Intro-Data-Visualization

February 16, 2023

1 DATA VISUALIZATION WITH PYTHON

1.1 LESSON 1: INTRODUCTION TO DATA VISUALIZATION

Data visualization refers to the process of creating graphical representations of data in order to effectively communicate information and insights. This can include charts, graphs, maps, and other types of visualizations that make it easier to understand patterns, trends, and relationships within large sets of data.

There are two main reasons for creating visuals using data:

- **Exploratory analysis** is done when you are searching for insights. These visualizations don't need to be perfect. You are using plots to find insights, but they don't need to be aesthetically appealing. You are the consumer of these plots, and you need to be able to find the answer to your questions from these plots.
- **Explanatory analysis** is done when you are providing your results for others. These visualizations need to provide you the emphasis necessary to convey your message. They should be accurate, insightful, and visually appealing.

There are five steps to **Data Analysis**:

1. **Gathering Data** - Collecting data from a variety of sources, including databases, CSV files, and web pages.
2. **Cleaning Data** - Fixing errors, removing duplicates, and filling in missing data.
3. **Exploring Data** - Finding patterns, anomalies, and outliers.
4. **Analyzing Data** - Using statistical methods to answer questions. Here, we can use **Explanatory** or **exploratory** visuals.
5. **Share** - Share your **Explanatory** Visuals

1.1.1 Python Data Visualization Libraries

In this course, you will make use of the following libraries for creating data visualizations:

- **Matplotlib**: a versatile library for visualizations, but it can take some coding effort to put together common visualizations.
- **Seaborn**: built on top of matplotlib, adds a number of functions to make common statistical visualizations easier to generate.
- **pandas**: while this library includes some convenient methods for visualizing data that hook into matplotlib, we'll mainly use it for its main purpose as a general tool for working with data.

1.2 LESSON 2: DESIGN OF VISUALIZATIONS

Before getting into the actual creation of visualizations later in the course, this lesson introduces design principles that will be useful both in exploratory and explanatory analysis. You will learn about different data types and ways of encoding data. You will also learn about properties of visualizations that can impact both the clarity of messaging as well as their accuracy.

In this lesson, you'll learn about the following topics related to the design of data visualizations.

- What makes a bad visual?
- Levels of measurement and types of data
- Continuous vs. discrete data
- Identifying data types
- What experts say about visual encodings
- Chart Junk
- Data-to-ink ratio
- Design integrity
- Using color and designing for color blindness
- Shape, size, and other tools

Visuals can be bad if they:

- Don't convey the desired message.
- Are misleading.

1.2.1 The Four Levels of Measurement

There are four levels of measurement that can be used to describe data: Qualitative or categorical types (non-numeric types) 1. **Nominal data**: pure labels without inherent order (no label is intrinsically greater or less than any other). Example of nominal data include: 1. Gender 2. Type of a fruit 3. Nationality 4. Genre of a movie 2. **Ordinal data**: labels with an intrinsic order or ranking (comparison operations can be made between values, but the magnitude of differences are not be well-defined). Example of ordinal data include: 1. Size of a shirt 2. Rating of a restaurant 3. Level of education 4. Letter grade in a class (A, B, C, D, F)

Quantitative or numeric types 1. **Interval data**: numeric values where absolute differences are meaningful (addition and subtraction operations can be made) 2. **Ratio data**: numeric values where relative differences are meaningful (multiplication and division operations can be made)

All quantitative-type variables also come in one of two varieties: **discrete** and **continuous**.

- **Discrete** quantitative variables can only take on a specific set values at some maximum level of precision. Examples include:
  - Number of children in a family,
  - Number of times a person has been to the doctor
  - Number of pages in a book
  - Number of students in a class
- **Continuous** quantitative variables can (hypothetically) take on values to any level of precision. Examples include:
  - Height of a person
  - Weight of a person
  - Temperature

– Amount of money in a bank account

1.2.2 Chart Junk

Chart junk is any visual element that is not necessary for conveying the message of the visualization. Chart junk can include: - Gridlines - 3D effects - Drop shadows - Unnecessary text

1.2.3 Data-to-Ink Ratio

The data-to-ink ratio is a measure of how much of the visual is used to convey the data versus how much is used to convey the visual itself.

1.2.4 Design Integrity

Design integrity is the idea that the visual should be designed to convey the message as clearly as possible. This means that the visual should be designed to be as simple as possible while still conveying the message. This can include:

- Removing unnecessary elements
- Using color and shape judiciously
- Using a consistent style
- Using a consistent color scheme

1.2.5 Color

Color can both help and hurt a data visualization.

Three tips for using color effectively. - Before adding color to a visualization, start with black and white. - When using color, use less intense colors - not all the colors of the rainbow, which is the default in many software applications. - Color for communication. Use color to highlight your message and separate groups of interest. Don't add color just to have color in your visualization.

1.2.6 Color Blindness

Color blindness is a condition where a person is unable to distinguish between certain colors. This can include: - Red and green - Blue and yellow

1.2.7 Shape, Size, and Other Tools

In addition to color, there are other tools that can be used to convey information in a visualization. These include: - Shape - Size - Orientation - Texture

1.3 LESSON 3: UNIVARIATE EXPLORATION OF DATA

1.3.1 Univariate Exploration

In this lesson, you'll learn about the first step in the data analysis process: **univariate exploration**. This is the process of looking at one variable at a time. You'll learn about the different types of plots that can be used for univariate exploration, and you'll learn how to use the pandas plotting functions to create these plots.

1.3.2 Lesson Outcomes

The current lesson will focus on introducing univariate visualizations: bar charts, and histograms. By the end of this lesson, you will be able to: - Create bar charts for qualitative variables, for example, the amount (number) of eggs consumed in a meal (categories: {breakfast, lunch, or dinner}). In general, bar chart maps categories to numbers. - Create Pie charts. A pie chart is a common univariate plot type that is used to depict relative frequencies for levels of a categorical variable. A pie chart is preferably used when the number of categories is less, and you'd like to see the proportion of each category. - Create histograms for quantitative variables. A histogram splits the (tabular) data into evenly sized intervals and displays the count of rows in each interval with bars. A histogram is similar to a bar chart, except that the "category" here is a range of values. - Analyze the bar charts and histograms.

1.3.3 What is Tidy Data?

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is tidy when: 1. Each variable forms a column. 2. Each observation forms a row. 3. Each type of observational unit forms a table.

1.3.4 Bar Charts

Bar charts are a common univariate plot type that is used to depict the count of observations for each level of a categorical variable. A bar chart is preferable when the number of categories is less, and you'd like to see the proportion of each category.

- For nominal data, the bars can be ordered by frequency to easily see which category is the most common.
- Ordinal data should not be re-ordered because the inherent ordering of the levels is typically more important to display.

Bar Charts Using Seaborn Seaborn has a function called `countplot()` that can be used to create bar charts. The function takes in a dataframe and the name of a column as arguments. The function will then create a bar chart with the counts of each category in the column.

```
import seaborn as sns
sns.countplot(data = df, x = 'column_name')
```

[ ]:
```
# Necessary imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
%matplotlib inline
```

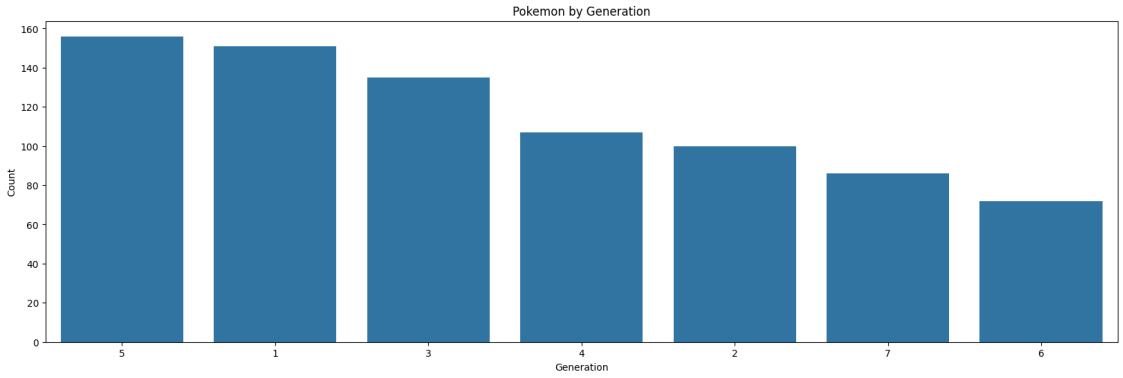
[ ]:
```
# Load in the dataset pokemon.csv in the folder data and assign it to the␣
 ↪variable pokemon
pokemon = pd.read_csv('data/pokemon.csv')
# Print the first 5 rows of the dataset
pokemon.head()
```

[ ]:

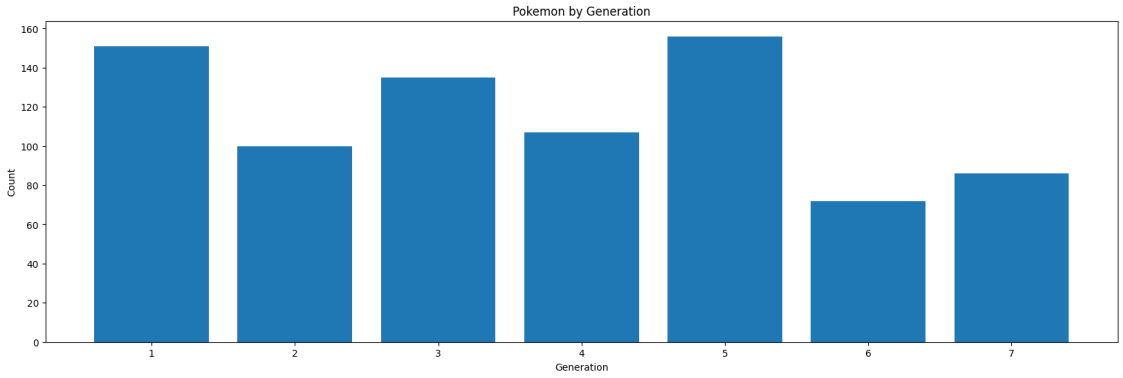
	id	species	generation_id	height	weight	base_experience	type_1	\
0	1	bulbasaur	1	0.7	6.9	64	grass	
1	2	ivysaur	1	1.0	13.0	142	grass	
2	3	venusaur	1	2.0	100.0	236	grass	
3	4	charmander	1	0.6	8.5	62	fire	
4	5	charmeleon	1	1.1	19.0	142	fire	

	type_2	hp	attack	defense	speed	special-attack	special-defense
0	poison	45	49	49	45	65	65
1	poison	60	62	63	60	80	80
2	poison	80	82	83	80	100	100
3	NaN	39	52	43	65	60	50
4	NaN	58	64	58	80	80	65

[ ]:
```
# Using Seaborn to plot a countplot of the pokemon generation column
plt.figure(figsize=(20,6))
plt.title('Pokemon by Generation')
color = sb.color_palette()[0]
order = pokemon['generation_id'].value_counts().index
# Plot the countplot
sb.countplot(data=pokemon, x='generation_id', color=color, order=order)
plt.xlabel('Generation')
plt.ylabel('Count');
```



[ ]:
```
# Create the same plot as above but this time use the matplotlib library
plt.figure(figsize=(20,6))
plt.title('Pokemon by Generation')
color = sb.color_palette()[0]
order = pokemon['generation_id'].value_counts().index
# Plot the barplot using plt.bar() and value_counts()
plt.bar(order, pokemon['generation_id'].value_counts(), color=color)
plt.xlabel('Generation')
plt.ylabel('Count');
```



Absolute Vs. Relative Frequency **Absolute frequency** and **relative frequency** are both ways to describe the number of times an event or outcome occurs in a data set.

Absolute frequency is the number of times an event or outcome occurs in a data set. For example, in a data set of 100 people, if 25 people have brown hair, the **absolute frequency** of brown hair is 25.

Relative frequency is the proportion of times an event or outcome occurs in a data set. It is calculated by dividing the **absolute frequency** by the total number of events or outcomes in the data set. In the above example, the **relative frequency** of brown hair would be 25/100, or 0.25.

Here is an example to illustrate the difference between Absolute and **relative frequency**:

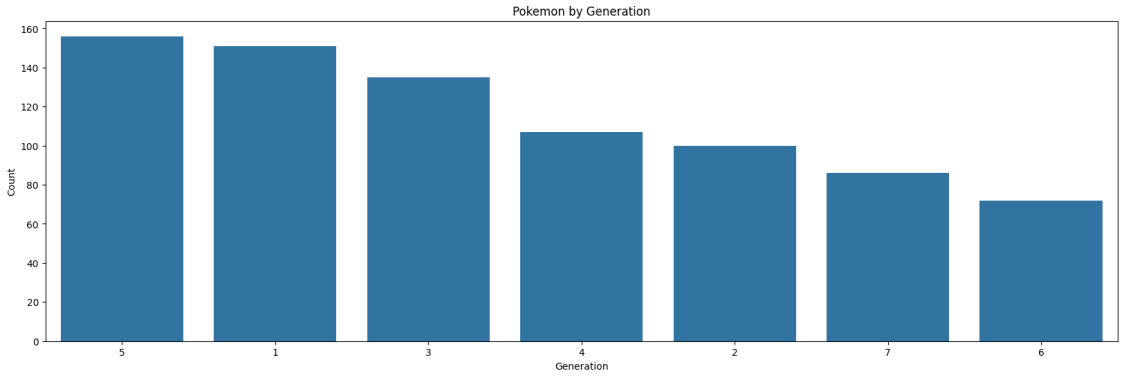
Let's say we have a data set of 1000 students, and we want to know how many students have a grade of A in their final exam. The data shows that there are 250 students with A grade.

- **Absolute frequency**: 250 students have A grade
- **Relative frequency**: 250/1000 = 0.25 or 25% Let's say we have another data set of 100 students and we want to know how many students have a grade of A in their final exam. The data shows that there are 25 students with A grade.
- **Absolute frequency**: 25 students have A grade
- **Relative frequency**: 25/100 = 0.25 or 25% In both cases, the **relative frequency** is the same (25%), but the **absolute frequency** is different, which is representative of the different sample size.

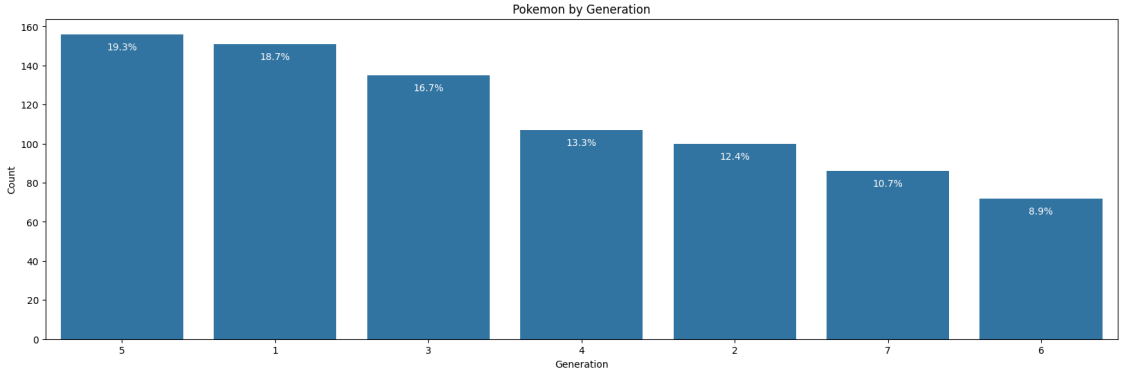
Relative frequency is often used when comparing different data sets or when comparing the same data set but with different sample sizes. **Absolute frequency** is useful to get a raw count of how many times an event or outcome occurs.

[ ]:
```
# Recreate the plot above using the seaborn library, with absolute
plt.figure(figsize=(20,6))
plt.title('Pokemon by Generation')
color = sb.color_palette()[0]
order = pokemon['generation_id'].value_counts().index
# Plot the countplot
```

```
sb.countplot(data=pokemon, x='generation_id', color=color, order=order)
plt.xlabel('Generation')
plt.ylabel('Count');
```



[ ]:
```
# Recreate the plot above using the seaborn library, with relative frequencies
plt.figure(figsize=(20,6))
plt.title('Pokemon by Generation')
color = sb.color_palette()[0]
order = pokemon['generation_id'].value_counts().index
# Plot the countplot
sb.countplot(data=pokemon, x='generation_id', color=color, order=order)
plt.xlabel('Generation')
plt.ylabel('Count')
# Add the code to display the relative frequencies
n_points = pokemon.shape[0]
cat_counts = pokemon['generation_id'].value_counts()
locs, labels = plt.xticks()
for loc, label in zip(locs, labels):
    count = cat_counts[int(label.get_text())]
    pct_string = '{:0.1f}%'.format(100*count/n_points)
    plt.text(loc, count-8, pct_string, ha = 'center', color = 'w')
```



1.3.5 Counting Missing Data

If you have a large dataframe, and it contains a few missing values (`None` or a `numpy.NaN`), then you can find the count of such missing value across the given label. For this purpose, you can use either of the following two analogous functions :

1. pandas.DataFrame.isna()
2. pandas.DataFrame.isnull()

[ ]:
```
# Read the sales_data.csv file in the data folder and assign it to the variable␣
↪sales_data
sales_data = pd.read_csv('data/sales_data.csv')
# Print the first 5 rows of the dataset
sales_data.head()
```

[ ]:
```
   Store        Date  Temperature  Fuel_Price  MarkDown1  MarkDown2  \
0      1  05/02/2010        42.31       2.572        NaN        NaN
1      1  12/02/2010        38.51       2.548        NaN        NaN
2      1  19/02/2010        39.93       2.514        NaN        NaN
3      1  26/02/2010        46.63       2.561        NaN        NaN
4      1  05/03/2010        46.50       2.625        NaN        NaN

   MarkDown3  MarkDown4  MarkDown5         CPI  Unemployment  IsHoliday
0        NaN        NaN        NaN  211.096358         8.106      False
1        NaN        NaN        NaN  211.242170         8.106       True
2        NaN        NaN        NaN  211.289143         8.106      False
3        NaN        NaN        NaN  211.319643         8.106      False
4        NaN        NaN        NaN  211.350143         8.106      False
```

[ ]:
```
# Show all the null values in the dataset
sales_data.isnull()
```

[ ]:
```
      Store   Date  Temperature  Fuel_Price  MarkDown1  MarkDown2  MarkDown3  \
0     False  False        False       False       True       True       True
1     False  False        False       False       True       True       True
2     False  False        False       False       True       True       True
3     False  False        False       False       True       True       True
4     False  False        False       False       True       True       True
…     …    …           …          …         …          …          …
8185  False  False        False       False      False      False      False
8186  False  False        False       False      False      False      False
8187  False  False        False       False      False      False      False
8188  False  False        False       False      False      False      False
8189  False  False        False       False      False      False      False

      MarkDown4  MarkDown5    CPI  Unemployment  IsHoliday
```

```
0          True       True  False         False      False
1          True       True  False         False      False
2          True       True  False         False      False
3          True       True  False         False      False
4          True       True  False         False      False
…           …          …     …              …          …
8185      False      False   True          True      False
8186      False      False   True          True      False
8187      False      False   True          True      False
8188      False      False   True          True      False
8189      False      False   True          True      False

[8190 rows x 12 columns]
```

We can use pandas functions to create a table with the number of missing values in each column. Once, you have the label-wise count of missing values, you try plotting the tabular data in the form of a bar chart.

[ ]:
```
sales_data.isna().sum()
```

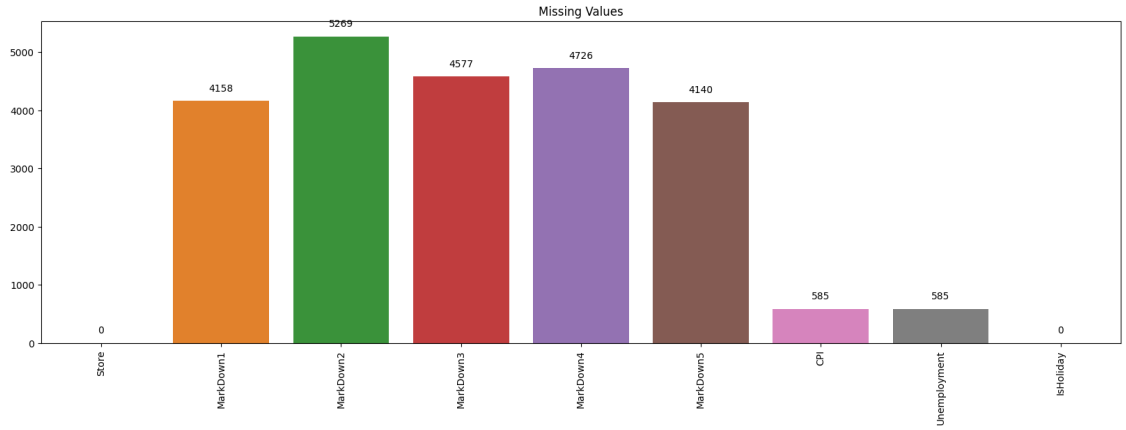
[ ]:
```
Store              0
Date               0
Temperature        0
Fuel_Price         0
MarkDown1       4158
MarkDown2       5269
MarkDown3       4577
MarkDown4       4726
MarkDown5       4140
CPI              585
Unemployment     585
IsHoliday          0
dtype: int64
```

[ ]:
```
na_counts = sales_data.drop(['Date', 'Temperature', 'Fuel_Price'], axis=1).
 ↪isna().sum()
print(na_counts)
```

```
Store              0
MarkDown1       4158
MarkDown2       5269
MarkDown3       4577
MarkDown4       4726
MarkDown5       4140
CPI              585
Unemployment     585
IsHoliday          0
dtype: int64
```

Plot the bar chart from the NaN tabular data, and also print values on each bar

```
[ ]: # Plot the bar chart from the NaN tabular data, and also print values on each␣
      ↪bar using the annotate function and barplot function from the seaborn library
     # Set the figure size to 20 by 6
     plt.figure(figsize=(20,6))
     # Set the title to 'Missing Values'
     plt.title('Missing Values')
     # Plot the countplot
     sb.barplot(x=na_counts.index.values, y=na_counts)

     # get the current tick locations and labels
     plt.xticks(rotation=90)

     # Logic to print value on each bar
     for i in range (na_counts.shape[0]):
         count = na_counts[i]
         # Refer here for details of the text() - https://matplotlib.org/3.1.1/api/
      ↪_as_gen/matplotlib.pyplot.text.html
         plt.text(i, count+300, count, ha = 'center', va='top')
```



Each Pokémon species has either `type_1`, `type_2` or both types that play a part in its offensive and defensive capabilities. The code below creates a new dataframe `pkmn_types` that club the rows of both `type_1` and `type_2`, so that the resulting dataframe has new column, `type_level`

The code below is using the pandas library in python to reshape a dataframe called `pokemon` using the `melt()` function.

The `melt()` function is used to **"unpivot"** a DataFrame from wide format to long format. The function takes several arguments: - **`pokemon`** is the DataFrame that is being reshaped - **`id_vars`** is a list of column(s) that should be used as identifier variables. In this case, the `id` and `species` columns are used as identifier variables and will not be reshaped. - **`value_vars`** is a list of column(s) that should be melted and turned into variable values. In this case, the `type_1` and `type_2` columns will be melted and turned into variable values - **`var_name`** is the name of the variable column. In

this case, it will be named `type_level` - **`value_name`** is the name of the value column. In this case, it will be named `type`

The final result will be a new DataFrame with a multi-level index, where the columns specified in **`id_vars`** will become the index and the columns specified in **`value_vars`** will be melted and become two columns: **`type_level`** and **`type`** columns.

