the merged DataFrame.
- This is because there is:
  - no `'Noah'` in the right DataFrame (`nyt_small`), and
  - no `'Freya'` in the left DataFrame (`baby_small`).
- The default type of join that `merge` performs is an **inner join**, which keeps the **intersection** of the join keys.



Different join types

We can change the type of join performed by changing the `how` argument in `merge`. Let's experiment!

In [59]:
```
# Note the NaNs!
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name', how='left')
```

Out[59]:

	Name	Sex	Count	Year	nyt_name	category
0	Noah	M	18407	2020	NaN	NaN
1	Julius	M	966	2020	Julius	mythology
2	Karen	F	330	2020	Karen	boomer
3	Noah	F	306	2020	NaN	NaN
4	Karen	M	6	2020	Karen	boomer

In [60]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name', how='right'
```

Out[60]:

	Name	Sex	Count	Year	nyt_name	category
0	Karen	F	330.0	2020.0	Karen	boomer
1	Karen	M	6.0	2020.0	Karen	boomer
2	Julius	M	966.0	2020.0	Julius	mythology
3	NaN	NaN	NaN	NaN	Freya	mythology

In [61]:
```
baby_small.merge(nyt_small, left_on='Name', right_on='nyt_name', how='outer'
```

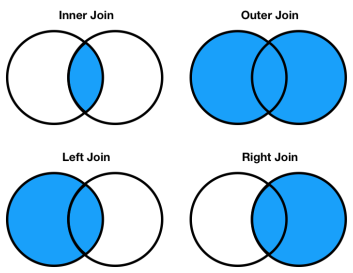
Out[61]:

	Name	Sex	Count	Year	nyt_name	category
0	NaN	NaN	NaN	NaN	Freya	mythology
1	Julius	M	966.0	2020.0	Julius	mythology
2	Karen	F	330.0	2020.0	Karen	boomer
3	Karen	M	6.0	2020.0	Karen	boomer
4	Noah	M	18407.0	2020.0	NaN	NaN
5	Noah	F	306.0	2020.0	NaN	NaN

Different join types handle mismatches differently

There are four types of joins.

- **Inner**: keep **only** matching keys (intersection).
- **Outer**: keep **all** keys in both DataFrames (union).
- **Left**: keep all keys in the left DataFrame, whether or not they are in the right DataFrame.
- **Right**: keep all keys in the right DataFrame, whether or not they are in the left DataFrame.
  - Note that `a.merge(b, how='left')` contains the same information as `b.merge(a, how='right')`, just in a different order.



Notes on the `merge` method

- `merge` is flexible – you can merge using a combination of columns, or the index of the DataFrame.
- If the two DataFrames have the same column names, `pandas` will add `_x` and `_y` to the duplicated column names to avoid having columns with the same name (change these the `suffixes` argument).
- There is, in fact, a `join` method, but it's actually a wrapper around `merge` with fewer options.
- **As always, the documentation is your friend!**

Lots of `pandas` operations do an implicit outer join!

- `pandas` will almost always try to match up index values using an outer join.
- It won't tell you that it's doing an outer join, it'll just throw `NaN` s in your result!

In [62]:
```python
df1 = pd.DataFrame({'a': [1, 2, 3]}, index=['hello', 'dsc80', 'students'])
df2 = pd.DataFrame({'b': [10, 20, 30]}, index=['dsc80', 'is', 'awesome'])
dfs_side_by_side(df1, df2)
```

	a
hello	1
dsc80	2
students	3

	b
dsc80	10
is	20
awesome	30

In [63]:
```python
df1['a'] + df2['b']
```

Out[63]:
```
awesome       NaN
dsc80        12.0
hello         NaN
is            NaN
students      NaN
dtype: float64
```

Many-to-one & many-to-many joins

One-to-one joins

- So far in this lecture, the joins we have worked with are called **one-to-one** joins.
- Neither the left DataFrame (`baby_small`) nor the right DataFrame (`nyt_small`) contained any duplicates in the join key.
- What if there are duplicated join keys, in one or both of the DataFrames we are merging?

In [64]:
```python
# Run this cell to set up the next example.
profs = pd.DataFrame(
[['Sam', 'UCB', 5],
 ['Sam', 'UCSD', 5],
 ['Janine', 'UCSD', 8],
 ['Marina', 'UIC', 7],
 ['Justin', 'OSU', 5],
 ['Soohyun', 'UCSD', 2],
 ['Suraj', 'UCB', 2]],
    columns=['Name', 'School', 'Years']
)

