AI Bootcamp

Grouping, Aggregating, and Binning Data with Pandas

Module 5 Day 2

Class Objectives

By the end of class, you will be able to:

1. Understand the concept of grouping data and explain its role in data analysis.
2. Differentiate between single and multiple aggregations when grouping data.
3. Apply one or more aggregation functions to grouped data.
4. Use the agg() function effectively to perform various aggregations on grouped data.
5. Use custom Python functions to transform grouped data.
6. Group data into multi-index and apply aggregations.
7. Understand the concept of binning and its application in data segmentation and analysis.

Instructor **Demonstration**

Pandas Grouping

The groupby() function allows you to **group pandas objects based on a common record**.

Reasons for Grouping Data

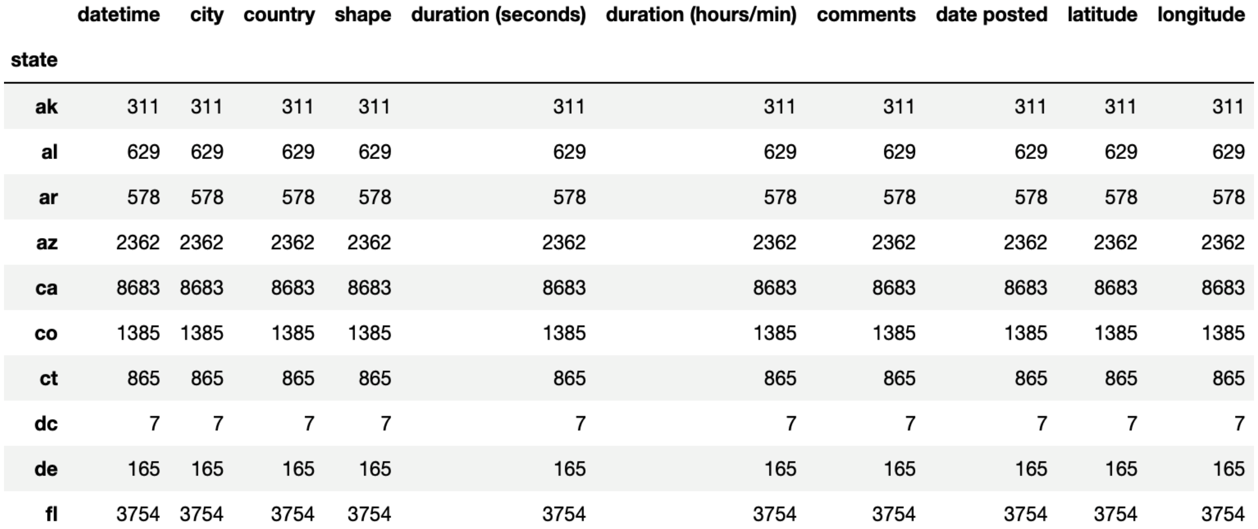
- Aggregation
- Data transformation
- Statistical analysis
- Data exploration

Pandas Grouping

```
# Use `groupby` to separate the data into fields according to state values.
grouped_usa_df = usa_ufo_df.groupby(['state'])

# The object returned is a GroupBy object and cannot be viewed normally.
print(grouped_usa_df)

# In order to be visualized, a data function must be used.
grouped_usa_df.count().head(10)
```



	datetime	city	country	shape	duration (seconds)	duration (hours/min)	comments	date posted	latitude	longitude
state										
ak	311	311	311	311	311	311	311	311	311	311
al	629	629	629	629	629	629	629	629	629	629
ar	578	578	578	578	578	578	578	578	578	578
az	2362	2362	2362	2362	2362	2362	2362	2362	2362	2362
ca	8683	8683	8683	8683	8683	8683	8683	8683	8683	8683
co	1385	1385	1385	1385	1385	1385	1385	1385	1385	1385
ct	865	865	865	865	865	865	865	865	865	865
dc	7	7	7	7	7	7	7	7	7	7
de	165	165	165	165	165	165	165	165	165	165
fl	3754	3754	3754	3754	3754	3754	3754	3754	3754	3754

Pandas Grouping

We can create a series using only GroupBy data. This is done by taking the pd.DataFrame() method and passing the desired GroupBy data in as the parameter.

