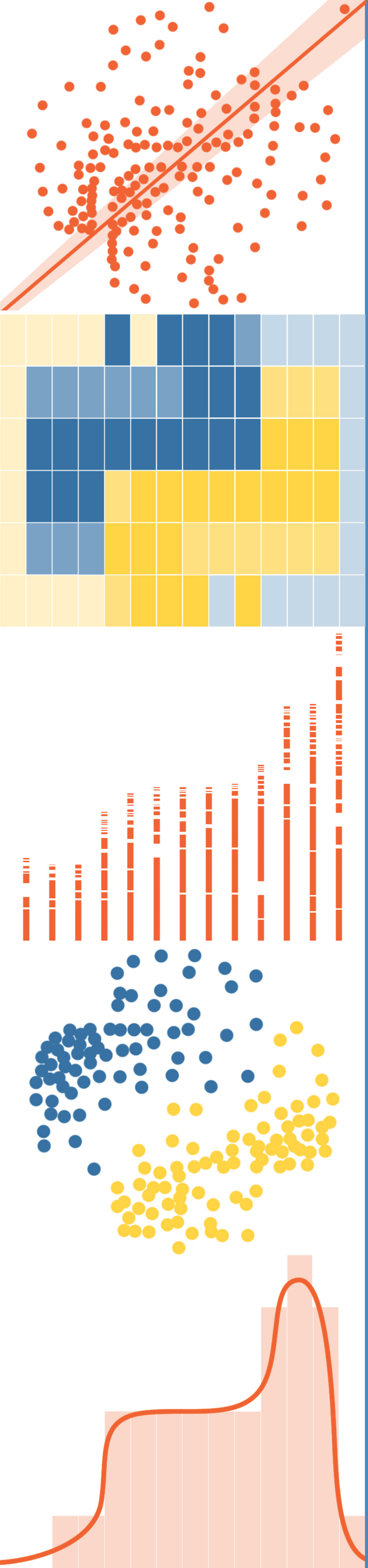


Data Visualization in Python

Daniel Nelson

StackAbuse

Explore and Manipulate Data and Create Engaging Interactive Plots with 9 Python Libraries

Data Visualization in Python

Explore and Manipulate Data and Create Engaging Interactive Plots with 9 Python Libraries

StackAbuse

Authored by Daniel Nelson
Edited by David Landup
Cover design by Jovana Ninković

First Edition: September 2020

Published by StackAbuse.com, a subsidiary of Unstack Software LLC.

The plots on the cover of this book, which vaguely represent the Python "two snakes" logo, were created using the open-source libraries described in this book. For the dataset and code used, you can find the repository on GitHub: https://github.com/StackAbuse/python-data-visualization-ebook-logo

Thank you to the Python Software Foundation for permission to use the logo in this book.

Contents

Preview

Thank you for taking the time to take a peek at our book. This was a short sample from "Data Visualization in Python" - a book for beginner to intermediate Python developers that guides you through simple data manipulation with Pandas, covers core plotting libraries like Matplotlib and Seaborn, and shows you how to take advantage of declarative and experimental libraries like Altair and VisPy.

If you've enjoyed this sample and would like to own a digital copy of the full book, you can find it at https://gum.co/data-visualization-in-python[1].

[1]https://gum.co/data-visualization-in-python

1. An Introduction To Data Visualization In Python

This book will cover the most relevant and unique attributes and features for 9 different libraries, before going on to demonstrate how to visualize data with them. This book will also cover the different types of data you can visualize in Python, in addition to common visualization techniques, tools, and plot types.

Before delving too deeply into the libraries themselves, it would be helpful to gain an intuition of how the landscape of Python's visualization libraries breaks down. To put that another way, it's helpful to understand how the different Python libraries are designed and related to one another. Understanding how the different libraries operate will help you choose the best library for your visualization project.

There are a number of different data visualization libraries and modules compatible with Python. Most of the Python data visualization libraries can be placed into one of four groups, separated based on their origins and focus.

The groups are:

- **Matplotlib-based libraries**
- **JavaScript libraries**
- **JSON libraries**
- **WebGL libraries**

Matplotlib-based Libraries

The first major group of libraries is those based on *Matplotlib*. Matplotlib is one of the oldest Python data visualization libraries, and thanks to its wealth of features and ease of use it is still one of the most widely used one. Matplotlib was first released back in 2003 and has been continuously updated since.

Matplotlib contains a large number of visualization tools, plot types, and output types. It produces mainly static visualizations. While the library does have some 3D visualization options, these options are far more limited than those possessed by other libraries like *Plotly* and *VisPy*. It is also limited in the field of interactive plots, unlike *Bokeh*, which we'll cover in a later chapter.

Because of Matplotlib's success as a visualization library, various other libraries have expanded on its core features over the years. These libraries are *Matplotlib-based*, using Matplotlib as an engine for their own visualization functions.

The libraries based upon Matplotlib add new functionality to the library by specializing in the rendering of certain data types or domains, adding new types of plots, or creating new high-level APIs for Matplotlib's functions.

They're used *alongside* Matplotlib, not *instead*, to expand its styling and plotting capabilities.

JavaScript-based Libraries

There are a number of *JavaScript-based* libraries for Python that specialize in data visualization. The adoption of HTML5 by web browsers enabled interactivity for graphs and visualizations, instead of only static 2D plots. Styling HTML pages with CSS can net beautiful visualizations.

These libraries wrap JavaScript/HTML5 functions and tools in Python, allowing the user to create new interactive plots. The libraries provide high-level APIs for the JavaScript functions, and the JavaScript primitives can often be edited to create new types of plots, all from within Python.

JSON-based Libraries

JavaScript Object Notation (JSON) is a data interchange format, containing data in a simple structured format that can be interpreted not only by JavaScript libraries but by almost any language. It's also human-readable.

There are various Python libraries designed to interpret and display JSON data. With JSON-based libraries, the data is fully contained in a JSON data file. This makes it possible to integrate plots with various visualization tools and techniques.

WebGL-based Libraries

The WebGL standard is a graphics standard that enables interactivity for 3D plots. Much like how HTML5 made interactivity for 2D plots possible (and plotting libraries were developed as a result), the WebGL standard gave rise to 3D interactive plotting libraries.

Python has several plotting libraries that are focused on the development of WebGL plots. Most of these 3D plotting libraries allow for easy integration and sharing via Jupyter notebooks and remote manipulation through the web.

Other Libraries

There are also a variety of other Python plotting libraries, many of which create Python wrappers for other languages and visualization platforms.

Popular Python Data Visualization Libraries

This book will cover the most popular data visualization libraries for Python, which fall into the five different categories defined above. The libraries covered in this book are: *Matplotlib, Pandas, Seaborn, Bokeh, Plotly, Altair, GGPlot, GeoPandas, and VisPy.*

You'll need to know what these different libraries are capable of, in order to choose the proper library for your project's needs. Let's take a quick look at these different libraries, some of their unique distinctive features, and what they're used for.

Matplotlib-based Python Libraries

Matplotlib

As already stated above, *Matplotlib* is one of the most common and widely used visualization libraries, used to create static 2D plots, although it does have some support for 3D visualizations. Matplotlib is structured in a fashion that allows the user to create and customize multiple plots for a single image, achieved through the creation of subplots. It's intended to make producing both simple and advanced plots straightforward and intuitive and has support for both static and interactive visualization modes. Though, it's relatively limited when it comes to interactive visualization.

Matplotlib is able to generate numerous different plot types and styles, and it can work along with general-purpose Python GUI libraries like *Qt* and *Tkinter*.

Pandas

Pandas is a data analysis and manipulation library. While Pandas does come with some visualization and plotting functions, the main reason Pandas is so popular and widely used is that the library makes manipulating data simple and straightforward. Pandas can read data in many different formats, and it creates a Python data object filled with rows and columns, called a `DataFrame`.

These rows and columns are easy to manipulate through built-in functions that let the user merge, split, view, filter, sort, and otherwise alter the data within them, all done with relatively simple commands.

For these reasons, Pandas is frequently used alongside the other data visualization libraries - to prepare the data in question for analysis.

Seaborn

Seaborn is a visualization library that adds onto Matplotlib's basic functions. Seaborn is intended to enable the easy creation of informative and attractive visualizations. Seaborn gives the user more control over their plots, letting them do things that aren't possible with normal Matplotlib.

This includes the ability to easily produce less common types of visualizations such as heatmaps, violin plots, and joint plots, amongst other plots. Seaborn's goal is to abstract away many of Matplotlib's low-level functions and methods, letting the user create visually impressive plots with less code compared to Matplotlib.

Seaborn gives you more customization options for your plots as well, allowing you to use preset themes or customize the plots to your liking. It also enables efficient handling of dataframes and time-series data.

GeoPandas

GeoPandas is an extension to the Pandas plotting library designed to make it easier to work with geospatial/geographical data. GeoPandas enables the types of data manipulation possible in Pandas on geometric data, letting you easily carry out visualization tasks that would typically require a spatial database.

GeoPandas allows you to specify the shape of graph regions using special shapefiles, and to clip points and lines to the boundary mask.

JavaScript-based Libraries

Bokeh

Bokeh is a visualization library that allows the user to create interactive visualizations that can be displayed in Jupyter notebooks and web browsers. Bokeh is focused on the production of highly interactive visualizations, unlike Matplotlib which has just a handful of interactive options. Visualizations in Bokeh are based around objects called *"glyphs"*, which you can render in numerous different shapes and styles.

Bokeh lets you choose different tools to include alongside your visualization. These tools let you select groups of data points, hover over points to see more information about them, zoom in on multiple graphs at once, and more.

It also allows you to construct numerous different plots with various styles, all the while maintaining high performance across large datasets. Bokeh supports HTML formatting and exporting and has native Pandas integration, allowing you to edit dataframes and the resulting visualizations easily.

With Bokeh, it's easy to create a well-styled interactive HTML file which you can then embed into a page or presentation.

Plotly

Like Bokeh, *Plotly* is *designed specifically* with the purpose of creating interactive plots. Plotly supports numerous use cases like statistical, geographic, scientific, and even 3D datasets. Similar to Bokeh's use of glyphs, the fundamental unit of a Plotly plot is the *"trace"*. You can combine multiple traces and display them all on a single figure.

Plotly for Python is based on JavaScript's Plotly library and it can be used to create more than 40 different types of plots and charts, each of which can be displayed in a Jupyter notebook or saved in an HTML file. Plotly allows the user to save their plots in the cloud or as a file on their device.

Plotly plots are interactive by default, and they can be created with JSON charts as well as easily embedded in web pages. You can also export Plotly graphs in a variety of different formats, such as PNG, SVG, PDF, and HTML to your local machine.

JSON-based Libraries

Altair

Altair is a Python library designed explicitly for the visualization of statistical data. Altair is based on the Vega and Vega-Lite standards, meaning that you use *visualization grammar* (specific phrases) that allow you to specify the level of interactivity and style you want your graph to have. Vega specifications are used to define how interactive visualizations are created in *JavaScript Object Notation* (JSON). Altair is a declarative library, and all you need to do is declare which kind of graph you'd like to create along with some desired features for it.

With Altair, you can produce effective visualizations with minimal code. You can often create complex plots with just a single line of code. However, Altair does lack some of the more advanced customization features of the other libraries.

Altair is designed to quickly create interactive statistical visualizations that can be integrated with *IPython* notebooks. Altair also lets you create compound charts comprised of different layers.

WebGL-Based

VisPy

VisPy is a 2D and 3D visualization library, created primarily to assist in the visualization of big data. Unlike the other libraries mentioned here, VisPy makes use of *Graphics Processing Units* (GPUs) to display the visualization of large datasets.

VisPy supports visualizations of scientific and statistical plots featuring millions of data points. It's intended to be scalable, easy to use, and fast. With having both low-level and high-level interfaces, VisPy makes it possible to create visualizations with relatively few lines of code and then edit those visualizations to your needed specifications.

It has OpenGL support, on which it currently bases some of its functionality, though it does require knowledge of the *OpenGL Shaders Language* (GLSL) to use.

Other

GGplot

GGplot is intended to make producing plots simple and efficient, rendering them with minimal code. It uses the *"Grammar of Graphics"* standard, borrowed from R. GGplot graphs contain consistent basic elements, which makes graphs uniform and easy to read.

GGplot lets you perform aesthetics mapping, meaning that you can control how variables within your dataset are mapped onto visual properties, defining mappings for different variables and layers of your graph.

4. Matplotlib

Matplotlib is the most widely used data visualization and plotting library in all of Python. In fact, as we've said before, many of the other libraries in this book utilize attributes of Matplotlib to display the plots they generate.

Much of Matplotlib's popularity comes from the fact that it is highly customizable, with users able to edit almost every aspect of a Matplotlib plot.

Matplotlib plots are comprised of a *hierarchy of objects*. At the top level of the plot, the `Figure` is what contains the rest of the plot elements. The intermediate and lower level plot elements are objects and elements like the `Axes`, `Labels`, `Ticks`, and `Legends`. All of these elements can be tweaked by the user.

In this section, we'll cover the features of Matplotlib, and when you would want to use it. We'll then move on to covering the layout and elements that comprise a Matplotlib plot, demonstrating how to customize these elements.

We'll then go over some examples of the visualizations that you can create with Matplotlib.

Features of Matplotlib

One reason for Matplotlib's enduring popularity is the fact that every element of a Matplotlib plot can be customized. Plots in Matplotlib are all based on `Figures`. The `Figure` is the whole window which holds a single plot or even multiple plots.

Within the `Figure`, various elements like `Axes`, `Lines`, and `Markers` can be created. Aspects like the size and angle of the plot's ticks, the position of the legend, and the thickness of lines can all be manipulated.

Matplotlib also allows you to create multiple plots within a single figure, with subsequent plots being referred to as subplots.

It offers support for both interactive and static visualization modes. When Matplotlib graphs are rendered as interactive graphs, they have to be displayed with one of a few different graphical user interface platforms like Qt, Tkinter, or WxWidgets.

When the visualization is saved to a drive as a file, the visualization is considered to be a hardcopy backend, which are noninteractive. Matplotlib can render visualizations in various file formats such as *JPG, PNG, SVG*, and *GIF*.

Matplotlib is best used for exploratory data analysis and for producing static plots for scientific publications. Matplotlib's core of features lets you quickly explore data for interesting patterns and render simple, static visualizations for reports.

However, if you need to produce interactive visualizations, visualize big data, or produce plots for inclusion in graphical user interfaces, you may be better off using one of the other libraries covered in this book.

Matplotlib supports both simple and complex visualization options. You can use a series of pre-set options to create visualizations, or you can create your own figures and axes that you can customize to your liking.

Anatomy and Customization of a Matplotlib Plot

As previously mentioned, one of Matplotlib's most loved features is that it lets the user customize just about every aspect of the plots it generates. It's important to understand how Matplotlib plots are constructed so that you can edit them to your liking.

For that reason, we'll spend some time covering the anatomy and structure of a Matplotlib plot:

- ***Figure*** - The figure is what contains all of the other elements of the plot. You can think of it as the canvas that all of the elements of the plot are painted on.
- ***Axes*** - Plots have X and Y axes, with one variable located on the X-axis and one variable on the Y-axis.
- ***Title*** - The title is the description given to the plot.
- ***Legend*** - contains information regarding what the various symbols within the plot represent.
- ***Ticks*** - Ticks are small lines used to point to different regions of the graph, mark specific items, or delineate different thresholds. For example, if the X-axis of a graph contains the values 0 to 100, ticks may show up at 0, 20, 40, 60, 80, and 100. Ticks run along the sides, as well as the bottom, of the graph.
- ***Grids*** - Grids are lines in the plot's background that make it easier to distinguish where different values on the X and Y axes intersect.
- ***Lines/Markers*** - Lines and markers are what represent the actual data within a plot. Lines are typically used to graph continuous values, while markers/points are used to graph discrete values.

Now that we've covered the elements of a Matplotlib plot, let's take some time to examine how you can customize these different attributes and components.

Plotting and Plot Customization

Creating a Plot and Figure

Plotting in Matplotlib is done with the use of the PyPlot interface, which has MATLAB-like commands. You can create visualizations with either a series of presets (the standard way), or you can create figures and axes to plot your data on yourself. We'll cover the simple way of creating plots first and then we'll go into how you can create customizable plots.

