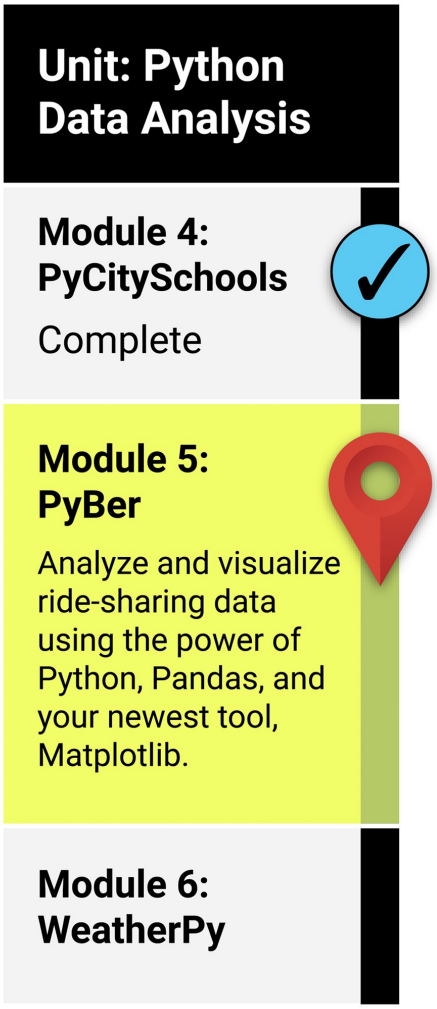
5.0.1: Visualizing Ride-Sharing Data

5.0.2: Module 5 Roadmap

Looking Ahead

In this module, you'll learn how to graph data using the Matplotlib library. Matplotlib has a rich set of features for creating and annotating charts that visualize data in a Data Series or DataFrame. You'll use Matplotlib to create line charts, bar charts, scatter plots, bubble charts, pie charts, and box-and-whisker plots, and make them visually compelling and informative by adding titles, axes labels, legends, and custom colors. You'll also be introduced to SciPy, a statistical Python package, and NumPy, a fundamental package for scientific computing in Python. You'll use Pandas, SciPy, and NumPy to perform summary statistics.

As you build a variety of charts, you'll leverage your knowledge of Python arrays and tuples, and learn how to apply Pandas methods, functions, and conditional expressions, as well as perform mathematical calculations on Series and DataFrames.



What You Will Learn

By the end of this module, you will be able to:

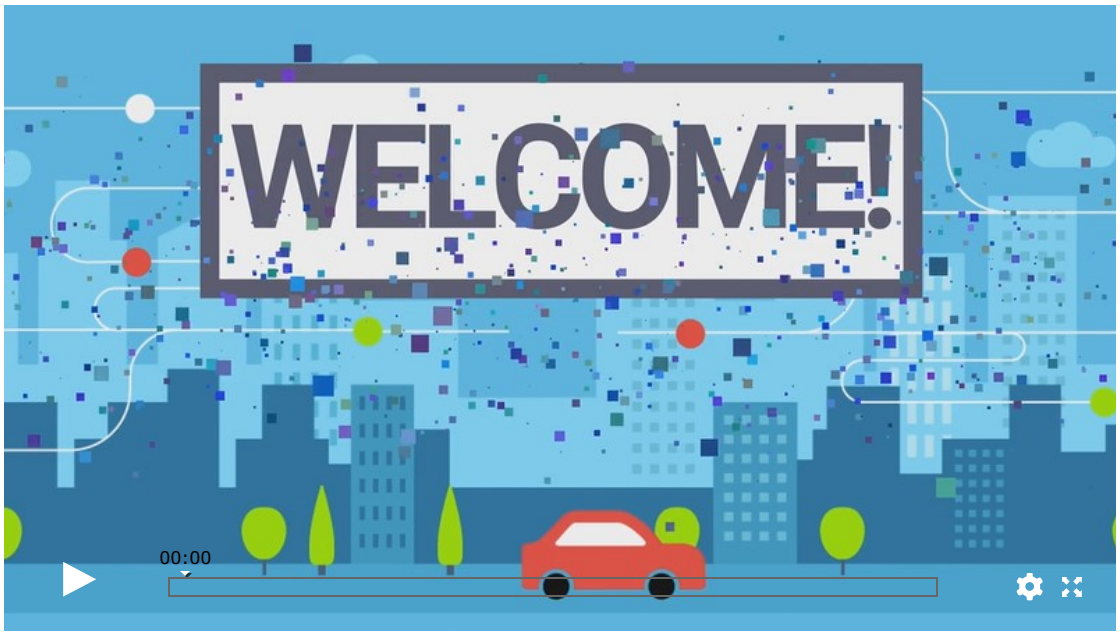
- Create line, bar, scatter, bubble, pie, and box-and-whisker plots using Matplotlib.
- Add and modify features of Matplotlib charts.
- Add error bars to line and bar charts.
- Determine mean, median, and mode using Pandas, NumPy, and SciPy statistics.

Planning Your Schedule

Here's a quick look at the lessons and assignments you'll cover in this module. You can use the time estimates to help pace your learning and plan your schedule.

- Welcome to Module 5 (30 minutes)
- Create Visualizations Using Matplotlib (3 hours)
- Convert CSV Files to Pandas DataFrames (1 hour)
- Create a Bubble Chart for Ride-Sharing Data (3 hours)
- Calculate Summary Statistics (2 hours)
- Pie Chart: Percentage of Total Fares by City Type (1 hour)
- Pie Chart: Percentage of Total Rides by City Type (1 hour)
- Pie Chart: Percentage of Total Drivers by City Type (1 hour)
- Application (5 hours)

5.0.3: PyBer



5.1.1: Create and Clone a New GitHub Repository

It's your second week as a data analyst at PyBer, a ride-sharing app company valued at $2.3 billion. You've just been assigned your first big project: analyze all the rideshare data from January to early May of 2019 and create a compelling visualization for the CEO, V. Isualize.

Even before starting at PyBer, you'd heard stories about V. Isualize. She is a former programmer who started out at MathWorks, a co-founder of PyBer, and is known for being extremely fair yet extremely demanding. Because of her programming expertise, she's particularly insistent that the analytical work be comprehensive and correct. This assignment is both a once-in-a-lifetime challenge and a once-in-a-lifetime opportunity. If you do well, your future at the organization is secure.

Your manager, Omar, is excited for your big break and has agreed to partner with you on getting the visualizations just right. Your first step is to create a GitHub repository for the project so that he can stay in the loop as you work.

Before we get started, let's create a GitHub repository for the project.

Create a GitHub Repository on macOS

> **Rewind**
>
> To create a new repository on GitHub in macOS, do the following:
>
> 1. Log in to GitHub.
> 2. Create a new repository.
> 3. Name the repository PyBer_Analysis.
> 4. Make the repository public.
> 5. Initialize the repository with a README.
> 6. Click “Create Repository.”
> 7. Click the green "Clone or download" button and make sure to use HTTPS.
> 8. Open the command line interface (CLI).
> 9. Navigate into the Class folder by using the necessary commands.
> 10. Once you are in the Class folder, type `git clone` + space and paste the URL that you copied, as shown here:
>
> ```
> <computer_name>:~ Class $ git clone https://github.com/<your_GitHub
> ```
>
> 11. Press Enter.
> 12. If prompted, enter your GitHub username and password. You’ll see progress indicators in the command line as the repo is cloned.
> 13. Once the repository has been cloned, you should see a folder on your computer with the repository name.

Create a GitHub Repository on Windows

> **Rewind**
>
> To create a new repository on GitHub in Windows, do the following:
>
> 1. Log in to GitHub.
> 2. Create a new repository.
> 3. Name the repository PyBer_Analysis.
> 4. Make the repository public.

5. Initialize the repository with a README.
6. Click “Create Repository.”
7. Click the green "Clone or download" button and make sure to use HTTPS.
8. Open Git Bash.
9. Navigate into the Class folder.
10. Once you are in the Class folder, type `git clone` + space and paste the URL that you copied, as shown here:

```
<computer_name>@<home_directory> MINGW32 ~/Class
$:~ Class <home_directory>$ git clone https://github.com/<your_GitH
```

11. Press Enter.
12. If prompted, enter your GitHub username and password. You’ll see progress indicators in the command line as the repository is cloned.
13. Once the repository has been cloned, you should see a folder on your computer with the repository name.

5.1.2: The Matplotlib Library

You've just received the work assignment of a lifetime: creating visualizations of rideshare data for PyBer to help improve access to ride-sharing services and determine affordability for underserved neighborhoods. As a new employee, this is a huge professional opportunity for you!

The first step is to make sure you have the right tools in place. A simple Excel chart isn't going to impress V. Isualize. Instead, you'll use the Python graphing library Matplotlib, which is a favorite tool among data scientists and data analysts because of its robust visualization features.

Omar has given you some backstory about the connection between V. Isualize and Matplotlib. Matplotlib was created as a Python alternative for MATLAB. MATLAB, which is short for matrix laboratory, was developed by the company MathWorks in the 1980s. It enabled scientists to perform linear algebra and numerical analysis without learning the programming language Fortran, which until then had been the best option for complex computations.

With this information, you realize that the CEO is an expert in creating data visualizations. You definitely can't afford to botch this up!

Matplotlib is a graphing and plotting library for Python that comes with the Anaconda installation. To use it with Jupyter Notebook, all we need to do is import it. Matplotlib is often used with another Python library, NumPy, a numerical mathematics library for making arrays or matrices. A **matrix** is a two-dimensional data structure, or a list of lists, in which numbers are arranged into rows and columns.

The following is an example of a 2 x 3 matrix. In this matrix there are two lists, `[1, 4, 7]` and `[-5, -2, 1]`.

```
A = [[1, 4, 7],
  [-5, -2, 1]]
```

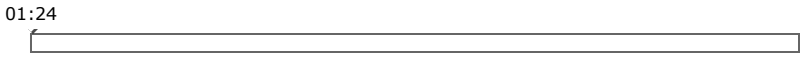
Matplotlib has an advantage over Excel when it comes to graphing because of the robust features it offers. You can use it to create a wide range of graphs, including line plots, bar plots, scatter plots, bubble charts, pie charts, histograms, 3D plots, log plots, and polar plots—to name just a few.

Matplotlib also has rich styling options. You can customize annotations for chart axes, titles and legends, the color and size of lines, bars, and bubbles, and the chart's background. It also lets you save and print publication-quality charts. Because of these features, Matlplotlib is one of the most popular plotting libraries for Python.

Matplotlib has two methods for graphing data. One uses MATLAB's plotting syntax and functionality and the other uses an object-oriented interface. MATLAB's plotting syntax is concise and the most useful when creating simple plots that require little coding. This method is most effective when graphing data directly from a DataFrame. The object-oriented method is better suited to more complicated graphs that require more coding, such as those with multiple lines or bars, or multiple plots in one graph.

For now, we'll practice graphing simple plots like line, bar, scatter, and pie charts, and annotating charts. We'll use Matplotlib's plotting methods in the Jupyter Notebook environment with data from the ride-sharing dataset.

By the end of this module, you will be able to create a variety of visualizations. Watch the following video to learn more about the charts you'll be working on.



Check the Version of Matplotlib

Before we get started on any project, it's good practice to make sure we have the latest version of the software we'll be using. Because we'll be using Matplotlib, let's check to make sure we have version 3.1.0 or greater.

Follow the instructions below for your operating system.

Check the Version on the Command Line in macOS

To begin, launch the command line and activate the PythonData environment.

> **Rewind**
> To activate the PythonData environment on the command line, type `conda activate PythonData`.

The command line should look something like this:

```
(PythonData) computer_name:~ home_directory$
```

At the prompt, type `$ python` to launch Python. The command line should look similar to this:

```
(PythonData) computer_name:~ home_directory$ python
Python 3.6.8 |Anaconda, Inc.| (default, Dec 29 2018, 19:04:46)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

At the Python prompt (`>>>`), type `import matplotlib` and press Enter to import Matplotlib.

Next, to check the version of Matplotlib, type `matplotlib.__version__` (there are two underscores before and after “version”) and press Enter:

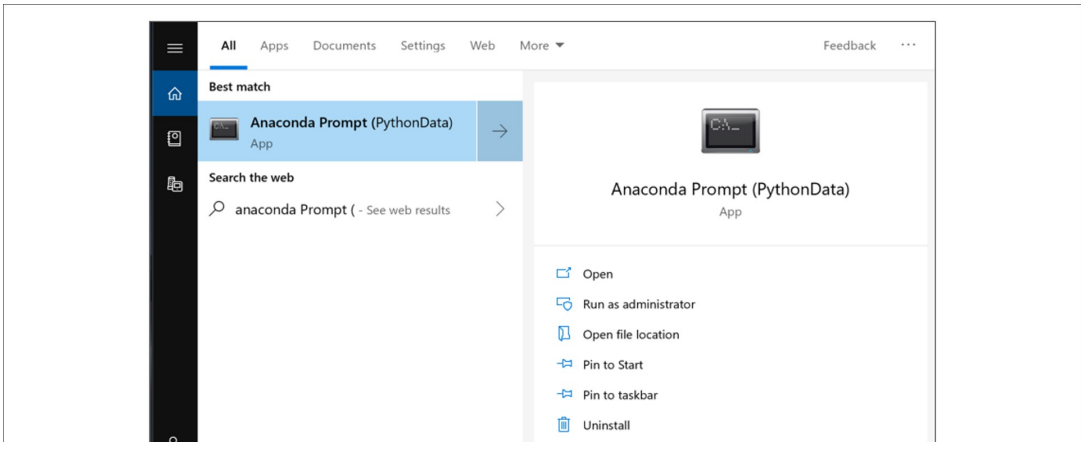
```
>>> matplotlib.__version__
```

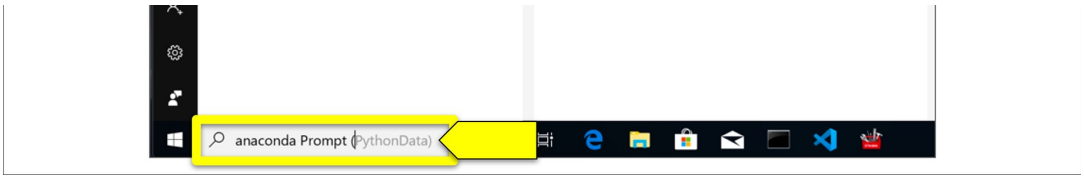
The output should be 3.1.0 or greater.

> **Note**
> To update Matplotlib for your development environment; with your PythonData environment activated, type `conda install -c conda-forge matplotlib` at the command prompt and press Enter.

Check the Version on the Command Line in Windows

In the search menu, type and launch “Anaconda Prompt (PythonData)”.





The Anaconda prompt will look something like this:

```
(PythonData) C:\Users\your_home_directory>
```

After the Python prompt (`>`), type `python` to launch Python. Your Anaconda prompt should look similar to this:

```
(PythonData) C:\Users\your_home_directory>python
Python 3.6.8 |Anaconda, Inc.| ((default, Feb 21 2019, 18:30:04) [MSC v.1916
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Now you'll import the Matplotlib library. At the Python prompt (`>>>`), type `import matplotlib` and press Enter.

Finally, to check the version of Matplotlib, type `matplotlib.__version__` (there are two underscores before and after "version") and press Enter:

```
>>> matplotlib.__version__
```

The output should be 3.1.0 or greater.

> **Note**
> To update Matplotlib for your development environment; with your PythonData environment activated, type `conda install -c conda-forge matplotlib` at the command prompt and press Enter.

Check the Version in Jupyter Notebook

Alternatively, you can check the version of Matplotlib in Jupyter Notebook. To do that, follow these directions

> **Rewind**
>
> To start Jupyter Notebook, navigate to the Class folder on your computer using the command line or Anaconda prompt.
>
> Activate the PythonData environment if it's not activated. Type and run `jupyter notebook`.

In Jupyter Notebook, create a new file if one hasn't been created. Add the following code to a new cell.

```
import matplotlib
matplotlib.__version__
```

When you run the cell, the output should be 3.1.0 or later.

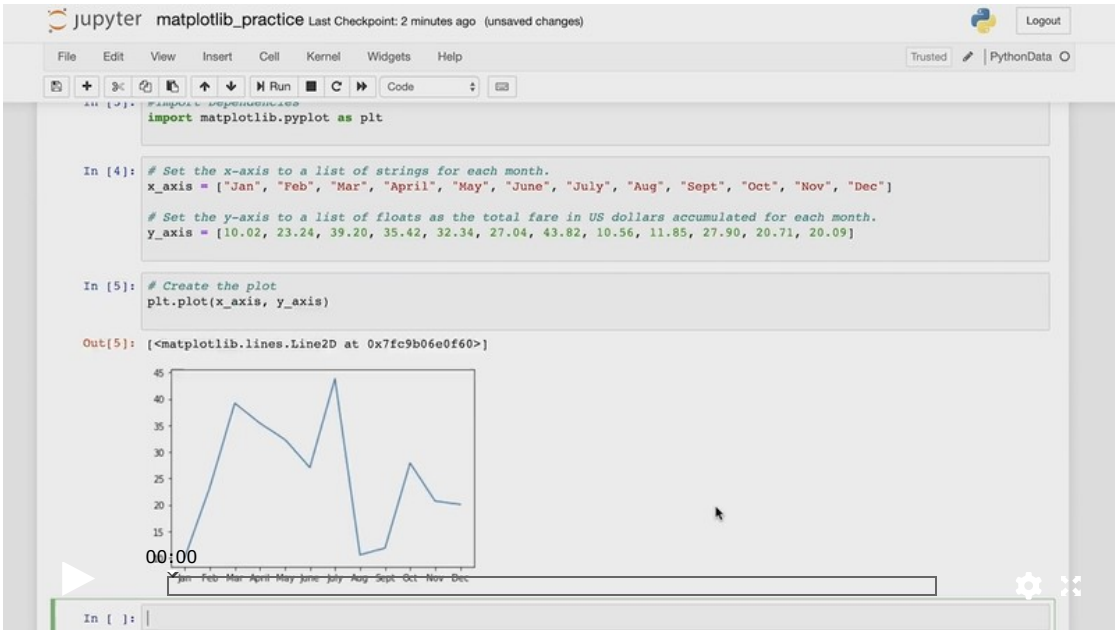
Now that you have an updated version of Matplotlib, let's dive into creating some visualizations!

> **Note**
>
> For more information about the latest Matplotlib version, see the **Matplotlib documentation (https://matplotlib.org/3.1.0/index.html)**.

5.1.3: Create Line Charts

With Matplotlib set up and ready to go, it's time to explore what this program can do! Omar has suggested that you start with one of the most common charts: a line chart.

Omar has also reminded you to comment on your code. This way, if he or the CEO wants to take a look at your repo, they'll know exactly what you're doing and whether you're heading in the right direction.



The line chart is one of the most common charts used to graph data. This type of chart is pretty straightforward: it plots continuous data as points and joins them with a line. Line charts allow you to compare multiple datasets on the same chart rather easily.

Different visualization types serve different purposes. The purpose of a line chart is to display data over time. In this way, a line chart is useful for determining the relationship between time and another value. In contrast, a pie chart, which we'll learn to create later, is not used to show changes over time but to compare parts of a whole.

We'll walk through how to create line charts with Matplotlib using both the MATLAB method and the object-oriented interface method. We'll follow this general approach for every chart we create in this module.

First, launch Jupyter Notebook in the PythonData environment inside the PyBer_Analysis folder.

Next, create a new Jupyter Notebook file and name it `matplotlib_practice`.

Create a Line Chart Using the MATLAB Method

To make sure our plots are displayed inline within the Jupyter Notebook directly below the code cell that produced it, we need to add `%matplotlib inline` to the first cell, as shown here:

```
%matplotlib inline
```

The % sign is sometimes called the **magic command**. When you run a cell that starts with `%`, "magic" happens behind the scenes and sets the back-end processor used for charts. Using it with the `inline` value displays the chart within the cell for a Jupyter Notebook file.

> **Note**
>
> Other values for the magic command are:
>
> - 'GTK3Agg'
> - 'GTK3Cairo'
> - 'MacOSX'
> - 'nbAgg'
> - 'Qt4Agg'

- 'Qt4Cairo'
- 'Qt5Agg'
- 'Qt5Cairo'
- 'TkAgg'
- 'TkCairo'
- 'WebAgg'
- 'WX'
- 'WXAgg'
- 'WXCairo'
- 'Agg'
- 'Cairo'
- 'Pdf'
- 'Pgf'
- 'Ps'
- 'Svg'
- 'template'

For more experienced users, if you're embedding Matplotlib in a web or other application, you might need to use a back-end processor, such as GTK, Qt, Tk, or Wx, that matches the toolkit you're using to build your application.

For more information, see the **Matplotlib documentation on back ends (https://matplotlib.org/3.1.1/api/matplotlib_configuration_api.html)** .

In the next cell, we'll import the `matplotlib` dependency. Then we'll chain the `pyplot` module to it using `matplotlib.pyplot`. This is the same as typing `from matplotlib import pyplot`. Finally, we'll rename `matplotlib.pyplot` as the alias `plt`.

To do this, type the following into the next cell:

```
# Import dependencies.
import matplotlib.pyplot as plt
```

The `pyplot` module will allow us to create all the necessary charts; create a plotting area in a figure; plot lines in a plotting area; and annotate the chart with labels, colors, and other decorative features.

> **Note**
> Pyplot provides a MATLAB-like interface for making plots.

In the next cell, we'll add some ride-sharing data so we have something to graph. The data we'll use at first is just a small portion of the data we'll work with later.

The data is in the form of arrays, or lists. One array will contain the months of the year, and the second will contain the total fares in US dollars for each month.

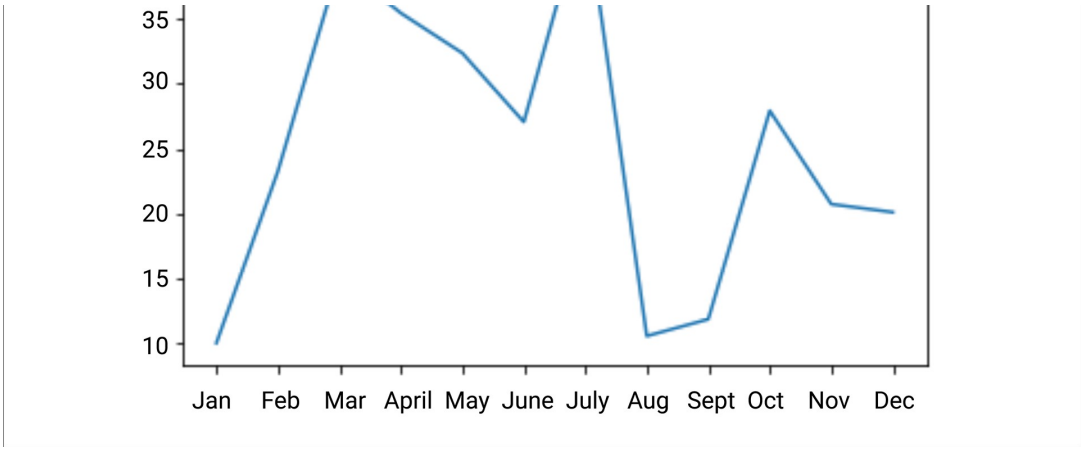
Add the following code to the new cell and run it:

```
# Set the x-axis to a list of strings for each month.
x_axis = ["Jan", "Feb", "Mar", "April", "May", "June", "July", "Aug", "Sept"

# Set the y-axis to a list of floats as the total fare in US dollars accumul
y_axis = [10.02, 23.24, 39.20, 35.42, 32.34, 27.04, 43.82, 10.56, 11.85, 27.
```

In the next cell, chain the `plot()` function to `plt`. Inside the parentheses of the `plot()` function, add the `x_axis` and `y_axis` parameters:

```
# Create the plot
plt.plot(x_axis, y_axis)
```

The purpose of `plt.plot()` is to create the figure with the current axes. When we execute this cell, the first variable, `x_axis`, is the data for the x-axis, and the second variable, `y_axis`, is the data for the y-axis. We have labeled our arrays to make it more convenient for this and other examples. The output in the next cell is a figure with the months on the x-axis and the fare in U.S. dollars on the y-axis, as shown here:



Create a Line Chart Using the Object-Oriented Interface

When we switch from using MATLAB to the object-oriented interface method, we use `ax.plot()` instead of `plt.plot()`. Let's create a simple line graph so we can see the differences between the approaches.

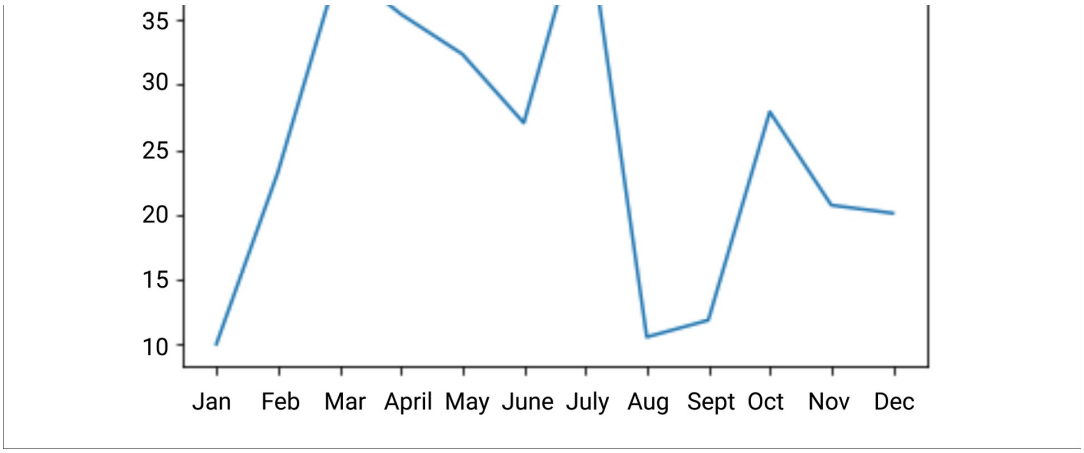
Before we dive in, here's a high-level view of how to create a figure and axes using the object-oriented method:

1. To create a figure, use `fig, ax = plt.subplots()`.
2. To define the axes, use `ax.plot()`.
3. Inside the parentheses of `ax.plot()`, add the axes separated by a comma.

Let's recreate the line chart we just made using the object-oriented interface method. In a new cell, add the following code:

```
# Create the plot with ax.plt()
fig, ax = plt.subplots()
ax.plot(x_axis, y_axis)
```

As you can see in the following chart, when we run this cell, the output is the same as we got when we used the MATLAB method:



Let's break down what the code is doing:

1. The `plt.subplots()` function returns a tuple that contains a figure and axes object(s).
2. `fig, ax = plt.subplots()` unpacks the tuple into the variables `fig` and `ax`.
3. The first variable, `fig`, references the figure.
4. The second variable, `ax`, references the axes of the figure and allows us to annotate the figure.
5. The `ax.plot(x_axis, y_axis)` plots the data from the x and y axes.

The `fig, ax = plt.subplots()` line of code can also be written like this.

```
# Create the plot with ax.plt()
fig = plt.figure()
ax = fig.add_subplot()
```

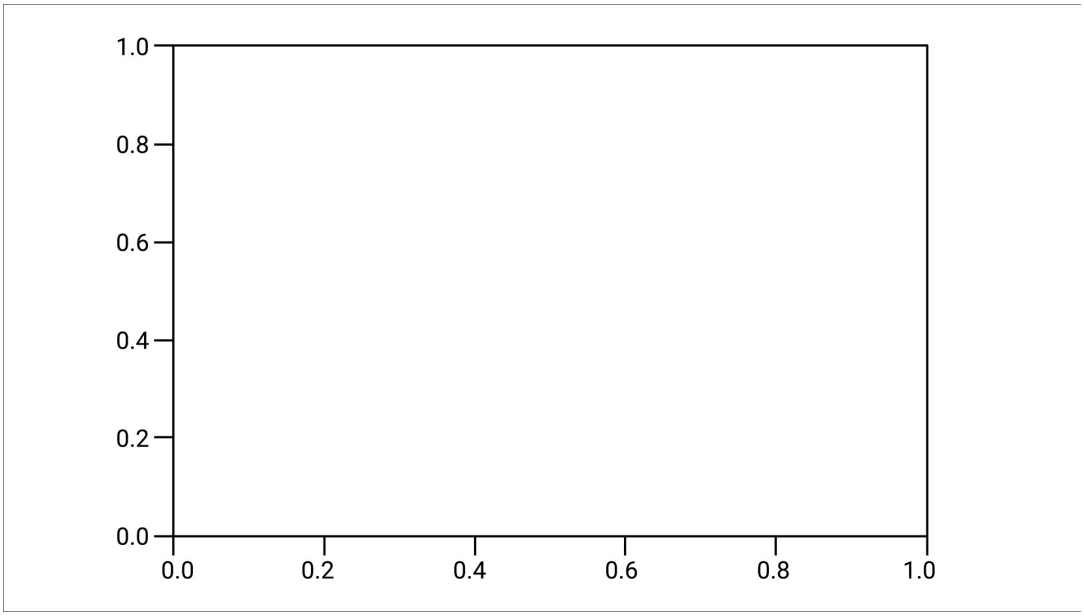
Where `fig` references the figure, and `ax` references the axes of the figure and allows us to annotate the figure.

In a new cell, replace `fig, ax = plt.subplots()` with `fig = plt.figure()` and `ax = fig.add_subplot()`.

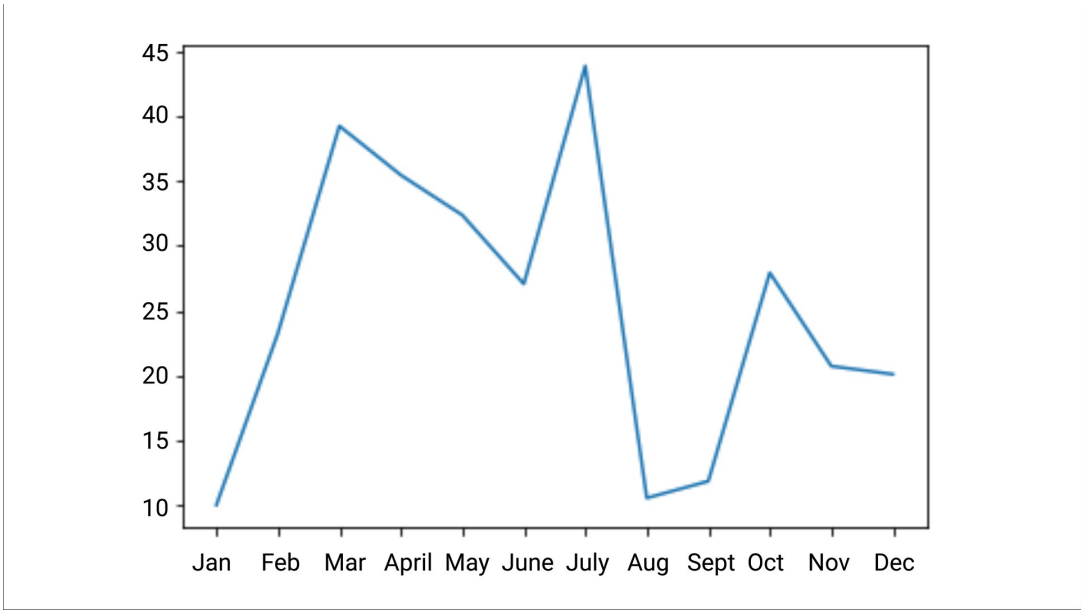
Your cell should look like this:

```
# Create the plot with ax.plt()
fig = plt.figure()
ax = fig.add_subplot()
```

When we execute this cell, we'll see that the result is an empty chart, as shown here:



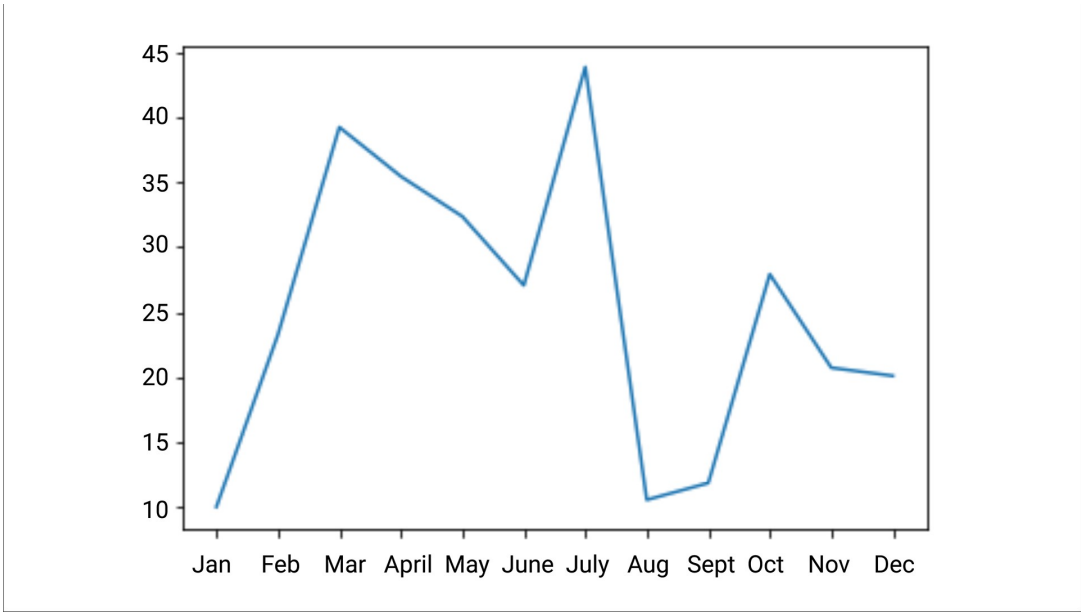
In the same cell, add `ax.plot(x_axis, y_axis)` to the cell and run the cell. The result is the same as using the MATLAB method, as you can see below:



To create a quick graph using the object-oriented interface method, use the following two lines:

```
# Create the plot with ax.plt()
ax = plt.axes()
ax.plot(x_axis, y_axis)
```

When we execute the cell, the result is the same as before, as you can see here:



IMPORTANT!

To change figure-level attributes (such as axis labels, a title, or a legend) or save the figure as an image, use `fig = plt.figure()`.

Use plt.show()

If you search for examples of Matplotlib graphs on the internet, you'll typically find examples that have `plt.show()` at the end of the graphing script.

The `plt.show()` function looks for all the active figure objects and opens a window that displays them if you have two or more sets of data to plot.

This is how we use the `plt.show()` function:

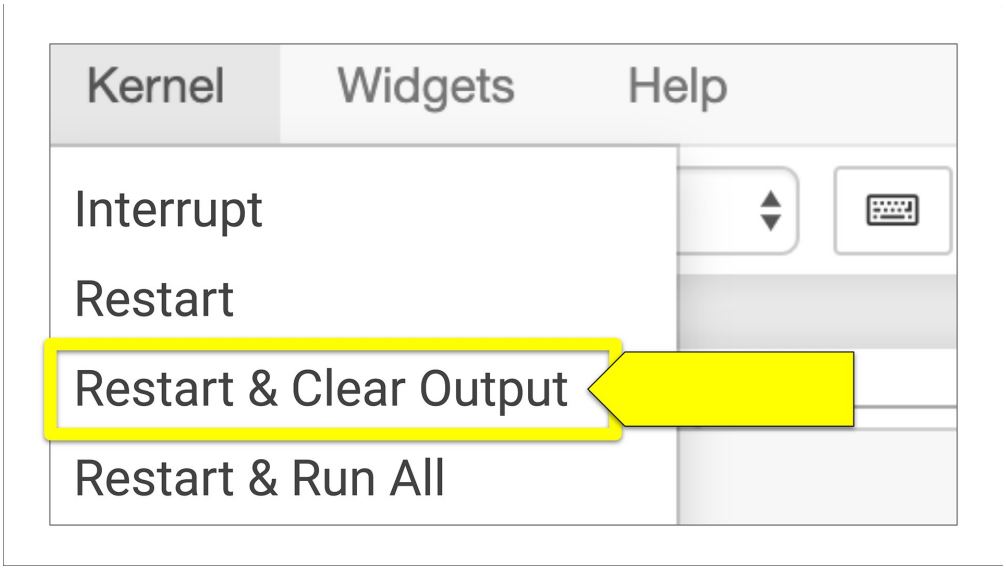
```
# Create the plot.
plt.plot(x_axis, y_axis)
plt.show()
```

IMPORTANT!

`plt.show()` should only be used once per session because it can cause the graph to display improperly. If you have problems with data not being displayed properly, you may have to restart your kernel and clear your output.

Rewind

To restart the kernel and clear the output, you will have to select “Restart & Clear Output” in the Kernel tab on Jupyter Notebook.



5.1.4: Annotate Charts

V. Isualize is famous for asking tough questions during presentations. During your weekly meeting with Omar, he gives you a tip that will make preparing easier and answer some of V. Isualize's questions before she has a chance to ask them! The tip? Annotate all your graphs.

Adding labels to the axes, a title, and a legend helps the viewer understand what they're looking at and helps tell your data story in a clean, clear manner. After all, even the most visually appealing graph isn't much use if we don't know what it's about!

With Matplotlib, we can add labels to the axes, a title, and a legend. We can also change the thickness of the graphed line and add markers for the data points.

Annotate Charts Using the MATLAB Method

Here are a few methods that you can use to annotate charts using the MATLAB method:

Matplotlib Functions	Feature
`plt.figure(figsize=(w, h))`	Change the size of the figure in pixels. Added on the first line of the script.

`plt.plot(x, y, label='line')`	Add a label that will be added to the legend.
`plt.xlim(min, max)`	Set the min and max range of the x-axis.
`plt.ylim(min, max)`	Set the min and max range of the y-axis.
`plt.xlabel('x label')`	Add a label to the x-axis.
`plt.ylabel('y label')`	Add a label to the y-axis.
`plt.title("Title")`	Add a title.
`plt.legend()`	Add a legend.
`plt.grid()`	Add a grid to the chart.
`plt.savefig("add a path and figure extension")`	Save the figure with the given extension. Added at the end of the script.

It is highly recommended to add x-axis and y-axis labels and a title to every graph you create so the viewer knows what it conveys. If you have more than one line or bar on a graph, making each line or bar stand out with a distinct color or line style is helpful, as is adding a legend for each dataset that the lines or bars represent. In addition, thoughtfully setting your x-axis and y-axis ranges can make the data more appealing.

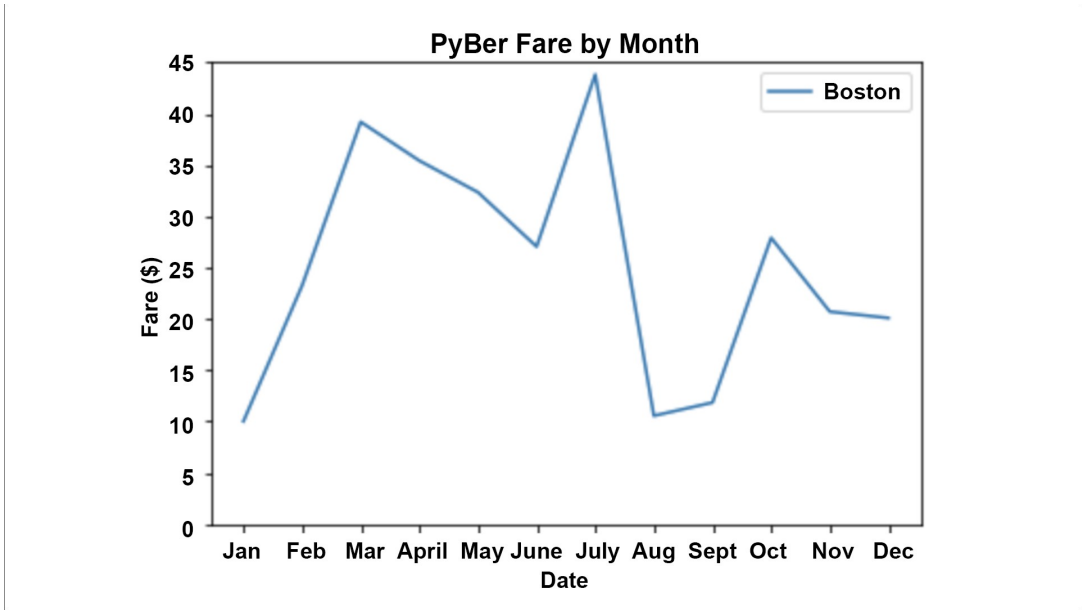
Let's annotate our ride-sharing line chart with some of these features to make it more informative and visually appealing.

Add the following code to a new cell and run the cell:

```
# Create the plot and add a label for the legend.
plt.plot(x_axis, y_axis, label='Boston')
# Create labels for the x and y axes.
```

```
plt.xlabel("Date")
plt.ylabel("Fare($)")
# Set the y limit between 0 and 45.
plt.ylim(0, 45)
# Create a title.
plt.title("PyBer Fare by Month")
# Add the legend.
plt.legend()
```

After running the cell, our graph has more information, as you can see here:



We can also change the width and color of the line and add markers for the data points. This is done by declaring parameters in the `plt.plot()` function.

To declare the parameters for line color, width, and marker, we use `color=`, `width=`, and `marker=`, respectively, inside the `plt.plot()` function.

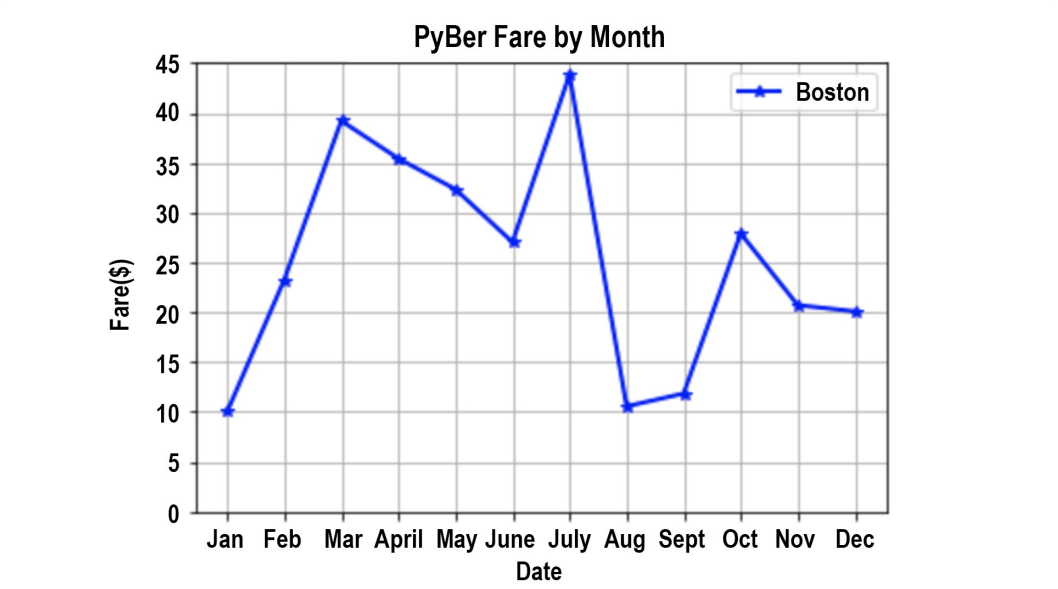
Let's go ahead and amend the code block to add a new line color, modified line width, and a marker for our data points.

Add the following code to a new cell:

```
# Create the plot.
plt.plot(x_axis, y_axis, marker="*", color="blue", linewidth=2, label='Bosto
# Create labels for the x and y axes.
plt.xlabel("Date")
```

```
plt.ylabel("Fare($)")
# Set the y limit between 0 and 45.
plt.ylim(0, 45)
# Create a title.
plt.title("PyBer Fare by Month")
# Add a grid.
plt.grid()
# Add the legend.
plt.legend()
```

After running the cell, our chart looks like this:



How would you use `xlim()` to show the months January through June on the graph? (Select all that apply.)

- [ ] `plt.xlim("Jan", "June")`
- [ ] `plt.xlim(Jan, June)`
- [ ] `plt.xlim(0, 5)`
- [ ] `ax.xlim("Jan", "June")`

Check Answer

Finish ▸

Note

For more information about how to create graphs using the MATLAB method, see the following documentation:

[Matplotlib documentation on matplotlib.pyplot.plot (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.plot.html)]

[Matplotlib documentation on the Pyplot API (https://matplotlib.org/3.1.0/api/index.html#the-pyplot-api)]

[Matplotlib documentation on the anatomy of a Matplotlib chart (https://matplotlib.org/tutorials/introductory/usage.html#parts-of-a-figure)]

Annotate Charts Using the Object-Oriented Interface

Annotating graphs using the object-oriented interface is similar to using the MATLAB approach, with a slightly different syntax.

