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

Lecture 3 – Aggregating

DSC 80, Fall 2025

Announcements 📣

- Lab 1 is due **Monday at 11:59pm.**
- Project 1 is released.
  - The checkpoint (Questions 1-7) is due on **Thursday, Oct 9th**.
  - The full project is due on **Tuesday, Oct 16th**.
- Lab 2 will be released by Monday.

Agenda

- Adding and modifying columns
- Data granularity and the `groupby` method.
- `DataFrameGroupBy` objects and aggregation.
- Other `DataFrameGroupBy` methods.
- Pivot tables using the `pivot_table` method.

You will need to code **a lot** today – make sure to pull the course repository to follow along.

Adding and modifying columns

Adding and modifying columns, using a copy

- DSC 10: To add a new column to a DataFrame, you can use the `assign` method.
  - To change the values in a column, add a new column with the same name as the existing column.
- Like most `pandas` methods, `assign` returns a new DataFrame.
  - **Pro** ✅ : This doesn't inadvertently change any existing variables.
  - **Con** ❌ : It is not very space efficient, as it creates a new copy each time it is called.

In [2]:
```
dogs = pd.read_csv(Path('data') / 'dogs43.csv', index_col='breed')
```

In [3]:
```
dogs.assign(cost_per_year=dogs['lifetime_cost'] / dogs['longevity'])
```

Out[3]:

breed	kind	lifetime_cost	longevity	size	weight	height	cost_per_year
Brittany	sporting	22589.0	12.92	medium	35.0	19.0	1748.37
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0	1589.02
English Cocker Spaniel	sporting	18993.0	11.66	medium	30.0	16.0	1628.90
...	...	...	...	...	...	...	...
Bullmastiff	working	13936.0	7.57	large	115.0	25.5	1840.95
Mastiff	working	13581.0	6.50	large	175.0	30.0	2089.38
Saint Bernard	working	20022.0	7.78	large	155.0	26.5	2573.52

43 rows × 7 columns

In [4]:
```
dogs
```

Out[4]:

breed	kind	lifetime_cost	longevity	size	weight	height
Brittany	sporting	22589.0	12.92	medium	35.0	19.0
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0
English Cocker Spaniel	sporting	18993.0	11.66	medium	30.0	16.0
...	...	...	...	...	...	...
Bullmastiff	working	13936.0	7.57	large	115.0	25.5
Mastiff	working	13581.0	6.50	large	175.0	30.0
Saint Bernard	working	20022.0	7.78	large	155.0	26.5

43 rows × 6 columns

💡 Pro-Tip: Method chaining

Chain methods together instead of writing long, hard-to-read lines.

In [5]:
```
# Finds the rows corresponding to the five cheapest to own breeds on a per-y
(dogs
 .assign(cost_per_year=dogs['lifetime_cost'] / dogs['longevity'])
 .sort_values('cost_per_year')
 .iloc[:5]
)
```

Out[5]:

breed	kind	lifetime_cost	longevity	size	weight	height	cost_per_year
Maltese	toy	19084.0	12.25	small	5.0	9.00	1557.88
Lhasa Apso	non-sporting	22031.0	13.92	small	15.0	10.50	1582.69
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.00	1589.02
Chihuahua	toy	26250.0	16.50	small	5.5	5.00	1590.91
Shih Tzu	toy	21152.0	13.20	small	12.5	9.75	1602.42

💡 Pro-Tip: `assign` for column names with special characters

You can also use `assign` when the desired column name has spaces (and other special characters) by unpacking a dictionary:

In [6]:
```
dogs.assign(**{'cost per year 💵': dogs['lifetime_cost'] / dogs['longevity']
```

Out[6]:

breed	kind	lifetime_cost	longevity	size	weight	height	cost per year 💵
Brittany	sporting	22589.0	12.92	medium	35.0	19.0	1748.37
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0	1589.02
English Cocker Spaniel	sporting	18993.0	11.66	medium	30.0	16.0	1628.90
...	...	...	...	...	...	...	...
Bullmastiff	working	13936.0	7.57	large	115.0	25.5	1840.95
Mastiff	working	13581.0	6.50	large	175.0	30.0	2089.38
Saint Bernard	working	20022.0	7.78	large	155.0	26.5	2573.52

43 rows × 7 columns

Adding and modifying columns, in-place

- You can assign a new column to a DataFrame **in-place** using `[]`.
  - This works like dictionary assignment.
  - This **modifies** the underlying DataFrame, unlike `assign`, which returns a new DataFrame.