Since duration (seconds) was converted to a numeric time, it can now be summed up per state.

```
state_duration = grouped_usa_df["duration (seconds)"].sum()
state_duration.head()
```

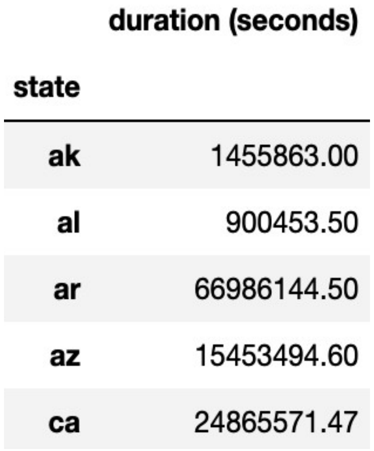
```
state
ak                  1455863.00
al                   900453.50
ar          66986144.50
az          15453494.60
ca          24865571.47
Name: duration (seconds), dtype: float64
```

Pandas Grouping

It is possible to create a DataFrames from GroupBy data by adding double brackets around the column name as follows:

By adding double brackets around the column we can create a DataFrame.

```
state_duration_df = grouped_usa_df[["duration (seconds)"]].sum()
state_duration_df.head()
```

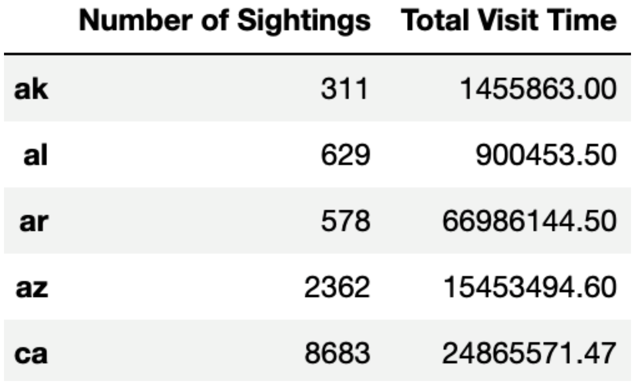


	duration (seconds)
state	
ak	1455863.00
al	900453.50
ar	66986144.50
az	15453494.60
ca	24865571.47

Pandas Grouping

It is possible to create new DataFrames using only GroupBy data. This can be done by taking the pd.DataFrame() method and passing the desired GroupBy data in dictionary format as the parameter.

```
# Create a new DataFrame using both duration and count.
state_summary_table = pd.DataFrame({"Number of Sightings": state_counts,
                                    "Total Visit Time": state_duration})
state_summary_table.head()
```



	Number of Sightings	Total Visit Time
ak	311	1455863.00
al	629	900453.50
ar	578	66986144.50
az	2362	15453494.60
ca	8683	24865571.47

Activity:

Training GroupBy

In this activity, you will use the groupby() function to calculate the average weight and membership length of gym members per trainer.

Suggested Time:

15 Minutes

Time’s up!
Let’s review

Questions?

How would we **sort the DataFrame** from the longest to the shortest membership in days?

sort_values(by='Membership (Days)',
ascending=False)
on the trainers_means DataFrame.

Questions?