For example, if the original DataFrame had columns `id`, `species`, `type_1`, and `type_2`, after using this code, you would end up with a new DataFrame with columns `id`, `species`, `type_level` and `type`, and where `type_1` and `type_2` values were stacked in two columns `type_level` and `type`

[ ]:
```
pokemon.shape
```

[ ]:
```
(807, 14)
```

[ ]:
```
# Using melt function, Select the 'id', and 'species' columns from␣
 ↪pokemon,Remove the 'type_1', 'type_2' columns from pokemon and assign the␣
 ↪result to the variable pkmn_types
pkmn_types = pd.melt(pokemon, id_vars=['id', 'species'], value_vars=['type_1',␣
 ↪'type_2'], var_name='type_level', value_name='type').dropna()
# Print the first 5 rows of the dataset
pkmn_types.head(), pkmn_types.shape
```

[ ]:
```
(   id     species type_level   type
 0   1   bulbasaur     type_1  grass
 1   2     ivysaur     type_1  grass
 2   3    venusaur     type_1  grass
 3   4  charmander     type_1   fire
 4   5  charmeleon     type_1   fire,
 (1212, 4))
```

The DataFrame is first melted using the `pd.melt()` method. This reshapes the DataFrame from wide to long format, with `id` and `species` columns as identifier variables and `type_1` and `type_2` columns as measured variables.

The resulting DataFrame is assigned to `pkmn_types` and then filtered using `dropna()` to remove any rows that contain missing data.

The output of the code is the first five rows and the shape of the resulting DataFrame, which shows the number of rows and columns in the DataFrame.

1.3.6 Lets remind ourselves of the melt() function

[ ]:
```
import pandas as pd

df = pd.DataFrame({
    'Name': ['John', 'Sarah', 'Peter'],
    'Math': [90, 80, 95],
    'Science': [85, 95, 92]
})
```

```
df
```

```
    Name  Math  Science
0   John    90       85
1  Sarah    80       95
2  Peter    95       92
```

To demonstrate the `melt()` function, let's say we want to reshape this DataFrame so that the columns `Math` and `Science` are melted into a single column called `Subject`, and the corresponding scores are melted into a `Score` column. We can do this using the melt function as follows:

```
melted_df = pd.melt(df, id_vars=['Name'], value_vars=['Math', 'Science'],␣
↪var_name='Subject', value_name='Score')

melted_df
```

```
    Name  Subject  Score
0   John     Math     90
1  Sarah     Math     80
2  Peter     Math     95
3   John  Science     85
4  Sarah  Science     95
5  Peter  Science     92
```

As you can see, the `melt()` function has transformed the original DataFrame by stacking the `Math` and `Science` columns on top of each other, and creating two new columns `Subject` and `Score`. The `id_vars` parameter specifies the columns to use as **identifier variables** (in this case, just `Name`). The `value_vars` parameter specifies the **columns to melt** (in this case, `Math` and `Science`). The `var_name` parameter specifies the **name of the column that will contain the column names** of the original DataFrame (in this case, `Subject`). The `value_name` parameter specifies the **name of the column that will contain the values** of the melted columns (in this case, 'Score').

To transform the melted DataFrame back to the original DataFrame, you can use the `pivot()` function in pandas. The `pivot()` function allows you to "unmelt" a DataFrame by reshaping it into its original shape.

Here's how you can use the pivot function to transform the melted DataFrame melted_df back to the original DataFrame:

```
original_df = melted_df.pivot(index='Name', columns='Subject', values='Score').
↪reset_index()
original_df
```

```
Subject   Name  Math  Science
0         John    90       85
1        Peter    95       92
2        Sarah    80       95
```

In the `pivot()` function, the `index` parameter specifies **the column(s) to use as the index**, the `columns` parameter specifies **the column to pivot**, and the `values` parameter specifies the **column to use as the values**.

In this case, we pivot on the `Subject` column, with `Name` as the index and `Score` as the values. The resulting DataFrame is transposed back to the original shape using the `reset_index()` method.

[ ]:
```
# Display the frequency of each type_level in the type_level column using the␣
 ↪value_counts function
pkmn_types['type_level'].value_counts()
```

[ ]:
```
type_1    807
type_2    405
Name: type_level, dtype: int64
```

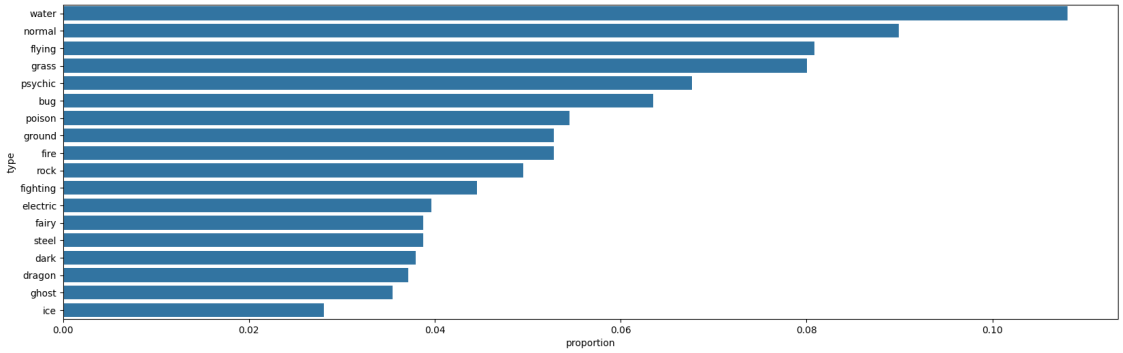
Your task is to use this dataframe to create a relative frequency plot of the proportion of Pokémon with each type, sorted from most frequent to least.

[ ]:
```
type_counts = pkmn_types['type'].value_counts()
# type_order the bars by the type column
type_order = type_counts.index
# Get the sum of all non-null values in the type column
n_pokemon = pkmn_types['type'].value_counts().sum()
# Get the count of each level in the type column
n_pokemon
```

[ ]:
```
1212
```

[ ]:
```
# Get the maximum count of the type_level column
max_type_count = type_counts[0]
# Calculate the maximum proportion
max_prop = max_type_count / n_pokemon
print(max_prop, max_type_count,)

# Calculate the tick locations and create an array of base tick locations
tick_props = np.arange(0, max_prop, 0.02)
tick_names = ['{:0.2f}'.format(v) for v in tick_props]
# Create the plot
plt.figure(figsize=(20,6))
base_color = sb.color_palette()[0]
sb.countplot(data=pkmn_types, y='type', color=base_color, order=type_order)
plt.xticks(tick_props * n_pokemon, tick_names)
plt.xlabel('proportion');
```

```
0.10808580858085809 131
```



1.3.7 Pie Charts

A pie chart is a type of chart that is used to display the relative proportions of different categories or groups in a dataset. In Seaborn, a library for data visualization in Python, a pie chart can be created using the pieplot() function. The basic syntax for creating a pie chart in Seaborn is:

```
plt.pie(df['count'], labels=df['fruits'], explode=explode,
autopct='%1.1f%%', shadow=False, startangle=90)
```

1.3.8 Donut Plot

A donut plot is a type of pie chart that is used to display the relative proportions of different categories or groups in a dataset. In Seaborn, a library for data visualization in Python, a donut plot can be created using the pieplot() function. The basic syntax for creating a donut plot in Seaborn is:

```
plt.pie(df['count'], labels=df['fruits'], explode=explode,
shadow=False, startangle=90, wedgeprops={'width':0.4})
```

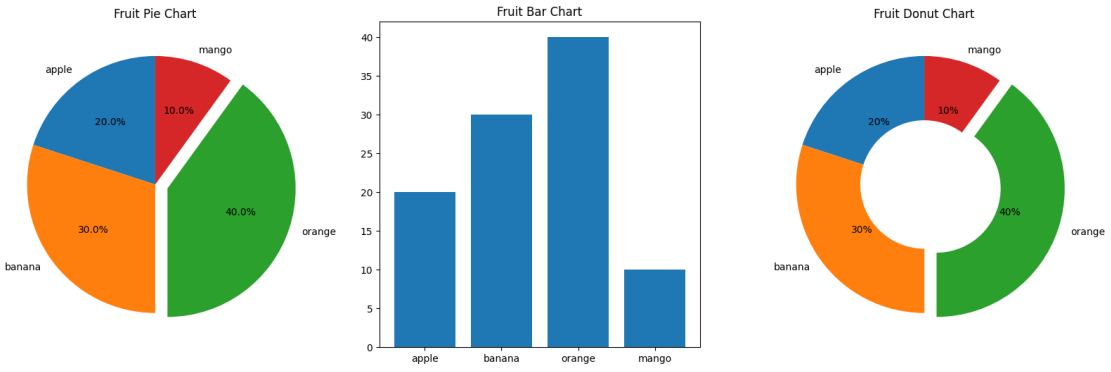
The x and y parameters are used to specify the data that will be plotted in the chart. The hue parameter can be used to group the data by a certain variable. The data parameter is used to specify the dataframe that contains the data. The order parameter is used to specify the order of the categories or groups in the chart. The hue_order parameter is used to specify the order of the subgroups.

The orient parameter is used to specify the orientation of the chart. The color parameter is used to specify the color of the chart. The palette parameter is used to specify the color palette that will be used for the chart. The saturation parameter is used to specify the saturation of the colors in the chart. The width parameter is used to specify the width of the chart.

The ax parameter is used to specify the matplotlib axis object that the chart will be plotted on. Additional keyword arguments can also be passed to the function to customize the appearance of the chart.

For example:

```
[ ]: import seaborn as sns
     import matplotlib.pyplot as plt

     # sample data
     data = {'fruits':['apple','banana','orange','mango'],
             'count':[20,30,40,10]}

     df = pd.DataFrame(data)
     palette_color = sns.color_palette('dark')
     # declaring exploding pie
     explode = [0, 0, 0.1, 0]


     # Create three subplots, one for the pie chart, one for the bar chart and one␣
      ↪for the donut chart using the subplots function from the matplotlib library
     fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,6))
     # Set the title for the pie chart to 'Fruit Pie Chart'
     ax1.set_title('Fruit Pie Chart')
     # Plot the pie chart
     ax1.pie(df['count'], labels=df['fruits'], explode=explode, autopct='%1.1f%%',␣
      ↪shadow=False, startangle=90)
     ax2.set_title('Fruit Bar Chart')
     ax2.bar(df['fruits'], df['count'])
     ax3.set_title('Fruit Donut Chart')
     ax3.pie(df['count'], labels=df['fruits'], explode=explode, autopct='%1.0f%%',␣
      ↪shadow=False, startangle=90, wedgeprops={'width': 0.5});
```



1.3.9 Histograms

A histogram is used to plot the distribution of a numeric variable. It's the quantitative version of the bar chart. However, rather than plot one bar for each unique numeric value, values are grouped into continuous bins, and one bar for each bin is plotted to depict the number. You can use either Matplotlib or Seaborn to plot the histograms. There is a mild variation in the specifics, such as plotting gaussian-estimation line along with bars in Seabron's distplot(), and the arguments that

you can use in either case.

The basic syntax for creating a histogram in Matplotlib is:

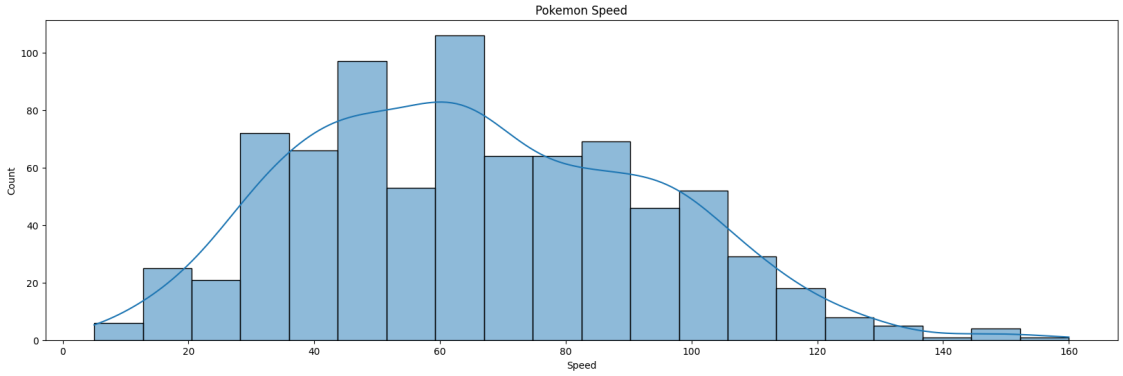
```
sb.histplot(pokemon['speed'], kde=True, bins=20)
```

1.3.10 Kernel Density Estimation

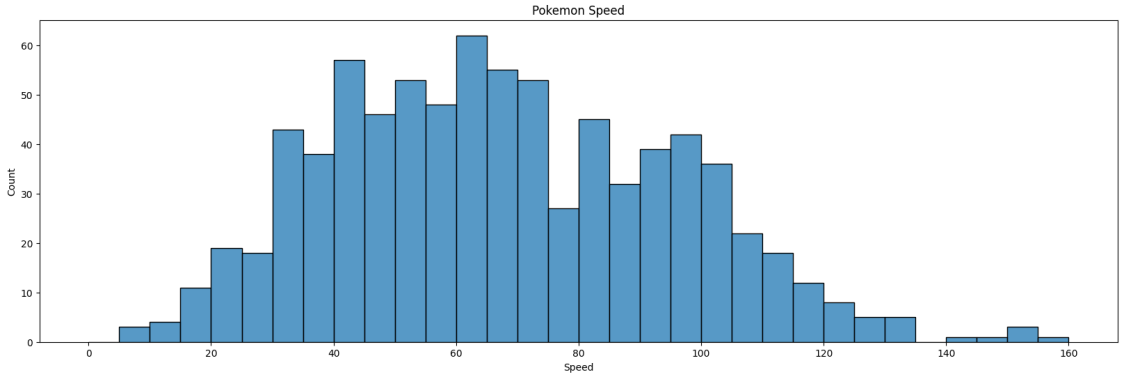
Kernel Density Estimation (KDE) is a method used to estimate the probability density function (PDF) of a random variable. It is a non-parametric way to estimate the distribution of a dataset. In simple terms, it is a way to smooth out a histogram of data points to create a continuous representation of the underlying data distribution.

The basic syntax for creating a KDE plot in Seaborn is: "'python sb.kdeplot(pokemon['speed'])

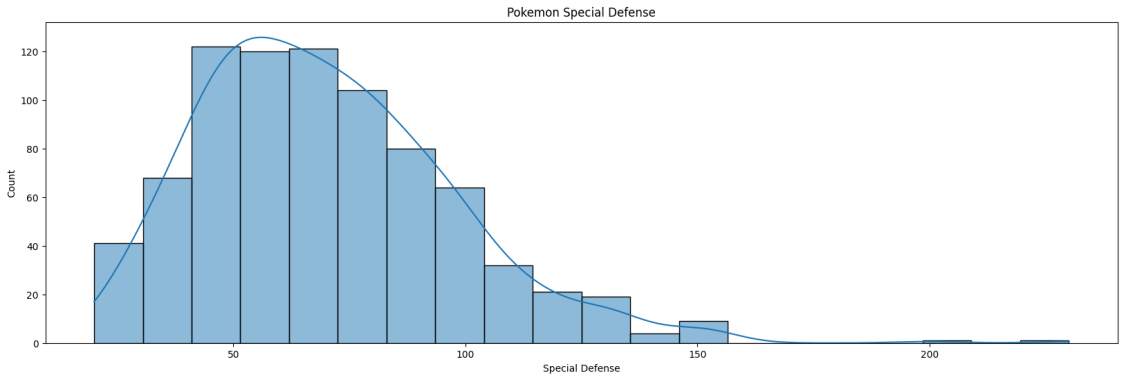
[ ]:
```
# Using pokemon dataframe, plot a histogram of the 'speed' column using the␣
↪seaborn library displot function
plt.figure(figsize=(20,6))
plt.title('Pokemon Speed')
# Plot the histogram
sb.histplot(pokemon['speed'], kde=True, bins=20)
plt.xlabel('Speed')
plt.ylabel('Count');
```



[ ]:
```
# Create bins with step size 5
bins = np.arange(0, pokemon['speed'].max()+5, 5)
# Plot the histogram using the bins and seaborn library histplot function
plt.figure(figsize=(20,6))
plt.title('Pokemon Speed')
sb.histplot(pokemon['speed'], bins=bins)
plt.xlabel('Speed')
plt.ylabel('Count');
```



[ ]:
```
# Using pokemon dataframe, plot a histogram of the 'special-defense' column␣
  ↪using the seaborn library displot function
plt.figure(figsize=(20,6))
# Set the title to 'Pokemon Special Defense'
plt.title('Pokemon Special Defense')
# Plot the histogram
sb.histplot(pokemon['special-defense'], kde=True, bins=20)
plt.xlabel('Special Defense')
plt.ylabel('Count');
```



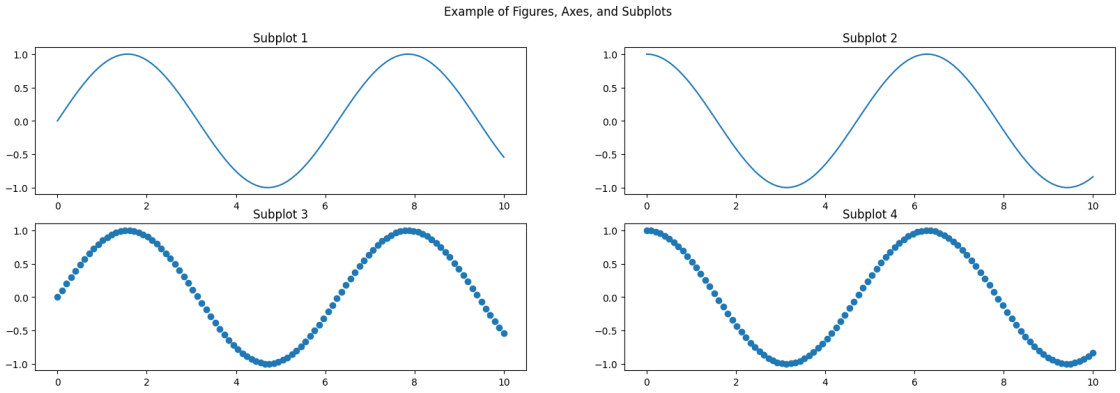
1.3.11 Figures, Axes, and Subplots

In Matplotlib, a **figure** is the overall window or page that contains one or more plots. It is an instance of the matplotlib.figure.Figure class and can be created using the `plt.figure()` function. The figure can contain one or multiple plots, and it contains all the elements of the visualization such as titles, labels, and legends.

An **axes** is an individual plot or chart within a figure. It is an instance of the matplotlib.axes._subplots.AxesSubplot class and is created using the `add_subplot()` method of the figure. The axes contains the data that is plotted, such as the x and y coordinates, as well as the tick marks, labels, and other details of the plot.

A **subplot** is a plotting area within an individual figure that contains one or more axes. It allows you to place multiple plots within a single figure. You can create a figure with a specific number of subplots using the `plt.subplots()` function. The function returns a figure and an array of axes, allowing you to easily create multiple plots in a single figure.

[ ]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Create a 2x2 grid of subplots
fig, axs = plt.subplots(2, 2, figsize=(20,6))

# Generate some data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Plot the first subplot
axs[0, 0].plot(x, y1)
axs[0, 0].set_title('Subplot 1')

# Plot the second subplot
axs[0, 1].plot(x, y2)
axs[0, 1].set_title('Subplot 2')

# Plot the third subplot
axs[1, 0].scatter(x, y1)
axs[1, 0].set_title('Subplot 3')

# Plot the fourth subplot
axs[1, 1].scatter(x, y2)
axs[1, 1].set_title('Subplot 4')

# Add title to the overall figure
fig.suptitle('Example of Figures, Axes, and Subplots')

# Show the plot
plt.show()
```



In this example, `plt.subplots(2, 2)` creates a 2x2 grid of subplots, which is stored in the variables `fig` and `axs`. The variable `fig` is the overall figure that contains all the subplots, while `axs` is an array of axes that represent each individual subplot.

In the following lines, `axs[0, 0].plot(x, y1)` plots the data `x` and `y1` on the first subplot, and `axs[0, 0].set_title('Subplot 1')` sets the title of the first subplot. Similarly, the second, third, and fourth subplots are plotted and titled.

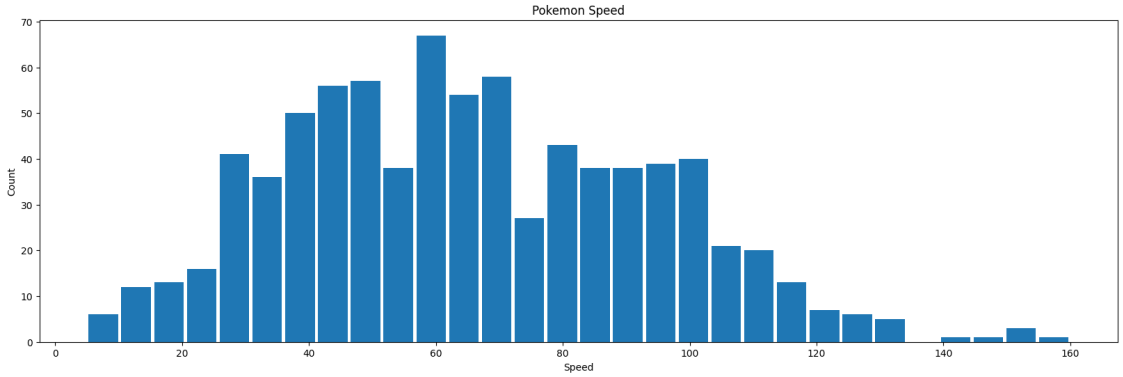
`fig.suptitle('Example of Figures, Axes, and Subplots')`sets the title of the overall figure. Finally, `plt.show()` displays the figure with all the subplots in it.

1.3.12 Choosing a Plot for Discrete Data

If you want to plot a **discrete quantitative** variable, it is possible to select either a *histogram* or a *bar chart* to depict the data.

- Here, the discrete means non-continuous values. In general, a discrete variable can be assigned to any of the limited (countable) set of values from a given set/range, for example, the number of family members, number of football matches in a tournament, number of departments in a university.
- The quantitative term shows that it is the outcome of the measurement of a quantity.

[ ]:
```
# Create a histogram of the 'speed' column using the pokemon dataframe and add␣
  ↪spacing between the bars using rwidth
plt.figure(figsize=(20,6))
# Set the title to 'Pokemon Speed'
plt.title('Pokemon Speed')
# Plot the histogram and add spacing between the bars using rwidth
plt.hist(pokemon['speed'], rwidth=0.9, bins=30)
plt.xlabel('Speed')
plt.ylabel('Count');
```



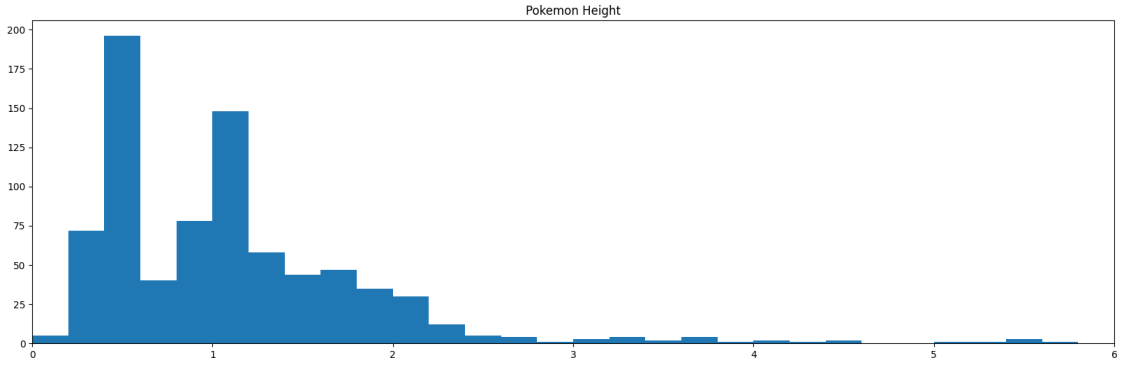
By adding gaps between bars, you emphasize the fact that the data is discrete in value. On the other hand, plotting your quantitative data in this manner might cause it to be interpreted as ordinal-type data, which can have an effect on overall perception.

For continuous numeric data, you should not make use of the "rwidth" parameter, since the gaps imply discreteness of value.

1.3.13 Descriptive Statistics, Outliers and Axis Limits

Visualizations will give you insights into the data that you can't get from descriptive statistics. A plot can show: - If the data is symmetric or skewed - Interesting areas for further investigation or clarification - Potential errors in the data In a histogram, you can observe whether or not there are outliers in your data.

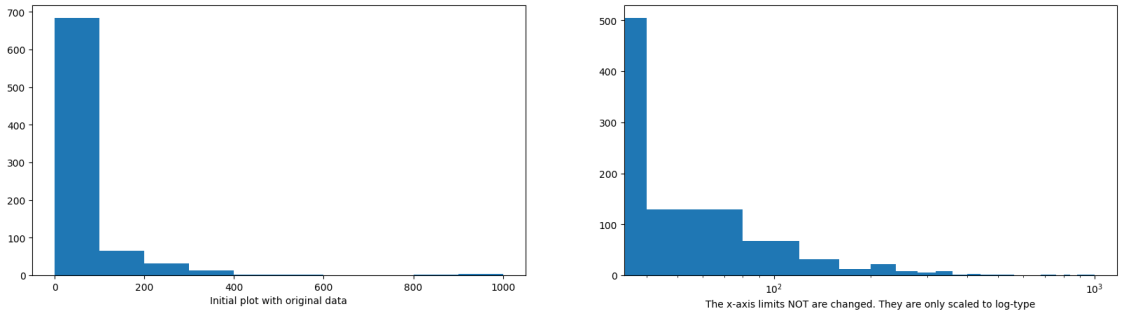
[ ]:
```
# Create a histogram of the 'height' column using the pokemon dataframe
plt.figure(figsize=(20,6))
plt.title('Pokemon Height')
# Set bins at intervals of 0.2
bins = np.arange(0, pokemon['height'].max()+0.2, 0.2)
# Plot the histogram and set xlim to 0 and 6
plt.hist(pokemon['height'],  bins=bins)
plt.xlim(0, 6);
```



1.3.14 Scales and Transformations

Certain data distributions will find themselves amenable to scale transformations. The most common example of this is data that follows an approximately log-normal distribution. This is data that, in their natural units, can look highly skewed: lots of points with low values, with a very long tail of data points with large values. However, after applying a logarithmic transform to the data, the data will follow a normal distribution.

[ ]:
```
plt.figure(figsize = [20, 5])