Here are a few methods you can use to annotate your chart using the object-oriented interface method:

Matplotlib Object-Oriented Functions	Feature
`fig, ax = plt.subplots(figsize=(w, h))`	Change the size of the figure in pixels. Add this in the `subplots()` function.
`ax.plot(x, y, label='line')`	Add a label that will be added to the legend.
`ax.set_xlim(min, max)`	

	Sets the min and max range of the x-axis.
`ax.set_ylim(min, max)`	Sets the min and max range of the y-axis.
`ax.set_xlabel('x label')`	Add a label to the x-axis.
`ax.set_ylabel('y label')`	Add a label to the y-axis.
`ax.set_title("Title")`	Add a title.
`ax.legend()`	Add a legend.
`ax.grid()`	Add a grid to the chart.
`** plt.savefig("add a path and figure extension")`	Saves the figure with the given extension. Added at the end of your script.

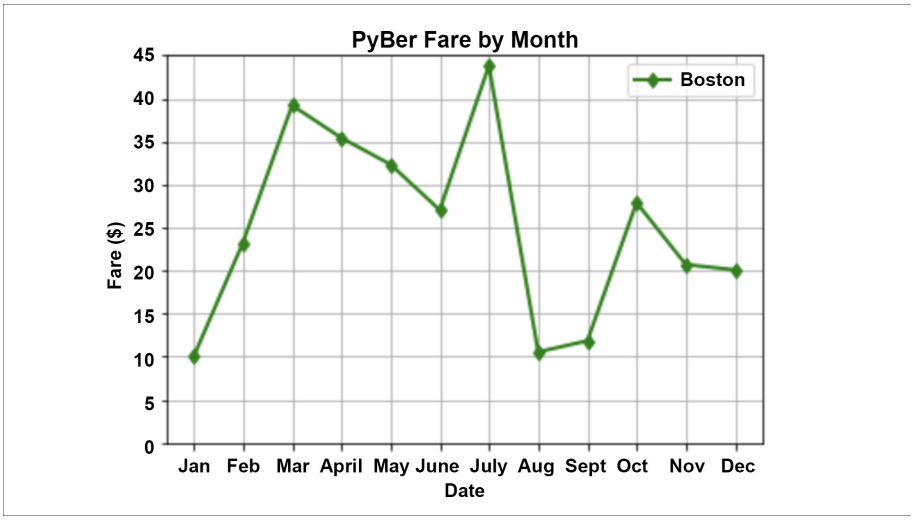
Now test out your skills with the following Skill Drill.

Skill Drill

Using the object-oriented approach, make the following changes to your line chart:

1. Change the line color to green.
2. Change the marker to a diamond.
3. Change the line width to 2 points.
4. Add a legend for the city of Boston.
5. Add a title and axes labels.
6. Add a y-axis limit.
7. Add grid lines.

When you're done, your chart should look similar to this:



Note

For more information on how to create charts using the object-oriented interface method, please refer to the following documentation:

Matplotlib documentation on matplotlib.axes.Axes.plot
(https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.plot.html)

Matplotlib documentation on the Axes class
(https://matplotlib.org/3.1.0/api/axes_api.html)

Great job on creating your first line chart! Knowing how to create a line chart is an essential skill to have for creating quick visualizations of your data.

Let's build on this and work on making our first bar chart, another essential graph to have in your data analysis tool belt.

5.1.5: Create Bar Charts Using the MATLAB Approach

Great job! You've completed the first graph in any data analyst's tool belt: the line chart. And you created it using two different methods!

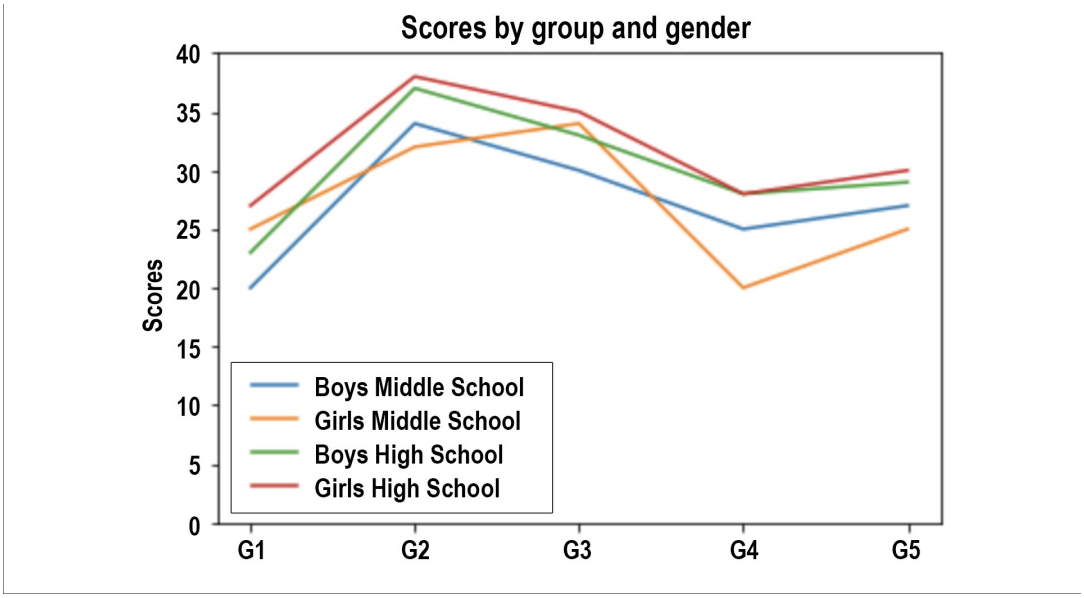
Of course, this was just step one—and you're not going to stop at step one! Both Omar and V. Isualize will want to see visualizations that really illuminate the data.

Next, you'll create and annotate vertical and horizontal bar charts. Bar charts are particularly helpful when comparing datasets over time or when your line chart starts to look cluttered because it has too many lines. So stretch, refill your coffee, and get back to coding!

A bar chart tells a different visual story than a line chart. There are many benefits to using a bar chart. They're good at displaying discrete data (i.e., data based on counts) in distinct columns. They also tend to be visually strong. For example, adding color to the bars can make a chart really shine for your audience.

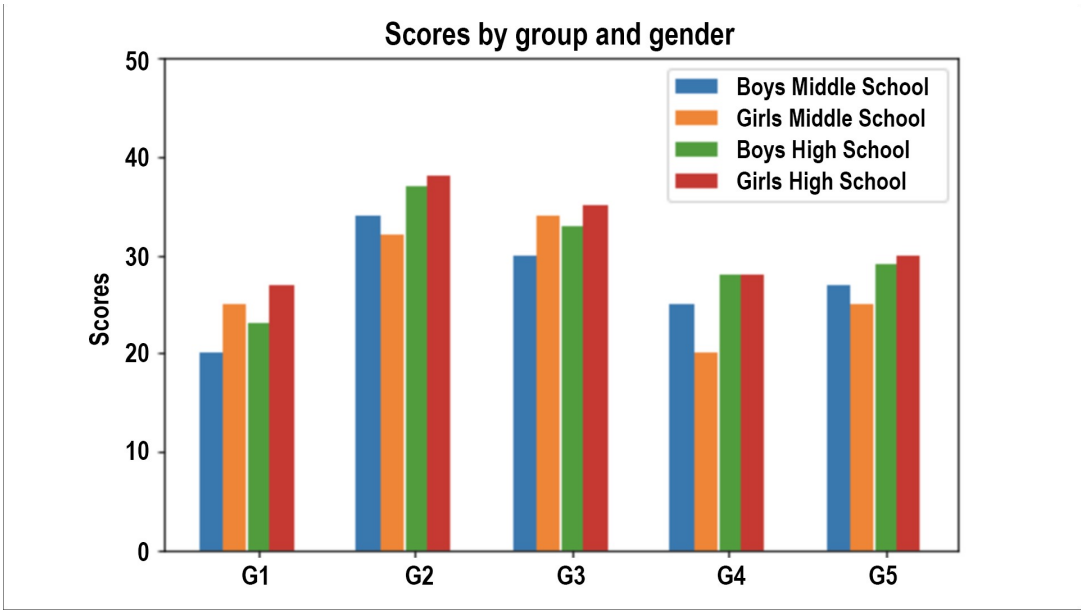
You can use a bar chart to clearly compare two or more datasets, either by displaying them side by side or by using a stacked bar chart. Bar charts can provide more visual appeal and information compared to having multiple lines on the same graph, which can mask the underlying information.

For instance, consider the following line chart, which has lines for four different groups:



This chart might be good for a quick glance at the data, but it's not one that you'd want to show stakeholders. The lines are too close together, two lines are almost on top of each other, and it's hard to tell which group is which from the line graph color. You could make the line thickness darker, but some lines might overlap in that case.

A better way to show this data would be a bar chart, where we can clearly see the differences among the groups and the bar color of each group:



Let's dive into learning how to create bar charts with Matplotlib using the MATLAB method. We'll use the same ride-sharing data as before to create a bar chart. Here's that data:

```
# Set the x-axis to a list of strings for each month.
x_axis = ["Jan", "Feb", "Mar", "April", "May", "June", "July", "Aug", "Sept"

# Set the y-axis to a list of floats as the total fare in US dollars accumul
y_axis = [10.02, 23.24, 39.20, 35.42, 32.34, 27.04, 43.82, 10.56, 11.85, 27.
```

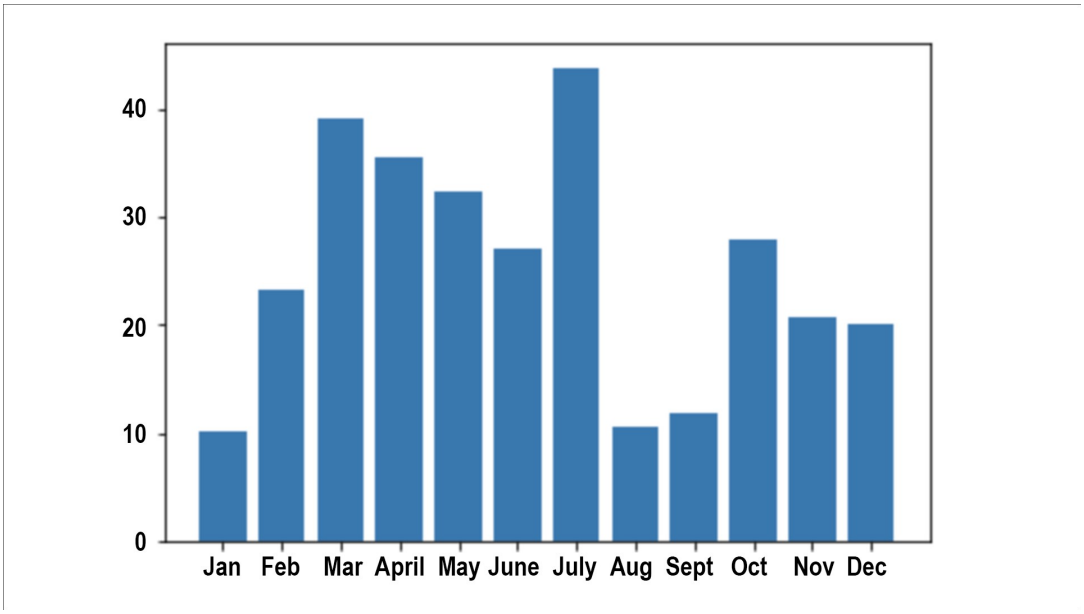
Create a Vertical Bar Chart

If you're graphing time-series data or ordinal values (e.g., age ranges, salary ranges, or groups), a vertical bar chart is a good choice. Let's learn how to do that.

To create a vertical bar chart, use the `plt.bar()` function. In the `matplotlib_practice.ipynb` file, add the following x and y data arguments inside parentheses:

```
# Create the plot
plt.bar(x_axis, y_axis)
```

When you run this cell, you'll see the ride-sharing data as a bar graph, as shown here:

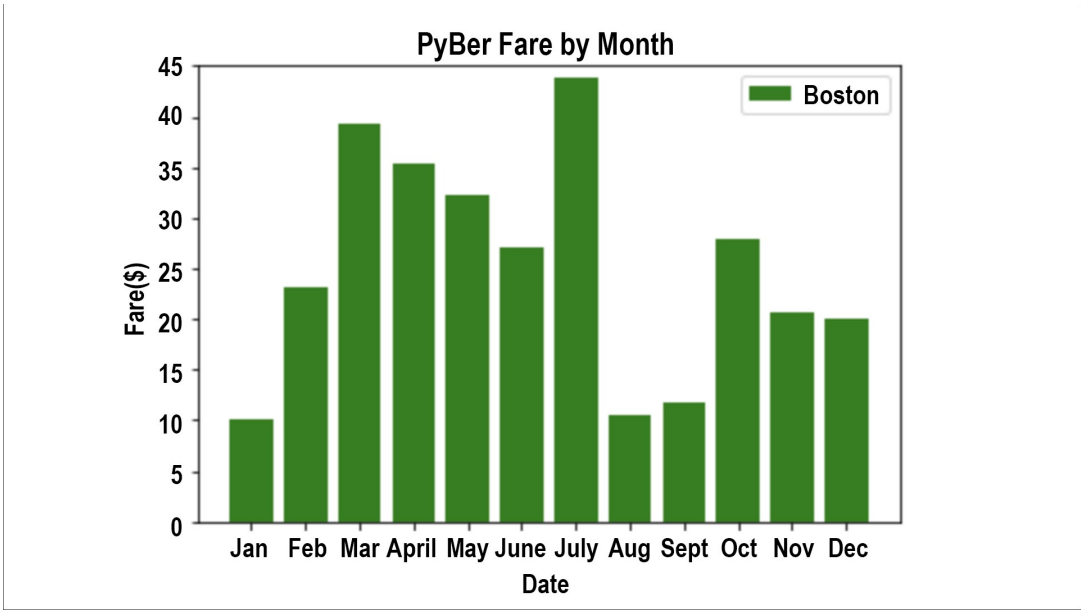


We can annotate the bar chart like we annotated the line chart. However, we won't need the `marker` or `linewidth` attributes.

To annotate the bar chart, add the following code to a new cell:

```
# Create the plot.
plt.bar(x_axis, y_axis, color="green", label='Boston')
# Create labels for the x and y axes.
plt.xlabel("Date")
plt.ylabel("Fare($)")
# Create a title.
plt.title("PyBer Fare by Month")
# Add the legend.
plt.legend()
```

After running the cell, our graph should look like this:



Note

For more information, see the [Matplotlib documentation on creating bar charts using the MATLAB method (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar)](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.bar.html#matplotlib.pyplot.bar).

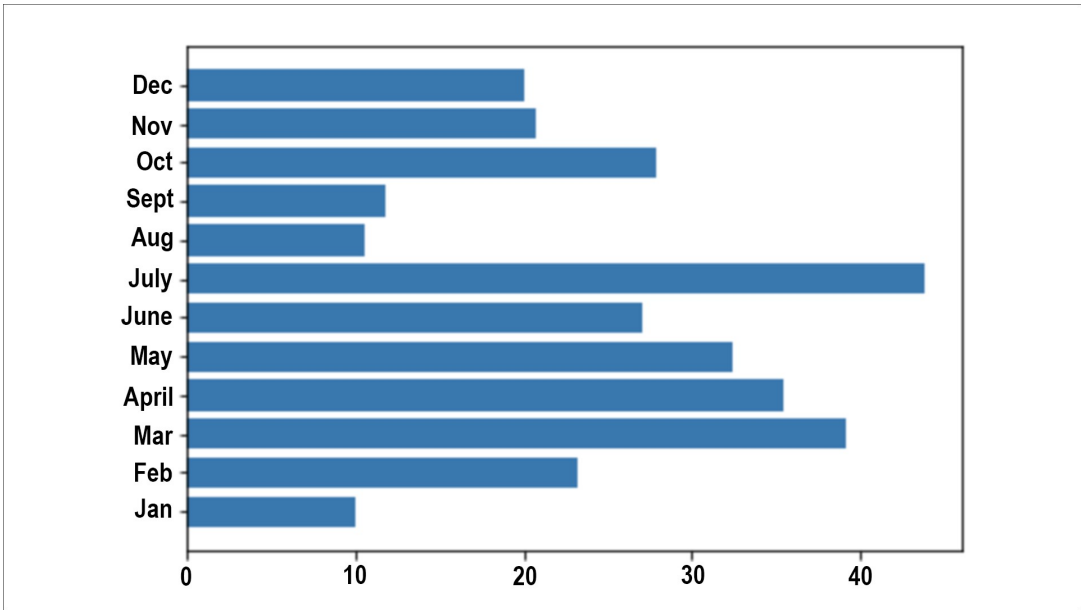
Create a Horizontal Bar Chart

If you're graphing nominal variables (e.g., favorite music groups among teens, number of votes by county or state) where you sort the data from greatest to least or least to greatest, it's best to use a horizontal bar chart.

To create a horizontal chart, we use the `plt.barh()` function:

```
# Create the plot
plt.barh(x_axis, y_axis)
```

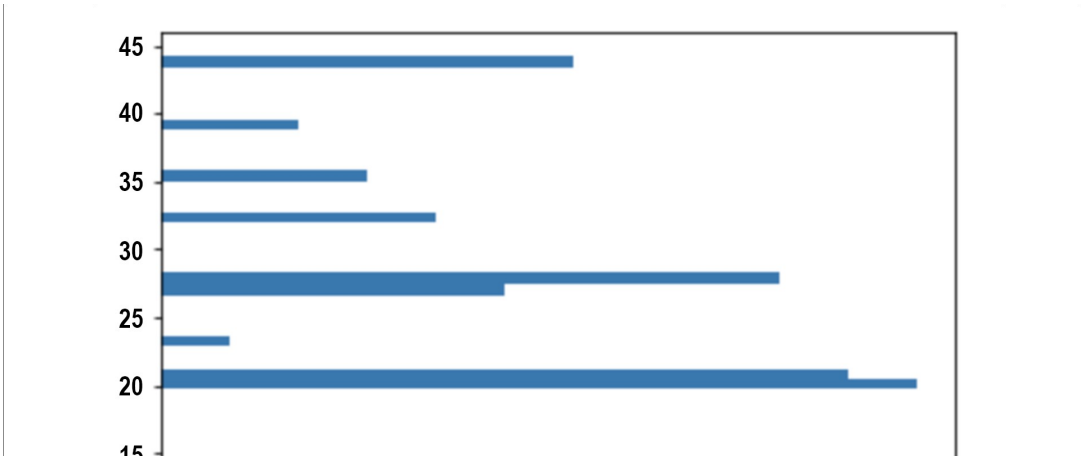
When we run this cell, our bar chart looks like this:

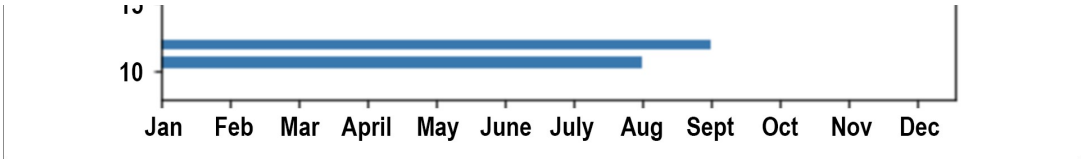


If you want the data on opposite axes, switch the arguments in the `barh()` function and run the cell again:

```
# Create the plot
plt.barh(y_axis, x_axis)
```

When we run this cell, our bar chart looks like this:





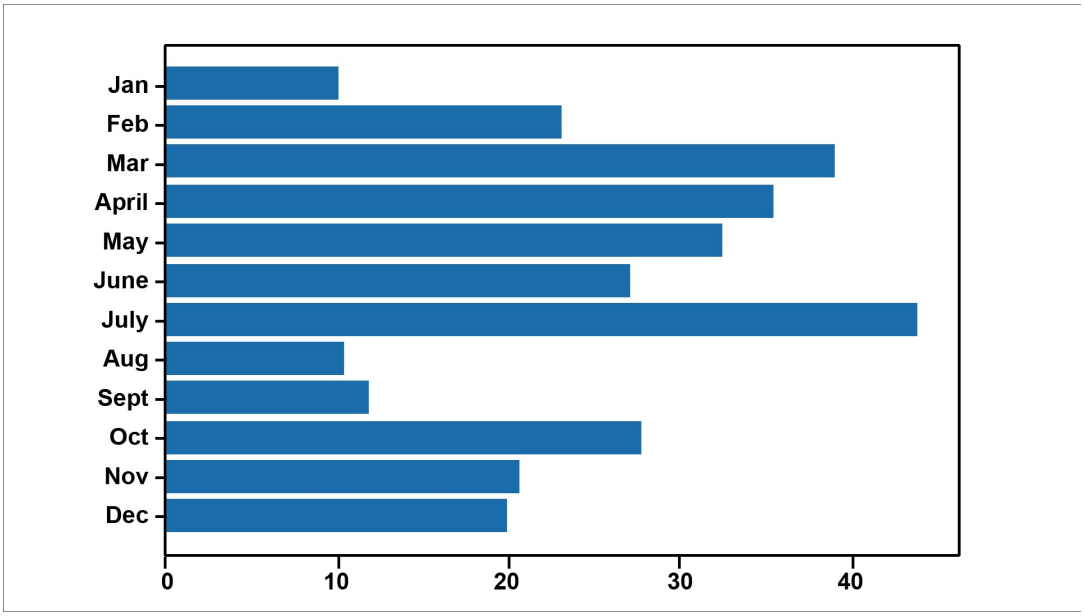
The horizontal bar chart is not a good way to represent the data because the bar should convey an *amount* on the x-axis—not a date. Trying to determine each month's average fare is more challenging in this chart compared to the previous horizontal chart, but it's good to know that we can switch data to graph on the opposite axes if we need to.

We should invert the y-axis of the previous chart to have "January" at the top and "December" at the bottom. Can you think of the reason for this? Right—because we don't want to show the CEO ride-sharing data with the months in the wrong order!

To invert the y-axis to have the months in ascending order, use the `gca()` method. The `gca()` method means "get current axes." We can chain the `gca()` method to the `invert_yaxis()` method by using `gca().invert_yaxis()`, as shown here:

```
# Create the plot.
plt.barh(x_axis, y_axis)
plt.gca().invert_yaxis()
```

When we run this cell, our bar chart should look like this:



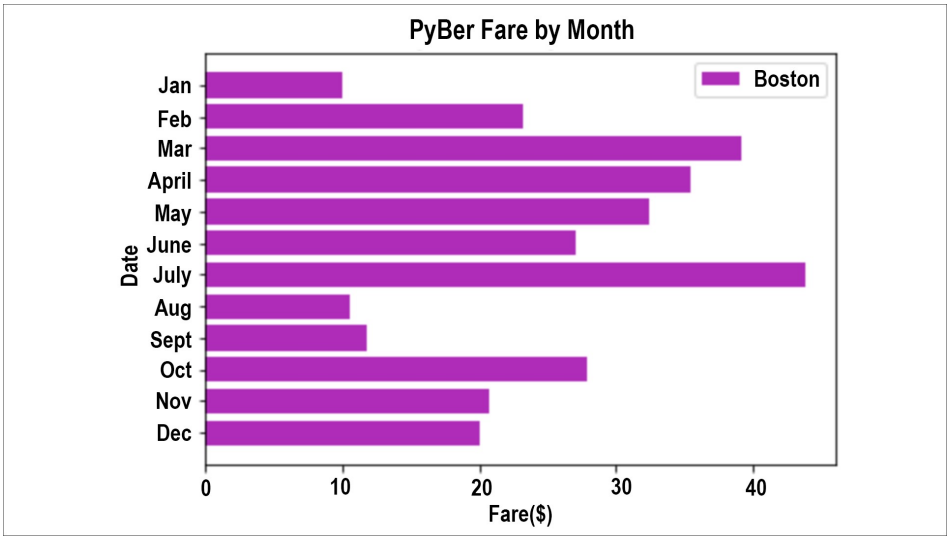
We can also annotate a horizontal bar chart as we did with a vertical bar chart.

Skill Drill

Using the Matplotlib MATLAB plotting approach, make the following changes:

1. Change the bar colors to magenta.
2. Add a legend for the city of Boston.
3. Add a title and axes labels.
4. Invert the y-axis so that January is at the top.

When you're done, your chart should look similar to this:



Note

For more information, see the [Matplotlib documentation on creating horizontal bar charts using the MATLAB method (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh)](https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh).

5.1.6: Create Bar Charts Using the Object-Oriented Approach

Now that you've made your first bar chart, you're starting to feel like a bit of a pro—as you should be! Bar charts are standard at organizations around the globe, and being able to make clearly annotated vertical and horizontal bar charts is a critical skill for any data visualizer.

Of course, V. Isualize has been working in the field for more than a decade, so you and Omar need to be ready in case she asks how to do this using the object-oriented approach. So let's learn how to do that now!

Similar to the line charts that we created using the object-oriented method, bar charts use the same formats. We'll use the same ride-sharing data, shown here:

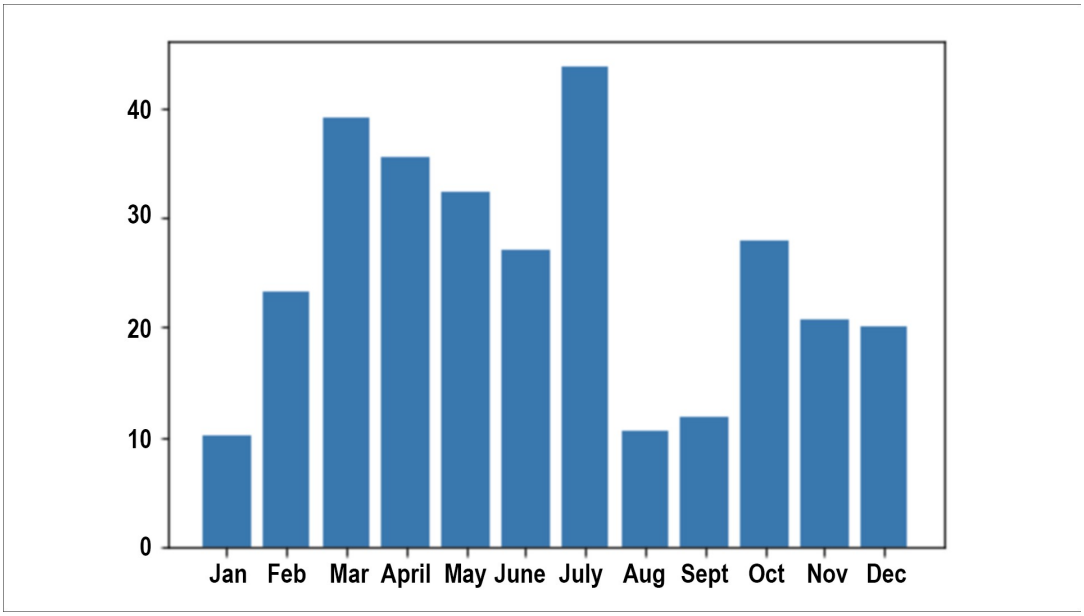
```
# Set the x-axis to a list of strings for each month.
x_axis = ["Jan", "Feb", "Mar", "April", "May", "June", "July", "Aug", "Sept"

# Set the y-axis to a list of floats as the total fare in US dollars accumul
y_axis = [10.02, 23.24, 39.20, 35.42, 32.34, 27.04, 43.82, 10.56, 11.85, 27.
```

Create a Vertical Bar Chart

To create a bar chart using the object-oriented interface method, use the `ax.bar()` function and add the x and y data parameters inside the parentheses. To do this, add the following code in a new cell and run the cell:

```
# Create the plot with ax.plt()
fig, ax = plt.subplots()
ax.bar(x_axis, y_axis)
```

When we run this cell, the output is the same as using the MATLAB method, as you can see here:



Using the code we used to create the previous bar graph, how would you change the bar color to black with a legend for the city of New York using the object-oriented interface method? (Select all that apply.)

- [ ] `plt.bar(x_axis, y_axis, color="black", label='New York')`
- [ ]
```
ax.set_color("black")
ax.set_label('New York')
```
- [ ] `ax.bar(x_axis, y_axis, color="k", label='New York')`
- [ ] `ax.bar(x_axis, y_axis, color="black", label='New York')`

Check Answer

Finish ▸

Note

> For more information, see the [Matplotlib documentation on creating bar charts using the object-oriented interface (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.bar.html)](https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.bar.html) .

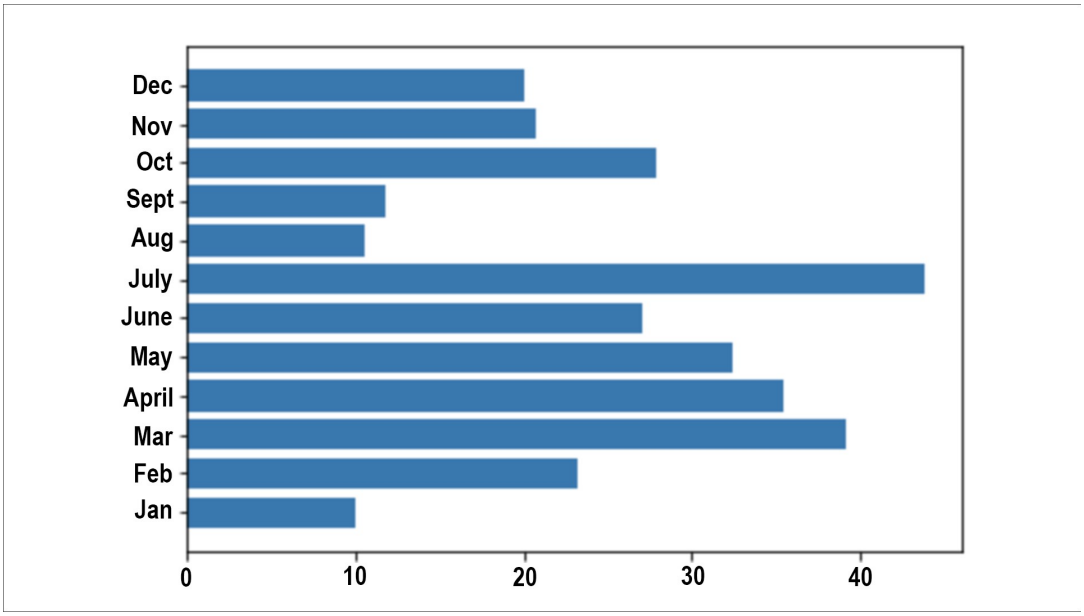
Create a Horizontal Bar Chart

To create a horizontal bar chart using the object-oriented interface method, use the `ax.barh()` function.

Add the following code in a new cell and run the cell:

```
# Create the plot with ax.plt()
fig, ax = plt.subplots()
ax.barh(x_axis, y_axis)
```

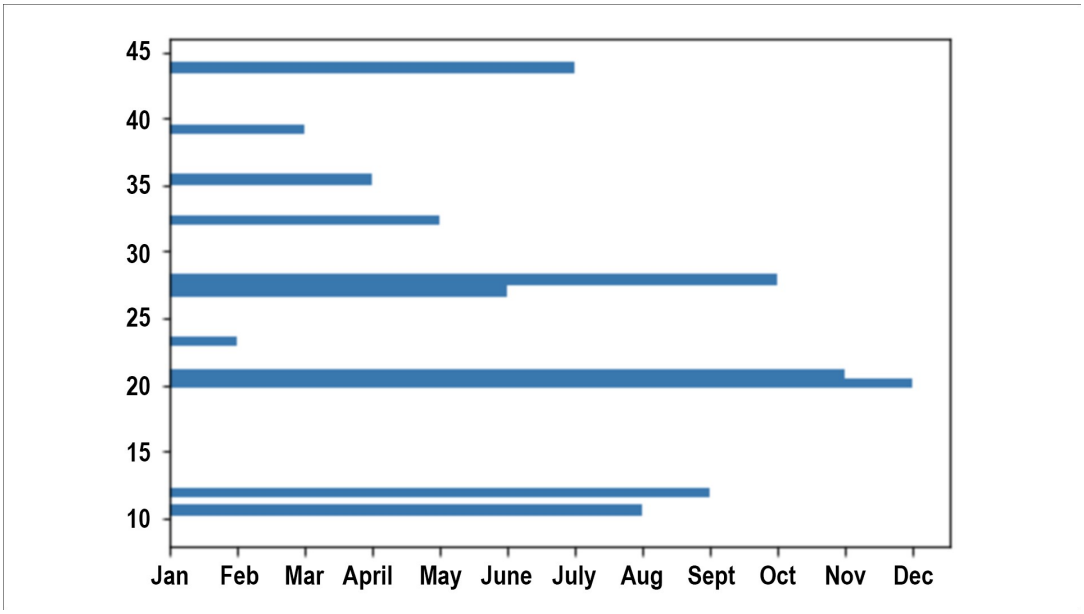
When we run this cell, our bar chart looks like this:



If we want the data on opposite axes, we need to switch the arguments in the `barh()` function, as shown here:

```
# Create the plot with ax.plt()
fig, ax = plt.subplots()
ax.barh(y_axis, x_axis)
```

When we run this cell, our bar chart looks like this.

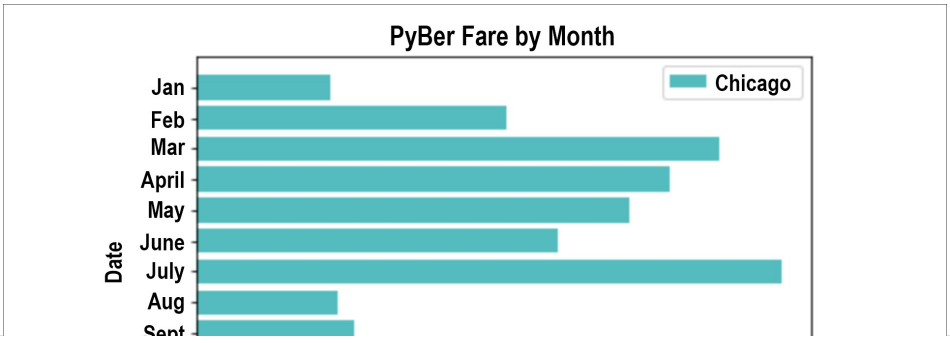


Now test your skills in the following Skill Drill.

Skill Drill

Using the object-oriented approach, make the following changes:

1. Change the bar color to cyan.
2. Add a legend for the city of Chicago.
3. Add a title and axes labels.
4. Switch the axis so that the Fare($) data is on the x-axis.
5. Invert the y-axis data.



Note

For more information, see the [Matplotlib documentation on creating horizontal bar charts using the object-oriented interface method (https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/barh.html#sphx-glr-gallery-lines-bars-and-markers-barh-py)](https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/barh.html#sphx-glr-gallery-lines-bars-and-markers-barh-py) .

Great job on creating bar charts! Next we'll learn how to create scatter plots and bubble charts.

5.1.7: Create Scatter Plots and Bubble Charts

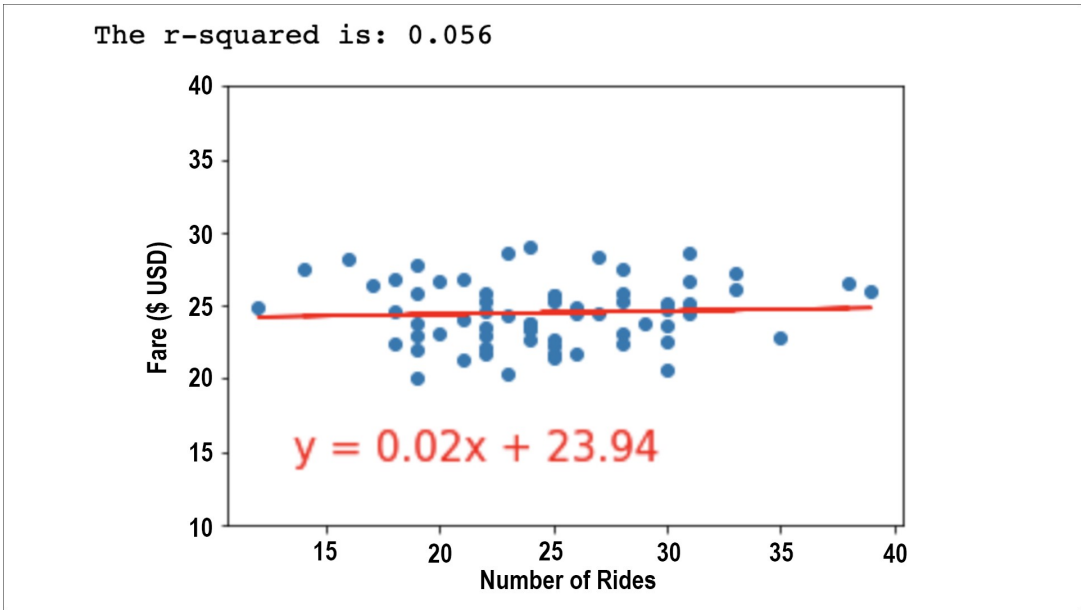
You've finished your bar charts, so you take a break and grab some lunch with a few other analysts on your team. While you're waiting for the food to arrive, you start talking with one of them, Sasha, and discover that she gave a presentation to V. Isualize and the executive team just last week!

When you tell Sasha that you're prepping some visualizations for V. Isualize on the rideshare data from 2019, she offers you a word of advice: be sure to include scatter plots! Sasha confides that she didn't think to include these in her presentation and V. Isualize seemed a bit disappointed, especially because scatter plots are particularly good at depicting multiple datasets on the same chart. Because Sasha just finished putting together some scatter plots as a follow-up to her own presentation, she offers to work with you after lunch to make sure you have some ready to go when you walk into the conference room to give your presentation.

You get your lunch to go, move a few meetings around, and head back to the office with Sasha ready to get to work on scatter plots. Thanks to her help, nothing in this presentation is going to catch you by surprise!

Scatter plots have many benefits over line and bar charts. A scatter plot visually depicts a positive, negative, or no-correlation relationship between two factors.

In the following scatter plot, we determine that there is no correlation between the number of rides and the average fare, because the *r* value is positive and close to zero. This is called a **regression analysis**. We'll walk through how to perform regression analysis in the next module.



Let's dive in to creating scatter plots. As we've done with other charts, we'll use the MATLAB method first and then the object-oriented interface.

Create a Scatter Plot Using the MATLAB Method

There are two ways to create scatter plots using the MATLAB approach:

- `plt.plot()`
- `plt.scatter()`

We'll learn how to use both of these, starting with with `plt.plot()`. We'll use the same ride-sharing data, shown here:

```
# Set the x-axis to a list of strings for each month.
x_axis = ["Jan", "Feb", "Mar", "April", "May", "June", "July", "Aug", "Sept"
```

```
# Set the y-axis to a list of floats as the total fare in US dollars accumul
y_axis = [10.02, 23.24, 39.20, 35.42, 32.34, 27.04, 43.82, 10.56, 11.85, 27.
```

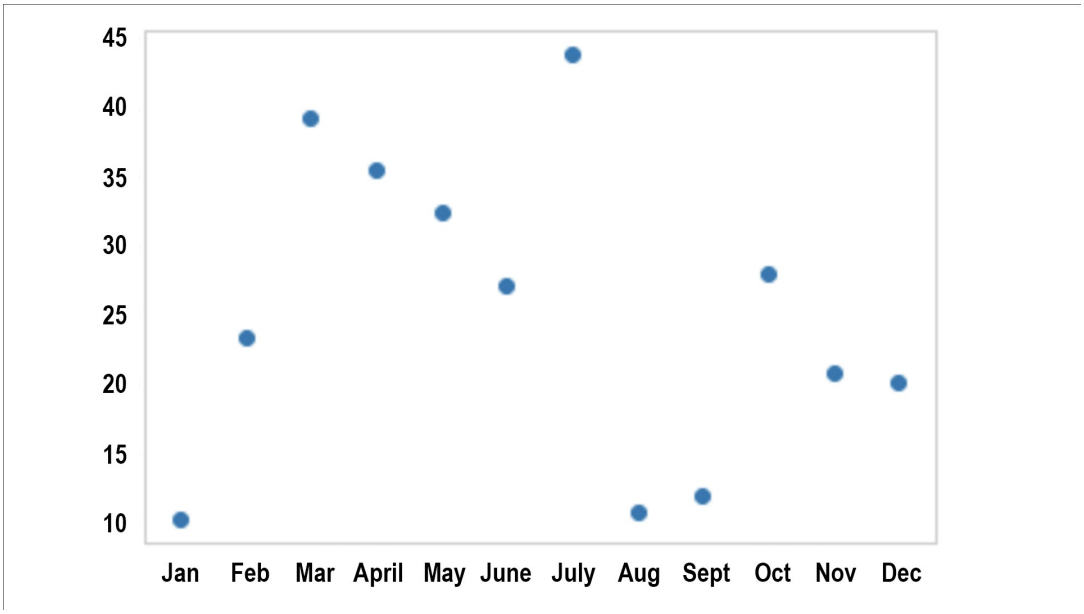
If we were only to use the `plt.plot()` function with our x-axis and y-axis, it would create a line plot and that's not what we're looking for. So how do we make it a scatter plot?

When we use the `plt.plot()` function to create a scatter plot, we need to add a lowercase "o" as a parameter inside the parentheses. This switches the plot from a line chart to a scatter plot.

Add the following code to your `matplotlib_practice.ipynb` Jupyter Notebook file:

```
plt.plot(x_axis, y_axis, 'o')
```

When you run this code, you should see the following scatter plot:



To create a scatter plot using the `plt.scatter()` function, add the x-axis and y-axis parameters inside the parentheses, as shown here:

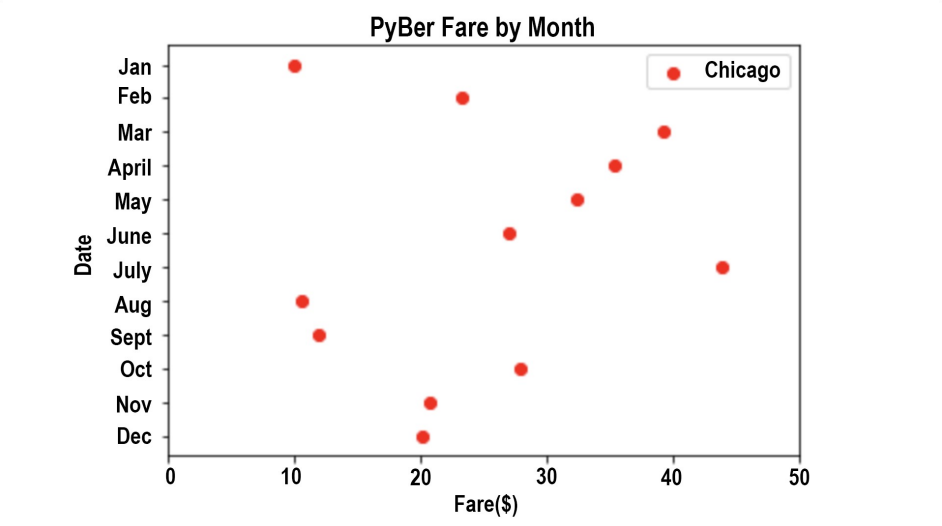
```
plt.scatter(x_axis, y_axis)
```

When we execute the code, the result should be identical to the graph when we used `plt.plot(x_axis, y_axis, 'o')`.

Skill Drill

Using the Matplotlib MATLAB plotting approach, make the following changes to your scatter plot:

1. Change the color of the markers to red.
2. Add a legend for the city of Chicago.
3. Add a title and axes labels.
4. Switch the axis so that the Fare($) data is on the x-axis.
5. Add a limit for the x-axis data.
6. Invert the y-axis so that January is at the top.



Create a Bubble Chart Using the MATLAB Method

Bubble charts are useful when presenting financial, population, and weather data because you can add a third and fourth factor to convey more information.

The first two factors are the x- and y-axes data, which we have been using quite frequently. By changing the “dot” into a “bubble,” we are adding a third factor: size. If

there is more than one dataset that uses the same axes, we can change the color of each marker for each dataset, which will add a fourth factor: color.

Let's see how to create a bubble chart by making a simple modification to the existing scatter plot code.

