The `groupby` method

- The `groupby` method helps us answer questions that involve performing some computation separately **for each group**.

- Most commonly, we'll use `groupby`, select column(s) to operate on, and use a built-in aggregation method.

aggregation method.

In [5]:
```
# The median 'bill_length_mm' of each 'species'.
penguins.groupby('species')['bill_length_mm'].median()
```

"separately, for each species, ..."

Out[5]:
```
species
Adelie       38.85
Chinstrap    49.55
Gentoo       47.40
Name: bill_length_mm, dtype: float64
```

index automatically set to "species"

- Most commonly, we'll use `groupby`, select column(s) to operate on, and use a built-in aggregation method.

In [5]:
```
# The median 'bill_length_mm' of each 'species'.
penguins.groupby('species')['bill_length_mm'].median()
```

Out[5]:
```
species
Adelie       38.85
Chinstrap    49.55
Gentoo       47.40
Name: bill_length_mm, dtype: float64
```

- There are four other special "grouping methods" we learned about last class that allow for advanced behavior, namely `agg`, `filter`, `transform`, and `apply`.

See "⭐ **The grouping method cheat sheet"** from last lecture for examples.

In [6]:
```
# The most common 'island' per 'species'.
penguins.groupby('species')['island'].agg(lambda s: s.value_counts().idxmax())
```

Out[6]:
```
species
Adelie        Dream
Chinstrap     Dream
Gentoo       Biscoe
Name: island, dtype: object
```

Dream
Dream
Biscoe → Dream
Dream most common.
⋮

In [ ]:
```
# Keeps the 'species' with at least 100 penguins.
...
```

- There are four other special "grouping methods" we learned about last class that allow for advanced behavior, namely `agg`, `filter`, `transform`, and `apply`.

See "⭐ **The grouping method cheat sheet"** from last lecture for examples.

In [6]:
```python
# The most common 'island' per 'species'.
penguins.groupby('species')['island'].agg(lambda s: s.value_counts().idxmax())
```

Out[6]:
```
species
Adelie          Dream
Chinstrap       Dream
Gentoo         Biscoe
Name: island, dtype: object
```

In [7]:
```python
# Keeps the 'species' with at least 100 penguins.
penguins.groupby('species').filter(lambda df: df.shape[0] >= 100)
```

Out[7]:

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Dream	41.3	20.3	194.0	3550.0	Male
1	Adelie	Torgersen	38.5	17.9	190.0	3325.0	Female
2	Adelie	Dream	34.0	17.1	185.0	3400.0	Female
...	...	...	...	...	...	...	...
326	Adelie	Dream	41.1	18.1	205.0	4300.0	Male
331	Adelie	Dream	39.7	17.9	193.0	4250.0	Male
332	Gentoo	Biscoe	45.1	14.5	207.0	5050.0	Female

265 rows × 7 columns

chinstraps missing!

not 333, because one species has fewer than 100 penguins

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 the output below, why are there only 5 rows, rather than $3 \times 3 = 9$ rows, when there are 3 unique `'species'` and 3 unique `'island'`s?

In [8]:
```
# Read this as:
species_and_island = (
    penguins.groupby(['species', 'island'])          # for every combination of 'species' and 'island' in the DataFrame,
    [['bill_length_mm', 'bill_depth_mm']].mean()     # calculate the mean 'bill_length_mm' and the mean 'bill_depth_mm'.
)
species_and_island
```

Out[8]:

		bill_length_mm	bill_depth_mm
species	**island**		
Adelie	**Biscoe**	38.98	18.37
	Dream	38.52	18.24
	Torgersen	39.04	18.45
Chinstrap	**Dream**	48.83	18.42
Gentoo	**Biscoe**	47.57	15.00

Chinstrap penguins only exist on one island!

MultiIndex

7.1

Pivot tables: An extension of grouping

- Pivot tables are a compact way to display tables for humans to read.

sex species	Female	Male
Adelie	3368.84	4043.49
Chinstrap	3527.21	3938.97
Gentoo	4679.74	5484.84

popular in business application

government data.

Pivot tables: An extension of grouping

- Pivot tables are a compact way to display tables for humans to read.

sex	Female	Male
species		
Adelie	3368.84	4043.49
Chinstrap	3527.21	3938.97
Gentoo	4679.74	5484.84

- Notice that each value in the table is the average of `'body_mass_g'` of penguins, for every combination of `'species'` and `'sex'`.