# HISTOGRAM ON LEFT: full data without scaling
plt.subplot(1, 2, 1)
plt.hist(data=pokemon, x='weight');
# Display a label on the x-axis
plt.xlabel('Initial plot with original data')

plt.subplot(1, 2, 2)

# Get the ticks for bins between [0 - maximum weight]
bins = np.arange(0, pokemon['weight'].max()+40, 40)
plt.hist(data=pokemon, x='weight', bins=bins);

# The argument in the xscale() represents the axis scale type to apply.
# The possible values are: {"linear", "log", "symlog", "logit", ...}
# Refer - https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.xscale.html
plt.xscale('log')
plt.xlabel('The x-axis limits NOT are changed. They are only scaled to log-type');
```

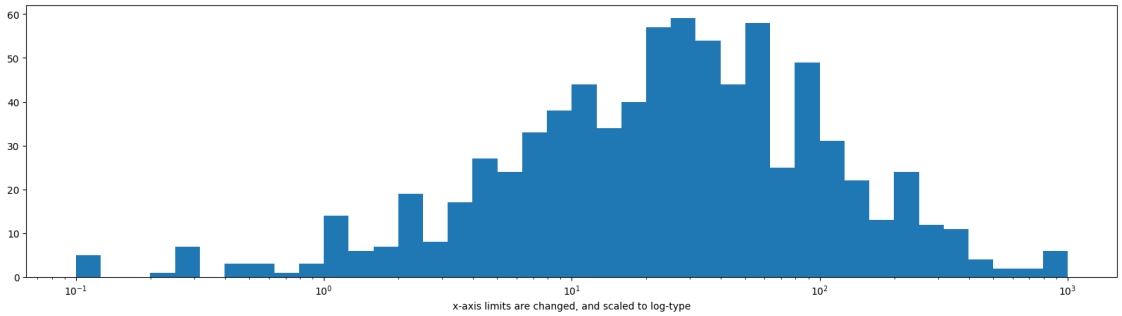


[ ]:
```
# Describe the weight column
pokemon['weight'].describe()
```

[ ]:
```
count    807.000000
mean      61.771128
```

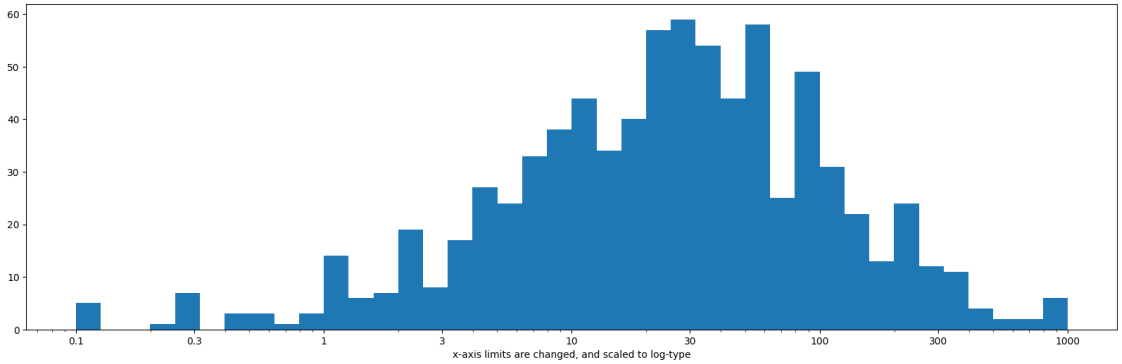
```
std      111.519355
min        0.100000
25%        9.000000
50%       27.000000
75%       63.000000
max      999.900000
Name: weight, dtype: float64
```

[ ]:
```
# Axis transformation
bins = 10 ** np.arange(-1, 3+0.1, 0.1)
plt.figure(figsize = [20, 5])
plt.hist(data=pokemon, x='weight', bins=bins);

# The argument in the xscale() represents the axis scale type to apply.
# The possible values are: {"linear", "log", "symlog", "logit", ...}
plt.xscale('log')

# Apply x-axis label
# Documentatin: [matplotlib `xlabel`](https://matplotlib.org/api/_as_gen/
  ↪matplotlib.pyplot.xlabel.html))
plt.xlabel('x-axis limits are changed, and scaled to log-type');
```



[ ]:
```
# Generate the x-ticks you want to apply on the x-axis in the list ticks
ticks = [0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000]
# Convert the ticks to string values
labels = ['{}'.format(v) for v in ticks]
# Get the bins from [0 - maximum weight] with step size 0.3
bins = 10 ** np.arange(-1, 3+0.1, 0.1)
plt.figure(figsize = [20, 6])
plt.hist(data=pokemon, x='weight', bins=bins)
# Apply x-axis label
plt.xlabel('x-axis limits are changed, and scaled to log-type')
# Apply log scale to x-axis
plt.xscale('log')
# Apply x-axis ticks
```

```
plt.xticks(ticks, labels);
```



1.3.15 Summary

In this lesson, you learned how to create a variety of different types of plots in Python using Matplotlib and Seaborn. - You learned how to create **bar charts**, **pie charts**, **donut plots** and **histograms**. - You also learned how to create figures, axes, and subplots, and how to choose the right plot for your data. Y - ou also learned how to use descriptive statistics, outliers, and axis limits to create better plots. - Finally, you learned about scales and transformations.

Glossary

1. **Univariate visualizations**: Visualize single-variables, such as bar charts, histograms, and line charts.
2. **Bivariate visualizations**: Plots representing the relationship between two variables measured on the given sample data. These plots help to identify the relationship pattern between the two variables.
3. **Ordinal data**: It is a categorical data type where the variables have natural and ordered categories. The distances between the categories are unknown, such as the survey options presented on a five-point scale.

1.3.16 Waffle Plots

A waffle chart is a type of chart that is used to display proportions or fractions of a whole. It is a square grid of squares, where each square represents a certain proportion or fraction of the total. The size of each square is proportional to the percentage of the total that it represents. It is also known as a square pie chart, or a pie chart where the segments are represented as squares in a grid pattern.

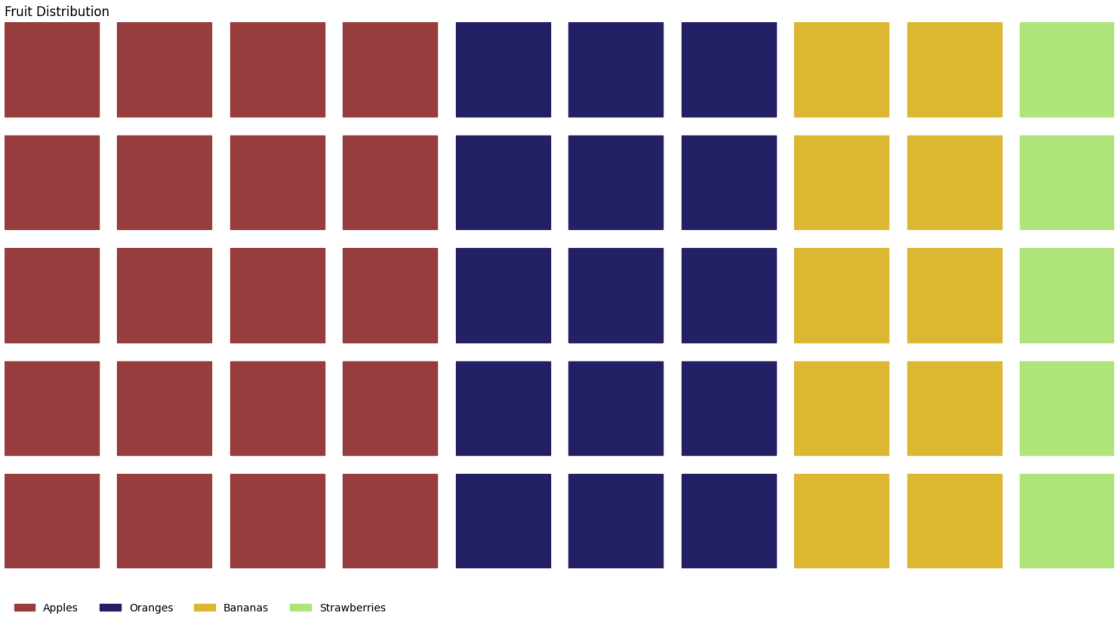
In Seaborn, a library for data visualization in Python, a waffle chart can be created using the `waffle` function. This function is not natively included in seaborn, but it's a common visualization that can be used with seaborn. You can use a library like pywaffle or plotly to create a waffle chart

```
fig = plt.figure(
    FigureClass=Waffle,
    rows=5,
```

```
    values=[48, 21, 15, 7, 7],
    labels=["A", "B", "C", "D", "E"],
    colors=("#232066", "#983D3D", "#DCB732", "#DCB732", "#DCB732"),
    title={'label': 'Waffle Plot', 'loc': 'left'},
    legend={'loc': 'upper left', 'bbox_to_anchor': (1, 1)}
)
```

There's no built-in function for waffle plots in Matplotlib or Seaborn, so we'll need to take some additional steps in order to build one with the tools available.

[ ]:
```
from pywaffle import Waffle

data = {'Apples': 20, 'Oranges': 15, 'Bananas': 10, 'Strawberries': 5}

fig = plt.figure(figsize=(15, 8),
    FigureClass=Waffle,
    rows=5,
    values=data,
    labels=list(data.keys()),
    colors=("#983D3D", "#232066", "#DCB732", "#AFE47B"),
    title={'label': 'Fruit Distribution', 'loc': 'left'},
    legend={'loc': 'lower left', 'bbox_to_anchor': (0, -0.1), 'ncol':␣
 ↪len(data), 'framealpha': 0}
)

plt.show()
```



[ ]:
```
# Generate a dictionary with names of students and their marks (out of 10)
marks = {'John': 19, 'Jane': 10, 'Jack': 7, 'Jill': 12, 'Joe': 6}

# Create a waffle chart using the dictionary
fig = plt.figure( figsize=(15, 8),
    FigureClass=Waffle,
    rows=3,
    values=marks,
    labels=list(marks.keys()),
    colors=("#232066", "#983D3D", "#DCB732", "#BAE47B", "#54206C"),
    title={'label': 'Student Marks', 'loc': 'left'},
    legend={'loc': 'lower left', 'bbox_to_anchor': (0, -0.2), 'ncol':␣
 ↪len(marks), 'framealpha': 0}
)

# Display the chart
plt.show()
```

1.4 LESSON 4: BIVARIATE EXPLORATION OF DATA

1.4.1 Bivariate exploration

Bivariate exploration of data is the process of analyzing the relationship between two variables in a dataset. This type of analysis is used to understand how two variables are related, and to identify patterns and trends in the data. Bivariate exploration can be done using various types of plots and charts such as scatter plots, line plots, and density plots.

For example, if you have a dataset that contains information about the `age` and `income` of individuals, you can use a `scatter plot` to examine the relationship between these two variables. A scatter plot is a type of plot that uses dots to represent individual data points, with the x-axis representing one variable (e.g., age) and the y-axis representing the other variable (e.g., income).

By the end of this lesson, you will be able to create and analyze different types of bivariate visualizations for all possible combinations of qualitative and quantitative variables. You will learn to code the following types of visualizations: - heat maps, - scatterplots, - violin plots, - box plots, - clustered bar charts, - faceting, and - line plots At the end of the lesson, we have introduced a few different visualizations, particularly, swarm, rug, strip, and stacked plots.

1.4.2 Bivariate Plots

- **Scatterplots**: For quantitative vs quantitative relationships
- **Violin plots**: For quantitative vs qualitative relationships
- **Box plots**: For quantitative vs qualitative relationships
- **Clustered bar charts**: For qualitative vs qualitative relationships
- **Heat maps**: For qualitative vs qualitative relationships
- **Line plots**: For quantitative vs quantitative relationships
- **Faceting**: For qualitative vs qualitative relationships

1.4.3 Scatterplots

A scatterplot is a type of plot that uses dots to represent individual data points, with the x-axis representing one variable (e.g., age) and the y-axis representing the other variable (e.g., income). The scatterplot is a useful tool for examining the relationship between two quantitative variables. It is also useful for identifying outliers in the data.

To quantify how strong the correlation is between the variables, we use a correlation coefficient. Pearson correlation coefficient (r) captures linear relationships. It is a value ranging from -1 to +1. A positive value of r indicates the increase in one variable tends to increase another variable. On the other hand, a negative r means the increase in one variable tends to cause a decrease in another variable. A value close to 0 indicates a weak correlation, and a value close to -1 and +1 indicates a strong correlation.

[ ]:
```
# Read the fuel-econ.csv file into a dataframe in data folder
fuel_econ = pd.read_csv('data/fuel-econ.csv')
fuel_econ.head()
```

[ ]:

	id	make	model	year	VClass	\
0	32204	Nissan	GT-R	2013	Subcompact Cars	
1	32205	Volkswagen	CC	2013	Compact Cars	
2	32206	Volkswagen	CC	2013	Compact Cars	
3	32207	Volkswagen	CC 4motion	2013	Compact Cars	
4	32208	Chevrolet	Malibu eAssist	2013	Midsize Cars	

	drive	trans	fuelType	cylinders	displ	\
0	All-Wheel Drive	Automatic (AM6)	Premium Gasoline	6	3.8	
1	Front-Wheel Drive	Automatic (AM-S6)	Premium Gasoline	4	2.0	
2	Front-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6	
3	All-Wheel Drive	Automatic (S6)	Premium Gasoline	6	3.6	
4	Front-Wheel Drive	Automatic (S6)	Regular Gasoline	4	2.4	

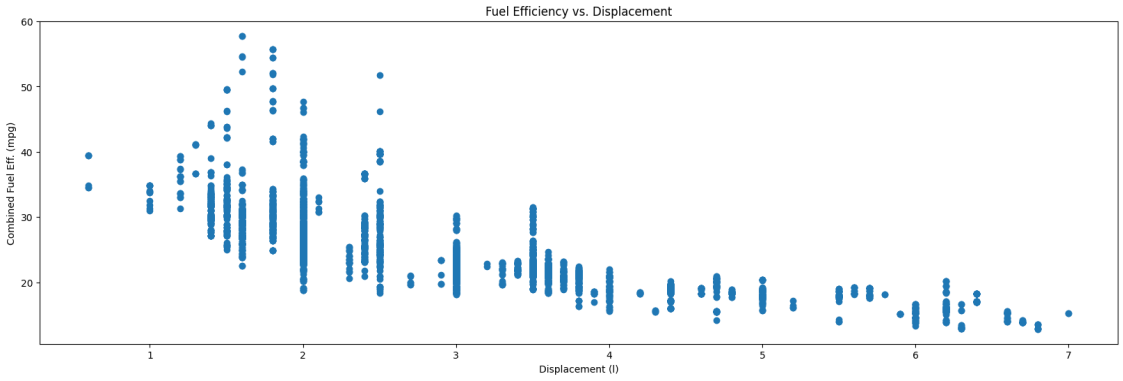
	pv2	pv4	city	UCity	highway	UHighway	comb	co2	feScore	\
0	79	0	16.4596	20.2988	22.5568	30.1798	18.7389	471	4	
1	94	0	21.8706	26.9770	31.0367	42.4936	25.2227	349	6	
2	94	0	17.4935	21.2000	26.5716	35.1000	20.6716	429	5	
3	94	0	16.9415	20.5000	25.2190	33.5000	19.8774	446	5	
4	0	95	24.7726	31.9796	35.5340	51.8816	28.6813	310	8	

```
   ghgScore
0         4
1         6
2         5
3         5
4         8
```

Scatter plot showing a negative correlation between two variables

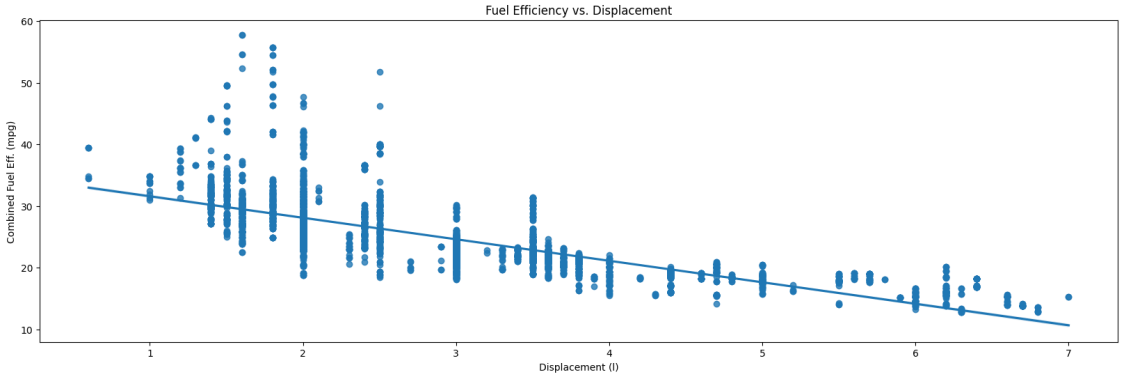
```
# Create a scatter plot of the 'weight' and 'mpg' columns
sns.scatterplot(x='weight', y='mpg', data=auto)
```

[ ]:
```
# Scatter plot of 'displ' and 'comb' using the fuel_econ dataframe
plt.figure(figsize=(20, 6))
plt.scatter(data=fuel_econ, x='displ', y='comb')
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
plt.title('Fuel Efficiency vs. Displacement');
```

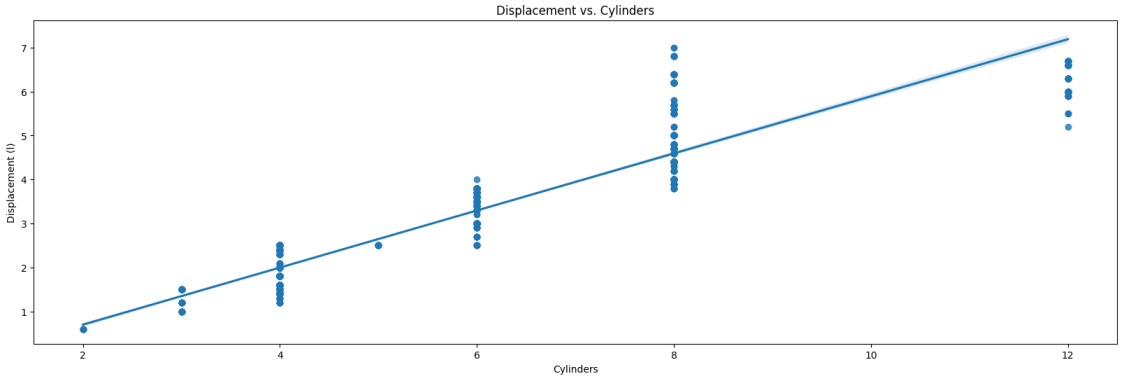


In the example above, the relationship between the two variables is negative because as higher values of the x-axis variable are increasing, the values of the variable plotted on the y-axis are decreasing.

[ ]:
```
# Scatter plot of 'displ' and 'comb' using the fuel_econ dataframe using the␣
↪regplot function
plt.figure(figsize=(20, 6))
sb.regplot(data=fuel_econ, x='displ', y='comb', fit_reg=True)
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)')
plt.title('Fuel Efficiency vs. Displacement');
```



[ ]:
```
# Scatter plot of 'cylinders' and 'disp' using the fuel_econ dataframe using␣
  ↪the regplot function
plt.figure(figsize=(20, 6))
sb.regplot(data=fuel_econ, x='cylinders', y='displ', fit_reg=True)
plt.xlabel('Cylinders')
plt.ylabel('Displacement (l)')
plt.title('Displacement vs. Cylinders');
```



1.4.4 Overplotting, Transparency, and Jitter

Overplotting is a common problem that occurs when there are too many data points on a plot, resulting in a cluttered and difficult-to-interpret visual. When data points overlap, they can obscure one another, making it difficult to see the underlying patterns in the data. This can happen in scatter plots, line plots, and other types of plots where data points are represented as markers or dots.

To address this problem, **transparency** and **jitter** are two techniques that can be used to make the data points more distinguishable.

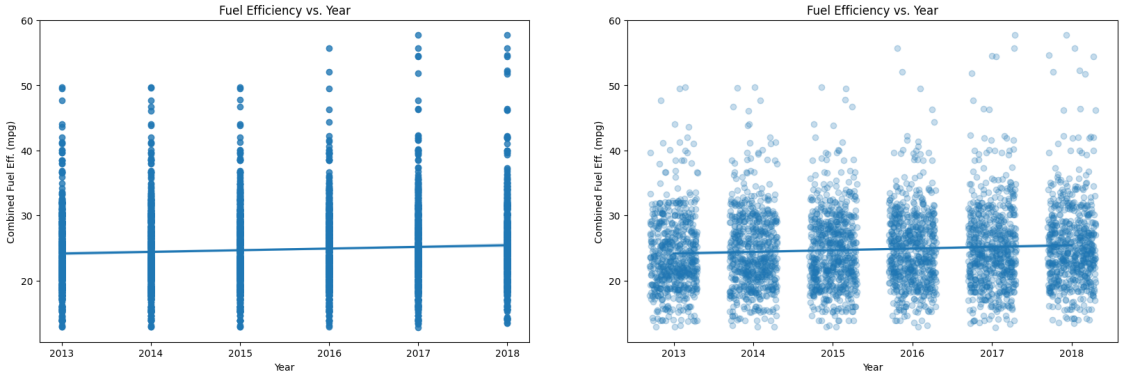
Transparency is a technique that makes markers or dots in a plot partially transparent, allowing data points that are overlaid to be seen more clearly. This is done by adjusting the alpha parameter

of the markers or dots. A lower alpha value will make the markers more transparent, allowing the data points underneath to be seen more clearly.

Jitter is a technique that adds random noise to the x and y coordinates of the data points. This causes the data points to be spread out, making them less likely to overlap and more distinguishable. Jitter can be added to a plot by applying a small random offset to the x and y coordinates of the data points.

The graph below shows the effect of transparency and jitter on a scatter plot. The left plot shows the original scatter plot, with no transparency or jitter. The middle plot shows the same scatter plot with transparency applied. The right plot shows the same scatter plot with transparency and jitter applied.

[ ]:
```
# Two subplot axes with left showing regplot of 'year' and 'comb' and right␣
 ↪showing regplot of 'year' and 'comb' with x-jitter of 0.3 and transparency␣
 ↪of 0.25
plt.figure(figsize=(20, 6))
plt.subplot(1, 2, 1)
sb.regplot(data=fuel_econ, x='year', y='comb', fit_reg=True)
plt.xlabel('Year')
plt.ylabel('Combined Fuel Eff. (mpg)')
plt.title('Fuel Efficiency vs. Year');