schools = pd.DataFrame({
    'Abr': ['UCSD', 'UCLA', 'UCB', 'UIC'],
    'Full': ['University of California San Diego', 'University of California
})

programs = pd.DataFrame({
    'uni': ['UCSD', 'UCSD', 'UCSD', 'UCB', 'OSU', 'OSU'],
    'dept': ['Math', 'HDSI', 'COGS', 'CS', 'Math', 'CS'],
    'grad_students': [205, 54, 281, 439, 304, 193]
})
```

Many-to-one joins

- Many-to-one joins are joins where **one** of the DataFrames contains duplicate values in the join key.
- The resulting DataFrame will preserve those duplicate entries as appropriate.

In [65]:
```
dfs_side_by_side(profs, schools)
```

	Name	School	Years
0	Sam	UCB	5
1	Sam	UCSD	5
2	Janine	UCSD	8
3	Marina	UIC	7
4	Justin	OSU	5
5	Soohyun	UCSD	2
6	Suraj	UCB	2

	Abr	Full
0	UCSD	University of California San Diego
1	UCLA	University of California, Los Angeles
2	UCB	University of California, Berkeley
3	UIC	University of Illinois Chicago

Note that when merging `profs` and `schools`, the information from `schools` is duplicated.

- `'University of California, San Diego'` appears three times.
- `'University of California, Berkeley'` appears twice.

In [66]:
```
profs.merge(schools, left_on='School', right_on='Abr', how='left')
```

Out[66]:

	Name	School	Years	Abr	Full
0	Sam	UCB	5	UCB	University of California, Berkeley
1	Sam	UCSD	5	UCSD	University of California San Diego
2	Janine	UCSD	8	UCSD	University of California San Diego
3	Marina	UIC	7	UIC	University of Illinois Chicago
4	Justin	OSU	5	NaN	NaN
5	Soohyun	UCSD	2	UCSD	University of California San Diego
6	Suraj	UCB	2	UCB	University of California, Berkeley

Many-to-many joins

Many-to-many joins are joins where both DataFrames have duplicate values in the join key.

In [67]:
```
dfs_side_by_side(profs, programs)
```

	Name	School	Years
0	Sam	UCB	5
1	Sam	UCSD	5
2	Janine	UCSD	8
3	Marina	UIC	7
4	Justin	OSU	5
5	Soohyun	UCSD	2
6	Suraj	UCB	2

	uni	dept	grad_students
0	UCSD	Math	205
1	UCSD	HDSI	54
2	UCSD	COGS	281
3	UCB	CS	439
4	OSU	Math	304
5	OSU	CS	193

Before running the following cell, try predicting the number of rows in the output.

In [68]:
```
profs.merge(programs, left_on='School', right_on='uni')
```

Out[68]:

	Name	School	Years	uni	dept	grad_students
0	Sam	UCB	5	UCB	CS	439
1	Sam	UCSD	5	UCSD	Math	205
2	Sam	UCSD	5	UCSD	HDSI	54
...	...	...	...	...	...	...
10	Soohyun	UCSD	2	UCSD	HDSI	54
11	Soohyun	UCSD	2	UCSD	COGS	281
12	Suraj	UCB	2	UCB	CS	439

13 rows × 6 columns

- `merge` stitched together every UCSD row in `profs` with every UCSD row in `programs`.
- Since there were 3 UCSD rows in `profs` and 3 in `programs`, there are $3 \cdot 3 = 9$ UCSD rows in the output. The same applies for all other schools.

Question 🤔

Fill in the blank so that the last statement evaluates to `True`.