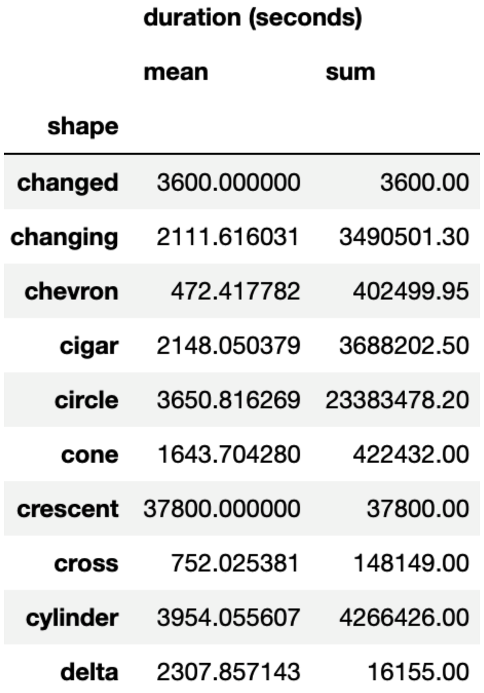
Instructor **Demonstration**

Multi-Index Aggregations on DataFrames

Multi-Aggregations on DataFrames

To apply two aggregations on one column we surround the column to which the aggregation will be applied in double brackets and pass in the aggregations in the agg() function as follows:

```
# The agg() function can be used to pass more than one aggregation.
ufo_shape_avg_sum = converted_ufo_df.groupby("shape")[["duration
(seconds)"]].agg(['mean','sum'])
ufo_shape_avg_sum.head(10)
```



	duration (seconds)	
	mean	**sum**
shape		
changed	3600.000000	3600.00
changing	2111.616031	3490501.30
chevron	472.417782	402499.95
cigar	2148.050379	3688202.50
circle	3650.816269	23383478.20
cone	1643.704280	422432.00
crescent	37800.000000	37800.00
cross	752.025381	148149.00
cylinder	3954.055607	4266426.00
delta	2307.857143	16155.00

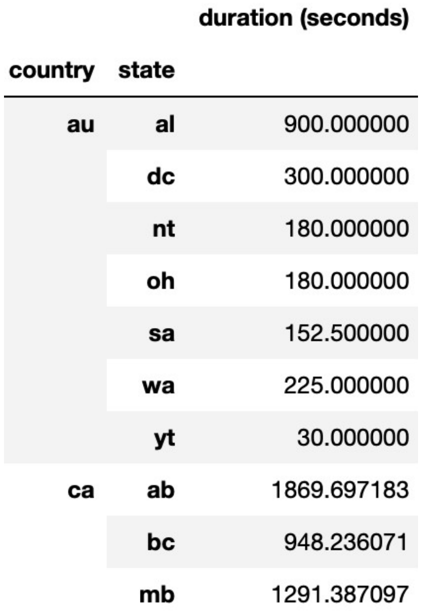
Grouping on Multiple Columns

We can create sophisticated data analysis by grouping data into multiple columns by performing multiple aggregations.

To group data on multiple columns we pass the desired columns in brackets to the groupby() function as follows:

```
# Get the average duration in seconds of UFOs by country and state.
country_state_avg_duration = converted_ufo_df.groupby(['country', 'state'])
[["duration (seconds)"]].mean()

country_state_avg_duration.head(10)
```



		duration (seconds)
country	**state**	
au	**al**	900.000000
	dc	300.000000
	nt	180.000000
	oh	180.000000
	sa	152.500000
	wa	225.000000
	yt	30.000000
ca	**ab**	1869.697183
	bc	948.236071
	mb	1291.387097