plt.subplot(1, 2, 2)
sb.regplot(data=fuel_econ, x='year', y='comb', fit_reg=True, x_jitter=0.3,␣
 ↪scatter_kws={'alpha': 0.25})
plt.xlabel('Year')
plt.ylabel('Combined Fuel Eff. (mpg)')
plt.title('Fuel Efficiency vs. Year');
```



1.4.5 Heat Maps

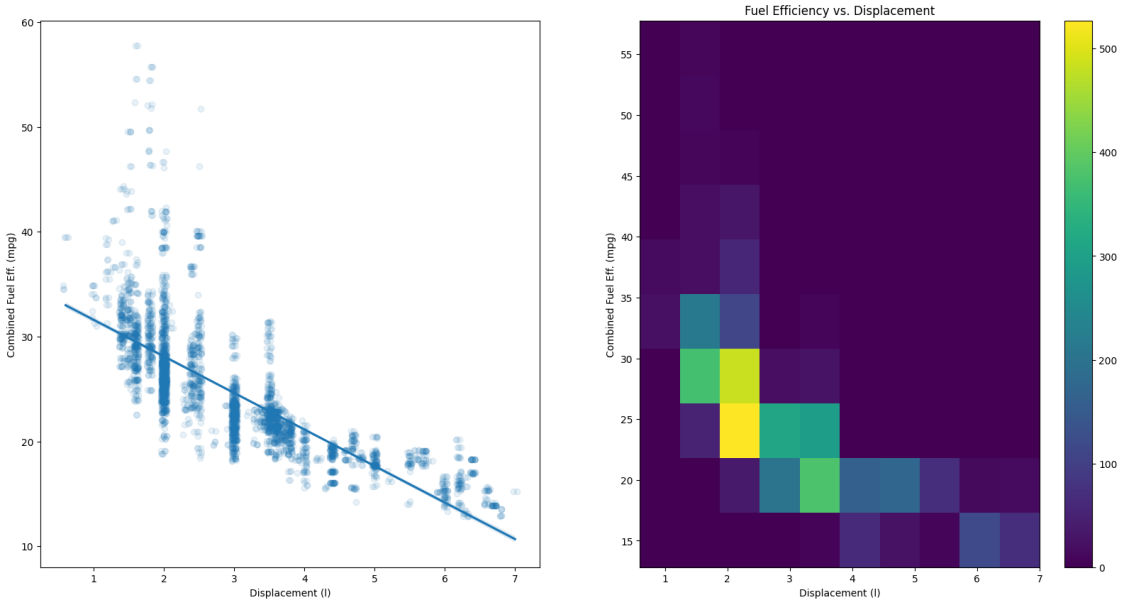
A **heat map** is a 2-d version of the histogram that can be used as an alternative to a scatterplot. Like a scatterplot, the values of the two numeric variables to be plotted are placed on the plot axes. Similar to a histogram, the plotting area is divided into a grid and the number of points in each

grid rectangle is added up. Since there won't be room for bar heights, counts are indicated instead by grid cell color. A heat map can be implemented with Matplotlib's `hist2d()` function.

Heat maps are useful in the following cases:

1. To represent a plot for discrete vs. another discrete variable
2. As an alternative to transparency when the data points are enormous

```
[ ]: # Create subplots of 2 columns
plt.figure(figsize=(20, 10))
plt.subplot(1, 2, 1)
sb.regplot(data=fuel_econ, x='displ', y='comb', fit_reg=True,  x_jitter=0.04,␣
 ↪scatter_kws={'alpha':1/10})
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)');

plt.subplot(1, 2, 2)
# Plot heatmap using the hist2d function
plt.hist2d(data=fuel_econ, x='displ', y='comb')
plt.colorbar()
plt.xlabel('Displacement (l)')
plt.ylabel('Combined Fuel Eff. (mpg)');
plt.title('Fuel Efficiency vs. Displacement');
```



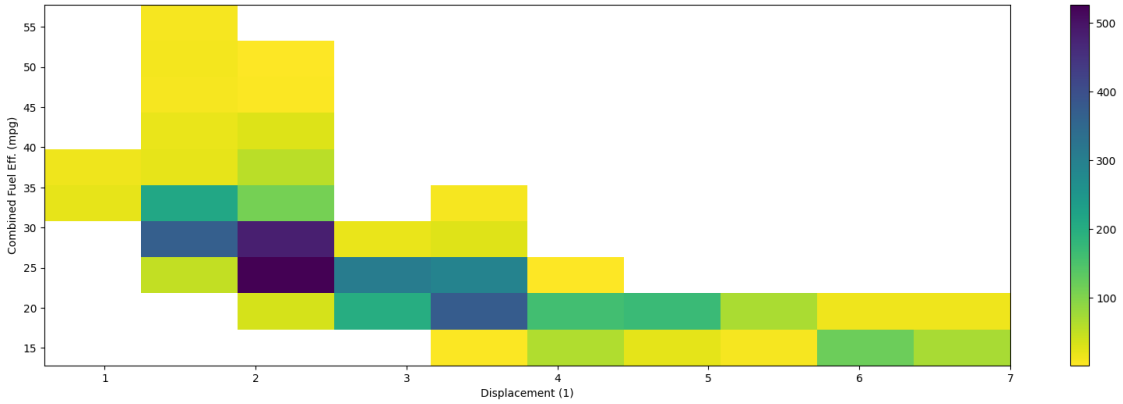
In the example above, we added a `colorbar()` function call to add a colorbar to the side of the plot, showing the mapping from counts to colors. To select a different color palette, you can set the "cmap" parameter in `hist2d`. For example, to use the "Blues" color palette, you can use the following code:

```
plt.hist2d(x, y, bins=20, cmap='Blues')
```

To distinguish cells with zero counts from those with non-zero counts, we can use cmin to set the minimum count value that will be mapped to a color. For example, to only show cells with at least one point by assigning `cmin=0.5` or higher, we can use the following code:

```
plt.hist2d(x, y, bins=20, cmap='Blues', cmin=0.5)
```

[ ]:
```
plt.figure(figsize=(20, 6))
# Use cmap to reverse the color map.
plt.hist2d(data = fuel_econ, x = 'displ', y = 'comb', cmin=0.5,␣
 ↪cmap='viridis_r')
plt.colorbar()
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');
```

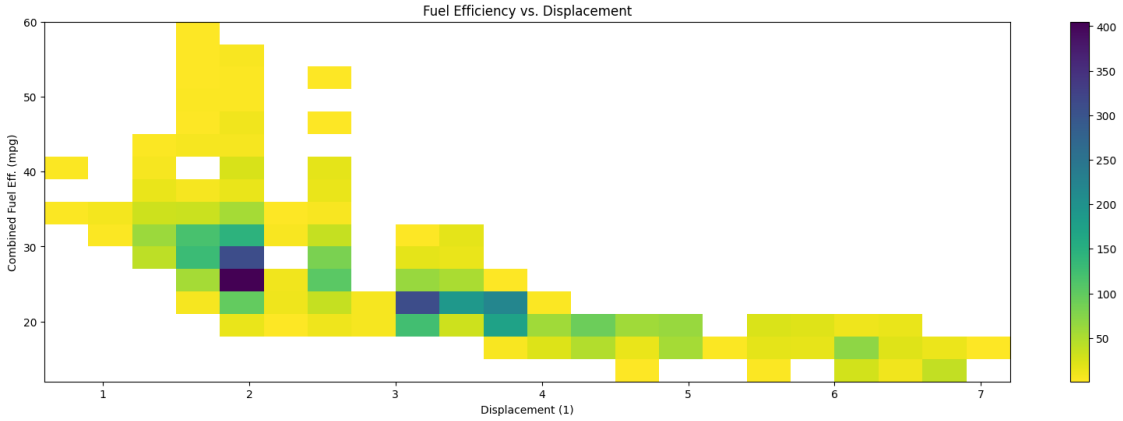


Adding Bins to Heat Maps To add bins to a heat map, we can use the `plt.hist2d()` function. The `plt.hist2d()` function takes in the following parameters:

- `x`: The x-axis data
- `y`: The y-axis data
- `bins`: The number of bins to use for the x and y axes

[ ]:
```
plt.figure(figsize=(20, 6))

# Create bins for x-axis and y-axis using np.arange
bins_x = np.arange(0.6, 7+0.3, 0.3)
bins_y = np.arange(12, 58+3, 3)

# Plot heatmap using the hist2d function
plt.hist2d(data=fuel_econ, x='displ', y='comb', bins=[bins_x, bins_y],␣
 ↪cmap='viridis_r', cmin=0.5)
plt.colorbar()
plt.xlabel('Displacement (1)')
```

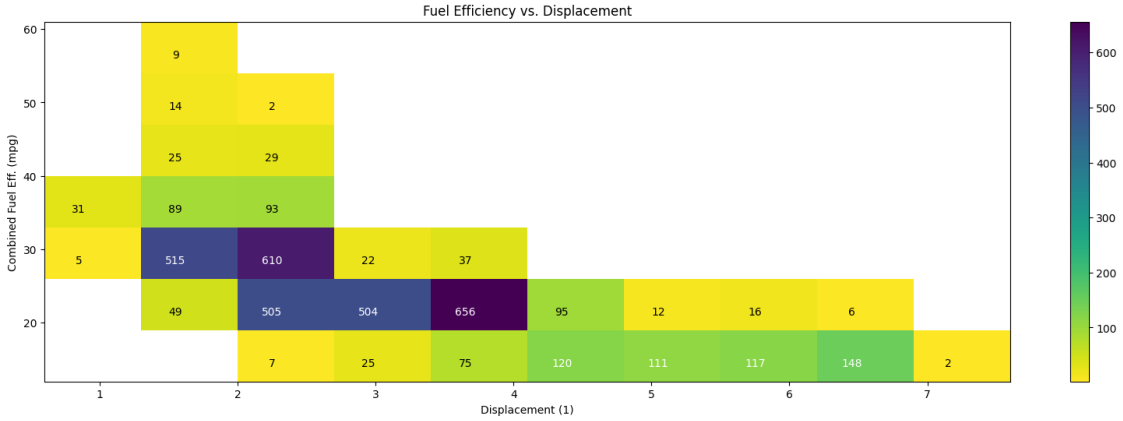
```
plt.ylabel('Combined Fuel Eff. (mpg)');
plt.title('Fuel Efficiency vs. Displacement');
```



Annotations on each cell If you have a lot of data, you might want to add annotations to cells in the plot indicating the count of points in each cell. From hist2d, this requires the addition of text elements one by one, much like how text annotations were added one by one to the bar plots in the previous lesson. We can get the counts to annotate directly from what is returned by hist2d, which includes not just the plotting object, but an array of counts and two vectors of bin edges.

[ ]:
```
plt.figure(figsize=(20, 6))

# Create bins for x-axis and y-axis using np.arange
bins_x = np.arange(0.6, 7+0.7, 0.7)
bins_y = np.arange(12, 58+7, 7)

# Plot heatmap using the hist2d function
h2d = plt.hist2d(data=fuel_econ, x='displ', y='comb', bins=[bins_x, bins_y],␣
 ↪cmap='viridis_r', cmin=0.5)
plt.colorbar()
plt.xlabel('Displacement (1)')
plt.ylabel('Combined Fuel Eff. (mpg)');
plt.title('Fuel Efficiency vs. Displacement');

# Add text annotations to each cell
counts = h2d[0]
for i in range(counts.shape[0]):
    for j in range(counts.shape[1]):
        c = counts[i,j]
        if c >= 100: # increase visibility on darker cells
            plt.text(bins_x[i]+0.25, bins_y[j]+2.5, int(c),
                     ha = 'center', va = 'center', color = 'white')
        elif c > 0:
```

```
            plt.text(bins_x[i]+0.25, bins_y[j]+2.5, int(c),
                     ha = 'center', va = 'center', color = 'black')
```

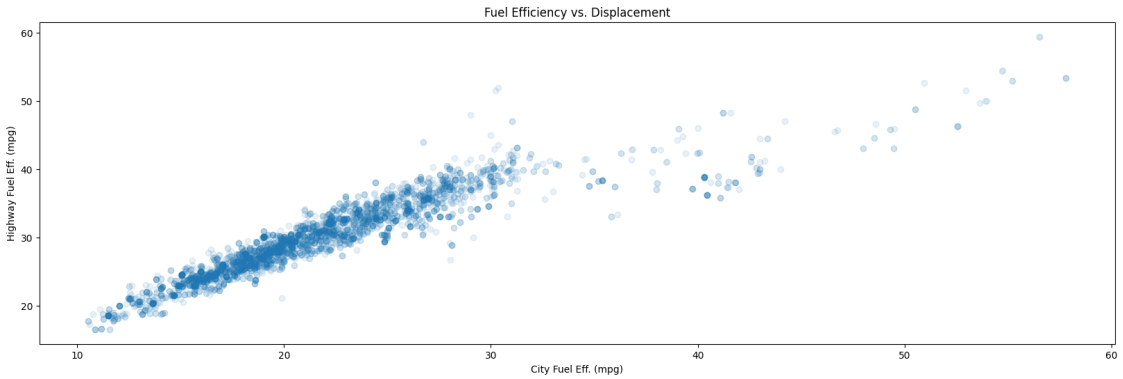


If you have too many cells in your heat map, then the annotations will end up being too overwhelming, too much to attend to. In cases like that, it's best to leave off the annotations and let the data and colorbar speak for themselves.

Let's look at the relationship between fuel mileage ratings for city vs. highway driving, as stored in the `city` and `highway` variables (in miles per gallon, or mpg). Use a scatter plot to depict the data.

1. What is the general relationship between these variables?
2. Are there any points that appear unusual against these trends?

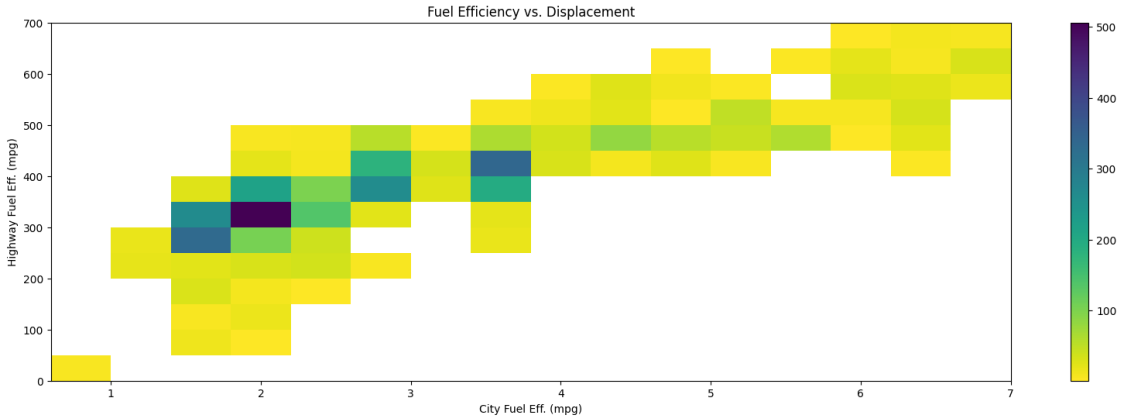
[ ]:
```
# Plot scatter plot of 'city' and 'highway' using the fuel_econ dataframe
plt.figure(figsize=(20, 6))
plt.scatter(data=fuel_econ, x='city', y='highway', alpha=1/10)
plt.xlabel('City Fuel Eff. (mpg)')
plt.ylabel('Highway Fuel Eff. (mpg)')
plt.title('Fuel Efficiency vs. Displacement');
```



Most of the data falls in a large blob between 10 and 30 mpg city and 20 to 40 mpg highway. Some transparency is added via 'alpha' to show the concentration of data. Interestingly, for most cars highway mileage is clearly higher than city mileage, but for those cars with city mileage above about 30 mpg, the distinction is less pronounced. In fact, most cars above 45 mpg city have better city mileage than highway mileage, contrary to the main trend. It might be good to call out this trend by adding a diagonal line to the figure using the ***plot*** *function.*

How does the engine size relate to a car's CO2 footprint? The 'displ' variable has the former (in liters), while the 'co2' variable has the latter (in grams per mile). Use a heat map to depict the data. How strong is this trend?

[ ]:
```
# Plot heatmap using the hist2d function with bins_x and bins_y. Use the␣
 ↪`displ` column for x-axis and `c02` column for y-axis
plt.figure(figsize=(20, 6))
bins_x = np.arange(0.6, fuel_econ['displ'].max()+.4, .4)
bins_y = np.arange(0, fuel_econ['co2'].max()+50, 50)

plt.hist2d(data=fuel_econ, x='displ', y='co2', bins=[bins_x, bins_y],␣
 ↪cmap='viridis_r', cmin=0.5)
plt.colorbar()
plt.xlabel('City Fuel Eff. (mpg)')
plt.ylabel('Highway Fuel Eff. (mpg)')
plt.title('Fuel Efficiency vs. Displacement');
```



In the heat map, I've set up a color map that goes from light to dark, and made it so that any cells without count don't get colored in. The visualization shows that most cars fall in a line where larger engine sizes correlate with higher emissions. The trend is somewhat broken by those cars with the lowest emissions, which still have engine sizes shared by most cars (between 1 and 3 liters).

1.4.6 Violin Plots

There are a few ways of plotting the relationship between one quantitative and one qualitative variable, that demonstrate the data at different levels of abstraction. The violin plot is on the lower level of abstraction. For each level of the categorical variable, a distribution of the values on

the numeric variable is plotted. The distribution is plotted as a kernel density estimate, something like a smoothed histogram. There is an extra section at the end of the previous lesson that provides more insight into kernel density estimates.

Seaborn's `violinplot()` function can be used to create violin plots. The `violinplot()` function takes in the following parameters:

- `x`: The x-axis data
- `y`: The y-axis data
- `data`: The data frame containing the data
- `inner`: The representation of the data inside the violin plot. Can be set to `None` to remove the bars inside the violin plot.
- `color`: The color of the violin plot
- `scale`: The scale of the violin plot. Can be set to `area` to scale the width of the violin plot by the number of observations in that bin. The syntax for creating a violin plot is as follows:

```
sns.violinplot(x, y, data=df, inner=None, color='lightgray')
```

[ ]:
```
df = pd.DataFrame({
    'Group': ['A', 'A', 'A', 'B', 'B', 'B', 'C', 'C', 'C'],
    'Value': [2, 5, 7, 3, 8, 10, 1, 6, 9]
})
df
```

[ ]:

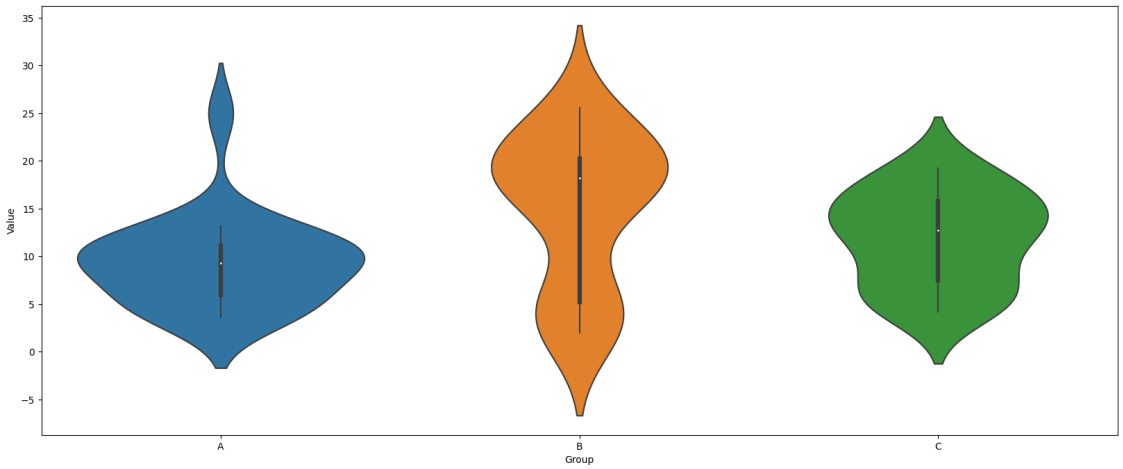
	Group	Value
0	A	2
1	A	5
2	A	7
3	B	3
4	B	8
5	B	10
6	C	1
7	C	6
8	C	9

This DataFrame has two columns: `Group` and `Value`. `Group` contains three unique groups (A, B, and C), and `Value` contains numerical values.

To demonstrate the `violinplot()` function in seaborn, we can create a violin plot of the `Value` column grouped by the `Group` column. A violin plot is a useful way to visualize the distribution of a numeric variable for different categories.

[ ]:
```
import seaborn as sns
plt.figure(figsize=(20, 8))
sns.violinplot(x='Group', y='Value', data=df)
```

[ ]:
```
<AxesSubplot: xlabel='Group', ylabel='Value'>
```

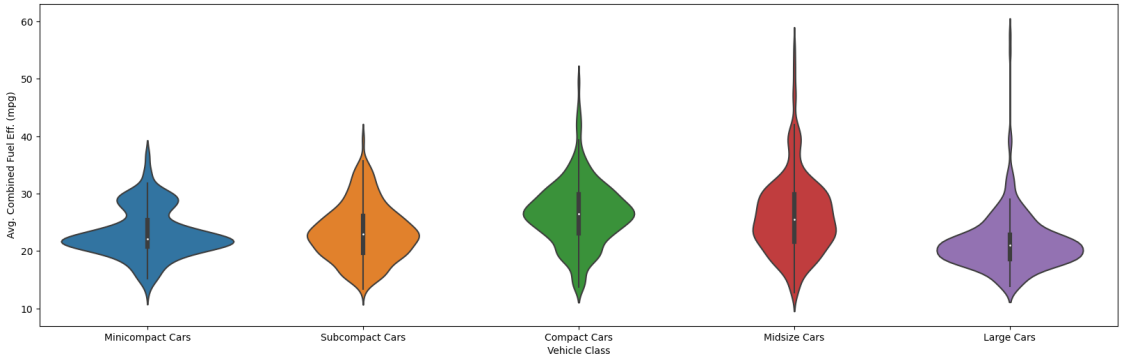


This code will produce a violin plot of the `Value` column grouped by the `Group` column. The `x` parameter specifies the column to group by (in this case, `Group`), and the `y` parameter specifies the numerical column to plot (in this case, `Value`).

The resulting plot will show a kernel density estimation of the distribution of each group, with the width of the violin indicating the density of values. The plot will also show the quartiles of each distribution as horizontal lines, and individual data points as dots.

Violin plot for plotting a Quantitative variable (fuel efficiency) versus Qualitative variable (vehicle class)

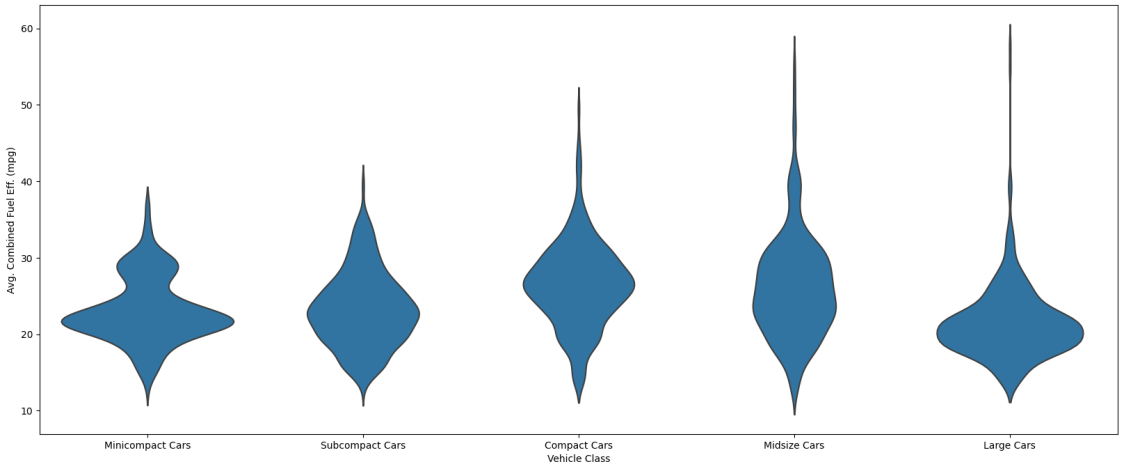
[ ]:
```
# Types of sedan cars
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars',␣
 ↪'Midsize Cars', 'Large Cars']