In a new cell, add the following code:

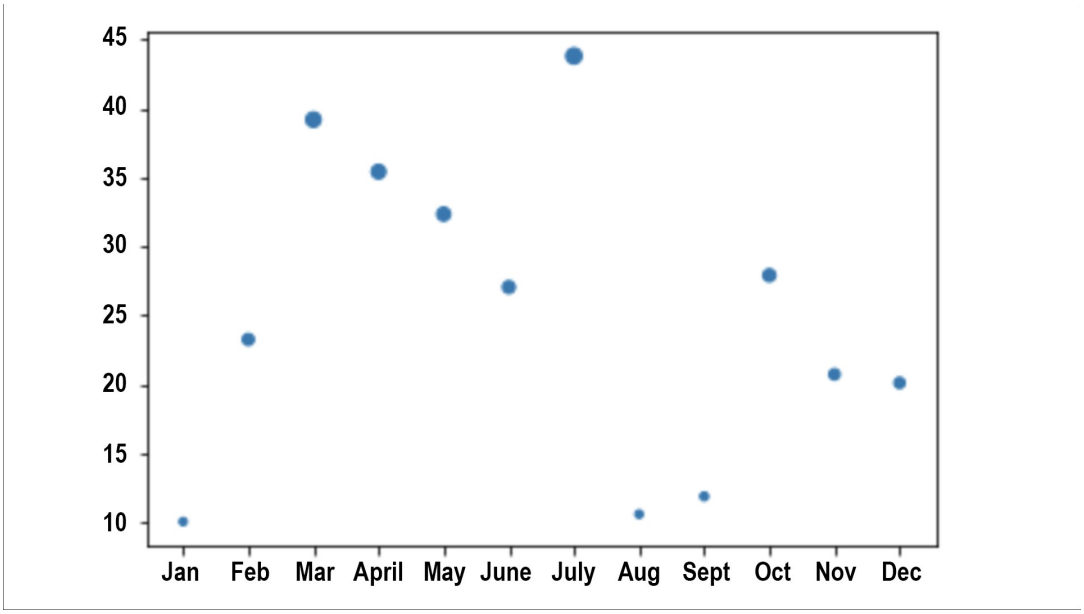
```
plt.scatter(x_axis, y_axis)
```

Next, to change the size of our marker on the graph, we have to use the `s`, or size parameter, for the marker. The `s` parameter must be equal to a feature of our data. We will make the size of each marker equal to the data floating-point decimal values of the y-axis by using `s=y_axis`.

Go ahead and edit the `plt.scatter()` function so that it looks like this:

```
plt.scatter(x_axis, y_axis, s=y_axis)
```

After running this cell, the marker size has changed based on the value of the fare in USD, as you can see here:



Some of the markers look very small. Let's make them bigger so that the chart is more legible and impactful.

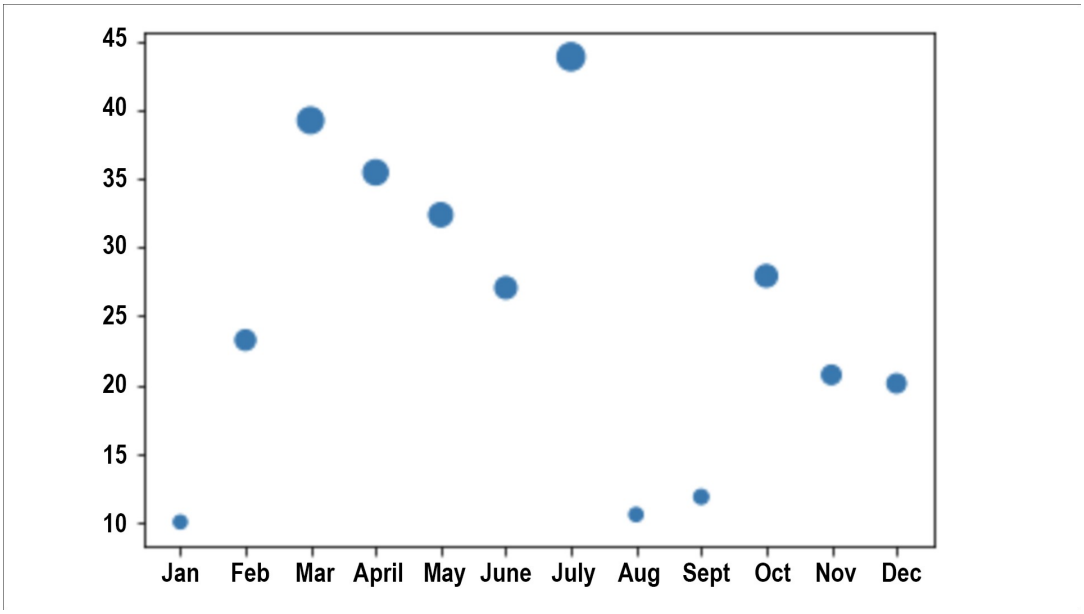
We can adjust the size by multiplying the data in the y-axis by any value. Let's multiply each data point in the y-axis by 3 and see what happens. To do this, we can iterate through the y-axis data and multiply each data point by 3 and add it to a new list, like this:

```
y_axis_larger = []
for data in y_axis:
  y_axis_larger.append(data*3)
```

Then we can use the new y-axis list for the s parameter:

```
plt.scatter(x_axis, y_axis, s=y_axis_larger)
```

After running this cell, the marker size has increased in size, as you can see here:



That's a much better chart to present to the CEO, isn't it?

We can refactor the code we used to create a scatter plot above, but instead of using a for loop to create the list `y_axis_larger` for the size parameter of the marker, we can use Python's list comprehension technique inside the `plt.scatter()` function. You can use **list comprehension** to replace many `for` and `while` loops. List comprehension is faster because it is optimized for Python to spot a predictable pattern during looping.

The format for list comprehension is as follows.

```
new_list = [expression for item in list if conditional]
```

Using the list comprehension technique to multiply each data point in the `y-axis` list, we would do the following in the `plt.scatter()` function:

```
plt.scatter(x_axis, y_axis, s = [i * 3 for i in y_axis])
```

When we run this line of code, the output will be the same as before.

> **Note**
> For more information about creating scatter plots and bubble charts using the MATLAB method, as well as using list comprehenson, refer to the following documentation:
>
> **Matplotlib documentation on matplotlib.pyplot.scatter (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter)**
>
> **Matplotlib documentation on a simple scatter plot (https://matplotlib.org/3.1.1/gallery/shapes_and_collections/scatter.html#sphx-glr-gallery-shapes-and-collections-scatter-py)**
>
> **Matplotlib documentation on list comprehension (https://docs.python.org/3.6/tutorial/datastructures.html#list-comprehensions)**

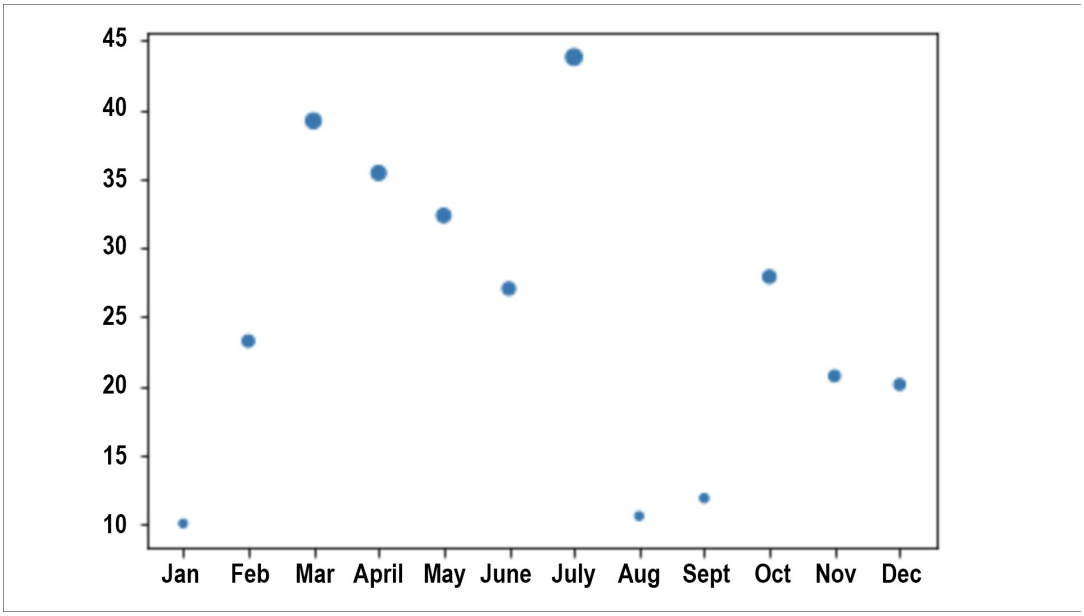
Create a Scatter Plot Using the Object-Oriented Interface

Now that we've created a scatter plot using the MATLAB approach, let's make one using the object-oriented approach.

In a new cell, add the following code:

```
fig, ax = plt.subplots()
ax.scatter(x_axis, y_axis)
```

When you run that cell, here's what your scatter plot will look like:



Great job! Now let's create a bubble chart.

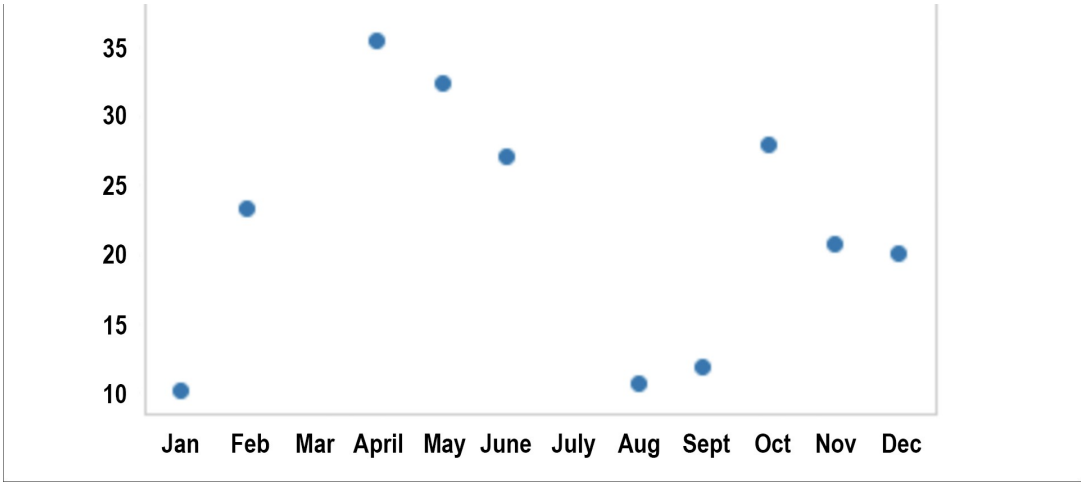
Create a Bubble Chart Using the Object-Oriented Interface

To create a bubble chart using the object-oriented interface approach, we'll add the `s` parameter just like we did when we used the MATLAB approach.

In a new cell, add the following code:

```
fig, ax = plt.subplots()
ax.scatter(x_axis, y_axis, s=y_axis)
```

When you run the cell, you should see the same graph that you created using the MATLAB approach:

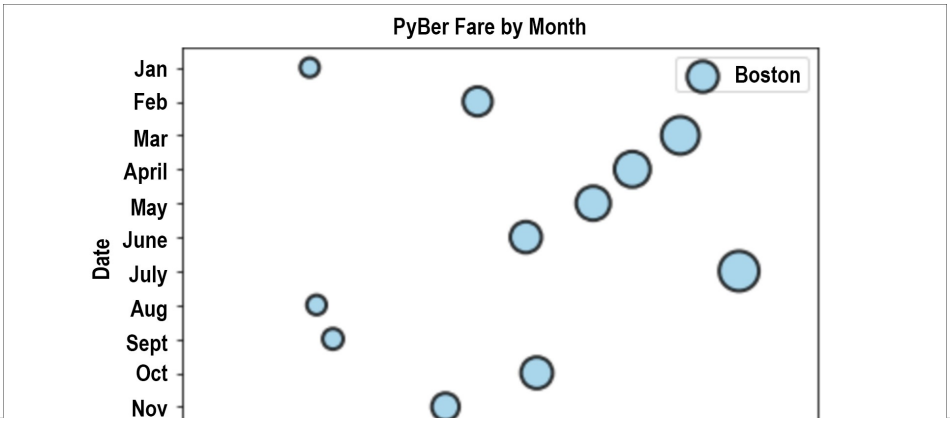


Now test your skills in the following Skill Drill.

Skill Drill

Using the object-oriented approach, make the following changes to your bubble chart:

1. Change the color of the markers to sky blue.
2. Change the size of the markers to 5 times each data point.
3. Make the color 20% transparent.
4. Add a black edge color to the circles.
5. Make the linewidth of the circles 2 points.
6. Add a legend for the city of Boston.
7. Add a title.
8. Switch the axis so that the Fare($) data is on the x-axis.
9. Add a limit for the x-axis data.
10. Invert the y-axis so that January is at the top.



Note

For more information on creating scatter plots and bubble charts using the object-oriented method, please refer to the following documentation:

Matplotlib documentation on matplotlib.axes.Axes.scatter (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.scatter.html)

Matplotlib documentation on scatter charts with a legend (https://matplotlib.org/3.1.0/gallery/lines_bars_and_markers/scatter_with_legend.html#sphx-glr-gallery-lines-bars-and-markers-scatter-with-legend-py)

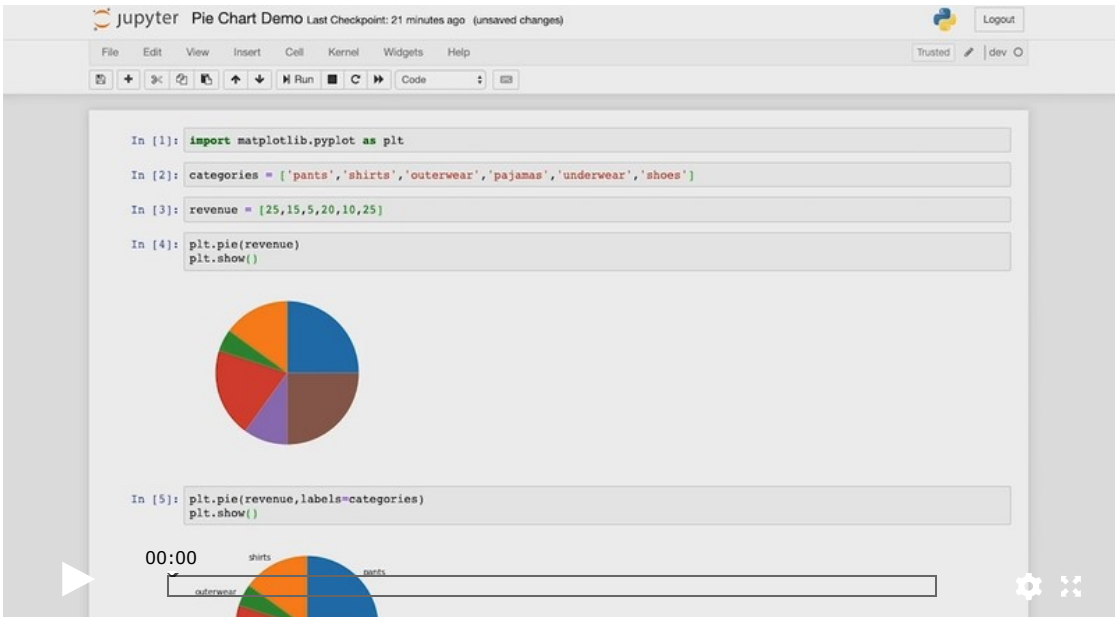
Matplotlib documentation on plotting a list of the named colors supported by Matplotlib (https://matplotlib.org/3.1.1/gallery/color/named_colors.html)

5.1.8: Create Pie Charts

After working all afternoon with Sasha, you've mastered the scatter plot. This presentation is starting to actually sound like it will be fun!

There's one last chart type to explore before you work on adding details to make your graphs stand out—and that's pie charts!

Pie charts aren't as common as line charts, bar charts, or scatter plots. Nonetheless, they provide a really great way to visualize percentages, which you know this dataset may call for. Additionally, they are named after a very delicious dessert, so working on them is always a bit sweeter than working on other charts!



Pie charts allow us to depict percentages of categories as wedges of a pie. One limitation of pie charts is that they can only represent one dataset or category. However, pie charts do have a high visual appeal. We can make them even more visually appealing by choosing vibrant colors and giving each pie wedge a label.

As we've done before, let's learn how to create a pie chart using both the MATLAB method and the object-oriented interface approach.

Create a Pie Chart Using the MATLAB Method

Let's begin with the MATLAB approach. As you've probably noticed, most of these charts follow a similar code structure. Just like we used the `plt.barh()` function to create a horizontal bar chart and the `plt.scatter()` function to create a scatter plot, we'll use the `plt.pie()` function to create a pie chart.

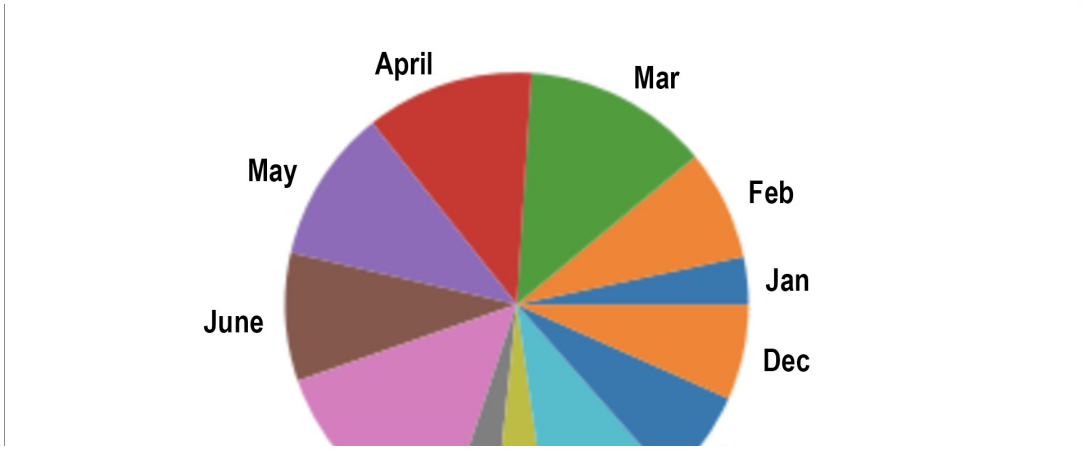
The `plt.pie()` function requires values that are in an array. We can also add labels that are in an array for each pie wedge, but this is optional.

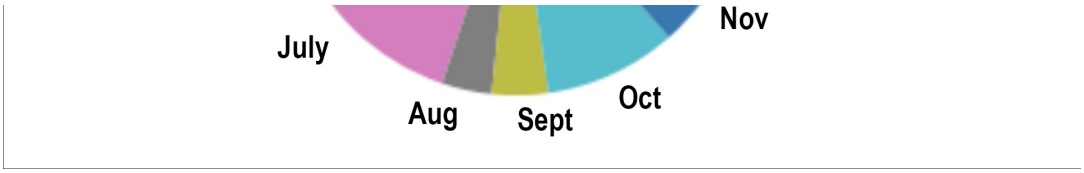
For our first pie chart, we'll use the x_axis data (months) as labels, and the y-axis (total fare for each month) as values.

Add the following code to a new cell:

```
plt.pie(y_axis, labels=x_axis)
plt.show()
```

When you run the cell, the pie chart should look similar to this:





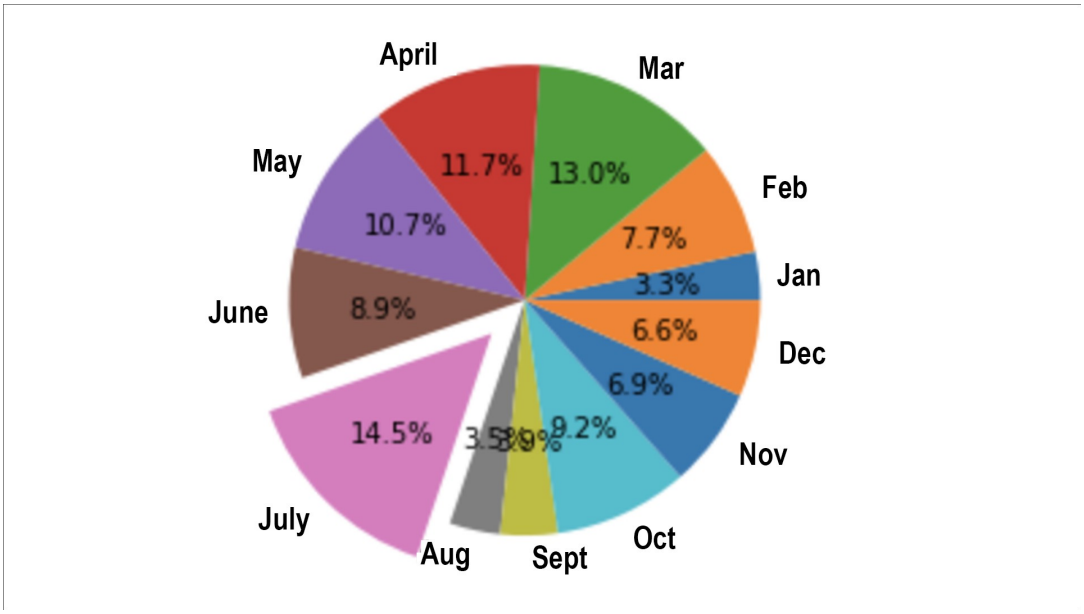
Let's make this chart more impactful by adding some bling. We have a few options with pie charts: we can add percentages to the wedges, add specific colors, and explode or "pop out" one or more wedges.

Let's add percentages for each month and "explode" the largest percentage, which is July, the seventh value in the `x_axis`. The "explode" parameter will offset the indicated wedge by a fraction of the radius, where "0" is zero distance from the center of the pie, and "1" is completely outside the diameter of the pie.

Add the following code to a new cell:

```
explode_values = (0, 0, 0, 0, 0, 0, 0.2, 0, 0, 0, 0, 0)
plt.pie(y_axis, explode=explode_values, labels=x_axis, autopct='%.1f%%')
```

When you run the code, the pie chart will look like this:

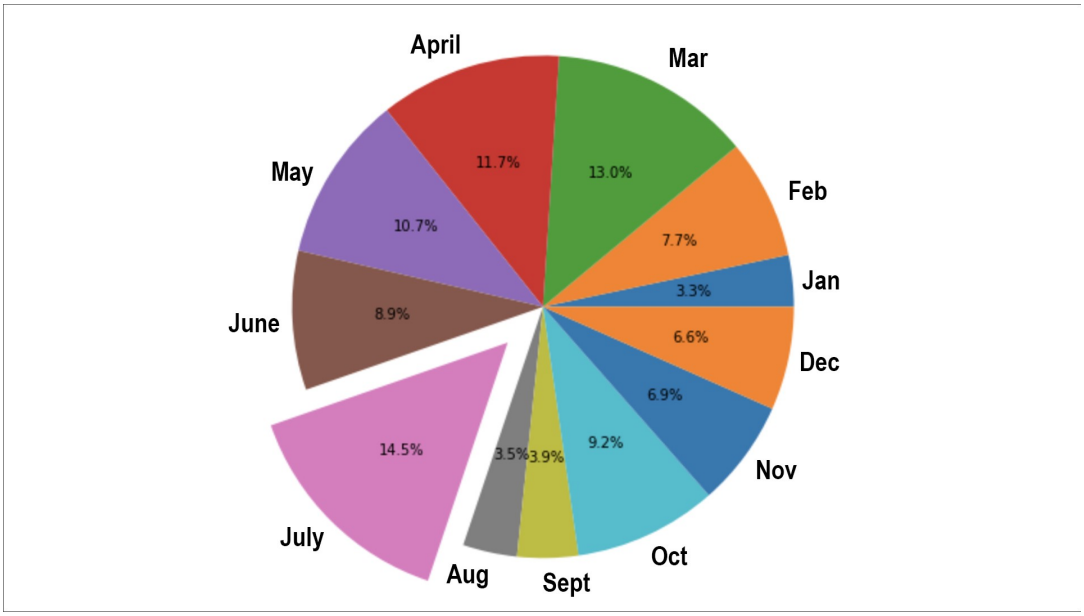


Let's break down what the graphing code is doing:

- To explode a pie wedge, we use array-like data points (like the tuple `explode_values = (0, 0, 0, 0, 0, 0, 0.2, 0, 0, 0, 0, 0)`) for each value in the pie chart.
  - The length of the tuple should be equal to the number of wedges in the pie chart.
  - The value specified to "explode" a wedge is the fraction of the radius from the center of the pie for each wedge.
- To pop out the seventh month, July, we use a decimal-point value of `0.2`.
- To add the percentage of each wedge on the pie chart, we use the `autopct` parameter and provide the format of one decimal place using `.1f%`.
- The `%` before and after the `.1f%` formats the number as a percentage.

The percentages on the pie chart seem a bit crowded. Let's increase the size of the pie chart by using `plt.subplots(figsize=(8, 8))`.

Add this code on the first line in the previous code cell and run the cell again. You should see the following chart:



Now the percentages are not so crowded. But the colors for each wedge may not be the best choice. When we don't specify the colors as parameters (i.e., `"colors="`), then Matplotlib provides default "colors in the currently active cycle," according to the **documentation** (https://matplotlib.org/3.1.1/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py) for the `"colors="` parameter.

Our last pie chart modification will be to change the color of each pie wedge. The choice of colors for your visualizations is a big deal, and frankly, not everyone has an eye for color. Color is also a matter of preference, and what pleases one person may turn off someone else. In general, it is a best practice to choose lighter, brighter, and analogous colors over darker tones.

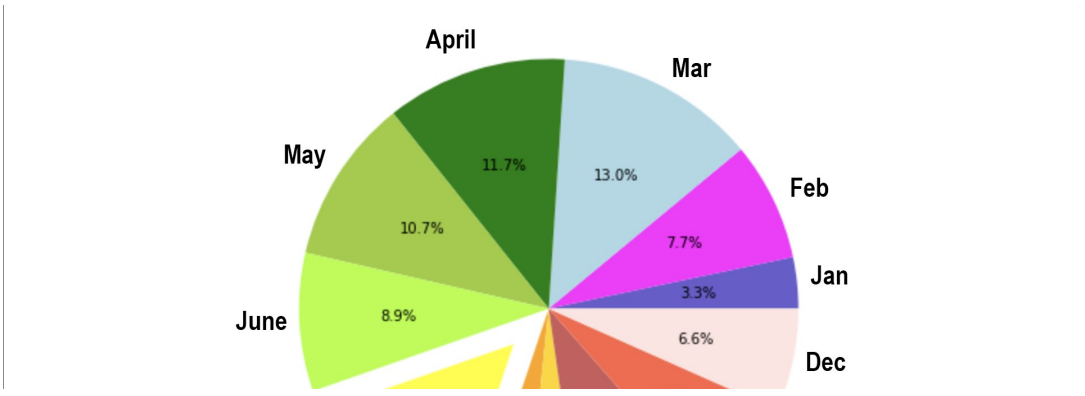
We're going to change the colors of each pie wedge by using **the CSS colors in the gallery in the Matplotlib documentation (https://matplotlib.org/3.1.1/gallery/color/named_colors.html#sphx-glr-gallery-color-named-colors-py)** .

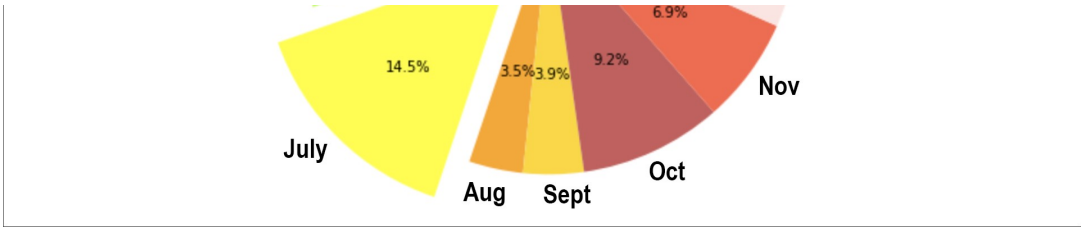
We'll add the `"colors"` parameter in the `pie()` function with our other parameters, but first we need to assign a `"colors"` variable to an array with the names of our 12 colors.

Add the following code to a new cell:

```
# Assign 12 colors, one for each month.
colors = ["slateblue", "magenta", "lightblue", "green", "yellowgreen", "gree
explode_values = (0, 0, 0, 0, 0, 0, 0.2, 0, 0, 0, 0, 0)
plt.subplots(figsize=(8, 8))
plt.pie(y_axis,
    explode=explode_values,
    colors=colors,
    labels=x_axis,
    autopct='%.1f%%')

plt.show()
```

When you run the cell, the pie chart should look like this:





> **Note**
> For more information, see the [Matplotlib documentation on creating a pie chart using the MATLAB method (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.pie.html#matplotlib.pyplot.scatter)](https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.pie.html#matplotlib.pyplot.scatter) .

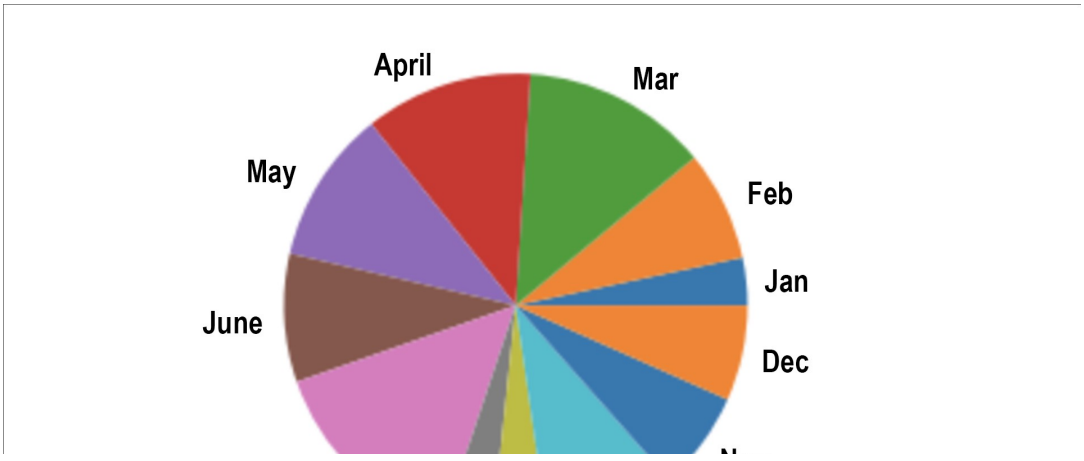
Create a Pie Chart Using the Object-Oriented Interface

Can you guess what we're going to do next? You got it—create the same pie chart using the object-oriented approach.

Add the following to your existing code:

```
fig, ax = plt.subplots()
ax.pie(y_axis,labels=x_axis)
plt.show()
```

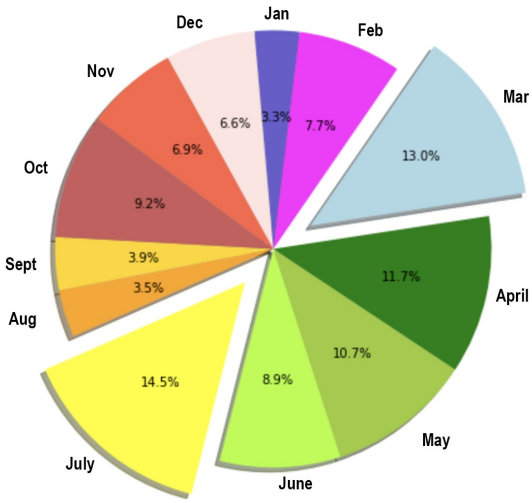
When you run this code, you'll see the same graph that we created using the MATLAB approach:



Skill Drill

Using the object-oriented approach, make the following changes to the pie chart:

1. Add a percentage to one decimal place to each wedge of the pie.
2. Increase the figure size to 8x8.
3. Explode the two highest percentage months.
4. Add a shadow.
5. Add a start angle so that January is at the top.
6. Reverse the order so that the month order is in a clockwise direction.
7. Add new colors of your own choosing or use the colors from the previous pie chart.



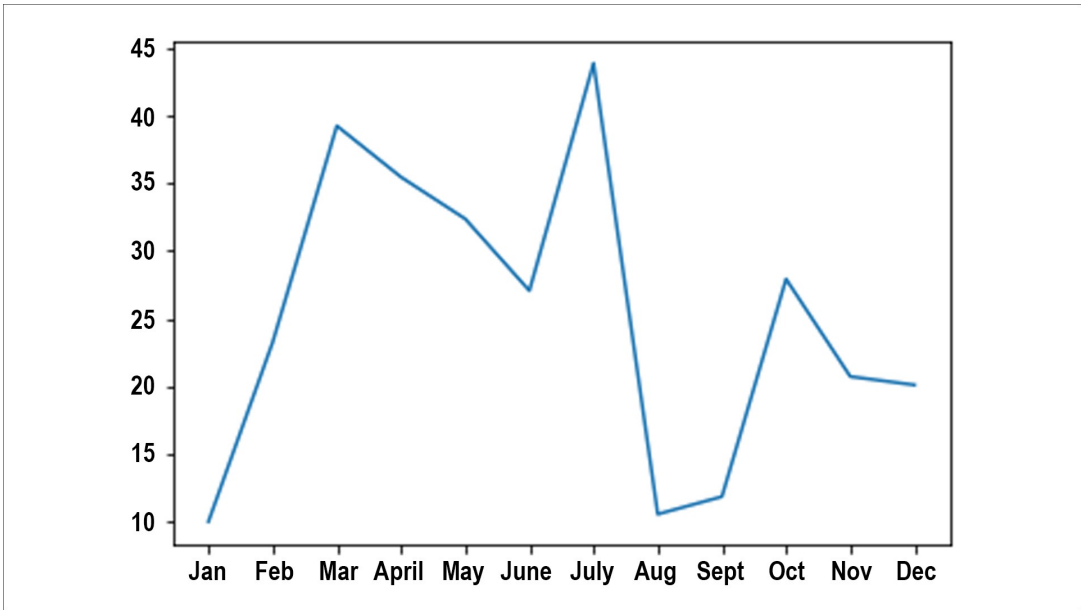
Which Chart Is Best?

Great job! You now know how to create line charts, bar charts, scatter plots, bubble charts, and pie charts. Before moving on to adding extra details to these charts, let's reflect. Which chart do you think is best for graphing the ride-sharing data? Your CEO

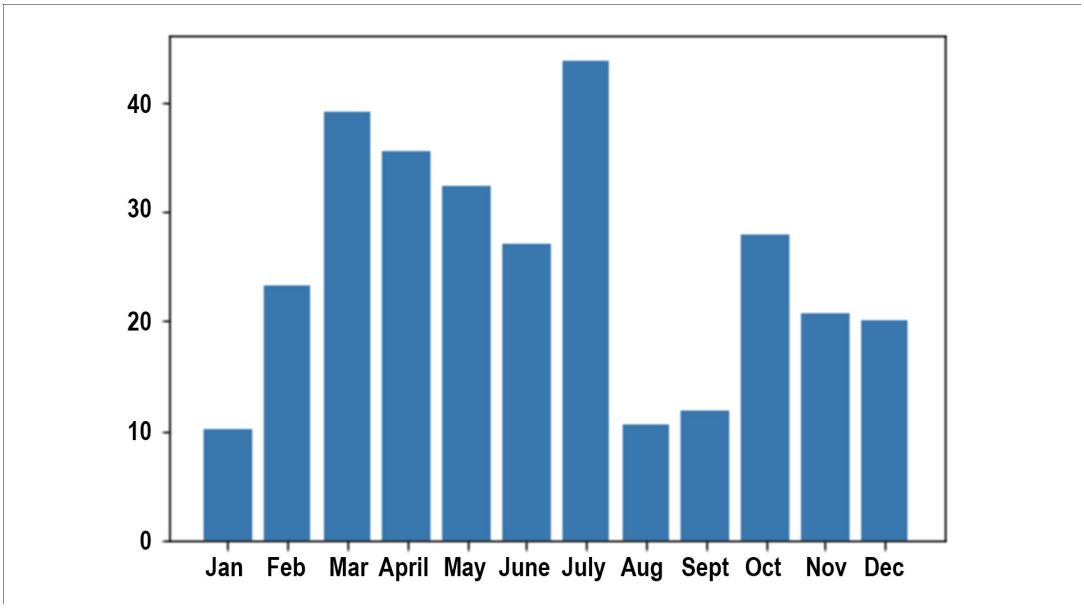
will most likely ask why you chose the charts you did, so you'd better be prepared to answer.

Let's review the pie chart first since it's fresh in our minds. In this chart, we can see that some months contributed a very small amount of the overall fare collected for the year. We can also see that March through July are the most lucrative for ride-sharing. How does this compare to our other charts?

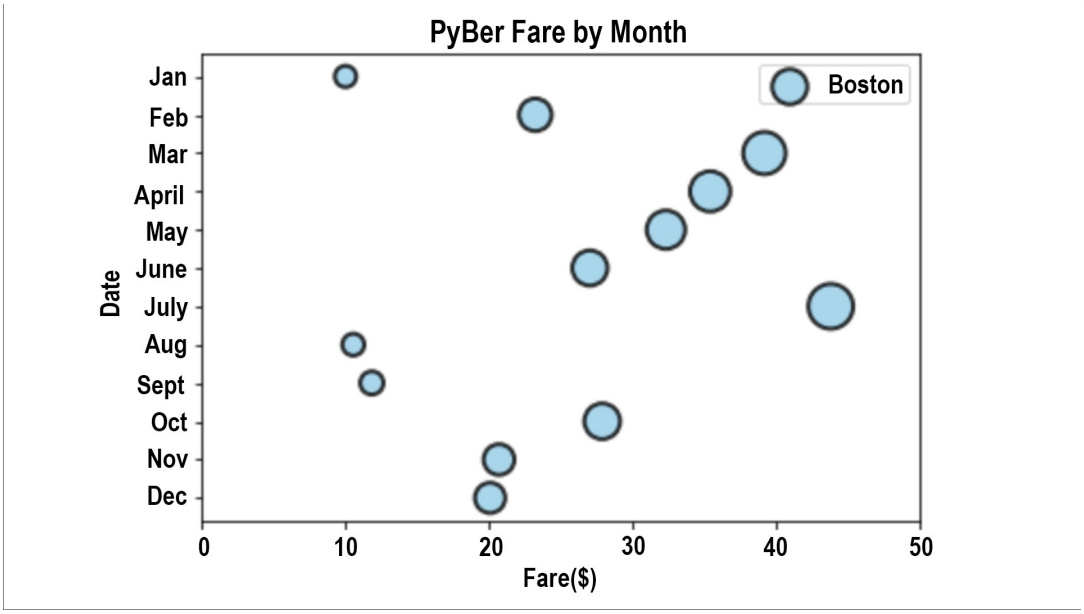
The line chart conveyed the same information: we can see that March through July are the most lucrative for ride-sharing, with a small peak in October.



The vertical and horizontal bar charts also conveyed the same information:



The scatter plot didn't do a great job of getting that information across, but the bubble chart we created in the Skill Drill did a decent job of conveying the same information as the line, bar, and pie charts, because the bubbles increased in diameter as the average fare increased. Still, the bubble chart may not be a good chart to use for this particular dataset, because there are only two data points for this scatter plot: the months on the y-axis and the fare amount on the x-axis. If we had a third parameter that could be the size of the marker, like the number of riders for each month, then using the scatter plot might be a good choice.



As a data analyst, creating visualizations is part of the exploratory data analysis process. Deciding which chart represents the data the best is trial and error. As you gain more experience at creating charts, you will know which charts will best represent the data.

Now that we have learned how to create a variety of charts, let's learn how to annotate charts by adding error bars and changing the major and minor ticks on the axes.

> **Note**
>
> For more information, see the [Matplotlib documentation on creating a pie chart using the object-oriented interface method (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.pie.html)](https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.axes.Axes.pie.html) .

5.1.9: Chart Extras

You've made a line chart, vertical and horizontal bar charts, scatter plots, bubble charts, and pie charts. But you're not going to stop there—oh, no—you're going to make the most impressive presentation your CEO has ever seen, and that means having a chart ready to show on the spot no matter the question she asks or edit she requests.

To do that, you need to think critically about the visualizations you've made so far. What if she wants to see error bars? Maybe you should add minor x- or y-ticks, or change the scale for the major ticks. Whatever the additional details she wants, you're going to be ready. So you roll up your sleeves and get back to coding!

With Matplotlib, we can add a variety of extras to our charts. Let’s add some more enhancements to our bag of tricks, like error bars, minor x- and y-ticks, and custom increments for the major x- and y-ticks.

Add Error Bars to a Line Chart

Adding error bars can show either the standard deviation, standard error, confidence intervals, or minimum and maximum values of a dataset. When added to a chart, they can visually show the variability of the plotted data. By looking at the error bars, one can infer the significance of the data.

To create a line chart with error bars using the MATLAB approach, we use the `errorbar()` function. We also add the x and y data parameters and the `yerr=` parameter inside parentheses, as shown here:

```
plt.errorbar(x, y, yerr=<value>)
```

To the ride-sharing line chart, let's add the standard deviation of the fare, or y-axis, as error bars.

Create a new Jupyter Notebook file. Add the following code blocks in order to a new cell and run each cell:

```
%matplotlib inline
```

```
# Import dependencies.
import matplotlib.pyplot as plt
import statistics
```

We are importing a new dependency, the statistics module, which comes with the Anaconda installation. We're going to use the statistics module so that we can add the y-error bars to line and bar charts. With the statistics module, we can calculate the standard deviation of the fare amount.

> **Rewind**
> The standard deviation is a measure of the amount of variation, or spread, of a set of values from the mean.

```
# Set the x-axis to a list of strings for each month.
x_axis = ["Jan", "Feb", "Mar", "April", "May", "June", "July", "Aug", "Sept"

# Set the y-axis to a list of floats as the total fare in US dollars accumul
y_axis = [10.02, 23.24, 39.20, 35.42, 32.34, 27.04, 43.82, 10.56, 11.85, 27.
```

Next, we'll calculate the standard deviation of the `y_axis` values using the statistics module.

Add the following code to a new cell:

```
# Get the standard deviation of the values in the y-axis.
stdev = statistics.stdev(y_axis)
stdev
```

When you run the cell, it will compute the standard deviation and output that number, as shown in the following image:

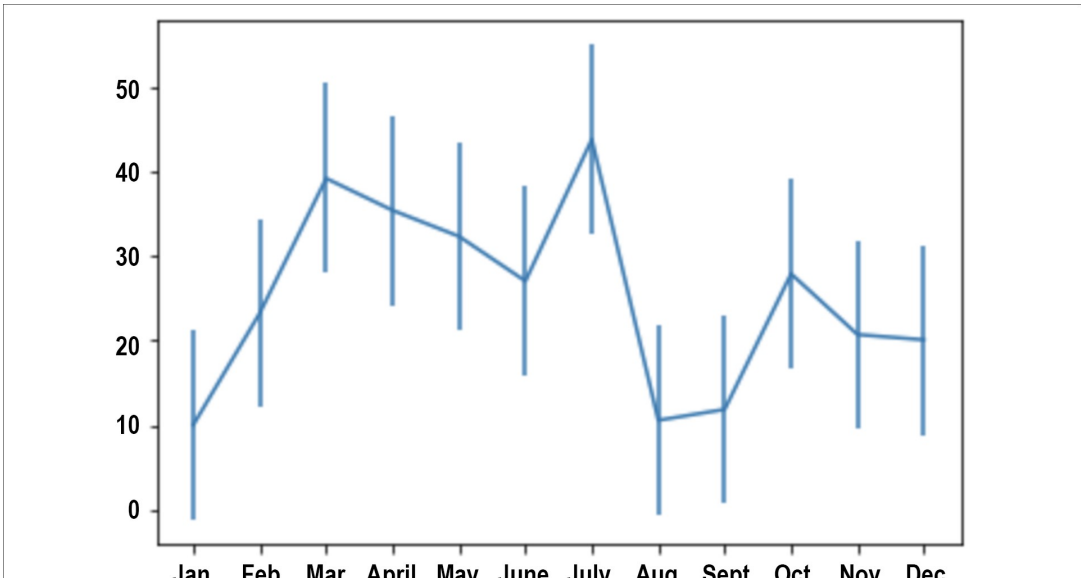
```
# Get the standard deviation of the values in the y_axis.
stdev = statistics.stdev(y_axis)
stdev
```

```
11.208367917035753
```

Next, add the `x_axis`, `y_axis`, and `yerr=stdev` to the `errorbar` function:

```
plt.errorbar(x_axis, y_axis, yerr=stdev)
```

When you run the cell, you'll see the following line chart with error bars for the fare amounts:

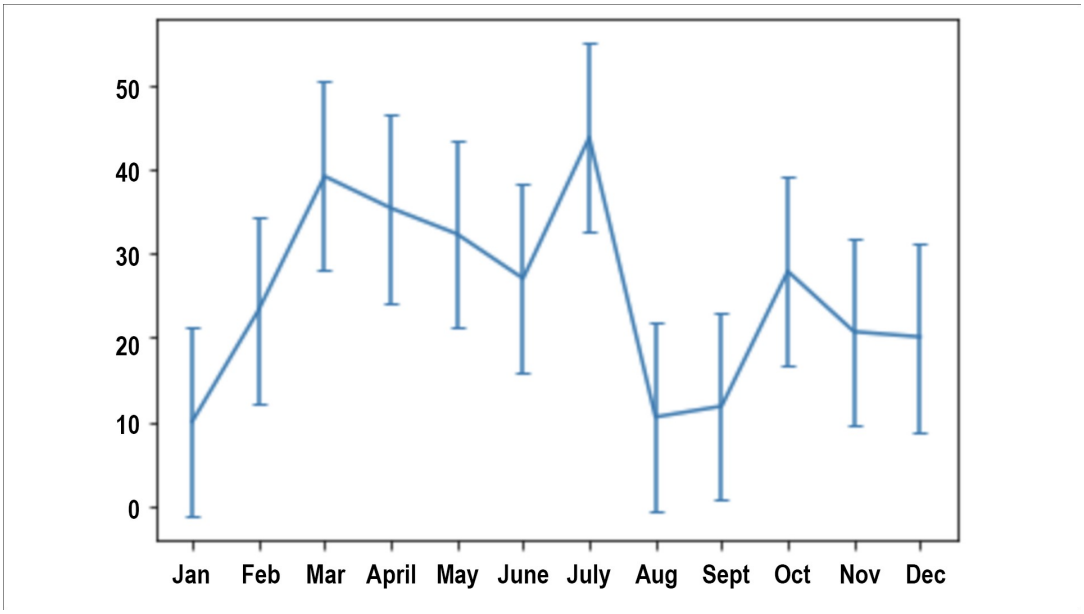


We can make the error bars look more professional by adding a “cap” to the error bars that’s equal to a scalar value. To do this, we use the `capsize=` parameter.