- This is the more "common" way of adding/modifying columns.
    - ⚠️ Warning: Exercise caution when using this approach, since this approach changes the values of existing variables.

In [7]:
```
# By default, .copy() returns a deep copy of the object it is called on,
# meaning that if you change the copy the original remains unmodified.
dogs_copy = dogs.copy()
dogs_copy.head(2)
```

Out[7]:

breed	kind	lifetime_cost	longevity	size	weight	height
Brittany	sporting	22589.0	12.92	medium	35.0	19.0
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0

In [8]:
```
dogs_copy['cost_per_year'] = dogs_copy['lifetime_cost'] / dogs_copy['longevi
dogs_copy
```

Out[8]:

breed	kind	lifetime_cost	longevity	size	weight	height	cost_per_year
Brittany	sporting	22589.0	12.92	medium	35.0	19.0	1748.37
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0	1589.02
English Cocker Spaniel	sporting	18993.0	11.66	medium	30.0	16.0	1628.90
...	...	...	...	...	...	...	...
Bullmastiff	working	13936.0	7.57	large	115.0	25.5	1840.95
Mastiff	working	13581.0	6.50	large	175.0	30.0	2089.38
Saint Bernard	working	20022.0	7.78	large	155.0	26.5	2573.52

43 rows × 7 columns

Note that we never reassigned `dogs_copy` in the cell above – that is, we never wrote `dogs_copy = ...` – though it was still modified.

Mutability

DataFrames, like lists, arrays, and dictionaries, are **mutable**. As you learned in DSC 20, this means that they can be modified after being created. (For instance, the list `.append` method mutates in-place.)

Not only does this explain the behavior on the previous slide, but it also explains the following:

In [9]:
```
dogs_copy
```

Out[9]:

	kind	lifetime_cost	longevity	size	weight	height	cost_per_year
breed							
Brittany	sporting	22589.0	12.92	medium	35.0	19.0	1748.37
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0	1589.02
English Cocker Spaniel	sporting	18993.0	11.66	medium	30.0	16.0	1628.90
...	...	...	...	...	...	...	...
Bullmastiff	working	13936.0	7.57	large	115.0	25.5	1840.95
Mastiff	working	13581.0	6.50	large	175.0	30.0	2089.38
Saint Bernard	working	20022.0	7.78	large	155.0	26.5	2573.52

43 rows × 7 columns

In [10]:
```
def cost_in_thousands():
    dogs_copy['lifetime_cost'] = dogs_copy['lifetime_cost'] / 1000
```

In [11]:
```
# What happens when we run this twice?
cost_in_thousands()
```

In [12]:
```
dogs_copy
```

Out[12]:

breed	kind	lifetime_cost	longevity	size	weight	height	cost_per_year
Brittany	sporting	22.59	12.92	medium	35.0	19.0	1748.37
Cairn Terrier	terrier	21.99	13.84	small	14.0	10.0	1589.02
English Cocker Spaniel	sporting	18.99	11.66	medium	30.0	16.0	1628.90
...	...	...	...	...	...	...	...
Bullmastiff	working	13.94	7.57	large	115.0	25.5	1840.95
Mastiff	working	13.58	6.50	large	175.0	30.0	2089.38
Saint Bernard	working	20.02	7.78	large	155.0	26.5	2573.52

43 rows × 7 columns

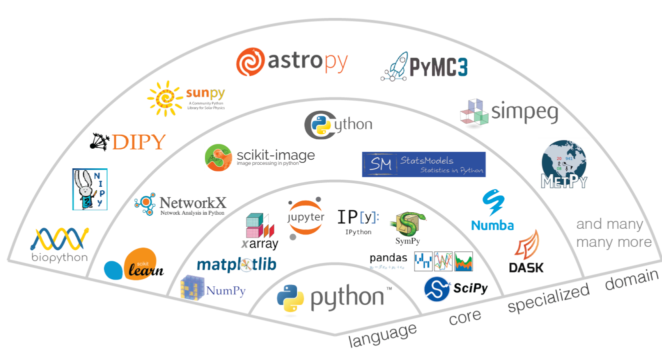
⚠️ Avoid mutation when possible

Note that `dogs_copy` was modified, even though we didn't reassign it! These unintended consequences can **influence the behavior of test cases on labs and projects**, among other things!

To avoid this, it's a good idea to avoid mutation when possible. If you must use mutation, include `df = df.copy()` as the first line in functions that take DataFrames as input.