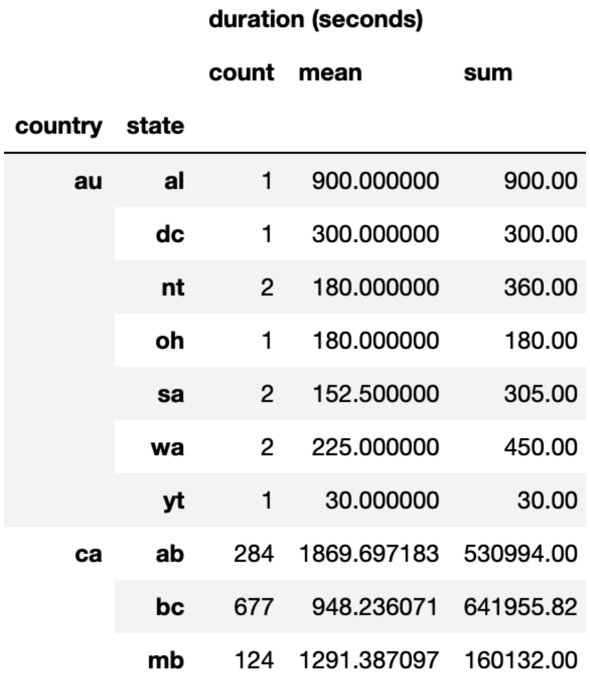
Grouping on Multiple Columns using Multiple Aggregations

To perform multiple aggregations on multi-indices we surround the column to which the aggregation will be applied in double brackets and pass in the aggregations in the agg() function as follows:

```
# The agg() function can be used to pass more than one aggregation.

country_state_duration_metrics = converted_ufo_df.groupby(['country', 'state'])[["duration (seconds)"]].agg(['count','mean','sum'])

country_state_duration_metrics.head(10)
```



		duration (seconds)		
		count	mean	sum
country	state			
au	al	1	900.000000	900.00
	dc	1	300.000000	300.00
	nt	2	180.000000	360.00
	oh	1	180.000000	180.00
	sa	2	152.500000	305.00
	wa	2	225.000000	450.00
	yt	1	30.000000	30.00
ca	ab	284	1869.697183	530994.00
	bc	677	948.236071	641955.82
	mb	124	1291.387097	160132.00

Flattening Multi-Indexed Columns to Single Columns

There are two common methods to flatten multi-indexed columns to single columns.

First, we can use the to_flat_index() method on the columns of the multi-indexed columns as follows:

```
country_state_duration_flatten = country_state_duration_metrics.copy()

# Get the columns after apply the to_flat_index().
country_state_duration_flatten.columns = country_state_duration_flatten.columns.to_flat_index()

# Display the columns.
Country_state_duration_flatten.columns
```

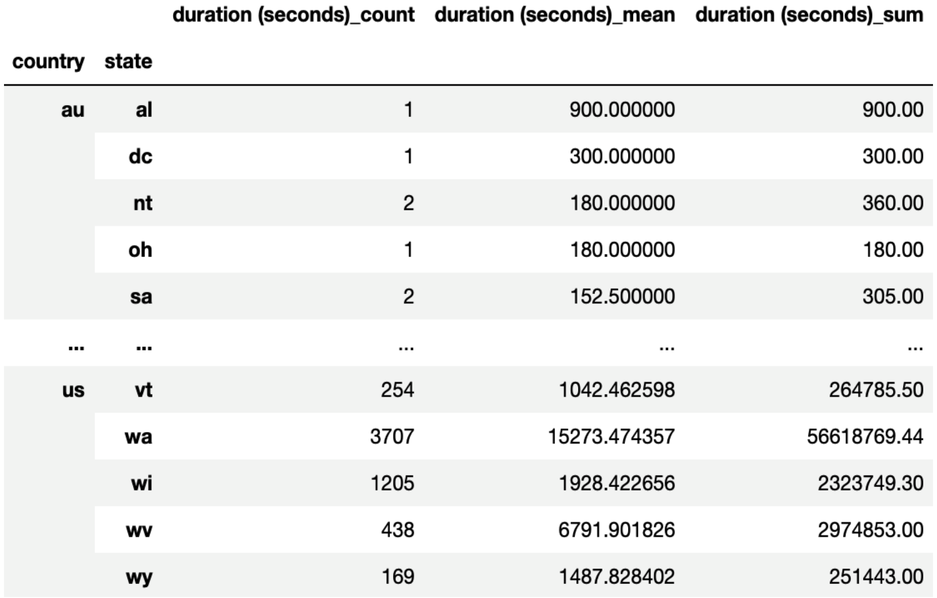
The output is a list of tuples that hold the column names:

```
Index([('duration (seconds)', 'count'), ('duration (seconds)', 'mean'),
       ('duration (seconds)', 'sum')],
      dtype='object')
```