Let’s add `capsize=3` to the error bars. Add the following code:

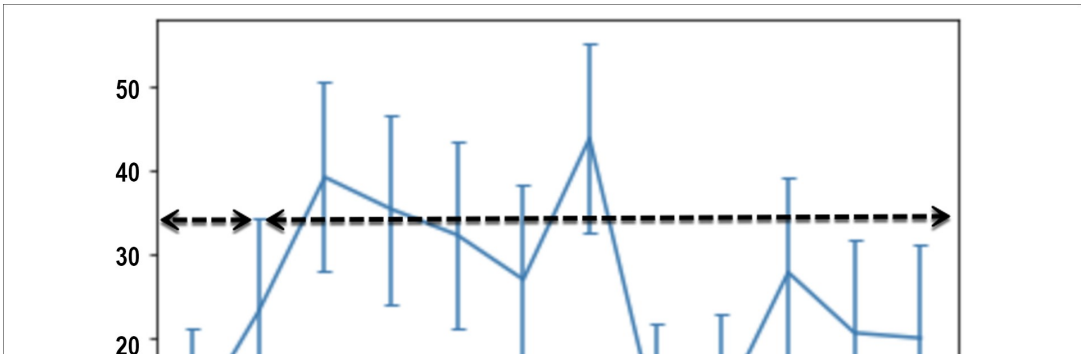
```
plt.errorbar(x_axis, y_axis, yerr=stdev, capsize=3)
```

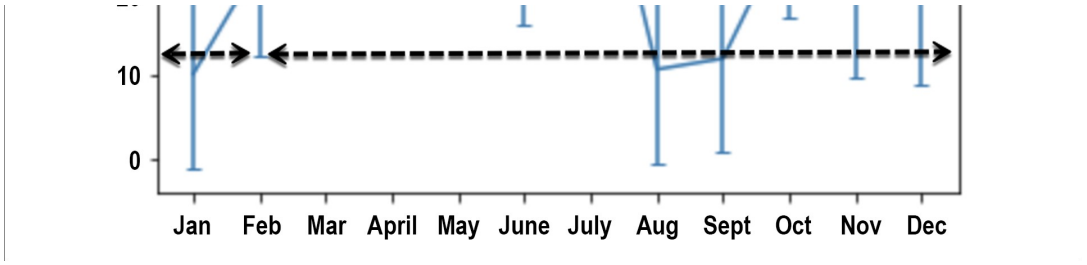
When you run the code, you’ll see the same chart but with caps on the error bars:



Now that we’ve added error bars to the line chart, we can begin to make assumptions about the plotted data.

For instance, in the following figure, we have added dotted lines for the one standard deviation, 11.2, above the mean and one standard deviation below the mean for February.



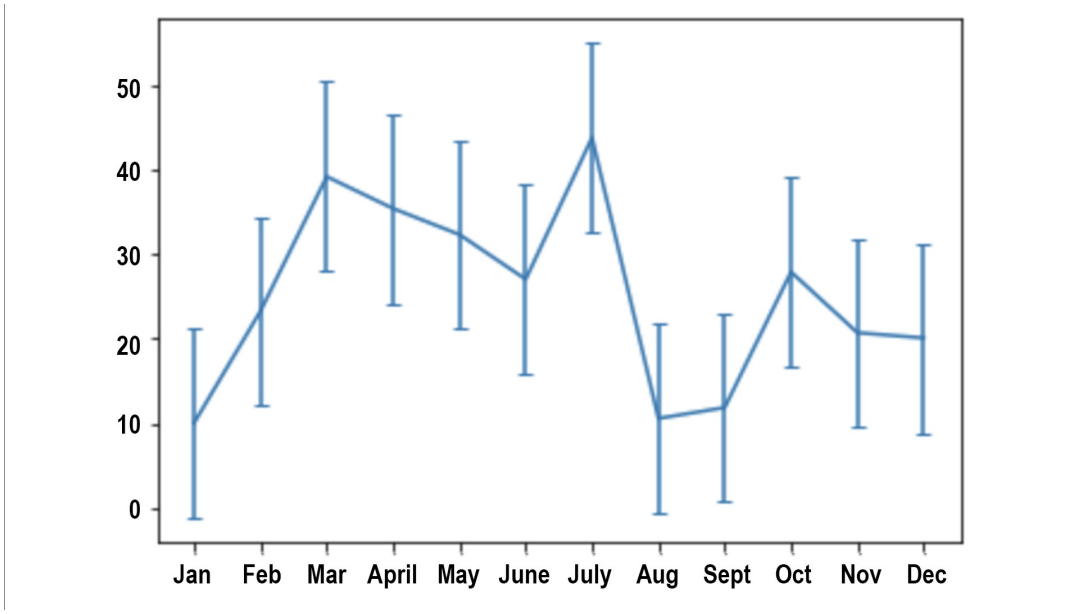


Here, the standard deviations above and below the mean overlap with the error bars from some of the other months. When comparing all the months like this, we can assume that there is very little variability in the fares from month to month. And overall, there is a wide range in the average fare price for every month.

Now let's add error bars and a capsize using the object-oriented approach. Add the following to a new cell.

```
fig, ax = plt.subplots()
ax.errorbar(x_axis, y_axis, yerr=stdev, capsize=3)
plt.show()
```

When you run the code block, you'll see the same graph we created using the MATLAB approach:



NOTE

> For more information about adding error bars to line charts using the MATLAB and the object-oriented methods, please refer to the following documentation:
>
> **Matplotlib documentation on matplotlib.pyplot.errorbar (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.errorbar.html)**
>
> **Matplotlib documentation on axes.Axes.errorbar (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.errorbar.html)**

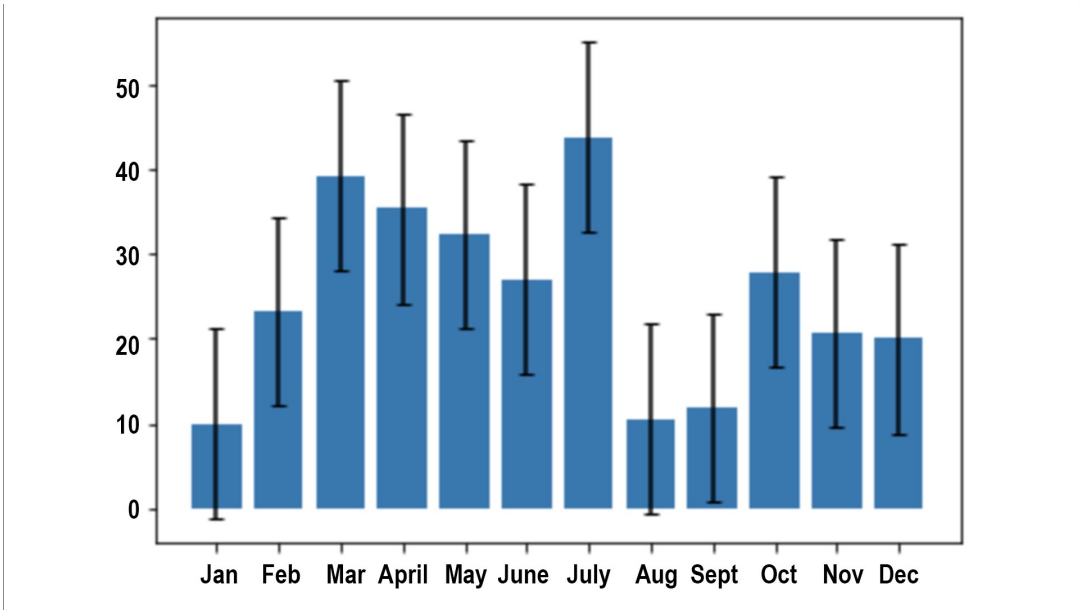
Add Error Bars to a Bar Chart

To add error bars and caps to a bar chart using the MATLAB approach, we can use the `yerr=` and the `capsize=` parameters with the `plt.bar()` function.

Add the following code to a new cell:

```
plt.bar(x_axis, y_axis, yerr=stdev, capsize=3)
```

When you run the cell, the bar chart with error bars will look like this:

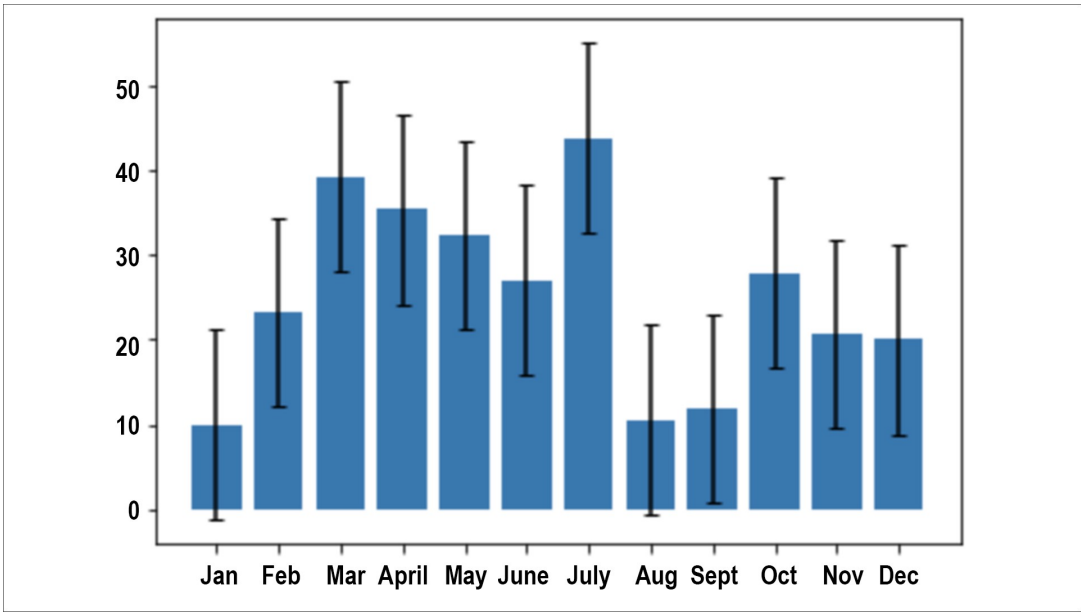


Now let's create the same graph using the object-oriented approach.

Add the following code to a new cell:

```
fig, ax = plt.subplots()
ax.bar(x_axis, y_axis, yerr=stdev, capsize=3)
plt.show()
```

When you run the code, you'll see the same graph that we created using the MATLAB approach:



Note

For more information about adding error bars to bar charts using the MATLAB and the object-oriented methods, please see the following documentation:

[Matplotlib documentation on matplotlib.pyplot.bar (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.bar.html)]

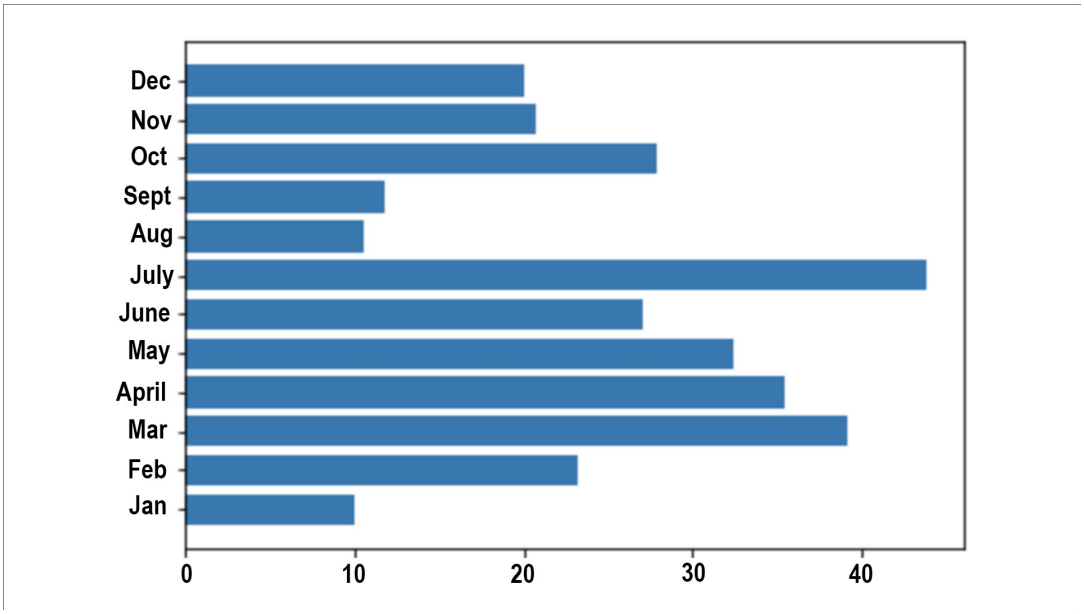
[Matplotlib documentation on axes.Axes.bar (https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.axes.Axes.bar.html)]

Change the Major Ticks

When we graphed our horizontal bar chart with the fare on the x-axis, Matplotlib automatically added the ticks. We can customize our charts so that the major ticks on

an axis are more spread out or closer together. Which one you choose depends on what the data looks like on a graph.

Take a look at the horizontal bar chart. When it was plotted, the major ticks on the x-axis were spaced every $10. This spacing makes it difficult to estimate the average fares for some months. Let's see if we can improve that.

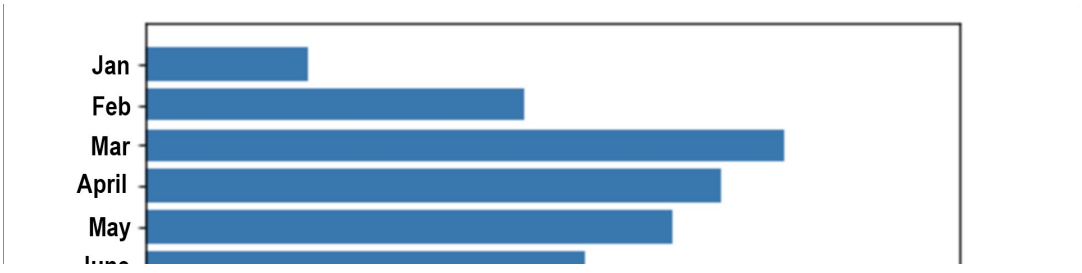


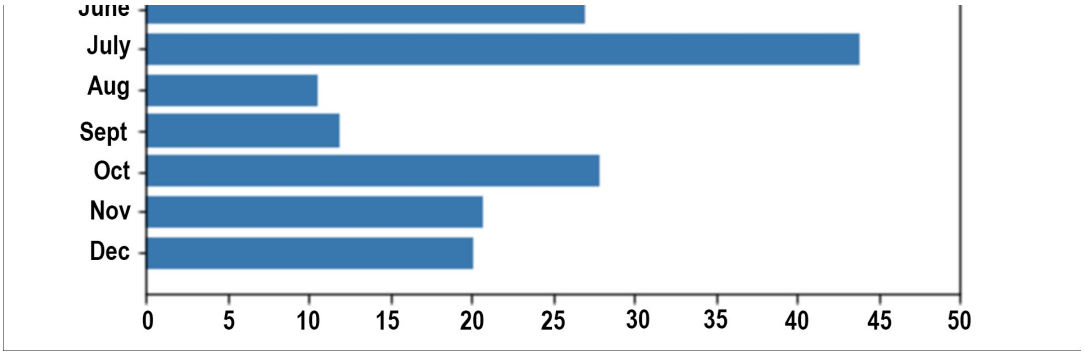
To adjust the major x-axis ticks on a horizontal bar chart, we use the `xticks()` function.

Add the following code to a new cell:

```
import numpy as np
plt.barh(x_axis, y_axis)
plt.xticks(np.arange(0, 51, step=5.0))
plt.gca().invert_yaxis()
```

When you run the cell, the output will be a horizontal bar chart with the major ticks at every $5 increment:





As you can see, setting the major ticks to $5 increments makes it easier to estimate the average fare for some of the months.

Did you notice that we added some code that you hadn't seen yet? Let's break it down:

- We imported the `numpy` module. NumPy is a numerical mathematics library that can be used to create arrays or matrices of numbers.
- We created a horizontal bar graph using the `barh()` function and passed the x- and y-axis data inside the parentheses.
- To create the $5 increments for the x-ticks, we use the `numpy` module to create an array of numbers using the `arange()` function. We defined the lower and upper limit of the x-axis and the increment frequency with the code: `np.arange(0, 51, step=5.0)`.
- Finally, we use `gca()` method to get the current axes and invert them.

Let's drill down a little more into `step=`. If we were to import the `numpy` module and add the code snippet—`np.arange(0, 51, step=5.0)`—in a new cell like this:

```
import numpy as np
np.arange(0, 51, step=5.0)
```

The output of the `arange()` function would be in increments of 5, from zero to 50. If we didn't include `step=`, we would get all the numbers from zero to 50.

Here's an example of creating a range between zero and 50 with an increment of 5:

```
np.arange(0, 51, step=5.0)
```

```
array([ 0.,  5., 10., 15., 20., 25., 30., 35., 40., 45., 50.])
```

You can create the same graph using the object-oriented approach with the following code:

```
fig, ax = plt.subplots()
ax.barh(x_axis, y_axis)
ax.set_xticks(np.arange(0, 51, step=5.0))
plt.show()
```

Alternatively, you can use `ax.xaxis.set_ticks()` instead of `ax.set_xticks()`.

If our y-axis had numerical values, we could change how often every major y-tick occurs by using the `yticks()` function for the MATLAB approach and the `set_yticks()` function for the object-oriented approach.

> **Note**
> For further information on NumPy and the `arange` function, please see the following documentation:
>
> **NumPy website (https://numpy.org/)**
>
> **SciPy page on numpy.arange (https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html)**

Add Minor Ticks

We can also add minor ticks without numerical values between the major ticks on the x- or y-axis. Minor ticks make it even easier to estimate values for the data in the chart.

To add minor ticks to the x-axis on a horizontal bar chart, we use the `set_minor_locator` function with the `xaxis` class in the object-oriented method, like this:

```
x.xaxis.set_minor_locator()
```

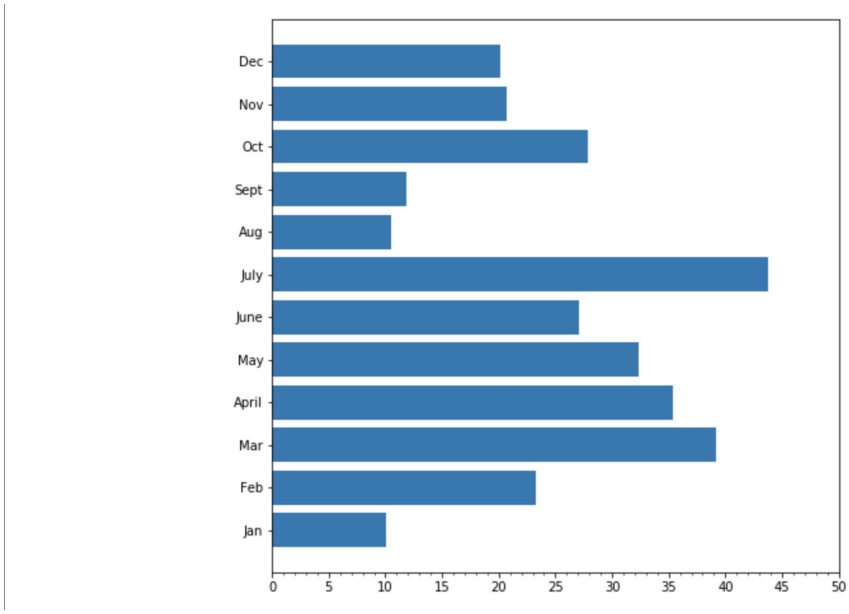
Inside the `set_minor_locator` function, we add the `MultipleLocator` method and pass how often we want our minor ticks.

To use these functions and methods, we have to import the `MultipleLocator` method from the `matplotlib.ticker` function.

Add the following code to a new cell:

```
from matplotlib.ticker import MultipleLocator
# Increase the size of the plot figure.
fig, ax = plt.subplots(figsize=(8, 8))
ax.barh(x_axis, y_axis)
ax.set_xticks(np.arange(0, 51, step=5.0))

# Create minor ticks at an increment of 1.
ax.xaxis.set_minor_locator(MultipleLocator(1))
plt.show()
```

Now, the figure will have minor ticks every $1 increment and major ticks at every $5 increment, as seen here:



Note

> For more information, see the **Matplotlib documentation on major and minor ticks (https://matplotlib.org/3.1.0/gallery/ticks_and_spines/major_minor_demo.html)** .

5.1.10: Plot a Pandas DataFrame and Series

Now that you're up to speed on all four types of charts (line, bar, scatter, and pie), you're eager to dig into the data. One of the questions V. Isualize is famous for asking during these presentations is about the exploratory part of the analytical process. In other words, when you have a new dataset, what is your process in determining relationships between variables in the data? Or what type of data do you have to work with? And are there any outliers or patterns in the data?

This critical, exploratory data analytic step can save roughly 15–50% of your time on a project because it provides a targeted plan for how to clean, sort, and create smaller datasets. It's not something a savvy analyst will skip. And of course you want V. Isualize to immediately recognize you as a savvy analyst!

Up to this point, you've been working with a small dataset, but soon you'll be analyzing large datasets that will be read into a DataFrame. Since you're getting to be an expert at creating visuals, Omar is going to show you how to plot data from a DataFrame just by referencing the data columns.

During data analysis, one best practice is to visualize the data once it's in a DataFrame to determine if it needs to be cleaned, sorted, or modified before it's analyzed. This

can also prevent the scenario of being knee deep in the analysis and having to go back to the beginning to change a data type or create a feature. One way to do that is to plot data directly from a DataFrame or Data Series.

We can quickly visualize our data from a DataFrame or Series by using the `plot()` function by adding the Series (`ds`) or DataFrame (`df`) in the following format: `ds.plot()` or `df.plot()`.

To plot a DataFrame, we add the x and y values inside the parentheses as the DataFrame columns we want.

Let's briefly walk through how to plot data from a DataFrame, but first let's download the practice dataset into our `Resources` folder.

1. In your PyBer_Analysis folder, create a folder named Resources.
2. Click the following link to download the PyBer_ride_data.csv file into your Resources folder.

   Download PyBer_ride_data.csv
   (https://courses.bootcampspot.com/courses/138/files/15538/download?wrap=1)
   (https://courses.bootcampspot.com/courses/138/files/15538/download?wrap=1)

   You should now have a CSV file named `PyBer_ride_data.csv` in your Resources folder.

3. Create a new Jupyter Notebook file named `PyBer_ride_data`.
4. In the first cell, do the following:
   1. Add the magic command `%matplotlib inline` on the first line.
   2. Import the following dependencies.
      - `matplotlib.pyplot as plt`
      - `numpy as np`
      - `Pandas as pd`
   3. Read the `PyBer_ride_data.csv` into a Pandas DataFrame.

Your Jupyter Notebook file should look like this:

```
%matplotlib inline
# Dependencies
```

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
# Load in csv
pyber_ride_df = pd.read_csv("Resources/PyBer_ride_data.csv")
pyber_ride_df
```

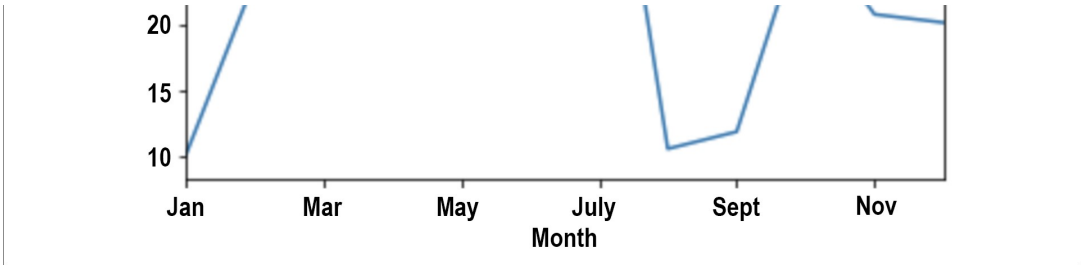
When we run this cell, the output is the ride-sharing data we've been working with in a DataFrame:

	Month	Avg. Fare ($USD)
0	Jan	10.02
1	Feb	23.24
2	Mar	39.20
3	April	35.42
4	May	32.34
5	June	27.04
6	July	43.82
7	Aug	10.56
8	Sept	11.85
9	Oct	27.90
10	Nov	20.71
11	Dec	20.09

We can plot the months along the x-axis and the fare on the y-axis using the `plot()` function.

```
pyber_ride_df.plot(x="Month", y="Avg. Fare ($USD)")
plt.show()
```

When we run this cell, it gives us a similar line chart to the one we saw before:



We can adjust the x-ticks to show all the months by editing our code to look like this:

```
# Set x-axis and tick locations.
x_axis = np.arange(len(pyber_ride_df))
tick_locations = [value for value in x_axis]
# Plot the data.
pyber_ride_df.plot(x="Month", y="Avg. Fare ($USD)")
plt.xticks(tick_locations, pyber_ride_df["Month"])
plt.show()
```

Let's examine the code to see what is going on:

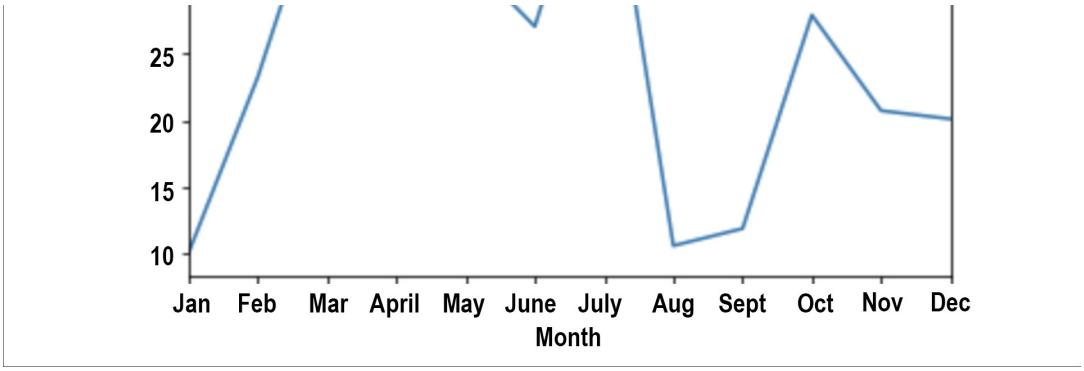
1. First, we need an array that is the length of the DataFrame. This will allow us to plot each month for each value in the array. With the NumPy module, we create an array of numbers using the `arange` function for the length of the `pyber_ride_df`:

```
x_axis = np.arange(len(uber_ride_df))
x_axis
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

2. Then we set the tick locations for every value, or month, in the x-axis using list comprehension.
3. We add the `xticks()` function and pass in the `tick_locations` and the labels of the x-ticks, `pyber_ride_df["Month"]`.

When we run this cell, we get all the months on the x-axis:



There are two ways to create a bar chart with the same data. In the first approach, we "chain" the `bar()` function to the `plot()` function. Let's give that a try.

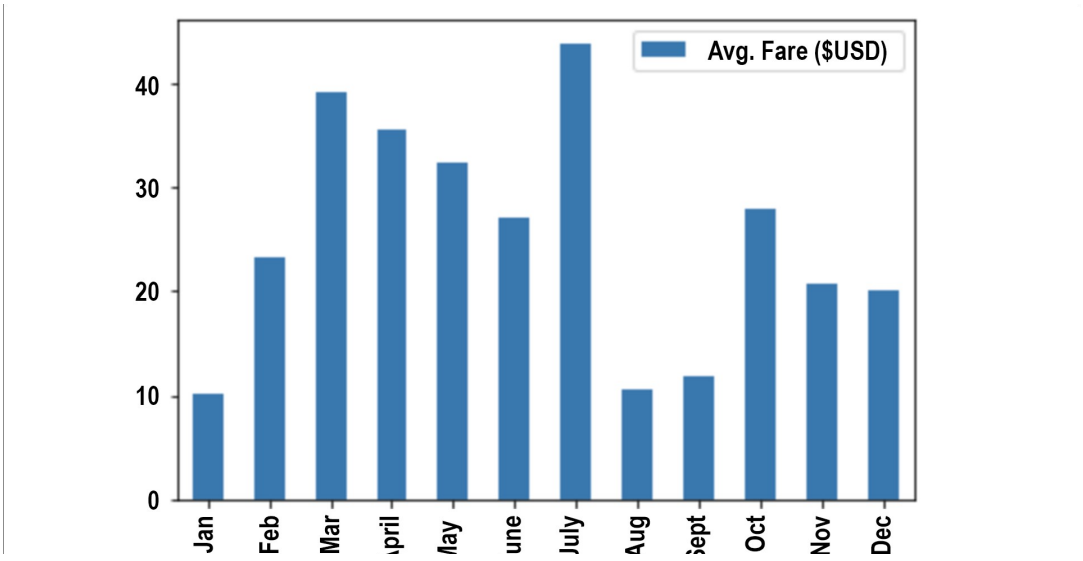
Add the following code to a new cell and run the cell to see a bar chart of the same data:

```
pyber_ride_df.plot.bar(x="Month", y="Avg. Fare ($USD)")
plt.show()
```

The other approach is to add the `kind` parameter to the `plot()` function. Add the following code to a new cell:

```
pyber_ride_df.plot(x="Month", y="Avg. Fare ($USD)", kind='bar')
plt.show()
```

Both of these approaches produce a bar chart with all the months on the x-axis, as you see here:



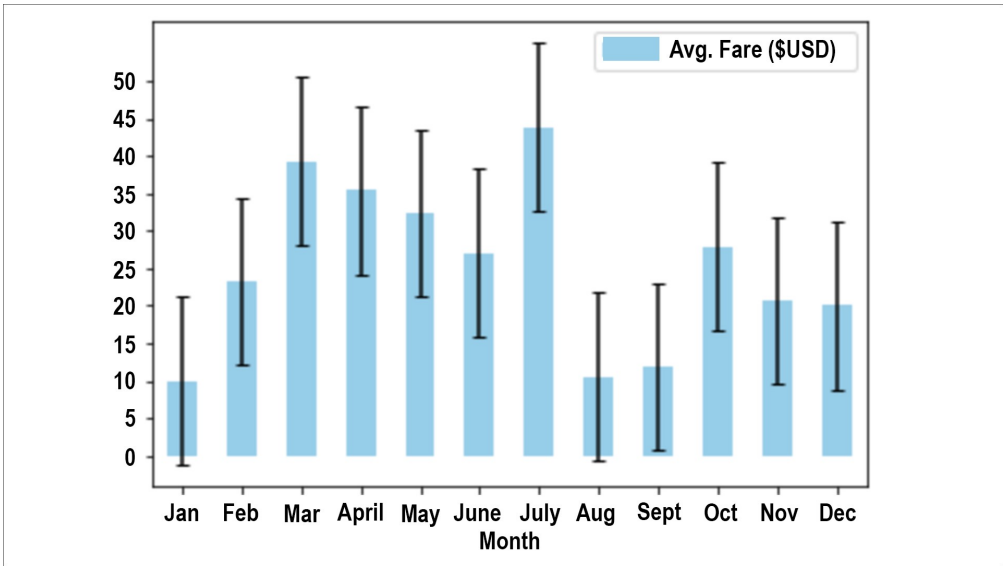
Now test your skills with the following Skill Drill.

Skill Drill

Using the data from the PyBer ride DataFrame, create a bar chart with the following annotations and characteristics:

1. Add error bars.
2. Add caps to the error bars.
3. Change the color of the bars to sky blue.
4. Rotate the labels on the x-axis to horizontal.
5. Set the y-axis increment to every $5.

Your chart should look similar to this:



Now that you're skilled in creating visualizations, it's time to move on to working with the "real" ride-sharing data. But before you do, here is a table that will help you decide which type of graph to create depending on what you want to show. We haven't yet covered all of these, like the stacked bar chart and the box-and-whisker plot, because we only had one dataset and we weren't looking for outliers. However, we may need to use them later in this module.

What Do You Want to Do?	Use These Charts
Compare values of datasets	Line, bar, scatter, and pie
Show how individual parts make up a whole	Pie and stacked bar
Show distribution of the data and outliers	Line, bar, scatter, and box-and-whisker
Show trends over time	Line or bar
Establish relationships between variables	Line, scatter, and bubble

Note

For more information on plotting a Series or DataFrame, please see the **Pandas documentation on visualization (https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html)** .

5.2.1: Import and Inspect CSV Files

The moment has finally come! You're ready to download the CSV files that are worth their weight in gold: two datasets containing four months of rideshare data, just waiting for you to unlock their secrets.

Import the Data

The first step is to import the data. To do that, follow these steps:

1. Click the following links to download the city_data.csv and ride_data.csv files into your Resources folder.
   (https://courses.bootcampspot.com/courses/138/files/14729/download?wrap=1)

   Download city_data.csv
   (https://courses.bootcampspot.com/courses/138/files/14729/download?wrap=1)
   (https://courses.bootcampspot.com/courses/138/files/14729/download?wrap=1)

   Download ride_data.csv
   (https://courses.bootcampspot.com/courses/138/files/14927/download?wrap=1)
   (https://courses.bootcampspot.com/courses/138/files/14927/download?wrap=1)

You should now have the following two CSV files in your Resources folder:

- `city_data.csv`
- `ride_data.csv`

Inspect the Data

Before we do any analysis, we will inspect the data and answer the following questions:

- How many columns and rows are there?
- What types of data are present?
- Is the data readable or does it need to be converted in some way?

If you open the `city_data.csv` file, you will see three columns: city, driver_count, and type. Here's a snapshot of the first ten rows of data in this file:

	A	B	C
1	city	driver_count	type
2	Richardfort	38	Urban
3	Williamsstad	59	Urban
4	Port Angela	67	Urban
5	Rodneyfort	34	Urban
6	West Robert	39	Urban
7	West Anthony	70	Urban
8	West Angela	48	Urban
9	Martinezhaven	25	Urban
10	Karenberg	22	Urban

Let's see how much data is in this file. Can you remember how to jump to the end of a column in Excel?

> **Rewind**
> To get to the last row of an Excel file, place the cursor in a column that doesn't have any empty cells and press Command + down arrow on a Mac, or CTRL+ down arrow on Windows.

Here's a snapshot of the last ten rows of the `city_data.csv` file:

112	Lake Jamie	4	Rural
113	Lake Latoyabury	2	Rural
114	North Jaime	1	Rural
115	South Marychester	1	Rural
116	Garzaport	7	Rural
117	Bradshawfurt	7	Rural
118	New Ryantown	2	Rural
119	Randallchester	9	Rural
120	Jessicaport	1	Rural
121	South Saramouth	7	Rural

We can see that there are 121 rows. Inspecting this file further, we notice that each column has a header. Each row contains a city that has a driver_count and the type of city: Urban, Suburban, or Rural.

Scrolling through this CSV file, we see that there are no empty rows. Once we add this CSV file into a Pandas DataFrame, we'll be able to determine the data type for each column.

If we open the `ride_data.csv` file, we can see four columns: city, date, fare, and ride_id. Here are the first 10 rows:

1	city	date	fare	ride_id
2	Lake Jonathanshire	1/14/19 10:14	13.83	5.74E+12
3	South Michelleport	3/4/19 18:24	30.24	2.34E+12
4	Port Samanthamouth	2/24/19 4:29	33.44	2.01E+12
5	Rodneyfort	2/10/19 23:22	23.44	5.15E+12
6	South Jack	3/6/19 4:28	34.58	3.91E+12
7	South Latoya	3/11/19 12:26	9.52	2.00E+12
8	New Paulville	2/27/19 11:17	43.25	7.93E+11
9	Simpsonburgh	4/26/19 0:43	35.98	1.12E+11
10	South Karenland	1/8/19 3:28	35.09	8.00E+12
11	North Jasmine	3/9/19 6:26	42.81	5.33E+12

And if we go to the end of the file, we can see that there are 2,376 rows:

2370	Lake Jamie	4/29/19 1:58	54.22	2.49E+12
2371	Bradshawfurt	1/30/19 10:55	51.39	1.33E+12

2372	Michaelberg	4/29/19 17:04	13.38	8.55E+12
2373	Lake Latoyabury	1/30/19 0:05	20.76	9.02E+12
2374	North Jaime	2/10/19 21:03	11.11	2.78E+12
2375	West Heather	5/7/19 19:22	44.94	4.26E+12
2376	Newtonview	4/25/19 10:20	55.84	9.99E+12

Looking more closely at this file, we notice that each column has a header. Each row contains a city that has a date when the ride was taken, the fare for the ride, and a 12-to-13 digit ride identification number.

There's no way to scroll through the `ride_data.csv` file efficiently, so we'll have to use Pandas to determine if there are empty rows and the data type for each column.

5.2.2: Overview of the Project

You may only be in your second week at PyBer, but there's one thing you've learned along the way that you know applies here too: if you fail to plan, you plan to fail.

So before you dive into these massive datasets, you sit down and scope out the analytical project to make sure you and Omar are on the same page about the final deliverables.

It would be foolish to start messing around with such a large dataset without knowing exactly what you're trying to get out of it. So you and Omar have a head-to-head to clarify the process and outcomes for this project. You come up with the following list of steps and deliverables:

- Import your data into a Pandas DataFrame.
- Merge your DataFrames.
- Create a bubble chart that showcases the average fare versus the total number of rides with bubble size based on the total number of drivers for each city type, including urban, suburban, and rural.
- Create a bubble chart that showcases the average fare versus the total number of rides with bubble size based on the total number of drivers for three city types.
- Determine the mean, median, and mode for the following:
    - The total number of rides for each city type.
    - The average fares for each city type.
    - The total number of drivers for each city type.

- Create box-and-whisker plots that visualize each of the following to determine if there are any outliers:
    - The number of rides for each city type.
    - The fares for each city type.
    - The number of drivers for each city type.
- Create a pie chart that visualizes each of the following data for each city type:
    - The percent of total fares.
    - The percent of total rides.
    - The percent of total drivers.

Omar has approved the project scope. It’s time to get to work!

5.2.3: Load and Read the CSV files

With the project scope approved, you're ready to kick off your analysis by making a new Jupyter Notebook file, loading the CSV files, and inspecting them to make sure you don't need to clean the data or change column names before (finally!) merging the two datasets.

Let's get started on the analysis! First, we need to open Jupyter Notebook and create a file for the project, and then we'll import the required libraries and load the data.

Open Jupyter Notebook on macOS

> **Rewind**
>
> On a Mac, to open Jupyter Notebook in the PyBer_Analysis folder:
>
> 1. On the command line, navigate to the PyBer_Analysis folder and activate the PythonData environment.
> 2. Type `jupyter notebook`.

Open Jupyter Notebook on Windows

> **Rewind**
>
> In Windows, to open Jupyter Notebook in the PyBer_Analysis folder:
>
> 1. Open the PythonData Anaconda Prompt for the PythonData environment.
> 2. Type `jupyter notebook`.

Next, create a new Jupyter Notebook file in your PyBer_Analysis folder and name it `PyBer`.

Load the CSV files

In the first cell, add the following code to import the Pandas and Matplotlib libraries with the Pyplot module, and run the cell:

```
# Add Matplotlib inline magic command
%matplotlib inline
# Dependencies and Setup
import matplotlib.pyplot as plt
import pandas as pd
```

In a new cell, declare variables that connect to the CSV files in the Resources folder:

```
# Files to load
city_data_to_load = "Resources/city_data.csv"
ride_data_to_load = "Resources/ride_data.csv"
```

> **Rewind**
>
> If you want to use `os.path.join()` to load CSV files, you need to import the `os` module with your dependencies, like this:
>
> `import os`

Next, we will read each CSV file in Pandas.

Read the City Data File

How do you read a CSV file into a DataFrame?

- ○ `pd.to_csv()`
- ○ `pd.open_csv()`
- ○ `pd.read_csv()`

To read a CSV file into Pandas, we use `pd.read_csv`. Add the following code to a new cell:

```
# Read the city data file and store it in a pandas DataFrame.
city_data_df = pd.read_csv(city_data_to_load)
city_data_df.head(10)
```

When you run the cell, the first 10 rows of your city data should look something like this:

	city	driver_count	type
0	Richardfort	38	Urban
1	Williamsstad	59	Urban
2	Port Angela	67	Urban
3	Rodneyfort	34	Urban
4	West Robert	39	Urban
5	West Anthony	70	Urban
6	West Angela	48	Urban
7	Martinezhaven	25	Urban
8	Karenberg	22	Urban
9	Barajasview	26	Urban

Read the Ride Data File

To load the `ride_data.csv` file into a Pandas DataFrame, add the following code to a new cell:

```
# Read the ride data file and store it in a pandas DataFrame.
ride_data_df = pd.read_csv(ride_data_to_load)
ride_data_df.head(10)
```

When you run the cell, the first 10 rows of the ride data should look something like this:

	city	date	fare	ride_id

0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873
1	South Michelleport	2019-03-04 18:24:09	30.24	2343912425577
2	Port Samanthamouth	2019-02-24 04:29:00	33.44	2005065760003
3	Rodneyfort	2019-02-10 23:22:03	23.44	5149245426178
4	South Jack	2019-03-06 04:28:35	34.58	3908451377344
5	South Latoya	2019-03-11 12:26:48	9.52	1994999424437
6	New Paulville	2019-02-27 11:17:56	43.25	793208410091
7	Simpsonburgh	2019-04-26 00:43:24	35.98	111953927754
8	South Karenland	2019-01-08 03:28:48	35.09	7995623208694
9	North Jasmine	2019-03-09 06:26:29	42.81	5327642267789

5.2.4: Explore the Data in Pandas

Often you'll be given more than one CSV file to work with and be asked to merge them to perform the analysis. Before you merge them, you should explore the data.

Missing, malformed, or incorrect data could lead to a poor or incorrect analysis, and the last thing we want is to be standing in front of the CEO with faulty information!

We'll start off by inspecting each DataFrame. When we're satisfied that there is no missing, malformed, or incorrect data, we'll merge the DataFrames.

Inspect the City Data DataFrame

For the `city_data_df` DataFrame, we need to:

1. Get all the rows that contain null values.
2. Make sure the driver_count column has an integer data type.
3. Find out how many data points there are for each type of city.

First, let's get all the rows that are not null.

> **Rewind**
> To get the name of each column and the number of rows that are not null, we can use the `df.count()` method.

> Another option is to use `df.isnull().sum()` method chaining.

We'll use the `df.count()` method to find the names of our columns and the number of rows that are not null.

Add the following code to a new cell:

```
# Get the columns and the rows that are not null.
city_data_df.count()
```

When you run the cell, the output shows 120 rows for each column:

> ```
> city_data_df.count()
> ```
>
> ```
> city           120
> driver_count   120
> type           120
> dtype: int64
> ```

And to make sure there are no null values, we can type and run the following code:

```
# Get the columns and the rows that are not null.
city_data_df.isnull().sum()
```

The output from this code shows that there are zero null values in all three columns.

Next, we need to see if the driver_count column has a numerical data type because we plan to perform mathematical calculations on that column.

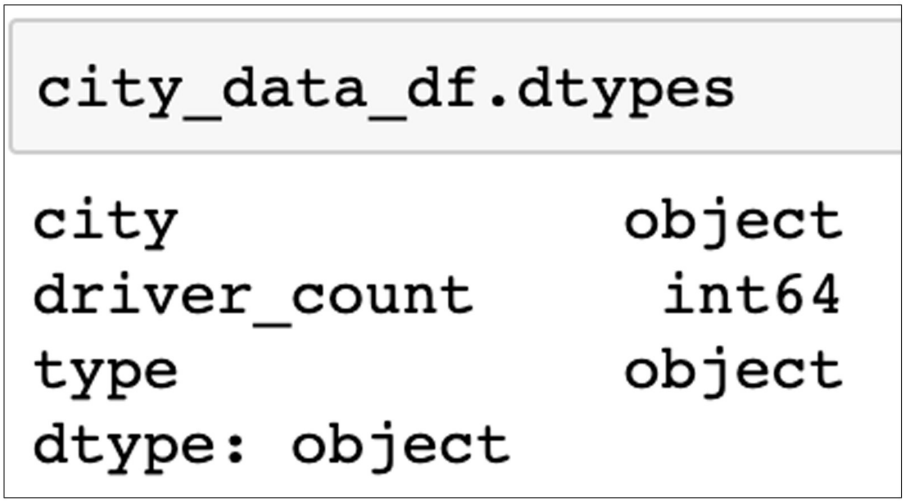
>

> **Rewind**
>
> To get the data types of each column, we use the `dtypes` on the DataFrame.