Also, some methods let you use the `inplace=True` argument to mutate the original. **Don't use this argument, since future `pandas` releases plan to remove it.**

`pandas` and `numpy`



`pandas` is built upon `numpy`!

- A Series in `pandas` is a `numpy` array with an index.
- A DataFrame is like a dictionary of columns, each of which is a `numpy` array.
- Many operations in `pandas` are fast because they use `numpy`'s implementations, which are written in fast languages like C.
- If you need access the array underlying a DataFrame or Series, use the `to_numpy` method.

In [13]:
```
dogs['lifetime_cost']
```

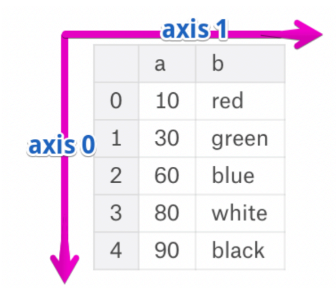
Out[13]:
```
breed
Brittany                   22589.0
Cairn Terrier              21992.0
English Cocker Spaniel     18993.0
                             ...
Bullmastiff                13936.0
Mastiff                    13581.0
Saint Bernard              20022.0
Name: lifetime_cost, Length: 43, dtype: float64
```

In [14]:
```
dogs['lifetime_cost'].to_numpy()
```

Out[14]:
```
array([22589., 21992., 18993., ..., 13936., 13581., 20022.])
```

Axes

- The rows and columns of a DataFrame are both stored as Series.
- The **axis** specifies the direction of a **slice** of a DataFrame.



- Axis 0 refers to the index (rows).
- Axis 1 refers to the columns.
- **These are the same axes definitions that 2D `numpy` arrays have!**

DataFrame methods with `axis`

- Many Series methods work on DataFrames.
- In such cases, the DataFrame method usually applies the Series method to every row or column.
- Many of these methods accept an `axis` argument; the default is usually `axis=0`.

In [15]:
```
dogs
```

Out[15]:

	kind	lifetime_cost	longevity	size	weight	height
breed						
Brittany	sporting	22589.0	12.92	medium	35.0	19.0
Cairn Terrier	terrier	21992.0	13.84	small	14.0	10.0
English Cocker Spaniel	sporting	18993.0	11.66	medium	30.0	16.0
...	...	...	...	...	...	...
Bullmastiff	working	13936.0	7.57	large	115.0	25.5
Mastiff	working	13581.0	6.50	large	175.0	30.0
Saint Bernard	working	20022.0	7.78	large	155.0	26.5

43 rows × 6 columns

In [16]:
```
# Max element in each column.
dogs.max()
```

Out[16]:
```
kind             working
lifetime_cost    26686.0
longevity           16.5
size               small
weight             175.0
height              30.0
dtype: object
```

In [17]:
```
# Max element in each row – errors, since there are different types in each
# dogs.max(axis=1)
```

In [18]:
```
# The number of unique values in each column.
dogs.nunique()
```

Out[18]:
```
kind              7
lifetime_cost    43
longevity        40
size              3
weight           37
height           30
dtype: int64
```

In [19]:
```
# describe doesn't accept an axis argument; it works on every numeric columr
dogs.describe()
```

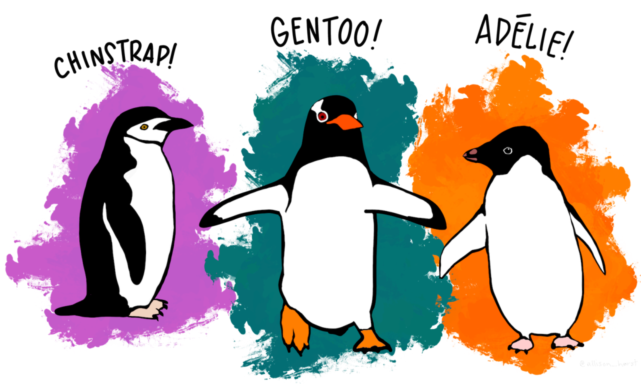
Out [19]:

	lifetime_cost	longevity	weight	height
count	43.00	43.00	43.00	43.00
mean	20532.84	11.34	49.35	18.34
std	3290.78	2.05	39.42	6.83
...	...	...	...	...
50%	21006.00	11.81	36.50	18.50
75%	22072.50	12.52	67.50	25.00
max	26686.00	16.50	175.00	30.00