# Returns the types for sedan_classes with the categories and orderedness
# Refer - https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.
 ↪api.types.CategoricalDtype.html
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)

# Use pandas.astype() to convert the "VClass" column from a plain object type␣
 ↪into an ordered categorical type
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);
plt.figure(figsize=(20, 6))
sb.violinplot(data=fuel_econ, x='VClass', y='comb')
plt.xlabel('Vehicle Class')
plt.ylabel('Avg. Combined Fuel Eff. (mpg)');
```



You can see that each level has been rendered in a different color, like how the plain `countplot()` was in the previous lesson. We can set the "color" parameter to make each curve the same color if it is not meaningful.

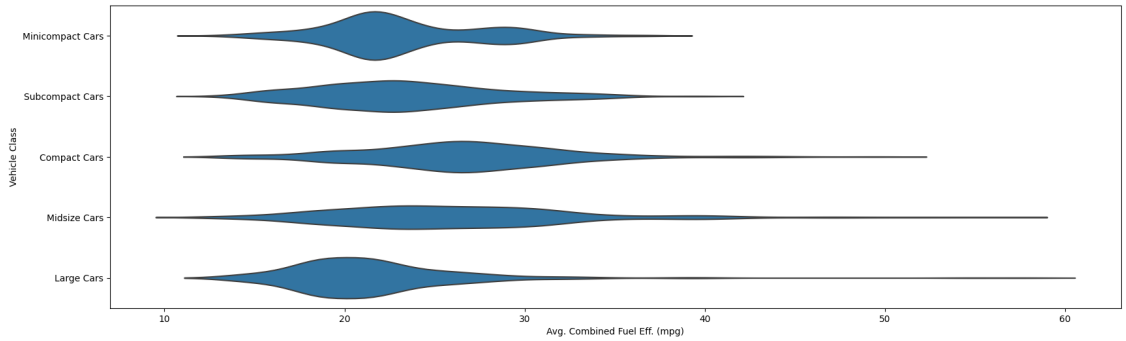
Inside each curve, there is a black shape with a white dot inside, a miniature box plot. A further discussion of box plots will be performed on the next page. If you'd like to remove the box plot, you can set the `inner = None` parameter in the `violinplot` call to simplify the look of the final visualization.

[ ]:
```
base_color = sb.color_palette()[0]

# The "inner" argument represents the datapoints in the violin interior.
# It can take any value from {"box", "quartile", "point", "stick", None}
# If "box", it draws a miniature boxplot.
plt.figure(figsize=(20, 8))
sb.violinplot(data=fuel_econ, x='VClass', y='comb', color=base_color,␣
 ↪inner=None)
plt.xlabel('Vehicle Class')
plt.ylabel('Avg. Combined Fuel Eff. (mpg)');
```



Much like how the bar chart could be rendered with horizontal bars, the violin plot can also be rendered horizontally. Seaborn is smart enough to make an appropriate inference on which orientation is requested, depending on whether "x" or "y" receives the categorical variable. But if both variables are numeric (e.g., one is discretely-valued) then the "orient" parameter can be used to specify the plot orientation.

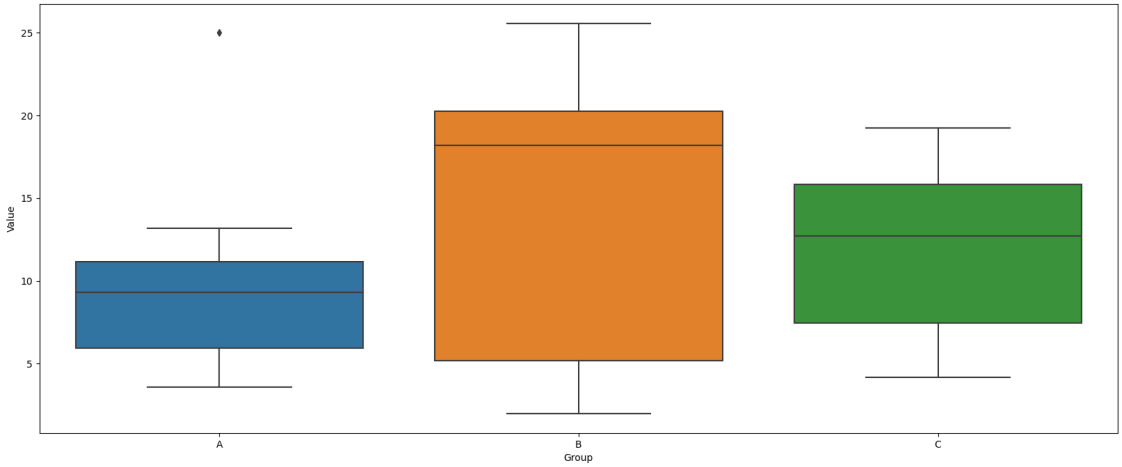
[ ]:
```
plt.figure(figsize=(20, 6))
sb.violinplot(data=fuel_econ, y='VClass', x='comb', color=base_color,␣
 ↪inner=None)
plt.ylabel('Vehicle Class')
plt.xlabel('Avg. Combined Fuel Eff. (mpg)');
```



1.4.7 Box Plots

Box plots are a useful way to visualize the distribution of a numeric variable for different categories. They are similar to violin plots, but they show only summary statistics: the minimum, first quartile, median, third quartile, and maximum. The box plot is on the lower level of abstraction. For each level of the categorical variable, a distribution of the values on the numeric variable is plotted. The distribution is plotted as a kernel density estimate, something like a smoothed histogram. There is an extra section at the previous lesson that provides more insight into kernel density estimates.

[ ]:
```
import seaborn as sns
plt.figure(figsize=(20, 8))
sns.boxplot(x='Group', y='Value', data=df)
plt.xlabel('Group')
plt.ylabel('Value');
```



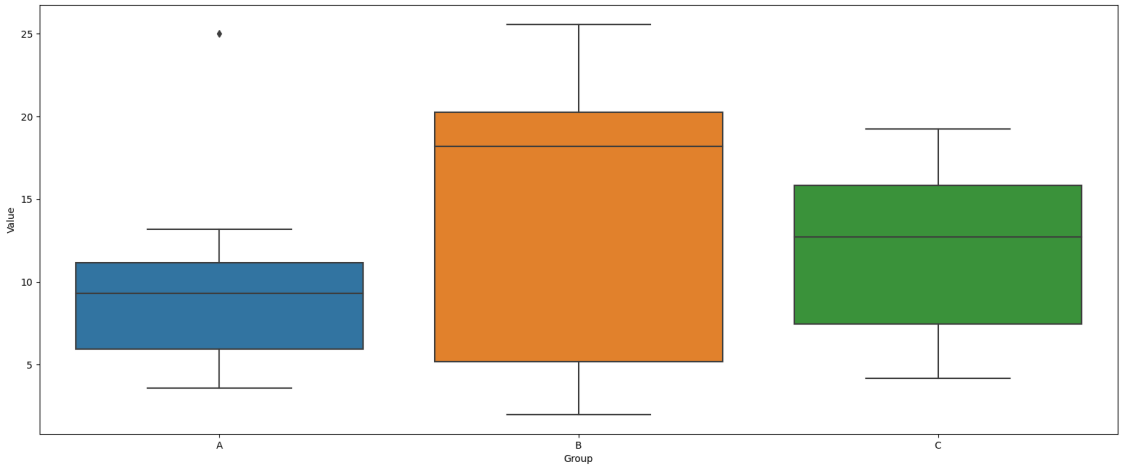
This code will produce a box plot of the `Value` column grouped by the `Group` column. The `x` parameter specifies the column to group by (in this case, `Group`), and the `y` parameter specifies the numerical column to plot (in this case, `Value`).

The resulting plot will show the distribution of each group, including the **median** (indicated by the horizontal line within each box), the **interquartile range** (IQR, represented by the height of each box), and the **range of the data** (represented by the "whiskers" extending from each box). Any points that lie outside the whiskers are considered **"outliers"** and are plotted individually as dots.

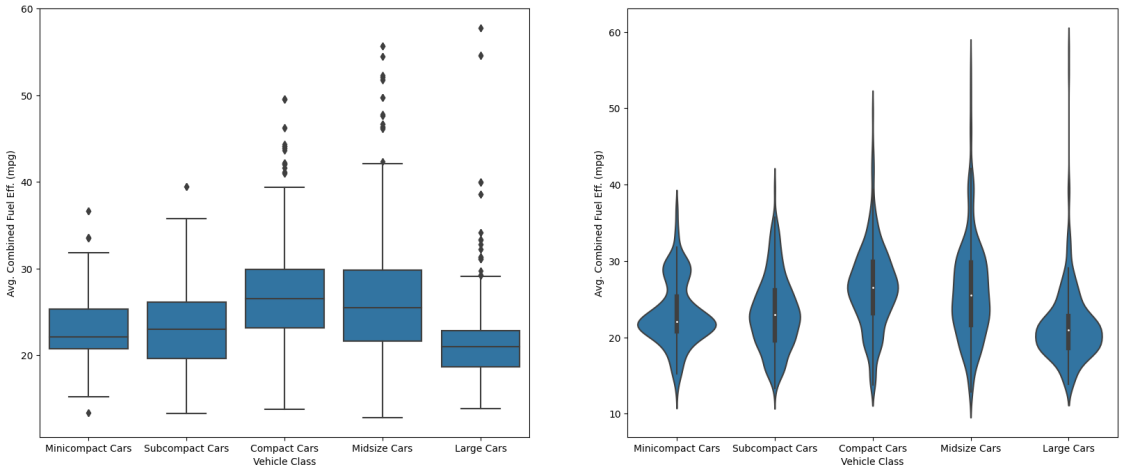
The box plot is a useful tool for comparing the distribution of numerical data across different categories. It provides a quick way to compare the medians, IQRs, and ranges of multiple groups at once.

[ ]:
```
np.random.seed(42)

df = pd.DataFrame({
    'Group': np.repeat(['A', 'B', 'C'], 20),
    'Value': np.concatenate([
        np.random.normal(10, 2, 15),
        np.random.normal(5, 1, 5),
        np.random.normal(20, 3, 15),
        np.random.normal(5, 1, 5),
        np.random.normal(15, 4, 15),
        np.random.normal(5, 1, 5)
    ])
})

# Add outliers
df.loc[5, 'Value'] = 25
df.loc[32, 'Value'] = 2
df.loc[40, 'Value'] = 10
```

[ ]:
```
plt.figure(figsize=(20, 8))
sns.boxplot(x='Group', y='Value', data=df)
plt.xlabel('Group')
plt.ylabel('Value');
```



[ ]:
```
# Using fuel_econ, y='VClass', x='comb', color=base_color, inner=None, and the␣
↪default order, generate a box plot and violin plot side-by-side
plt.figure(figsize=(20, 8))
plt.subplot(1, 2, 1)
sb.boxplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.ylabel('Avg. Combined Fuel Eff. (mpg)')
plt.xlabel('Vehicle Class')

plt.subplot(1, 2, 2)
sb.violinplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.ylabel('Avg. Combined Fuel Eff. (mpg)')
plt.xlabel('Vehicle Class');
```

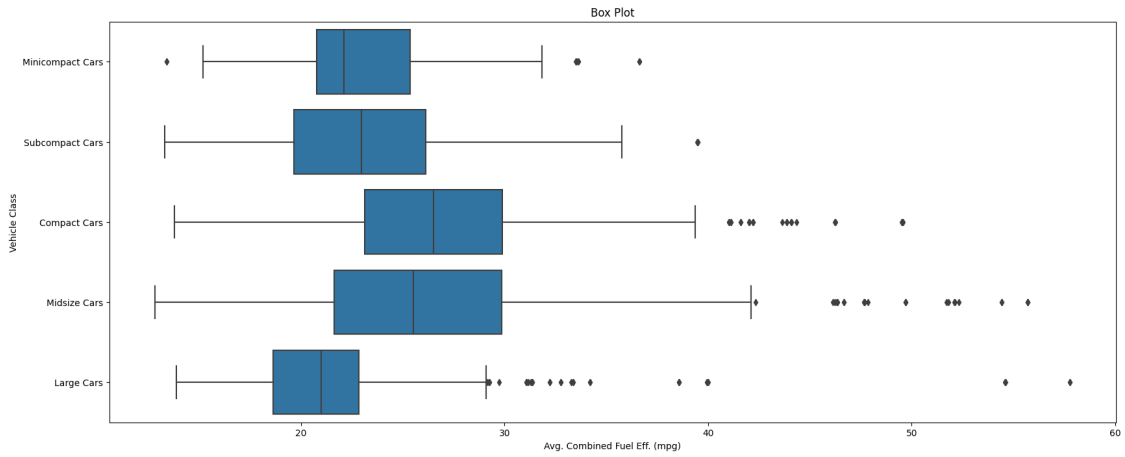


The inner boxes and lines in the violin plot match up with the boxes and whiskers in the box plot. Comparing the two plots, the box plot is a cleaner summary of the data than the violin plot. The box plot is also more compact, which makes it easier to compare distributions across multiple groups.

As with violinplot, boxplot can also render horizontal box plots by setting the numeric and categorical features to the appropriate arguments.

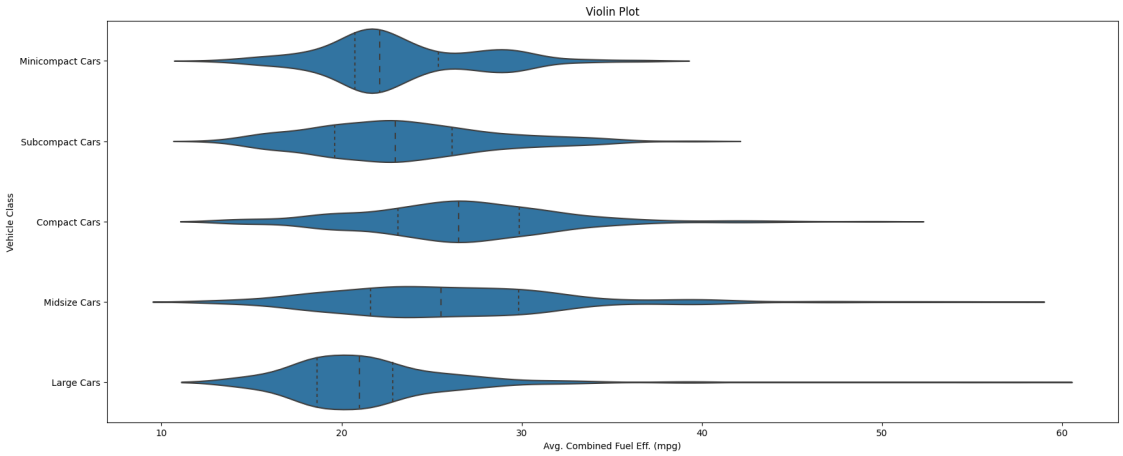
[ ]:
```
plt.figure(figsize=(20, 8))
sb.boxplot(data=fuel_econ, y='VClass', x='comb', color=base_color)
plt.xlabel('Avg. Combined Fuel Eff. (mpg)')
plt.ylabel('Vehicle Class')
plt.title('Box Plot')
```

[ ]:
```
Text(0.5, 1.0, 'Box Plot')
```



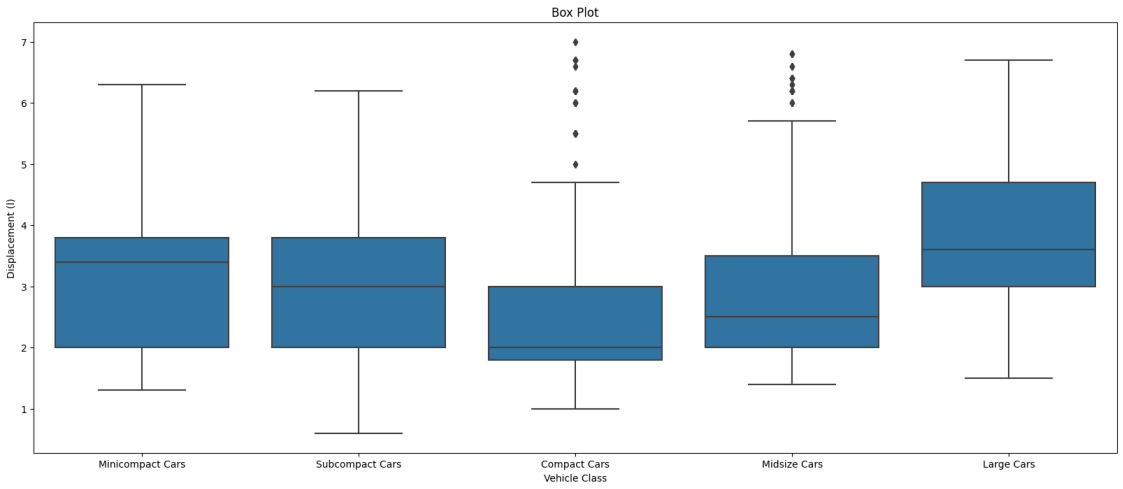
In violinplot, there is an additional option for plotting summary statistics in the violin, beyond the default mini box plot. By setting `inner = 'quartile'`, three lines will be plotted within each violin area for the three middle quartiles. The line with thick dashes indicates the median, and the two lines with shorter dashes on either side the first and third quartiles.

[ ]:
```
plt.figure(figsize=(20, 8))
sb.violinplot(data=fuel_econ, y='VClass', x='comb', color=base_color,␣
 ↪inner='quartile')
plt.xlabel('Avg. Combined Fuel Eff. (mpg)')
plt.ylabel('Vehicle Class')
plt.title('Violin Plot');
```



What is the relationship between the size of a car and the size of its engine? The vehicle classes can be found in the VClass column, while the engine sizes are in the displ column (in liters).

[ ]:
```
# What is the relationship between the size of a car and the size of its engine?
↪ The vehicle classes can be found in the VClass column, while the engine␣
↪sizes are in the displ column (in liters).
plt.figure(figsize=(20, 8))
sb.boxplot(data=fuel_econ, x='VClass', y='displ', color=base_color)
plt.xlabel('Vehicle Class')
plt.ylabel('Displacement (l)')
plt.title('Box Plot');
```



1.4.8 Clustered Bar Charts

Clustered bar charts are a useful way to visualize the distribution of a numeric variable for different categories. They are similar to stacked bar charts, but they show the distribution of the numeric variable for each category, rather than the total value of the numeric variable for each category. The clustered bar chart is on the lower level of abstraction. For each level of the categorical variable, a distribution of the values on the numeric variable is plotted. The distribution is plotted as a kernel density estimate, something like a smoothed histogram. There is an extra section at the previous lesson that provides more insight into kernel density estimates.

[ ]:
```
import pandas as pd

df = pd.DataFrame({
    'Group': ['A', 'A', 'B', 'B', 'C', 'C'],
    'Category': ['X', 'Y', 'X', 'Y', 'X', 'Y'],
    'Value': [10, 12, 15, 13, 8, 10]
})

df
```

[ ]:

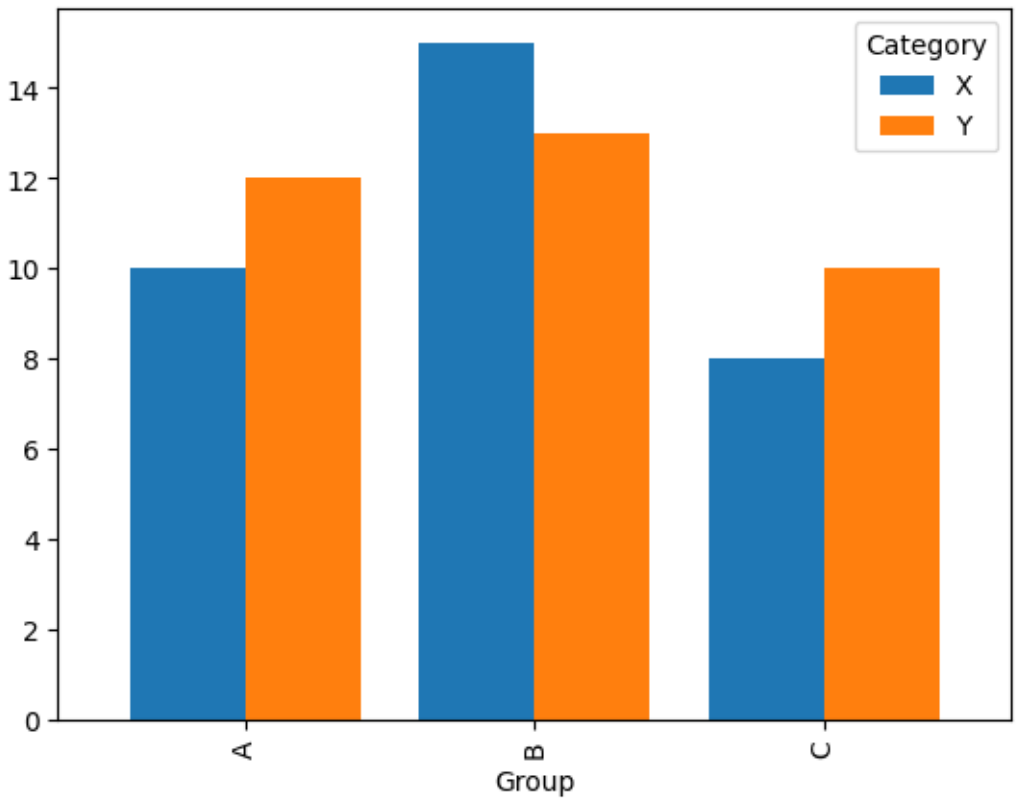
	Group	Category	Value
0	A	X	10
1	A	Y	12
2	B	X	15
3	B	Y	13
4	C	X	8
5	C	Y	10

This DataFrame has three columns: `Group`, `Category`, and `Value`. `Group` contains three unique groups (A, B, and C), `Category` contains two unique categories (X and Y), and `Value` contains numerical values.

To create a clustered bar chart in pandas, we can use the pivot function to reshape the data and then plot the resulting DataFrame:

[ ]:
```
plt.figure(figsize=(20, 8))
pivot_df = df.pivot(index='Group', columns='Category', values='Value')
pivot_df.plot(kind='bar', width=0.8);
```