Add the following code to a new cell:

```
# Get the data types of each column.
city_data_df.dtypes
```

Run the cell, and you'll see that the output shows the data types for each column. Most importantly, the data type for the driver_count column is an integer:



Finally, we'll check to see how many data points there are for each type of city. To do this, we'll use the `sum()` method on the `city_data_df` for the type column where the condition equals each city in the DataFrame.

> **Rewind**
>
> We can use the `unique()` method on a specific column, which will return an array, or list, of all the unique values of that column

Add the following code to a new cell:

```
# Get the unique values of the type of city.
city_data_df["type"].unique()
```

Run the cell, and you'll see that the output will be an array of different types of cities.

```
# Get the unique values of the type of city.
city_data_df["type"].unique()
```

```
array(['Urban', 'Suburban', 'Rural'], dtype=object)
```

Now we can use the `sum()` method on the `city_data_df` for the type column where the condition equals either Urban, Suburban, or Rural.

To get the number of data points for the Urban cities, add the following code to a new cell:

```
# Get the number of data points from the Urban cities.
sum(city_data_df["type"]=="Urban")
```

When you run this cell, the output will show that the number of data points for the Urban cities is 66.

```
# Get the number of data points from the Urban cities.
sum(city_data_df["type"]=="Urban")
```

```
66
```

What are the number of data points for the Suburban and Rural cities?

- ○ Suburban:18, Rural:36
- ○ Suburban:36, Rural:66
- ○ Suburban:36, Rural:18

Check Answer

Finish ▸

Inspect Ride Data DataFrame

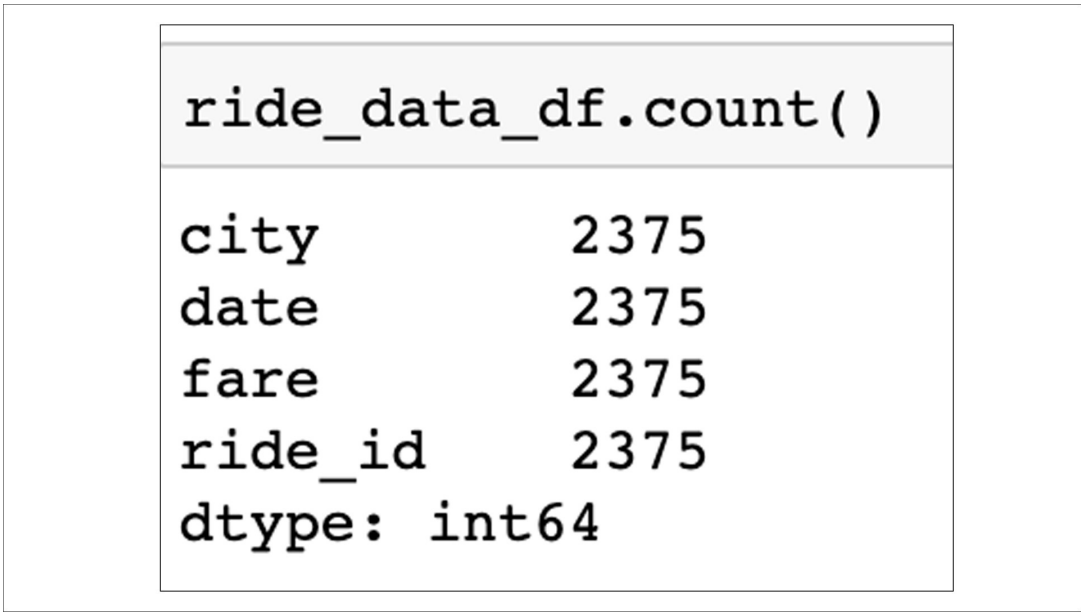
For the `ride_data_df` DataFrame, we need to:

1. Get all the rows that contain null values.
2. Make sure the fare and ride_id columns are numerical data types.

First, let's get all the rows that are not null. Add the following code:

```
# Get the columns and the rows that are not null.
ride_data_df.count()
```

When you run the cell, you'll see that the output is 2,375 rows for each column, as shown here:



And to make sure there are no null values, we can type and run the following code:

```
# Get the columns and the rows that are not null.
ride_data_df.isnull().sum()
```

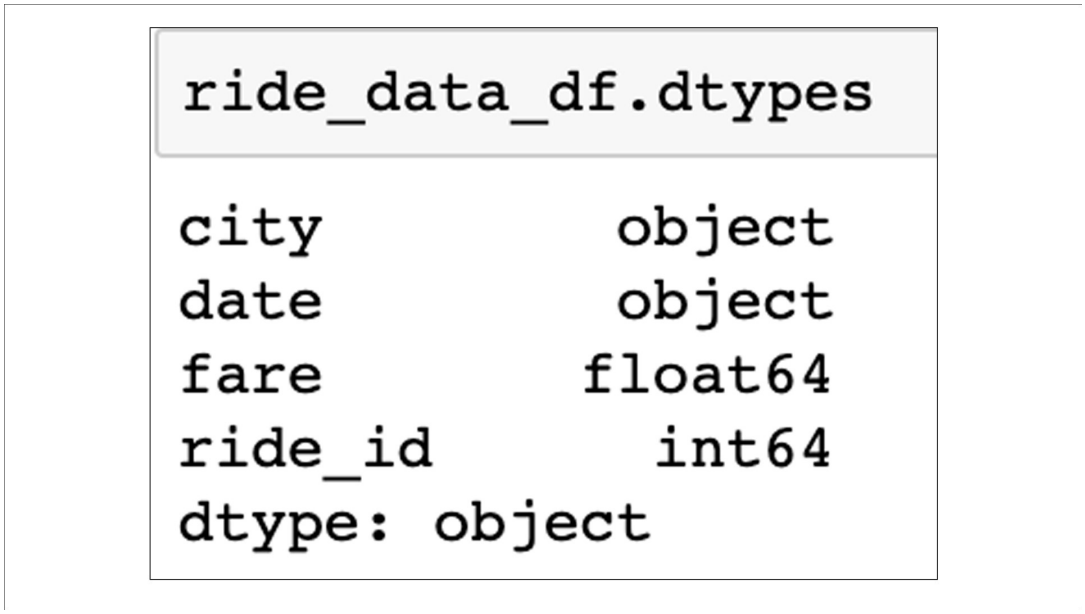
The output from this code shows that there are zero null values in all three columns.

Next, we need to determine if the fare and ride_id columns are numerical data types so that we can perform mathematical calculations on those columns.

Add the following code to a new cell:

```
# Get the data types of each column.
ride_data_df.dtypes
```

Run the cell. The output shows the data types for each column. And more importantly, the data types for the fare and ride_id columns are floating-point decimal, and integer, respectively.



Both of the DataFrames look good and can now be merged.

Merge DataFrames

Before we merge the DataFrames, let's review each DataFrame.

The columns in the `city_data_df` DataFrame are:

- city
- driver_count
- type

The columns in the `ride_data_df` are:

- city
- date
- fare
- ride_id

> **Rewind**
> When we merge two DataFrames, we merge on a column with the same data, and the same column name, in both DataFrames. We use the following syntax to do that:
>
> `new_df = pd.merge(leftdf, rightdf, on=["column_leftdf", "column_rightdf"])`
>
> We may have to merge the DataFrames using the `how=` parameter either left, right, inner, or outer depending how we want to merge the DataFrames. The default is inner.

Looking at the columns in the two DataFrames, we can see that the column the DataFrames have in common is `city`. Therefore, we will merge the two DataFrames on the `city` column, and then add the `city_data_df` to the end of the `ride_data_df` DataFrame with the constraint `how="left"`.

Add the following code to a new cell and run the cell to merge the two DataFrames.

```
# Combine the data into a single dataset
pyber_data_df = pd.merge(ride_data_df, city_data_df, how="left", on=["city",

# Display the DataFrame
pyber_data_df.head()
```

The merged DataFrame, `pyber_data_df`, should look like this:

	city	date	fare	ride_id	driver_count	type
0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873	5	Urban

0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873
1	South Michelleport	2019-03-04 18:24:09	30.24	2343912425577
2	Port Samanthamouth	2019-02-24 04:29:00	33.44	2005065760003
3	Rodneyfort	2019-02-10 23:22:03	23.44	5149245426178
4	South Jack	2019-03-06 04:28:35	34.58	3908451377344
5	South Latoya	2019-03-11 12:26:48	9.52	1994999424437
6	New Paulville	2019-02-27 11:17:56	43.25	793208410091
7	Simpsonburgh	2019-04-26 00:43:24	35.98	111953927754
8	South Karenland	2019-01-08 03:28:48	35.09	7995623208694
9	North Jasmine	2019-03-09 06:26:29	42.81	5327642267789

5.2.4: Explore the Data in Pandas

Often you'll be given more than one CSV file to work with and be asked to merge them to perform the analysis. Before you merge them, you should explore the data.

Missing, malformed, or incorrect data could lead to a poor or incorrect analysis, and the last thing we want is to be standing in front of the CEO with faulty information!

We'll start off by inspecting each DataFrame. When we're satisfied that there is no missing, malformed, or incorrect data, we'll merge the DataFrames.

Inspect the City Data DataFrame

For the `city_data_df` DataFrame, we need to:

1. Get all the rows that contain null values.
2. Make sure the driver_count column has an integer data type.
3. Find out how many data points there are for each type of city.

First, let's get all the rows that are not null.

> **Rewind**
> To get the name of each column and the number of rows that are not null, we can use the `df.count()` method.

> Another option is to use `df.isnull().sum()` method chaining.

We'll use the `df.count()` method to find the names of our columns and the number of rows that are not null.

Add the following code to a new cell:

```
# Get the columns and the rows that are not null.
city_data_df.count()
```

When you run the cell, the output shows 120 rows for each column:

> ```
> city_data_df.count()
> ```
>
> ```
> city           120
> driver_count   120
> type           120
> dtype: int64
> ```

And to make sure there are no null values, we can type and run the following code:

```
# Get the columns and the rows that are not null.
city_data_df.isnull().sum()
```

The output from this code shows that there are zero null values in all three columns.

Next, we need to see if the driver_count column has a numerical data type because we plan to perform mathematical calculations on that column.

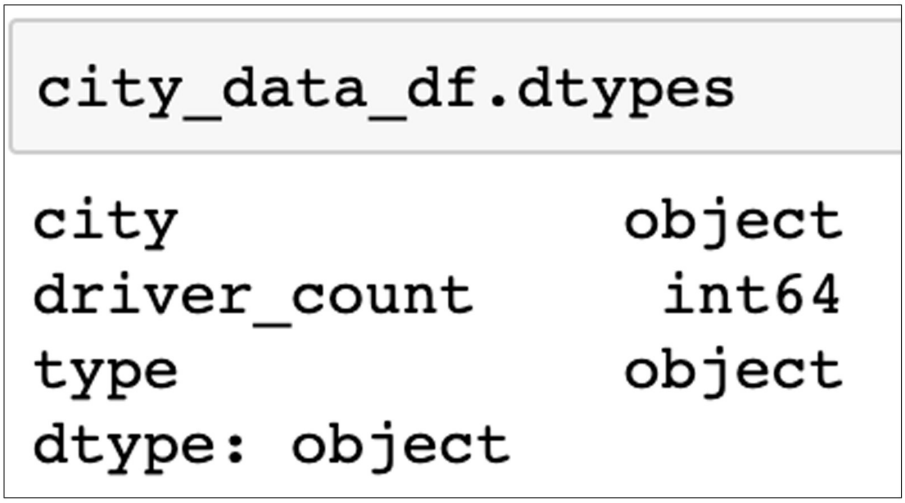
>

> **Rewind**
>
> To get the data types of each column, we use the `dtypes` on the DataFrame.

Add the following code to a new cell:

```
# Get the data types of each column.
city_data_df.dtypes
```

Run the cell, and you'll see that the output shows the data types for each column. Most importantly, the data type for the driver_count column is an integer:



Finally, we'll check to see how many data points there are for each type of city. To do this, we'll use the `sum()` method on the `city_data_df` for the type column where the condition equals each city in the DataFrame.

> **Rewind**
>
> We can use the `unique()` method on a specific column, which will return an array, or list, of all the unique values of that column

Add the following code to a new cell:

```
# Get the unique values of the type of city.
city_data_df["type"].unique()
```

Run the cell, and you'll see that the output will be an array of different types of cities.

```
# Get the unique values of the type of city.
city_data_df["type"].unique()
```

```
array(['Urban', 'Suburban', 'Rural'], dtype=object)
```

Now we can use the `sum()` method on the `city_data_df` for the type column where the condition equals either Urban, Suburban, or Rural.

To get the number of data points for the Urban cities, add the following code to a new cell:

```
# Get the number of data points from the Urban cities.
sum(city_data_df["type"]=="Urban")
```

When you run this cell, the output will show that the number of data points for the Urban cities is 66.

```
# Get the number of data points from the Urban cities.
sum(city_data_df["type"]=="Urban")
```

```
66
```

What are the number of data points for the Suburban and Rural cities?

- ○ Suburban:18, Rural:36
- ○ Suburban:36, Rural:66
- ○ Suburban:36, Rural:18

Check Answer

Finish ▸

Inspect Ride Data DataFrame

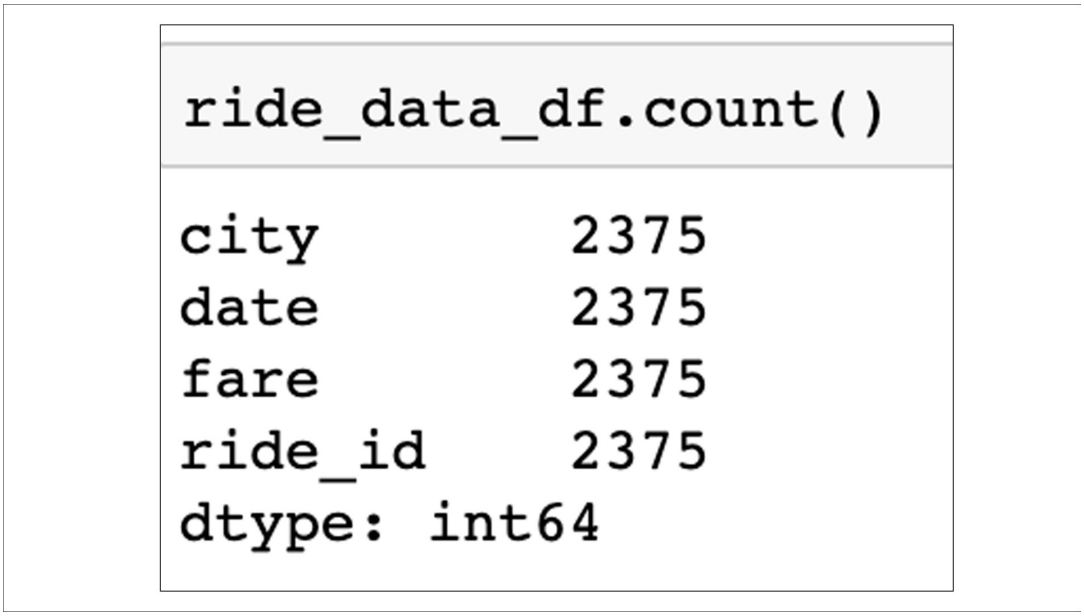
For the `ride_data_df` DataFrame, we need to:

1. Get all the rows that contain null values.
2. Make sure the fare and ride_id columns are numerical data types.

First, let's get all the rows that are not null. Add the following code:

```
# Get the columns and the rows that are not null.
ride_data_df.count()
```

When you run the cell, you'll see that the output is 2,375 rows for each column, as shown here:



And to make sure there are no null values, we can type and run the following code:

```
# Get the columns and the rows that are not null.
ride_data_df.isnull().sum()
```

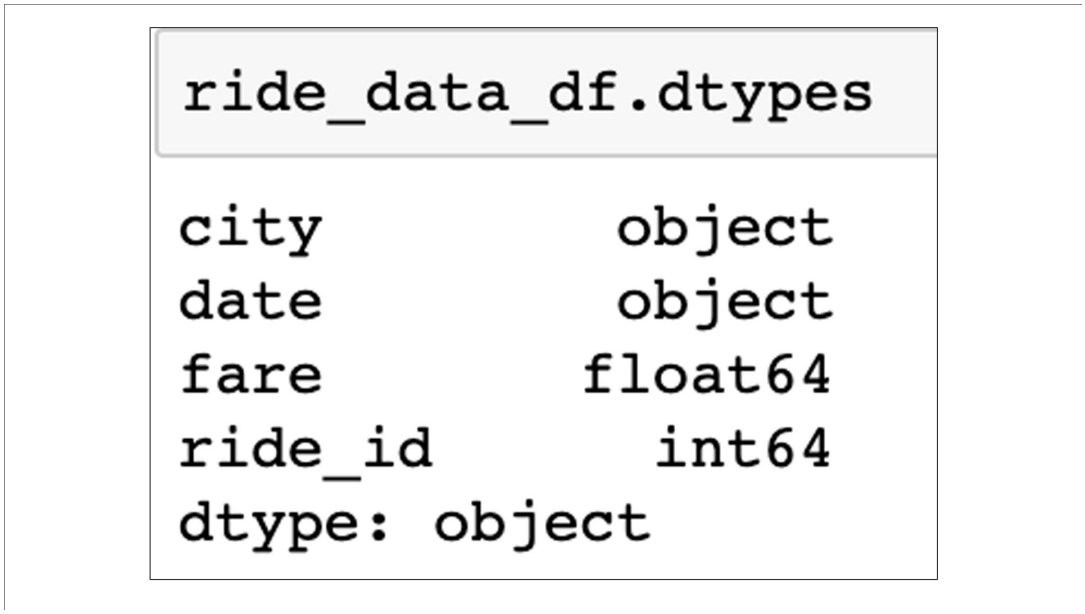
The output from this code shows that there are zero null values in all three columns.

Next, we need to determine if the fare and ride_id columns are numerical data types so that we can perform mathematical calculations on those columns.

Add the following code to a new cell:

```
# Get the data types of each column.
ride_data_df.dtypes
```

Run the cell. The output shows the data types for each column. And more importantly, the data types for the fare and ride_id columns are floating-point decimal, and integer, respectively.



Both of the DataFrames look good and can now be merged.

Merge DataFrames

Before we merge the DataFrames, let's review each DataFrame.

The columns in the `city_data_df` DataFrame are:

- city
- driver_count
- type

The columns in the `ride_data_df` are:

- city
- date
- fare
- ride_id

> **Rewind**
> When we merge two DataFrames, we merge on a column with the same data, and the same column name, in both DataFrames. We use the following syntax to do that:
>
> `new_df = pd.merge(leftdf, rightdf, on=["column_leftdf", "column_rightdf"])`
>
> We may have to merge the DataFrames using the `how=` parameter either left, right, inner, or outer depending how we want to merge the DataFrames. The default is inner.

Looking at the columns in the two DataFrames, we can see that the column the DataFrames have in common is `city`. Therefore, we will merge the two DataFrames on the `city` column, and then add the `city_data_df` to the end of the `ride_data_df` DataFrame with the constraint `how="left"`.

Add the following code to a new cell and run the cell to merge the two DataFrames.

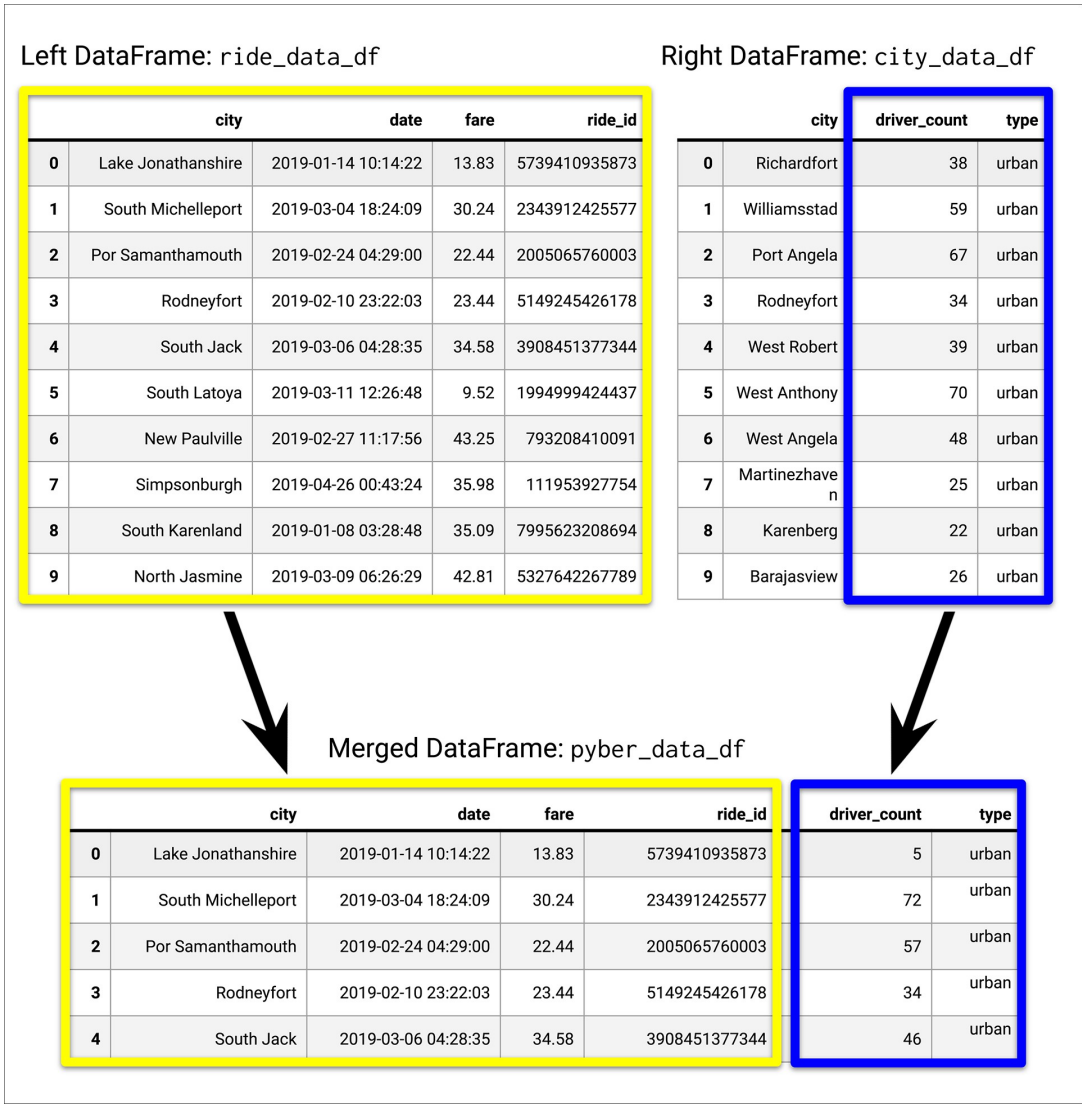
```
# Combine the data into a single dataset
pyber_data_df = pd.merge(ride_data_df, city_data_df, how="left", on=["city",

# Display the DataFrame
pyber_data_df.head()
```

The merged DataFrame, `pyber_data_df`, should look like this:

	city	date	fare	ride_id	driver_count	type
0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873	5	Urban

1	South Michelleport	2019-03-04 18:24:09	30.24	2343912425577	72	Urban
2	Port Samanthamouth	2019-02-24 04:29:00	33.44	2005065760003	57	Urban
3	Rodneyfort	2019-02-10 23:22:03	23.44	5149245426178	34	Urban
4	South Jack	2019-03-06 04:28:35	34.58	3908451377344	46	Urban

In the `pyber_data_df` DataFrame, all the columns from the `city_data_df` are the first four columns after the index. The driver_count and type columns from the `ride_data_df` are added at the end, as shown in the following image:



	city	date	fare	ride_id
0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873
1	South Michelleport	2019-03-04 18:24:09	30.24	2343912425577
2	Por Samanthamouth	2019-02-24 04:29:00	22.44	2005065760003
3	Rodneyfort	2019-02-10 23:22:03	23.44	5149245426178
4	South Jack	2019-03-06 04:28:35	34.58	3908451377344
5	South Latoya	2019-03-11 12:26:48	9.52	1994999424437
6	New Paulville	2019-02-27 11:17:56	43.25	793208410091
7	Simpsonburgh	2019-04-26 00:43:24	35.98	111953927754
8	South Karenland	2019-01-08 03:28:48	35.09	7995623208694
9	North Jasmine	2019-03-09 06:26:29	42.81	5327642267789

	city	driver_count	type
0	Richardfort	38	urban
1	Williamsstad	59	urban
2	Port Angela	67	urban
3	Rodneyfort	34	urban
4	West Robert	39	urban
5	West Anthony	70	urban
6	West Angela	48	urban
7	Martinezhaven	25	urban
8	Karenberg	22	urban
9	Barajasview	26	urban

	city	date	fare	ride_id	driver_count	type
0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873	5	urban
1	South Michelleport	2019-03-04 18:24:09	30.24	2343912425577	72	urban
2	Por Samanthamouth	2019-02-24 04:29:00	22.44	2005065760003	57	urban
3	Rodneyfort	2019-02-10 23:22:03	23.44	5149245426178	34	urban
4	South Jack	2019-03-06 04:28:35	34.58	3908451377344	46	urban

5.2.5: Commit Your Code

Congratulations! Your datasets are merged, now you have a clear understanding of the information you have to work with. And it's only 10:00 pm!

Before signing off for the night, take a moment to commit your code to GitHub. After all, committing early and often has saved your progress on projects multiple times before, and the stakes are high on this analysis. Plus, Omar said he might have time to look over your work early tomorrow, so you want him to have access to the most up-to-date files.

Now that you have read the CSV files into DataFrames, inspected the DataFrames, and merged the DataFrames, save `PyBer.ipynb` to your PyBer_Analysis folder on your computer. Next, save the `PyBer.ipynb` file to the GitHub repository. Follow the steps for your operating system.

Commit Your Code on macOS

1. Launch the command line.
2. Navigate to your PyBer_Analysis folder using the necessary commands.
3. Check the status using `git status` and press Enter. You might see the following in your command line:
    - The `PyBer.ipynb` file
    - The Resources folder and everything in it
    - The `matplotlib_practice.ipynb` file

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
 (use "git add <file>..." to include in what will be committed)


PyBer.ipynb
   /Resources
   matplotlib_practice.ipynb

nothing added to commit but untracked files present (use "git add" to tr
```

4. Type `git add PyBer.ipynb Resources/` to add the `PyBer.ipynb` file and the CSV files and press Enter. You might see something like this:

```
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)


new file:  PyCitySchools.ipynb
   new file:  Resources/PyBer_ride_data.csv
   new file:  Resources/city_data.csv
   new file:  Resources/ride_data.csv

Untracked files:
 (use "git add <file>..." to include in what will be committed)

   matplotlib_practice.ipynb
```

5. Commit the files to be added to the repository by typing `git commit -m "Adding PyBer.ipynb file and Resources folder"` and press Enter. The output should look similar to this:

```
[master 73b8150] Adding PyBer.ipynb file and Resources folder.
 4 files changed, 3277 insertions(+)
 create mode 100644 PyBer.ipynb
```

```
 create mode 100644 Resources/PyBer_ride_data.csv
 create mode 100644 Resources/city_data.csv
 create mode 100644 Resources/ride_data.csv
```

6. Add the file to your repository using `git push` and press Enter. The output should look similar to this:

```
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (7/7), 48.47 KiB | 9.69 MiB/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/<your_GitHub_account>/PyBer_Analysis.git
  635d64c..73b8150 master -> master
```

7. Refresh your GitHub page to see the changes to your repository.

Commit Your Code on Windows

1. Launch Git Bash.
2. Navigate to the PyBer_Analysis folder using the necessary commands.
3. Check the status by typing `git status` and pressing Enter. You might see something like this in your bash command line:
    - The `PyBer.ipynb` file
    - The Resources folder and everything in it
    - The `matplotlib_practice.ipynb` file

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
 (use "git add <file>..." to include in what will be committed)


PyBer.ipynb
   /Resources
   matplotlib_practice.ipynb
```

```
nothing added to commit but untracked files present (use "git add" to tr
```

4. Type `git add PyBer.ipynb Resources/` to add the `PyBer.ipynb file` and the CSV files and press Enter. You might see something like this:

```
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
 (use "git reset HEAD <file>..." to unstage)


new file:  PyCitySchools.ipynb
   new file:  Resources/PyBer_ride_data.csv
   new file:  Resources/city_data.csv
   new file:  Resources/ride_data.csv

Untracked files:
 (use "git add <file>..." to include in what will be committed)

   matplotlib_practice.ipynb
```

5. Commit the files to be added to the repository by typing `git commit -m "Adding PyBer.ipynb file and Resources folder."` and press Enter. Your output should look similar to this:

```
[master 73b8150] Adding PyBer.ipynb file and Resources folder.
 4 files changed, 3277 insertions(+)
 create mode 100644 PyBer.ipynb
 create mode 100644 Resources/PyBer_ride_data.csv
 create mode 100644 Resources/city_data.csv
 create mode 100644 Resources/ride_data.csv
```

6. Add the file to your repository by typing `git push` and press Enter. Your output should look similar to this:

```
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
```

```
Writing objects: 100% (7/7), 48.47 KiB | 9.69 MiB/s, done.
Total 7 (delta 0), reused 0 (delta 0)
To https://github.com/<your_GitHub_account>/PyBer_Analysis.git
  635d64c..73b8150 master -> master
```

7. Refresh your GitHub page to see the changes to your repository.

5.3.1: Create DataFrames for Each Type of City

After working late last night, you get into the office bright and early and check your email. You have an email from Omar!

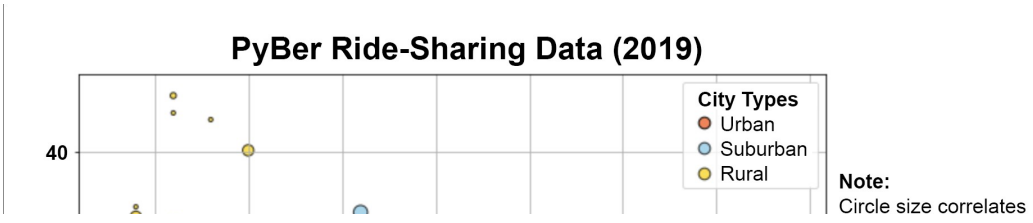
“Thanks for committing your code. I had a chance to take a look and I have to say, nice work! It looks like you are ready to dig into that scatter plot we discussed. I have some time this afternoon if you’d like to go over it!”

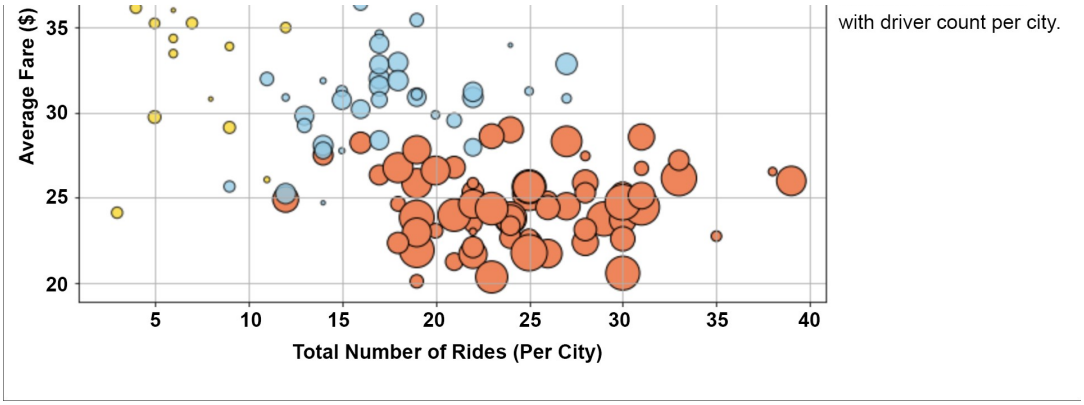
Energized and encouraged, you set up your workspace for the day and dig back in. Bubble charts, here we come!

Omar has asked you to create a bubble chart that showcases the average fare versus the total number of rides with bubble size based on the average number of drivers for each city type: urban, suburban, and rural.

> **Note**
> The terms **scatter plot** and **scatter chart**, as well as **bubble chart** and **bubble plot**, are often used interchangeably in the data visualization field.

When you’re done, the bubble chart should look similar to this:





with driver count per city.

For the bubble chart, we will need to plot the following:

- The average fare for each type of city on the y-axis
- The total number of rides for each type city on the x-axis
- Make the size of each marker, or bubble, correlate to the average number of drivers for each type of city

Here are the steps to create a bubble chart:

1. To get the average fare, we can use the Pandas `mean()` method for each city in the "fare" column.
2. To get the total number of rides for each city, we can use the Pandas `count()` method for each city in the "ride_id" column.
3. To make the bubble size correlate to the number of drivers for each city, we can get the average `driver_count` for each city.

Completing the steps will be more efficient if we create separate DataFrames for each type and then create Data Series for each step.

Let's create three separate DataFrames, one for each type of city.

Each new DataFrame can be created by filtering the `pyber_data_df` DataFrame where the city type is equal to Urban, Suburban, or Rural.

Add the following code in a new cell to create the Urban cities DataFrame:

```
# Create the Urban city DataFrame.
urban_cities_df = pyber_data_df[pyber_data_df["type"] == "Urban"]
urban_cities_df.head()
```

When you run this cell, you'll get a DataFrame that contains only the cities with the type equal to Urban, as you can see in this snapshot of the first five rows of the output:

	city	date	fare	ride_id	driver_count	type
0	Lake Jonathanshire	2019-01-14 10:14:22	13.83	5739410935873	5	Urban
1	South Michelleport	2019-03-04 18:24:09	30.24	2343912425577	72	Urban
2	Port Samanthamouth	2019-02-24 04:29:00	33.44	2005065760003	57	Urban
3	Rodneyfort	2019-02-10 23:22:03	23.44	5149245426178	34	Urban
4	South Jack	2019-03-06 04:28:35	34.58	3908451377344	46	Urban

Now add the following code in a new cell and run the cell to create the `suburban_cities_df` and the `rural_cities_df` DataFrames:

```
# Create the Suburban and Rural city DataFrames.
suburban_cities_df = pyber_data_df[pyber_data_df["type"] == "Suburban"]
rural_cities_df = pyber_data_df[pyber_data_df["type"] == "Rural"]
```

What is the first city that appears in the `rural_cities_df`?

- ○ Barronchester
- ○ Lake Latoyabury
- ○ Randallchester

Check Answer

Finish ▸

Great job on creating the DataFrames for each city type. Next, we'll get the total number of rides from each city type.

5.3.2: Get the Number of Rides for Each City Type

Looking at the x-axis title that was given to you, you see that you will need to get the number of rides for each city for each city type. Omar says there is a quick way to do this using Pandas. Thinking back to your previous experience using Pandas, your mind quickly does a rewind to the time you did something similar.

To get the number of rides for each city by each type of city, we have to create a Series where the index is the name of the city and the column is the number of rides for that city.

> **Rewind**
> To create a Data Series with one of the columns in a DataFrame, we can use the `groupby()` function and add the column inside the parentheses.
>
> Using the `groupby()` function can be used to group large amounts of data when we want to compute mathematical operations on these groups.

We'll use the `groupby()` function to create a Series of data that has the name of the city as the index, apply the `count()` method to the Series for each city, and select the ride_id column.

Add the following code to a new cell:

```
# Get the number of rides for urban cities.
urban_ride_count = urban_cities_df.groupby(["city"]).count()["ride_id"]
urban_ride_count.head()
```

Run the cell, and you'll see that the output is a Series with the number of rides shown for each city. Here's a snapshot of the first five rows of the output:

```
city
Amandaburgh          18
Barajasview          22
Carriemouth          27
Christopherfurt      27
Deanville            19
Name: ride_id, dtype: int64
```

Using the same approach, we can create the `suburban_ride_count` and the `rural_ride_count` Series. To do so, add the following code to a new cell and run the cell.

```
# Create the suburban and rural ride count.
suburban_ride_count = suburban_cities_df.groupby(["city"]).count()["ride_id"

rural_ride_count = rural_cities_df.groupby(["city"]).count()["ride_id"]
```

We now have one of the three datasets we need to create a bubble chart. Next, we'll use the `groupby()` function on the city type DataFrames to get the average fare for each city type. This will be our second dataset.

5.3.3: Get the Average Fare for Each City Type

Now that you have the rides parsed by city type, you look at the bubble chart Omar gave you and realize that you need to get the average city fare for each city type. This will allow you to provide further insight into the data.

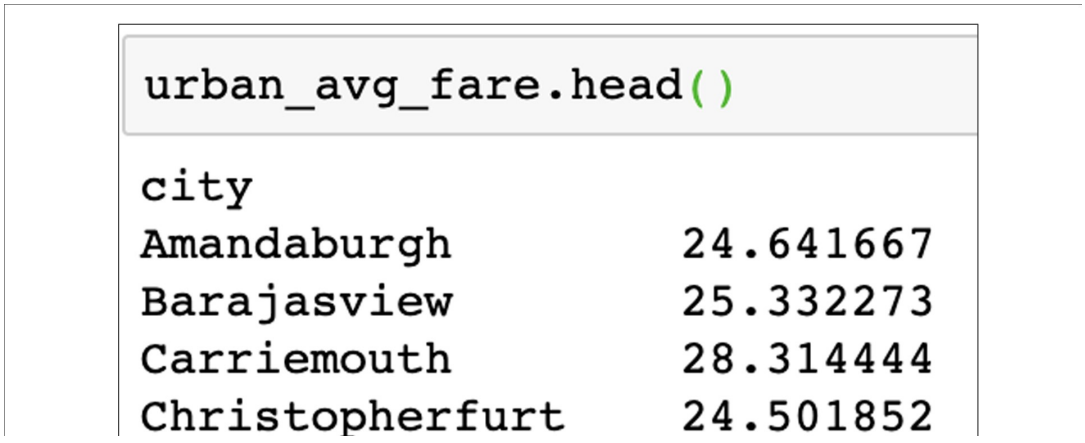
Using the separate DataFrames for each city type, we can calculate the average fare for each city in the urban, suburban, and rural cities.

Using the `groupby()` function, we can chain the `mean()` method to get the averages of the fare column.

Add the following code to a new cell and run the cell.

```
# Get average fare for each city in the urban cities.
urban_avg_fare = urban_cities_df.groupby(["city"]).mean()["fare"]
urban_avg_fare.head()
```

The output after running the cell will be a Series with the average fare for each city in the urban cities. This snapshot shows the first five rows of output:



```
Deanville                 25.842632
Name: fare, dtype: float64
```

Using the same approach, we can calculate the average fare for suburban and rural cities. Add the following code to a new cell.

```
# Get average fare for each city in the suburban and rural cities.
suburban_avg_fare = suburban_cities_df.groupby(["city"]).mean()["fare"]
rural_avg_fare = rural_cities_df.groupby(["city"]).mean()["fare"]
```

Now we have two of three datasets we need to create a bubble chart. Next, we'll use the `groupby()` function on the city type DataFrames to get the average number of drivers for each city type. This will be our third and final dataset.

5.3.4: Get the Average Number of Drivers for Each City Type

With the number of rides and the average fare for each city type, you can create a simple scatter, but Omar said that V. Isualize wants to see how the rides and fare data stack are affected by the average number of drivers for each city type. This will help V. Isualize make key decisions about where resources and support are needed.

The last data point we need for our bubble chart is the average number of drivers for each city in the urban, suburban, and rural cities.

To get the average number of drivers for each city in the urban, suburban, and rural cities, we can use the `groupby()` function and get the `mean()` of the `driver_count` column.

Add the following code to a new cell and run the cell.

```
# Get the average number of drivers for each urban city.
urban_driver_count = urban_cities_df.groupby(["city"]).mean()["driver_count"
urban_driver_count.head()
```

After running the cell, the output will show a Series with the average number of drivers for each urban city, as shown in the following image.

```
Amandaburgh          12.0
Barajasview          26.0
Carriemouth          52.0
Christopherfurt      41.0
Deanville            49.0
Name: driver_count, dtype: float64
```

We can repurpose our code and change the variables to calculate the average number of drivers for suburban and rural cities.

Add the following code to a new cell and run the cell.

```
# Get the average number of drivers for each city for the suburban and rural
suburban_driver_count = suburban_cities_df.groupby(["city"]).mean()["driver_
rural_driver_count = rural_cities_df.groupby(["city"]).mean()["driver_count"
```

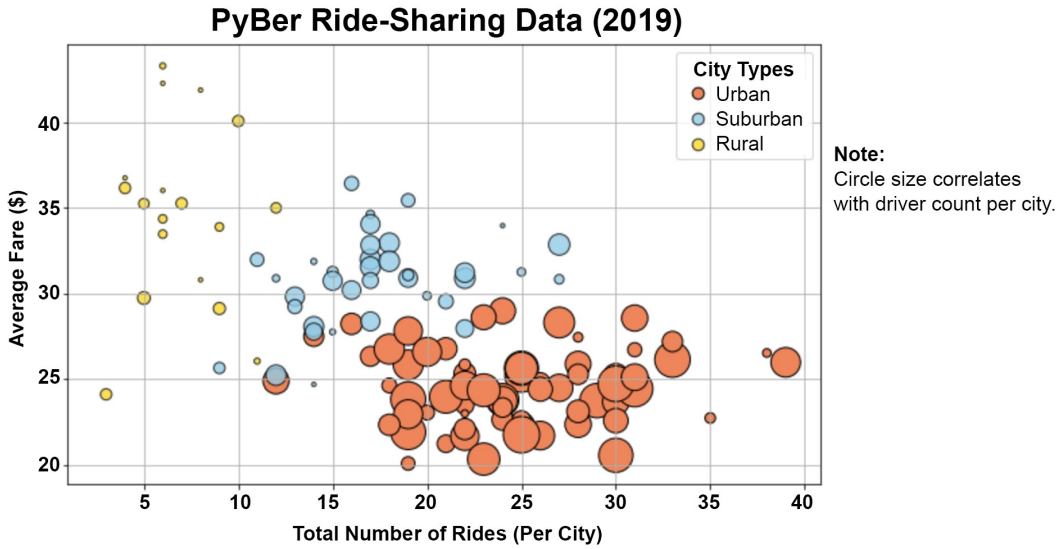
Now we have all our data and can begin to create our bubble chart!

5.3.5: Create Bubble Charts

As you get ready to create a bubble chart, you start thinking that it would be cool to use the company's color scheme for each type of city. You know V. Isualize came up with the color scheme herself back in the first few days of the company: gold for profitability, sky blue for strategy, and coral because she loves the ocean! You can use each color for a city type.

You know that it's details like this that will set your presentation apart. Early promotion, here you come!

If we look at the final product, we can see that the bubble chart contains three different scatter plots, one for each type of city:



Our first task will be to create a scatter plot for each type of city where the following conditions are met:

- The x-axis is the number of rides for each city.
- The y-axis is the average fare for each city.
- The size of each marker is the average number of drivers in each city.

Let's create each scatter plot individually and add them all to one chart.

The first scatter plot we'll make is for urban cities. We'll create our plots using the MATLAB method.

> **Rewind**
> To create a scatter plot using the MATLAB method, use the `plt.scatter()` function.