8 rows × 4 columns

Data granularity and the `groupby` method

Example: Palmer Penguins

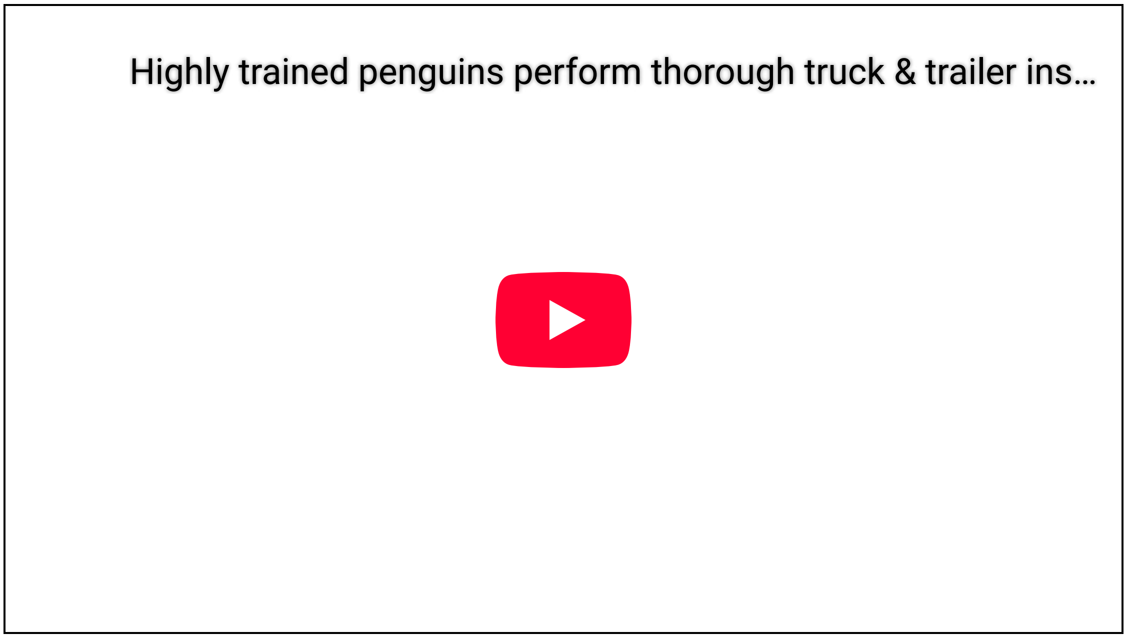


Artwork by @allison_horst

The dataset we'll work with for the rest of the lecture involves various measurements taken of three species of penguins in Antarctica.

In [20]:
```
IFrame('https://www.youtube-nocookie.com/embed/CCrNAHXUstU?si=-DntSyUNp5Kwit
       width=560, height=315)
```

Out [20]:



In [21]:

```
import seaborn as sns
penguins = sns.load_dataset('penguins').dropna()
penguins
```

Out[21]:

	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

Here, each row corresponds to a single penguin, and each column corresponds to a different attribute (or feature) we have for each penguin. Data formatted in this way is called tidy data.

Granularity

- Granularity refers to what each observation in a dataset represents.
  - Fine: small details.
  - Coarse: bigger picture.

- If you can control how your dataset is created, you should opt for **finer granularity**, i.e. for more detail.
  - You can always remove details, but it's difficult to add detail that isn't already there.
  - But obtaining fine-grained data can take more time/money.

- Today, we'll focus on how to **remove** details from fine-grained data, in order to help us understand bigger-picture trends in our data.

Aggregating

Aggregating is the act of combining many values into a single value.

- What is the mean `'body_mass_g'` for all penguins?

In [22]:
```
penguins['body_mass_g'].mean()
```

Out[22]:
```
np.float64(4207.057057057057)
```

- What is the mean `'body_mass_g'` **for each species**?

In [23]:
```
# ???
```

Naive approach: looping through unique values

In [24]:
```
species_map = pd.Series([], dtype=float)

for species in penguins['species'].unique():
    species_only = penguins.loc[penguins['species'] == species]
    species_map.loc[species] = species_only['body_mass_g'].mean()

species_map
```

Out[24]:
```
Adelie       3706.16
Chinstrap    3733.09
Gentoo       5092.44
dtype: float64
```

- For each unique `'species'`, we make a pass through the entire dataset.
  - The asymptotic runtime of this procedure is $\Theta(ns)$, where n is the number of rows and s is the number of unique species.

- While there are other loop-based solutions that only involve a single pass over the DataFrame, we'd like to avoid Python loops entirely, as they're slow.