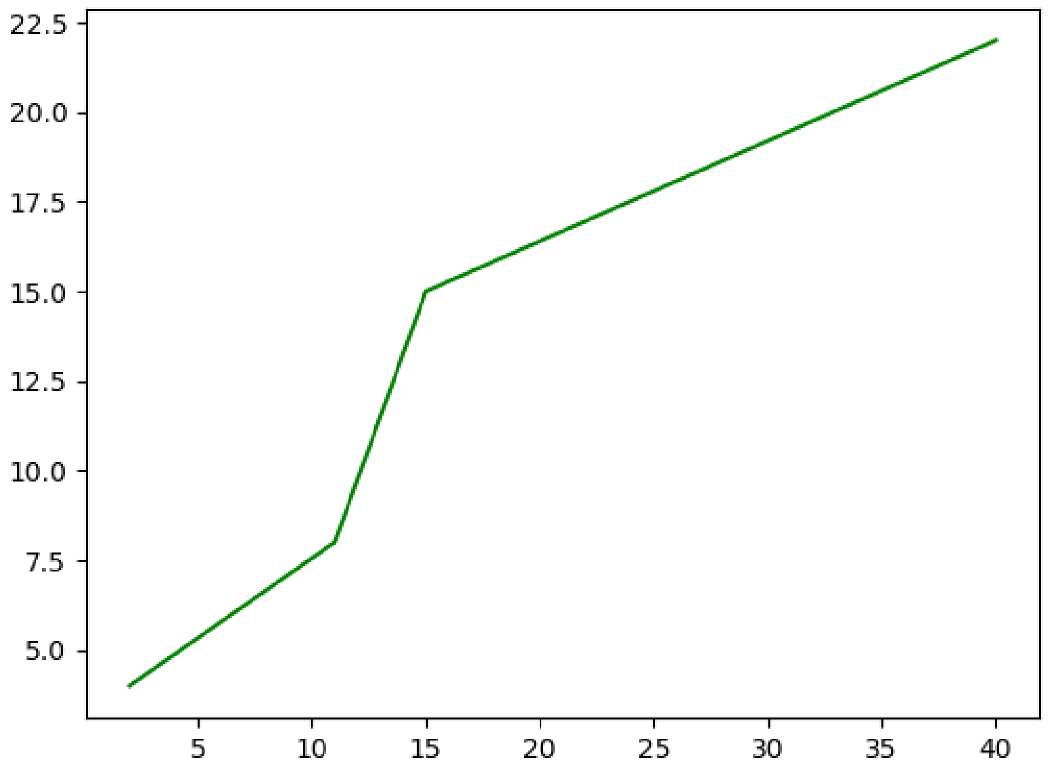
PyPlot allows the user to quickly generate professional, standardized plots with just a few lines of code.

First, we'll import `matplotlib` and the `pyplot` module. After importing the PyPlot module, it's very simple to call any one of a number of different plotting functions and pass the data you want to visualize into the desired plot function.

Then we'll create a simple plot will some random numbers. When we create plots in Matplotlib, the first set of values are those on the X-axis, while the second set of numbers is the Y-axis values.

It is possible to plot with just the X-axis values, as Matplotlib will use default values for the Y-axis. You can also pass in a color for the lines:

```python
import matplotlib.pyplot as plt
plt.plot([2, 11, 15, 40], [4, 8, 15, 22], color='g')
plt.show()
```



The `plot()` function actually constructs the plot with its elements. The `show()` function is what displays the plot to us when we run the code.

Pyplot mimics aspects of MATLAB's plotting style, meaning that you can style the plot with a series of style commands. One of the style commands is `color`, which we saw above.

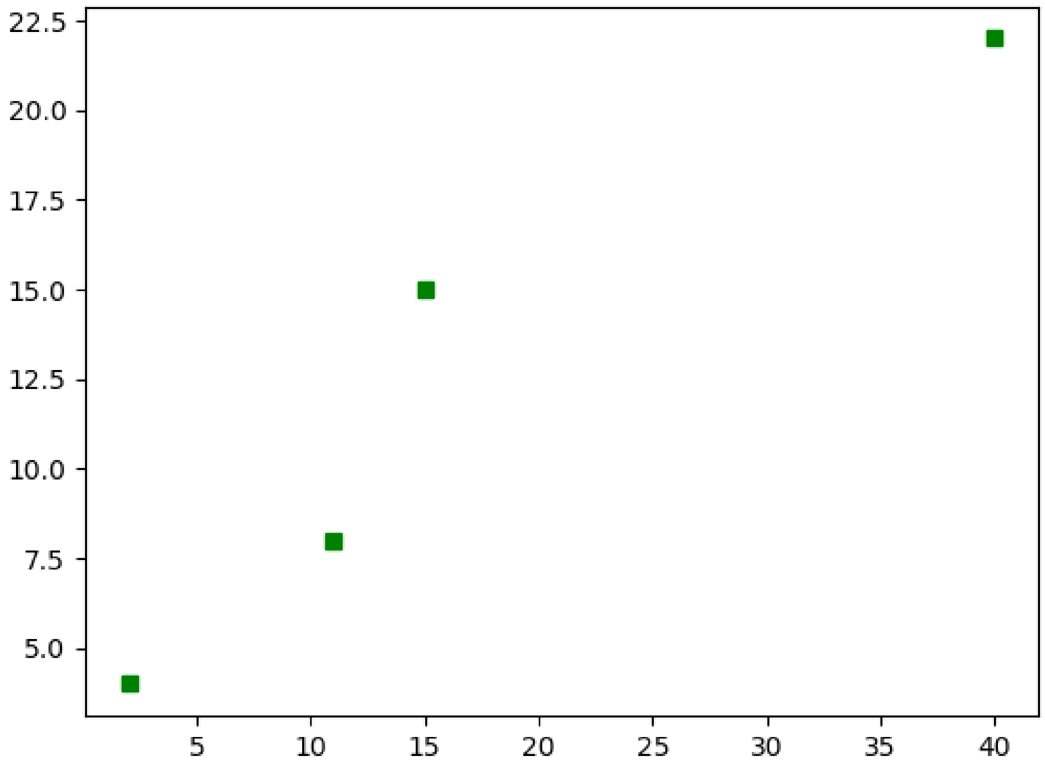
You can also change the symbols used to plot the variables. By default, a solid line is drawn, but you can select other symbols like circles, squares, or triangles.

You can pass the color and symbol instructions in as the third argument of the call to construct the plot. You can view some of the various options for plotting symbols here[2].

You can use `--` to create dashes, `s` for squares, or `^` for triangles. For colors, you can use `r` for red, `b` for blue, and `g` for green.

Here's how we could create a plot with *green squares*:

```python
plt.plot([2, 11, 15, 40], [4, 8, 15, 22], 'gs')
plt.show()
```



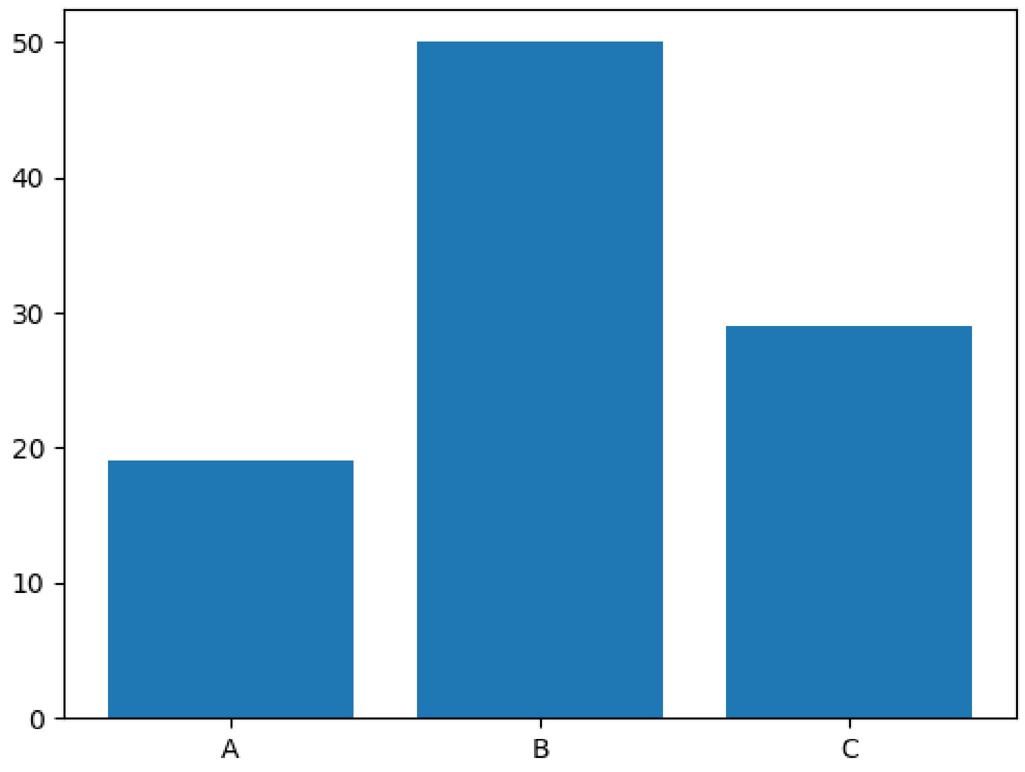
[2]https://matplotlib.org/3.2.2/api/markers_api.html

The plots we made above were continuous variables, now we'll explore how to create plots using categorical variables.

You can plot categorical variables by specifying the different categories and values in the form of lists and then passing those variables to the adequate plotting function. For example, bar charts are commonly used for categorical values.

Let's create and plot a bar chart:

```python
names = ['A', 'B', 'C']
values = [19, 50, 29]

plt.bar(names, values)
plt.show()
```



There's an alternate way of creating plots with Matplotlib. The method above allows you to quickly create plots, but if you want more control over how the plot is created, you can create a `Figure` object yourself.

Without creating a `Figure` object, Matplotlib creates a default one for you, with the default settings. To change them, you can use the `figure()` function of the `pyplot` module to create a figure and then specify some properties. For example, you can set the dimensions of the figure you want to create. The dimensions are passed in using a list with four values between `0` and `1`.

The four numbers specify the dimensions in this order: *left, bottom, width, height*. You can also do this with the `add_subplot()` function, discussed below.

Let's create a figure and add some information regarding the axes.

Axes

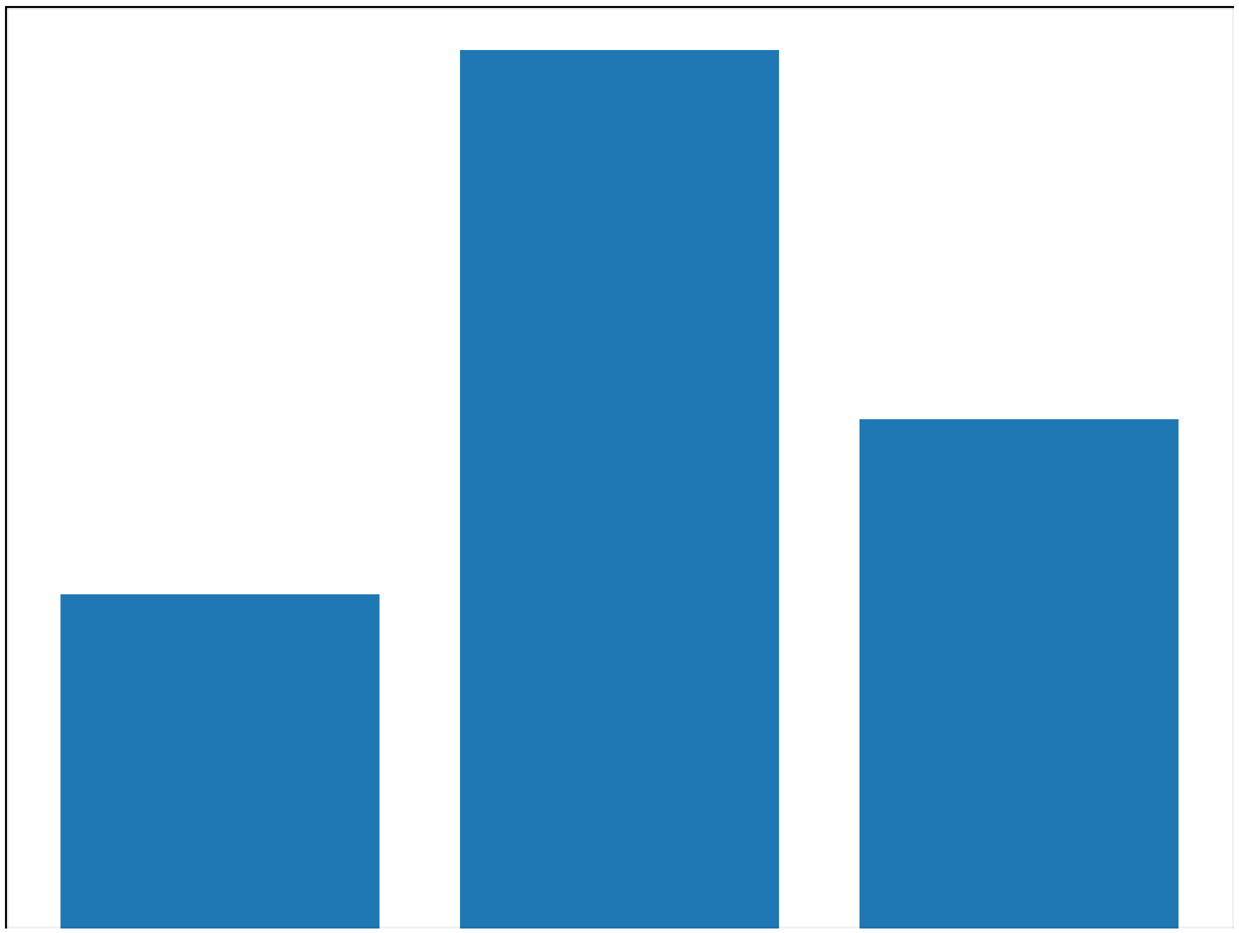
`Axes` objects sit within the figure you have created. Creating an axes object will give you greater control over how data is visualized and other elements of the plot are created.

When using an axes object, you can control how individual subplots are displayed on those axes. You can think of it like this: Figures hold axes and every axes object can store its own plots. The `Axes` instance will contain most of the elements of a figure and you can have multiple `Axes` for a single figure.

These elements include ticks, lines, text, polygons, etc. We'll explore how to change these elements throughout the *Customizing a Plot* section, up ahead.

For now, let's just create an axes object on a figure:

```
fig = plt.figure()
ax = fig.add_axes([0, 0, 1, 1])
names = ['A', 'B', 'C']
values = [19, 50, 29]
ax.bar(names, values)
plt.show()
```



The `fig.add_axes()` function returns a new `Axes` object which we've packed in `ax`. Using this object, we'll be adding elements. For example, we've called `ax.bar()` to plot a bar graph instead of calling `plt.bar()` like before.

`ax` belongs to the `fig` so everything added to the `ax` will also be added to the `fig`.

The arguments we've passed to the `add_axes()` function were `[0, 0, 1, 1]`. These are the `left`, `bottom`, `width`, and `height` of the `ax` object.

The numbers are fractions of the figure the `Axes` object belongs to, so we've told it to start at the bottom-left point (`0` for `left` and `0` for `bottom`) and to have the same height and width of the parent figure (`1` for `width` and `1` for `height`).

We can't really *see* the `ax` at this point, other than the plot is missing some elements as opposed to the previous example where they were set to default.

You can also delete axes through the use of the `delaxes()` function:

```
fig.delaxes(ax)
```

Now that we know the general method for creating plots in Matplotlib, let's take a look at the many options you have at your disposal for customizing these plots.

Subplots

Matplotlib allows you to create multiple plots within the same figure. In order to add multiple plots, you need to create a “subplot” for each plot in the figure you’d like to use.

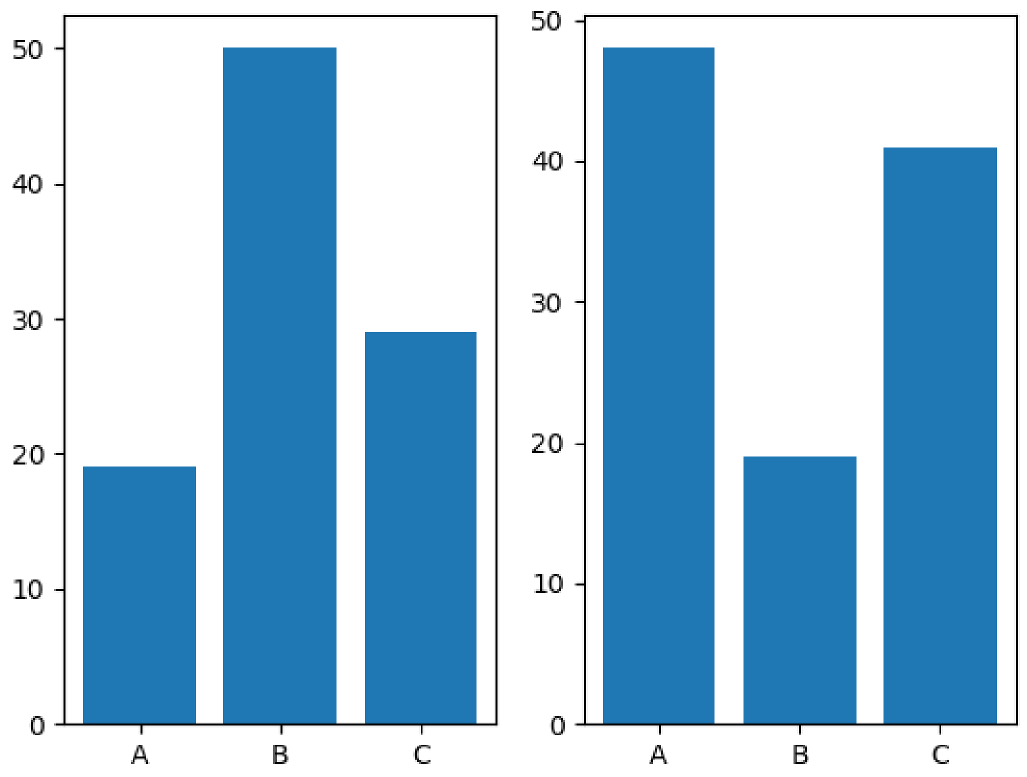
This is done with the `add_subplot()` function, which accepts a series of numeric arguments.