```
<Figure size 2000x800 with 0 Axes>
```



[ ]:
```
# Types of sedan cars
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars',␣
 ↪'Midsize Cars', 'Large Cars']

# Returns the types for sedan_classes with the categories and orderedness
# Refer - https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.
 ↪api.types.CategoricalDtype.html
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)

# Use pandas.astype() to convert the "VClass" column from a plain object type␣
 ↪into an ordered categorical type
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);
```

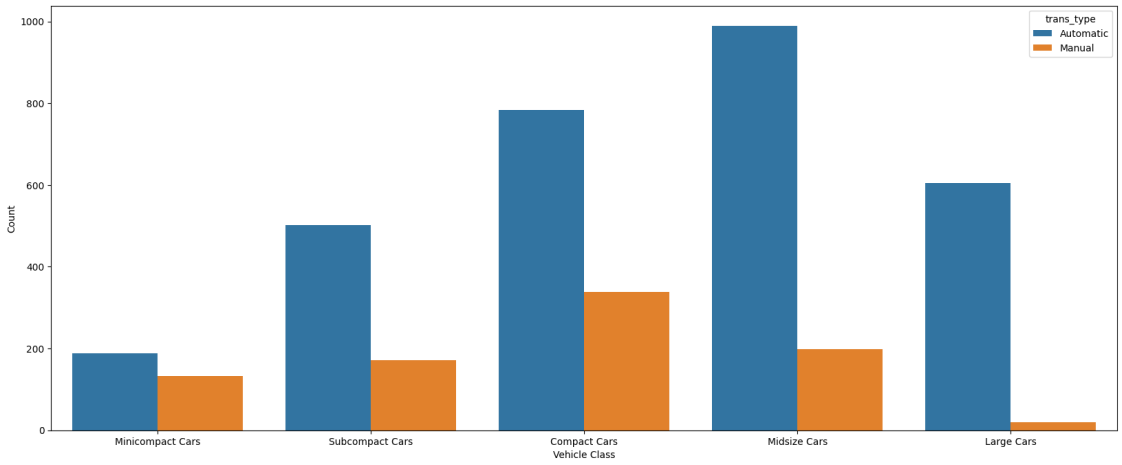
[ ]:
```
# The existing `trans` column has multiple sub-types of Automatic and Manual.
# But, we need plain two types, either Automatic or Manual. Therefore, add a␣
 ↪new column.

# The Series.apply() method invokes the `lambda` function on each value of␣
 ↪`trans` column.
# In python, a `lambda` function is an anonymous function that can have only␣
 ↪one expression.
```

```
fuel_econ['trans_type'] = fuel_econ['trans'].apply(lambda x:x.split()[0])
fuel_econ.head()
```

[ ]:
```
      id        make           model  year           VClass  \
0  32204      Nissan            GT-R  2013  Subcompact Cars
1  32205  Volkswagen              CC  2013     Compact Cars
2  32206  Volkswagen              CC  2013     Compact Cars
3  32207  Volkswagen      CC 4motion  2013     Compact Cars
4  32208   Chevrolet  Malibu eAssist  2013     Midsize Cars

                drive              trans          fuelType  cylinders  displ  \
0    All-Wheel Drive    Automatic (AM6)  Premium Gasoline          6    3.8
1  Front-Wheel Drive  Automatic (AM-S6)  Premium Gasoline          4    2.0
2  Front-Wheel Drive     Automatic (S6)  Premium Gasoline          6    3.6
3    All-Wheel Drive     Automatic (S6)  Premium Gasoline          6    3.6
4  Front-Wheel Drive     Automatic (S6)  Regular Gasoline          4    2.4

   …  pv4     city    UCity  highway  UHighway     comb  co2  feScore  \
0  …    0  16.4596  20.2988  22.5568   30.1798  18.7389  471        4
1  …    0  21.8706  26.9770  31.0367   42.4936  25.2227  349        6
2  …    0  17.4935  21.2000  26.5716   35.1000  20.6716  429        5
3  …    0  16.9415  20.5000  25.2190   33.5000  19.8774  446        5
4  …   95  24.7726  31.9796  35.5340   51.8816  28.6813  310        8

   ghgScore  trans_type
0         4   Automatic
1         6   Automatic
2         5   Automatic
3         5   Automatic
4         8   Automatic

[5 rows x 21 columns]
```

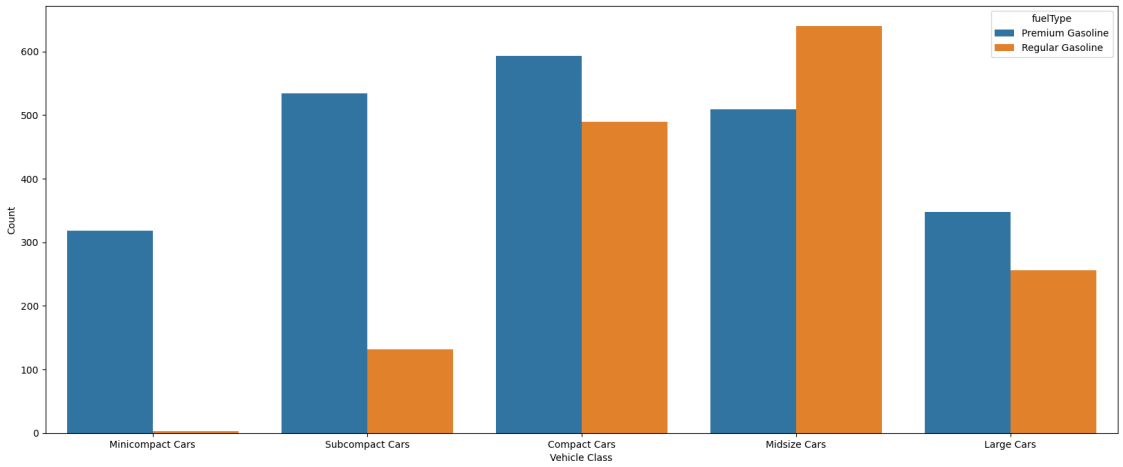
[ ]:
```
plt.figure(figsize=(20, 8))
sb.countplot(data = fuel_econ, x = 'VClass', hue = 'trans_type')
plt.xlabel('Vehicle Class')
plt.ylabel('Count')
```

[ ]:
```
Text(0, 0.5, 'Count')
```



Use a plot to explore whether or not there are differences in recommended fuel type depending on the vehicle class. Only investigate the difference between the two main fuel types found in the 'fuelType' variable: Regular Gasoline and Premium Gasoline. (The other fuel types represented in the dataset are of much lower frequency compared to the main two, that they'll be more distracting than informative.)

[ ]:
```
# Use a plot to explore whether or not there are differences in recommended␣
 ↪fuel type depending on the vehicle class. Only investigate the difference␣
 ↪between the two main fuel types found in the 'fuelType' variable: Regular␣
 ↪Gasoline and Premium Gasoline.
plt.figure(figsize=(20, 8))
# Select the rows where the fuel type is either "Regular Gasoline" or "Premium␣
 ↪Gasoline"
fuel_econ_sub = fuel_econ.query('fuelType in ["Regular Gasoline", "Premium␣
 ↪Gasoline"]')
# Use seaborn's countplot to plot the number of vehicles in each class, with␣
 ↪the color representing the fuel type
sb.countplot(data=fuel_econ_sub, x='VClass', hue='fuelType')
plt.xlabel('Vehicle Class')
plt.ylabel('Count');
```



From this plot, you can see that more cars use premium gas over regular gas, and that the smaller cars are biased towards the premium gas grade. It is only in midsize sedans where regular gasoline was used in more cars than premium gasoline.

1.4.9 Faceting

Faceting is a useful way to visualize the distribution of a numeric variable for different categories. It is similar to a clustered bar chart, but it shows the distribution of the numeric variable for each category, rather than the total value of the numeric variable for each category. The faceting is on the lower level of abstraction. For each level of the categorical variable, a distribution of the values on the numeric variable is plotted. The distribution is plotted as a kernel density estimate, something like a smoothed histogram. There is an extra section at the previous lesson that provides more insight into kernel density estimates.

[ ]:
```
df = sns.load_dataset('tips')
df.head()
```

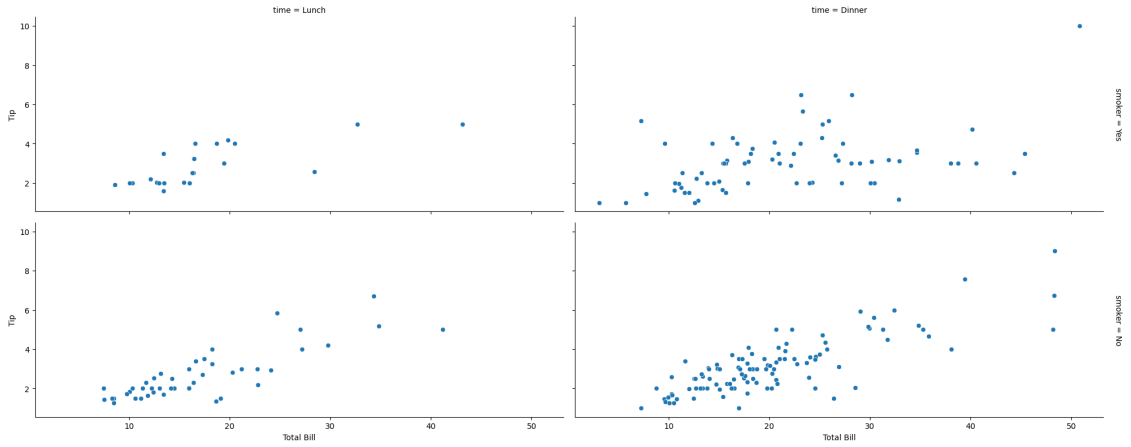
[ ]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

This DataFrame is loaded using Seaborn's built-in `load_dataset` function, which loads a sample dataset of restaurant tips. It has several columns including `total_bill`, `tip`, `sex`, `smoker`, `day`, `time`, and `size`. The dataset contains information about the total bill, tip amount, and various attributes of the diners (such as gender and smoking status).

To create a FacetGrid with Seaborn, we can use the following code:

[ ]:
```
g = sns.FacetGrid(df, row='smoker', col='time', margin_titles=True, height=4,␣
 ↪aspect=2.5)
g.map(sns.scatterplot, 'total_bill', 'tip')
g.set_axis_labels('Total Bill', 'Tip');
```



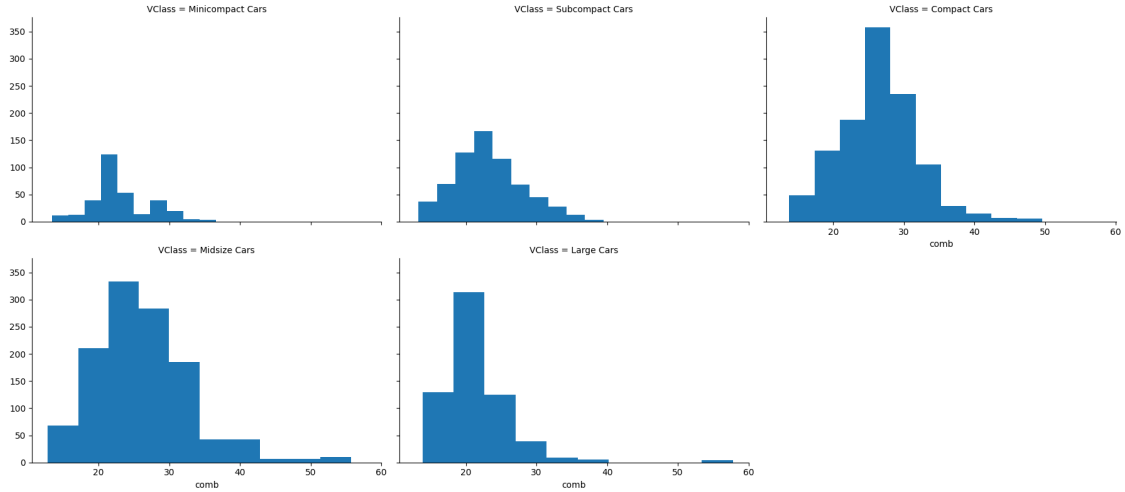
This code creates a `FacetGrid` object that will allow us to facet the data based on the values in the **'smoker'** and **'time'** columns. The `margin_titles=True` parameter adds titles to the top and right margins of the plot.

We then use the `map` method to apply a `scatterplot` to each facet of the FacetGrid. The `scatterplot` function takes two column names as its input and produces a scatter plot of the values in those columns.

The resulting plot will show a grid of scatter plots, with one plot for each unique combination of 'smoker' and 'time'. The x-axis of each plot represents the 'total_bill' column and the y-axis represents the 'tip' column. This type of plot is useful for visualizing how different variables relate to each other across multiple categories or groups.

[ ]:
```
# Convert the "VClass" column from a plain object type into an ordered␣
 ↪categorical type
sedan_classes = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars',␣
 ↪'Midsize Cars', 'Large Cars']
vclasses = pd.api.types.CategoricalDtype(ordered=True, categories=sedan_classes)
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclasses);

# Plot the Seaborn's FacetGrid
g = sb.FacetGrid(data = fuel_econ, col = 'VClass', height=4, aspect=1.5,␣
 ↪col_wrap=3)
g.map(plt.hist, "comb")
```

[ ]:
```
<seaborn.axisgrid.FacetGrid at 0x7fe20ab10af0>
```



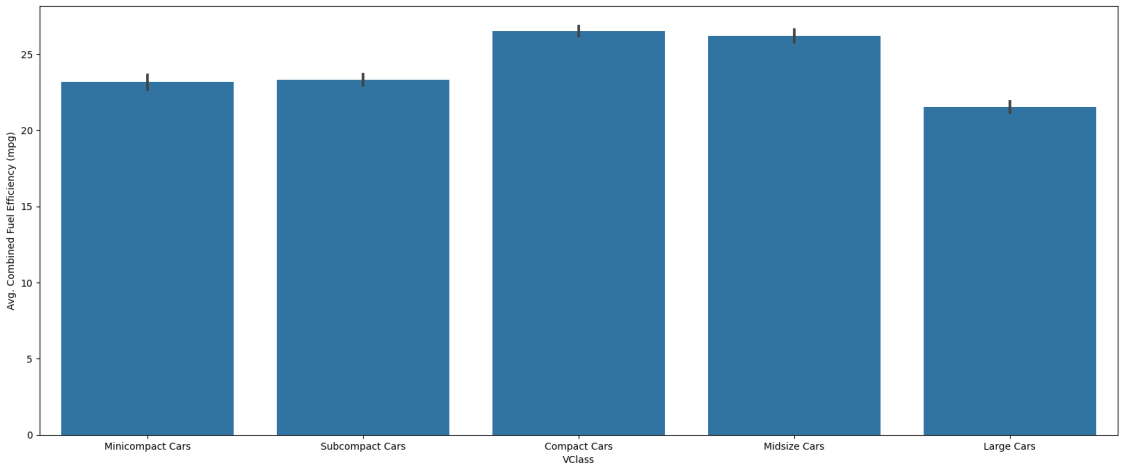
Notice that each subset of the data is being plotted independently. Each uses the default of ten bins from hist to bin together the data, and each plot has a different bin size. Despite that, the axis limits on each facet are the same to allow clear and direct comparisons between groups. It's still worth cleaning things a little bit more by setting the same bin edges on all facets. Extra visualization parameters can be set as additional keyword arguments to the map function.

1.4.10 Adapted Bar Charts

Histograms and bar charts were introduced in the previous lesson as depicting the distribution of numeric and categorical variables, respectively, with the height (or length) of bars indicating the number of data points that fell within each bar's range of values. These plots can be adapted for use as bivariate plots by, instead of indicating count by height, indicating a mean or other statistic on a second variable.

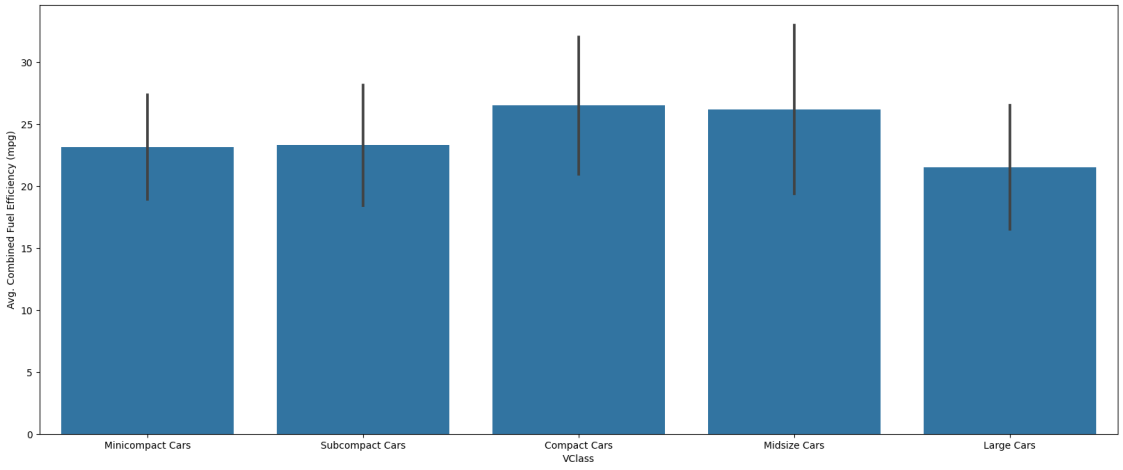
For example, we could plot a numeric variable against a categorical variable by adapting a bar chart so that its bar heights indicate the mean of the numeric variable. This is the purpose of seaborn's `barplot` function:

[ ]:
```
plt.figure(figsize=(20, 8))
base_color = sb.color_palette()[0]
sb.barplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)');
```



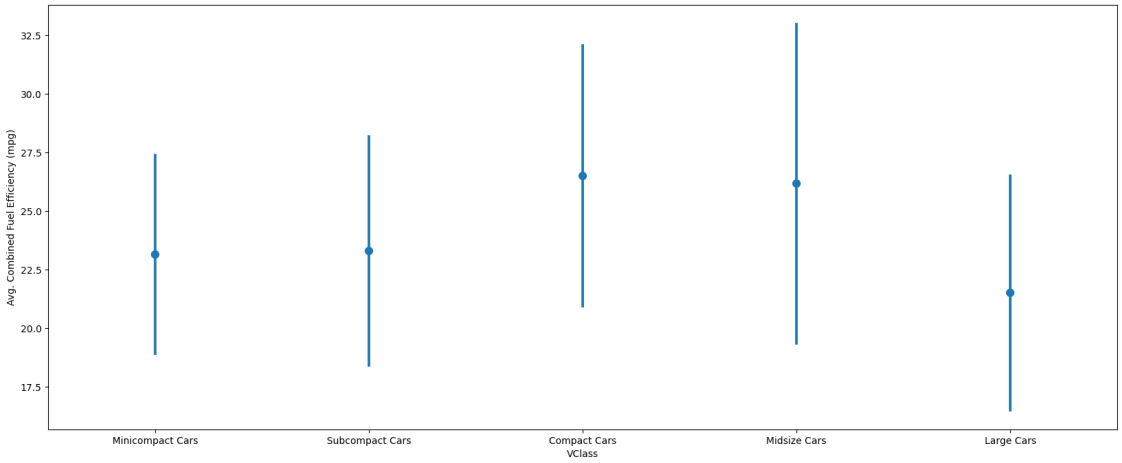
The bar heights indicate the mean value on the numeric variable, with error bars plotted to show the uncertainty in the mean based on variance and sample size.

[ ]:
```
# Lets adapt the above code to use mean instead of count
plt.figure(figsize=(20, 8))
base_color = sb.color_palette()[0]
sb.barplot(data=fuel_econ, x='VClass', y='comb', color=base_color,␣
 ↪errorbar='sd')
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)');
```

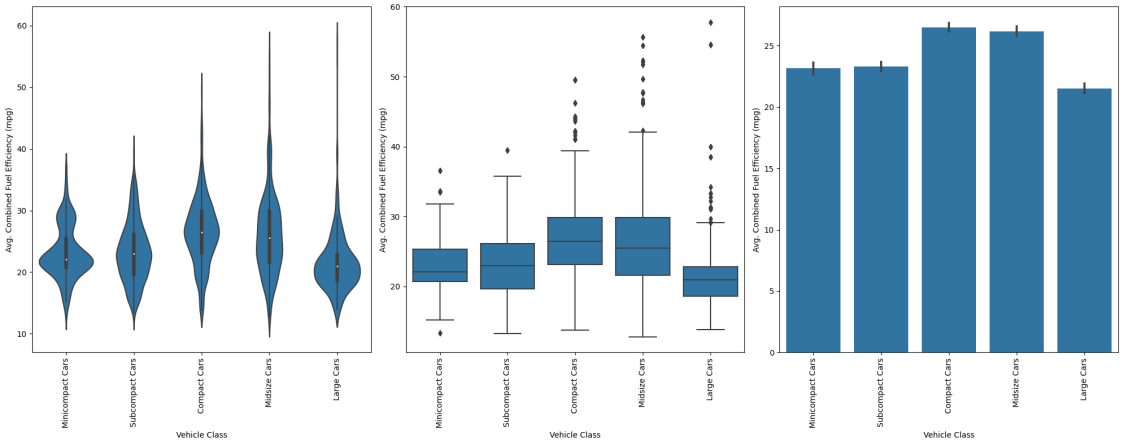


As an alternative, the `pointplot()` function can be used to plot the averages as points rather than bars. This can be useful if having bars in reference to a 0 baseline aren't important or would be confusing.

[ ]:
```
plt.figure(figsize=(20, 8))
sb.pointplot(data=fuel_econ, x='VClass', y='comb', color=base_color,␣
 ↪errorbar='sd', linestyles="")
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)');
```



[ ]:
```
# Using 3 subplots, violin plots, box plots, and bar plots to compare the␣
 ↪distributions of combined fuel efficiency across vehicle classes.
plt.figure(figsize=(20, 8))
plt.subplot(1, 3, 1)
sb.violinplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.subplot(1, 3, 2)
sb.boxplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.subplot(1, 3, 3)
sb.barplot(data=fuel_econ, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.tight_layout();
```



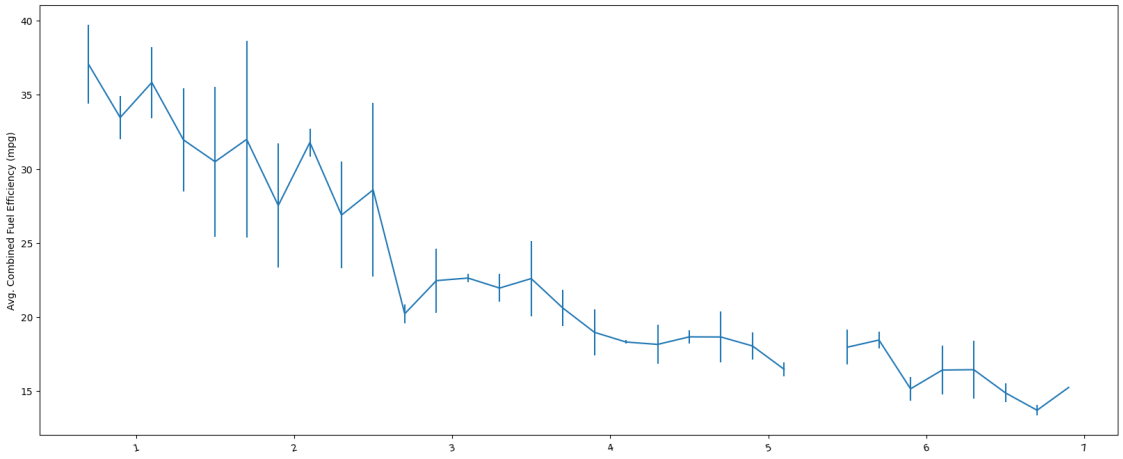
1.4.11 Line Charts

Line charts are a useful way to visualize the distribution of a numeric variable for different categories. They are similar to scatter plots. In contrast to a scatterplot, where all data points are plotted, in a line plot, only one point is plotted for every unique x-value or bin of x-values (like a histogram). If there are multiple observations in an x-bin, then the y-value of the point plotted in the line plot will be a summary statistic (like mean or median) of the data in the bin. The plotted points are connected with a line that emphasizes the sequential or connected nature of the x-values.

[ ]:
```
# Set a number of bins into which the data will be grouped.
# Set bin edges, and compute center of each bin
bin_edges = np.arange(0.6, 7+0.2, 0.2)
bin_centers = bin_edges[:-1] + 0.1