Grouping

A better solution, as we know from DSC 10, is to use the `groupby` method.

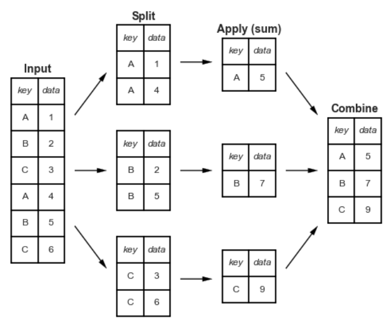
In [25]:
```
penguins.groupby('species')['body_mass_g'].mean()
```

Out[25]:
```
species
Adelie       3706.16
Chinstrap    3733.09
Gentoo       5092.44
Name: body_mass_g, dtype: float64
```

Somehow, the `groupby` method computes what we're looking for in just one line. How?

"Split-apply-combine" paradigm

The `groupby` method involves three steps: **split**, **apply**, and **combine**. This is the same terminology that the `pandas` documentation uses.



- **Split** breaks up and "groups" the rows of a DataFrame according to the specified **key**. There is one "group" for every unique value of the key.
- **Apply** uses a function (e.g. aggregation, transformation, filtration) within the individual groups.
- **Combine** stitches the results of these operations into an output DataFrame.
- The split-apply-combine pattern can be **parallelized** to work on multiple computers or threads, by sending computations for each group to different processors.

More examples

Before we dive into the internals, let's look at a few more examples.

Question 🤔

Code: `dream`

What proportion of penguins of each `'species'` live on `'Dream'` island?

Your output should look like:

```
species
Adelie       0.38
Chinstrap    1.00
Gentoo       0.00
```

In [26]:
```
# Fill this in
```

`DataFrameGroupBy` objects and aggregation

`DataFrameGroupBy` objects

We've just evaluated a few expressions of the following form.

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.groupby('species')['bill_length_mm'].mean()
```

Out[28]:
```
species
Adelie       38.82
Chinstrap    48.83
Gentoo       47.57
Name: bill_length_mm, dtype: float64
```

There are two method calls in the expression above: `.groupby('species')` and `.mean()`. What happens in the `.groupby()` call?

In [29]:
```
penguins.groupby('species')
```

Out[29]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x111e0c9d0>
```

Peeking under the hood

If `df` is a DataFrame, then `df.groupby(key)` returns a `DataFrameGroupBy` object.

This object represents the "split" in "split-apply-combine".

In [30]:
```
# Simplified DataFrame for demonstration:
penguins_small = penguins.iloc[[0, 150, 300, 1, 251, 151, 301], [0, 5, 6]]
penguins_small
```

Out[30]:

	species	body_mass_g	sex
0	Adelie	3750.0	Male
156	Chinstrap	3725.0	Male
308	Gentoo	4875.0	Female
1	Adelie	3800.0	Female
258	Gentoo	4350.0	Female
157	Chinstrap	3950.0	Female
309	Gentoo	5550.0	Male

In [31]:
```
# Creates one group for each unique value in the species column.
penguin_groups = penguins_small.groupby('species')
penguin_groups
```

Out[31]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x111e0e710>
```

`DataFrameGroupBy` objects are iterable:

In [32]:
```
for key, group in penguins.groupby('species'):
    print(key, type(group))
```

```
Adelie <class 'pandas.core.frame.DataFrame'>
Chinstrap <class 'pandas.core.frame.DataFrame'>
Gentoo <class 'pandas.core.frame.DataFrame'>
```

`DataFrameGroupBy` objects have a `groups` attribute, which is a dictionary in which the keys are group names and the values are lists of row labels.

In [33]:
```
penguin_groups.groups
```

Out[33]:
```
{'Adelie': [0, 1], 'Chinstrap': [156, 157], 'Gentoo': [308, 258, 309]}
```

`DataFrameGroupBy` objects also have a `get_group(key)` method, which returns a DataFrame with only the values for the given key.

In [34]:
```
penguin_groups.get_group('Chinstrap')
```

Out[34]:

	species	body_mass_g	sex
156	Chinstrap	3725.0	Male
157	Chinstrap	3950.0	Female

In [35]:
```
# Same as the above!
penguins_small[penguins_small['species'] == "Chinstrap"]
```

Out[35]:

	species	body_mass_g	sex
156	Chinstrap	3725.0	Male
157	Chinstrap	3950.0	Female

We usually don't use these attributes and methods, but they're useful in understanding how `groupby` works under the hood.

`.groupby` is "lazy"

- The actual groups aren't computed until needed.
- This is an optimization that helps when working with really large datasets.

In [36]:
```
n = 1_000_000
large_df = pd.DataFrame({
    'letter': np.random.choice(['A', 'B', 'C'], n),
    'number': np.random.uniform(size=n)
})
large_df
```

Out[36]:

	letter	number
0	A	0.27
1	B	0.24
2	B	0.14
...	...	...
999997	C	0.84
999998	A	0.66
999999	B	0.92

1000000 rows × 2 columns

In [37]:
```
%%timeit
# notice how this doesn't take longer when we change n to 10_000_000
grouped = large_df.groupby('letter')
```

```
5.73 µs ± 39 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [38]:
```
%%timeit
# we can't avoid computing the groups now...
len(large_df.groupby('letter'))
```

```
26.7 ms ± 426 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Aggregation

- Once we create a `DataFrameGroupBy` object, we need to **apply** some function to each group, and **combine** the results.
- The most common operation we apply to each group is an **aggregation**.
  - Remember, aggregation is the act of combining many values into a single value.
- To perform an aggregation, use an aggregation method on the `DataFrameGroupBy` object, e.g. `.mean()`, `.max()`, `.median()`, etc.

Let's look at some examples.

In [39]:
```
penguins_small
```

Out[39]:

	species	body_mass_g	sex
0	Adelie	3750.0	Male
156	Chinstrap	3725.0	Male
308	Gentoo	4875.0	Female
1	Adelie	3800.0	Female
258	Gentoo	4350.0	Female
157	Chinstrap	3950.0	Female
309	Gentoo	5550.0	Male

In [40]:
```
penguins_small.groupby('species')['body_mass_g'].mean()
```

Out[40]:
```
species
Adelie       3775.0
Chinstrap    3837.5
Gentoo       4925.0
Name: body_mass_g, dtype: float64
```

In [41]:
```
# Whoa, what happened in the sex column?
penguins_small.groupby('species').sum()
```

Out[41]:

species	body_mass_g	sex
Adelie	7550.0	MaleFemale
Chinstrap	7675.0	MaleFemale
Gentoo	14775.0	FemaleFemaleMale

In [42]:
```
penguins_small.groupby('species').first()
```

Out[42]:

species	body_mass_g	sex
Adelie	3750.0	Male
Chinstrap	3725.0	Male
Gentoo	4875.0	Female

In [43]:
```
penguins_small.groupby('species').max()
```

Out[43]:

species	body_mass_g	sex
Adelie	3800.0	Male
Chinstrap	3950.0	Male
Gentoo	5550.0	Male

Column independence

Within each group, the aggregation method is applied to **each column independently**.

In [44]:
```
penguins_small.groupby('species').max()
```

Out[44]:

species	body_mass_g	sex
Adelie	3800.0	Male
Chinstrap	3950.0	Male
Gentoo	5550.0	Male

It **is not** telling us that there is a `'Male'` `'Adelie'` penguin with a `'body_mass_g'` of `3800.0`!

In [45]:
```
# This penguin is Female!
penguins_small.loc[(penguins['species'] == 'Adelie') & (penguins['body_mass_
```

Out[45]:

	species	body_mass_g	sex
1	Adelie	3800.0	Female

Question 🤔

Find the `species`, `island`, and `body_mass_g` of the heaviest `Male` and `Female` penguins in `penguins` (not `penguins_small`).

Hint: there's usually more than one way to do things. Do you *need* `.groupby()` for this? *Can* you use `.groupby()` for this?

In [46]:
```
# Your code goes here.
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Column selection and performance implications

- By default, the aggregator will be applied to **all** columns that it can be applied to.
  - `max`, `min`, and `sum` are defined on strings, while `median` and `mean` are not.

- If we only care about one column, we can select that column before aggregating **to save time**.
  - `DataFrameGroupBy` objects support `[]` notation, just like `DataFrame`s.