The first number specifies how many rows you want to add to the figure, the second number specifies how many columns you want to add, and the third number specifies the number of the plot that you want to add.

This means that if you in passed in `111` into the `add_subplots()`function, one new subplot would be added to the figure. Meanwhile, if you used the numbers `221`, the resulting plot would have four axes with two columns and two rows - and the subplot you’re forming is in the 1st position.

Here’s how we would create two subplots in the same figure, notice that we have created two axes objects:

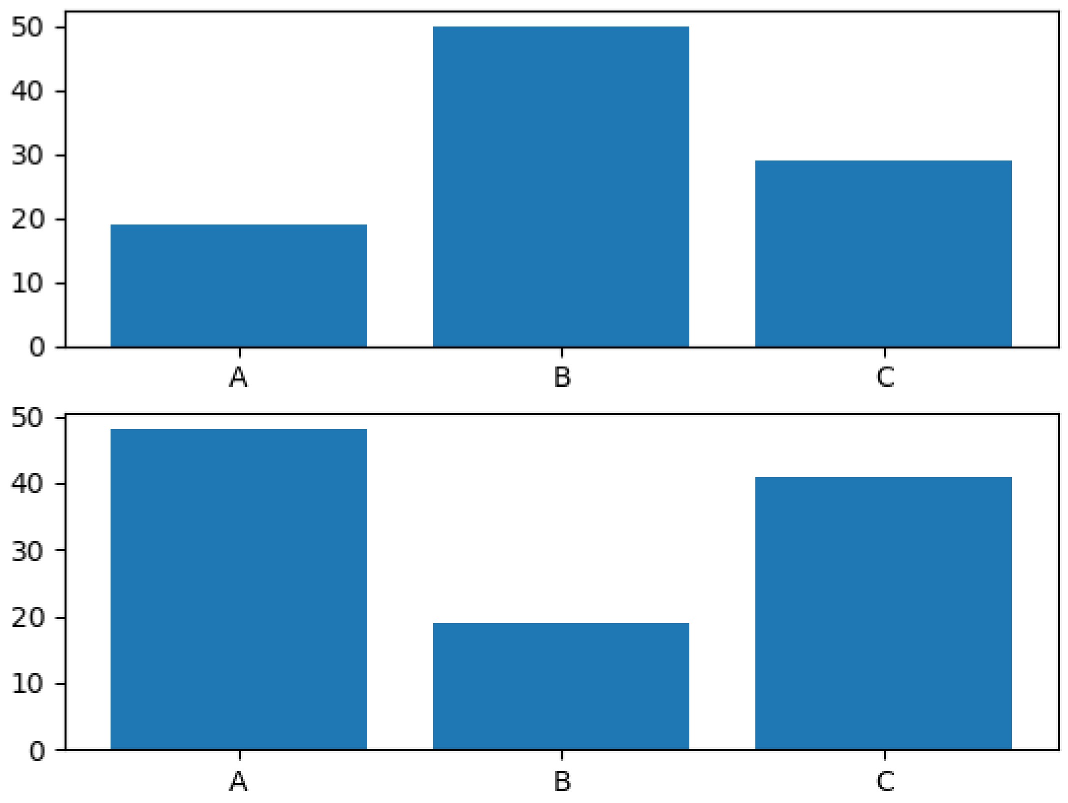
```
fig = plt.figure()

names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [48, 19, 41]

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

ax.bar(names, values)
ax2.bar(names, values_2)
plt.show()
```



We’ve created two sublots in a figure with 1 row and 2 columns. They’re sitting side by side. If we had created a figure with 2 rows and 1 column:

```
fig = plt.figure()

names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [48, 19, 41]

ax = fig.add_subplot(211)
ax2 = fig.add_subplot(212)

ax.bar(names, values)
ax2.bar(names, values_2)
plt.show()
```

We’d be looking at something like this:



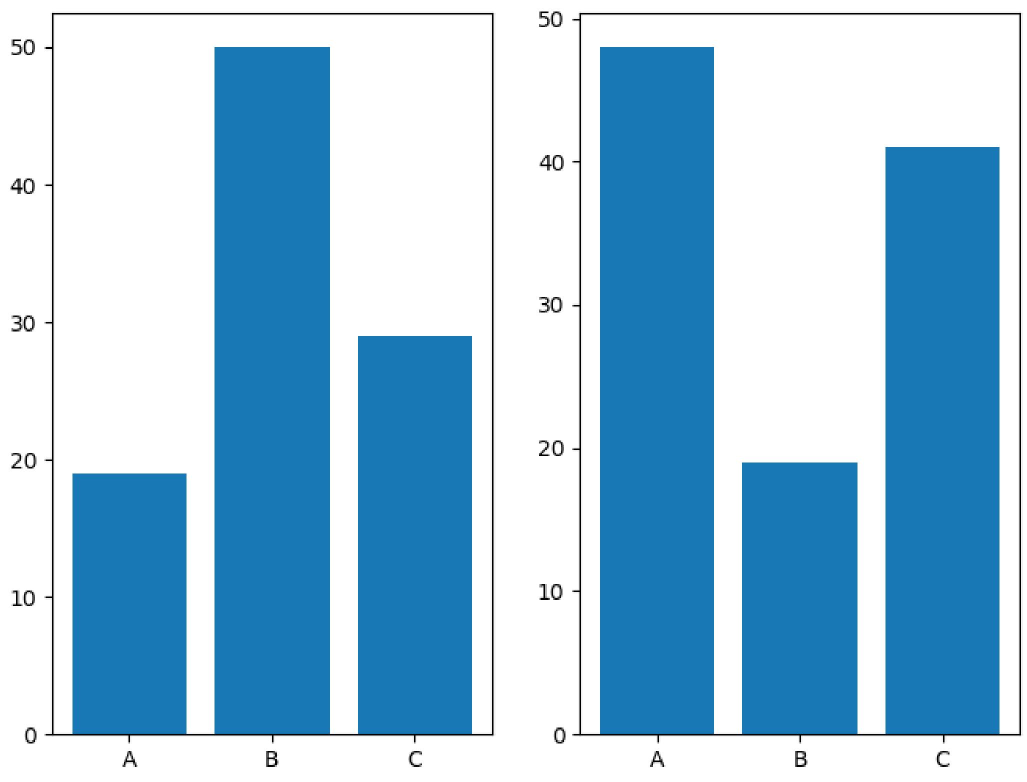
Changing Figure Sizes

As you add more subplots and details, the figure might end up becoming pretty cramped and hard to read. You'll want to be able to change the size of your figure to best match how your data is displayed.

You can alter the size of your visualization by passing a `figsize` argument to your `figure()` function. You can also use the `figsize` argument along with the `subplots()` function, allowing you to adjust the size of individual subplots.

For instance, here is how you would create an 8x6 figure:

```python
names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [48, 19, 41]

fig = plt.figure(figsize=(8.0,6.0))

# Adds subplot on position 1
ax = fig.add_subplot(121)
# Adds subplot on position 2
ax2 = fig.add_subplot(122)

ax.bar(names, values)
ax2.bar(names, values_2)
plt.show()
```



Note that the `figsize` is set in *inches*. This means that the plot we just created is 8 inches in width and 6 inches in height.

There is no native way to use the metric system in this case, though, you can define a function that converts centimeters to inches:

```
def cm_to_inch(value):
    return value/2.54
```

And then adjust the size of the plot like this:

```
fig = plt.figure(figsize=(cm_to_inch(10),cm_to_inch(15)))
```

Customizing A Plot

We've covered how to create plots and add the `Axes` object to a `Figure` which allows us further customization. Now, let's use that object to make some finer adjustments to the plots we're working with.

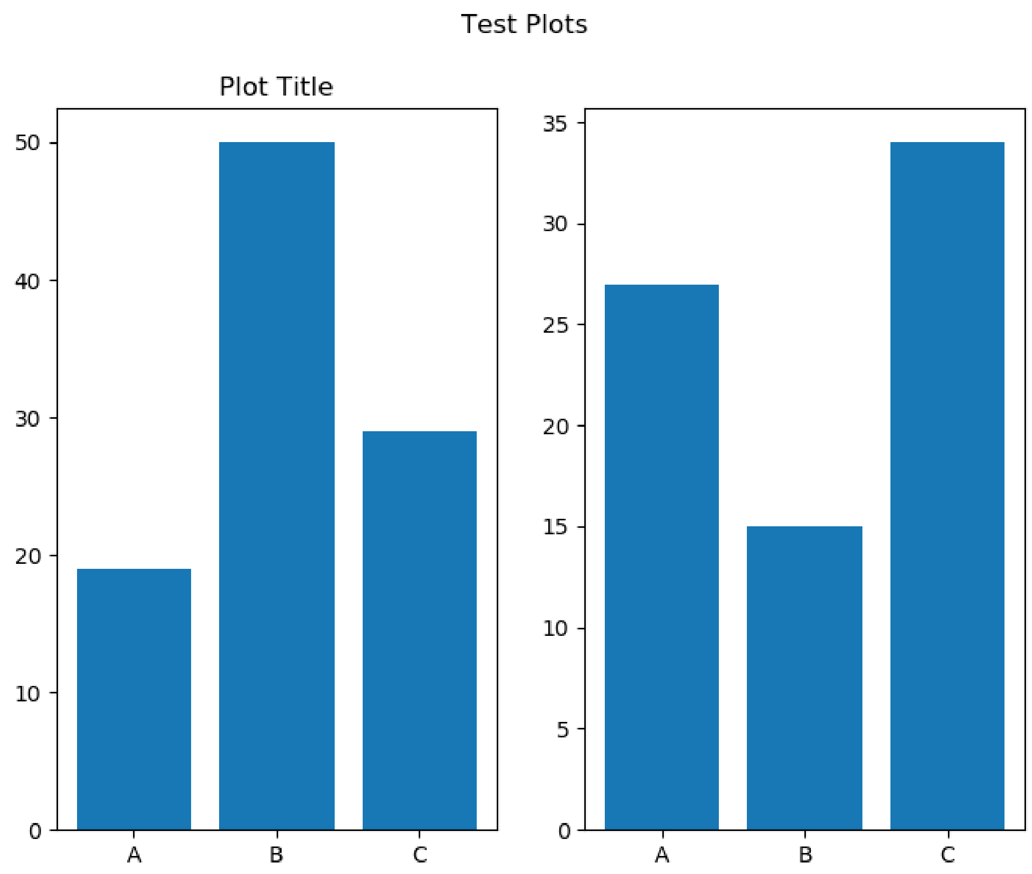
You can customize things like markers, ticks, line widths, line styles, legend, text, and annotations.

Plots Titles

You can specify the title of a plot by using either the `set()` function and passing in the `title` argument, or by using the `set_title()` function.

If you are using figure objects you'll want to use the `suptitle()` function to control your plot titles:

```
names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

fig = plt.figure(figsize=(8.0,6.0))

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
# Sets the title of the sublot on position 1
ax.set_title('Plot Title')

ax.bar(names, values)
ax2.bar(names, values_2)
# Sets the title of the entire figure
plt.suptitle('Test Plots')
plt.show()
```



Labels and Legend

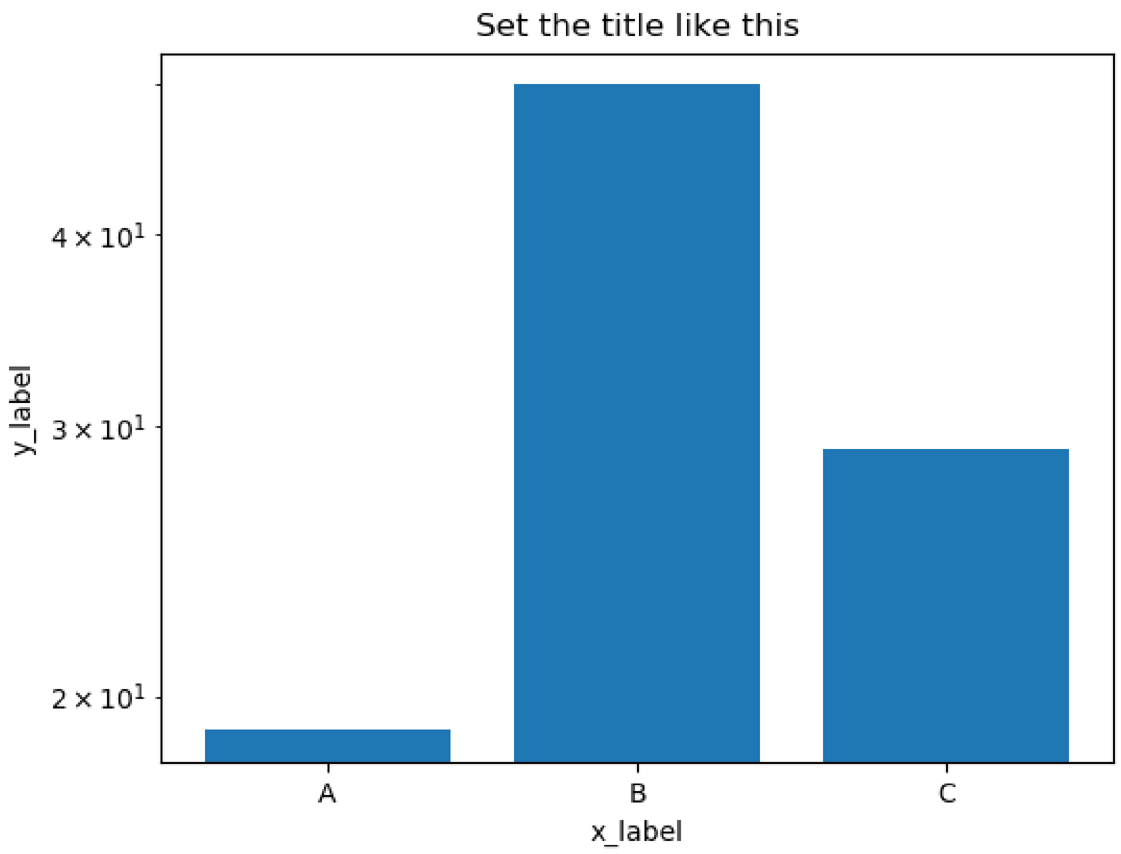
Much like you can set the title of your plot, you can also label the individual axes of your plot. If you've been using the default plotting scheme you can label the axes with the `xlabel()` and `ylabel()` functions respectively.

However, you can also use the `set()` function on your axes object, or use `ax.set_xlabel()` and `ax.set_ylabel()` to set them individually:

```
names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.bar(names, values)
plt.suptitle('Test Plots')
plt.show()
```

Here's how we would set them using `set_xlabel()` and `set_ylabel()`:

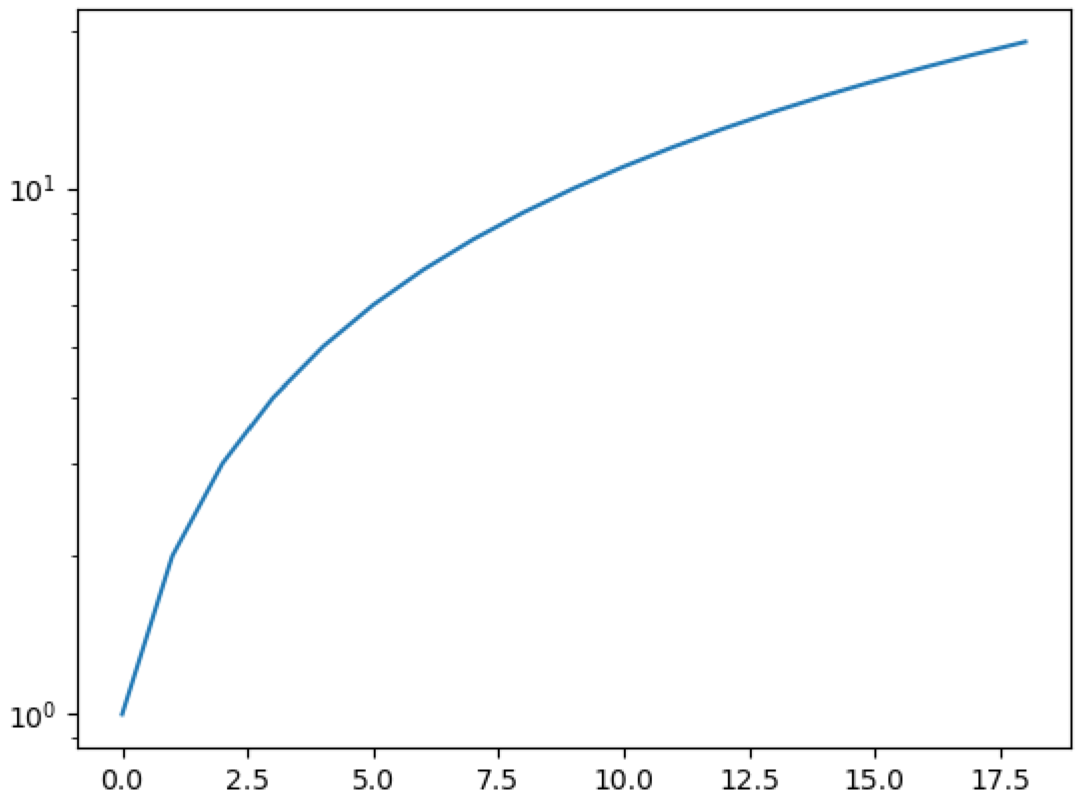
```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_title('Set the title like this')
ax.set_xlabel('x_label')
ax.set_ylabel('y_label')

names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

plt.bar(names, values)
plt.show()
```



If you need to plot data on a nonlinear-scale you can change the scale of your axis by using the `xscale()`and `yscale()` functions and providing these functions with the type of scale you want (ex. `'log'`). Matplotlib supports[3] the linear scale, log scale, symmetrical log scale, and logit scales:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

plt.plot(values)
plt.yscale('log')
plt.show()
```

[3]https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.yscale.html

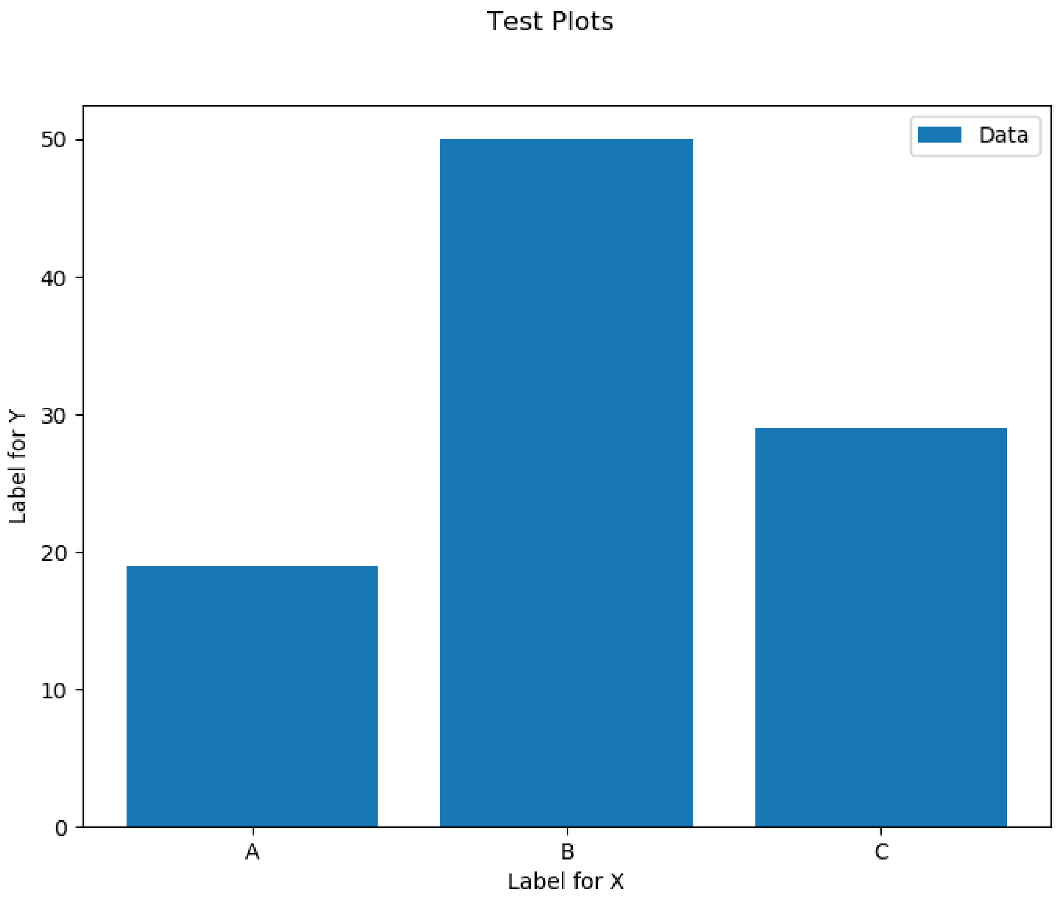


As you can see now, the scales of the x and y axes aren't the same. The values are linear, but the graph isn't.

You can adjust the position of the legend by using the `loc` or `location` argument on the axis/figure object. You then pass in the desired location of the legend to these functions. Below, we'll tell it to place the argument in the "upper right". We can also pass in the name of the values we want to represent as a list. There's only one type of data in this graph, so we only specify one element.

There's an alternate way of positioning the legend, you can also use the `legend` function on an axes object and pass in your desired position/coordinates to the `bbox_to_anchor` argument. Here, we'll use the `legend()` function:

```python
names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

fig = plt.figure(figsize=(8.0,6.0))

plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.bar(names, values)
plt.legend(['Data'], loc="upper right")
plt.suptitle('Test Plots')
plt.show()
```

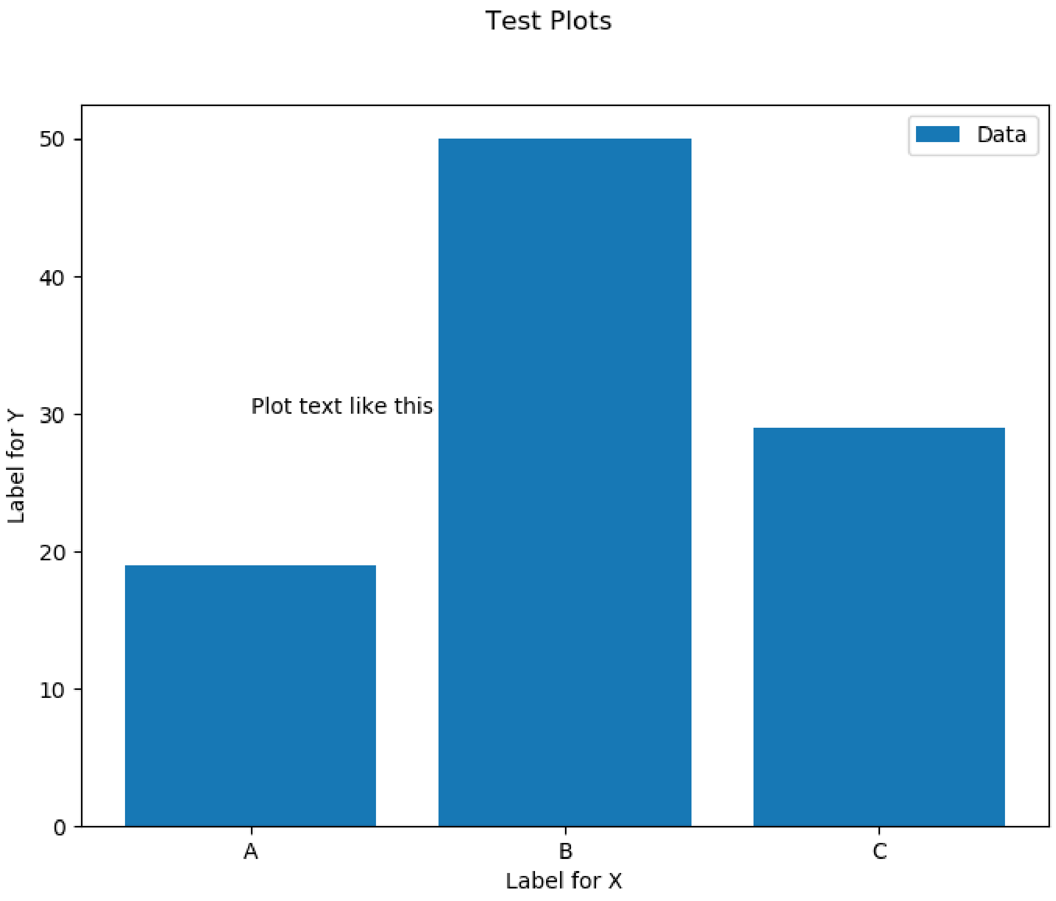


The legend can be found in the upper-right hand corner of the plot, with the value of *"Data"*.

Text

You can also write text on the plot itself through the use of the `text()` function, which will write directly on the axes object. This works with the standard Pyplot plotting scheme:

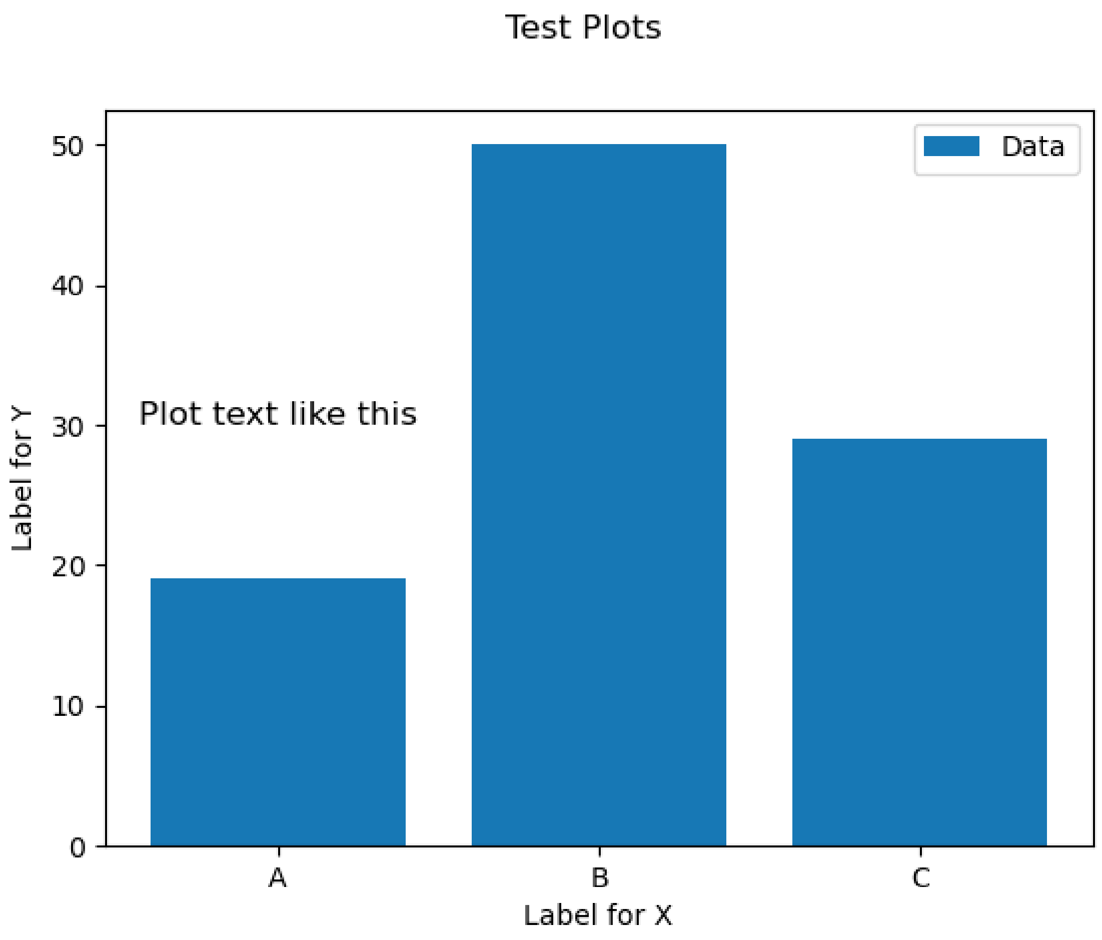
```python
plt.text(0, 30, r'Plot text like this')
plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.bar(names, values)
plt.legend(['Data'], loc="upper right")
plt.suptitle('Test Plots')
plt.show()
```



We've set the "value" of the text to be `0` and `30`. The center of our `A` category is the `0` value for the X-axis. The text starts there, on the height of `30` on the Y-axis.

We can also override the default properties and set the text *center* to be level with our `0` value and change the font size:

```python
plt.text(0, 30, r'Plot text like this', fontsize=12, horizontalalignment='center')
plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.bar(names, values)
plt.legend(['Data'], loc="upper right")
plt.suptitle('Test Plots')
plt.show()
```

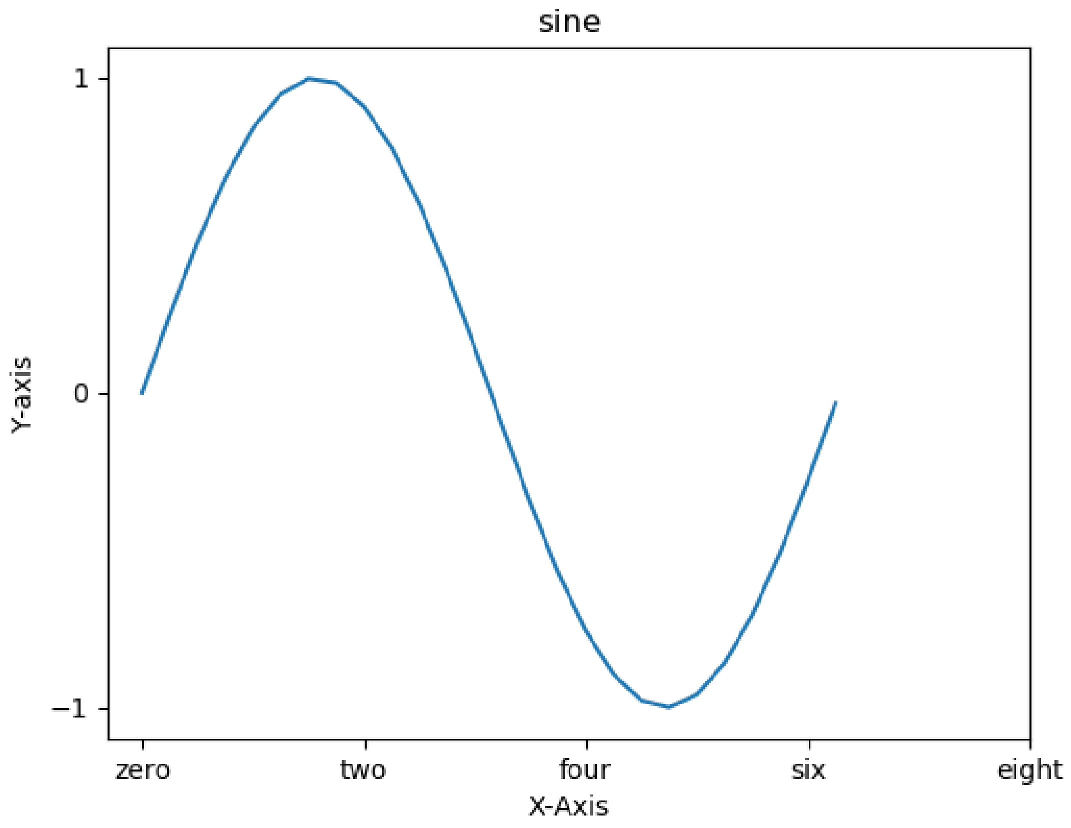


Ticks

You can edit the position and labels of your plot's ticks. With the `set_xticks()` and `set_yticks()` functions, you can pass in lists of the positions where you want the ticks to be displayed.

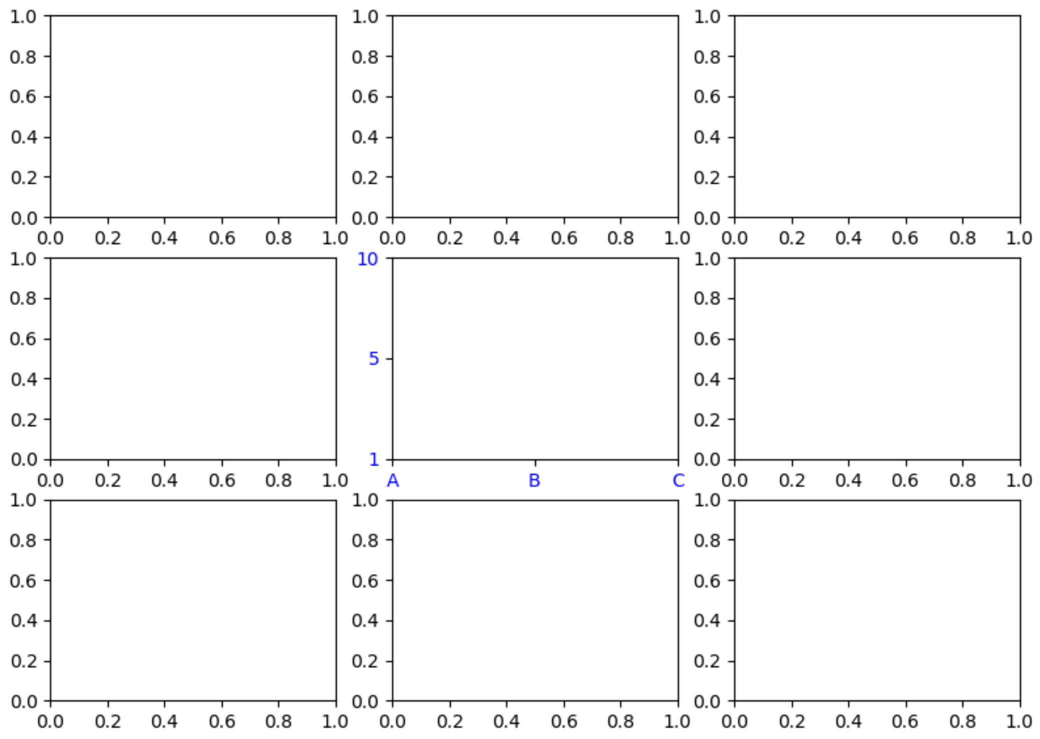
Afterwards, you can use `set_xlabels()` and `set_ylabels()` to label the ticks as you want. There are a variety of other options for customizing ticks as well:

```python
import matplotlib.pyplot as plt
import numpy as np
import math

range = np.arange(0, math.pi*2, 0.25)
sin = np.sin(range)
fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.75, 0.75])
ax.plot(range, sin)
ax.set_xlabel('X-Axis')
ax.set_ylabel('Y-axis')
ax.set_title('sine')
ax.set_xticks([0,2,4,6,8])
ax.set_xticklabels(['zero','two','four','six','eight'])
ax.set_yticks([-1,0,1])

plt.show()
```



If you want to customize just a single subplot, you can do this by using the `sca()` method. It takes in a list of axes objects and you then specify which subplot you want to transform using the row-column format. Below, we'll create a grid of 9 subplots and modify the middle grid by selecting it with the `sca()` method and then modifying the labels of the subplot using `xticks` and `yticks`:

```
fig, axes = plt.subplots(3, 3)

plt.sca(axes[0, 1])

plt.xticks([0, 1, 2], ["A", "B", "C"], color="blue")
plt.yticks([0, 1, 2], [1, 5, 10], color="blue")
plt.show()
```

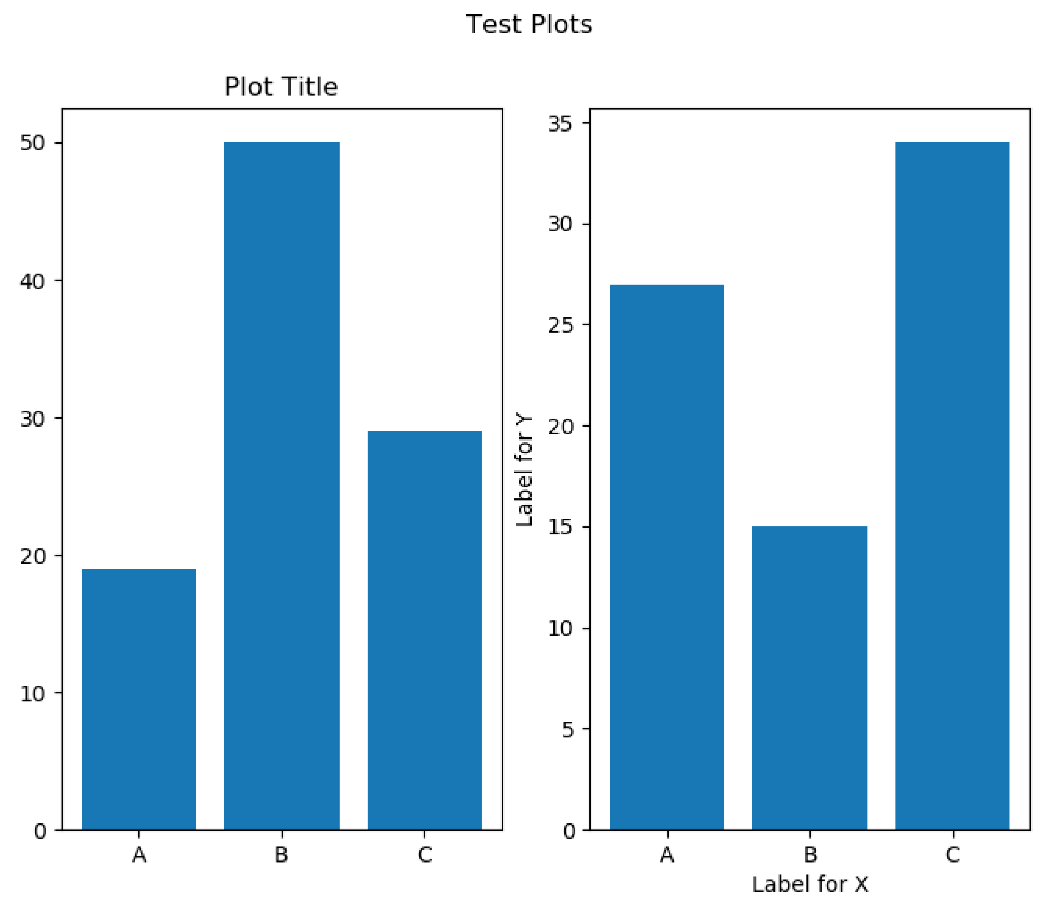


Layout

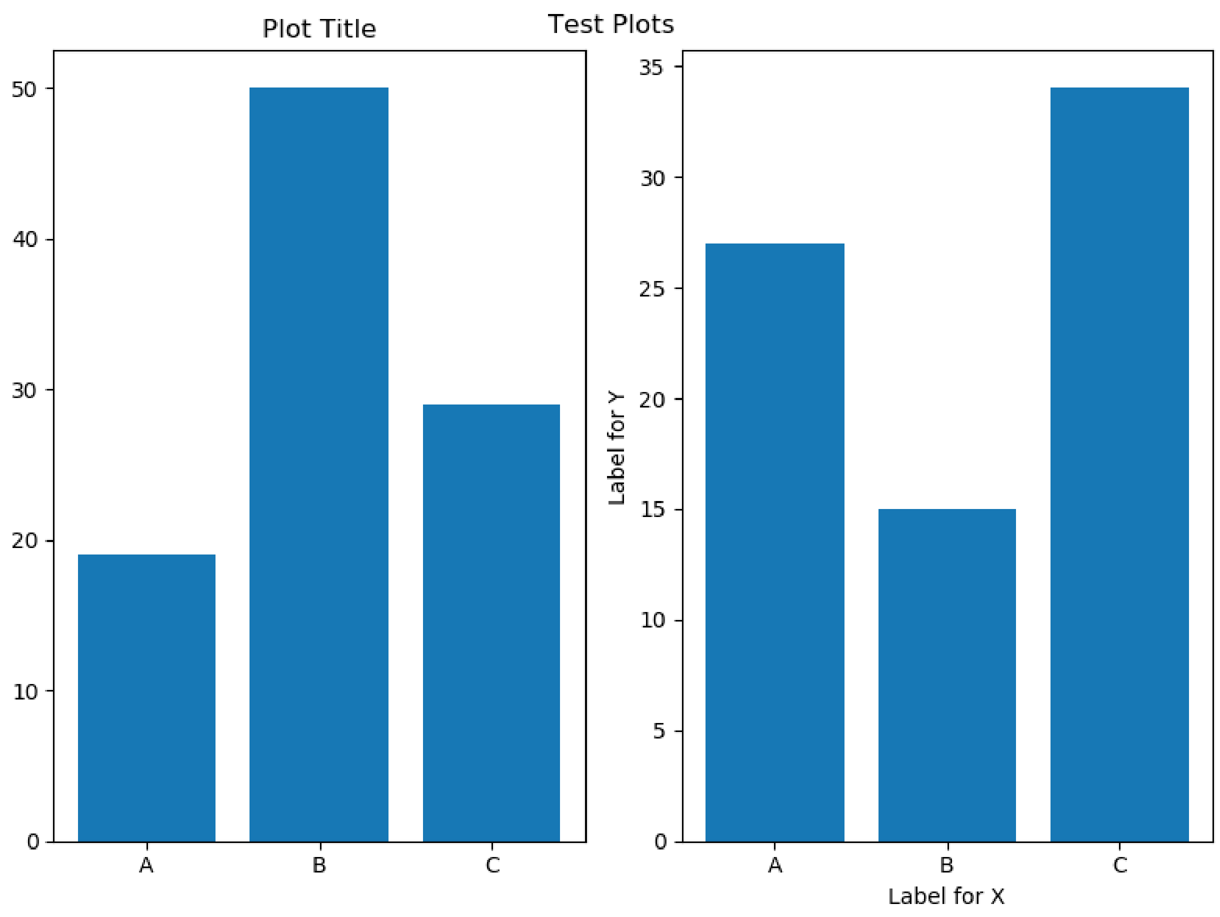
Something you may want to consider as you create your plots is that you may have regions of your plot where this is a lot of white space between elements.

If this is not something you want, it can be fixed via the `tight_layout()` function. The `tight_layout()` function automatically calculates positions for the elements of your plot, ensuring they fit tightly within the figure and look good. Here's what our figure looks like before adjusting the layout:

```
names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

fig = plt.figure(figsize=(8.0,6.0))

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax.set_title('Plot Title')
ax.bar(names, values)
ax2.bar(names, values_2)

plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.suptitle('Test Plots')

plt.show()
```

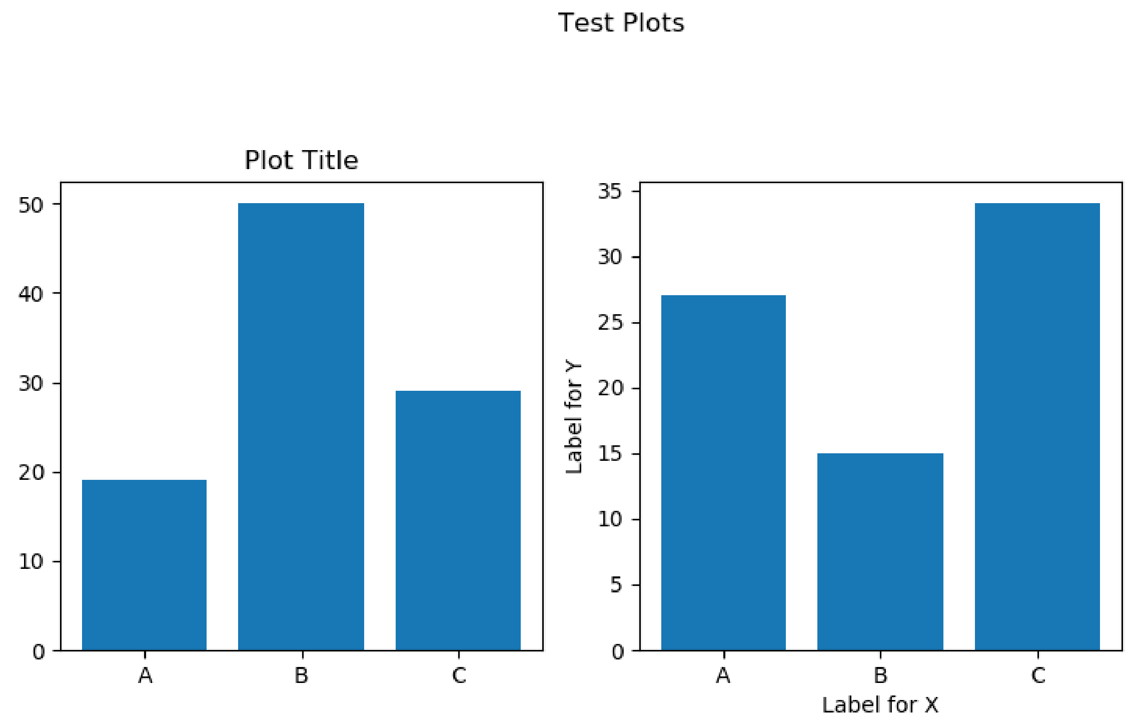


Now here's what happens when we use the `tight_layout()` function:



If you'd like more control over the layout of subplots you can use the `subplots_adjust()` function, which lets you set the parameters between subplots as you desire:

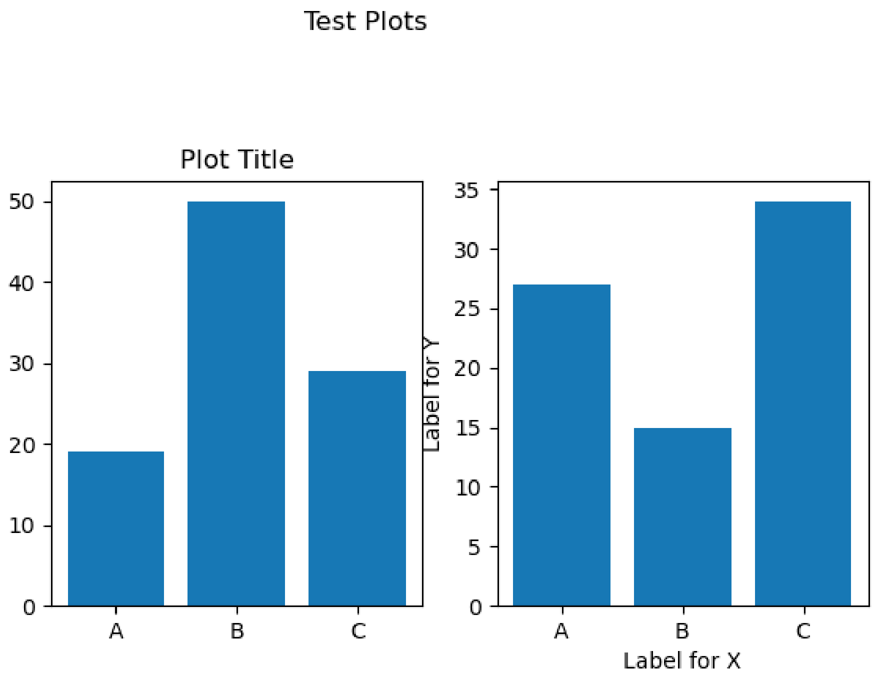
```python
names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

fig = plt.figure(figsize=(8.0,6.0))

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax.set_title('Plot Title')
ax.bar(names, values)
ax2.bar(names, values_2)

plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.suptitle('Test Plots')

plt.subplots_adjust(0.05, 0.3, 0.90, 0.8)

plt.show()
```



For example, if we adjusted this to:

```
plt.subplots_adjust(0.25, 0.35, 0.90, 0.8)
```

We'd be looking at a different layout:

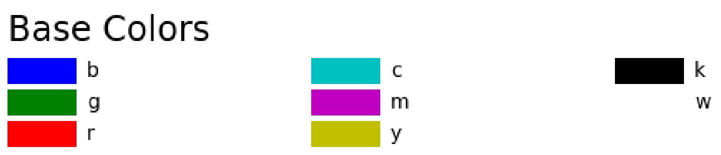


Although it might not be too obvious from the images, test these parameters out for yourself and you'll see a significant difference.

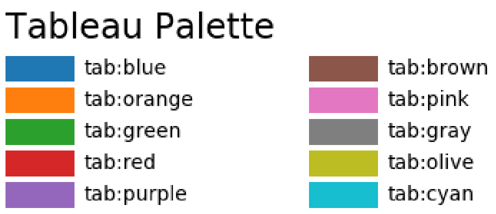
Changing Colors of Elements

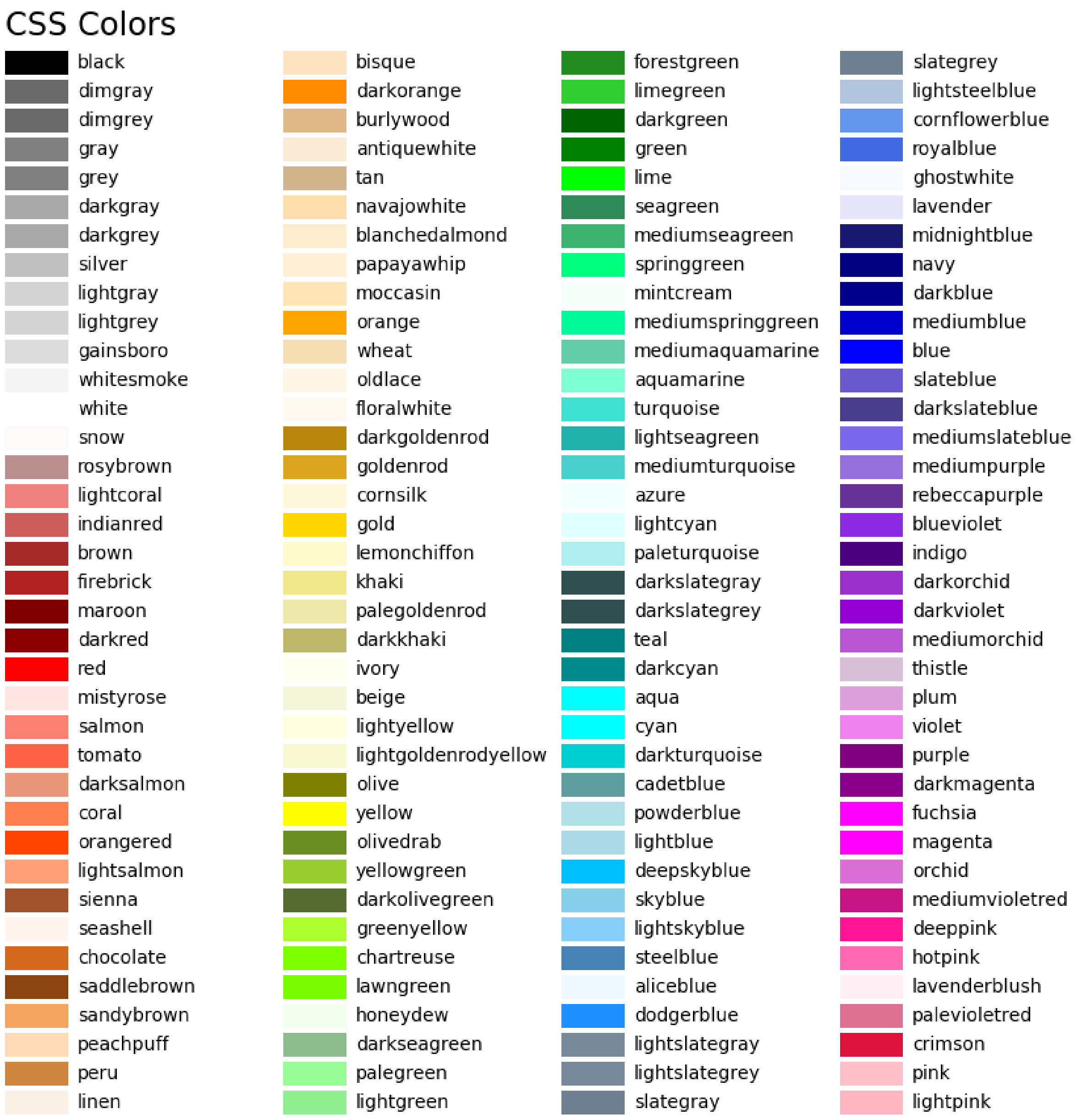
Matplotlib lets you change the colors of elements. For example, you can change the color of the wedges in a pie plot or recolor a line. These are typically changed for clarity and aesthetics.

As per Matplotlib's documentation[4], here are the colors it supports:



[4]https://matplotlib.org/3.1.0/gallery/color/named_colors.html





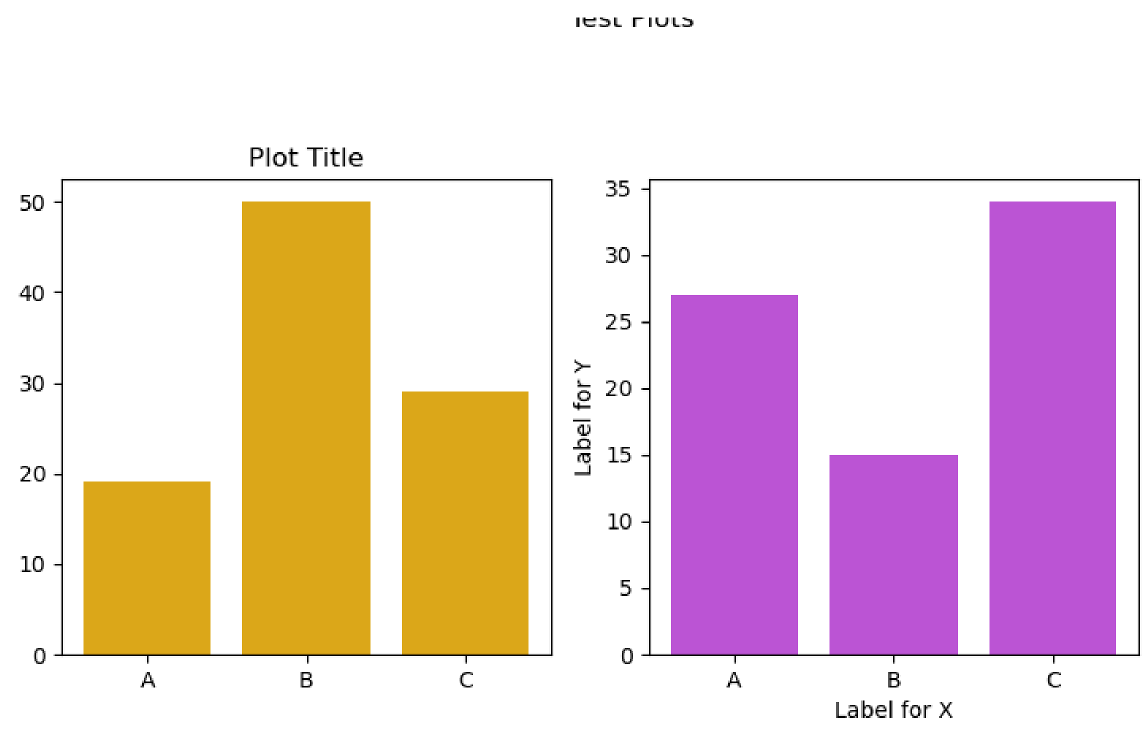
Credits: Matplotlib[5]

[5] https://matplotlib.org/3.1.0/gallery/color/named_colors.html

For example, let's change the color of the previous plot's elements:

```
import matplotlib.pyplot as plt
import pandas as pd

names = ['A', 'B', 'C']
values = [19, 50, 29]
values_2 = [27, 15, 34]

fig = plt.figure(figsize=(8.0,6.0))

ax = fig.add_subplot(121)
ax2 = fig.add_subplot(122)
ax.set_title('Plot Title')
ax.bar(names, values, color='goldenrod')
ax2.bar(names, values_2, color='mediumorchid')

plt.xlabel("Label for X")
plt.ylabel("Label for Y")
plt.suptitle('Test Plots')

plt.subplots_adjust(0.05, 0.3, 0.90, 0.8)

plt.show()
```

This would result in:



Note: You can also use CSS codes for these colors, such as `#000000`.

Visualization Examples

Now that we have covered the structure of Matplotlib plots and gone over how to customize your plots, we can take a look at some of the different types of plots you can produce with Matplotlib.

Matplotlib allows you to produce many different plot types, and we won't cover *every last plot type.*

Instead, we'll take a look at the plots you're most likely to use in the future, as well as the general approach to making them.

Pie Plot

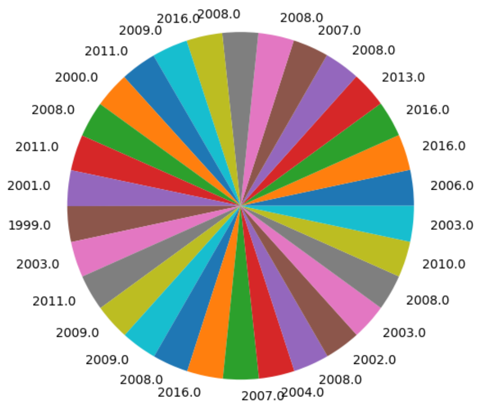
We've already covered how you can produce bar charts above, so we'll start by looking at a pie chart. We'll use the video game sales data we used in the chapter on Pandas to once again make a pie chart. You can generate a pie chart with the `pie()` function which accepts a 1-dimensional array of data.

The data is then plotted, in wedges, counterclockwise.

If the `labels` are provided, they're plotted with the wedges as well.

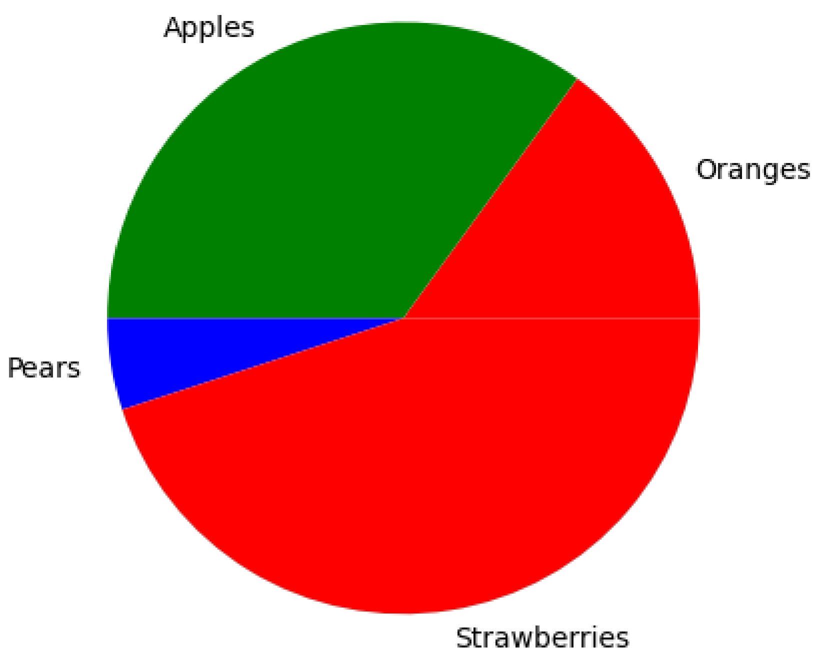
The process is largely the same. We use Pandas to select the column we want to visualize (year of release) and provide it as both the data and labels. We've also selected just the last 30 entries by using the tail function:

```
file = pd.read_csv("vgsales.csv")

plt.pie(file['Year'].tail(30), labels=file['Year'].tail(30))
plt.show()
```



You can also set an array of colors that the pie chart will cycle through:

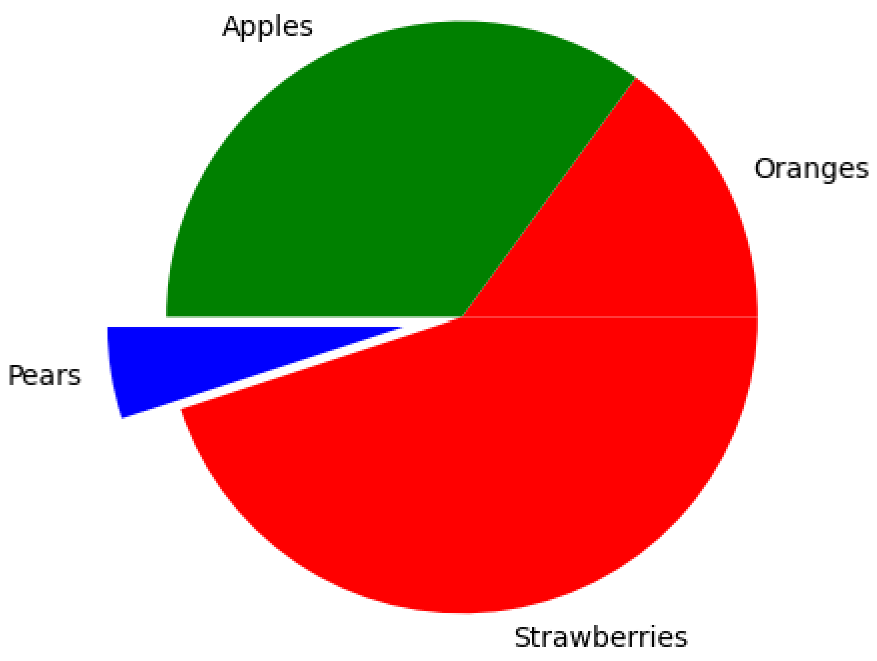
```
values = [15, 35, 5, 45]
labels = 'Oranges', 'Apples', 'Pears', 'Strawberries'
colors = {'r', 'g', 'b', 'r'}

plt.pie(values, labels=labels, colors=colors)
plt.show()
```



If you provide fewer colors than values/labels, it'll just cycle through them once it finishes.

Other arguments are also available, such as the `explode` argument:

```python
values = [15, 35, 5, 45]
labels = 'Oranges', 'Apples', 'Pears', 'Strawberries'
colors = {'r', 'g', 'b', 'r'}
explode = [0, 0, 0.2, 0]

plt.pie(values, labels=labels, colors=colors, explode=explode)
plt.show()
```



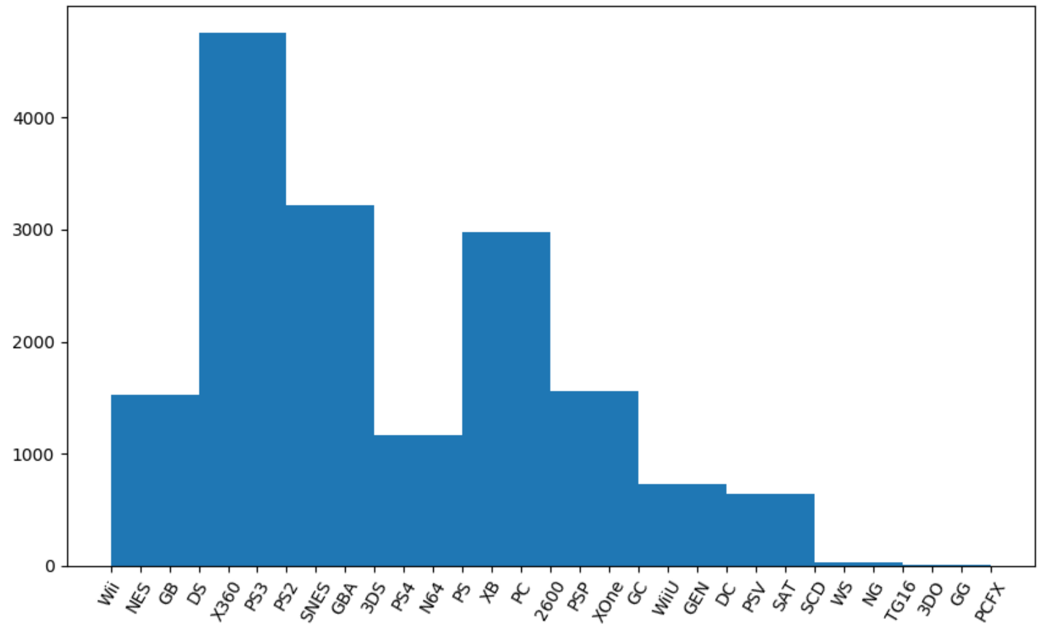
It’s an array of `explode` values. `0` leaves the wedge as is, and any value higher than that will offset it from the center. Here, we’ve *exploded* pears by `0.2` to highlight it.

Histogram

Here’s a histogram of the number of releases on different gaming platforms. We just use the `hist()` function from Matplotlib and PyPlot.

We’ve can also rotate the tick labels to make them slightly easier to read since there are so many of them:

```
file = pd.read_csv("vgsales.csv")
platform = file['Platform']

plt.hist(platform)
plt.xticks(rotation=60)
plt.show()
```

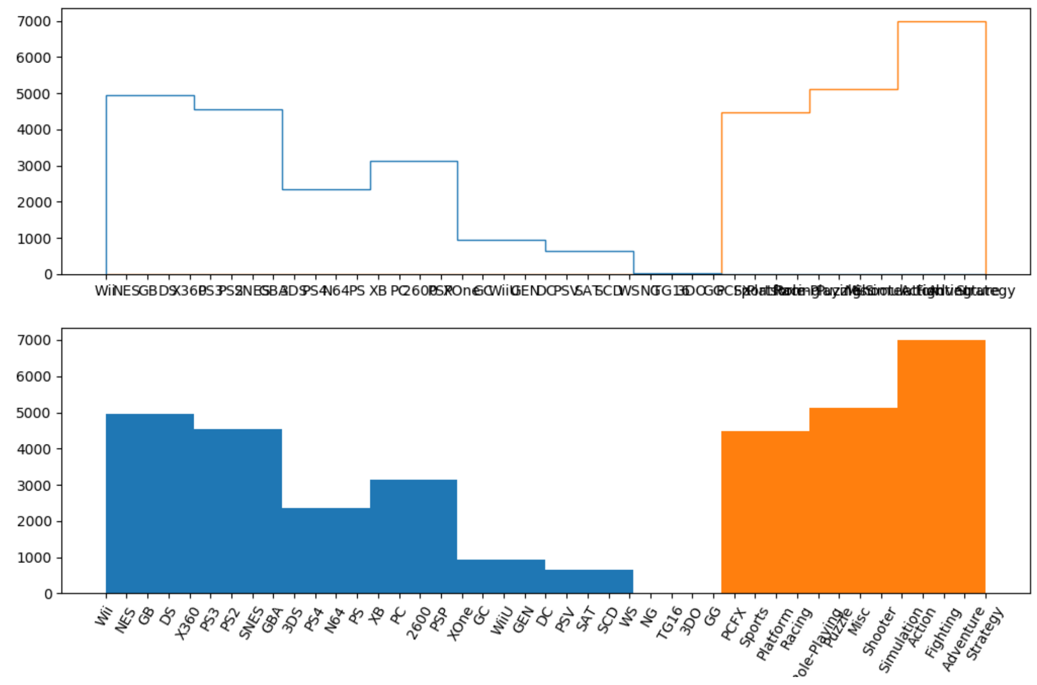


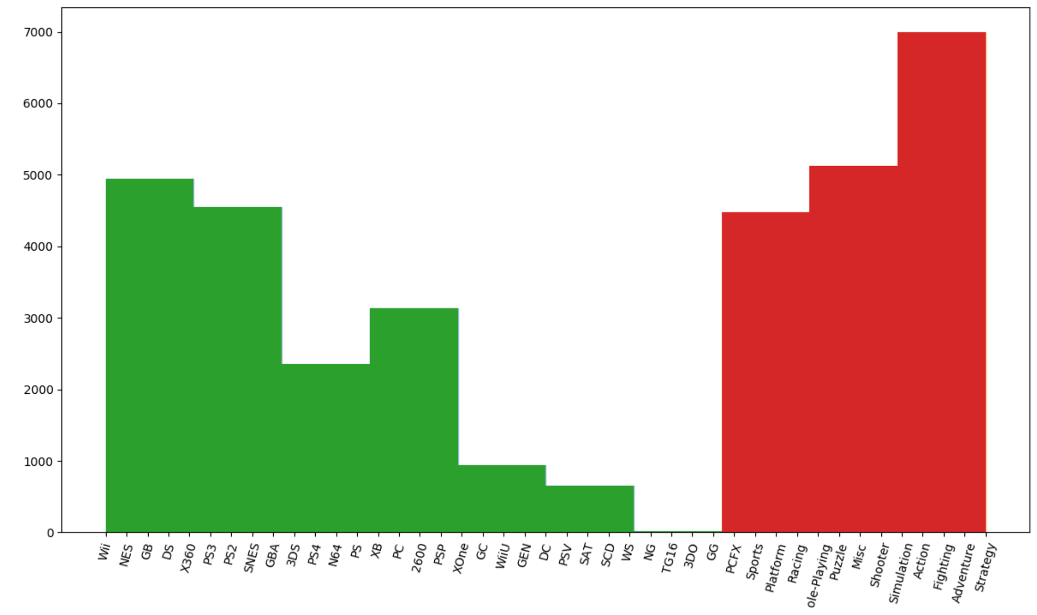
There are various types of histogram plots, and these can be specified with the `histtype` argument. The default is the *bar* histogram, though we can also plot different types such as:

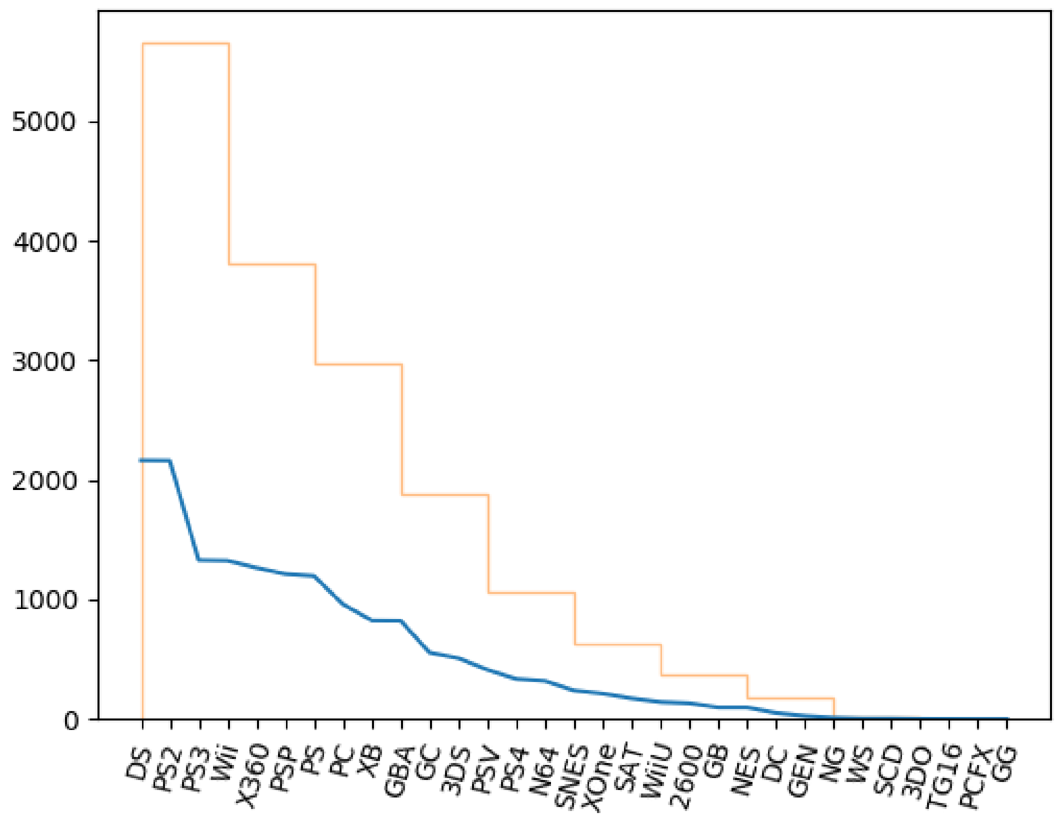
- `barstacked`
- `step`
- `stepfilled`

Though, these are only ever useful if you stack multiple datasets in a single plot. For example:

```python
file = pd.read_csv("vgsales.csv")

platform = file['Platform']
genres = file['Genre']

data = [platform, genres]

fig, ax = plt.subplots(2)
ax[0].hist(data, histtype='step')
ax[1].hist(data, histtype='stepfilled')
plt.xticks(rotation=60)
plt.show()

plt.hist(data, histtype='step', alpha=0.5)
plt.hist(data, histtype='stepfilled')
plt.xticks(rotation=75)
plt.show()

fig_2, ax = plt.subplots()
ps3 = file['Platform'].value_counts(dropna=False)

# Counts = list(file['Platform'].values())
plt.plot(ps3)
plt.hist(platform, histtype='step', alpha=0.5)
plt.xticks(rotation=75)
plt.show()
```




0
1000
2000
3000
4000
5000
6000
7000
Wii
NES
GB
DS
X360
PS3
PS2
SNES
GBA
3DS
PS4
N64
PS
XB
PC
2600
PSP
XOne
GC
WiiU
GEN
DC
PSV
SAT
SCD
WS
NG
TG16
3DO
GG
PCFX
Sports
Platform
Racing
ole-Playing
Puzzle
Misc
Shooter
Simulation
Action
Fighting
Adventure
Strategy



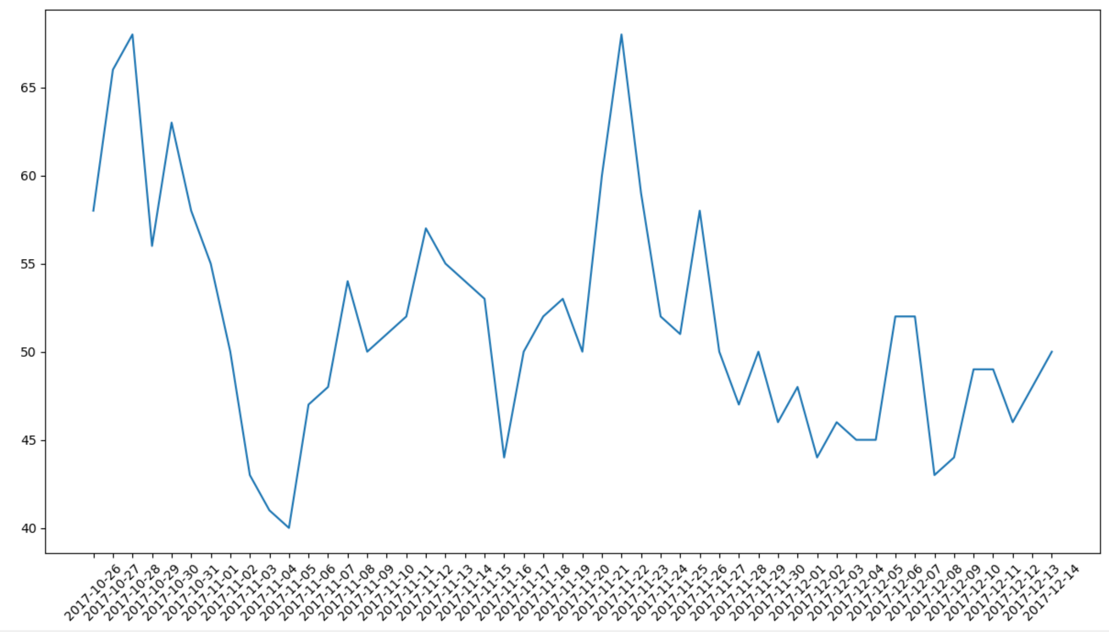
Line Plot

Line plots are commonly used to track changes over time. Here, we are using data on rain amounts for the city of Seattle[6]. We've used Pandas to select the last 50 dates in the dataset and visualize the maximum temperature for that day (in Fahrenheit).

Notice that you can use the generic plot function to accomplish this, you don't need to use a line special plotting function:

```
file = pd.read_csv("seattleWeather.csv")
plt.plot(file['DATE'].tail(50), file['TMAX'].tail(50))
plt.xticks(rotation=45)

plt.show()
```

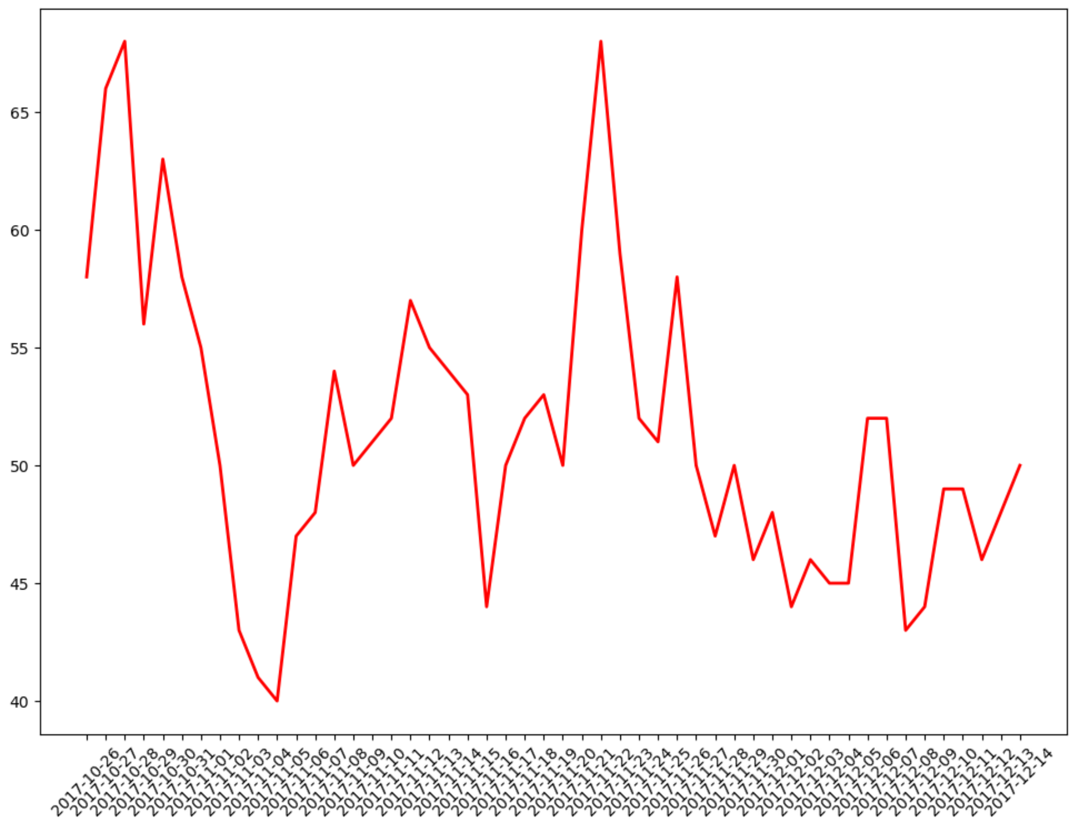
[6]https://www.kaggle.com/rtatman/did-it-rain-in-seattle-19482017



Of course, you can customize line plots by changing line properties. For example, you can change it's color or width:

```
file = pd.read_csv("seattleWeather.csv")
plt.plot(file['DATE'].tail(50), file['TMAX'].tail(50), color='r', linewitdh=2.0)
plt.xticks(rotation=45)

plt.show()
```

This results in:

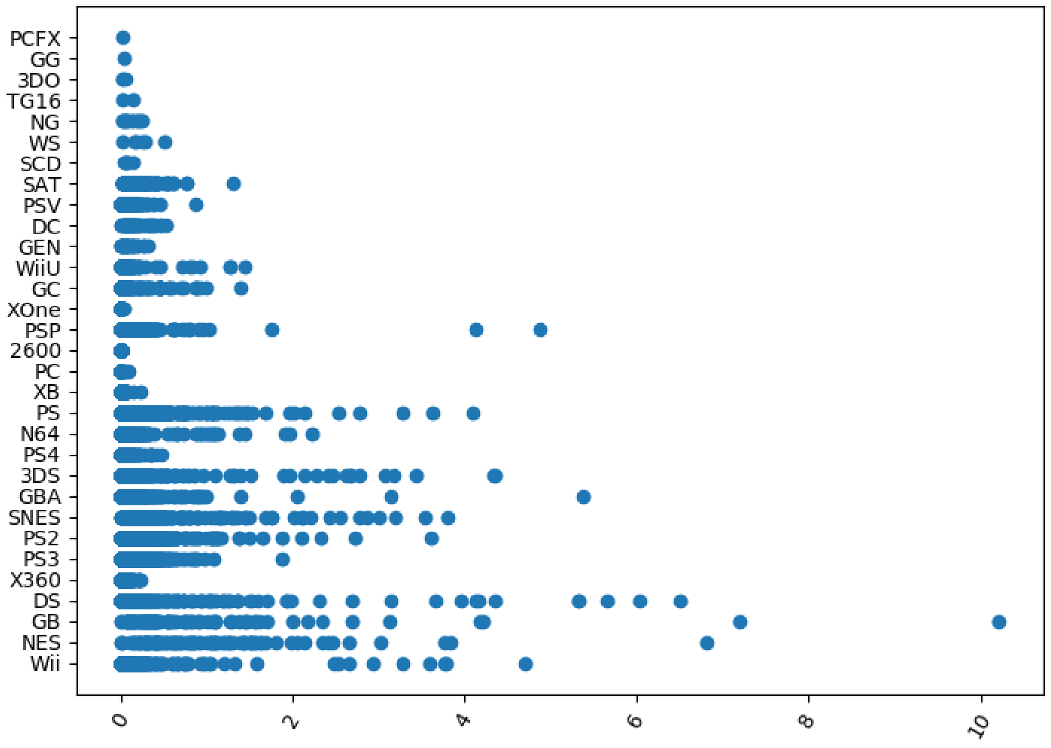


Scatter Plot

You can create scatter plots with the `scatter()` function. Scatter plots are creates using two continuous variables, which can help make relationships between the two variables apparent. You can also create a scatter plot using just one continuous variable and a categorical variable.

Below, we visualize sales in Japan by platform, and the dots show instances of games and the numbers they have sold in millions (by platform):

```
file = pd.read_csv("vgsales.csv")

plt.scatter(file['JP_Sales'], file['Platform'])

plt.xticks(rotation=60)
plt.show()
```



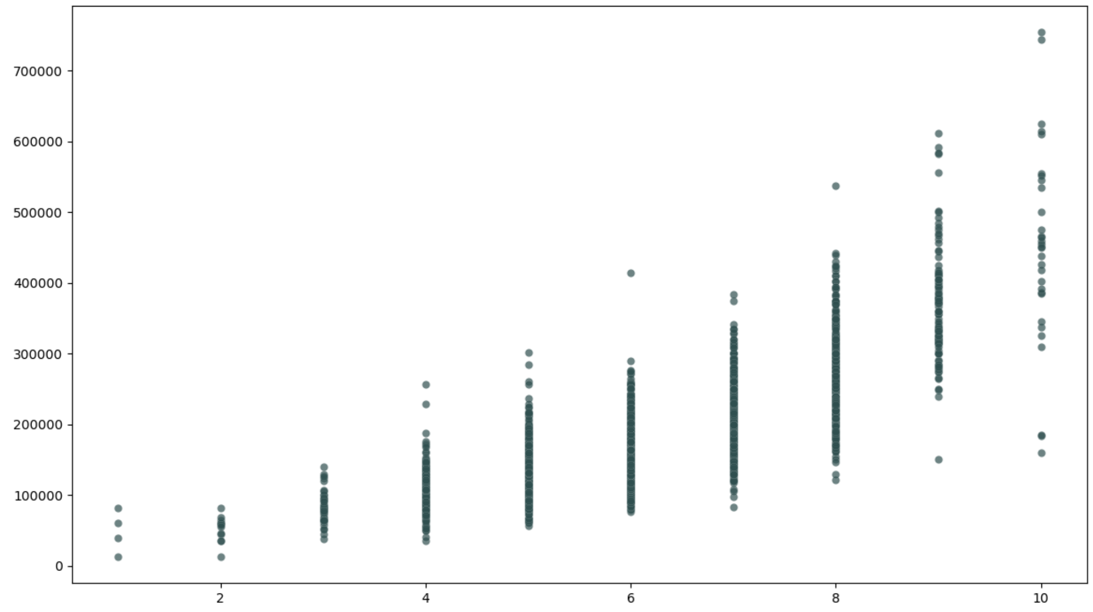
Here, we're using a continuous and a categorical variable.

Or, we could use the *Ames Housing*[7] dataset that keeps track of sold properties in an Iowa State city - Ames, let's load it in and compare the `SalePrice` variable with the `Overall Qual` variable. The `Overall Qual` variable refers to the quality of the materials used to build the property:

```
file = pd.read_csv('AmesHousing.csv')

fig, ax = plt.subplots(figsize=(14,8))
ax.scatter(x=file['Overall Qual'], y=file['SalePrice'], color="darkslategrey",
           edgecolors="white", linewidths=0.1, alpha=0.7);
plt.show()
```

This will result in:

[7] https://www.kaggle.com/prevek18/ames-housing-dataset

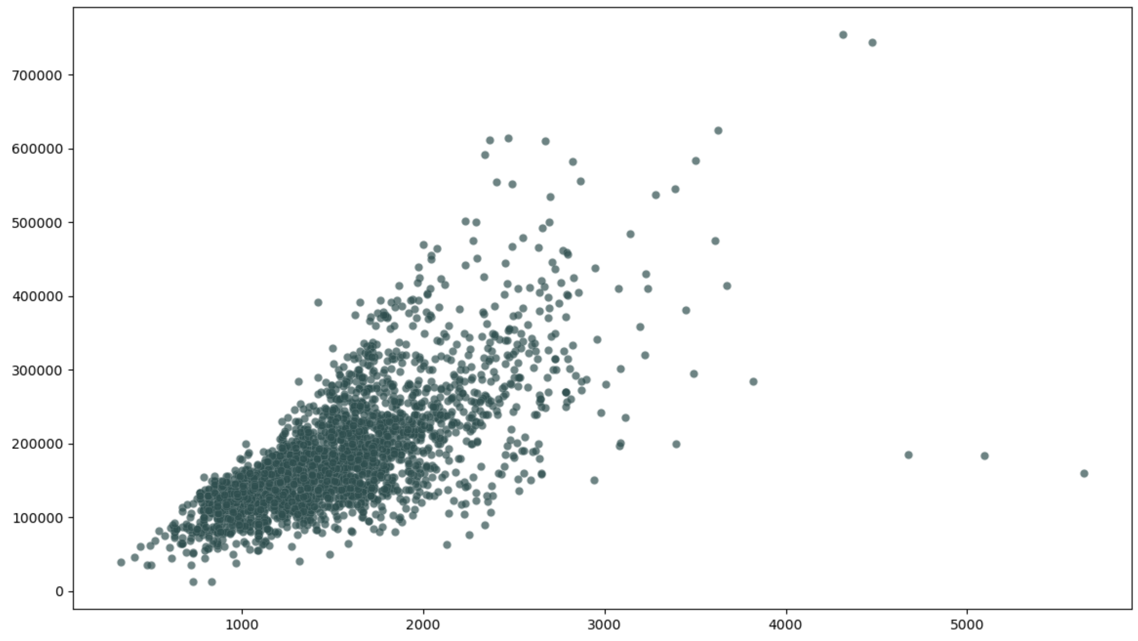


Here, it's really easy to see a high positive correlation between a property's price and the overall quality of the materials used to build it. This is also a relationship between a categorical and a continuous variable.

Now, let's compare the ground area above street-level with the sale price:

```
df = pd.read_csv('AmesHousing.csv')

fig, ax = plt.subplots(figsize=(14,8))
ax.scatter(x=df['Gr Liv Area'], y=df['SalePrice'], color="darkslategrey",
           edgecolors="white", linewidths=0.1, alpha=0.7);
plt.show()
```

This results in:



Here, we can still see a significant correlation between the two variables. The larger the living area above ground-level, the higher the price is, in most cases. This is a relationship between two continuous variables.

Scatter Plots are a great way to visualize correlations and relationships between variables such as these.

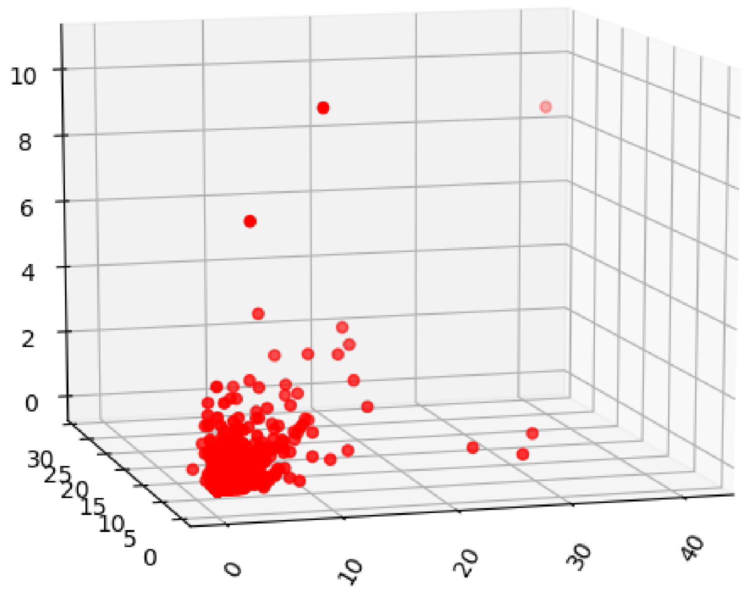
3D Plots

Matplotlib also supports the creation of 3D plots. In order to create 3D plots using Matplotlib, you need to import the `Axes3D` module from `mpl_tookits.mplot3d`.

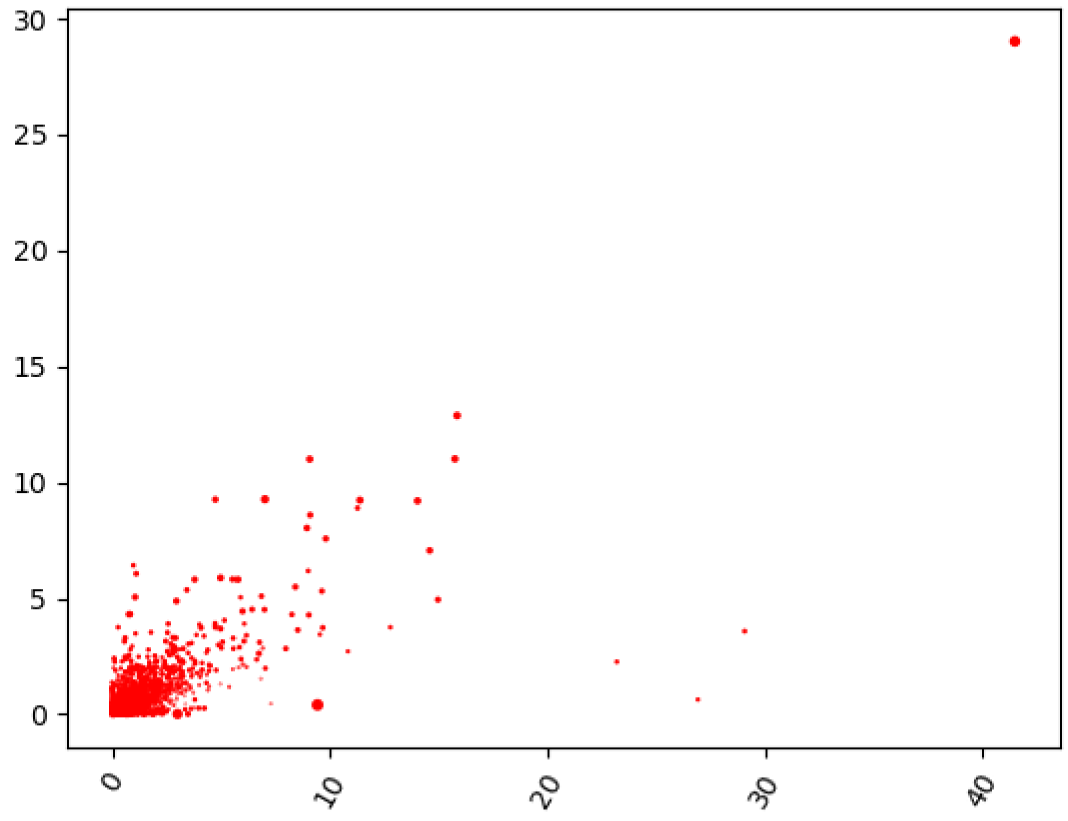
You can create 3D versions of common plot types, which allows you to include more variables and rotate the plot around to get better views. To add a 3D subplot to a `Figure`, we'll set the `projection=3d` in the `add_subplot()` function.

We can then pass in our features. Here's a scatter plot, though this time, between three different categories in the `vgsales.csv` dataset:

```python
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

file = pd.read_csv("vgsales.csv")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = file['NA_Sales']
y = file['EU_Sales']
z = file['Other_Sales']

ax.scatter(x, y, z, c='r')
plt.xticks(rotation=60)
plt.show()
```



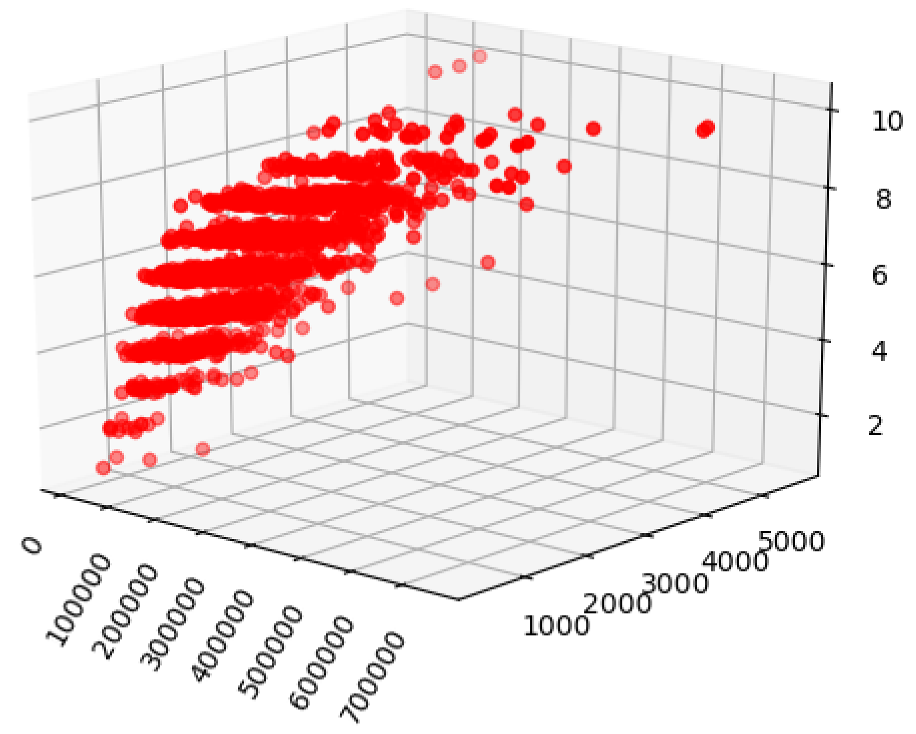
If we were to have this exact same code, but projected in 2D, it'd look like:



With a 3D projection, we can explore more dimensions of the data we're analyzing. For example, let's go back to `AmesHousing.csv` and plot a scatter plot for `SalePrice`, `Gr Liv Area` and `Overall Qual` together:

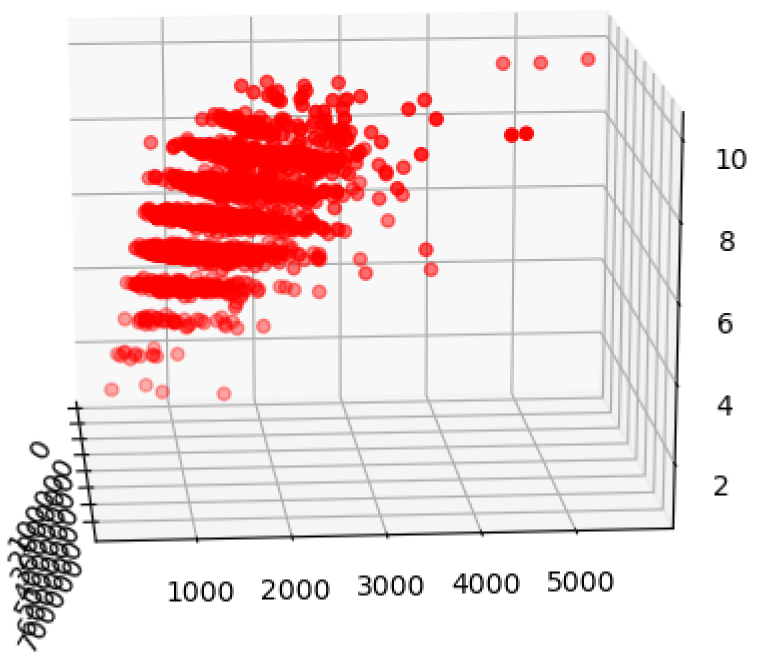
```
import pandas as pd
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

file = pd.read_csv('AmesHousing.csv')

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = file['SalePrice']
y = file['Gr Liv Area']
z = file['Overall Qual']

ax.scatter(x, y, z, c='r')
plt.xticks(rotation=60)
plt.show()
```

This will result in:



Here, we can clearly see the price going up, both with the overall quality of the materials used, but also the living area above ground-level.

If we rotate the 3D plot a bit, the effect of `Gr Liv Area` is more obvious:



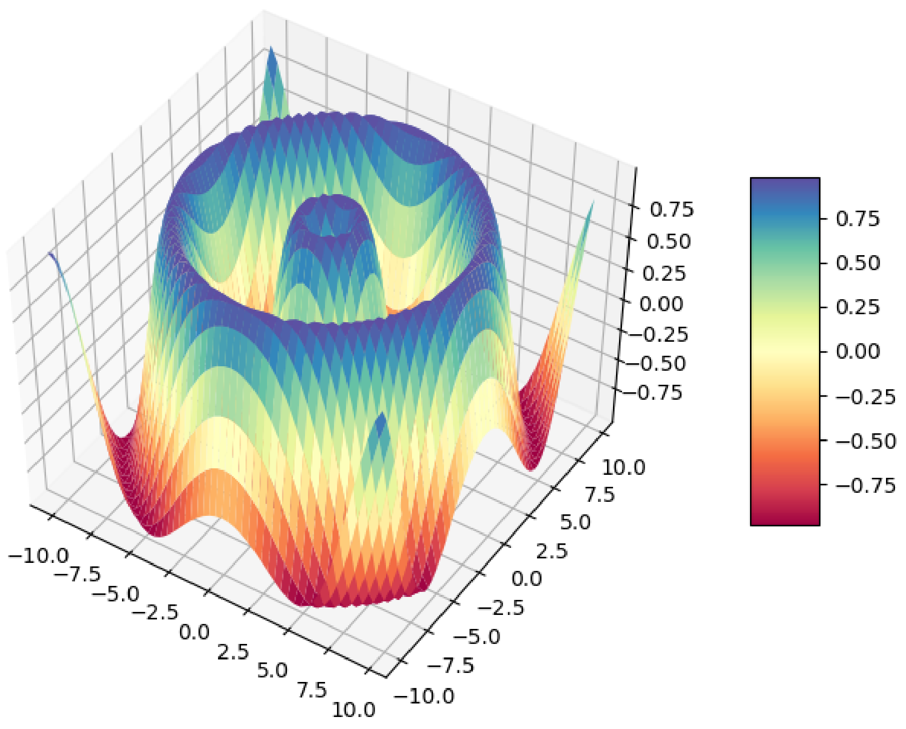
You can also create plots that aren't just 3D versions of standard plots. For instance, surface plots are used to demonstrate relationships between two independent variables and one dependent variable.

This helps the topology of surfaces and relationships be discerned. In order to create a surface plot, you'll need to import the `cm` module to choose a color-fill scheme for the plot. The `cm` module is a baseclass for all scalar to RGBA mappings.

Here, we'll use NumPy to create values for X, Y, and Z and then pass the variables into the `plot_-surface()` function:

```python
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

fig = plt.figure()

# The .gca() function is "Get Current Axes".
# It returns the current axes matching the **kwargs
# If the axes doesn't exist (it doesn't here), it'll be created automatically
# Here, we're packing the returned value into an `ax` object
# You can create 3D plots like this if you aren't using subplots
```

```
ax = fig.gca(projection='3d')

X = np.arange(-10, 10, 0.40)
Y = np.arange(-10, 10, 0.40)
X, Y = np.meshgrid(X,Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)

# Creating the surface with Numpy variables
# And specifying the Colormap to be Spectral
surf = ax.plot_surface(X, Y, Z, cmap=cm.Spectral)

# Adding a colorbar to the right of the plot
# For a user-friendly bar of values
fig.colorbar(surf, shrink=0.5,aspect=5)
plt.show()
```



As you have probably realized by now, Matplotlib is a powerful visualization library that is both fairly intuitive and highly customizable. It remains the standard plotting library in Python for a reason.

However, Matplotlib could afford to be more user-friendly when it comes to generating plots. For this reason, we’ll be taking a look at Seaborn next. Seaborn was designed to build upon Matplotlib and allow users to make impressive visualizations with just a few lines of code.

Seaborn also supports more plot types out of the box than Matplotlib does.

Preview

Thank you for taking the time to take a peek at our book. This is the end of the preview. To get the remaining 9 chapters and ~220 pages, you can purchase the book at https://gum.co/data-visualization-in-python[8].

[8]https://gum.co/data-visualization-in-python