# Cut the bin values into discrete intervals. Returns a Series object.
displ_binned = pd.cut(fuel_econ['displ'], bin_edges, include_lowest = True)
displ_binned
```

[ ]:
```
0       (3.6, 3.8]
1       (1.8, 2.0]
2       (3.4, 3.6]
3       (3.4, 3.6]
4       (2.2, 2.4]
           …
3924    (1.6, 1.8]
3925    (1.8, 2.0]
3926    (1.8, 2.0]
3927    (3.2, 3.4]
3928    (3.2, 3.4]
Name: displ, Length: 3929, dtype: category
Categories (32, interval[float64, right]): [(0.599, 0.8] < (0.8, 1.0] < (1.0,
1.2] < (1.2, 1.4] … (6.2, 6.4] < (6.4, 6.6] < (6.6, 6.8] < (6.8, 7.0]]
```

```
[ ]: plt.figure(figsize=(20, 8))
     # For the points in each bin, we compute the mean and standard error of the␣
      ↪mean.
     comb_mean = fuel_econ['comb'].groupby(displ_binned).mean()
     comb_std = fuel_econ['comb'].groupby(displ_binned).std()

     # Plot the summarized data
     plt.errorbar(x=bin_centers, y=comb_mean, yerr=comb_std)
     plt.xticks(rotation=15);
     plt.ylabel('Avg. Combined Fuel Efficiency (mpg)');
```



Instead of computing summary statistics on fixed bins, you can also make computations on a rolling window through use of pandas' rolling method. Since the rolling window will make computations on sequential rows of the dataframe, we should use sort_values to put the x-values in ascending order first.

1.4.12 Summary

In this lesson, you have learned to code the following types of visualizations:

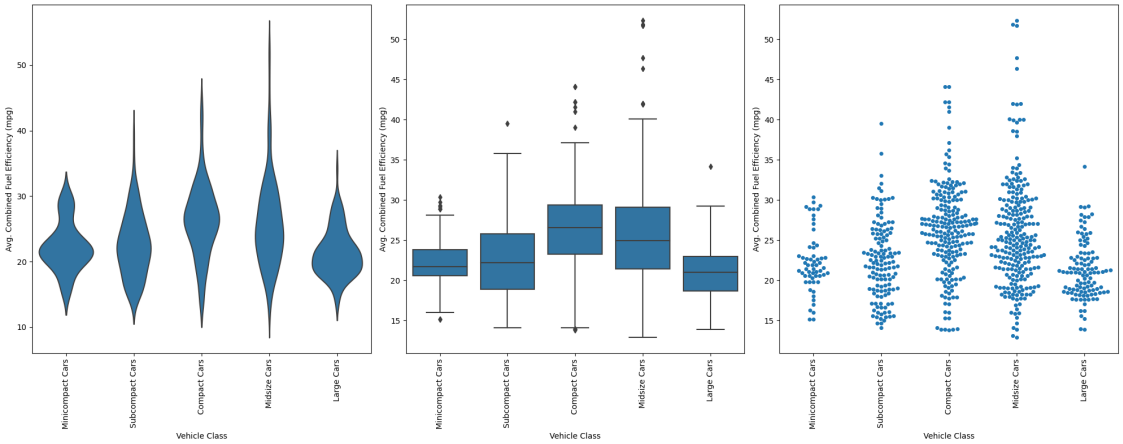
- **scatterplots**: show the relationship between two quantitative variables
- **clustered bar charts**: show the relationship between two qualitative variables
- **heat maps**: used as 2D histograms and bar charts
- **violin and box plots**: show the relationship between one quantitative and one quantitative variable
- **faceting**: adapt univariate plots to bivariate datavariate ex
- **line plots**: show changes in value across time

1.4.13 Swarm Plot

A swarm plot is a type of categorical scatterplot where the points are adjusted (only along the categorical axis) so that they don't overlap. This gives a better representation of the distribution

of values, although it does not scale as well to large numbers of observations (both in terms of the ability to show all the points and in terms of the computation needed to arrange them).

[ ]:
```
# Using 3 subplots, violin plots, box plots, and swarm plot to compare the␣
 ↪distributions of combined fuel efficiency across vehicle classes.

# limit the rows to be plotted to make the plot easier to read to 150
fuel_econ_subset = fuel_econ.sample(750, random_state=42)
plt.figure(figsize=(20, 8))
plt.subplot(1, 3, 1)
sb.violinplot(data=fuel_econ_subset, x='VClass', y='comb', color=base_color,␣
 ↪inner=None)
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.subplot(1, 3, 2)
sb.boxplot(data=fuel_econ_subset, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.subplot(1, 3, 3)
sb.swarmplot(data=fuel_econ_subset, x='VClass', y='comb', color=base_color)
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.tight_layout();
```



Looking at the plots side by side, you can see relative pros and cons of the swarm plot. Unlike the violin plot and box plot, every point is plotted, so we can now compare the frequency of each group in the same plot. While there is some distortion due to location jitter, we also have a more concrete picture of where the points actually lie, removing the long tails that can be present in violin plots.

However, it is only reasonable to use a swarm plot if we have a small or moderate amount of data.

If we have too many points, then the restrictions against overlap will cause too much distortion or require a lot of space to plot the data comfortably.

1.5 LESSON 5: MULTIVARIATE EXPLORATION

1.5.1 Introduction

In this lesson, you will learn how to visualize the relationship between three or more variables.

1.5.2 Multivariate Exploration

Multivariate exploration is the process of visualizing the relationship between three or more variables. This is a more complex task than univariate and bivariate exploration, and it requires more thought and planning. The goal of multivariate exploration is to find patterns and relationships in the data that are not obvious when looking at only one or two variables at a time.

1.5.3 Lesson Overview

In this lesson we will be covering the following topics: - Non-Positional Encodings - Color Palettes - Faceting in Two Directions - Adaptations of Bivariate Plots - Plot Matrices - Feature Engineering

1.5.4 Non-Positional Encodings for Third Variables

There are four major cases to consider when we want to plot three variables together: - Three numeric variables - Two numeric variables and one categorical variable - One numeric variable and two categorical variables - Three categorical variables

A **numerical variable** is a variable where the value has meaning (i.e., weight or age), but a value such as a phone number doesn't have meaning in the numbers alone. A **categorical variable** is a variable that holds a type (i.e., species or hair color).

If we have at least two numeric variables, as in the first two cases, one common method for depicting the data is by using a scatterplot to encode two of the numeric variables, then using a non-positional encoding on the points to convey the value on the third variable, whether numeric or categorical. (You will see additional techniques later in the lesson that can also be applied to the other two cases, i.e., where we have at least two categorical variables.)

Three main non-positional encodings stand out: 1. shape 2. size 3. color

Encoding via Color The most common non-positional encoding is color. Color is a good choice for encoding a third variable because it is easy for humans to distinguish between different hues. However, it is important to note that color is not a good choice for encoding a third variable when the third variable is a categorical variable with more than two levels. This is because it is difficult for humans to distinguish between more than a few different hues, and it is also difficult to distinguish between hues that are similar to each other.

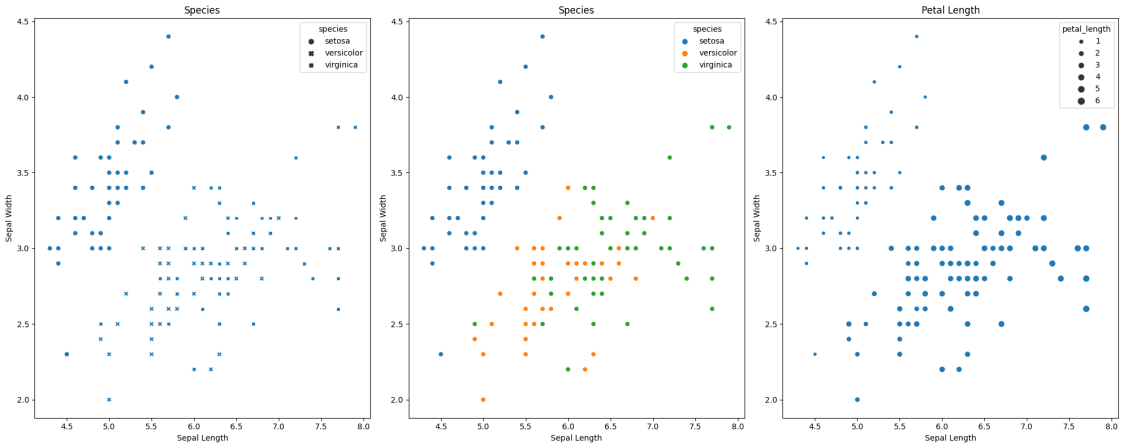
Encoding via Shape Another common non-positional encoding is shape. This is a good choice for encoding a third variable when the third variable is a categorical variable with more than two levels. Unfortunately, there is no built-in way to automatically assign different shapes in a single call of the scatter or regplot function. Instead, we need to write a loop to call our plotting function

multiple times, isolating data points by categorical level and setting a different "marker" argument value for each one.

Encoding via Size The third common non-positional encoding is size. This is a good choice for encoding a third variable when the third variable is a numeric variable. Unfortunately, there is no built-in way to automatically assign different sizes in a single call of the scatter or regplot function. Instead, we need to write a loop to call our plotting function multiple times, isolating data points by size level and setting a different "s" argument value for each one.

[ ]:
```
import seaborn as sns
import pandas as pd

df = sns.load_dataset('iris')
```

[ ]:
```
plt.figure(figsize=(20, 8))
# A 3 subplot. 1 plot to show color, 1 plot to show shape, and 1 plot to show␣
 ↪size
plt.subplot(1, 3, 1)
sns.scatterplot(data=df, x='sepal_length', y='sepal_width', style='species')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Species')
plt.subplot(1, 3, 2)
sns.scatterplot(data=df, x='sepal_length', y='sepal_width', hue='species')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Species')
plt.subplot(1, 3, 3)
sns.scatterplot(data=df, x='sepal_length', y='sepal_width', size='petal_length')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Petal Length')
plt.tight_layout();
```

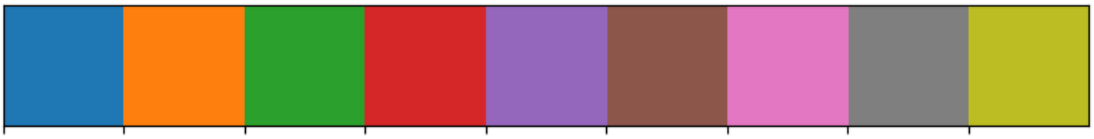


1.5.5 Color Palettes

Depending on the type of data you have, you may want to change the type of color palette that you use to depict your data. There are three major classes of color palette to consider: 1. Qualitative 2. Sequential 3. Diverging

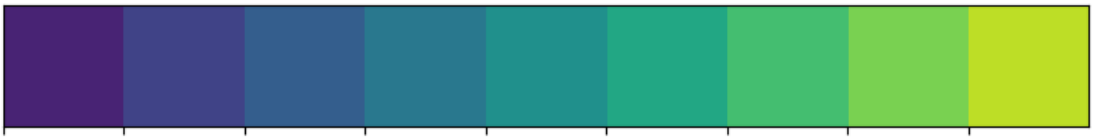
Qualitative palettes are built for nominal-type data. This is the palette class taken by the default palette. In a qualitative palette, consecutive color values are distinct so that there is no inherent ordering of levels implied. Colors in a good qualitative palette should also try and avoid drastic changes in brightness and saturation that would cause a reader to interpret one category as being more important than the others

[ ]:
```
sb.palplot(sb.color_palette(n_colors=9))
```



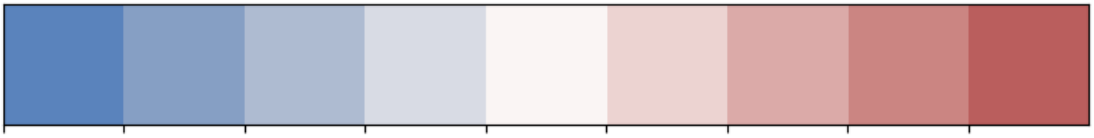
For other types of data (ordinal and numeric), a choice may need to be made between a **sequential** scale and a **diverging** scale. In a **sequential palette**, consecutive color values should follow each other systematically. Typically, this follows a light-to-dark trend across a single or small range of hues, where light colors indicate low values and dark colors indicate high values. The default sequential color map, "viridis".

[ ]:
```
sb.palplot(sb.color_palette('viridis', 9))
```



Most of the time, a sequential palette will depict ordinal or numeric data just fine. However, if there is a meaningful zero or center value for the variable, you may want to consider using a **diverging palette**. In a diverging palette, two sequential palettes with different hues are put back to back, with a common color (usually white or gray) connecting them

[ ]:
```
sb.palplot(sb.color_palette('vlag', 9))
```



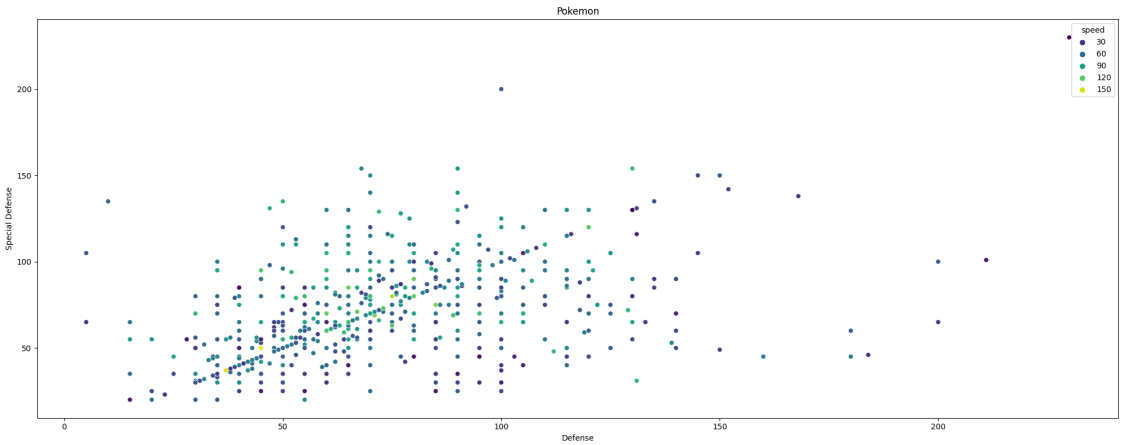
1.5.6 Selecting Color Palettes

If you want to change the color map for your plot, the easiest way of doing so is by using one of the built-ins from Matplotlib or Seaborn. This part of the Matplotlib documentation has a list of strings that can be understood for color mappings. For most of your purposes, stick with the palettes noted in the top few tables as built-in for Matplotlib ('viridis', etc.) or from ColorBrewer; the remaining palettes may not be as perceptually consistent. Seaborn also adds in a number of its own palettes:

- **Qualitative** (all up to 6 colors): 'deep', 'pastel', 'dark', 'muted', 'bright', 'colorblind'
- **Sequential**: 'rocket' (white-orange-red-purple-black), 'mako' (mint-green-blue-purple-black)
- **Diverging**: 'vlag' (blue-white-red), 'icefire' (blue-black-orange) For all of these strings, appending `_r` reverses the palette, which is useful if a sequential or diverging palette is rendered counter to your expectations.

Question 1 To start, let's look at the relationship between the Pokémon combat statistics of Speed, Defense, and Special-Defense. If a Pokémon has higher defensive statistics, does it necessarily sacrifice speed? Create a single plot to depict this relationship.

[ ]:
```
# Plot scatter plot of defence vs special defence for pokemon
plt.figure(figsize=(20, 8))
sb.scatterplot(data=pokemon, x='defense', y='special-defense', hue='speed',␣
 ↪palette='viridis')
plt.xlabel('Defense')
plt.ylabel('Special Defense')
plt.title('Pokemon')
plt.tight_layout();
```



To complete the second task, we need to first reshape the dataset so that all Pokémon types are recorded in a single column. This will add duplicates of Pokémon with two types, which is fine for the task to be performed.

[ ]:
```
# Melt the type columns into a single column
type_cols = ['type_1', 'type_2']
non_type_cols = pokemon.columns.difference(type_cols)
melted_df = pokemon.melt(id_vars=non_type_cols, value_vars=type_cols,␣
 ↪var_name='type_level', value_name='type').dropna()
melted_df.head()
```

[ ]:
```
   attack  base_experience  defense  generation_id  height  hp  id  \
0      49               64       49              1     0.7  45   1
1      62              142       63              1     1.0  60   2
2      82              236       83              1     2.0  80   3
3      52               62       43              1     0.6  39   4
4      64              142       58              1     1.1  58   5

   special-attack  special-defense     species  speed  weight type_level  \
0              65               65   bulbasaur     45     6.9     type_1
1              80               80     ivysaur     60    13.0     type_1
2             100              100    venusaur     80   100.0     type_1
3              60               50  charmander     65     8.5     type_1
4              80               65  charmeleon     80    19.0     type_1

    type
0  grass
1  grass
2  grass
3   fire
4   fire
```

1.5.7 Question 2

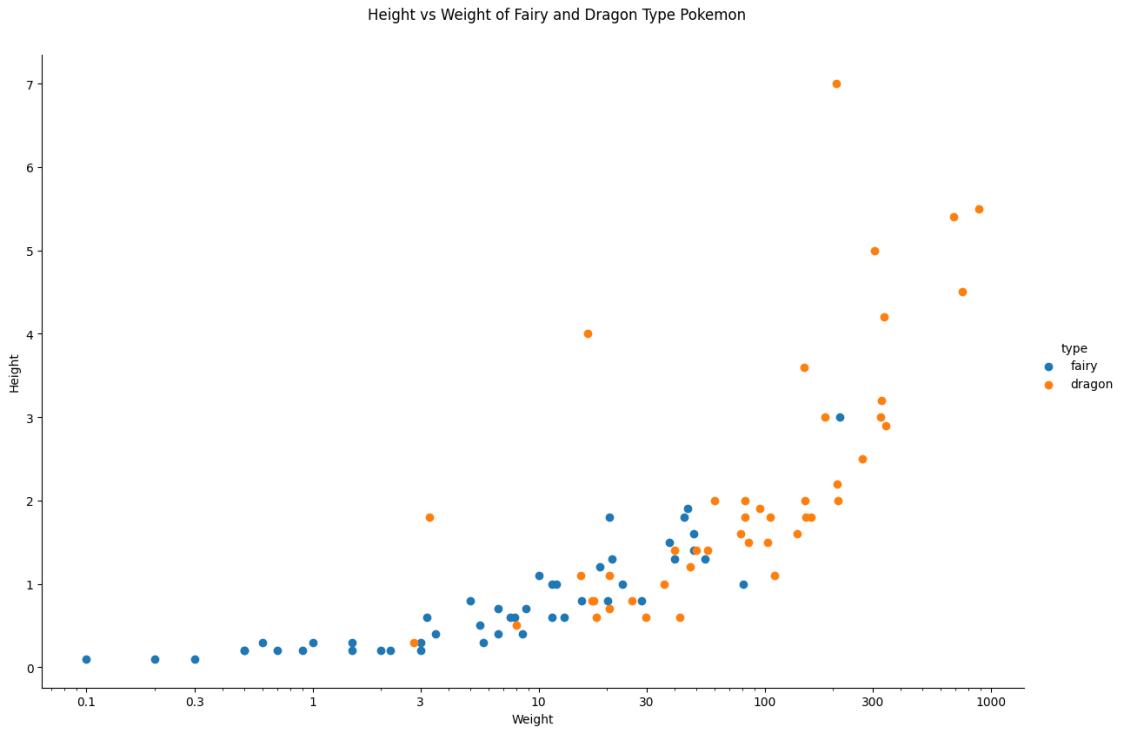
How do weights and heights compare between Fairy type Pokémon and Dragon type Pokémon?

[ ]:
```
# Select the type where it is fairy or dragon using the query function and␣
 ↪assign it to the variable fairy_dragon_df
fairy_dragon_df = melted_df.query('type == "fairy" or type == "dragon"')
fairy_dragon_df.head()
```

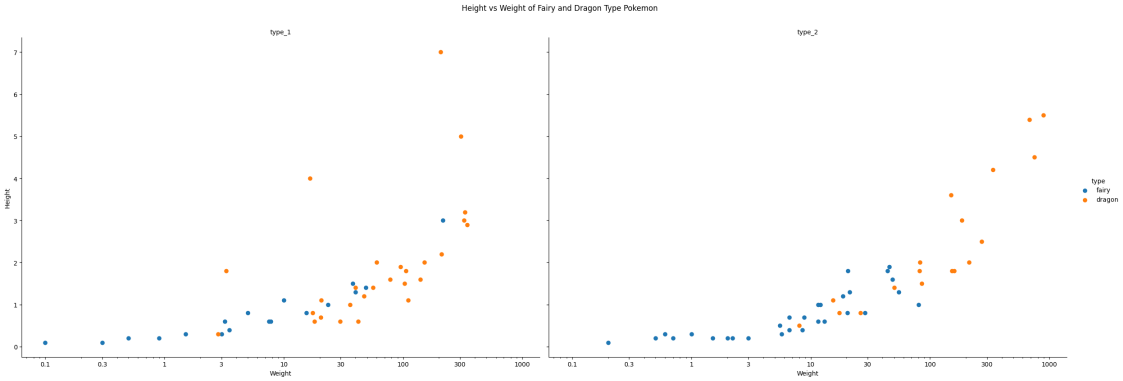
[ ]:
```
     attack  base_experience  defense  generation_id  height  hp   id  \
34       45              113       48              1     0.6  70   35
35       70              217       73              1     1.3  95   36
146      64               60       45              1     1.8  41  147
```