In [47]:
```
# Back to the big penguins dataset!
penguins
```

Out[47]:

	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 [48]:
```
# Works, but involves wasted effort since the other columns had to be aggreg
penguins.groupby('species').sum()['bill_length_mm']
```

Out[48]:
```
species
Adelie       5668.3
Chinstrap    3320.7
Gentoo       5660.6
Name: bill_length_mm, dtype: float64
```

In [49]:
```
# This is a SeriesGroupBy object!
penguins.groupby('species')['bill_length_mm']
```

Out[49]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x111e40510>
```

In [50]:
```
# Saves time!
penguins.groupby('species')['bill_length_mm'].sum()
```

Out[50]:
```
species
Adelie       5668.3
Chinstrap    3320.7
Gentoo       5660.6
Name: bill_length_mm, dtype: float64
```

To demonstrate that the former is slower than the latter, we can use `%%timeit`. For reference, we'll also include our earlier `for`-loop-based solution.

In [51]:
```
%%timeit
penguins.groupby('species').sum()['bill_length_mm']
```

```
236 µs ± 5.89 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [52]:
```
%%timeit
penguins.groupby('species')['bill_length_mm'].sum()
```

```
80.7 µs ± 879 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [53]:
```
%%timeit
species_map = pd.Series([], dtype=float)
```

```
for species in penguins['species'].unique():
    species_only = penguins.loc[penguins['species'] == species]
    species_map.loc[species] = species_only['body_mass_g'].mean()

species_map
```

```
621 µs ± 22.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Takeaways

- It's important to understand *what* each piece of your code evaluates to – in the first two timed examples, the code is almost identical, but the performance is quite different.

  ```
  # Slower
  penguins.groupby('species').sum()['bill_length_mm']

  # Faster
  penguins.groupby('species')['bill_length_mm'].sum()
  ```
- The `groupby` method is much quicker than `for`-looping over the DataFrame in Python. It can often produce results using just a **single, fast pass** over the data, updating the sum, mean, count, min, or other aggregate for each group along the way.

- You should **always** select the columns you want after `groupby`, unless you really know what you're doing!

Beyond default aggregation methods

- There are many built-in aggregation methods.
- What if you want to apply different aggregation methods to different columns?
- What if the aggregation method you want to use doesn't already exist in `pandas`?

The `aggregate` method

- The `DataFrameGroupBy` object has a general `aggregate` method, which aggregates using one or more operations.
  - Remember, aggregation is the act of combining many values into a single value.
- There are many ways of using `aggregate`; refer to the documentation for a comprehensive list.
- Example arguments:
  - A single function.
  - A list of functions.
  - A dictionary mapping column names to functions.
- Per the documentation, `agg` is an alias for `aggregate`.

Example

How many penguins are there of each `'species'`, and what is the mean `'body_mass_g'` of each `'species'`?

In [54]:
```
(penguins
 .groupby('species')
 ['body_mass_g']
 .aggregate(['count', 'mean'])
)
```

Out[54]:

species	count	mean
Adelie	146	3706.16
Chinstrap	68	3733.09
Gentoo	119	5092.44

Example

What is the maximum `'bill_length_mm'` of each `'species'`, and which `'island'`s is each `'species'` found on?

In [55]:
```
(penguins
 .groupby('species')
 .aggregate({'bill_length_mm': 'max', 'island': 'unique'})
)
```

Out[55]:

species	bill_length_mm	island
Adelie	46.0	[Torgersen, Biscoe, Dream]
Chinstrap	58.0	[Dream]
Gentoo	59.6	[Biscoe]

Example

What is the **interquartile range** of the `'body_mass_g'` of each `'species'`?

In [56]:
```
# Here, the argument to agg is a function,
# which takes in a pd.Series and returns a scalar.

def iqr(s):
    return np.percentile(s, 75) - np.percentile(s, 25)
```

```
(penguins
 .groupby('species')
 ['body_mass_g']
 .agg(iqr)
)
```

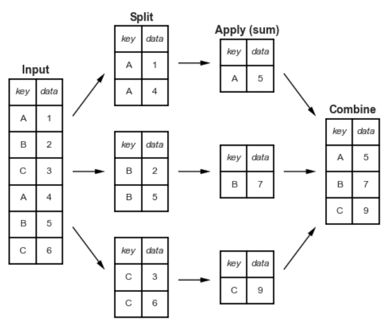
Out[56]:

```
species
Adelie       637.5
Chinstrap    462.5
Gentoo       800.0
Name: body_mass_g, dtype: float64
```

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 [57]:

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

In [58]:

```
z_score(penguins['body_mass_g'])
```

Out[58]:
```
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 [59]:
```
z_mass = (penguins
          .groupby('species')
          ['body_mass_g']
          .transform(z_score))
z_mass
```

Out[59]:
```
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 [60]:
```
penguins.assign(z_mass=z_mass)
```

Out[60]:

	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 [61]:
```
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'`.

- 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 [62]:
```
penguins.groupby('species')['body_mass_g'].mean()
```

Out[62]:
```
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 [63]:
```
(penguins
 .groupby('species')
 .filter(lambda df: df['bill_length_mm'].mean() > 39)
)
```

Out[63]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mas
152	Chinstrap	Dream	46.5	17.9	192.0	350
153	Chinstrap	Dream	50.0	19.5	196.0	390
154	Chinstrap	Dream	51.3	19.2	193.0	365
...	...	...	...	...	...	
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

187 rows × 7 columns

No more `'Adelie'` s!

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

In [64]:
```
(penguins
 .groupby('species')
 .filter(lambda df: df.shape[0] > 100)
)
```

Out[64]:

	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!

Question 🤔

Answer the following questions about grouping:

- In `.agg(fn)`, what is the input to `fn`? What is the output of `fn`?
- In `.transform(fn)`, what is the input to `fn`? What is the output of `fn`?
- In `.filter(fn)`, what is the input to `fn`? What is the output of `fn`?

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 [65]:

```
penguins
```

Out[65]:

	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 [66]:
```
species_and_island = (
    penguins
    .groupby(['species', 'island'])
    [['bill_length_mm', 'body_mass_g']]
    .mean()
)
species_and_island
```

Out[66]:

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

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 [67]:
```
species_and_island
```

Out[67]:

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 [68]:
```
species_and_island['body_mass_g']
```

Out[68]:
```
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 [69]:
```
species_and_island.loc['Adelie']
```

Out[69]:

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

In [70]:
```
species_and_island.loc[('Adelie', 'Torgersen')]
```

Out[70]:
```
bill_length_mm      39.04
body_mass_g       3708.51
Name: (Adelie, Torgersen), dtype: float64
```

In [71]:
```
species_and_island.reset_index()
```

Out[71]:

	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 [72]:
```
(penguins
 .groupby(['species', 'island'], as_index=False)
 [['bill_length_mm', 'body_mass_g']]
 .mean()
)
```

Out[72]:

	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 [73]:
```
baby_path = Path('data') / 'baby.csv'
baby = pd.read_csv(baby_path)
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]:
```
# Your code goes 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` 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 [75]:
```
last_5_years = baby.query('Year >= 2018')
last_5_years
```

Out[75]:

	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 [76]:
```
last_5_years.pivot_table(
    index='Year',
    columns='Sex',
    values='Count',
    aggfunc='sum',
)
```

Out[76]:

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

In [77]:
```
# Look at the similarity to the snippet above!
(last_5_years
 .groupby(['Year', 'Sex'])
 [['Count']]
 .sum()
)
```

Out[77]:

Year	Sex	Count
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 [78]:
```
penguins
```

Out[78]:

	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 [79]:
```
penguins.pivot_table(
    index='species',
    columns='island',
    values='bill_length_mm', # Choice of column here doesn't actually matter
```

```
    aggfunc='count',
)
```

Out[79]:

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 [80]:

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

Out[80]:

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 [81]:

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

Out[81]:

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!

We will most likely end lecture here.

Distributions

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 [82]:

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

Out[82]:

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 [83]:
```
joint = counts / counts.sum().sum()
joint
```

Out[83]:

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 [84]:
```
joint
```

Out[84]:

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

In [85]:
```
# Recall, joint.sum(axis=0) sums across the rows,
# which computes the sum of the **columns**.
joint.sum(axis=0)
```

Out[85]:
```
sex
Female    0.5
Male      0.5
dtype: float64
```

In [86]:
```
joint.sum(axis=1)
```

Out[86]:
```
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 [87]:
```
counts
```

Out[87]:

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 [88]:
```
counts
```

Out[88]:

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

In [89]:
```
counts.sum(axis=0)
```

Out[89]:
```
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 [90]:
```
counts / counts.sum(axis=0)
```

Out[90]:

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 🤔 (Answer at dsc80.com/q)

Code: `dist`

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

Hint: Use `.T`.

In [91]:
```
# Your code goes here.
```

Summary, next time

Summary

- Grouping allows us to change the level of granularity in a DataFrame.
- Grouping involves three steps – split, apply, and combine.
  - Usually, what is applied is an aggregation, but it could be a transformation or filtration.
- `pivot_table` aggregates data based on two categorical columns, and reshapes the result to be "wide" instead of "long".

Next time

- Simpson's paradox.
- Merging.
  - Review this diagram from DSC 10!
- The pitfalls of the `apply` method.