- **You can think of pivot tables as grouping using two columns, then "pivoting" one of the group labels into columns.**

```python
    index='species',
    columns='sex',
    values='body_mass_g',
    aggfunc='mean'
)
```

Out[11]:

sex	Female	Male
species		
Adelie	3368.84	4043.49
Chinstrap	3527.21	3938.97
Gentoo	4679.74	5484.84

easier to read

same info!

In [13]:
```python
penguins.groupby(['species', 'sex'])[['body_mass_g']].mean()
```

Out[13]:

		body_mass_g
species	**sex**	
Adelie	**Female**	3368.84
	Male	4043.49
Chinstrap	**Female**	3527.21
	Male	3938.97
Gentoo	**Female**	4679.74
	Male	5484.84

Out[20]:

```
species    island
Gentoo     Biscoe       119
Chinstrap  Dream         68
Adelie     Dream         55
           Torgersen     47
           Biscoe        44
Name: count, dtype: int64
```

Biscoe Dream Tor. - .

Adelie
Chinstrap
Gentoo

- But the data is arguably easier to interpret when we do use `pivot_table`:

In [ ]:

```
penguins.pivot_table(
    index=''
)
```

```
Torgersen    47
Biscoe       44
Name: count, dtype: int64
```

- But the data is arguably easier to interpret when we do use `pivot_table`:

```
In [22]: penguins.pivot_table(
             index='species',
             columns='island',
             values='body_mass_g',
             aggfunc='count'
         )
```

"values" doesn't really matter if a sing 'count'!

Out[22]:

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

```
        Torgersen    47
        Biscoe       44
Name: count, dtype: int64
```

- But the data is arguably easier to interpret when we do use `pivot_table`:

In [22]:
```
penguins.pivot_table(
    index='species',
    columns='island',
    values='body_mass_g',
    aggfunc='count'
)
```

Out[22]:

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

no Chinstraps on Torgersen!

Example: Phone sales 📱

In [31]:
```
# The DataFrame on the left contains information about phones on the market.
# The DataFrame on the right contains information about the stock I have in my stores.
dfs_side_by_side(phones, inventory)
```

	Model	Price	Screen
0	iPhone 16	799	6.1
1	iPhone 16 Pro Max	1199	6.9
2	Samsung Galaxy S24 Ultra	1299	6.8
3	Pixel 9 Pro	999	6.3

	Handset	Units	Store
0	iPhone 16 Pro Max	50	Briarwood
1	iPhone 16	40	Somerset
2	Pixel 9 Pro	10	Arbor Hills
3	Pixel 9 Pro	15	12 Oaks
4	iPhone 16	100	Briarwood
5	iPhone 15	5	Oakland Mall

50 × 1199
+
40 × 799
+
⋮

- **Question**: If I sell all of the phones in my inventory, how much will I make in revenue?

```
In [41]: phones.merge(inventory, left_on='Model', right_on='Handset', how='inner')
```

Out[41]:

	Model	Price	Screen	Handset	Units	Store
0	iPhone 16	799	6.1	iPhone 16	40	Somerset
1	iPhone 16	799	6.1	iPhone 16	100	Briarwood
2	iPhone 16 Pro Max	1199	6.9	iPhone 16 Pro Max	50	Briarwood
3	Pixel 9 Pro	999	6.3	Pixel 9 Pro	10	Arbor Hills
4	Pixel 9 Pro	999	6.3	Pixel 9 Pro	15	12 Oaks

```
In [42]: phones.merge(inventory, left_on='Model', right_on='Handset', how='left')
```

left right

keeps every row from phones, even ones not in inventory!

Out[42]:

	Model	Price	Screen	Handset	Units	Store
0	iPhone 16	799	6.1	iPhone 16	40.0	Somerset
1	iPhone 16	799	6.1	iPhone 16	100.0	Briarwood
2	iPhone 16 Pro Max	1199	6.9	iPhone 16 Pro Max	50.0	Briarwood
3	Samsung Galaxy S24 Ultra	1299	6.8	NaN	NaN	NaN
4	Pixel 9 Pro	999	6.3	Pixel 9 Pro	10.0	Arbor Hills
5	Pixel 9 Pro	999	6.3	Pixel 9 Pro	15.0	12 Oaks

missing value

```
In [ ]: phones.merge(inventory, left_on='Model', right_on='Handset', how='right')
```

4	Pixel 9 Pro	999	6.3	Pixel 9 Pro	15	12 Oaks

In [42]:
```
phones.merge(inventory, left_on='Model', right_on='Handset', how='left')
```

Out[42]:

	Model	Price	Screen	Handset	Units	Store
0	iPhone 16	799	6.1	iPhone 16	40.0	Somerset
1	iPhone 16	799	6.1	iPhone 16	100.0	Briarwood
2	iPhone 16 Pro Max	1199	6.9	iPhone 16 Pro Max	50.0	Briarwood
3	Samsung Galaxy S24 Ultra	1299	6.8	NaN	NaN	NaN
4	Pixel 9 Pro	999	6.3	Pixel 9 Pro	10.0	Arbor Hills
5	Pixel 9 Pro	999	6.3	Pixel 9 Pro	15.0	12 Oaks

In [43]:
```
phones.merge(inventory, left_on='Model', right_on='Handset', how='right')
```

Out[43]:

	Model	Price	Screen	Handset	Units	Store
0	iPhone 16 Pro Max	1199.0	6.9	iPhone 16 Pro Max	50	Briarwood
1	iPhone 16	799.0	6.1	iPhone 16	40	Somerset
2	Pixel 9 Pro	999.0	6.3	Pixel 9 Pro	10	Arbor Hills
3	Pixel 9 Pro	999.0	6.3	Pixel 9 Pro	15	12 Oaks
4	iPhone 16	799.0	6.1	iPhone 16	100	Briarwood
5	NaN	NaN	NaN	iPhone 15	5	Oakland Mall

keeps all rows from inventory, even if not in phones

In [ ]:
```
phones.merge(inventory, left_on='Model', right_on='Handset', how='outer')
```

0	iPhone 16 Pro Max	1199.0	6.9	iPhone 16 Pro Max	50	Briarwood
1	iPhone 16	799.0	6.1	iPhone 16	40	Somerset
2	Pixel 9 Pro	999.0	6.3	Pixel 9 Pro	10	Arbor Hills
3	Pixel 9 Pro	999.0	6.3	Pixel 9 Pro	15	12 Oaks
4	iPhone 16	799.0	6.1	iPhone 16	100	Briarwood
5	NaN	NaN	NaN	iPhone 15	5	Oakland Mall

In [44]:
```
phones.merge(inventory, left_on='Model', right_on='Handset', how='outer')
```

Out[44]:

	Model	Price	Screen	Handset	Units	Store
0	iPhone 16	799.0	6.1	iPhone 16	40.0	Somerset
1	iPhone 16	799.0	6.1	iPhone 16	100.0	Briarwood
2	iPhone 16 Pro Max	1199.0	6.9	iPhone 16 Pro Max	50.0	Briarwood
3	Samsung Galaxy S24 Ultra	1299.0	6.8	NaN	NaN	NaN
4	Pixel 9 Pro	999.0	6.3	Pixel 9 Pro	10.0	Arbor Hills
5	Pixel 9 Pro	999.0	6.3	Pixel 9 Pro	15.0	12 Oaks
6	NaN	NaN	NaN	iPhone 15	5.0	Oakland Mall

keeps everything!

```
inventory.groupby('Handset')['Units'].sum()
```

	Handset	Units	Store
0	iPhone 16 Pro Max	50	Briarwood
1	iPhone 16	40	Somerset
2	Pixel 9 Pro	10	Arbor Hills
3	Pixel 9 Pro	15	12 Oaks
4	iPhone 16	100	Briarwood
5	iPhone 15	5	Oakland Mall

	Handset	Units	Store
0	iPhone 16 Pro Max	50	Briarwood
1	iPhone 16	40	Somerset
2	Pixel 9 Pro	10	Arbor Hills
3	Pixel 9 Pro	15	12 Oaks
4	iPhone 16	100	Briarwood
5	iPhone 15	5	Oakland Mall

suggest improvement

```
inventory.groupby('Handset')['Units'].sum()
```

	Handset	Units	Store
0	iPhone 16 Pro Max	50	Briarwood
1	iPhone 16	40	Somerset
2	Pixel 9 Pro	10	Arbor Hills
3	Pixel 9 Pro	15	12 Oaks
4	iPhone 16	100	Briarwood
5	iPhone 15	5	Oakland Mall

	Series
0	50
1	40
2	10
3	15
4	100
5	5

suggest improvement

```
inventory.groupby('Handset')['Units'].sum()
```

	Series
0	50
1	40
2	10
3	15
4	100
5	5

Handset	*Series*
Pixel 9 Pro	25
iPhone 15	5
iPhone 16	140
iPhone 16 Pro Max	50

suggest improvement

☐ pin ☐ no-hover URL: https://pandastutor.com/vis.html#code=%23%20The%20code%20below%20just%20i

Report bugs or send feedback | FAQ and unsupported features | Join our private mailing list

```
Name: Model, dtype: object
```