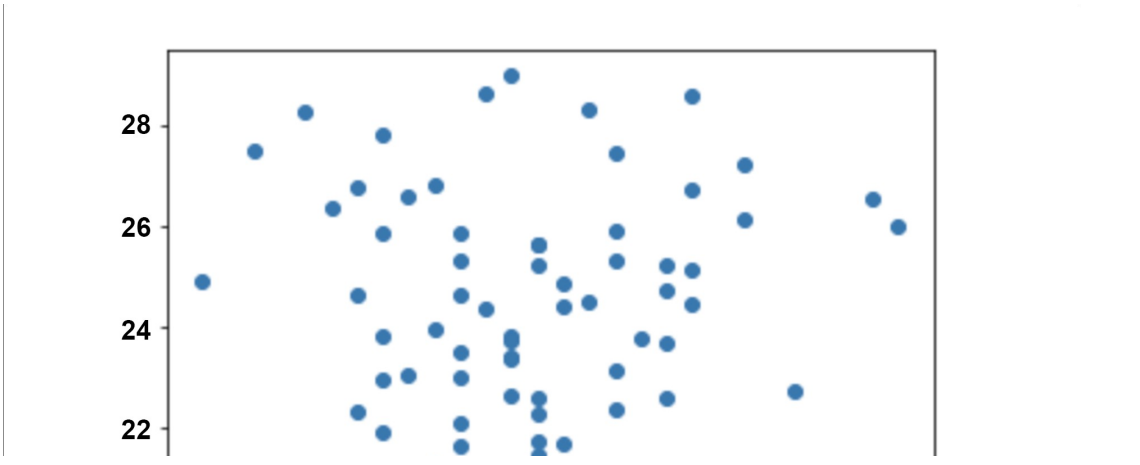
Create the Urban Cities Bubble Chart

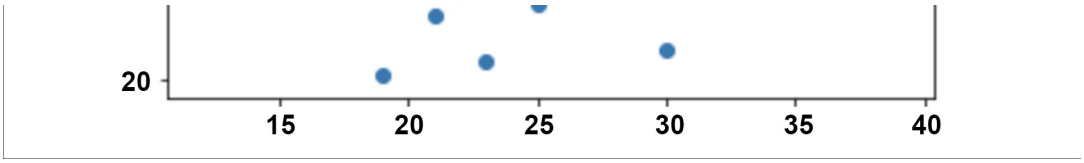
To the `plt.scatter()` function, let's add the x- and y-axis data, where the x-axis is the `urban_ride_count` and the y-axis is the `urban_avg_fare`.

Add the following code to a new cell:

```
# Build the scatter plots for urban cities.
plt.scatter(urban_ride_count, urban_avg_fare)
```

When you run the cell, you'll see the following urban cities scatter plot:





Using the knowledge we gained about adding features to the chart, let's add a title, axes labels, and a legend, and change the color and size of the markers.

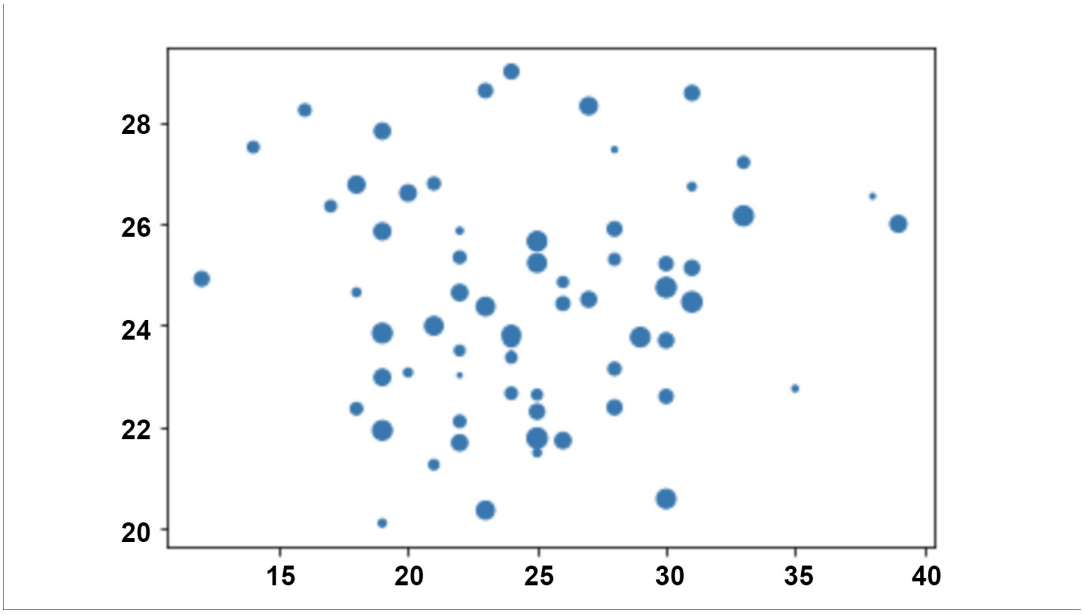
Let's start by editing the `plt.scatter(urban_ride_count, urban_avg_fare)` code to increase the size of the markers and add a label for the legend.

> **Rewind**
> To change the size of the markers, add the `s=` parameter to the `scatter()` function.

Go ahead and edit the existing code to look like this:

```
# Build the scatter plots for urban cities.
plt.scatter(urban_ride_count,
      urban_avg_fare,
      s=urban_driver_count)
```

When you run the cell, you'll see the following chart with markers of varying sizes:



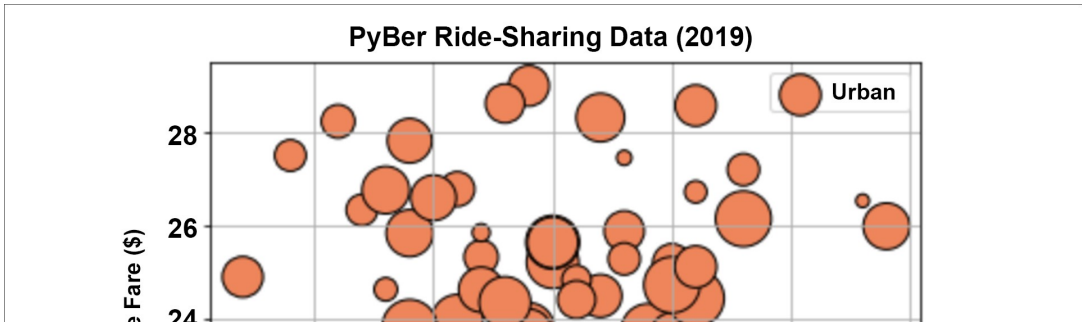
Some of the bubbles are still too small, so let's increase the size by a factor of 10 and add a black edge color to the circles that have a line width of 1. While we're changing these features, let's pay homage to the company's color scheme and make the urban markers coral and 20% transparent. We can also add a title, labels for the axes, a legend, and a grid.

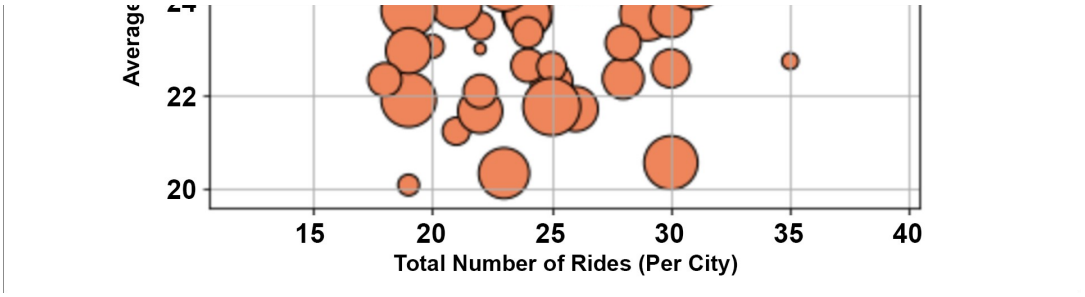
> **Rewind**
> To change the marker color, we add the `color=` parameter to the `scatter()` function.
>
> To add a title, x-axis and y-axis labels, and a legend, we use the `plt.title()`, `plt.ylabel()`, `plt.xlabel()`, and `plt.legend()` functions, respectively.

Add the following code to a new cell and run the cell.

```
# Build the scatter plots for urban cities.
plt.scatter(urban_ride_count,
      urban_avg_fare,
      s=10*urban_driver_count, c="coral",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Urban")
plt.title("PyBer Ride-Sharing Data (2019)")
plt.ylabel("Average Fare ($)")
plt.xlabel("Total Number of Rides (Per City)")
plt.grid(True)
# Add the legend.
plt.legend()
```

The chart in the output cell will have coral markers that vary in diameter based on the average number of drivers in each city, a title, and axes labels.





Create the Suburban Cities Bubble Chart

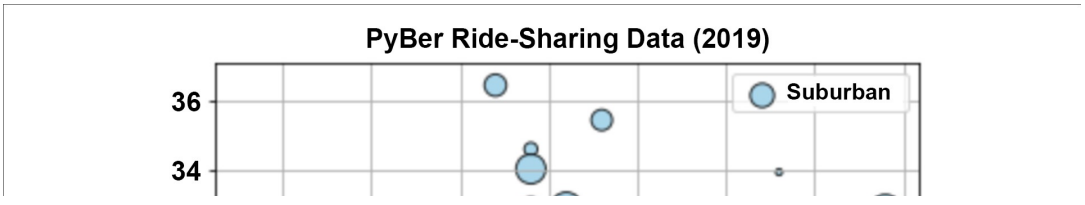
Now that we know what the final individual bubble chart should look like, we can repurpose the code and change some variables to create the suburban cities bubble chart.

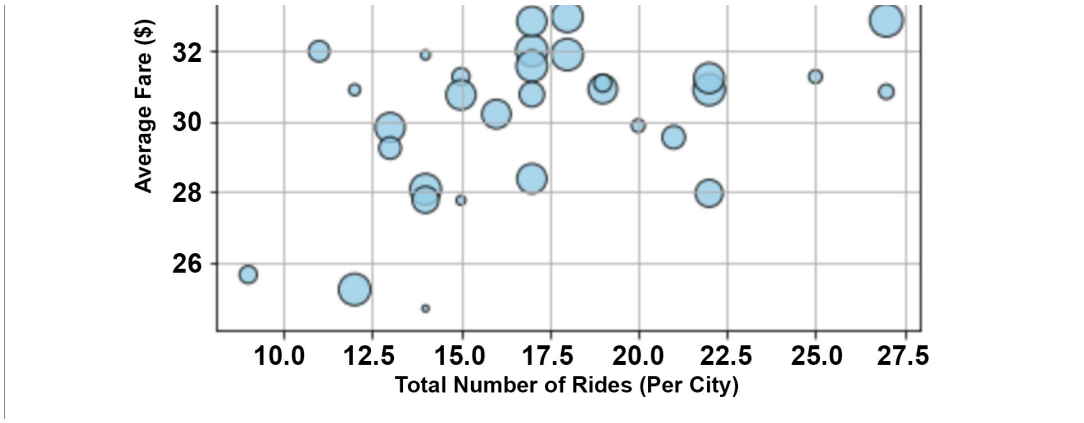
Let's create the same bubble chart as we did for the urban cities chart but change the color to sky blue.

Add the following code to a new cell and run the cell.

```
# Build the scatter plots for suburban cities.
plt.scatter(suburban_ride_count,
      suburban_avg_fare,
      s=10*suburban_driver_count, c="skyblue",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Suburban")
plt.title("PyBer Ride-Sharing Data (2019)")
plt.ylabel("Average Fare ($)")
plt.xlabel("Total Number of Rides (Per City)")
plt.grid(True)
# Add the legend.
plt.legend()
```

The chart in the output cell will have sky-blue markers that vary in diameter based on the average number of drivers in each city, a title, and axes labels.





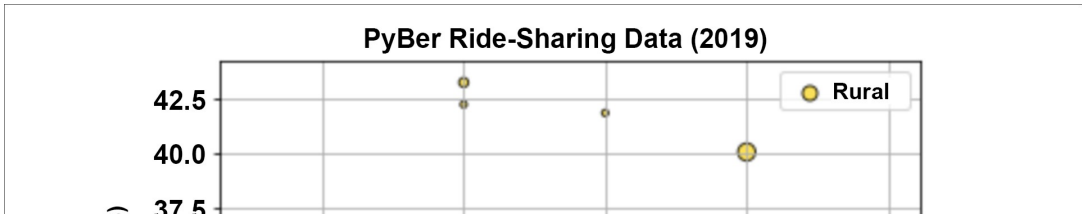
Create the Rural Cities Bubble Chart

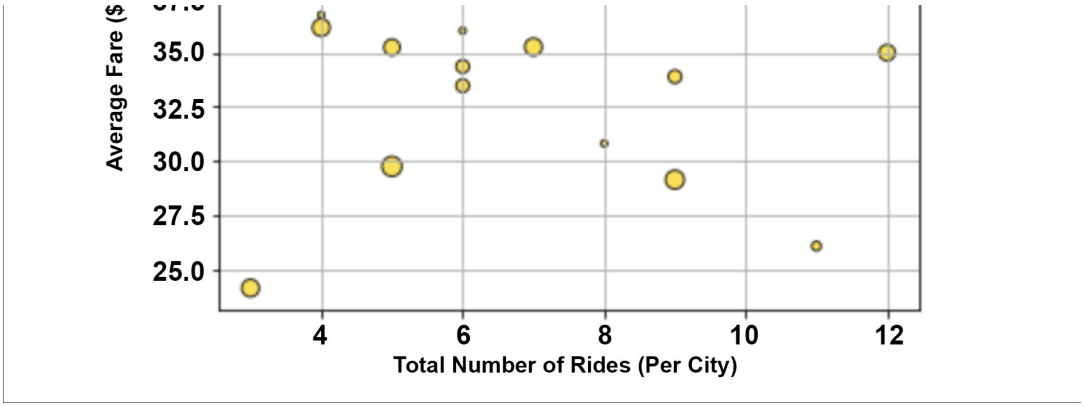
We have two of the three bubble charts. Let's create the final bubble chart: the rural cities bubble chart. Repurposing the code from the suburban cities bubble chart, we'll change the x- and y-axes variables and the size parameter to the rural city data, and we will change the color of the circle to gold.

Add the following code to a new cell and run the cell.

```
# Build the scatter plots for rural cities.
plt.scatter(rural_ride_count,
      rural_avg_fare,
      s=10*rural_driver_count, c="gold",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Rural")
plt.title("PyBer Ride-Sharing Data (2019)")
plt.ylabel("Average Fare ($)")
plt.xlabel("Total Number of Rides (Per City)")
plt.grid(True)
# Add the legend.
plt.legend()
```

When you run the cell, you'll see the following chart with gold markers that vary in diameter based on the average number of drivers in each city, a title, and axes labels.





Congratulations on creating the individual bubble charts for each city type! Next, we will combine these three charts into one chart.

5.3.6: Create a Bubble Chart for All Cities

Now for the final piece! You decide that instead of showing each bubble chart separately in your presentation, you'll show one bubble chart that that has all the city types. Given that the different city types have different colors, this will make your bubble chart stand out and show how each city type compares with one another.

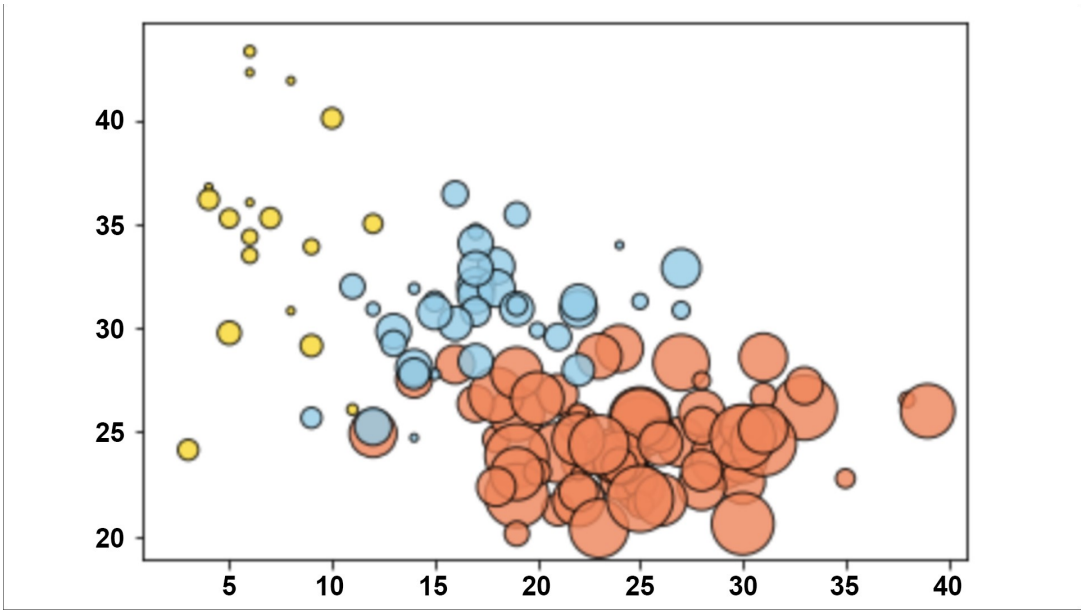
To create a bubble chart that showcases all the different city types in one chart, we'll combine our three scatter plot code blocks in one Jupyter Notebook cell.

Add the three `plt.scatter()` functions for each chart to one cell and run the cell.

```
# Add the scatter charts for each type of city.
plt.scatter(urban_ride_count,
      urban_avg_fare,
      s=10*urban_driver_count, c="coral",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Urban")

plt.scatter(suburban_ride_count,
      suburban_avg_fare,
      s=10*suburban_driver_count, c="skyblue",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Suburban")

plt.scatter(rural_ride_count,
      rural_avg_fare,
      s=10*rural_driver_count, c="gold",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Rural")
```

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

In the output window, behold the (beautiful!) chart created from this code.



Did you notice that we did not have to change the x-limit? That's because plotting all the data on one chart formats the x-axis automatically. We could change the y-limit from 0 to 40, but that might crowd the bubbles in the middle of the chart, making it harder to see any differences in the data.

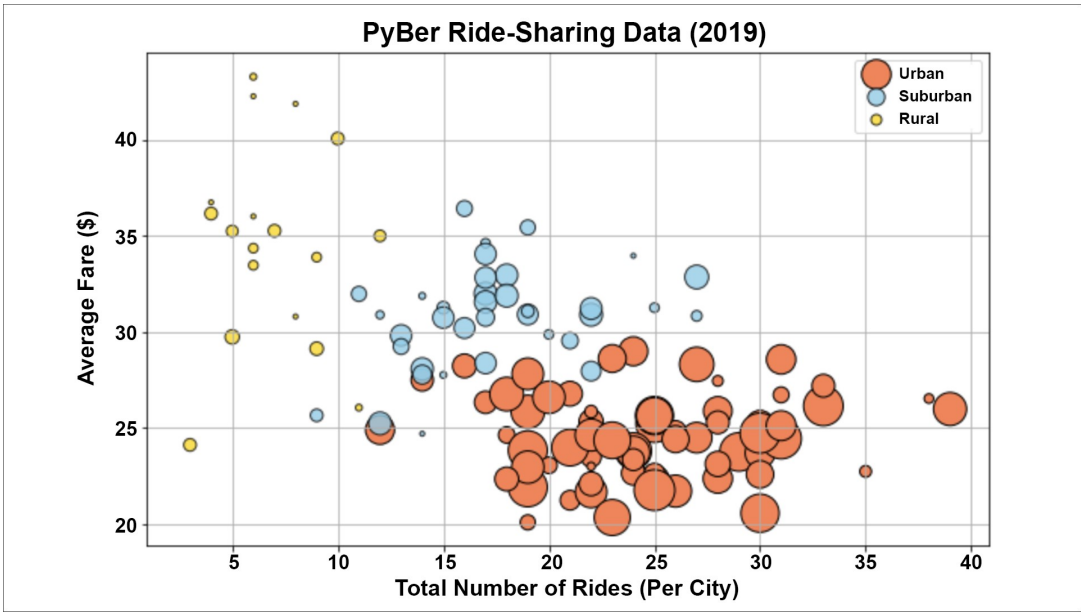
Let's add a title, labels for the axes, a legend, and a grid for all three charts and increase the font size of the axes labels to 12 and the title to 20. We'll also enlarge the figure so the markers are more spread out.

Edit the existing code to look like this.

```
# Build the scatter charts for each city type.
plt.subplots(figsize=(10, 6))
plt.scatter(urban_ride_count,
      urban_avg_fare,
      s=10*urban_driver_count, c="coral",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Urban")

plt.scatter(suburban_ride_count,
```

```
      suburban_avg_fare,
      s=10*suburban_driver_count, c="skyblue",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Suburban")

plt.scatter(rural_ride_count,
      rural_avg_fare,
      s=10*rural_driver_count, c="gold",
      edgecolor="black", linewidths=1,
      alpha=0.8, label="Rural")

# Incorporate the other graph properties
plt.title("PyBer Ride-Sharing Data (2019)", fontsize=20)
plt.ylabel("Average Fare ($)", fontsize=12)
plt.xlabel("Total Number of Rides (Per City)", fontsize=12)
plt.grid(True)

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

Next, run the cell, and in the output window you'll see this updated chart.



Did you notice something that we need to fix? The markers in the legend have different sizes, which are automatically determined based on the average size of the marker. Therefore, we'll need to customize the legend to scale them to the same size.

We can declare a variable for the legend function, `lgnd = plt.legend()`, and add parameters for font size, legend location, and legend title, along with some other features.

After we declare the variable for the legend, we can use `legendHandles[]._sizes` to set the font size of the marker in the legend to a fixed size. Inside the brackets, we can add the list element for the number of markers in the legend.

Add the following code to your scatter plot code in place of `plt.legend()`:

```
# Create a legend
lgnd = plt.legend(fontsize="12", mode="Expanded",
         scatterpoints=1, loc="best", title="City Types")
lgnd.legendHandles[0]._sizes = [75]
lgnd.legendHandles[1]._sizes = [75]
lgnd.legendHandles[2]._sizes = [75]
lgnd.get_title().set_fontsize(12)
```

Let's break down what this code is doing for the legend:

1. We made the font size for the text "small" with `fontsize=`.
2. Then we expanded the legend horizontally using `mode=` to fit the area. Because the font size is small, this is optional.
3. We added the number of scatter points in the legend for each marker to be 1. We can add multiple marker points by increasing the number.
4. The location setting, `loc=`, for the legend is where it will fit the "best" based on the plotting of the data points.
5. We added a legend title.
6. We set each marker in the legend equal to 75-point font by using the `legendHandles[]._sizes` attribute and list indexing in the brackets to reference one of the three scatter plots.
7. Finally, we increased the font size of the legend title to 12.

Next, we need to add a note to the right of the chart to let the viewer know that the circle size correlates with the driver count for each city. Our note will say: "Note: Circle

size correlates with driver count per city." To do this, we'll use the `plt.text()` function and add the text. Inside the function, we add the x and y coordinates for the chart and the text in quotes.

The x and y coordinates are based on the chart coordinates. We can see that our chart has a width (i.e., x, between 0 and 42; and y, between 18 and 50). Our x position can be ~ 42, and the y position can be in the middle, ~ 32–35.

Add the following code after the code that creates the legend:

```
# Incorporate a text label about circle size.
plt.text(42, 35, "Note:\nCircle size correlates\nwith driver count per city.
```

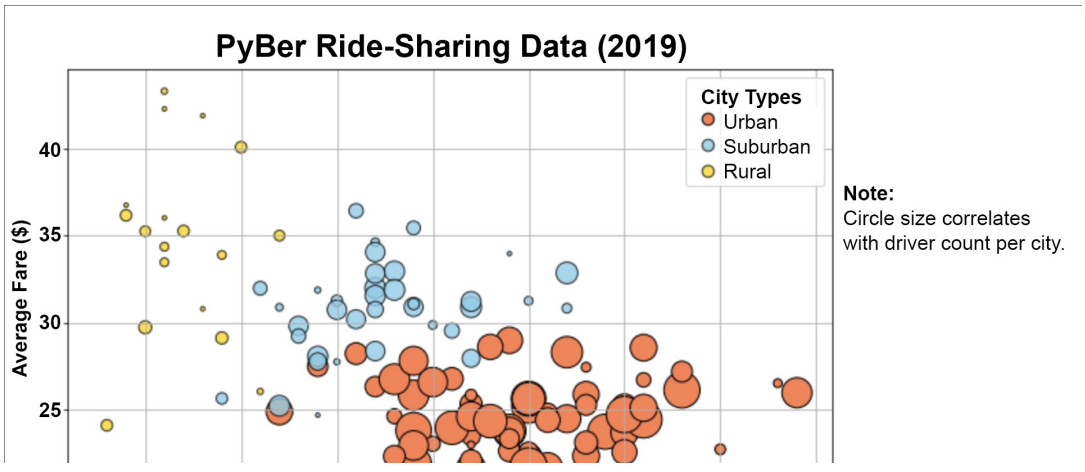
Finally, we need to save the chart in a folder using the `plt.savefig()` function, and provide a direct path to the folder and filename of the saved image.

Add the following code after the code that added the text:

```
# Save the figure.
plt.savefig("analysis/Fig1.png")
```

> **IMPORTANT!**
> Before we run the whole code block, make sure you create a folder named "analysis" in the PyBer_Analysis folder.

After we run the cell, our final chart will look like this:



Congratulations on creating the ride-sharing bubble chart!

Note

For further information on creating legends and adding text to charts, see the following documentation:

Matplotlib documentation on matplotlib.pyplot.legend
(https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.legend.html)

Matplotlib documentation on matplotlib.axes.Axes.legend
(https://matplotlib.org/2.0.0/api/_as_gen/matplotlib.axes.Axes.legend.html)

Matplotlib documentation on matplotlib.pyplot.text
(https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.text.html)

5.3.7: Commit Your Code

It's almost time for that afternoon meeting with Omar and you know he's going to be excited to see your results. To add the cherry on top of your already impressive work, you quickly commit your code to GitHub so Omar will be able to access it before he even asks.

Add, Commit, Push

After you save your `PyBer.ipynb` file to your computer, add it to the PyBer_Analysis GitHub repository.

1. Launch the command line or Git Bash.
2. Navigate to your PyBer_Analysis folder.
3. Check the status by using `git status`.
4. Add all the file(s) by using `git add .`.
5. Commit the files to be added to the repository and add a Git message by using `git commit -m "message"`.
6. Add the file to your repository by using `git push`.
7. Refresh your GitHub page to see the changes to your repository.

5.4.1: Summary Statistics for Number of Rides by City Type

After Omar looks at your bubble chart, he suggests that you should add some statistical analysis because it will help you demonstrate the relevance of the data, especially the number of rides for each city. This will help V. Isualize and other stakeholders make decisions about which types of cities need more driver support.

The old adage "There are many ways to skin a cat" comes to mind when getting the summary statistics. We'll use and compare the following three ways to calculate the summary statistics:

- The Pandas `describe()` function on the DataFrame or Series.
- The Pandas `mean()`, `median()`, and `mode()` methods on a Series.
- The NumPy `mean()` and `median()` functions, and the SciPy stats `mode()` function on a Series.

> **Rewind**
> The **measures of central tendency** refer to the tendency of data to be toward the middle of the dataset. The three key measures of central tendency are the mean, median, and mode.

Pandas describe() Function

The `describe()` function is a convenient tool to get a high-level summary statistics on a DataFrame or Series. After running the function, the output will show the count,

mean, standard deviation, minimum value, 25%, 50%, and 75% percentiles, and maximum value from a DataFrame column that has numeric values.

> **Rewind**
>
> Remember—quartiles are percentiles. The lower quartile is the 25th percentile. The upper quartile is the 75th percentile.

Let's use the `describe()` function on the urban, suburban, and rural DataFrames. Add the following code to a new cell:

```
# Get summary statistics.
urban_cities_df.describe()
```

The output from running this cell is:

```
# Get summary statistics.
urban_cities_df.describe()
```

	fare	ride_id	driver_count
count	1625.000000	1.625000e+03	1625.000000
mean	24.525772	4.873485e+12	36.678154
std	11.738649	2.907440e+12	20.075545
min	4.050000	1.458810e+10	3.000000
25%	14.550000	2.400244e+12	22.000000
50%	24.640000	4.711188e+12	37.000000
75%	34.580000	7.451579e+12	52.000000
max	44.970000	9.991538e+12	73.000000

> **Skill Drill**
>
> Use the `describe()` function on the `suburban_cities_df` and `rural_cities_df` DataFrames and compare the outputs of all three DataFrames.

Now let's calculate the summary statistics of the ride count for each city type. Add the following code to a new cell and run the cell.

```
# Get summary statistics.
urban_ride_count.describe()
```

The output from running this cell will show the total number, the average, the standard deviation, the maximum, minimum, and the 25%, 50%, and 75% quartiles.

```
# Get summary statistics.
urban_ride_count.describe()
```

```
count     66.000000
mean      24.621212
std        5.408726
min       12.000000
25%       21.000000
50%       24.000000
75%       28.000000
max       39.000000
Name: ride_id, dtype: float64
```

Using the describe() function on the `suburban_ride_count` , what is the highest number of rides for a suburban city?

Check Answer

Finish ▸

Using the describe() function on the `rural_ride_count` , what is the lowest number of rides for a rural city?

Check Answer

Pandas mean(), median(), and mode() Methods

If we want to get only the mean without getting the complete summary statistics, we can use the `mean()` method.

Add the following code to a new cell and run the cell.

```
# Calculate the mean of the ride count for each city type.
round(urban_ride_count.mean(),2), round(suburban_ride_count.mean(),2), rou
```

The output will be the average ride count for each city type rounded to two decimal places:

```
# Calculate the mean of the ride count for each city type.
round(urban_ride_count.mean(), 2), round(suburban_ride_count.mean(),2), round(rural_ride_count.mean(),2)
```

```
(24.62, 17.36, 6.94)
```

Notice that the mean of the ride count for each city type using the `mean()` method is the same value that was returned using the `describe()` function.

> **Finding**
> If we compare the average number of rides between each city type, we'll notice that the average number of rides in the rural cities is about 4 and 3.5 times lower than urban and suburban cities, respectively.

To get the median of DataFrame or Series, we can also use the Pandas `median()` method in the same way as we used the `mean()` method.

What is the median for the number or rides in the urban cities?

Check Answer

Finish ▸

Similarly, we can use the `mode()` method to get the mode of the ride counts for each city. Add the following code to a new cell:

```
# Calculate the mode of the ride count for the urban cities.
urban_ride_count.mode()
```

The output will be the mode or modes of the Series. In this case, we have two modes—one at 22 and one at 25:

```
# Calculate the mode of the ride count for the urban cities.
urban_ride_count.mode()
```

```
0    22
1    25
dtype: int64
```

> **Rewind**
>
> A dataset like this Series can have any number of modes—or even no mode!

What is the mode for the number or rides in the suburban cities?

Check Answer

Finish ▸

NumPy mean() and median() Functions and SciPy mode() Function

An optional approach to calculating the mean, median, and mode of a DataFrame or Series is to use the NumPy and SciPy statistics modules. We introduce these methods

because there might come a time when you're working in the Python interpreter or VS Code environment instead of the Jupyter Notebook environment.

Whether you are using the use the Python interpreter, VS Code, or Jupyter Notebook environment, we will need to import the NumPy and SciPy statistics modules. Add the following import statements to a new cell in your `PyBer.ipynb` file and run the cell.

```
# Import NumPy and the stats module from SciPy.
import numpy as np
import scipy.stats as sts
```

Let's calculate the mean, median, and mode—otherwise known as the **measures of central tendency** for the ride counts—and print out those measures.

To get the measures of central tendency of the ride counts for the urban cities, add the following code block.

```
# Calculate the measures of central tendency for the ride count for the urba
mean_urban_ride_count = np.mean(urban_ride_count)
print(f"The mean for the ride counts for urban trips is {mean_urban_ride_cou

median_urban_ride_count = np.median(urban_ride_count)
print(f"The median for the ride counts for urban trips is {median_urban_ride

mode_urban_ride_count = sts.mode(urban_ride_count)
print(f"The mode for the ride counts for urban trips is {mode_urban_ride_cou
```

When we run the cell, we get the following output:

```
The mean for the ride counts for urban trips is 24.62.
The median for the ride counts for urban trips is 24.0.
The mode for the ride counts for urban trips is ModeResult(mode=array([22]), count=array([7])).
```

Let's go over what the output gives us:

- The mean and median values that were returned are the same values that were returned using the `describe()` function and the `mean()` and `median()` methods, respectively.

- With SciPy statistics, the mode result that's returned is the mode that appears the most frequently.
- `ModeResult` returned two attributes:
  - The first attribute, `mode`, is 22.
  - The second attribute, `count`, is the number of times it occurs in the dataset, in this case, 7.

Unlike the Pandas `mode()` method, the `sts.mode()` method will return the number of times the mode appears in the dataset.

How many times does 17 appear in the suburban cities?

Check Answer

Finish ▸

Fill in the blanks.

The mode for the rural cities is _____ and it appears _____ times.

Check Answer

Finish ▸

There are a few choices of methods to use to get the mean, median, and mode of a dataset. The method you choose is a matter of preference and depends on whether you're working with the Pandas, NumPy, or statistics modules.

5.4.2: Summary Statistics for the Fare by City Type

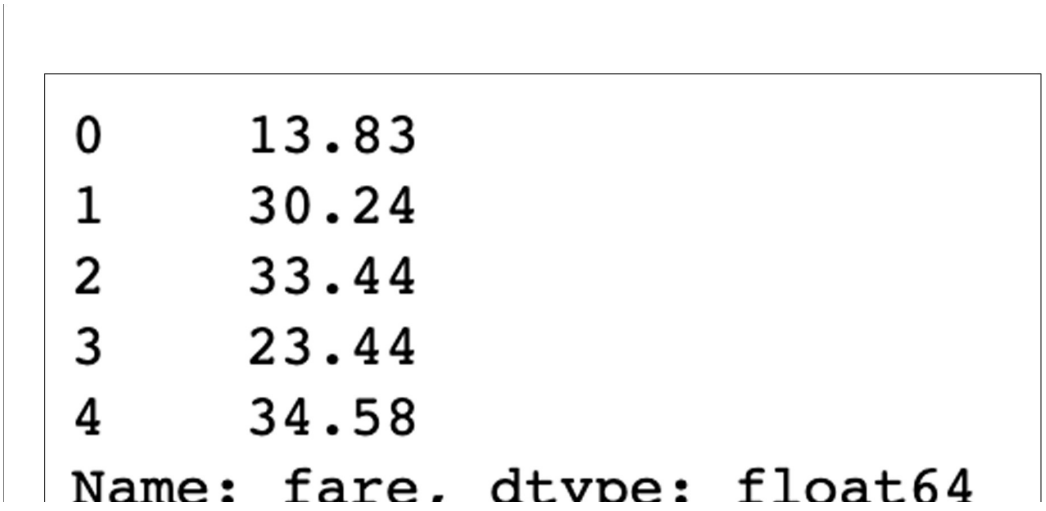
While you're gathering summary statistics, a lightbulb goes off: you should calculate the summary statistics for the average fares for each city type. This will help you determine which city types are generating the most money. And V. Isualize will definitely want to know that!

In order to get the summary statistics for the average fare for each city type, we'll need to get the data from the "fare" column in each city type DataFrame.

Add the following code to a new cell to create a Series with all the fares from the "fare" column for the `urban_cities_df` DataFrame:

```
# Get the fares for the urban cities.
urban_fares = urban_cities_df["fare"]
urban_fares.head()
```

When you run the cell, the output will be all the fares for the urban cities:



Now we can calculate the mean, median, and mode for the `urban_fares` Series. To get the mean and median, we'll use the NumPy mean and median functions; to get the mode, we'll use the SciPy statistics mode function, `sts.mode()`. Using this mode function returns how many times the mode appears in the dataset.

Add the following code to a new cell and run the cell.

```
# Calculate the measures of central tendency for the average fare for the
mean_urban_fares = np.mean(urban_fares)
print(f"The mean fare price for urban trips is ${mean_urban_fares:.2f}.")

median_urban_fares = np.median(urban_fares)
print(f"The median fare price for urban trips is ${median_urban_fares:.2f}

mode_urban_fares = sts.mode(urban_fares)
print(f"The mode fare price for urban trips is {mode_urban_fares}.")
```

The output from the code will look like this:

```
The mean fare price for urban trips is $24.53.
The median fare price for urban trips is $24.64.
The mode fare price for urban trips is $ModeResult(mode=array([22.86]), count=array([5])).
```

The mean is $24.53, the median is $24.64, and the mode is $22.86, which appears five times.

The average fare price is higher in which type of city?

- ○ Urban
- ○ Rural
- ○ Suburban

Check Answer

Finish ▸

5.4.3: Summary Statistics for the Number of Drivers by City Type

Now that you have the summary statistics for the revenue by city type, you think it would be a good idea to get the summary statistics for the number of drivers by city type. After all, you want to be well prepared to answer any question in your presentation. Having this data in hand will allow V. Isualize and other stakeholders to know which cities need more driver support.

In order to perform summary statistics for the number of drivers by city type, we need to create a Series for each city type based on the driver_count column in each city type DataFrame.

Let's get the Series for the urban drivers by adding the following code to a new cell:

```
# Get the driver count data from the urban cities.
urban_drivers = urban_cities_df['driver_count']
urban_drivers.head()
```

When you run the cell, the output will look like this:

```
#  Get the driver count data from the urban cities.
urban_drivers = urban_cities_df['driver_count']
urban_drivers.head()
```

```
0     5
1    72
2    57
```

```
3    34
4    46
Name: driver_count, dtype: int64
```

Skill Drill

Calculate the mean, median, and mode for the urban, suburban, and rural driver count Series using NumPy and SciPy statistics module.

After calculating the mean number of drivers for each city, what is the average number of drivers for the suburban cities?

- ○ 37
- ○ 14
- ○ 4

Check Answer

Finish ▸

5.4.4: Create Box-and-Whisker Plots

You're feeling pretty good about your upcoming presentation. You reconvene with Sasha to thank her for her advice and see what final tips she might have. She thinks it would be a good idea to show V. Isualize if there are any outliers in the data. To do this, she suggests using box-and-whisker plots.

Up until now, we have gained experience creating a variety of charts where each chart can showcase the data in different ways, and we have performed summary statistics on the number of rides, the fare, and the number of drivers for each city type. Now, we are going to visualize the summary statistics and determine if there are any outliers by using box-and-whisker plots.

> **Rewind**
> Box-and-whisker plots are an effective way to show a lot of information about distribution in a small amount of space, especially outliers.

Box-and-Whisker Plots for Ride Count Data

Creating a box-and-whisker plot requires that we use the `ax.boxplot()` function, which takes an array inside the parentheses. We can also add a title and axes labels as we have done before.

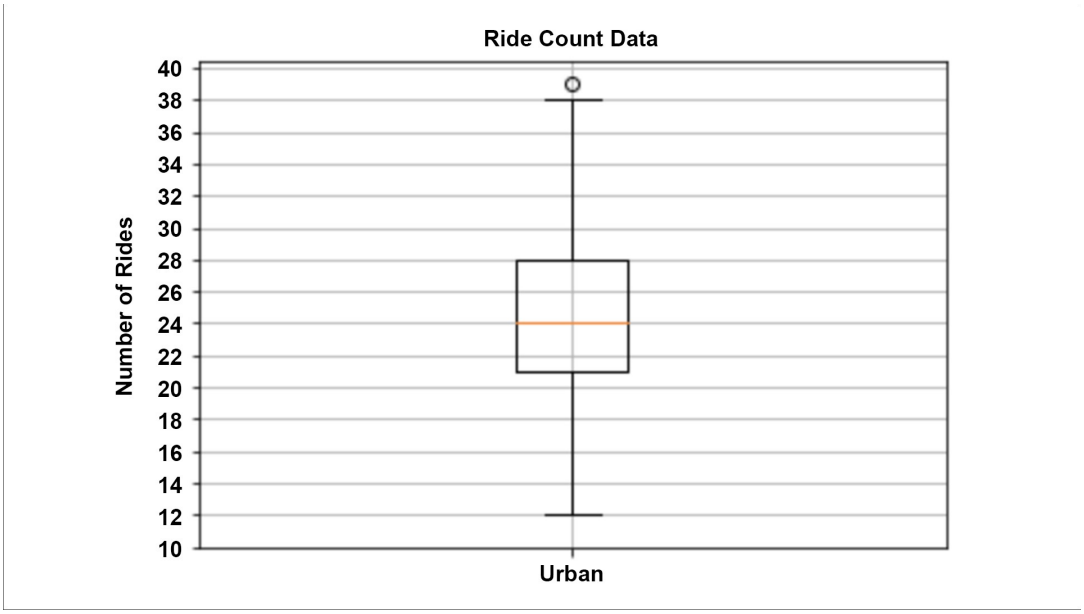
Let's create our `urban_ride_count` box-and-whisker plot. In a new cell, add the following code:

```
# Create a box-and-whisker plot for the urban cities ride count.
x_labels = ["Urban"]
fig, ax = plt.subplots()
ax.boxplot(urban_ride_count, labels=x_labels)
# Add the title, y-axis label and grid.
ax.set_title('Ride Count Data (2019)')
ax.set_ylabel('Number of Rides')
ax.set_yticks(np.arange(10, 41, step=2.0))
ax.grid()
plt.show()
```

Some of this code looks familiar, but a few lines are new. Let's break down what the code is doing.

- First, we create the x-axis labels with a list, `x_labels = ["Urban"]`.
- Next, the data and labels are passed in the `boxplot` function.
- Finally, we set the `y_ticks` with a range from 10 to 41 with ticks at an increment of 2. This will help determine where the minimum and maximum lie as well as any outliers.

When you run the cell, the urban ride count data box-and-whisker plot will look like this:



Looking at this box-and-whisker plot, we can see:

1. There is at least one outlier, which is close to 40. This our maximum data point, 39.
2. The minimum is 12.
3. The median is 24 or the 50th percentile.
4. The standard deviation is about 5 because the box upper and lower boundaries represent the upper and lower quartiles.

Rewind

We generated the same summary the box-and-whisker plot visual shows us by getting the high-level summary statistics using `urban_ride_count.describe()`.

```
# Get summary statistics.
urban_ride_count.describe()
```

```
count    66.000000
mean     24.621212
std       5.408726
min      12.000000
25%      21.000000
50%      24.000000
75%      28.000000
max      39.000000
Name: ride_id, dtype: float64
```

Please type an answer to fill in the blanks for the following sentence.

There are _____ outliers for the number of suburban rides and _____ outliers for the number of rural rides.

Check Answer

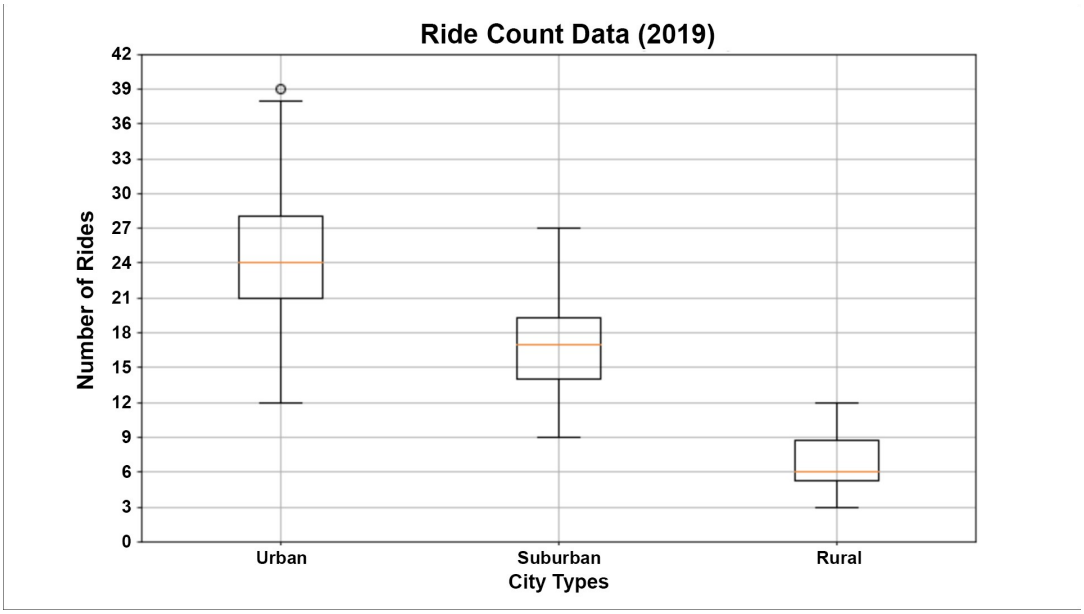
Finish ▸

To show all the city type box-and-whisker plots on one chart we need to modify the `boxplot()` function and add other features. We will increase the size of the chart and the font of the title and axes labels.

Add the following code to the new cell and run the cell.