Flattening Multi-Indexed Columns to Single Columns

Next, we can convert the multi-index columns to single columns by using a list comprehension and join the items in each tuple to create our new columns as follows:

```
# Use a list comprehension to join the each tuple for each column.
country_state_duration_flatten.columns =
['_'.join(column) for column in
country_state_duration_metrics.columns]
# Display the flattened DataFrame
country_state_duration_flatten
```



		duration (seconds)_count	duration (seconds)_mean	duration (seconds)_sum
country	**state**			
au	**al**	1	900.000000	900.00
	dc	1	300.000000	300.00
	nt	2	180.000000	360.00
	oh	1	180.000000	180.00
	sa	2	152.500000	305.00
...	**...**	...	...	...
us	**vt**	254	1042.462598	264785.50
	wa	3707	15273.474357	56618769.44
	wi	1205	1928.422656	2323749.30
	wv	438	6791.901826	2974853.00
	wy	169	1487.828402	251443.00

Flattening Multi-Indexed Columns to Single Columns

The second method is to get the levels of the multi-index by using the get_level_values() method and passing the value for each level, i.e., “0”, “1”, etc.

```
# Get the first level of the multi-index.
level_0 = country_state_duration_metrics.columns.get_level_values(0)
print(level_0)
# Get the second level of the multi-index
level_1 = country_state_duration_metrics.columns.get_level_values(1)
print(level_1)
```

The output for each level is a list that holds the column names:

```
Index([‘duration (seconds)’, ‘duration (seconds)’, ‘duration (seconds)’], dtype=’object’)
Index([‘count’, ‘mean’, ‘sum’], dtype=’object’)
```

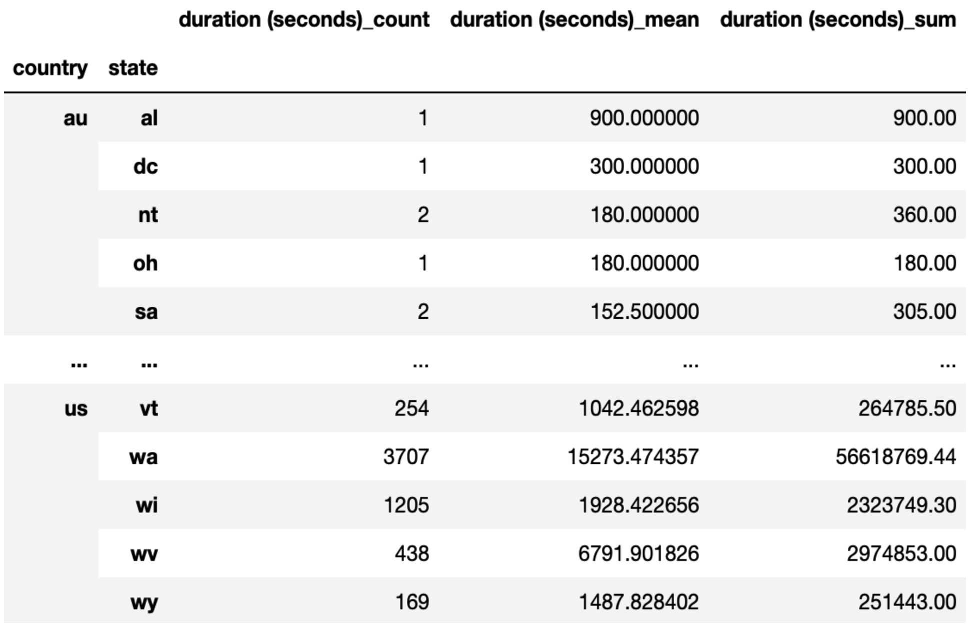
Flattening Multi-Indexed Columns to Single Columns