In [46]:
```
left = set(phones['Model'])
left
```

Out[46]:
```
{'Pixel 9 Pro', 'Samsung Galaxy S24 Ultra', 'iPhone 16', 'iPhone 16 Pro Max'}
```

In [47]:
```
inventory['Handset']
```

Out[47]:
```
0    iPhone 16 Pro Max
1            iPhone 16
2          Pixel 9 Pro
3          Pixel 9 Pro
4            iPhone 16
5            iPhone 15
Name: Handset, dtype: object
```

In [48]:
```
right = set(inventory['Handset'])
right
```

Out[48]:
```
{'Pixel 9 Pro', 'iPhone 15', 'iPhone 16', 'iPhone 16 Pro Max'}
```

- To quickly check *which* join key values are in the left DataFrame but not the right, or vice versa, create sets out of the join keys and use the `difference` method.

In [49]:
```
left.difference(right)
```

which elements are in left but not right?

Out[49]:
```
{'Samsung Galaxy S24 Ultra'}
```

In [50]:
```
right.difference(left)
```

Out[50]:
```
{'iPhone 15'}
```

Without writing code, how many rows are in `midwest_cities.merge(schools, on='city')`?

A. 4 B. 5 C. 6 D. 7 E. 8

In [52]:
```
dfs_side_by_side(midwest_cities, schools)
```

	city	state	today_high_temp
0	Ann Arbor	Michigan	79
1	Detroit	Michigan	83
2	Chicago	Illinois	87
3	East Lansing	Michigan	87

	name	city	state	graduation_rate
0	University of Michigan	Ann Arbor	Michigan	0.87
1	University of Chicago	Chicago	Illinois	0.94
2	Wayne State University	Detroit	Michigan	0.78
3	Johns Hopkins University	Baltimore	Maryland	0.92
4	UC San Diego	La Jolla	California	0.81
5	Concordia U-Ann Arbor	Ann Arbor	Michigan	0.83
6	Michigan State University	East Lansing	Michigan	0.91

$2+1+1+1 = 5$

In [ ]:
```
...
```

In [ ]:

Followup activity

Without writing code, how many rows are in `midwest_cities.merge(schools, on='state')`?

4+4+1+4 = 13

```
In [53]: dfs_side_by_side(midwest_cities, schools)
```

	city	state	today_high_temp
0	Ann Arbor	Michigan	79
1	Detroit	Michigan	83
2	Chicago	Illinois	87
3	East Lansing	Michigan	87

	name	city	state	graduation_rate
0	University of Michigan	Ann Arbor	Michigan	0.87
1	University of Chicago	Chicago	Illinois	0.94
2	Wayne State University	Detroit	Michigan	0.78
3	Johns Hopkins University	Baltimore	Maryland	0.92
4	UC San Diego	La Jolla	California	0.81
5	Concordia U Ann Arbor	Ann Arbor	Michigan	0.83
6	Michigan State University	East Lansing	Michigan	0.91

(don't sue me)

↑ kind of a butterfly wing

```
In [ ]: ...
```

```
In [ ]:
```

13 rows × 6 columns

In [57]:
```
dfs_side_by_side(midwest_cities, schools)
```

	city	state	today_high_temp
0	Ann Arbor	Michigan	79
1	Detroit	Michigan	83
2	Chicago	Illinois	87
3	East Lansing	Michigan	87

	name	city	state	graduation_rate
0	University of Michigan	Ann Arbor	Michigan	0.87
1	University of Chicago	Chicago	Illinois	0.94
2	Wayne State University	Detroit	Michigan	0.78
3	Johns Hopkins University	Baltimore	Maryland	0.92
4	UC San Diego	La Jolla	California	0.81
5	Concordia U-Ann Arbor	Ann Arbor	Michigan	0.83
6	Michigan State University	East Lansing	Michigan	0.91

In [56]:
```
midwest_cities.merge(schools)
```

if you don't specify the "on", it merges on all shared column names

Out[56]:

	city	state	today_high_temp	name	graduation_rate
0	Ann Arbor	Michigan	79	University of Michigan	0.87
1	Ann Arbor	Michigan	79	Concordia U-Ann Arbor	0.83
2	Detroit	Michigan	83	Wayne State University	0.78
3	Chicago	Illinois	87	University of Chicago	0.94
4	East Lansing	Michigan	87	Michigan State University	0.91

localhost

- The Series `apply` method allows us to use a function on every element in a Series.

In [93]:
```
def clean_term(term_string):
    return int(term_string.split()[0])
```

In [92]:
```
int('60 months'.split()[0])
```

Out[92]:
```
60
```

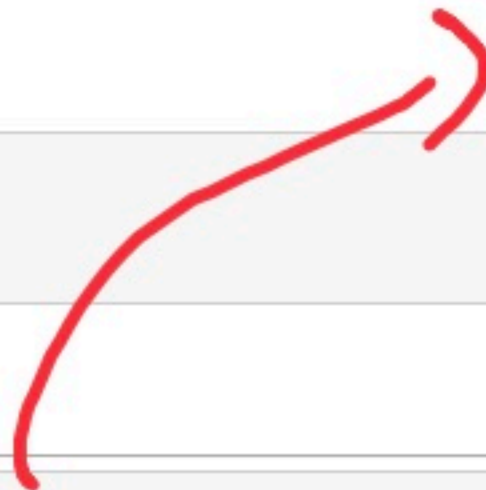
clean_term: one string → one int

In [94]:
```
clean_term('544 months')
```

Out[94]:
```
544
```

In [95]:
```
clean_term('36 months')
```

Out[95]:
```
36
```



In [96]:
```
loans['term'].apply(clean_term)
```

Out[96]:
```
0       60
1       36
2       36
        ..
6297    60
6298    60
6299    60
Name: term, Length: 6300, dtype: int64
```

- Mental model: the operation that comes after `.str` is used on every element of the Series that comes before `.str`.

`s.str.operation`

In [101]:
```
# Here, we use .split() on every string in loans['term'].
loans['term']
```

Out[101]:
```
0        60 months
1        36 months
2        36 months
           ...
6297     60 months
6298     60 months
6299     60 months
Name: term, Length: 6300, dtype: object
```

In [103]:
```
loans['term'].str.split()
```

using .split() on every string in loans['term']

Out[103]:
```
0       [60, months]
1       [36, months]
2       [36, months]
            ...
6297    [60, months]
6298    [60, months]
6299    [60, months]
Name: term, Length: 6300, dtype: object
```

In [ ]:

In [ ]:
```
...
```

In [101]:
```python
# Here, we use .split() on every string in loans['term'].
loans['term']
```

Out[101]:
```
0       60 months
1       36 months
2       36 months
          ...
6297    60 months
6298    60 months
6299    60 months
Name: term, Length: 6300, dtype: object
```

In [103]:
```python
loans['term'].str.split()
```

Out[103]:
```
0       [60, months]
1       [36, months]
2       [36, months]
            ...
6297    [60, months]
6298    [60, months]
6299    [60, months]
Name: term, Length: 6300, dtype: object
```

In [110]:
```python
loans['term'].str.split().str[0].astype(int)
```

Out[110]:
```
0       60
1       36
2       36
        ..
6297    60
6298    60
6299    60
Name: term, Length: 6300, dtype: int64
```

one Series

does [0] on every list in the one Series

In [ ]: ...

- When dealing with values containing dates and times, it's good practice to convert the values to "timestamp" objects.

In [111]:
```
# Stored as strings.
loans['issue_d']
```

Out[111]:
```
0       Jun-2014
1       Jun-2017
2       Dec-2016
          ...
6297    Nov-2015
6298    Dec-2014
6299    Jun-2015
Name: issue_d, Length: 6300, dtype: object
```

- To do so, we use the `pd.to_datetime` function.

It takes in a date format string; you can see examples of how they work here.

In [112]:
```
pd.to_datetime(loans['issue_d'])
```

Out[112]:
```
0       2014-06-01
1       2017-06-01
2       2016-12-01
           ...
6297    2015-11-01
6298    2014-12-01
6299    2015-06-01
Name: issue_d, Length: 6300, dtype: datetime64[ns]
```

each elt is a datetime

- There are a few steps we've performed to clean up our dataset.
  - Convert loan `'term'`s to integers.
  - Convert loan issue dates, `'issue_d'`s, to timestamps.

- When we manipulate DataFrames, it's best to define individual functions for each step, then use the `pipe` **method** to chain them all together.

The `pipe` method takes in a function that maps $\texttt{DataFrame} \rightarrow \texttt{anything}$, but typically $\texttt{anything}$ is a $\texttt{DataFrame}$.

takes in a DF

returns a DF.

In [118]:
```
def clean_term_column(df):
    return df.assign(
        term=df['term'].str.split().str[0].astype(int)
    )
def clean_date_column(df):
    return (
        df
        .assign(date=pd.to_datetime(df['issue_d'], format='%b-%Y'))
        .drop(columns=['issue_d'])
    )
```

In [ ]:
```
...
```

In [ ]:
```
# Same as above, just way harder to read and write.
...
```