```
# Add all ride count box-and-whisker plots to the same graph.
x_labels = ["Urban", "Suburban","Rural"]
ride_count_data = [urban_ride_count, suburban_ride_count, rural_ride_count]
fig, ax = plt.subplots(figsize=(10, 6))
ax.set_title('Ride Count Data (2019)',fontsize=20)
ax.set_ylabel('Number of Rides',fontsize=14)
ax.set_xlabel("City Types",fontsize=14)
ax.boxplot(ride_count_data, labels=x_labels)
ax.set_yticks(np.arange(0, 45, step=3.0))
ax.grid()
# Save the figure.
plt.savefig("analysis/Fig2.png")
plt.show()
```

When we run this cell, the box-and-whisker plot below has all three individual box-and-whisker plots with the city type on the x-axis.



Finding

> There is one outlier in the urban ride count data. Also, the average number of rides in the rural cities is about 4- and 3.5-times lower per city than the urban and suburban cities, respectively.

One of our tasks was to find out if there were any outliers. We know that the outlier for the `urban_ride_count` is 39. From this information, we can find out which city has the highest rider count.

> **Rewind**
> Recall that the `urban_ride_count` is a Series with the index of the city and the data the number of rides for each city.
>
> ```
> city
> Amandaburgh          18
> Barajasview          22
> Carriemouth          27
> Christopherfurt      27
> Deanville            19
> Name: ride_id, dtype: int64
> ```

We can get all the "True" values where the `urban_ride_count` equals 39. Then, we can filter the `urban_ride_count` Series for all the "True" values and get the city name from the index, like this:

```
# Get the city that matches 39.
urban_city_outlier = urban_ride_count[urban_ride_count==39].index[0]
print(f"{urban_city_outlier} has the highest rider count.")
```

The output from running this cell is `West Angela has the highest rider count.`

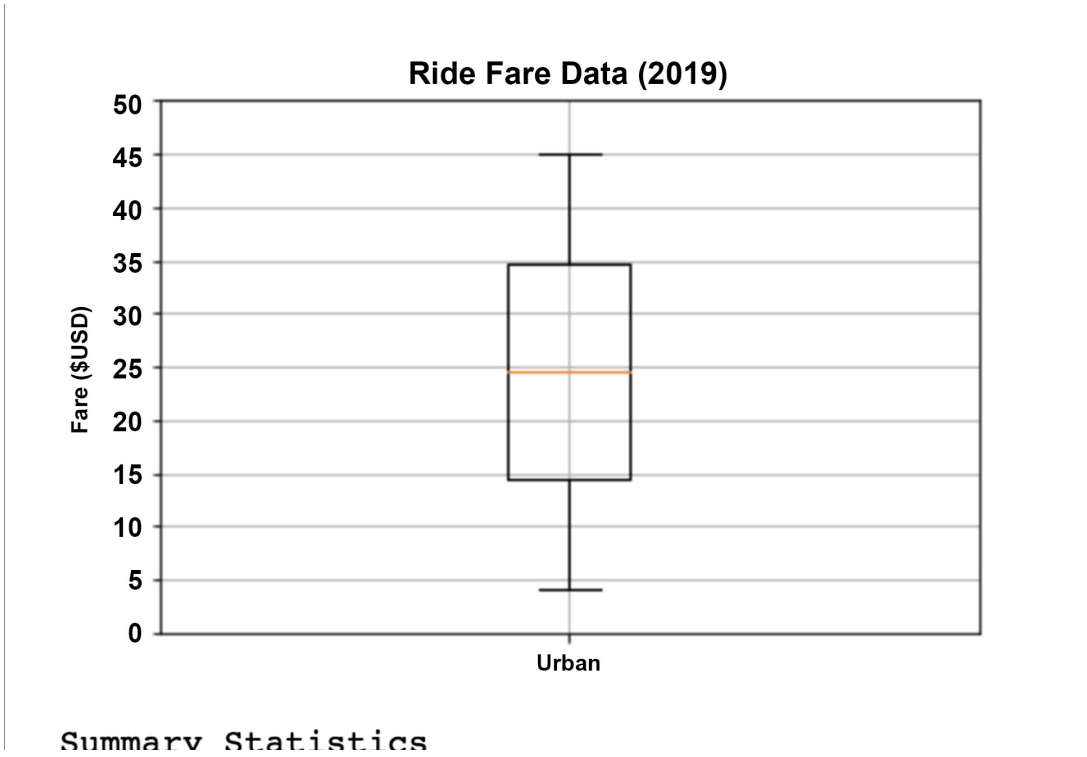
Box-and-Whisker Plots for Ride Fare Data

Next, let's create box-and-whisker plots for the ride fare data with summary statistics.

For the fare data, we will use the `urban_fares` Series we created earlier. Add the following code to the new cell:

```
# Create a box-and-whisker plot for the urban fare data.
x_labels = ["Urban"]
fig, ax = plt.subplots()
ax.boxplot(urban_fares, labels=x_labels)
# Add the title, y-axis label and grid.
ax.set_title('Ride Fare Data (2019)')
ax.set_ylabel('Fare($USD)')
ax.set_yticks(np.arange(0, 51, step=5.0))
ax.grid()
plt.show()
print("Summary Statistics")
urban_fares.describe()
```

When you run the cell, the urban fare data box-and-whisker plot will look like this:



Summary Statistics

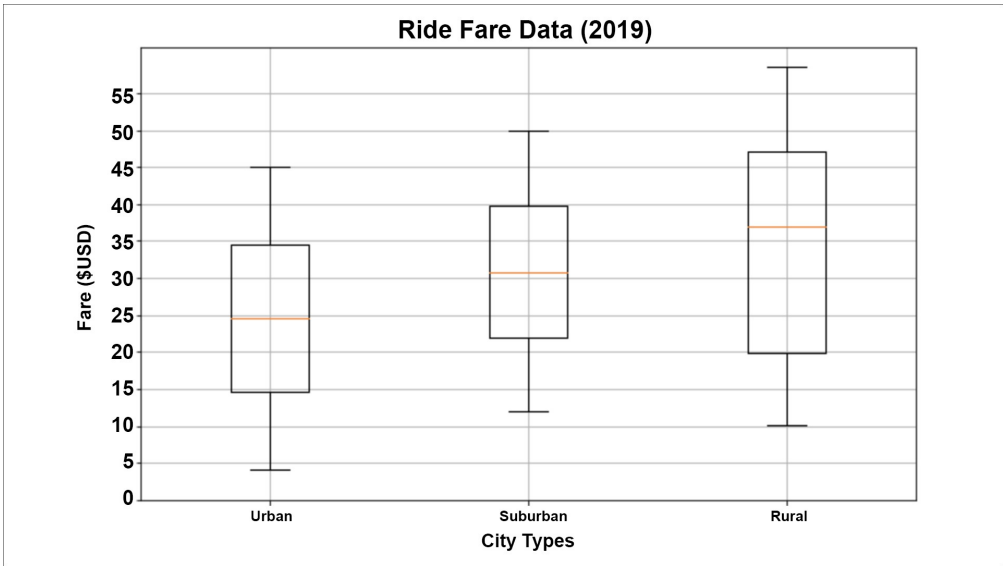
```
count    1625.000000
mean       24.525772
std        11.738649
min         4.050000
25%        14.550000
50%        24.640000
75%        34.580000
max        44.970000
Name: fare, dtype: float64
```

Skill Drill

Create box-and-whisker plots for the `suburban_fares` and the `rural_fares` with summary statistics.

Skill Drill

Create a box-and-whisker plot that has all three city types' fare data in one plot that looks similar to the following image. Save the combined box-and-whisker plot as `Fig3.png` to your "analysis" folder.



Finding

> From the combined box-and-whisker plots, we see that there are no outliers. However, the average fare for rides in the rural cities is about $11 and $5 more per ride than the urban and suburban cities, respectively. Why do you think there is such a big difference? By looking at the number of riders for each city, can you get a sense of the overall revenue?

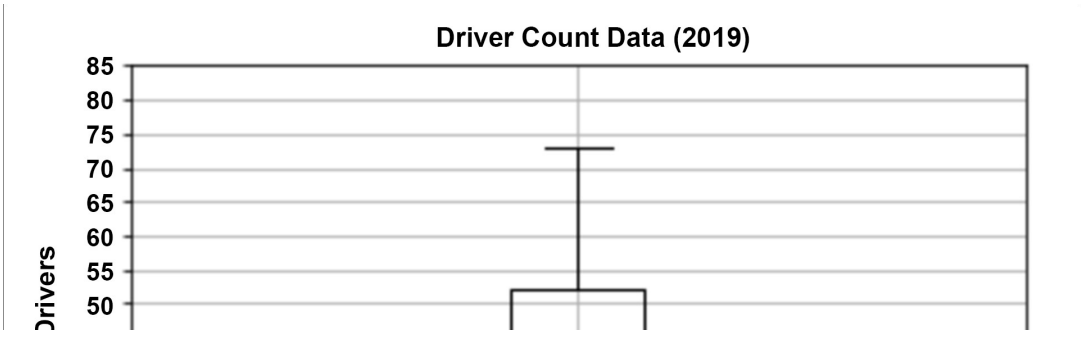
Box-and-Whisker Plots for Driver Count Data

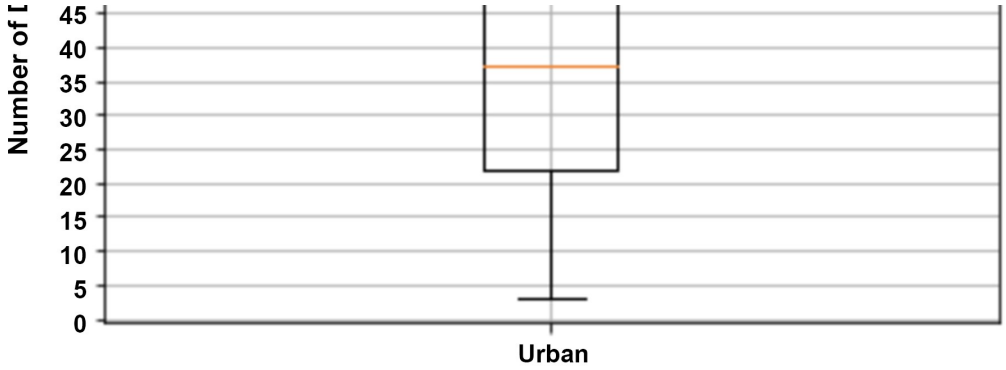
We're getting really good at creating box-and-whisker plots! We need to do one last set of box-and-whisker plots. Let's create a box-and-whisker plot for the driver count data with summary statistics.

For the driver count data, we'll use the `urban_drivers` Series we created earlier. Add the following code to a new cell:

```
# Create the box-and-whisker plot for the urban driver count data.
x_labels = ["Urban"]
fig, ax = plt.subplots()
ax.boxplot(urban_drivers,labels=x_labels)
# Add the title, y-axis label and grid.
ax.set_title('Driver Count Data (2019)')
ax.set_ylabel('Number of Drivers)')
ax.set_yticks(np.arange(0, 90, step=5.0))
ax.grid()
plt.show()
print("Summary Statistics")
urban_drivers.describe()
```

When you run the cell, the urban fare data box-and-whisker plot will look like this:





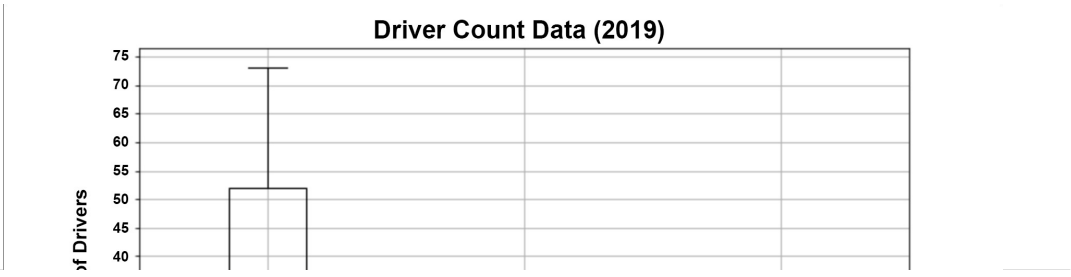

```
Summary Statistics

count    1625.000000
mean       36.678154
std        20.075545
min         3.000000
25%        22.000000
50%        37.000000
75%        52.000000
max        73.000000
Name: driver_count, dtype: float64
```

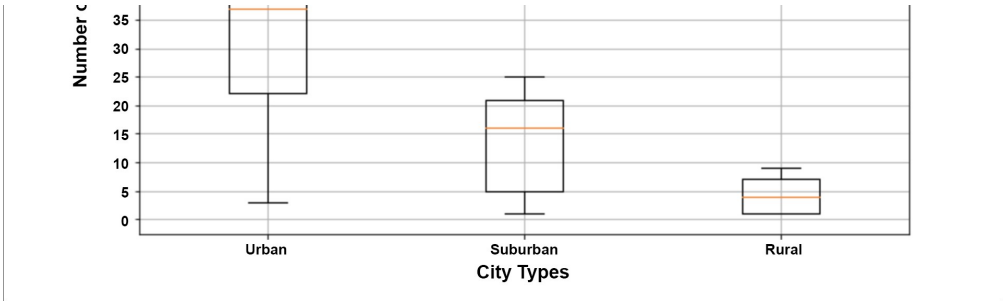
Skill Drill

Using the code for the box-and-whisker plots for the urban drivers, create box-and-whisker plots for the `suburban_drivers` and the `rural_drivers` Series with summary statistics.

Skill Drill

Create a box-and-whisker plot that has all three city types' driver count data in one box-and-whisker plot that looks similar to the following image. Save this combined box-and-whisker plot as `Fig4.png` in your "analysis" folder.





Finding

The average number of drivers in rural cities is nine to four times less per city than in urban and suburban cities, respectively. By looking at the driver count data and fare data, can you get a sense of the overall revenue?

Note

For more information on creating box-and-whisker plots using the object-oriented interface method, see the following documentation:

Matplotlib documentation on statistics visualizations (https://matplotlib.org/3.1.1/gallery/index.html#statistics)

Matplotlib documentation on box plots (https://matplotlib.org/examples/statistics/boxplot_demo.html)

5.4.5: Commit Your Code

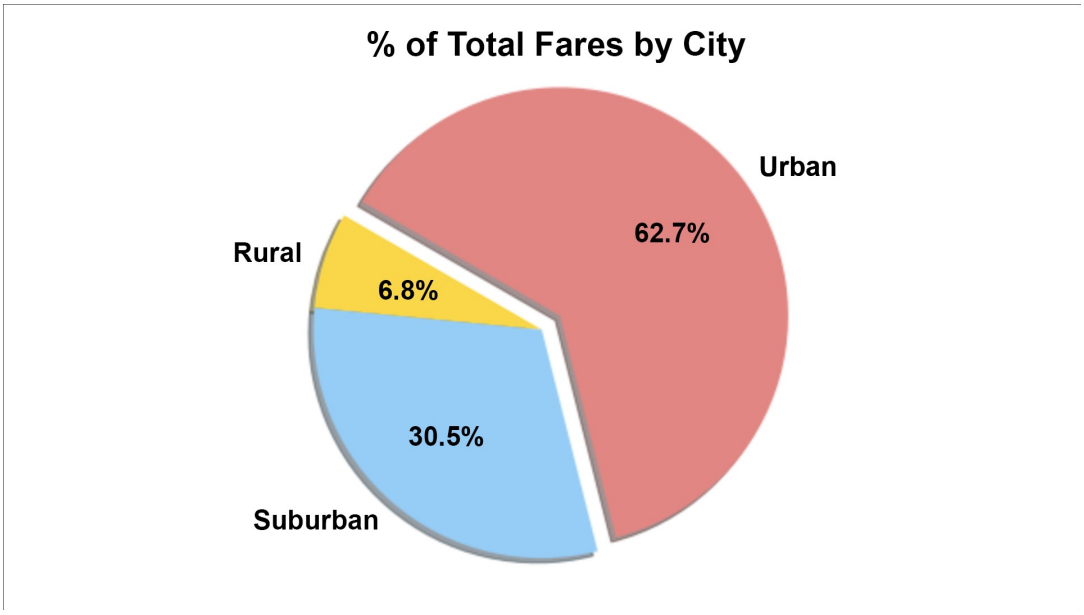
Omar is going to love these results—and V. Isualize may want to look at your code herself. In anticipation for your big break, you make sure to update your GitHub repository.

> **Add, Commit, Push**
>
> After you save your `PyBer.ipynb` file to your computer, add it to the PyBer_Analysis GitHub repository by following these steps:
>
> 1. Launch the command line or Git Bash.
> 2. Navigate to your PyBer_Analysis folder.
> 3. Check the status with `git status`.
> 4. Add all the file(s) by using `git add .`.
> 5. Commit the files to be added to the repository and add a Git message, with `git commit -m "message"`.
> 6. Add the file to your repository by using `git push`.
> 7. Refresh your GitHub page to see the changes to your repository.

5.5.1: Get the Percentage of Fares for Each City Type

As you are working on the final pieces of your presentation, you hear from Sasha that V. Isualize has a habit of asking presenters for alternate visualizations in meetings, and you are going to be ready with all the possibilities. You know pie charts aren't as common, but they are a good way to show percentages. So you get to work on creating a pie chart that showcases the percentage of fares for each type of city.

To showcase the percentage of the overall fares for each type of city, where each pie wedge will represent the percentage of total fares for each city type, the pie chart should look like this.



To create this pie chart, we will need to do the following:

1. Get the total fares for each city type.
2. Get the total for all the fares for all the city types.
3. Calculate the percentage of the total fares for each city type.

To get the total fares for city type, we need to create a Series where the index is the type of city on the `pyber_data_df` DataFrame and the column is the sum of the fares for city type.

> **Rewind**
> To create a Data Series with one of the columns in a DataFrame, we can use the `groupby()` function and add the column inside the parentheses.

To calculate the percentage of the total fares for each city type based on the total fares, we will:

1. Use the `groupby()` function on the `pyber_data_df` DataFrame and group by the `type` of city column so the type of city is the index.
2. Apply the `sum()` method on the `fare` column to get the Series with the total number of fares for each city type.
3. Divide the total fares for each city type by the total of all the fares and multiply by 100.

We will create a Series using the `groupby()` function on the `pyber_data_df` DataFrame that has the type of city as an index. Then we'll apply the `sum()` method to the Series for each type of city and select the fare column.

Add the following code to a new cell and run the cell.

```
# Get the sum of the fares for each city type.
sum_fares_by_type = pyber_data_df.groupby(["type"]).sum()["fare"]
sum_fares_by_type
```

The output is a Series with the sum of the fares for each type of city:

```
type
```

```
Rural           4327.93
Suburban       19356.33
Urban          39854.38
Name: fare, dtype: float64
```

Next, we'll get the total fares by using the `sum()` method on the fare column of the `pyber_data_df` DataFrame.

Add the following code to a new cell:

```
# Get the sum of all the fares.
total_fares = pyber_data_df["fare"].sum()
total_fares
```

When you run the cell, the output is 63538.64, or $63,538.64, which is the sum of the fares for all the city types.

```
# Get the sum of all the fares.
total_fares =  pyber_data_df["fare"].sum()
total_fares
```

```
63538.64
```

Next, we can calculate the percentage of total fares for each city type by dividing the `sum_fares_by_type` Series by the `total_fares` Series and multiplying by 100, like this:

```
# Calculate the percentage of fare for each city type.
type_percents = 100 * sum_fares_by_type / total_fares
type_percents
```

As you become more adept at programming, you can perform the calculation in one line of code, as follows:

```
# Calculate the percentage of fare for each city type.
type_percents = 100 * pyber_data_df.groupby(["type"]).sum()["fare"] / pyber_
type_percents
```

Executing the code will give us the following percentages:

```
type
Rural          6.811493
Suburban      30.463872
Urban         62.724635
Name: fare, dtype: float64
```

Now we are ready to create our pie chart!

5.5.2: Pie Chart for the Percentage of Fares by City Type

While breaking down the percentage fares by each city type, you have a moment of brilliance: you're going to keep this on-brand by using the company's color scheme of gold, sky blue, and coral.

We will create a pie chart by using the MATLAB approach, using the `plt.pie()` function.

> **Rewind**
> To create a pie chart with the `plt.pie()` function, we need an array that contains the values and labels we are plotting.

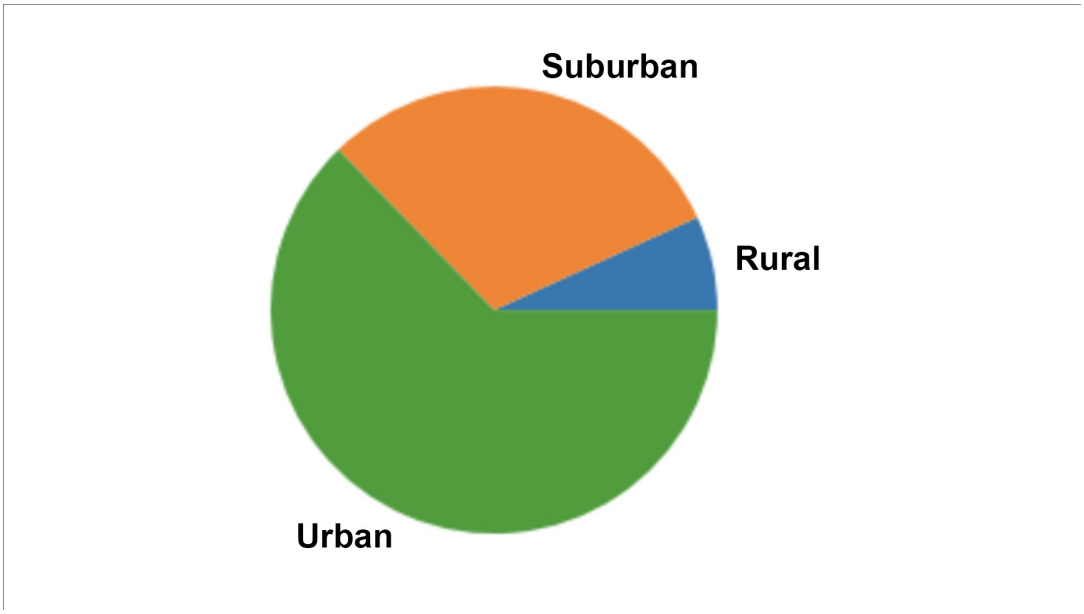
For the pie chart, each pie wedge will represent a city and its percentage of the total fares. The labels will be the city type.

We can use `type_percents` for the values for each pie wedge and create an array for the labels.

Let's add the following code to a new cell:

```
# Build the percentage of fares by city type pie chart.
plt.pie(type_percents, labels=["Rural", "Suburban", "Urban"])
plt.show()
```

Our pie chart will look like this after we run the cell:

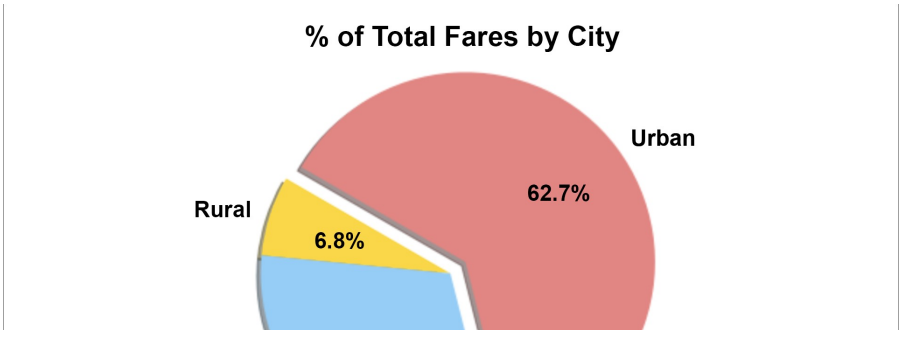


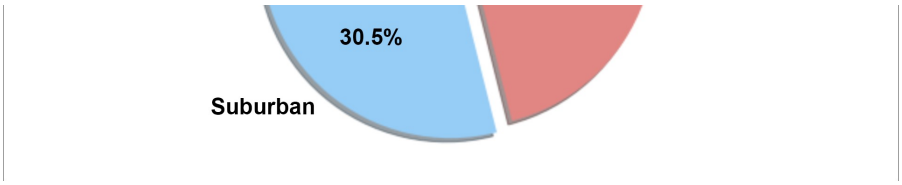
Now that we have a basic pie chart, we can add a title, add the percentages, change the color to adhere to the company color scheme, add a shadow to the pie chart, and adjust the start angle.

Let's edit the `plt.pie()` function with the following added features:

```
# Build the percentage of fares by city type pie chart.
plt.pie(type_percents,
    labels=["Rural", "Suburban", "Urban"],
    colors=["gold", "lightskyblue", "lightcoral"],
    explode=[0, 0, 0.1],
    autopct='%1.1f%%',
    shadow=True, startangle=150)
plt.title("% of Total Fares by City Type")
# Show Figure
plt.show()
```

When you run the cell, the pie chart should look like this.





Before we save the pie chart, let's change the font size to 14. To do this, we will need to edit the code block.

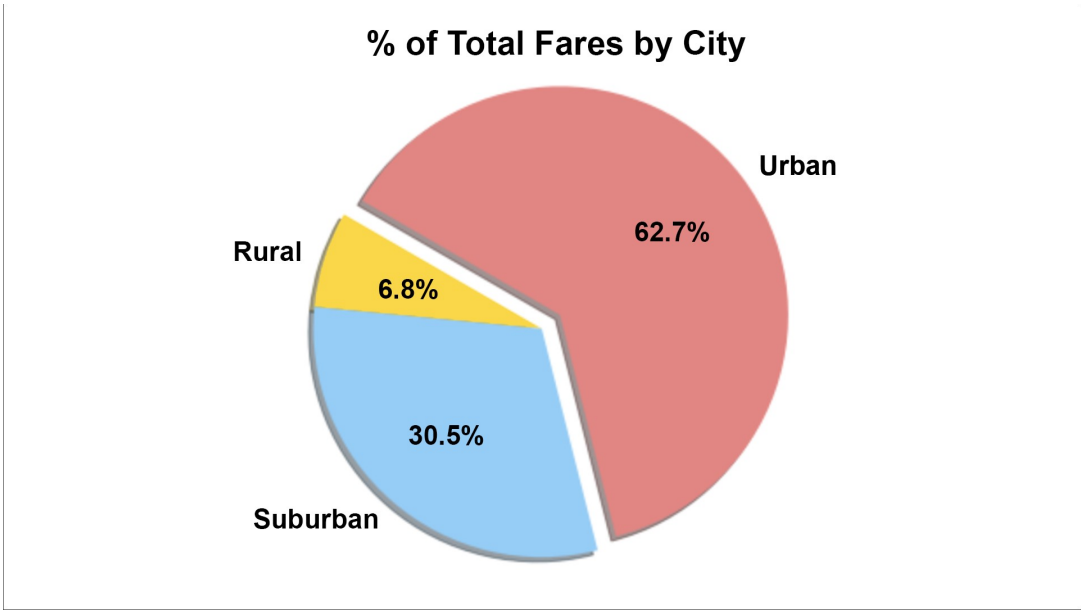
First of all, there is no parameter for `fontsize` in pie charts like there is for scatter plots. However, Matpltolib has a way to change the default parameters for charts by using the `rcParams`, which accesses the run and configure settings for the Matplotlib parameters.

To change the default parameters we need to import the matplotlib library, which is not the same as the matplotlib.pyplot we imported for graphing with the `plt()` function. Let's edit our code block above and add `import matplotlib as mpl` on the first line. After the `plt.title()`, we will change the font size by adding `mpl.rcParams['font.size'] = 14`.

The code block should look like this after setting the font size:

```
# Import mpl to change the plot configurations using rcParams.
import matplotlib as mpl
# Build Pie Chart
plt.subplots(figsize=(10, 6))
plt.pie(type_percents,
    labels=["Rural", "Suburban", "Urban"],
    colors=["gold", "lightskyblue", "lightcoral"],
    explode=[0, 0, 0.1],
    autopct='%1.1f%%',
    shadow=True, startangle=150)
plt.title("% of Total Fares by City Type")
# Change the default font size from 10 to 14.
mpl.rcParams['font.size'] = 14
# Save Figure
plt.savefig("analysis/Fig5.png")
# Show Figure
plt.show()
```

When you run the cell, the final pie chart should look the same as before, but larger.



Note

For more information, see the [Matplotlib documentation on customizing Matplotlib with style sheets and rcParams (https://matplotlib.org/users/customizing.html)](https://matplotlib.org/users/customizing.html) .

Now that you have created the first pie chart showing the percentage of total fares by city type, we need to create two more pie charts: the percentage of total rides by city type and the percentage of total drivers by city type.

5.5.3: Commit Your Code

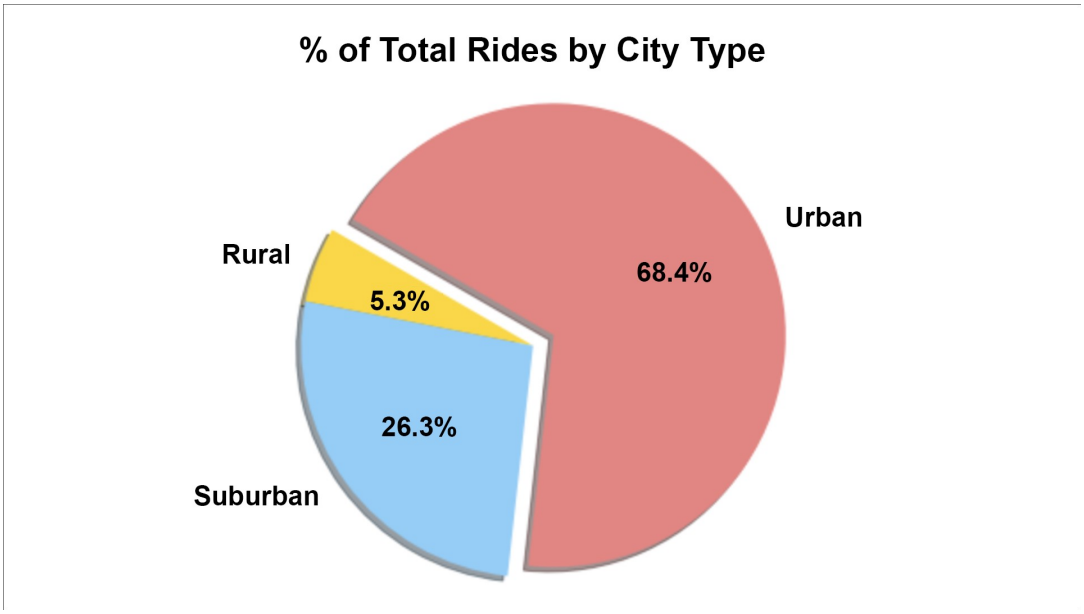
You're ready to move on to the next pie chart, but remember what Omar told you—commit early and often! So you quickly commit your code and refill your coffee.

> **Add/Commit/Push**
>
> After you save your `PyBer.ipynb file` to your computer, add it to the PyBer_Analysis GitHub repository.
>
> 1. Launch the command line or Git Bash.
> 2. Navigate to your PyBer_Analysis folder.
> 3. Check the status with `git status`.
> 4. Add all the file(s) by using `git add .`.
> 5. Commit the files to be added to the repository and add a Git message with `git commit -m ""`.
> 6. Add the file to your repository by using `git push`.
> 7. Refresh your GitHub page to see the changes to your repository.

5.6.1: Calculate Ride Percentages

Feeling confident about making pie charts, you quickly get to work on writing the script to create a pie chart that showcases the percentage of total rides by city type.

The second pie chart we need to create will showcase the percentage of total rides for each type of city, where each pie wedge represents the percentage of total rides for each city type. The final pie chart should look like this:



To create this pie chart, we need to do the following:

- Get the total number of rides for each city type.
- Get the total rides for all the city types.
- Calculate the percentage of the total rides for each city type.

To get the total rides for each type of city, we need to create a Series of data where the index is the type of city, and the column for the Series is the number of the rides for the type of city. This is similar to how we created the Series for the percentage of fares for each city type.

To calculate the percentage of rides for each city type based on all the rides, we will:

1. Use the `groupby()` function on the `pyber_data_df` DataFrame and group by the type of city column:

   ```
   pyber_data_df.groupby(["type"])
   ```

2. Apply the `count()` function on the `ride_id` column to get the Series with the total number of rides for each city type:

   `pyber_data_df.groupby(["type"]).count()["ride_id"]`

3. Get the number of total rides using the `count()` function on the `ride_id` column on the `pyber_data_df` DataFrame:

   ```
   pyber_data_df["ride_id"].count()
   ```

4. Divide the total number of rides for each city type by the total rides for all the cities and divide by 100.

Add the following code to a new cell and run the cell.

```
# Calculate the percentage of rides for each city type.
ride_percents = 100 * pyber_data_df.groupby(["type"]).count()["ride_id"] / p
ride_percents
```

When you run the cell, the output of the code will be the following percentages:

```
type
Rural          5.263158
Suburban      26.315789
Urban         68.421053
```

```
Name: ride_id, dtype: float64
```

Now we're ready to create our pie chart!

5.6.2: Pie Chart for Percentage of Rides by City Type

You have another hour before you presentation and need to create two more pie charts. But you don't panic because you know you can reuse the code for creating the first pie chart. All you have to do is change one of the variables. After a quick break, you sit down and crank out the code for the percentage of rides by city type pie chart.

Using the code block from the pie chart showcasing the percentage of fares for each city type, we will create a pie chart for the percentage of rides by city type, where each wedge represents a city and its percentage of the total rides.

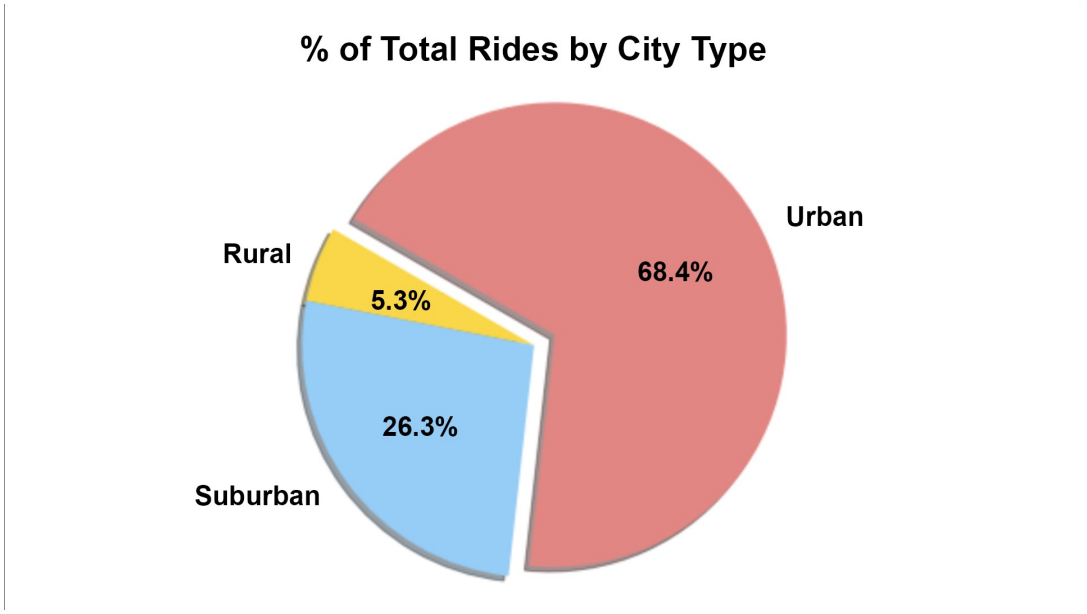
Copy the code block that created the percentage of fares for each city pie chart, and do the following:

1. Replace `type_percents` with `ride_percents`, which will represent the values for each pie wedge.
2. We'll use the same arrays for `labels` and `colors` as before.
3. We will use the same parameters, `explode`, `autopct`, `shadow=True`, and `startangle=150` as before.
4. We will change the font size with `mpl.rcParams['font.size'] = 14` as before. There is no need to import `matplotlib as mpl`, since this was already done for the previous pie chart.
5. Change the title to "% of Total Rides by City Type."
6. Save the figure as `Fig6.png`.

In a new cell, add the following code and run the cell.

```
# Build percentage of rides by city type pie chart.
plt.subplots(figsize=(10, 6))
plt.pie(ride_percents,
    labels=["Rural", "Suburban", "Urban"],
    colors=["gold", "lightskyblue", "lightcoral"],
    explode=[0, 0, 0.1],
    autopct='%1.1f%%',
    shadow=True, startangle=150)
plt.title("% of Total Rides by City Type")
# Change the default font size from 10 to 14.
mpl.rcParams['font.size'] = 14
# Save Figure
plt.savefig("analysis/Fig6.png")
# Show Figure
plt.show()
```

In the output, the final pie chart should look like this:



5.6.3: Commit Your Code

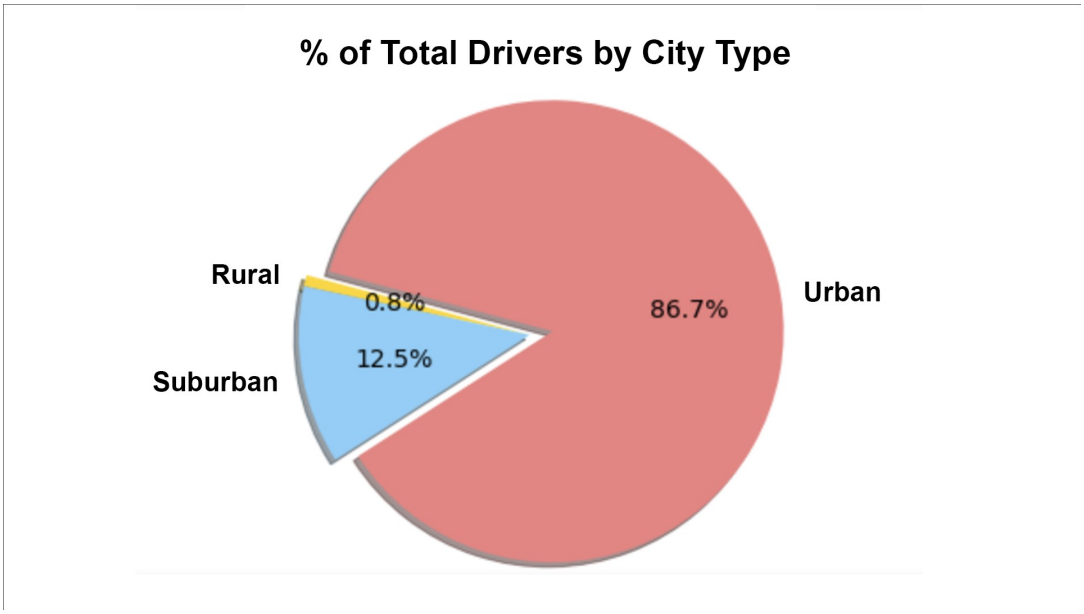
You know the drill. Let's commit the work you have done to the GitHub repository so Omar can look it over.

> **Add, Commit, Push**
>
> After you save your `PyBer.ipynb` file to your computer, add it to the PyBer_Analysis GitHub repository.
>
> 1. Launch the command line or Git Bash.
> 2. Navigate to the PyBer_Analysis folder.
> 3. Check the status using `git status`.
> 4. Add all the file(s) by using `git add .`.
> 5. Commit the files to the repository with a message by using `git commit -m "message"`.
> 6. Add the file to the repository by using `git push`.
> 7. Refresh your GitHub page to see the changes to your repository.

5.7.1: Calculate Driver Percentages

One final chart. That's it. Omar knows you are tired of creating pie charts, but like the last one all you have to do is reuse the code and change a variable. Working fast so you have time to practice your presentation, you crank out percentage of total drivers by city type.

The final pie chart will be the percentage of the total drivers for each city type, where each pie wedge will be the percentage of total drivers. The final pie chart should look similar to this:



To create this pie chart, we need to do the following:

- Get the total number of drivers for each city type.
- Get the total drivers for all the city types.

- Calculate the percentage of the total drivers for each city type.

We will calculate the `driver_percents` like we calculated the `type_percents` and the `ride_percents`.

1. Use the `groupby()` function on the `pyber_data_df` DataFrame and group by the "type" of city column.
2. Apply the `sum()` function on the "driver_count" column to get the Series with the total number of drivers for each city type.
3. Get the number of total rides using the `count()` function on the "ride_id" column on the `pyber_data_df` DataFrame.
4. Divide the Series for the total number of drivers for each city type by the number of total drivers and multiply by 100.

Add the following code to a new cell:

```
# Calculate the percentage of drivers for each city type.
driver_percents = 100 * pyber_data_df.groupby(["type"]).sum()["driver_count"
driver_percents
```

When you run the cell, the output of the code will be the following percentages:

```
type
Rural         0.781557
Suburban     12.472893
Urban        86.745550
Name: driver_count, dtype: float64
```

Now we are ready to create our pie chart!

5.7.2: Pie Chart for the Percentage of Drivers for Each City Type

It's almost presentation time. Refactor the code for the previous pie charts, change out one variable, and run the cell to save your last pie chart image. Once you add to your presentation you're all set!

Using the code block from the previous pie charts, we will create a pie chart for the percentage of drivers by city type, where each wedge represents a city and its percentage of the total drivers.

Copy the code block that created the percentage total rides for each city pie chart, and do the following:

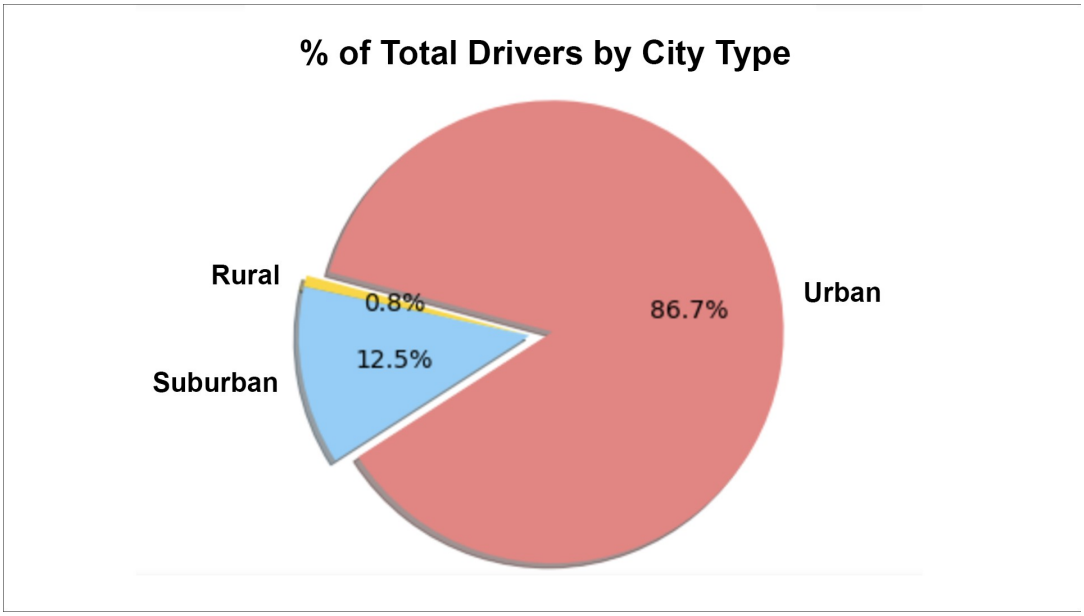
1. Replace `ride_percents` with `driver_percents`, which will represent the values for each pie wedge.
2. We'll use the same arrays for `labels` and `colors` as before.
3. We will use the same parameters, `explode`, `autopct`, and `shadow=True`, but change the `startangle` to 165.
4. We'll use the same code to change the font size as before.
5. Change the title to "% of Total Rides by City Type."
6. Save the figure as `Fig7.png`.

In a new cell, add the following code and run the cell.