Then, we combine the level names as follows:

```
# Combine the levels.

country_state_duration_metrics.columns = level_0 +"_"+ level_1

country_state_duration_metrics
```



		duration (seconds)_count	duration (seconds)_mean	duration (seconds)_sum
country	**state**			
au	**al**	1	900.000000	900.00
	dc	1	300.000000	300.00
	nt	2	180.000000	360.00
	oh	1	180.000000	180.00
	sa	2	152.500000	305.00
...	**...**	...	...	...
us	**vt**	254	1042.462598	264785.50
	wa	3707	15273.474357	56618769.44
	wi	1205	1928.422656	2323749.30
	wv	438	6791.901826	2974853.00
	wy	169	1487.828402	251443.00

Activity:

Airline Delays Multi-Index and Aggregations

In this activity, you will practice creating multiple indices and aggregations to gain insights into airline delays.

Suggested Time:

15 Minutes

Time’s up!
Let’s review

Questions?

Break

15 mins

Instructor **Demonstration**

Customizing Aggregations

Creating Custom Functions

We can create our own custom functions to help us summarize data.

- First, we write a Python function that takes a parameter.
- If we want to calculate the average of that parameter, we apply the mean() function.

```
# Create a custom function that will calculate the average of DataFrame column.
def custom_avg(x):
    return x.mean()
```

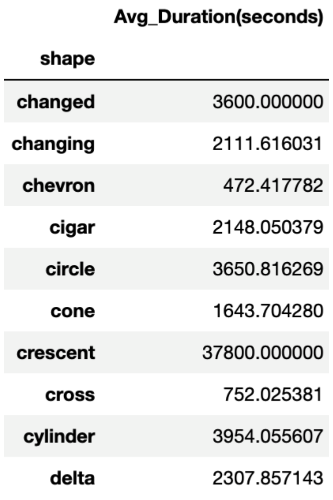
Applying Custom Functions for Aggregations

In the GroupBy object we “apply” the custom function using the apply() function.

- In the apply() function, we will call our custom function on every item in the "duration (seconds)" column by using lambda x:.
- The x refers to the pd.Series() data from the "duration (seconds)" column.
- The one-dimensional data x["duration (seconds)"] will be passed to the custom_avg() function.
- The aggregation column will be named "Avg_Duration(seconds)".

```
# Show the average time in seconds for each UFO shape.
avg_ufo_duration_shape = converted_ufo_df.groupby("shape").apply(lambda x:
pd.Series({"Avg_Duration(seconds)": custom_avg(x["duration (seconds)"])}))

# Display the DataFrame.
avg_ufo_duration_shape
```



	Avg_Duration(seconds)
shape	
changed	3600.000000
changing	2111.616031
chevron	472.417782
cigar	2148.050379
circle	3650.816269
cone	1643.704280
crescent	37800.000000
cross	752.025381
cylinder	3954.055607
delta	2307.857143

Creating Multiple Custom Functions

```
# First, create two custom functions.
# 1) One function calculates the total count of items for a DataFrame column.
def custom_count(x):
    return x.count()

# 2) The second adds up the values for a DataFrame column.
def custom_sum(x):
    return x.sum()
```