```
df = profs.merge(programs, left_on='School', right_on='uni')
df.shape[0] == (____).sum()
```

Don't use `merge` (or `join`) in your solution!

In [69]:
```
dfs_side_by_side(profs, programs)
```

	Name	School	Years
0	Sam	UCB	5
1	Sam	UCSD	5
2	Janine	UCSD	8
3	Marina	UIC	7
4	Justin	OSU	5
5	Soohyun	UCSD	2
6	Suraj	UCB	2

	uni	dept	grad_students
0	UCSD	Math	205
1	UCSD	HDSI	54
2	UCSD	COGS	281
3	UCB	CS	439
4	OSU	Math	304
5	OSU	CS	193

In [70]:
```
# Your code goes here.
```

Returning back to our original question

Let's find the popularity of baby name categories over time. To start, we'll define a DataFrame that has one row for every combination of `'category'` and `'Year'`.

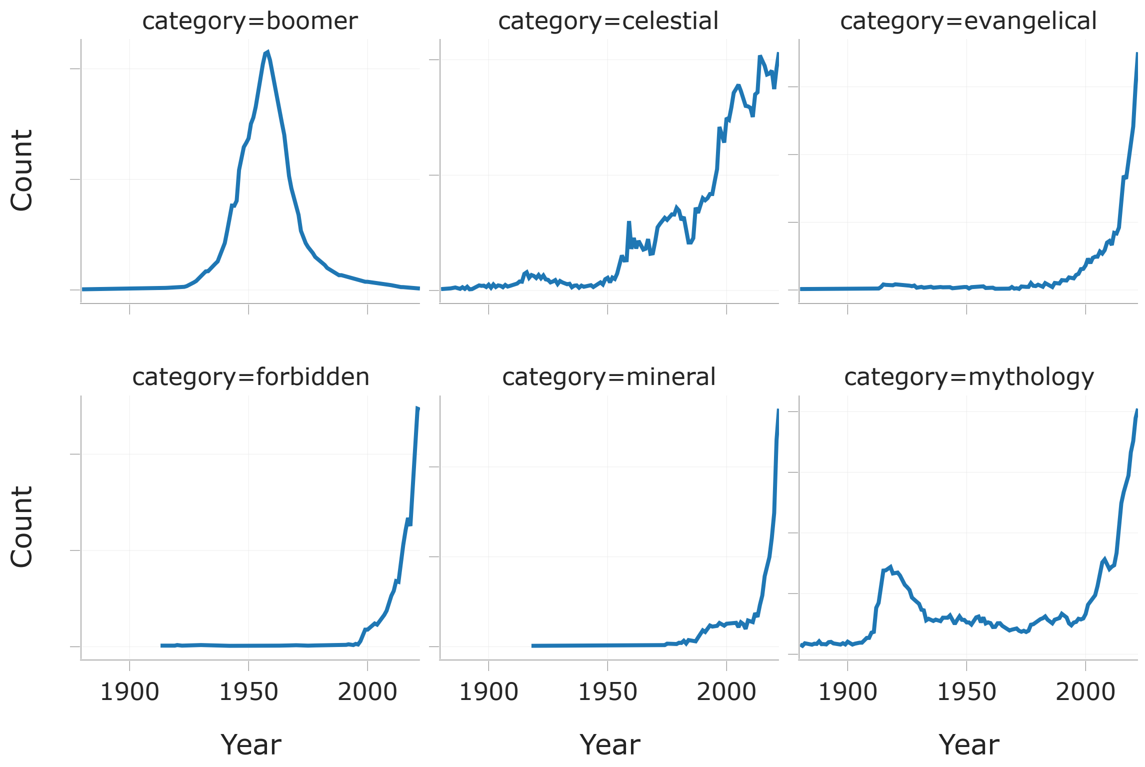
In [71]:
```
cate_counts = (
    baby
    .merge(nyt, left_on='Name', right_on='nyt_name')
    .groupby(['category', 'Year'])
    ['Count']
    .sum()
    .reset_index()
)
cate_counts
```

Out[71]:

	category	Year	Count
0	boomer	1880	292
1	boomer	1881	298
2	boomer	1882	326
...	...	...	...
659	mythology	2020	3516
660	mythology	2021	3895
661	mythology	2022	4049

662 rows × 3 columns

In [72]:
```
# We'll talk about plotting code soon!
import plotly.express as px
fig = px.line(cate_counts, x='Year', y='Count',
              facet_col='category', facet_col_wrap=3,
              facet_row_spacing=0.15,
              width=600, height=400)
fig.update_yaxes(matches=None, showticklabels=False)
```



Questions? 🤔

Transforming

Transforming values

- A **transformation** results from performing some operation on every element in a sequence, e.g. a Series.
- While we haven't discussed it yet in DSC 80, you learned how to transform Series in DSC 10, using the `apply` method. `apply` is very flexible – it takes in a function, which itself takes in a single value as input and returns a single value.

In [73]:
```
baby
```

Out[73]:

	Name	Sex	Count	Year
0	Liam	M	20456	2022
1	Noah	M	18621	2022
2	Olivia	F	16573	2022
...	...	...	...	...
2085155	Wright	M	5	1880
2085156	York	M	5	1880
2085157	Zachariah	M	5	1880

2085158 rows × 4 columns

In [74]:
```python
def number_of_vowels(string):
    return sum(c in 'aeiou' for c in string.lower())

baby['Name'].apply(number_of_vowels)
```

Out[74]:
```
0          2
1          2
2          4
          ..
2085155    1
2085156    1
2085157    4
Name: Name, Length: 2085158, dtype: int64
```

In [75]:
```python
# Built-in functions work with apply, too.
baby['Name'].apply(len)
```

Out[75]:
```
0          4
1          4
2          6
          ..
2085155    6
2085156    4
2085157    9
Name: Name, Length: 2085158, dtype: int64
```

The price of `apply`

Unfortunately, `apply` runs really slowly!

In [76]:
```python
%%timeit
baby['Name'].apply(number_of_vowels)
```

```
953 ms ± 7.79 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [77]:
```python
%%timeit
res = []
```

```
for name in baby['Name']:
    res.append(number_of_vowels(name))
```

```
796 ms ± 9.74 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Internally, `apply` actually just runs a `for` -loop!

So, when possible – say, when applying arithmetic operations – we should work on Series objects directly and avoid `apply` !

The price of `apply`

In [78]:
```
%%timeit
baby['Year'] // 10 * 10 # Rounds down to the nearest multiple of 10.
```

```
3.54 ms ± 17.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [79]:
```
%%timeit
baby['Year'].apply(lambda y: y // 10 * 10)
```

```
361 ms ± 36.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

100x slower!

The `.str` accessor

For string operations, `pandas` provides a convenient `.str` accessor.

In [80]:
```
%%timeit
baby['Name'].str.len()
```

```
251 ms ± 2.47 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [81]:
```
%%timeit
baby['Name'].apply(len)
```

```
262 ms ± 1.36 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It's very convenient and **runs about the same speed as `apply` !**

Other data representations

Representations of tabular data

- In DSC 80, we work with DataFrames in `pandas` .
    - When we say `pandas` DataFrame, we're talking about the `pandas` API for its DataFrame objects.
        - API stands for "application programming interface." We'll learn about these more soon.

- When we say "DataFrame", we're referring to a general way to represent data (rows and columns, with labels for both rows and columns).

- There many other ways to work with data tables!
  - Examples: R data frames, SQL databases, spreadsheets, or even matrices from linear algebra.
  - When you learn SQL in DSC 100, you'll find many similaries (e.g. slicing columns, filtering rows, grouping, joining, etc.).
  - **Relational algebra** captures common data operations between many data table systems.
- Why use DataFrames over something else?

DataFrames vs. spreadsheets

- DataFrames give us a **data lineage**: the code records down data changes. Not so in spreadsheets!
- Using a general-purpose programming language gives us the ability to handle much larger datasets, and we can use distributed computing systems to handle massive datasets.

DataFrames vs. matrices

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \\ 0 & 0 \end{bmatrix}$$

- Matrices are mathematical objects. They only hold numbers, but have many useful properties (which you've learned about in your linear algebra class, Math 18).
- Often, we process data from a DataFrame into matrix format for machine learning models. You saw this a bit in DSC 40A, and we'll see this more in DSC 80 in a few weeks.

DataFrames vs. relations

- Relations are the data representation for relational database systems (e.g. MySQL, PostgreSQL, etc.).
- You'll learn all about these in DSC 100.
- Database systems are much better than DataFrames at storing **many large** data tables and handling concurrency (many people reading and writing data at the same time).
- Common workflow: load a subset of data in from a database system into `pandas`, then make a plot.

- Or: load and clean data in `pandas`, then store it in a database system for others to use.

Summary

- There is no "formula" to automatically resolve Simpson's paradox! Domain knowledge is important.
- We've covered most of the primary DataFrame operations: subsetting, aggregating, joining, and transforming.

Next time

Data cleaning: applying what we've already learned to real-world, messy data!