```
# Build percentage of rides by city type pie chart.
plt.subplots(figsize=(10, 6))
plt.pie(driver_percents,
```

```
    labels=["Rural", "Suburban", "Urban"],
    colors=["gold", "lightskyblue", "lightcoral"],
    explode=[0, 0, 0.1],
    autopct='%1.1f%%',
    shadow=True, startangle=165)
plt.title("% of Total Rides by City Type")
# Change the default font size from 10 to 14.
mpl.rcParams['font.size'] = 14
# Save Figure
plt.savefig("analysis/Fig7.png")
# Show Figure
plt.show()
```

When you run the cell, the pie chart should look like this:



5.7.3: Commit Your Final Code

You have some time to spare before the presentation, so you make one final commit to the GitHub repository—you know that V. Isualize has a habit of asking programmers to pull up their code during the presentation in the conference room.

> **Add, Commit, Push**
>
> After you save your `PyBer.ipynb` file to your computer, add it to the PyBer_Analysis GitHub repository.
>
> 1. Launch the command line or Git Bash.
> 2. Navigate to the PyBer_Analysis folder.
> 3. Check the status using `git status`.
> 4. Add all the file(s) by using `git add .`.
> 5. Commit the files to the repository with a message by using `git commit -m "message"`.
> 6. Add the file to the repository by using `git push`.
> 7. Refresh your GitHub page to see the changes to your repository.

Module 5 Challenge

Submit Assignment

Due Feb 16 by 11:59pm **Points** 100 **Submitting** a text entry box or a website url

> Congratulations on a job well done! You knew your hard work would pay off, but you had no idea the presentation would go over that well! Omar is delighted, to say nothing of how you feel. There were a few late nights, but impressing such a demanding CEO after just a few weeks on the job was totally worth it.
>
> Your future at PyBer is looking bright. Not only did you walk out of that presentation with a spring in your step, but you have a brand-new assignment, one that will require you to use everything you learned while putting together your presentation to solve a new challenge: determining if there is a correlation between the average fare and the total rides for each city type for the individual scatter plots, and if there is any statistical significance between the different city types for each box-and-whisker plot.
>
> Now, you have been given a new challenge.

In this challenge, you will use your Python skills as well as your knowledge of the Pandas and Matplotlib libraries in Jupyter Notebook to create a summary DataFrame. You'll use Pandas functions and methods, including the `format()` and `map()` functions. You will also create a new DataFrame using a variety of Pandas methods and functions, and create a multiple-line graph from the DataFrame.

Background

You've been asked by your CEO to create an overall snapshot of the ride-sharing data. In addition to your scatter and pie charts, she would like to see a summary table of key metrics of the ride-sharing data by city type, and a multiple-line graph that shows the average fare for each week by each city type.

Objectives:

The goals for this challenge are for you to:

- Use Pandas functions like `groupby`, `pivot`, `resample`, and `reset_index` on a DataFrame.
- Use Pandas methods and attributes on a DataFrame or Series.
- Create a new DataFrame from multiple `groupby()` Series.
- Format columns of a DataFrame.
- Create a multiple-line graph.
- Annotate and apply styling to the chart.

Part 1 Instructions

Create a PyBer Summary DataFrame

Create a summary DataFrame that showcases the following for each city type:

- Total Rides
- Total Drivers
- Total Fares
- Average Fare per Ride
- Average Fare per Driver

Your final summary DataFrame should look like this:

	Total Rides	Total Drivers	Total Fares	Average Fare per Ride	Average Fare per Driver
Rural	125	78	$4,327.93	$34.62	$55.49
Suburban	625	490	$19,356.33	$30.97	$39.50
Urban	1,625	2,405	$39,854.38	$24.53	$16.57

To create the summary DataFrame, follow these steps:

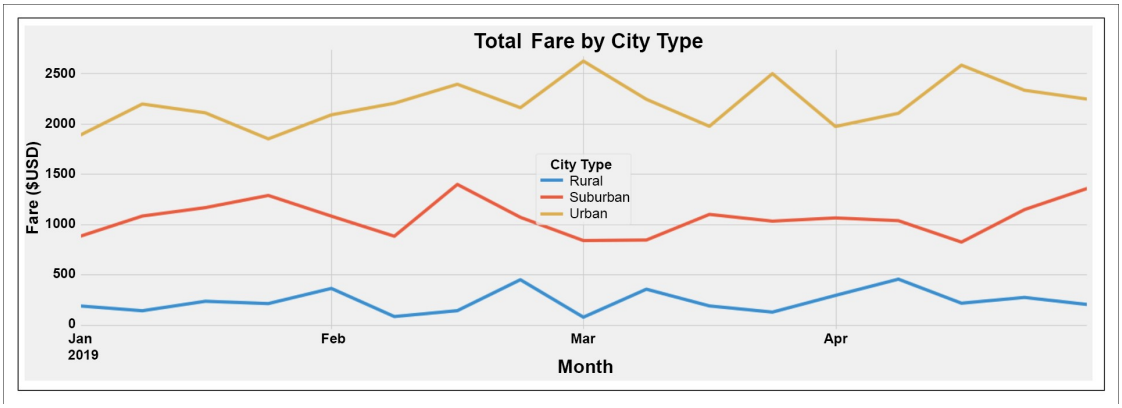
1. Get the total rides, total drivers, and total fares for each city type using the `groupby()` function on the city type using the merged DataFrame or separate DataFrames.

2. Calculate the average fare per ride and the average fare per driver by city type.
3. Delete the index name.
4. Create the summary DataFrame with the appropriate columns and apply formatting where appropriate.

Part 2 Instructions

Create a Multiple-Line Plot for the Sum of the Fares for Each City Type

For the second part of this challenge, plot the sum of the fares for each city type so that your final line chart looks like this:



After merging the DataFrames, do the following:

1. Rename columns `{'city': 'City', 'date':'Date','fare':'Fare', 'ride_id': 'Ride Id','driver_count': 'No. Drivers', 'type':'City Type'}`.
2. Set the index to the Date column.
3. Create a new DataFrame for fares and include only the Date, City Type, and Fare columns using the `copy()` method on the merged DataFrame.
4. Drop the extra Date column.
5. Set the index to the datetime data type.
6. Check the DataFrame using the `info()` method to make sure the index is a datetime data type.
7. Calculate the `sum()` of fares by the type of city and date using `groupby()` to create a new DataFrame.

8. Reset the index, which is needed for Step 10.
9. Create a pivot table DataFrame with the Date as the index and `columns = 'City Type'` with the Fare for each Date in each row. **Note:** There will be NaNs in some rows, which will be taken care of when you sum based on the date.
10. Create a new DataFrame from the pivot table DataFrame on the given dates `'2019-01-01':'2019-04-28'` using `loc`.
11. Create a new DataFrame by setting the DataFrame you created in Step 11 with `resample()` in weekly bins, and calculate the `sum()` of the fares for each week.
12. Using the object-oriented interface method, plot the DataFrame you created in Step 12 using the `df.plot()` function. Things to consider with your plotting:
    - Import the style from Matplotlib.
    - Use the graph style fivethirtyeight.
    - Add a title.
    - Add x- and y-axes labels according to the final figure.
    - Save the figure to the "analysis" folder.
    - Make the figure size large enough so it's not too small.

Complete the following tasks when you are done with both parts of the challenge.

1. Write a summary in your `README.md` file for the data in the summary DataFrame.
2. Write a summary in your `README.md` file on what the multiple line graph tells you about the fares for each city type over time.

> **Hint** Further Reading
>
> - **Using "copy" on a DataFrame** (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.copy.html)
> - **Using "info()" on a DataFrame** (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.info.html)
> - **Creating a pivot table** (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.pivot_table.html)
> - **Using "resample()" on a DataFrame** (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.resample.html)
> - **Using the fivethirtyeight graphing style** (https://matplotlib.org/3.1.1/gallery/style_sheets/fivethirtyeight.html)

> **Add, Commit, Push** When you're done, add your `PyBer_Challenge.ipynb` file and any related image files to the PyBer_Analysis GitHub repository.

Submission

To submit your challenge assignment, click Submit and then provide the URL of your PyBer_Analysis GitHub repository for grading.

Module 5 - Challenge Rubric.pdf

Note: You are allowed to miss up to two Challenge assignments and still earn your certificate. If you complete all Challenge assignments, your lowest two grades will be dropped. If you wish to skip this assignment, click Submit then indicate you are skipping by typing "I choose to skip this assignment" in the text box.

Module 5 Challenge Rubric

Criteria	Ratings					Pts
Written Report Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** Presents a cohesive written analysis that correctly summarizes and explains the information listed in the detailed rubric.	**15.0 pts** **Approaching Mastery** Presents a cohesive written analysis that correctly summarizes and explains three of the specified items listed in the detailed rubric.	**10.0 pts** **Progressing** Presents a developing written analysis that summarizes and explains three of the items as listed in the detailed rubric.	**5.0 pts** **Emerging** Presents a limited written analysis or no written analysis that summarizes and explains at least one of the items listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
Summary DataFrame Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** The notebook generates a summary DataFrame that contains all of the items listed in the detailed rubric.	**15.0 pts** **Approaching Mastery** The notebook generates a summary DataFrame that contains at least three to four of the items listed in the detailed rubric.	**10.0 pts** **Progressing** The notebook generates a summary DataFrame that contains at least one to two of the items listed in the detailed rubric.	**5.0 pts** **Emerging** The notebook does not generate a summary DataFrame, but does create a DataFrame with the correct column names.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
Summary DataFrame Formatting Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** The notebook generates a summary DataFrame that completes all of the items listed in the detailed rubric.	**15.0 pts** **Approaching Mastery** The notebook generates a summary DataFrame that completes two of the items listed in the detailed rubric.	**10.0 pts** **Progressing** The notebook generates a summary DataFrame that completes at least one to of the items listed in the detailed rubric.	**5.0 pts** **Emerging** The notebook generates a summary DataFrame that completes where none of the items are completed as listed in the detailed rubric and the values of the rows are incorrect.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
Multiple-Line Chart-Graph Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** Line chart accurately matches the line chart provided in homework description.	**15.0 pts** **Approaching Mastery** A line chart that does not match the line chart provided in homework, but contains three separate line	**10.0 pts** **Progressing** A line chart that does not match the line chart provided in homework, but contains three separate line graphs of the data	**5.0 pts** **Emerging** A line chart that does not match the line chart provided in homework, but contains one line graph of the data	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains	20.0 pts

Criteria	Ratings					Pts
		graphs of the data and time period specified in one output file	and time period specified in three separate output files	and time period specified.	evidence of academic dishonesty	
Multiple-Line Chart-Formatting Please see detailed rubric linked in Challenge description.	**20.0 pts** **Mastery** The line chart contains all of the items as listed in the detailed rubric.	**15.0 pts** **Approaching Mastery** The line chart contains two of the items as listed in the detailed rubric.	**10.0 pts** **Progressing** The line chart contains one of the items as listed in the detailed rubric.	**5.0 pts** **Emerging** The line chart contains none of the items as listed in the detailed rubric.	**0.0 pts** **Incomplete** No submission was received -OR- Submission was empty or blank -OR- Submission contains evidence of academic dishonesty	20.0 pts
						Total Points: 100.0

Module 5 Career Connection

Use Your Brand Statement to Tell Your Story

You should think about your brand statement as the means by which you tell potential employers who you are: your education, accomplishments, and skills. Your story should be brief and adaptable so that it can be tailored to your resume, LinkedIn profile, or other materials.

Take a look at Milestone 2 for a guide on developing your brand statement, and then submit a draft to your Profile Coach for review.

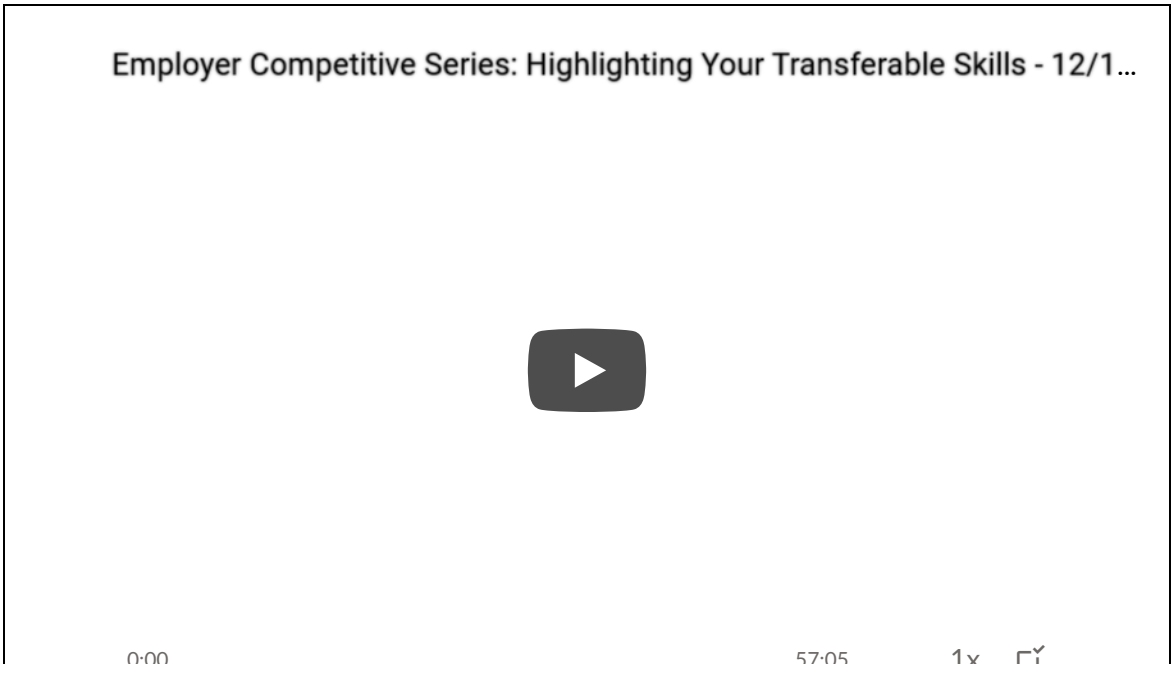
Career Services Next Step: **Link Milestone 2 (https://courses.bootcampspot.com/courses/138/pages/milestone-2-creating-your-professional-brand-statement)**

Milestone 2: Creating Your Professional Brand Statement

"A summary statement can be a powerful branding tool that helps send the message that you're the right one for the job." - [The Muse (https://www.themuse.com/advice/the-resume-summary-statement-when-you-need-one-and-how-to-do-it)](https://www.themuse.com/advice/the-resume-summary-statement-when-you-need-one-and-how-to-do-it)

Key Takeaways

- By the end of this MIlestone, you will be able to develop a draft of your professional brand statement.
- After reviewing the instructions below, develop a draft of your brand statement, and **submit to a Profile Coach at the bottom of this page** for feedback.
- For more insight on developing your brand statement , view the "Transferable Skills" workshop below.



Your brand statement expresses your professional value and provides a clear and concise introduction for employers. Think of it as the story you tell the industry about who you are, which includes your education, accomplishments, and skills. This story should be **brief** and **adaptable**, allowing it to be tailored according to its use (resume, LinkedIn summary, networking events, etc).

For a guided experience on highlighting your transferable skills and finding ways to tell your story, see the video above.

Getting Started

1. Review the brand statement criteria and samples below to develop a draft for review by a Profile Coach.
2. If this is your first time developing a brand statement, you can use our **'Step-by-Step Guide'** **(http://bit.ly/2Hecild)** for more support.
3. If you'd like additional resources on brand statements, see the resources we've included at the end of this guide.
4. Once you're done, **submit your draft for review at the bottom of this page**. A Profile Coach will offer unlimited feedback to help you develop an Employer Competitive brand statement.

Brand Statement Criteria

Your brand statement helps you develop your professional story as it relates to the role you are pursuing. You can **modify and adapt** your brand statement to other professional materials (resume, LinkedIn, portfolio, cover letter, pitch).

Use the following criteria to develop your brand statement.
Your Profile Coach will use the same set of criteria when providing feedback.

- **Concise** — Consists of 75 to 150 words. Keep it focused, and make every line count.
- **Targets role** — First line presents you in your desired role. Avoids identifying as a student (as you will use this after graduation).
    - Ex: Data Analyst, Business Analyst, Systems Engineer

- **Includes Education** — Includes **only relevant** degrees, certifications, and/or trainings. Ensure you have included the bootcamp to demonstrate your technical training.
    - Ex: Certificate in Data Visualization from XYZ University.
- **Includes Skills & Strengths** — Includes 3-5 relevant technical skills and professional strengths that align with desired role.
    - Bonus:
        - If available, you use the job description to determine which are the best skills to include.
        - Show how you have applied these skills and strengths in roles or projects.
- **Demonstrates value** — Showcases professional or academic achievements, accomplishments, major successful projects, and recognitions.
    - Hint: Aim for professional, but if you lack professional, pull from academic.
- **Includes Motivation/Aim** – Determine what motivates you professionally. What end-results do you hope to achieve in your role?
- **Positions Yourself** — Sell, don't summarize! Connect how your past experience, skills, and/or training have prepared you for your desired role. This is a place where you can give examples to support your claims.
    - Hint: To show that you are a team player, give an example of how you have worked successfully in a team environment. Show outcomes of your work.
- **Presentation**— No spelling or grammar errors. No slang. No redundant word choices. Varied sentence structures.

Sample Brand Statements

Sample 1 – Entry Level
Business analyst with a background in mathematics and newly acquired skills in Excel, VBA, Python, and SQL from Georgia Tech Professional Education. Insatiable intellectual curiosity and ability to mine hidden gems located within large sets of structured, semi-structured, and raw data. Enjoys leveraging background and skill set to support detailed and efficient analysis.

Sample 2 – Mid Level
Data architect/modeler with a certificate in Data Visualization and several years of

experience in sales and service industries. Proven technical and leadership aptitude in data warehousing and management environments. Extensive qualifications in partnering with business, leadership, and teams to define needs, evaluate risks and issues, and implement architecture, tools, and best practices to enhance decision-making and drive competitive growth. Well versed in presenting to executive staff and driving cross-functional collaboration across varied organizational subcultures, and global teams.

Sample 3 – Senior Level
Accomplished Data Scientist with a B.S. and M.S. in Computer Science and Engineering. History of success with developing machine learning predictive models, implementing efficient and high-performance applications, and collaborating with different levels of staff to exceed company expectations. Excels at leading teams to success and in designing and executing technical solutions. Made extensive positive impact within government, for-profit, and nonprofit organizations. Thrives in environments where strengths in modern analysis tools are utilized.

Additional Resources

- 3 Elevator Speech Examples for the Job Hunt **(https://www.roberthalf.com/blog/job-market/3-elevator-speech-examples-for-the-job-hunt)**
- Employers Give Elevator Pitch Advice **(https://www.youtube.com/watch?v=MFDaI30_hgs)**

(https://www.youtube.com/watch?v=MFDaI30_hgs)

- **Visit the 'Brand Statement' section of your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents)** - for answers to common questions about brand statements as well as guides and additional resources on developing a pitch.

Milestone 4: Creating Your Resume

"Job seekers have just 7.4 seconds to make an impression on recruiters, but can improve their resumes for extra attention."
– Ladders, Inc. (https://www.theladders.com/)

Key Takeaways

- By the end of this Milestone, you will be able to develop a draft of your resume.
- After reviewing the instructions below, develop a draft of your resume, and **submit to a Profile Coach at the bottom of this page** for feedback.
- For more insight on developing your resume, view the "Polishing Your Resume & Cover Letter" workshop below.

Your resume is your chance to impress an employer with your skills and accomplishments. An Employer Competitive resume focuses on **results**, uses **dynamic language**, and is **organized**, visually clean, and mistake-free.

For a guided experience on developing your resume, see the video above.

Getting Started

1. Review the resume criteria and templates below to develop a draft for review by a Profile Coach.
2. If this is your first time developing a resume, you can use our **'Step-by-Step Guide' (http://bit.ly/2HeDUFL)** for more support.
3. If you'd like additional support with resumes, see the resources we've included at the end of this guide.
4. Once you're done, **submit your draft for review at the bottom of this page**. A Profile Coach will offer unlimited feedback to help you develop an Employer Competitive resume.

Resume Criteria

Use the following criteria to develop your resume.
Your Profile Coach will use the same set of criteria when providing feedback.

Content

Heading includes all the first-order information the employer needs.

- Include name, phone number, professional email address (not hotmail, yahoo, aol, or school), city & state, zip code, written out hyperlinks to LinkedIn and Github.

Summary section. Try to include at least 3-5 of the following:

- Title of role (don't identify as a student)
- Background experience that connects to the role you are pursuing
- No pronouns
- 2-3 soft skills (ex. adaptable; time management; communication; innovative; collaborative; conflict resolution)

- Number of years of related experience (keep below 10 years)
- Accomplishments, recognitions, or awards.
- Training or certifications (bootcamp)

Highlights skills and projects completed.

- Technical Skills section
- Up to 3 of your strongest projects, with brief description, languages used, and written link to code

Experience clearly laid out, with accomplishments highlighted - not job duties.

- Experience listed in reverse chronological order, with job title, job description, company name, city & state, and dates of employment
- Start every bullet with an action verb; do not use the same verb more than once.
- Quantify your work to cite accomplishments (do not list job duties).

Education listed in reverse chronological order, with locations & certification received.

- Education listed at the end of the resume, unless you don't have a lot of experience or you have particularly relevant degrees.
- Include Boot Camp as the most recent item in education.

Passes the Applicant Tracking System.

- Include standard heading titles (Education, Projects, Technical Skills, Summary, Experience).
- Spell-out acronyms and abbreviations (abbreviated months are acceptable).
- Use bullets instead of asterisks.
- Avoid images, icons, or photographs.
- Avoid colored text.
- Avoid use of columns, tables, text boxes, or graphs.
- Ensure you are using keywords that match the job description and that align with required skills and strengths needed for the specific role resume is targeting.

Design and Format

Clean & Simple Design

- Design does not get in the way of necessary text/content
  Text fills the page without overcrowding
- Balanced margins, between 0.5” - 1”
- No more than 1 page if you are new to the field, 2 pages if you have relevant experience
- Name and headlines stand out
- Few (or no) hanging lines where just a few words take up an entire line

Consistent and Professional Text

- Font size of 11 or 12
- Consistent and professional font style (it’s okay to use different fonts for the headings and body) Professional font styles include: Arial, Calibri, Cambria, Georgia, Helvetica, Times New Roman
- Consistent use of bold, italic, and underline; same bullet point style for all lists

Correct Grammar, Spelling & Punctuation

- Consistent punctuation throughout
- No grammar errors; no spelling errors
- No personal pronouns (I, we, he or she)
- Abbreviations or acronyms are not used unless necessary

Easy to read and professional sounding tone

- No jargon, slang, or superlative adjectives like “great,” “good,” or “awesome.”

Resume Templates

Here are templates that you can use to get started on creating your resume. Simply save a copy of template sheet so you can edit and adapt it. Be sure to replace text in blue with your own, and follow the instructions in brackets. .

[Click here to view resume templates.](http://bit.ly/2WzVDh9)

Additional Resources

- [4 Winning Strategies for Tech Resumes](https://www.blueridgeresumes.com/jobseeker/2017/3/23/4-winning-strategies-for-tech-resumes)
- [6 Ways to Pass the 6-second Resume Test](https://www.youtube.com/watch?v=I_v12n1bZFc)

(https://www.youtube.com/watch?v=I_v12n1bZFc)

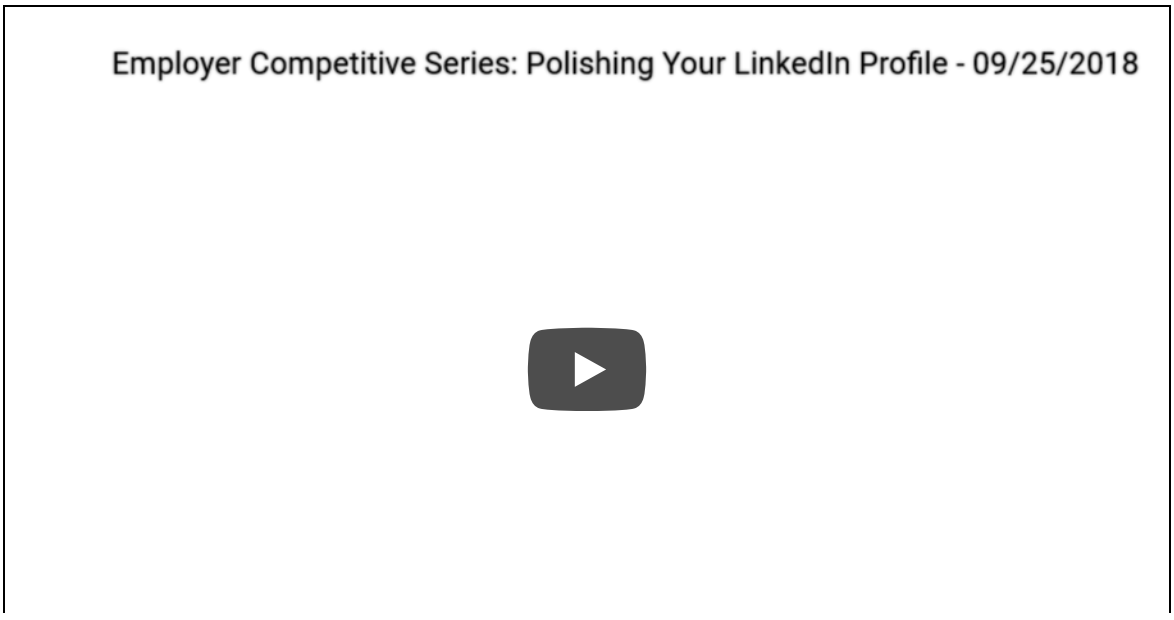
- [Visit the 'Resume' section of your Career Services Resource Library](https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents) – for answers to common questions about resumes as well as guides and additional resources.

Milestone 5: Polish Your Github and LinkedIn Profiles

"41% of employers say that they might not interview a candidate if they can't find them online."
– CareerBuilder (https://www.careerbuilder.com/share/aboutus/pressreleasesdetail.aspx?sd=4%2F28%2F2016&id=pr945&ed=12%2F31%2F2016)

Key Takeaways

- By the end of this Milestone, you will be able to develop or polish your professional online profiles.
- After reviewing the guide below, develop your Github and LinkedIn profiles, and submit to a Profile Coach at the bottom of this page for feedback.
- For more insight on building a LinkedIn profile, view the "Polishing Your LinkedIn Profile" workshop below.



Your **online materials** are just as important as your resume when it comes to getting an interview. With that, it's very important that they give a great impression and show your **professional value**. Be on your way to becoming Employer Competitive by developing strong materials that are **tailored to your target jobs** and that tell a compelling story of the value you'll add to employers.

For a guided experience on developing one of these online materials, view the "Polishing Your LinkedIn Profile" workshop video above.

Getting Started

1. Review the criteria below to develop or polish your Github and LinkedIn profiles for review by a Profile Coach.
2. If this is your first time developing your LinkedIn profile, you can use the following **'Step-by-Step Guide'** **(http://bit.ly/2WwLC4w)** for more support.
3. If you'd like more insight on developing online career materials, see the resources we've included at the end of this guide.
4. Once you're done, **submit links to your profiles at the bottom of this page for review**. A Profile Coach will offer unlimited feedback to help you develop an Employer Competitive online presence.

GitHub Profile Criteria

Employer Competitive Github

A competitive Github profile highlights projects that you're proud of, documents them through a readme and in code comments, and presents understandable and reader friendly code. It shows meaningful contributions both to individual and group projects, clear readmes, and modular, clean code.

***Use the following criteria** to develop your Github profile.*
Your Profile Coach will use the same set of criteria when providing feedback.

Professional Profile

- A photo or an image other than the default Github identicon
- Email or other contact info is listed in profile
- A descriptive tagline, e.g. “Data analyst working primarily in Python”

Clean, Well-organized Repositories

- Each repository contains one project
- The code in each repository is working
- The code is organized into an appropriate directory structure
- Each repository is appropriately named (i.e. Austin Weather Analysis vs. homework12)
- Each repository has a descriptive tagline

Readable Code

- Variables are clear
- Code uses appropriate white spacing, e.g. indentation
- Jupyter Notebook files contain comments and headings

Code Follows Technical Standards

- Code is linted for errors
- Repository contains .gitignore files when necessary

Commit Histories

- At least 200 commits by the end of the program
- Continued activity at least once per week after graduation
- No profanity in commit history
- Meaningful commit messages

Project Focused ReadMes

- Each project contains a Readme file
- Each project contains a summary that clearly states the problem you’re trying to solve

- If necessary, each project contains technical details required to run the code
- If appropriate, each project contains screenshots of the data that you've collected and explanations as to why you've displayed it in this way

Correct Grammar, Spelling & Punctuation

- Consistent punctuation throughout
- No grammar errors or spelling errors

LinkedIn Profile Criteria

Employer Competitive LinkedIn

An Employer Competitive LinkedIn profile reflects your personal brand, demonstrates your accomplishments, and showcases your interests. It offers an online presence that speaks to your professional values, and it also helps facilitate more efficient networking.

Use the following criteria *to develop your LinkedIn profile.*
Your Profile Coach will use the same set of criteria when providing feedback.

Compelling Introductory Information

- Professional profile photo.
- Customized background image.
- Up to date contact information (email).
- Catchy headline that incorporates your target role.
- Clear summary statement that speaks to your experience, background, and professional qualifications.

Easy-to-Follow Experience & Education Section

- Experience listed in reverse chronological order with job title, job description, company name, city & state, and dates of employment.
- Experience section includes accomplishments - not just job duties.

- Education section in reverse chronological order and includes Bootcamp.

Skills, Recommendations, Accomplishments & Interests Sections Provides a Fuller Picture of Who You Are

- At least 20 skills, both technical and transferable.
- At least 2 - 4 recommendations that attest to your skill set and work ethic.
- At least 20 interests displayed, with a mix of personal and professional interests.

No spelling or grammar errors

- All spelling is accurate with consistent punctuation.
- Tone consistent throughout.
- All links are working.

Note: You can customize your LinkedIn URL. In the upper-right corner of your profile, you'll see and can click "Edit public profile and URL". This is your chance to personalize your URL, which will make you easier to find through Google or Bing.

Additional Resources

- **7 LinkedIn Tips that Get You Hired (https://jobhuntingu.com/2014/01/27/7-linkedin-headline-tips-get-hired/)**
- **Github Readme Template (https://gist.github.com/PurpleBooth/109311bb0361f32d87a2#file-readme-template-md)**
- **Visit your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents)** – for answers to common questions and additional resources for developing your online professional materials.

Please paste a link to your GitHub and LinkedIn profiles below. Also, make sure your profiles are set to public.

Example Submission:

GitHub: https://github.com/yourgithubprofile

LinkedIn: https://www.linkedin.com/in/yourlinkedinprofile/

Milestone 6: Preparing for a Successful Interview

"40% of interviewers state that overall confidence was a reason for not taking their candidacy further."
– Twin Employment & Training (https://www.twinemployment.com/blog/8-surprising-statistics-about-interviews)

Key Takeaways

- By the end of this Milestone, you will be able to demonstrate successful interviewing strategies.
- Review this guide, and mark complete.
- For more insight on interviewing, click here to register for a behavioral or technical interviewing workshop. (https://careerservicesonlineevents.splashthat.com/)

Employer Competitive candidates succeed in the interview process by preparing for common questions and **practicing the stories** that illustrate their skills, unique talent, and experiences. They understand how to **confidently speak** about their technical and transferable skills in a way that **sells their professional value** to employers.

For a practice session on interviewing, click here to register for a behavioral or technical interviewing workshop. (https://careerservicesonlineevents.splashthat.com/)

Getting Started

1. Review the guide below for insight on behavioral and technical networking.

2. If you'd like additional support with interviewing, review our **'Step-by-Step Guide to Preparing for a Successful Interview'** (http://bit.ly/2Vu3ArO) or review the resources we've included at the end of this guide.
3. Once you're done, mark this milestone complete. Reminder: This Milestone **does not** require a submission to a Profile Coach.

Behavioral Interviewing

Behavioral interviews consist of concise yet detailed descriptions of your past professional accomplishments and experiences with an emphasis on how you reacted to different situations. Employers listen to your "stories" to evaluate if you have the skills and experience that meet their job requirements and fit the company culture.

The secret to nailing your behavioral interview is to have thoughtful stories that are consistent with your brand and also illustrate your strengths and professional value.

Successful Behavioral Interviewing

Your personal brand qualities are evident in responses –

- Your elevator pitch (based on your personal brand statement) should be clear and polished as you open the interview.
- The qualities you list in your elevator pitch should come through in your responses throughout the interview— use your "stories" to highlight those qualities again and again.

You use results driven language –

- You describe specific skills/accomplishments using the S.T.A.R. method (Situation, Task, Action, Result)
- These specific examples provide clear evidence of your skills and accomplishments

Your answers are clear and succinct –

- Your answers concise yet descriptive.
- You ask for a minute to think if you need it.

You've researched the company –

- You have specific questions prepared to ask about the company.
- Your questions reflect that you have listened to the conversation.
- You respond to questions with specifics about the company, which demonstrates that you've done your research.

You use positive body language –

- You lean forward, nod, track the conversation, and sustain eye contact.
- You smile at appropriate times and have a cool, confident (but not cocky) demeanor.

You are professional and prepared –

- You arrive early.
- If a phone interview, you are in a quiet place.
- You are dressed appropriately.

You follow up after the interview –

- You send a follow-up email within 24 hours to thank them for taking time to meet with you. In the personalized email, you touch on points discussed in interview and express interest in next steps.
- If you haven't done so already, you connect with your interviewers on LinkedIn.
- You follow up for an update after two weeks.
- If passed on for the position, you ALWAYS ask for feedback.

Technical Interviewing

If you're moving on to next steps in the interview process, you're probably wondering how to prepare. Here, we'll cover types of technical interviews so you can understand how to better prepare.

Before we get started:

- Figure out your strengths and gaps. If there are any coding languages that you feel you can improve on, start working on them now. (https://community.modeanalytics.com/sql/tutorial/introduction-to-sql/)

Technical Interviewer:

The Technical Interviewer is usually someone with a technical background. They assess your technical skill set and may also ask you behavioral interview questions just to get to know you a little bit better. The technical interviewer usually assesses your skill set in on of the following ways:

- **Whiteboarding Sessions:**
  Whiteboarding sessions are 30 minutes to full day interviews designed to test your critical thinking and problem solving skills. You will be given a question that has to do with coding and be asked to answer this question using a whiteboard.
- **Pair Programming Sessions:**
  Pair programming sessions are interviews that last 30 minutes to full day and are designed to test how you solve problems under pressure. You will be tasked with a coding challenge where you will have to program alongside members of the technical team.
- **Take Home Assessments:**
  Take home assessments are challenges that the technical interviewer will send you before the first or second interview. You will have to complete a program or coding challenge and then discuss your thought process with the technical interviewer.

Additional Resources

- How to Answer: Behavioral Interview Questions (https://www.youtube.com/watch?v=mH3DaeJSwy4)

(https://www.youtube.com/watch?v=mH3DaeJSwy4)

- **Using the STAR Method in Behavioral Interviews (https://www.youtube.com/watch?v=qKBubKO-798)**

(https://www.youtube.com/watch?v=qKBubKO-798)

- **What You Need to Know to Ace Your Technical Interview (https://www.glassdoor.com/blog/technical-interview-tips/)**
- **Visit the 'Prepare for a Successful Interview' section of your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents)** – for answers to common questions and other useful resources resources.

Please check the box below to verify that you have completed this milestone.

○ Yes, I have completed this milestone.

Check Answer

Finish ▸

Milestone 7: Employer Ready Reference Guide

"The secret to getting ahead is getting started."
– Mark Twain

Key Takeaways

- By the end of this Milestone, you will have revised your career materials to become Employer Ready.
- Review this guide, and **submit your revised career materials at the bottom of this page** for review.
- For more insight on getting started, view the "Starting Your Job Search" workshop below.

Now that you have created your professional materials and have received initial feedback from your Profile Coach, it's time for a final review. Submit your materials to your Profile Coach again for any last tips and suggestions on your revisions. Once you have been approved as Employer Ready, you are ready to begin working with your Career Director!

For a guided experience on getting started, see the video above.

Getting Started

1. See below for quick access to guides from previous Milestones.
2. If you'd like additional support with developing your career materials, see the resources we've included at the end of this guide.
3. Once you're done, **submit your revised career materials to a Profile Coach at the bottom of this page** for review.

GETTING STARTED

Use the following resources to develop or revise your career materials from previous Milestones.

RESUME

- **Resume Criteria and Guide (http://bit.ly/2HdA2WF)**
- **Step-by-Step Guide to Creating a Resume (http://bit.ly/2HeDUFL)**
- **Resume Templates (http://bit.ly/2WzVDh9)**

GITHUB & LINKEDIN

- **Online Materials Guide | Data (http://bit.ly/2Jx0UT9)**
- **LinkedIn Step-by-Step | Data (http://bit.ly/2WwLC4w)**

Additional Resources

- **Visit your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?**

key=datastudents) – for answers to common questions as well as guides and additional resources on developing your career materials.

Upload your file here, or submit a link to your document in the next question! (or both)

Upload files

(PDF, JPG, GIF, PNG, RTF, Word, Open Office file formats supported)

0 / 12 File Limit

Please paste a link to your document if submitting your resume as a link, and GitHub and LinkedIn profiles links below. Also, make sure your profiles are set to public.

Example Submission:

Google Resume (If you didn't upload as a file above):
https://drive.google.com/files/thiswouldbealinktoyourfile

GitHub: https://github.com/yourgithubprofile

LinkedIn: https://www.linkedin.com/in/yourlinkedinprofile/

Copy Cut Paste 0 / 75 Word Limit

Please check the box below to verify that you have completed this milestone.

○ Yes, I have completed this milestone.

Check Answer

Finish ▸

Milestone 3: Build Your Visibility

"80% of jobs never get posted and are only found through networking."
– The Muse (https://www.themuse.com/advice/6-insider-job-search-facts-thatll-make-you-rethink-how-youre-applying)

Key Takeaways

- By the end of this Milestone, you will be able to develop a plan for networking and outreach.
- Review this guide, and mark it complete.
- For more insight on networking, view the "Expand Your Network" workshop below.

Employer Competitive candidates stand out in their search by making themselves visible and selling their strengths. They're clear on their search **goals**, the **value** they're able to add to their target jobs, and the importance of **increasing their visibility** to help them reach those goals.

For a guided experience on networking, see the video above.

Getting Started

1. Review the guide below for best practices on in-person and online networking.
2. If you'd like additional support with networking, see the resources we've included at the end of this guide.
3. Once you're done, mark this milestone complete. Reminder: This Milestone **does not** require a submission to a Profile Coach.

About Networking

Successful networking involves building relationships, and it's important to note, that building a strong network doesn't happen overnight. It takes time and consistent effort, but it will be worth it!

Networking IS....

- A chance to learn more about an industry and what employers are looking for
- A chance to gain visibility and make new connections
- A chance to gain confidence in your ability to describe your interests, skills, values (which will help you in a real interview)

Networking IS NOT...

- Asking for a job
- An interview for employment
- A guarantee of employment or employability
- Just a business card swap at a meeting or conference
- Lots of connections on LinkedIn

In-Person Networking

(1) Identify Your Current Network

You may have more people in your network than you realize. Use the categories below to help you think about who's in your network and how you might reach out to them.

- Family
- Friends
- Former Colleagues/ Supervisors
- Professors, Trainers, Etc.
- Other Connections

(2) Expand your Network

Consider the following approaches to expand your network:

- **Reach out to every person on your networking list above**, and send them your materials with a specific ask. “Asks” can include a quick chat on the phone for advice or a lunch date to talk about your target industry as well as recommendations for who you should connect with next.
- If you’re currently employed, **ask your boss for projects that require you to interact with new departments or individuals**. For example, you can propose that you help the company enhance its website, and in doing so, you'll interact with other developers and/or the marketing department.
- **Find volunteer opportunities**. Get involved in an organization or group that interests you, and offer to contribute some of your new tech skills. You may meet people who can be helpful.
- **Create business cards** that include your target role, links to Github, LinkedIn, and a QB code to scan for your resume.
- Continue to use **LinkedIn weekly to connect with employees and decision makers**. Look for people who might have secondary connections to you. Send personal messages about your passions and common interests, and request informational interviews.
- **Here is a great reminder of all the many places where you can network. (http://www.jobmonkey.com/best-places-to-network/)**

(3) Attend Networking Events

It's always helpful to set a goal when attending networking events (e.g. "I will have 3 meaningful conversations that may lead to potential follow-up," or "I will not leave until I have entered into at least 5 conversations."). Establishing a goal allows you to set a measurable standard of success for the event, which can help change your experience of networking into a positive one.

TIP: Always bring business cards with you to events. On the back of the cards you receive, take notes about the person you're speaking with so that you can follow up in a personal way.

Here are some additional tips that will help you differentiate yourself at an event:

1. **Master the use of tech language** – The better your vocabulary (especially as it relates to your industry), the more impressed people will be. Being confident, articulate, and knowledgeable will help you create a strong first impression.
2. **Eye Contact** – Always maintain eye contact when you're speaking with someone. Looking away can make you appear less confident. Also, remember to smile.
3. **Leave personal space** – Don't stand too close to anyone. Keep a reasonable distance.
4. **Acknowledge your understanding** – When someone else is talking, acknowledge that you heard them with non-verbal body language such as nodding.
5. **Wait your turn** – Successful professionals are also good listeners. Allow your new connection the opportunity to complete their thoughts before offering a response.
6. **Watch body language** – Mirror the body language of the person with whom you are interacting. If they sit down, you should sit down too— they may be ready for a longer conversation. Try not to cross or fold your arms, as that may create the appearance that you are guarded. Overall, be mindful of both you and your new connection's body language.
7. **Be curious** – Open the conversation with questions. Focus on the other person's interests first, and show genuine interest.

Here are some conversation starters that might help you as well. (https://www.themuse.com/advice/30-brilliant-networking-conversation-starters)

> *"Networking is more about farming than it is about hunting. It's about cultivating relationships."*
>
> – Ivan Misner

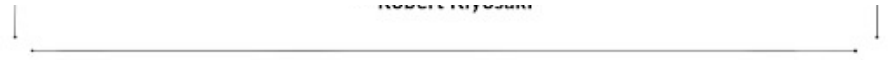
(4) Follow-Up

Networking only works if you follow up! After meeting new contacts, follow up with a personal message soon after you've met.

When reaching out, whether via email or phone, here are a few tips:

1. **Remind them how they know you**. Always begin by referencing a common person, event, educational experience, work experience, organization, or award that creates a common bond.
2. **Be clear on what you bring to the table**. Express interest in the person's work, and add value instead of asking for something. Sharing interesting articles, making introductions to helpful contacts, supporting the contact's endeavors, and engaging with their LinkedIn posts are great ways to add value.
3. **Be flexible with scheduling**. Make it easy and convenient for the contact to say yes to connecting again!
4. **Do your homework!** Research your new connections to help you better foster a relationship with them. LinkedIn and general internet searches provide instant access to information on your targeted connections.
5. **Don't give up, and don't take it personally**. Some people hesitate to reach out again for fear of being ignored, rejected, or of being a pest. It's okay if someone doesn't take you up on your offer. If you are reaching out to people regularly, you'll get more accepted invitations than passes.
6. **Breathe, and stay calm**. It's perfectly normal to be nervous about calling people. Networking is a skill that requires practice. It may help to practice your calls with friends or family. It may also help to remember that you're not calling to ask for favors — you are asking to learn from someone. Most people love sharing their expertise!

> *"The richest people in the world look for and build networks. Everyone else looks for work."*
>
> – Robert Kiyosaki



(5) Request Informational Interviews & Seek Mentors

Informational interviews are your opportunity to explore whether a your goals or current opportunities really are the right match for you. They're also great ways to expand your network through introductions from the connection you're interviewing.

Before the Interview

DO

- Research the individual — use LinkedIn, Google them, or personal connections to prepare.
- Prepare a list of questions (at least 4).
- Review a list of conversation starters for informational interviews, and have a few ready to go.
- Be ready to deliver your elevator pitch.

DO NOT

- Plan to “wing it” — while these are not job interviews, preparation is needed.
- Script every second of the interview — you need to build a relationship as well.
- Assume this person is going to lead the conversation or listen to you talk the entire time.

During the Interview

DO

- Smile, be aware of appropriate eye contact, and lean forward.
- Ask questions to demonstrate interest and active listening.
- Find a personal connection through interests, passions, or hobbies.
- Listen for ways you may be able to help or volunteer for them.
- Use varied tones and volumes to demonstrate your passion and enthusiasm.
- Describe work you have done that might be interesting.

DO NOT

- Complain about previous employers or peers.
- Dominate conversation — be sure to let them talk.
- Answer questions with one word answers — be concise, but be thorough too.
- Look at your phone during the conversation.

After the Interview

DO

- Jot down notes to remember the conversation.
- Write a thank you email.
- Follow up about once a month with updates and check ins.

DO NOT

- Follow up too frequently (more than about once per month).
- Text a thank you — this should be a more formal thank you.

NOTE: Informational interviews should lead to more interviews, volunteer or open-source projects, or ideas about new directions to take. For more on informational interviews, check out this **article called 5 Tips for Non-Awkward Informational Interviews (https://www.themuse.com/advice/5-tips-for-nonawkward-informational-interviews)** .

Online Networking

SOCIAL MEDIA

- Make sure your profiles look polished on platforms like LinkedIn, Angel.co, and any others. Examples of excellent Data profiles can be found here:
  - **https://www.linkedin.com/in/robinhchoi/ (https://www.linkedin.com/in/robinhchoi/)**
  - **https://www.linkedin.com/in/carlosmarin2/ (https://www.linkedin.com/in/carlosmarin2/)**
  - **https://www.linkedin.com/in/dylansather/ (https://www.linkedin.com/in/dylansather/)**

    - https://www.linkedin.com/in/leonardo-apolonio/ (https://www.linkedin.com/in/leonardo-apolonio/)
- On LinkedIn, Facebook, and other platforms, follow companies, thought leaders, and professionals in the industry. Learn how to do this here: http://bit.ly/2Ec5ikA (http://bit.ly/2Ec5ikA). Engage with these companies and individuals through likes and comments on their posts.
- Look for alumni groups for your current or past organizations.
- Use these templates to help you draft your outreach messages. (http://bit.ly/2vRZj24)

Additional Resources

- 10 Simple Ways to Improve Your Networking Skills (https://www.youtube.com/watch?v=E5xTbn6OnAA)

(https://www.youtube.com/watch?v=E5xTbn6OnAA)

- How to Find Your Next Job Over Coffee (https://blog.udacity.com/2014/11/informational-interviews-how-to-find.html)
- Visit the 'Build Your Visibility' section of your Career Services Resource Library (https://legacy.gitbook.com/read/book/cstrilogy/career-resources-data-analytics-library?key=datastudents) – for outreach templates, potential jobs, and additional networking resources.

Please check the box below to verify that you have completed this milestone.

○ Yes, I have completed this milestone.

Check Answer

Finish ▸