Applying Multiple Custom Functions for Aggregations

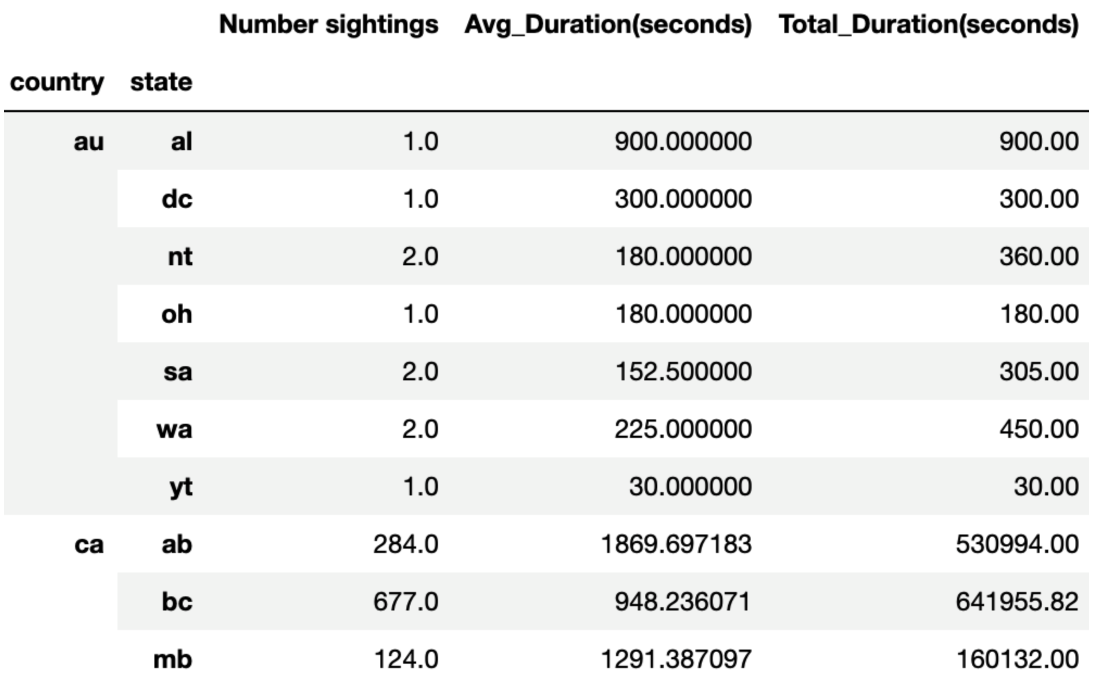
We can pass in multiple custom functions in the apply() function.

```
# Use the custom functions to determine total sightings, and average and total duration in seconds.
country_state_total_avg = converted_ufo_df.groupby(['country', 'state']).\
    apply(lambda x: pd.Series({"Number sightings": custom_count(x["duration (seconds)"]),
                              "Avg_Duration(seconds)": custom_avg(x["duration (seconds)"]),
                              "Total_Duration(seconds)": custom_sum(x["duration (seconds)"])}))



# Display the DataFrame.
country_state_total_avg.head(10)
```

Applying Multiple Custom Functions for Aggregations



		Number sightings	Avg_Duration(seconds)	Total_Duration(seconds)
country	state			
au	al	1.0	900.000000	900.00
	dc	1.0	300.000000	300.00
	nt	2.0	180.000000	360.00
	oh	1.0	180.000000	180.00
	sa	2.0	152.500000	305.00
	wa	2.0	225.000000	450.00
	yt	1.0	30.000000	30.00
ca	ab	284.0	1869.697183	530994.00
	bc	677.0	948.236071	641955.82
	mb	124.0	1291.387097	160132.00

Activity:

Customizing Delayed Flight Aggregations

In this activity, you will practice creating custom functions to use for aggregations to gain insights into airline delays.

Suggested Time:

20 Minutes

Time’s up!
Let’s review

Questions?

Instructor **Demonstration**

Binning DataFrames

The **binning method** places values into groups to enable more vigorous dataset customization.

Binning Data: Creating bins and groups

First, we create the bins in which the data will be held and the group names for the bins.

```
  # Bins are 0, 59.9, 69.9, 79.9, 89.9, 100.
bins = [0, 59.9, 69.9, 79.9, 89.9, 100]

# Create the names for the five bins.
group_names = ["F", "D", "C", "B", "A"]
```

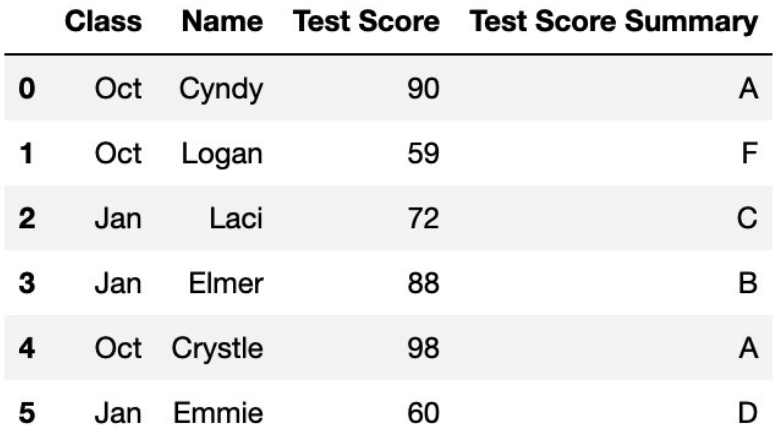
The bins will have the following cutoff values: (0, 59.9], (60, 69.9], (70, 79.9], (80, 89.9], (90, 100].

Binning Data with pd.cut()

Use pd.cut() when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable.

We add the names of the five bins in a new column, “Test Score Summary,” where the cut function is applied.

```
# Slice the data and place it into bins.
test_scores_df["Test Score Summary"] =
pd.cut(test_scores_df["Test Score"], bins, labels=group_name
include_lowest=True)
test_scores_df
```

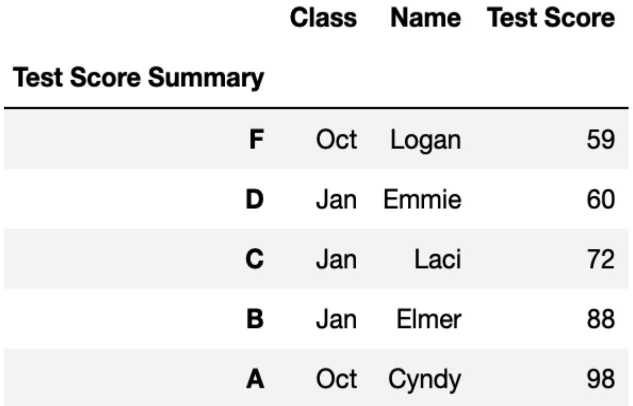


	Class	Name	Test Score	Test Score Summary
0	Oct	Cyndy	90	A
1	Oct	Logan	59	F
2	Jan	Laci	72	C
3	Jan	Elmer	88	B
4	Oct	Crystle	98	A
5	Jan	Emmie	60	D

Binning Data

Binning is powerful because we can group the DataFrame according to those values, and then conduct a higher-level analysis.

```
# Create a group based off of the bins.
test_scores_df = test_scores_df.groupby("Test Score Summary")
test_scores_df.max()
```



	Class	Name	Test Score
Test Score Summary			
F	Oct	Logan	59
D	Jan	Emmie	60
C	Jan	Laci	72
B	Jan	Elmer	88
A	Oct	Cyndy	98

Activity:

Binning Movies

You will now put your binning skills to use by creating bins for movies based on their IMDd user vote count. After creating the bins, group the DataFrame based on those bins and then perform some analysis on them.

Suggested Time:

20 Minutes

Time’s up!
Let’s review

Questions?

Recap

After today’s lesson you are able to:

1. Understand the concept of grouping data and explain its role in data analysis.
2. Differentiate between single and multiple aggregations when grouping data.
3. Apply one or more aggregation functions to grouped data.
4. Use the agg() function effectively to perform various aggregations on grouped data.
5. Use custom Python functions to transform grouped data.
6. Group data into multi-index and apply aggregations.
7. Understand the concept of binning and its application in data segmentation and analysis.

Next

In the next lesson, we will start exploring pivoting, pivoting with multi-index and multi-aggregations, and reshaping data.

Questions?

The End