```
147      84              147       65              1     4.0  61  148
148     134              270       95              1     2.2  91  149

     special-attack  special-defense    species  speed  weight type_level  \
34               60               65   clefairy     35     7.5     type_1
35               95               90   clefable     60    40.0     type_1
146              50               50    dratini     50     3.3     type_1
147              70               70  dragonair     70    16.5     type_1
148             100              100  dragonite     80   210.0     type_1

       type
34    fairy
35    fairy
146  dragon
147  dragon
148  dragon
```

[ ]:
```
# Create a facetgrid
g = sb.FacetGrid(data=fairy_dragon_df,  height=8, aspect=1.5, hue='type')
g.map(plt.scatter, 'weight', 'height')
g.set_axis_labels('Weight', 'Height')
g.set_titles('{col_name}')
g.fig.suptitle('Height vs Weight of Fairy and Dragon Type Pokemon', y=1.05)
# Set scale to log x
g.set(xscale='log')
x_ticks = [0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000]
g.set(xticks=x_ticks, xticklabels=x_ticks)
g.add_legend();
```



[ ]:
```
# Create a facetgrid
g = sb.FacetGrid(data=fairy_dragon_df,  height=8, aspect=1.5, hue='type',␣
 ↪col='type_level')
g.map(plt.scatter, 'weight', 'height')
g.set_axis_labels('Weight', 'Height')
g.set_titles('{col_name}')
g.fig.suptitle('Height vs Weight of Fairy and Dragon Type Pokemon', y=1.05)
# Set scale to log x
g.set(xscale='log')
x_ticks = [0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000]
g.set(xticks=x_ticks, xticklabels=x_ticks)
g.add_legend();
```



1.5.8 Faceting for Multivariate Data

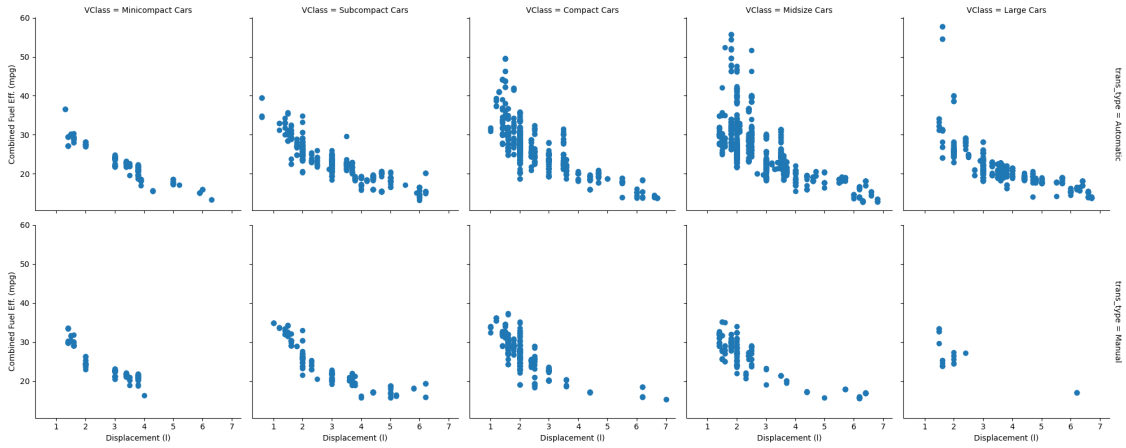
In the previous lesson, you saw how FacetGrid could be used to subset your dataset across levels of a categorical variable, and then create one plot for each subset. Where the faceted plots demonstrated were univariate before, you can actually use any plot type, allowing you to facet bivariate plots to create a multivariate visualization.

FacetGrid also allows for faceting a variable not just by columns, but also by rows. We can set one categorical variable on each of the two facet axes for one additional method of depicting multivariate trends.

```
g = sb.FacetGrid(data = df, col = 'cat_var2', row = 'cat_var1', height = 2.5,
                 margin_titles = True)
g.map(plt.scatter, 'num_var1', 'num_var2')
```

Setting `margin_titles = True` means that instead of each facet being labeled with the combination of row and column variable, labels are placed separately on the top and right margins of the facet grid. This is a boon, since the default plot titles are usually too long.

[ ]:
```
# Create a facetgrid with vehicle data with rows as transmission type and␣
 ↪columns as vehicle class
g = sb.FacetGrid(data=fuel_econ,  height=4, col='VClass', row='trans_type',␣
 ↪margin_titles=True)
g.map(plt.scatter, 'displ', 'comb')
g.set_axis_labels('Displacement (l)', 'Combined Fuel Eff. (mpg)');
```



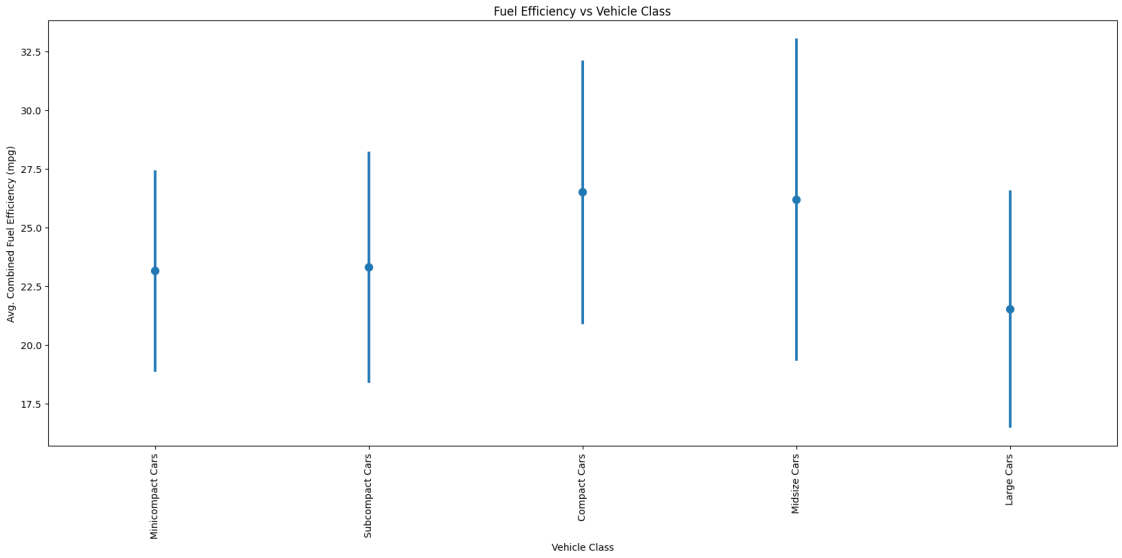
1.5.9 Other Adaptations of Bivariate Plots

You also saw one other way of expanding univariate plots into bivariate plots in the previous lesson: substituting count on a bar chart or histogram for the mean, median, or some other statistic of a

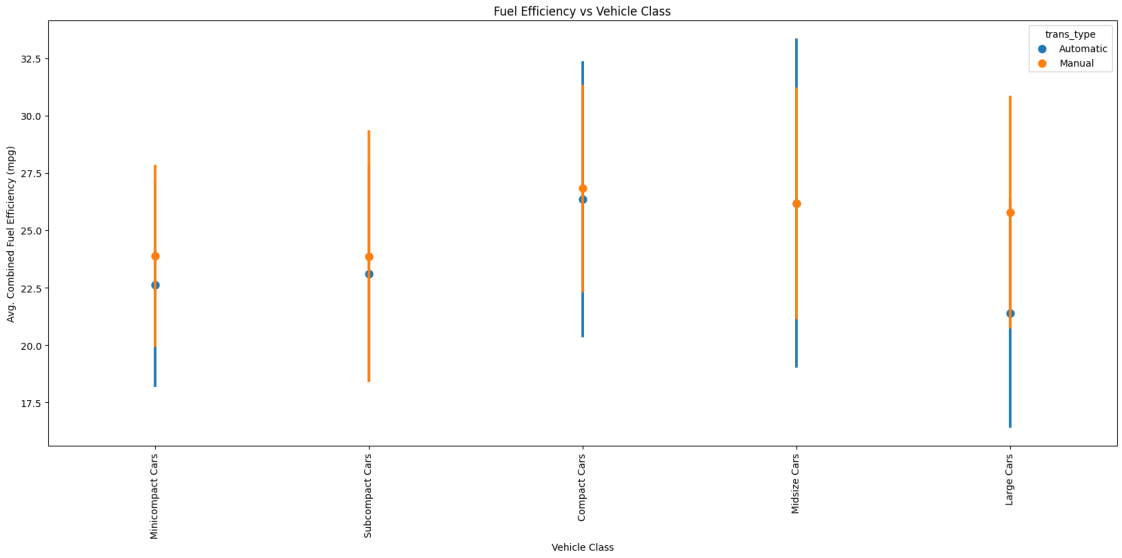
second variable. This adaptation can also be done for bivariate plots like the heat map, clustered bar chart, and line plot, to allow them to depict multivariate relationships.

If we want to depict the mean of a third variable in a **2-d histogram**, we need to change the weights of points in the hist2d function similar to how we changed the weights in the 1-d histogram.

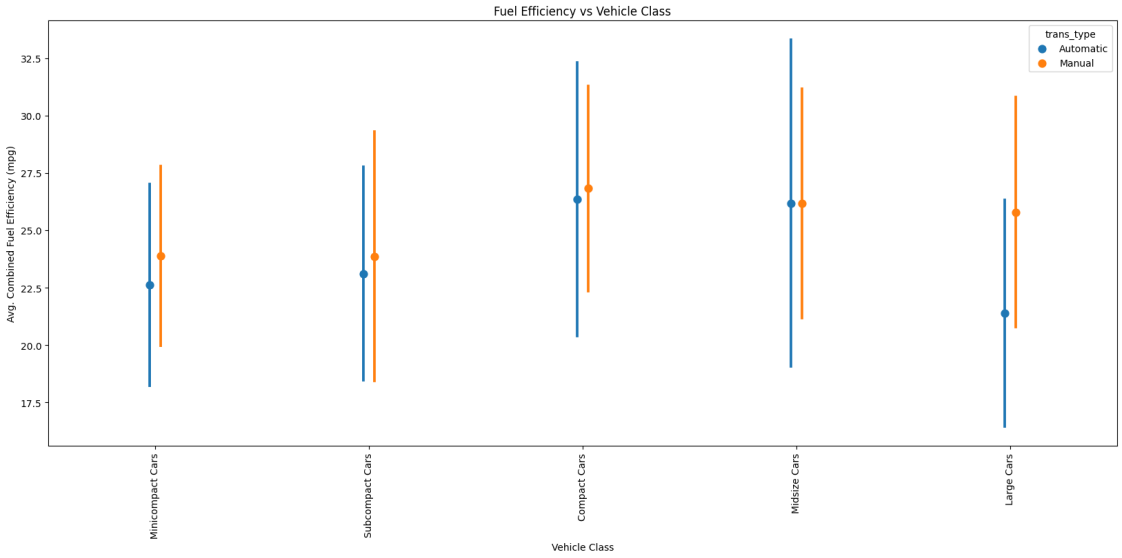
[ ]:
```
# Plot a pinplot of fuel efficiency vs vehicle class
plt.figure(figsize=(20, 8))
sb.pointplot(data=fuel_econ, x='VClass', y='comb', linestyles='', errorbar='sd')
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.title('Fuel Efficiency vs Vehicle Class');
```



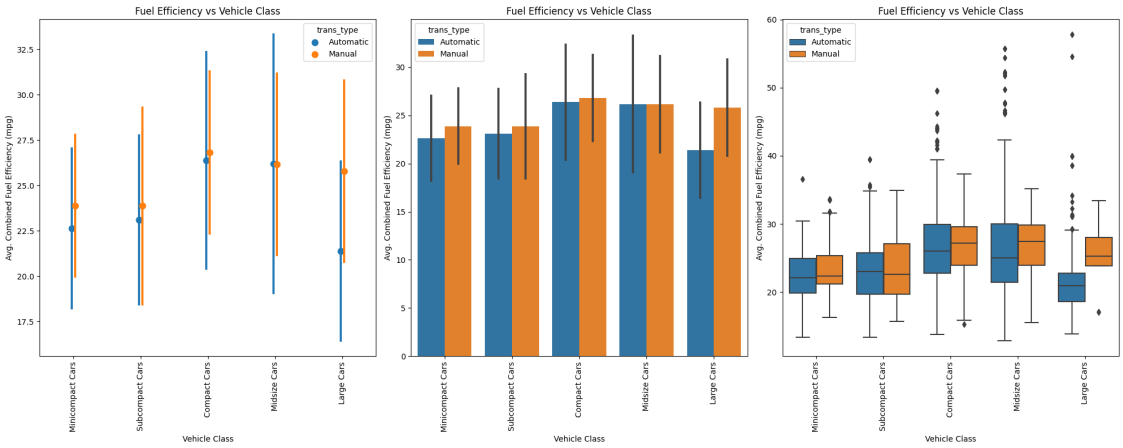
[ ]:
```
# Lets add a hue to be transmission type
plt.figure(figsize=(20, 8))
sb.pointplot(data=fuel_econ, x='VClass', y='comb', linestyles='',␣
 ↪errorbar='sd', hue='trans_type')
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.title('Fuel Efficiency vs Vehicle Class');
```



[ ]:
```
# Add a dodge to the plot
plt.figure(figsize=(20, 8))
sb.pointplot(data=fuel_econ, x='VClass', y="comb", errorbar='sd',␣
 ↪hue='trans_type', dodge=True, linestyles='')
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.title('Fuel Efficiency vs Vehicle Class');
```



```
[ ]: # Create a 3 subplot of pointplot, barplot, and boxplot of fuel efficiency vs␣
      ↪vehicle class and transmission type
     plt.figure(figsize=(20, 8))
     plt.subplot(1, 3, 1)
     sb.pointplot(data=fuel_econ, x='VClass', y="comb", errorbar='sd',␣
      ↪hue='trans_type', dodge=True, linestyles='')
     plt.xticks(rotation=90)
     plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
     plt.xlabel('Vehicle Class')
     plt.title('Fuel Efficiency vs Vehicle Class')
     plt.subplot(1, 3, 2)
     sb.barplot(data=fuel_econ, x='VClass', y="comb", errorbar='sd',␣
      ↪hue='trans_type')
     plt.xticks(rotation=90)
     plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
     plt.xlabel('Vehicle Class')
     plt.title('Fuel Efficiency vs Vehicle Class')
     plt.subplot(1, 3, 3)
     sb.boxplot(data=fuel_econ, x='VClass', y="comb", hue='trans_type')
     plt.xticks(rotation=90)
     plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
     plt.xlabel('Vehicle Class')
     plt.title('Fuel Efficiency vs Vehicle Class')
     plt.tight_layout();
```

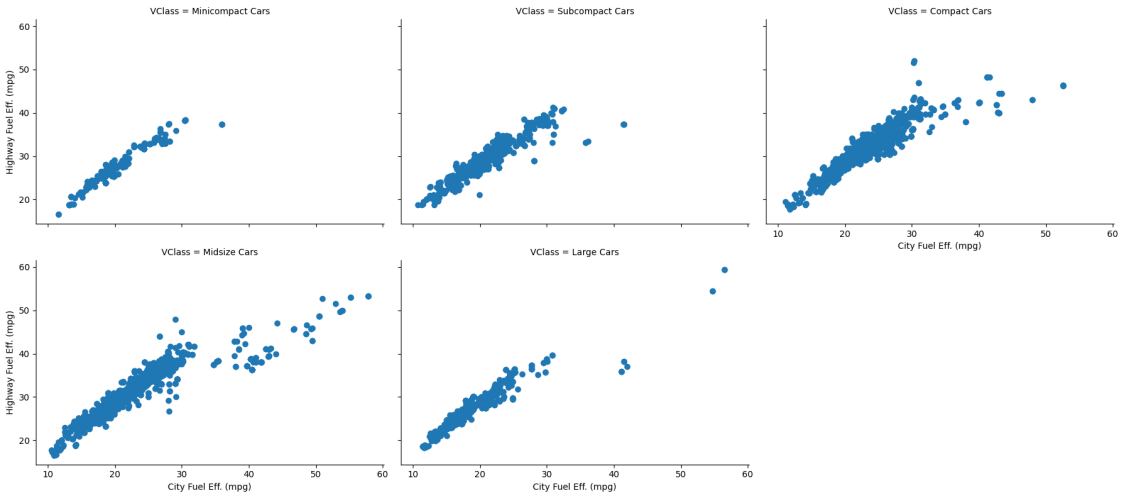


1.5.10 Question 1

Plot the city ('city') vs. highway ('highway') fuel efficiencies (both in mpg) for each vehicle class ('VClass'). Don't forget that vehicle class is an ordinal variable with levels {Minicompact Cars, Subcompact Cars, Compact Cars, Midsize Cars, Large Cars}.

[ ]:
```
# Arrange fuel_econ by VClass in ordinal order of Minicompact Cars, Subcompact␣
 ↪Cars, Compact Cars, Midsize Cars, Large Cars
vclass_order = ['Minicompact Cars', 'Subcompact Cars', 'Compact Cars', 'Midsize␣
 ↪Cars', 'Large Cars']
# Use api.types.CategoricalDtype to create an ordered categorical type
vclass = pd.api.types.CategoricalDtype(ordered=True, categories=vclass_order)
# Convert the VClass column of fuel_econ to an ordered categorical type
fuel_econ['VClass'] = fuel_econ['VClass'].astype(vclass)
```

[ ]:
```
# Plot the city ('city') vs. highway ('highway') fuel efficiencies (both in␣
 ↪mpg) for each vehicle class ('VClass').

# Create a facetgrid
g = sb.FacetGrid(data=fuel_econ,  height=4, aspect=1.5, col='VClass',␣
 ↪col_wrap=3, margin_titles=True)
g.map(plt.scatter, 'city', 'highway')
g.set_axis_labels('City Fuel Eff. (mpg)', 'Highway Fuel Eff. (mpg)');
```



Due to overplotting, I've taken a faceting approach to this task. There don't seem to be any obvious differences in the main cluster across vehicle classes, except that the minicompact and large sedans' arcs are thinner than the other classes due to lower counts. The faceted plots clearly show that most of the high-efficiency cars are in the mid-size and compact car classes.

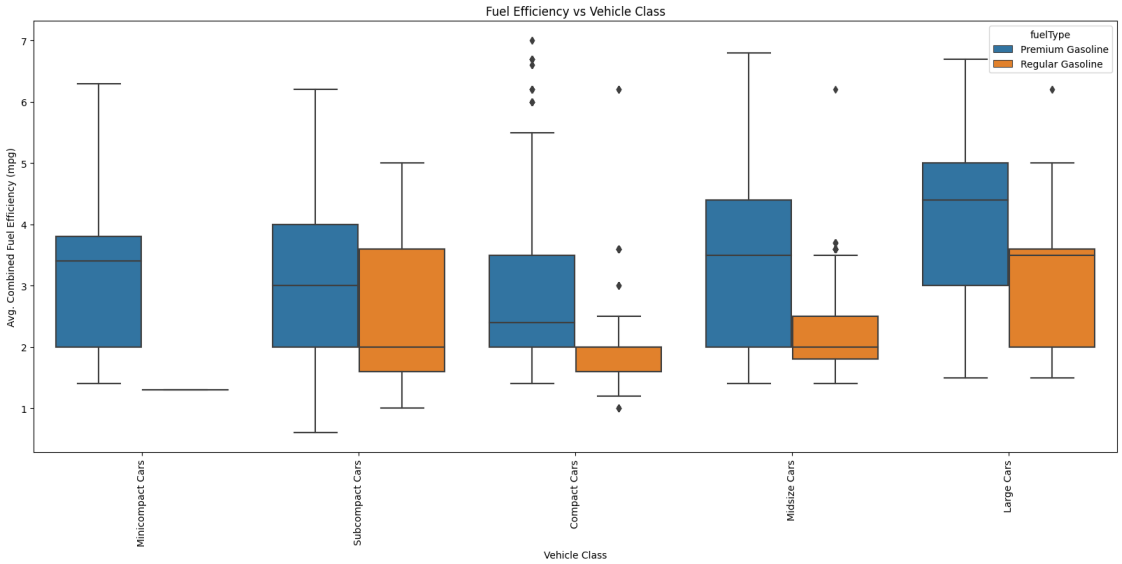
1.5.11 Question 2

Plot the relationship between engine size ('displ', in liters), vehicle class, and fuel type ('fuelType'). For the lattermost feature, focus only on Premium Gasoline and Regular Gasoline cars. What kind of relationships can you spot in this plot?

[ ]:
```
# Filter only Premium Gasoline and Regular Gasoline
fuel_econ_sub = fuel_econ.query('fuelType in ["Premium Gasoline", "Regular␣
↪Gasoline"]')
fuel_econ_sub.head()
```

[ ]:
```
      id        make           model  year           VClass  \
0  32204      Nissan            GT-R  2013  Subcompact Cars
1  32205  Volkswagen              CC  2013     Compact Cars
2  32206  Volkswagen              CC  2013     Compact Cars
3  32207  Volkswagen      CC 4motion  2013     Compact Cars
4  32208   Chevrolet  Malibu eAssist  2013     Midsize Cars

                drive              trans          fuelType  cylinders  displ  \
0    All-Wheel Drive    Automatic (AM6)  Premium Gasoline          6    3.8
1  Front-Wheel Drive  Automatic (AM-S6)  Premium Gasoline          4    2.0
2  Front-Wheel Drive     Automatic (S6)  Premium Gasoline          6    3.6
3    All-Wheel Drive     Automatic (S6)  Premium Gasoline          6    3.6
4  Front-Wheel Drive     Automatic (S6)  Regular Gasoline          4    2.4

   …  pv4     city    UCity  highway  UHighway     comb  co2  feScore  \
0  …    0  16.4596  20.2988  22.5568   30.1798  18.7389  471        4
1  …    0  21.8706  26.9770  31.0367   42.4936  25.2227  349        6
2  …    0  17.4935  21.2000  26.5716   35.1000  20.6716  429        5
3  …    0  16.9415  20.5000  25.2190   33.5000  19.8774  446        5
4  …   95  24.7726  31.9796  35.5340   51.8816  28.6813  310        8

   ghgScore  trans_type
0         4   Automatic
1         6   Automatic
2         5   Automatic
3         5   Automatic
4         8   Automatic

[5 rows x 21 columns]
```

[ ]:
```
# Plot the vehicle vs displ type using clustered box plot
plt.figure(figsize=(20, 8))
sb.boxplot(data=fuel_econ_sub, x='VClass', y='displ', hue='fuelType')
plt.xticks(rotation=90)
plt.ylabel('Avg. Combined Fuel Efficiency (mpg)')
plt.xlabel('Vehicle Class')
plt.title('Fuel Efficiency vs Vehicle Class');
```



I went with a clustered box plot on this task since there were too many levels to make a clustered violin plot accessible. The plot shows that in each vehicle class, engine sizes were larger for premium-fuel cars than regular-fuel cars. Engine size generally increased with vehicle class within each fuel type, but the trend was noisy for the smallest vehicle classes.

[ ]: