# Agenda

**After this session, you will know**

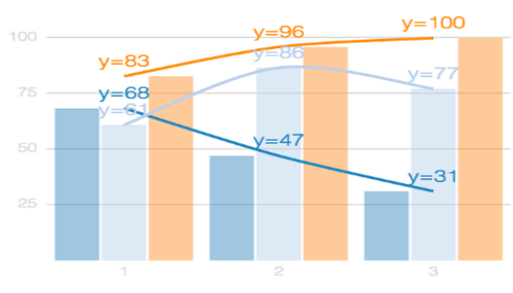
- Introduction to Data Visualization
- Introduction to Matplotlib
- Introduction to Seaborn

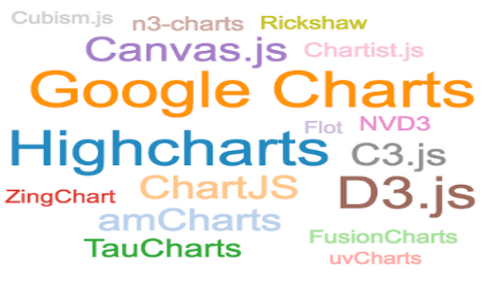
# Introduction to Data Visualization

# Introduction to Data Visualization

## Why build visuals?

1. For Exploratory Data Analysis
2. To communicate data clearly
3. Share unbiased representation of data
4. Use them to support recommendations to different stakeholders



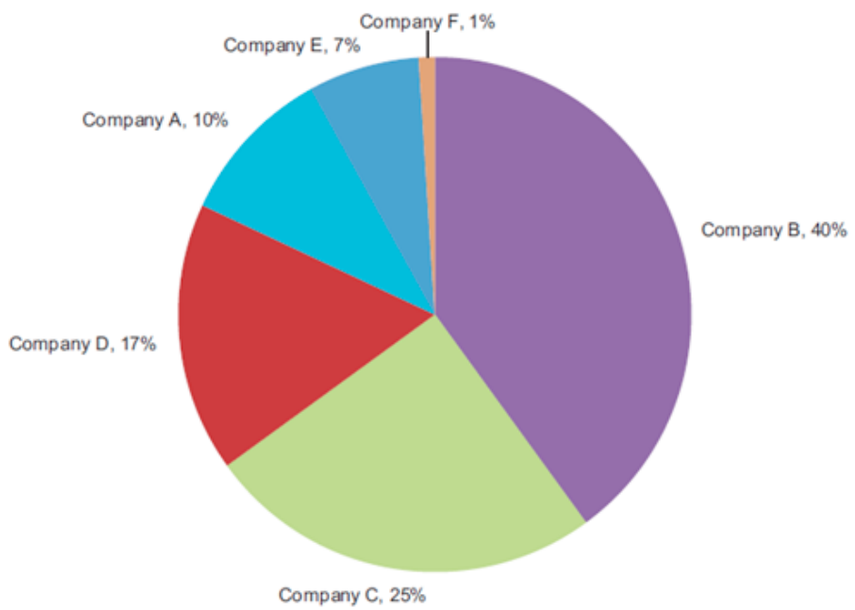

*"A picture is worth a thousand words."*
*- Fred R Barnard*

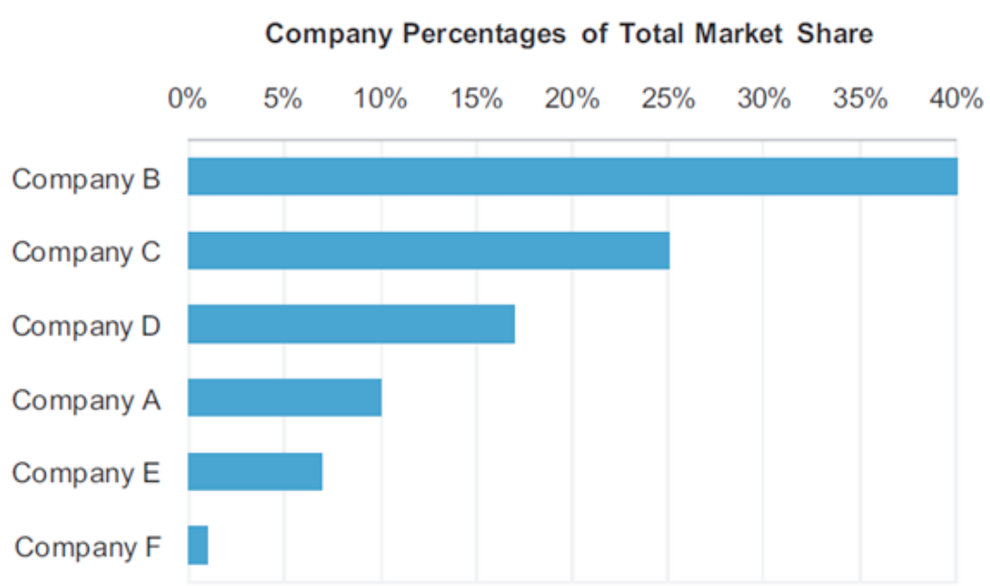
# Introduction to Data Visualization

## When creating a visual, always remember:

1. More need not be effective
2. More need not be attractive
3. More need not be impactive

For example:



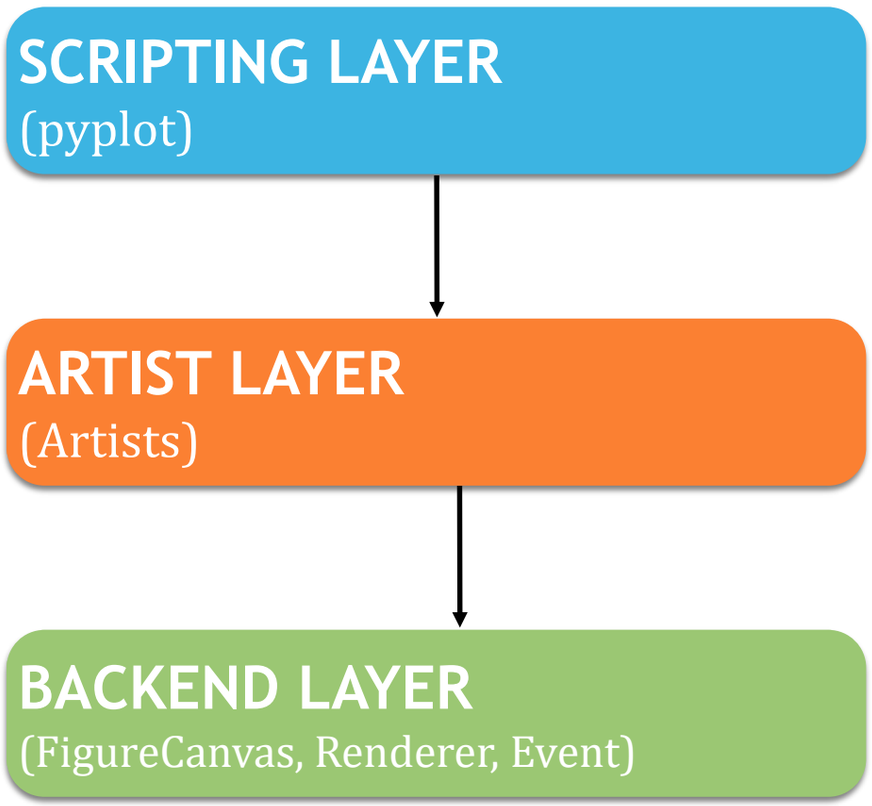
# Introduction to Matplotlib

# Introduction to Matplotlib

- Matplotlib is probably the single most used Python package for 2D-graphics
- It provides both a very quick way to visualize data from Python and publication-quality figures in many formats
- The library is huge, at something like 70,000 total lines of code
- Matplotlib is home to several different interfaces (ways of constructing a figure) and capable of interacting with a handful of different back-ends (Backends deal with the process of how charts are actually rendered, not just structured internally)

# Matplotlib Architecture

## Three distinct Layers:




## 1. Scripting Layer:

Comprised mainly of **pyplot,** a scripting interface that is lighter than the Artist Layer

# Matplotlib Architecture

## 2. Artist Layer:

- Comprised of one main object - **Artist**
  - Knows how to use the Renderer to draw on the canvas
- Title, lines, tick labels and images, all correspond to individual Artist instances
- Two types of Artist objects:
  1. **Primitive**: Line2D, Rectangle, Circle and Text
  2. **Composite:** Axis, Tick, Axes and Figure
- Each composite artist may contain other composite artists as well as primitive artists

# Matplotlib Architecture

## 3. Backend Layer:

Has three built-in abstract interface classes:

1. FigureCanvas: **matplotlib:backend_bases.FigureCanvas**
    - Encompasses the area onto which the figure is drawn

2. Renderer: **matplotlib.backend_bases.Renderer**
    - Knows how to draw on the FigureCanvas

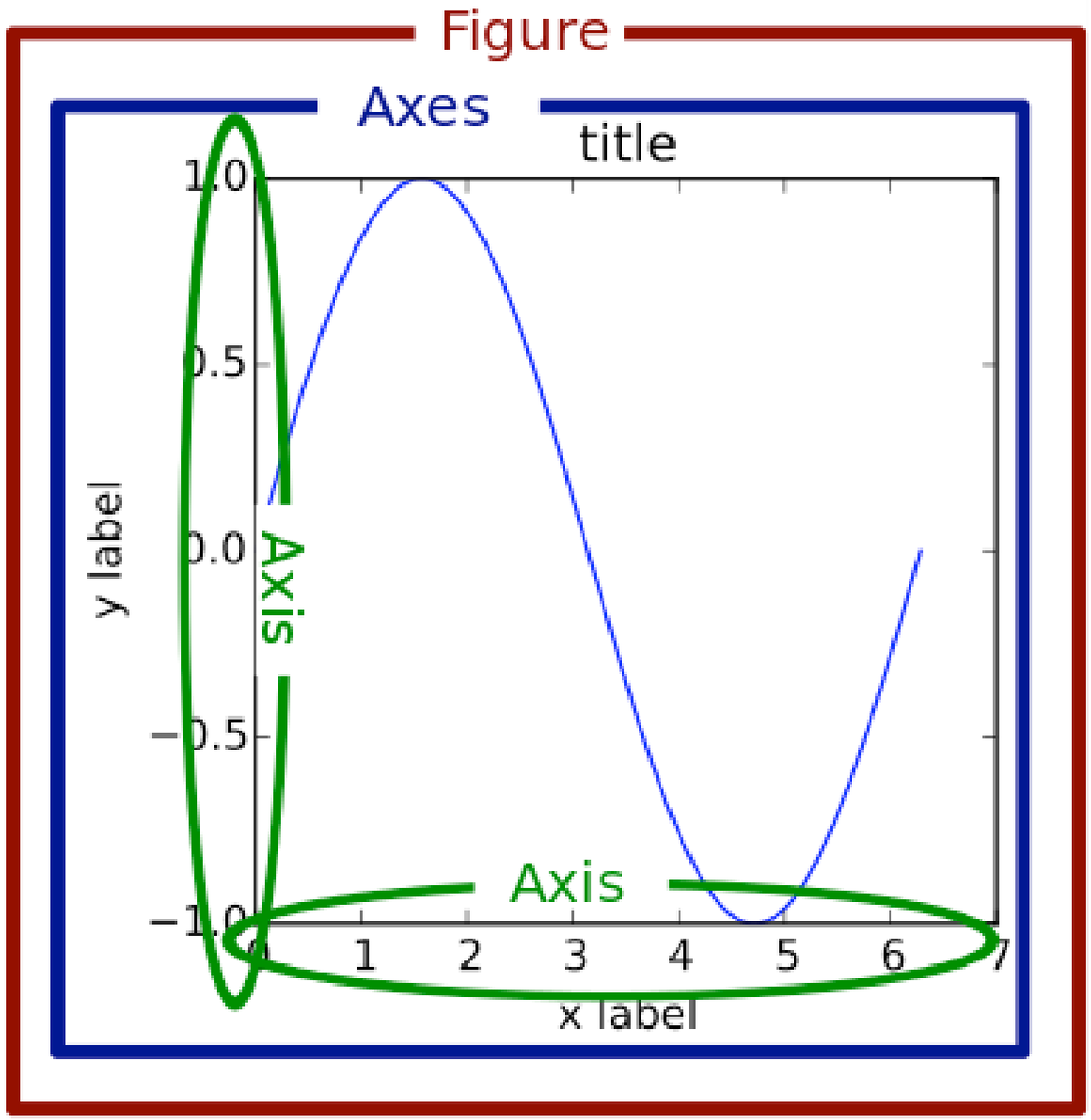
3. Event: **matplotlib.backend_bases.Event**
    - Handles user inputs such as keyboard strokes and mouse clicks

# The Matplotlib Object Hierarchy

- A “hierarchy” here means that there is a tree-like structure of matplotlib objects underlying each plot

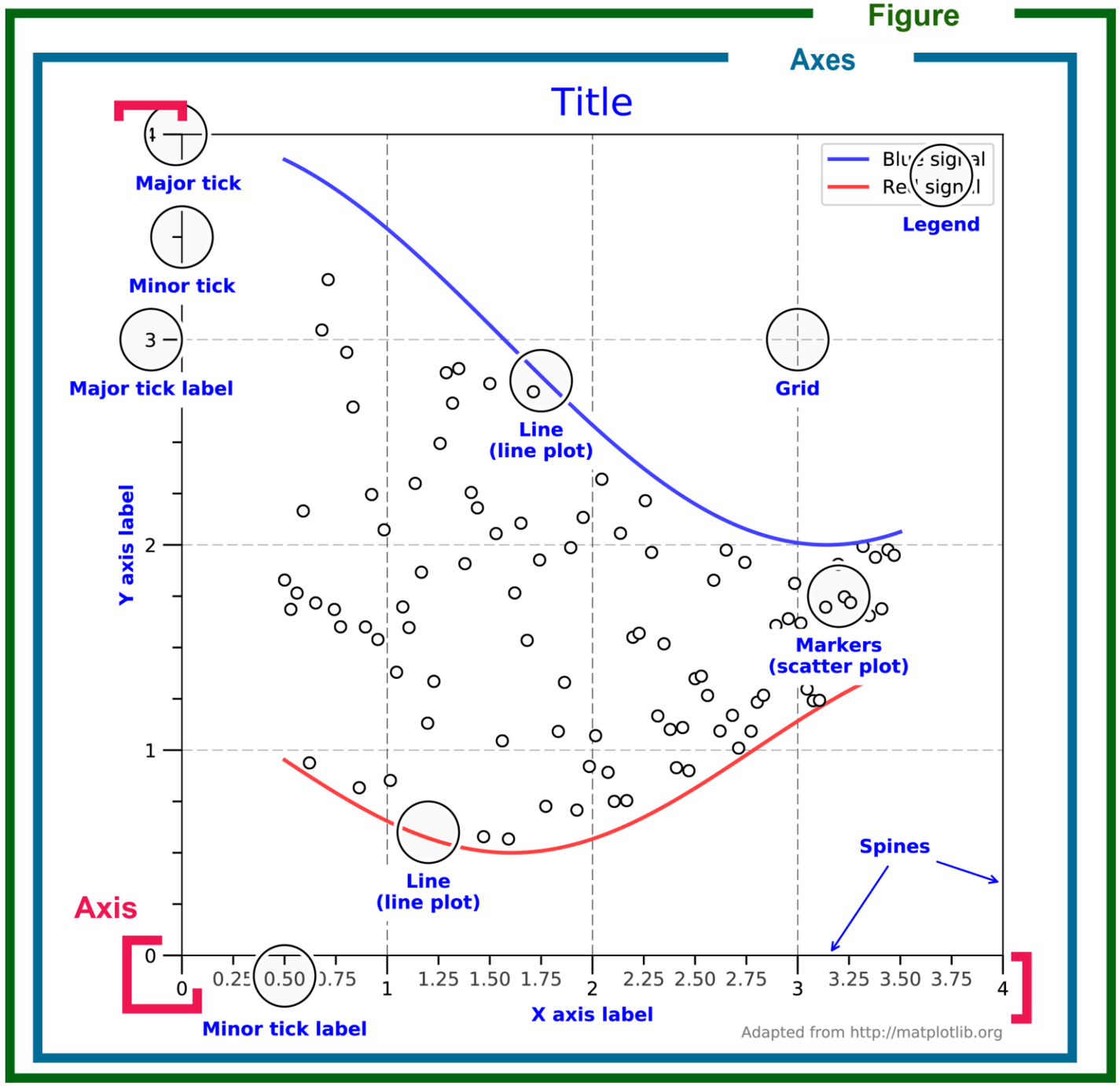
- A Figure object is the outermost container for a matplotlib graphic, which can contain multiple Axes objects

- Below the Axes in the hierarchy are smaller objects such as tick marks, individual lines, legends, and text boxes



# Anatomy of a Figure

- Matplotlib presents this as a figure anatomy, rather than an explicit hierarchy:



# Customizing your objects

**Figure**: A figure is the whole plotting area that contains all plot elements. Multiple subplots may be tiled in grid within one figure.

**Subplot**: A subplot is a subregion in a figure that contains all of the relevant data to be displayed on the same axes.

**Axis**: An axis measures the value of a point at a certain position. Most plots contain two axes, x (horizontal) and y (vertical). Sometimes, multiple axes, each containing one data series, can be overlaid in the same plotting area of a 2D plot.

**Axes**: In Matplotlib, the keyword in plural, axes, refers to the combination of axes in a plot. This can be intuitively understood as the data plotting area, as shown in the preceding figure.

**Spine**: Spines are the four lines that denote the boundaries of the data area.

**Grid**: Grids are lines inside the data area that aid the reading of values.

# Customizing your objects

**Title**: A name of the figure that describes the figure clearly and succinctly.

**Axis labels**: A word description of each axis. Units should be given if applicable.

**Ticks**: Ticks are marks of division on a plot axis. We can add major ticks and minor ticks to a figure.

**Tick labels:** Both major and minor ticks can be labeled. Besides printing the input data values directly, there are also formatters specific for date, logarithmic scale, and so on. We can also use our own defined functions to format the labels.

**Legend**: A legend has labels of each data series. It usually appears as a box that matches the styles of a data series with the corresponding names.

**Patches**: Different shapes can be added by matplotlib.patches, including rectangles, circles, ellipses, rings, sectors, arrows, wedges, and polygons.

**Alpha:** The alpha is a property in Artists layer. It is a scalar and it can take value range of 0 to 1.

# matplotlib Installation and Usage

- Using conda package manager of anaconda

  *conda install matplotlib*

- Once installed just import into python

```
## Package Plan ##

  environment location: /Users/ anaconda3

  added / updated specs:
    - matplotlib


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    conda-4.5.5                |            py36_0         1.0 MB

The following packages will be UPDATED:

    conda: 4.5.4-py36_0 --> 4.5.5-py36_0

Proceed ([y]/n)?
```

# Importing the pyplot

- To create a pandas DataFrame from objects such as lists and ndarrays, you may call:

```
import pandas as pd
```

To start creating Matplotlib plots, we first import the plotting API pyplot by entering this command:

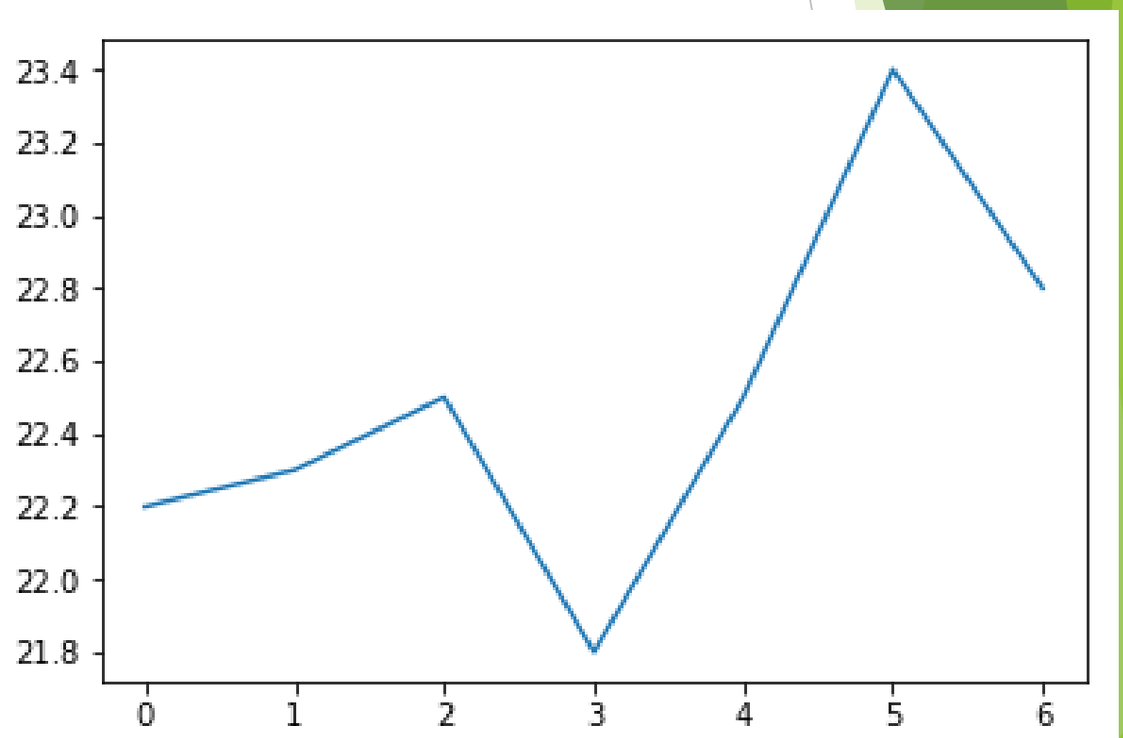
```
import matplotlib.pyplot as plt
```

# Our first plot with Matplotlib

- After importing matplotlib.pyplot as plt, we draw line plots with the plt.plot() command.

- Here is a code snippet for a simple example of plotting the temperature of the week:

```
# Import the Matplotlib module
Import matplotlib.pyplot as plt

# Use a list to store the daily temperature
t = [22.2,22.3,22.5,21.8,22.5,23.4,22.8]

# Plot the daily temperature t as a line plot
plt.plot(t)

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

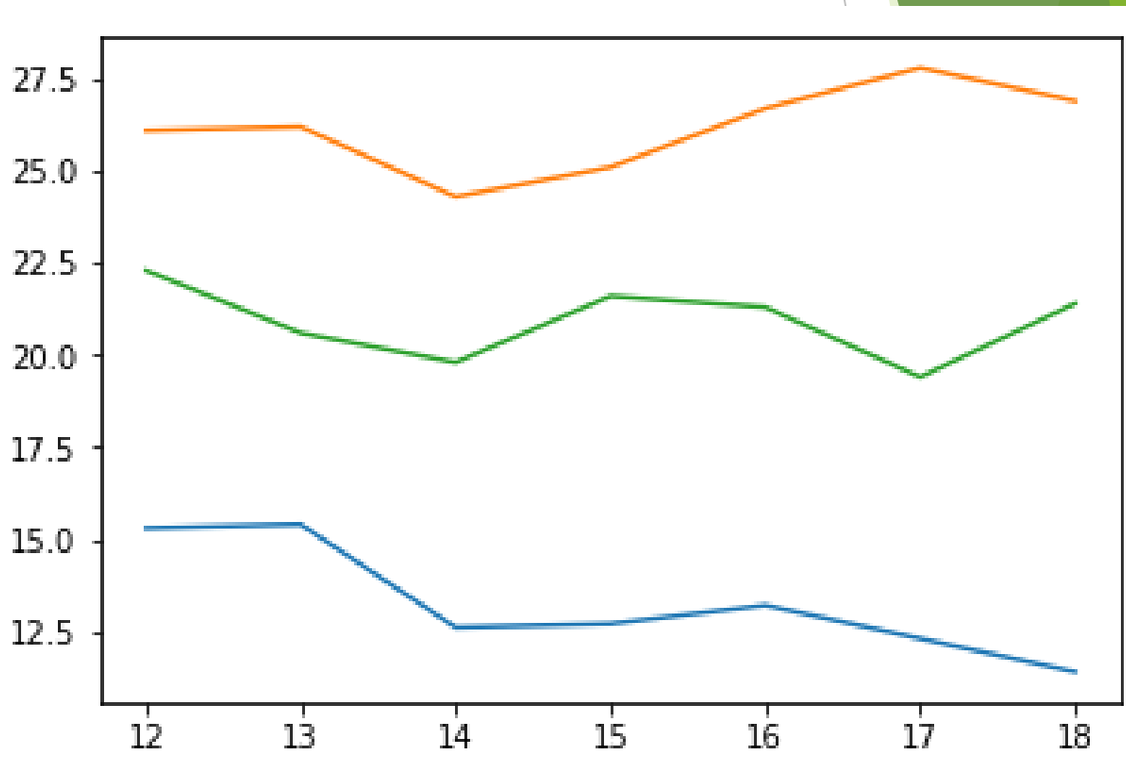



Remember to conclude each plot with plt.show().

# Multiline Plots

To create a multiline plot, we can draw a line plot for each data series before concluding the figure. Let's try plotting the temperatures of three different cities with the following code:

```
import matplotlib.pyplot as plt

# Prepare the data series
date = [12,13,14,15,16,17,18]
Tokyo = [15.3,15.4,12.6,12.7,13.2,12.3,11.4]
Hawaii = [26.1,26.2,24.3,25.1,26.7,27.8,26.9]
Hong_Kong = [22.3,20.6,19.8,21.6,21.3,19.4,21.4]

# Plot the lines for each data series
plt.plot(date,Tokyo)
plt.plot(date,Hawaii)
plt.plot(date,Hong_Kong)

plt.show()
```



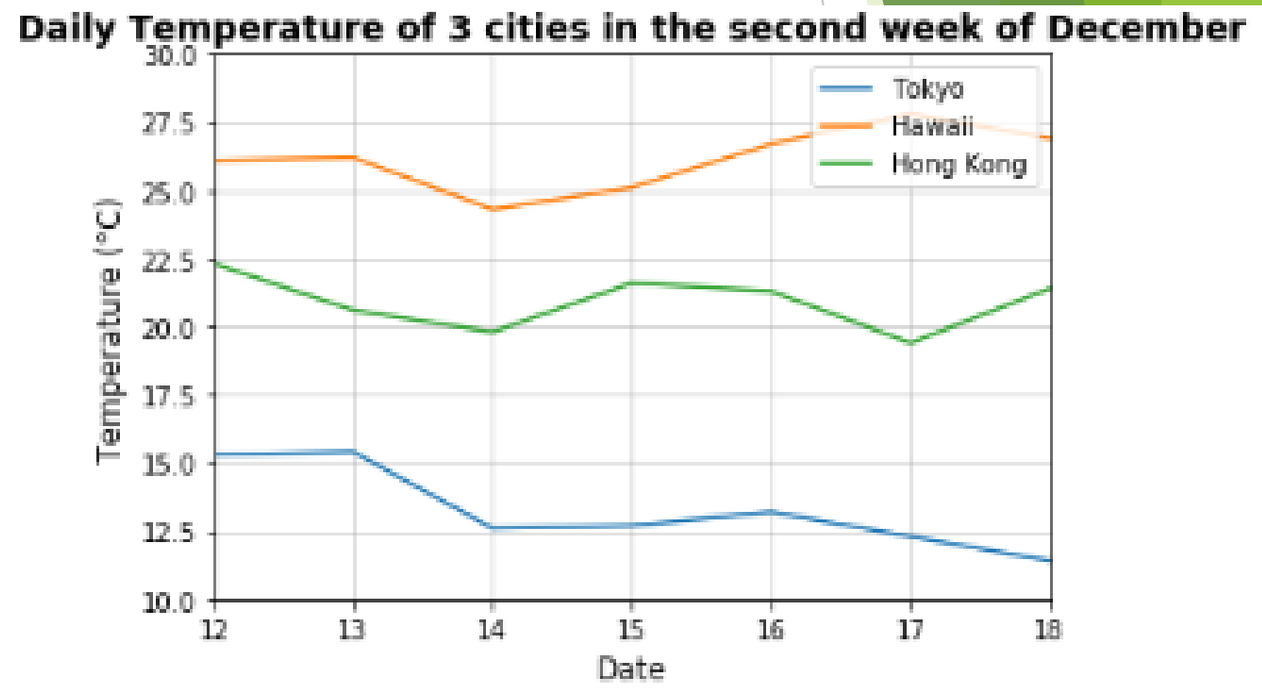

This example is adapted from the maximum temperatures of three cities in a week in December 2018.

# Adjusting Axes, Labels, Titles and Legends in Multiline Plots

- To give meaning to the values on the x and y axes, we need information about the nature and type of data, and its corresponding units. We can provide this information by placing axis labels with **plt.xlabel()** or **plt.ylabel()**.
- We can turn on the background gridlines by calling **plt.grid(True**) before **plt.show()**
- To describe the information of the plotted data, we can give a title to our figure. This can be done simply with the command **plt.title(yourtitle)**
- To match data series on plots with their labels, such as by their line styles and marker styles, we use **plt.legend(loc= 'lower right')**

# Adjusting axes, labels, titles and legends in Multiline Plots

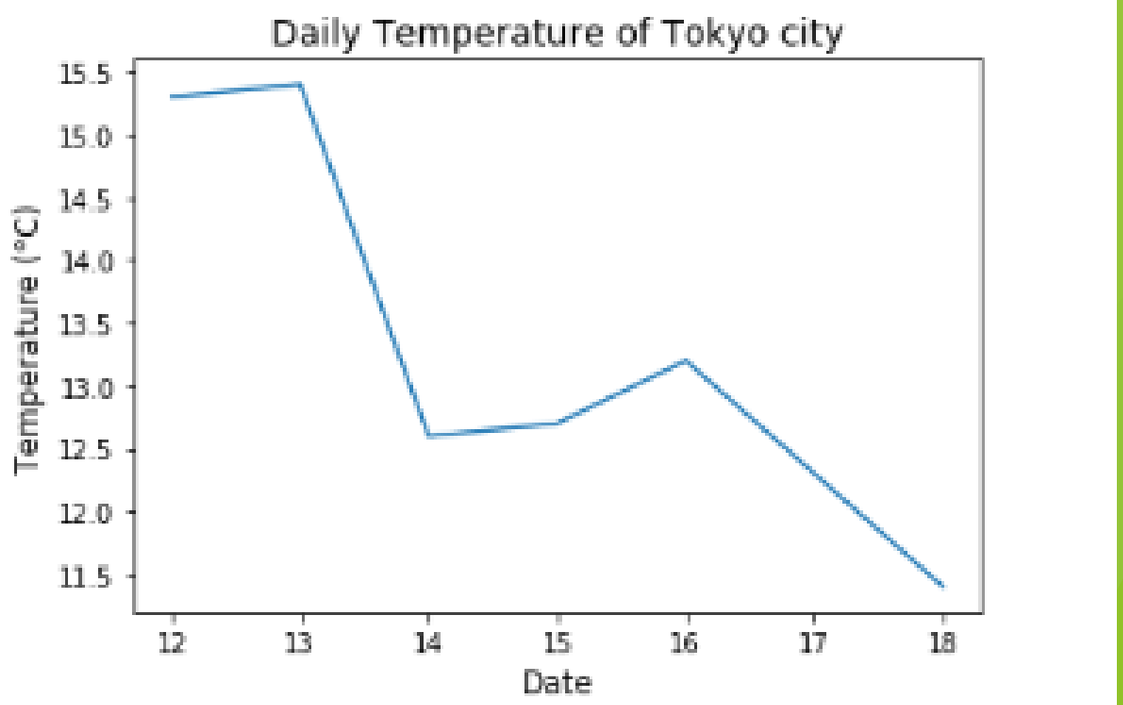
```
import matplotlib.pyplot as plt

# Prepare the data series
d = [12,13,14,15,16,17,18]
t0 = [15.3,15.4,12.6,12.7,13.2,12.3,11.4]
t1 = [26.1,26.2,24.3,25.1,26.7,27.8,26.9]
t2 = [22.3,20.6,19.8,21.6,21.3,19.4,21.4]

# Plot the lines for each data series
plt.plot(d,t0,label='Tokyo')
plt.plot(d,t1,label='Hawaii')
plt.plot(d,t2,label='Hong Kong')

# Set the limit for each axis
plt.xlim(12,18)
plt.ylim(10,30)
# Adding axis label
plt.xlabel('Date',size=10)
plt.ylabel('Temperature (°C)',size=10)
# Adding a grid
plt.grid(True,linewidth=0.5,color='#aaaaaa',linestyle='-')
# Adding a title
plt.title("Daily Temperature of 3 cities in the second week of
December", size=14, fontweight='bold')
# Adding a legend
plt.legend(loc='upper right' )
# Show the plot
plt.show()
```



# Saving Plot to a File

- To save a figure, we put **plt.savefig(outputpath)** at the end of plotting commands. It can be used in place of **plt.show()**, to directly save the figure without displaying it.
- If you want to save the figure as a file as well as display it on the notebook output, you can call both **plt.savefig()** and **plt.show()**.

```
import matplotlib.pyplot as plt

# Prepare the data series
date = [12,13,14,15,16,17,18]
Tokyo = [15.3,15.4,12.6,12.7,13.2,12.3,11.4]

# Plot the lines for each data series
plt.plot(date,Tokyo)

# Adding axis label
plt.xlabel('Date',size=12)
plt.ylabel('Temperature (°C)',size=12)

# Adding a title
plt.title("Daily Temperature of Tokyo city", size=14)
plt.show()
plt.savefig('123')
```



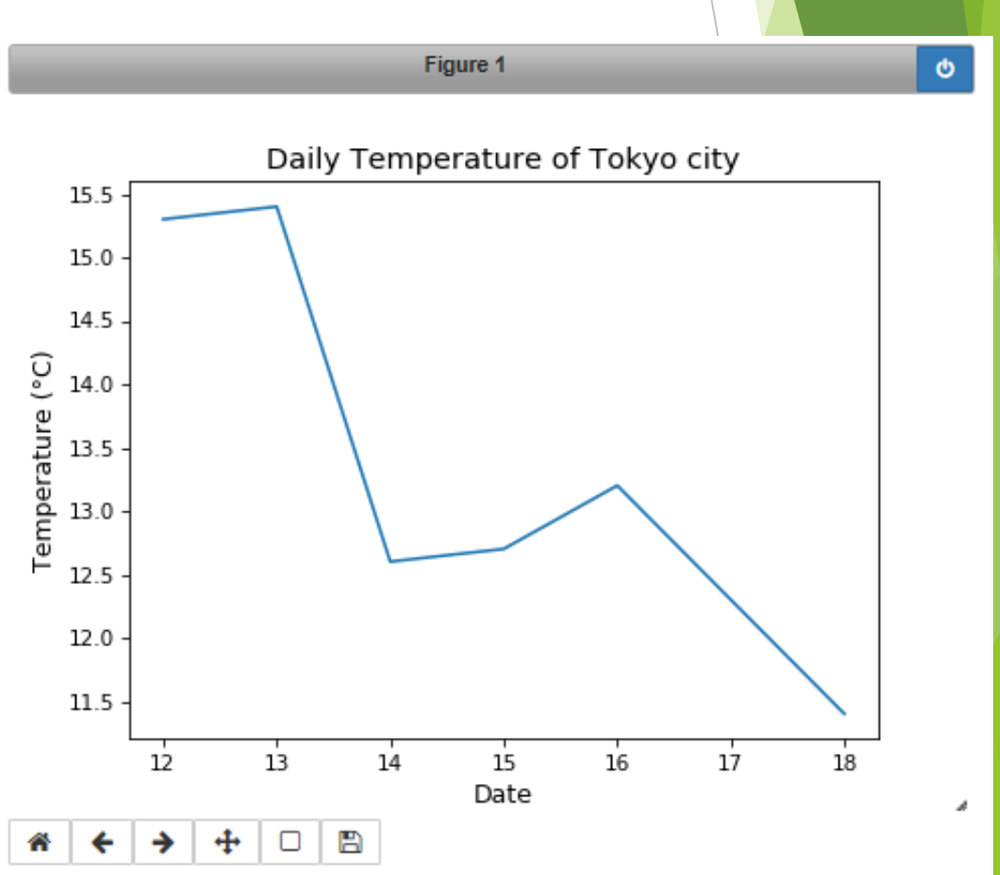

- Reversing the order can result in the plot elements being cleaned up, leaving a blank canvas for the saved figure file.

# Interactive Navigation Toolbar

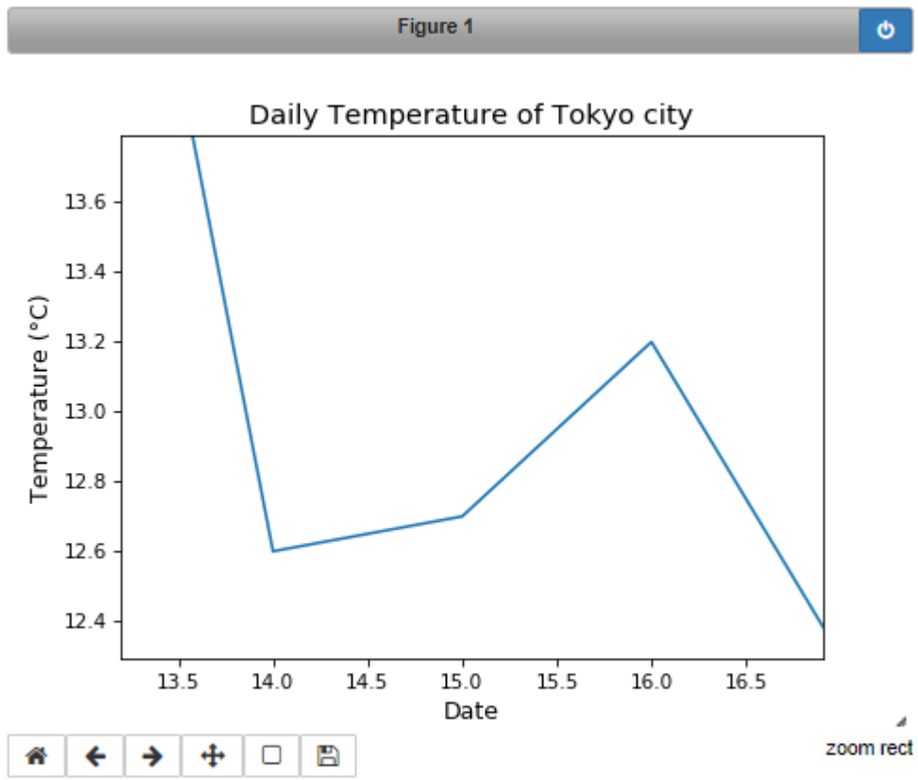
- Matplotlib is well integrated into Jupyter Notebook natively; such integration allows the plots to be displayed directly as static images as the output of each notebook cell. At times, we might be interested in the interactive GUI of Matplotlib, such as for zooming or panning a plot to view from different angles.
- Reversing the order can result in the plot elements being cleaned up, leaving a blank canvas for the saved figure file.

```
%matplotlib notebook
```

```
import matplotlib.pyplot as plt

# Prepare the data series
date = [12,13,14,15,16,17,18]
Tokyo = [15.3,15.4,12.6,12.7,13.2,12.3,11.4]

# Plot the lines for each data series
plt.plot(date,Tokyo)
# Adding axis label
plt.xlabel('Date',size=12)
plt.ylabel('Temperature (°C)',size=12)
# Adding a title
plt.title("Daily Temperature of Tokyo city", size=14)
plt.show()
```



# Interactiv lbar

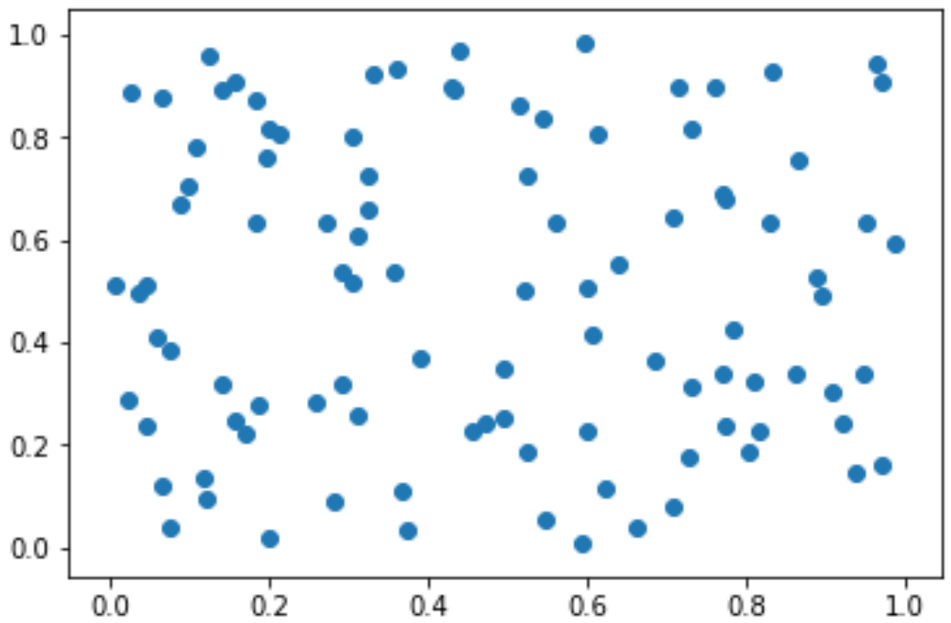



- You can find a tool bar in the bottom-left corner. The button functions from left to right are as follows:
- **Home logo**: Reset original view
- **Left arrow:** Back to previous view
- **Right arrow**: Forward to next view
- **Four-direction arrow**: Pan by holding down the left mouse key; zoom with the right arrow key on the screen
- **Rectangle**: Zoom by dragging rectangle on the plot
- **Floppy disk icon**: Download the plot

# Scatter chart

Basic plot type is scatter plot, a plot of dots. You can draw it by calling plt.scatter(x,y). The following example shows a scatter plot of random dots:

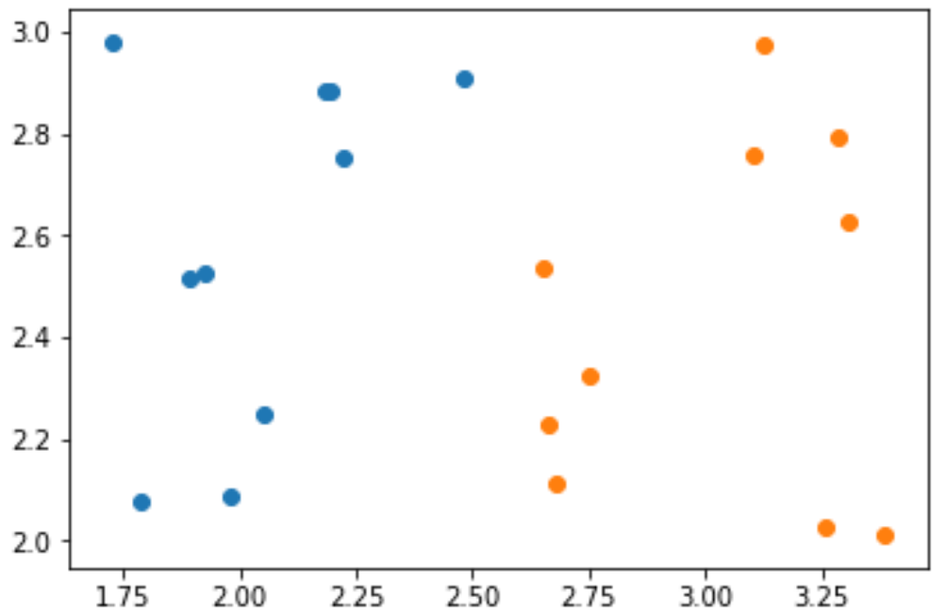
```
import numpy as np
import matplotlib.pyplot as plt

# Set the random seed for NumPy function to keep the results reproducible
np.random.seed(42)

# Generate a 2 by 100 NumPy Array of random decimals between 0 and 1
r = np.random.rand(2,100)

# Plot the x and y coordinates of the random dots on a scatter plot
plt.scatter(r[0],r[1])

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



# Scatter chart to show clusters

- While we have seen a scatter plot of random dots before, scatter plots are most useful with representing discrete data points that show a trend or clusters. Each data series will be plotted in different color per plotting command by default, which helps us distinguish the distinct dots from each series.
- We will generate two artificial clusters of data points using a simple random number generator function in NumPy

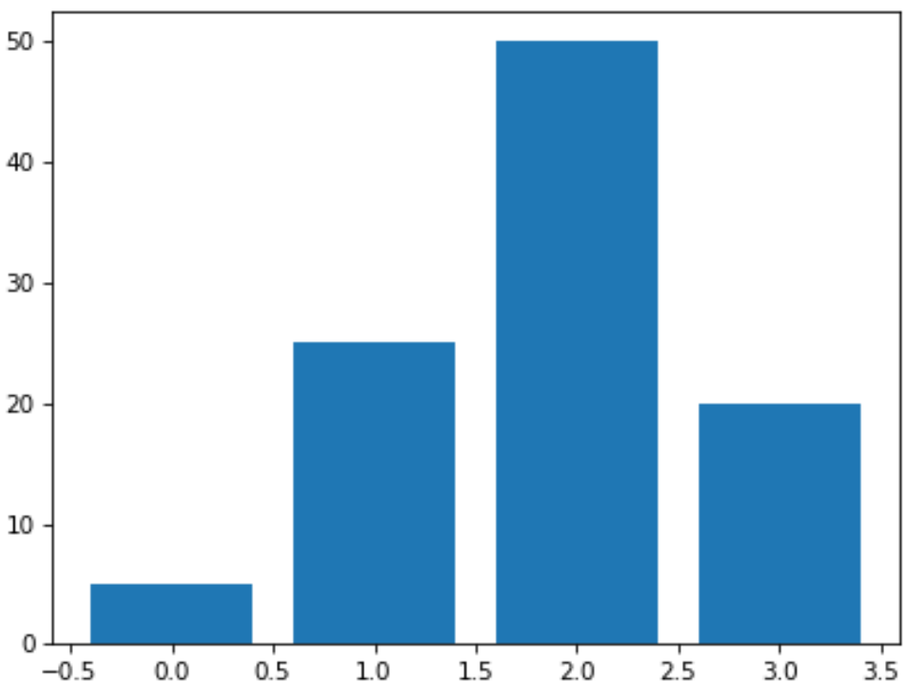
```
import matplotlib.pyplot as plt

# seed the random number generator to keep results reproducible
np.random.seed(123)

# Generate 10 random numbers around 2 as x-coordinates of the first data series
x0 = np.random.rand(10)+1.5

# Generate the y-coordinates another data series similarly
np.random.seed(321)
y0 = np.random.rand(10)+2
np.random.seed(456)
x1 = np.random.rand(10)+2.5
np.random.seed(789)
y1 = np.random.rand(10)+2
plt.scatter(x0,y0)
plt.scatter(x1,y1)

plt.show()
```



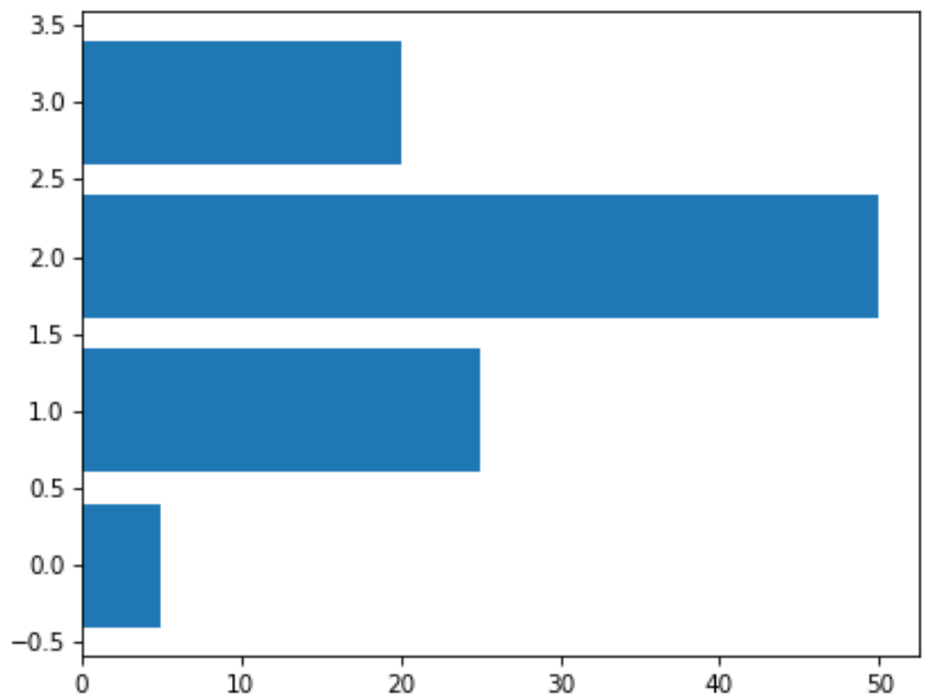
# Bar Charts

- A bar chart is a graph with rectangular bars.
- The graph usually compares different categories.
- The graphs can be plotted vertically (bars standing up) or horizontally (bars laying flat from left to right), the most usual type of bar graph is vertical.

```
import matplotlib.pyplot as plt

data = [5., 25., 50., 20.]

plt.bar(range(len(data)), data)
plt.show()
```




```
import matplotlib.pyplot as plt

data = [5., 25., 50., 20.]

plt.barh(range(len(data)), data)
plt.show()
```



# Histogram

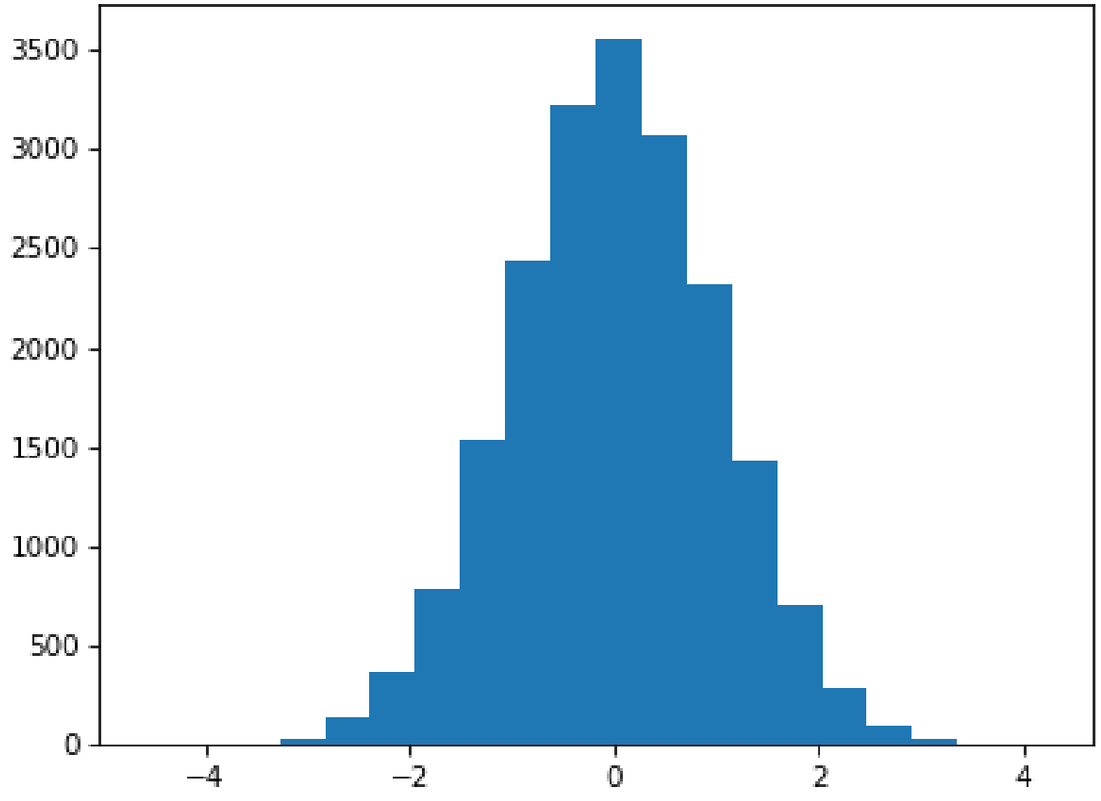
Histograms are graphical representations of a probability distribution.

A histogram is just a specific kind of a bar chart.

The following script draws 1000 values from a normal distribution and then generates histograms with 20 bins:

The histogram will change a bit each time we run the script as the dataset is randomly generated. The preceding script will display the following graph:

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


X = np.random.randn(20000)


plt.hist(X, bins = 20)
plt.show()
```



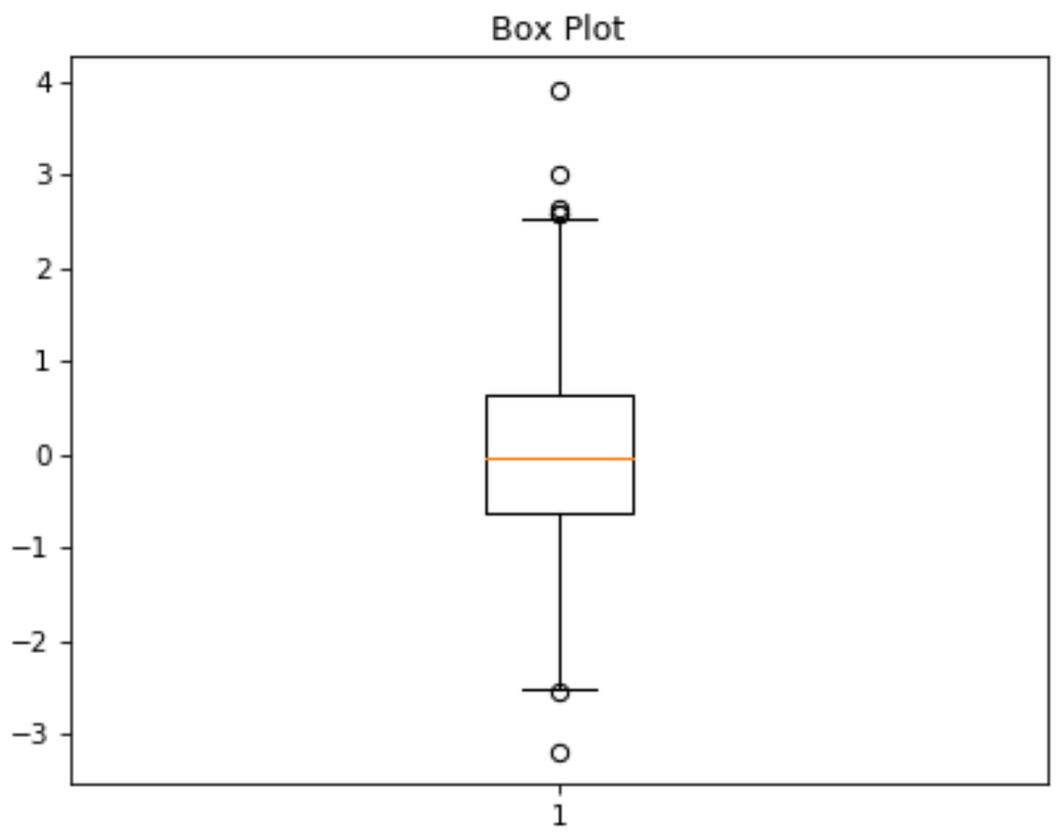
# Boxplot

Boxplot allows you to compare distributions of values by conveniently showing the

median, quartiles, maximum, and minimum of a set of values.

The following script shows a boxplot for 1000 random values drawn from a normal distribution. A boxplot will appear that represents the samples we drew from the random distribution. Since the code uses a randomly generated dataset, the resulting figure will change slightly every time the script is run.

```
import numpy as np

import matplotlib.pyplot as plt

data = np.random.randn(1000)

plt.title('Box Plot')

plt.boxplot(data)

plt.show()
```



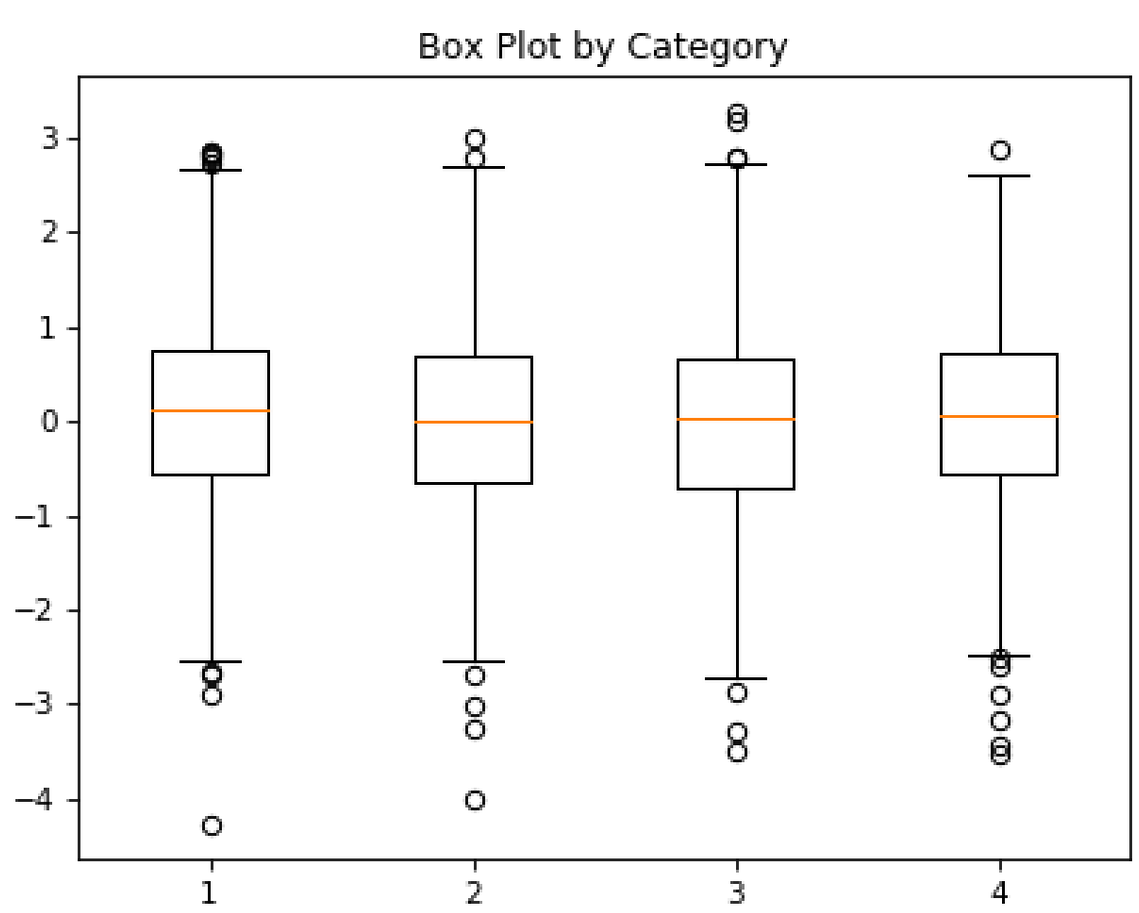
# Boxplot by Category

Boxplot allows you to compare distributions of values by conveniently showing the median, quartiles, maximum, and minimum of a set of values.

The following script shows a boxplot for 1000 random values drawn from a normal distribution.

A boxplot will appear that represents the samples we drew from the random distribution. Since the code uses a randomly generated dataset, the resulting figure will change slightly every time the script is run.

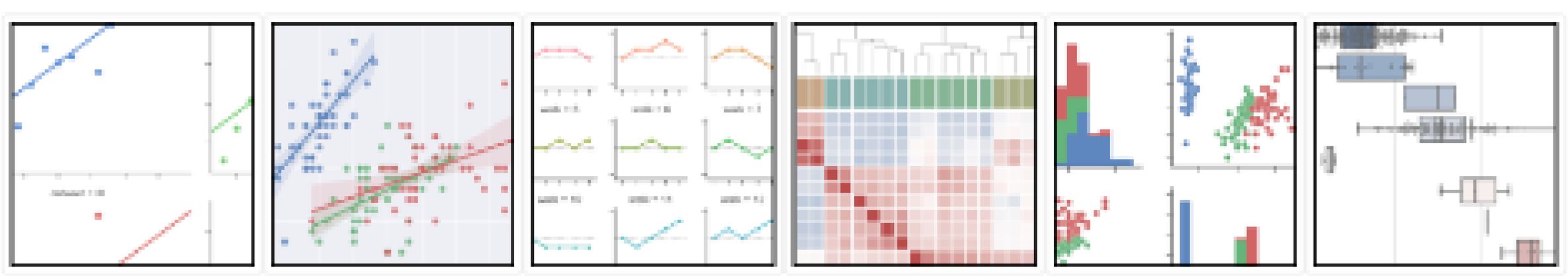
```
import numpy as np

import matplotlib.pyplot as plt

data = np.random.randn(1000, 4)

plt.title('Box Plot by

Category')

plt.boxplot(data)

plt.show()
```



# Introduction to Seaborn

# Introduction to Seaborn

- Seaborn is a library for making attractive and informative statistical graphics in Python
- It is built on top of matplotlib and tightly integrated with the PyData stack, including support for numpy and pandas data structures and statistical routines from scipy and statsmodels



Seaborn

# Features of Seaborn

- Several *built-in themes* for styling matplotlib graphics
- Tools for choosing *color palettes* to make beautiful plots that reveal patterns in your data
- Functions for visualizing *univariate and bivariate* distributions or for comparing them between subsets of data
- Tools that fit and visualize linear regression models for different kinds of independent and dependent variables
- Functions that visualize matrices of data and use clustering algorithms to discover structure in those matrices
- A function to plot statistical timeseries data with flexible estimation and representation of uncertainty around the estimate
- High-level abstractions for structuring grids of plots that let you easily build complex visualizations

# Dataset

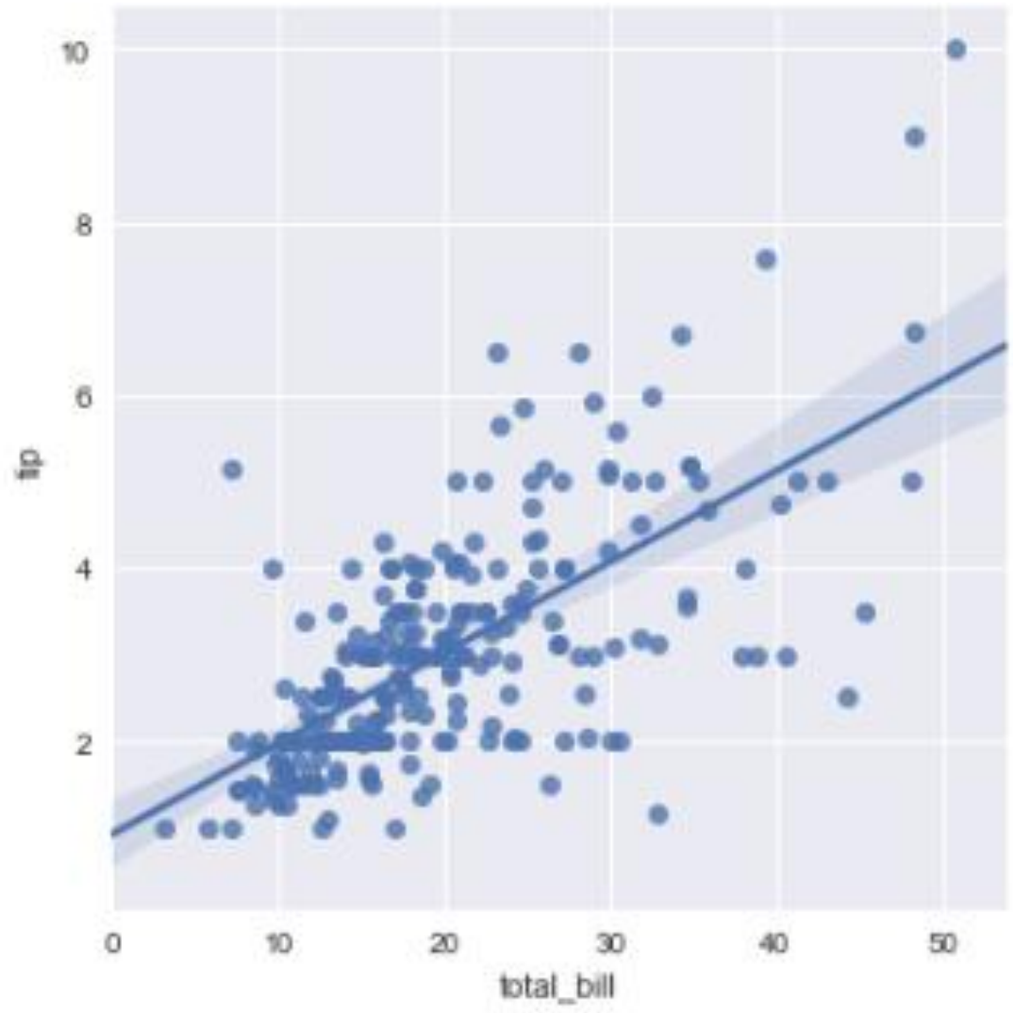
We are using tips form seaborn inbuilt dataset

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

# Scatter Plot using lmplot()

- **lmplot()** function in Seaborn is used to visualize a linear relationship as determined through regression

```
>>>import pandas as pd
>>>import matplotlib.pyplot as plt
>>>import seaborn as sns
>>>tips =sns.load_dataset('tips')
>>>sns.lmplot(x= 'total_bill', y='tip', data=tips)
>>>plt.show()
```



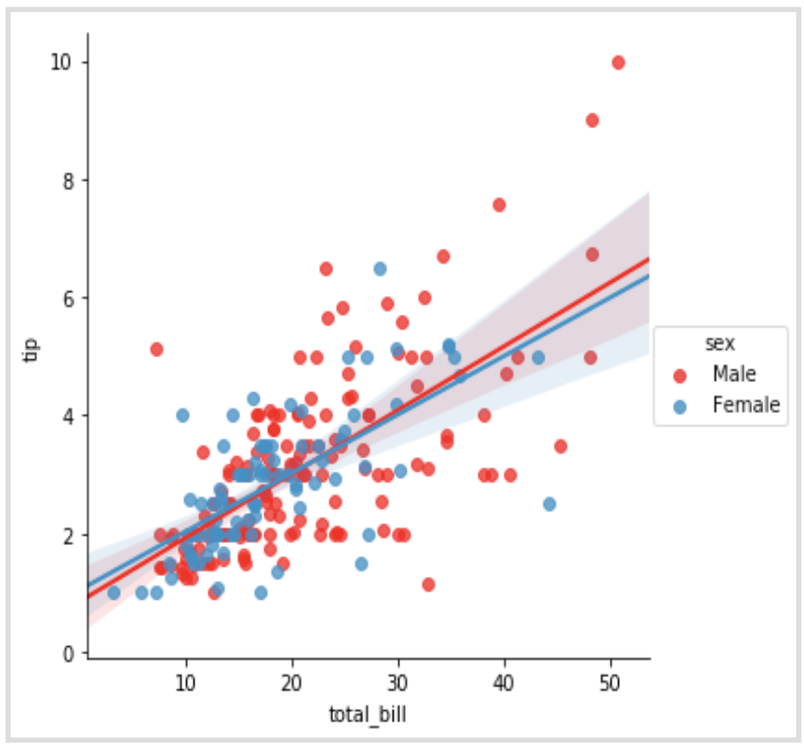
# Categorical Variable

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

# Adding colour to categorical variables using hue

- **Hue** will make parameters to control the order of levels of this variable.
- **Palette-** Colors to use for the different levels of the hue variable. Should be something that can be interpreted by color_palette(), or a dictionary mapping hue levels to matplotlib colors.

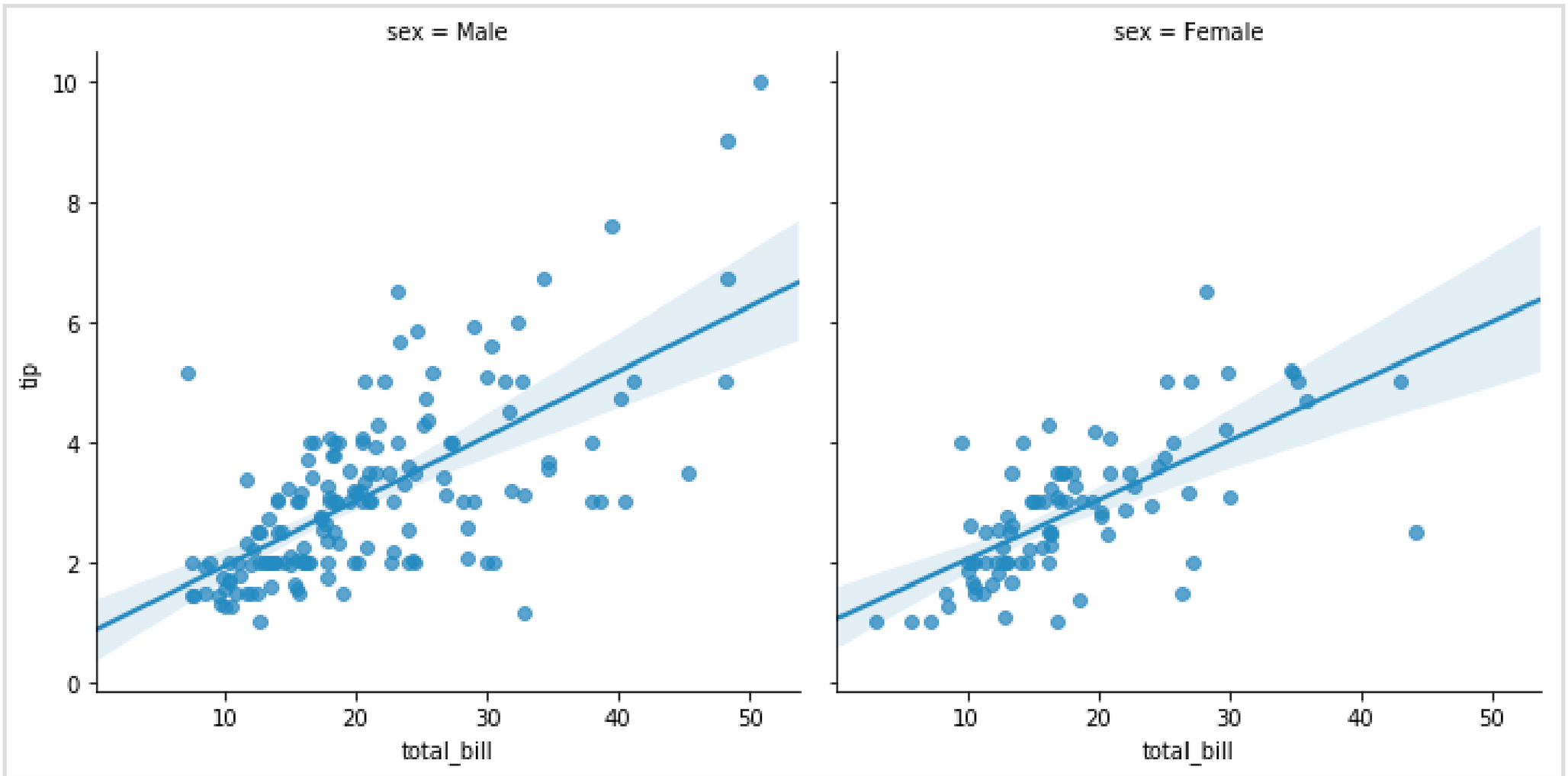
```
>>>sns.lmplot(x='total_bill', y='tip', data=tips, hue='sex',
        palette='Set1')
>>>plt.show()
```



# Separating categorical variables using col

- Provide with a plotting function and the name(s) of variable(s) in the dataframe to plot, col argument will separate the data and plot based on variables (Male and Female)

```
>>>sns.lmplot(x='total_bill', y='tip', data=tips, col='sex')
>>>plt.show()
```



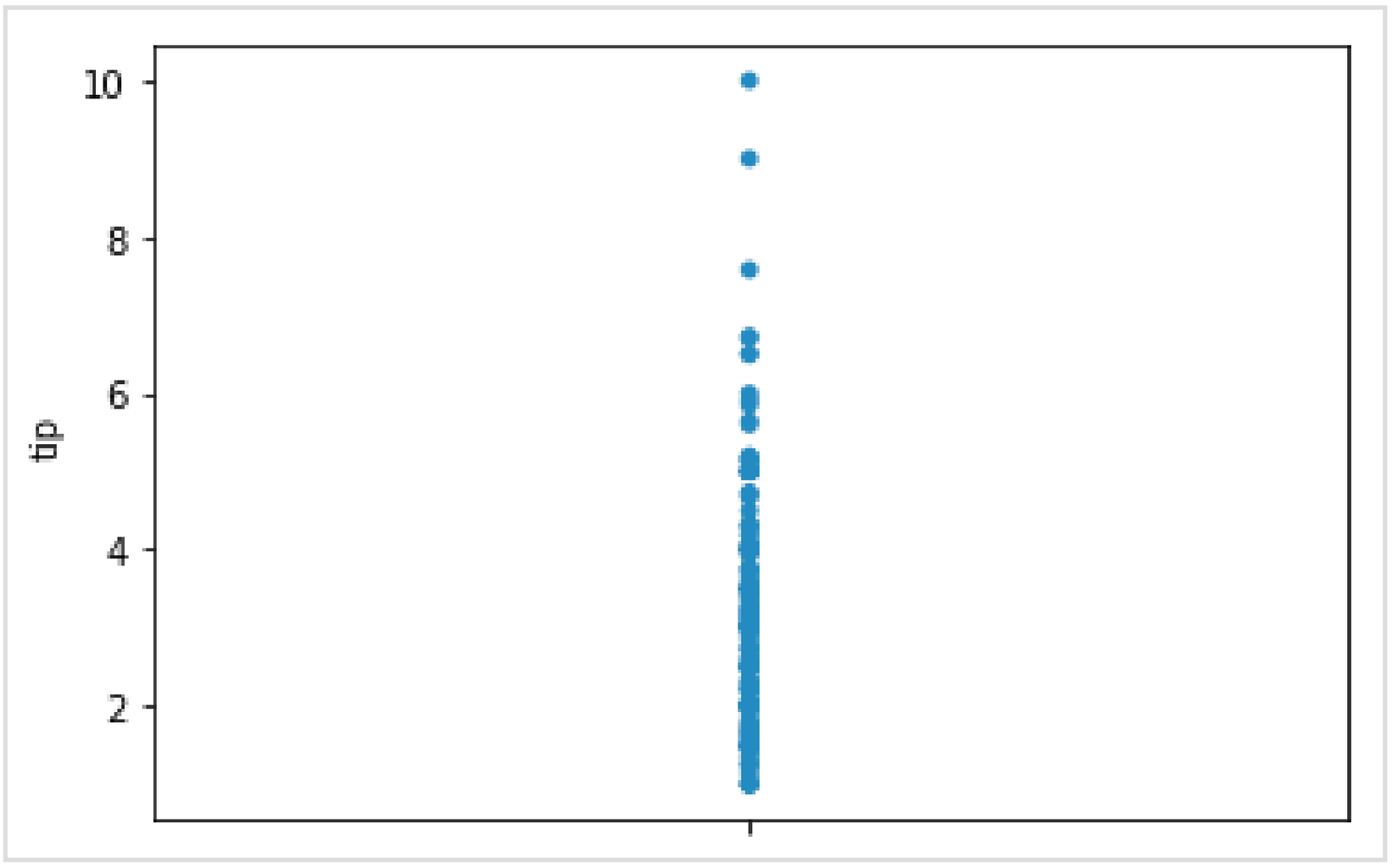
# Visualising Univariate Distributions : Strip Plots

Univariate → “one variable”

Visualization techniques for sampled univariate data

- **Strip Plots**

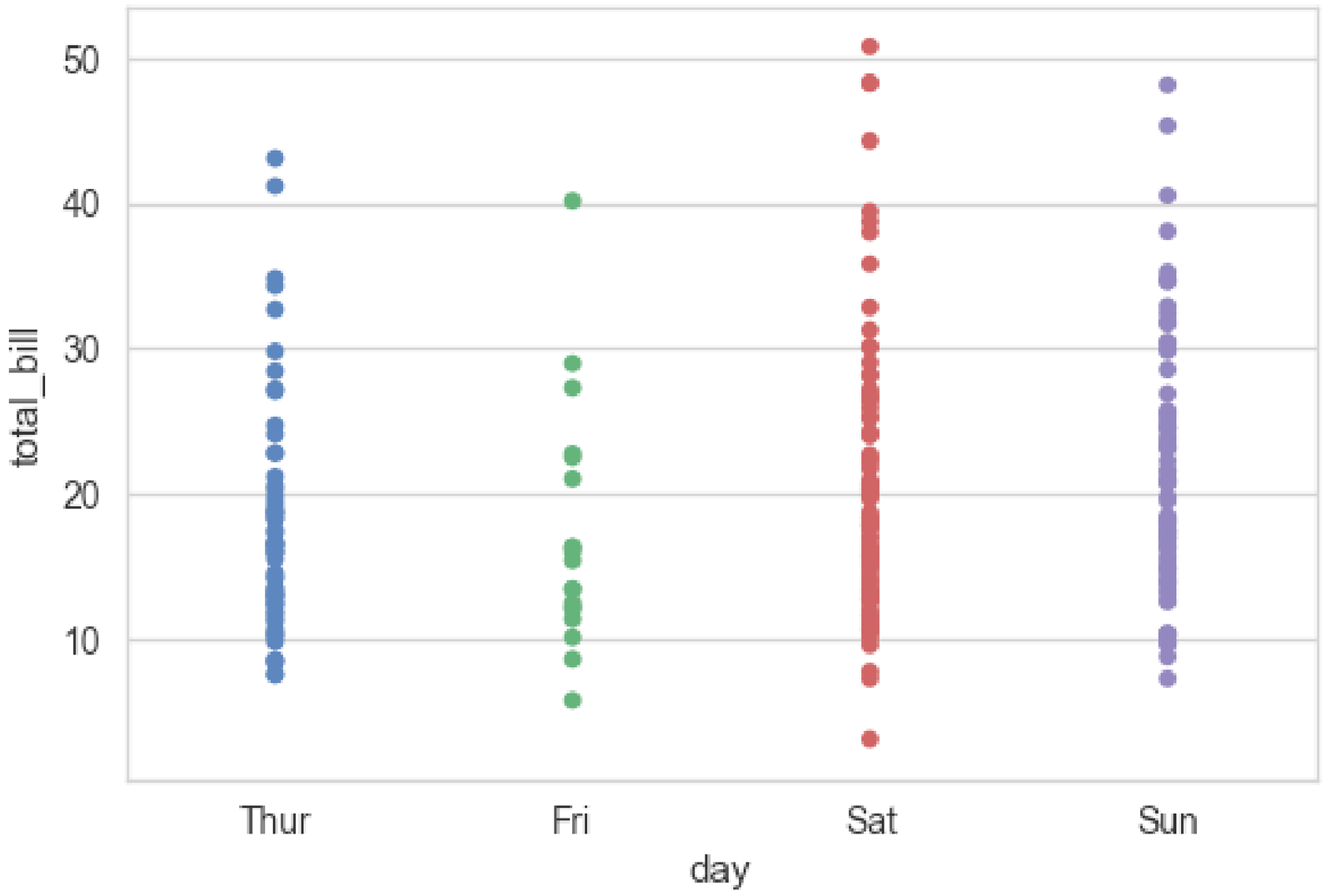
```
>>>sns.stripplot(y= 'tip', data=tips)

>>>plt.ylabel('tip ($)')

>>>plt.show()
```



# Categorical strip plots

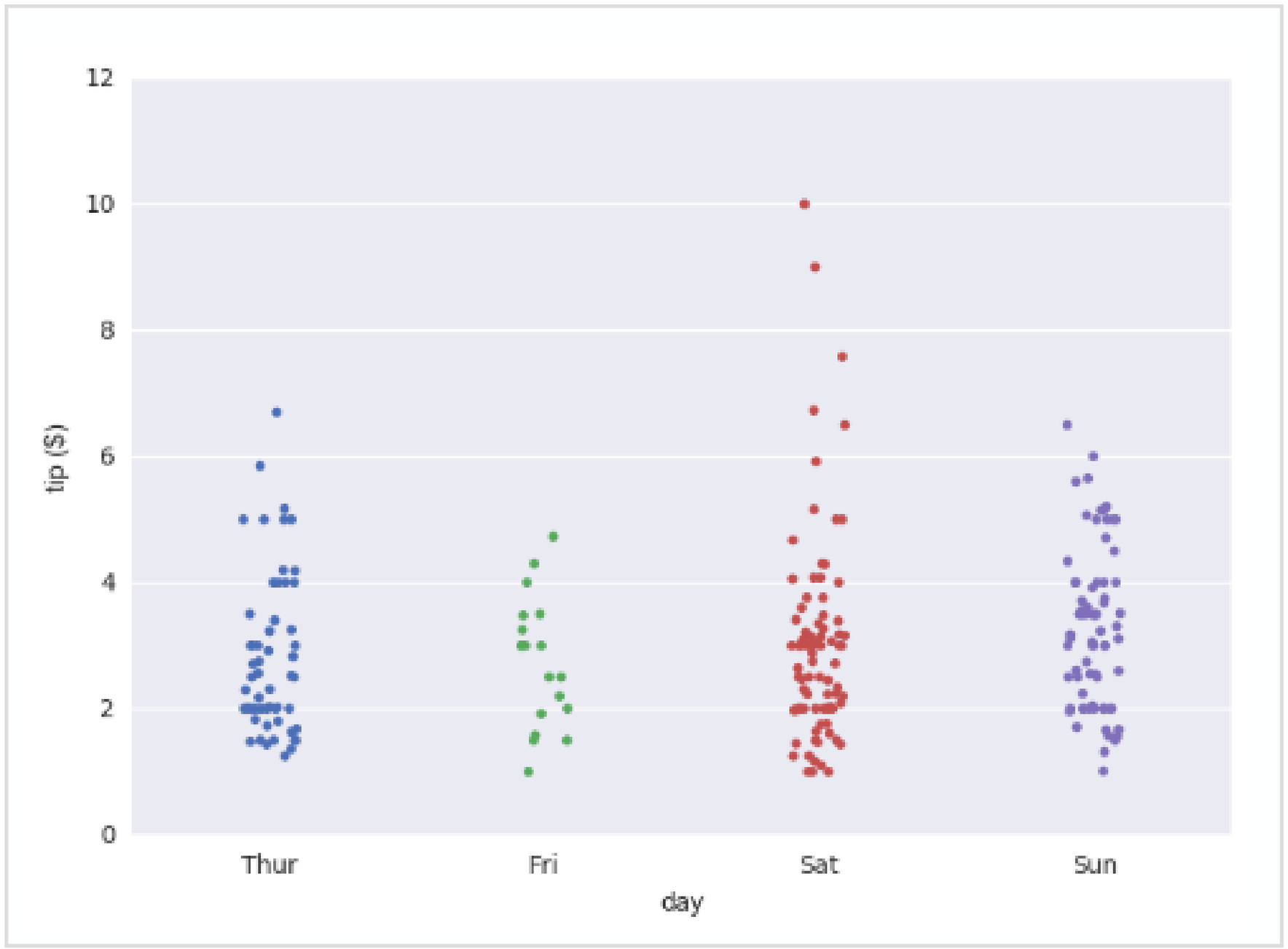
- A simple way to show the the values of some quantitative variable across the levels of a categorical variable uses stripplot(), which generalizes a scatterplot to the case where one of the variables is categorical:

```
>>>sns.stripplot(x="day", y="total_bill", data=tips)
```



# Spreading out strip plots

```
>>>sns.stripplot(x='day', y='tip', data=tip, size=4, jitter=True)
>>>plt.ylabel('tip ($)')
>>>plt.show()
```



# Visualizing Multivariate Distributions

Bivariate → “two variables”
Multivariate → “multiple variables”

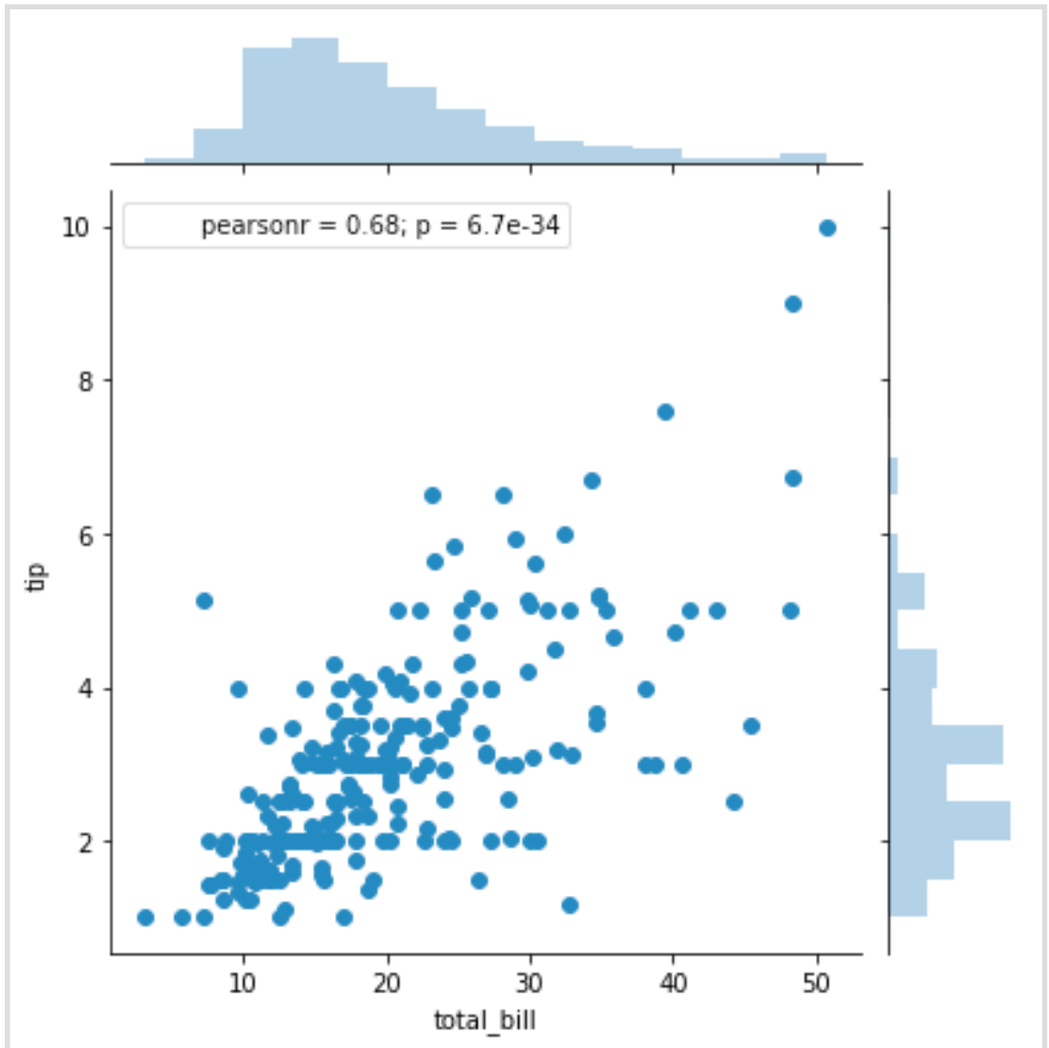
Visualizing relationships in multivariate data

- **Joint plots**
- **Pair plots**
- **Heat maps**

# Plotting bivariate distributions

- **jointplot()** creates a multi-panel figure that shows both the bivariate (or joint) relationship between two variables along with the univariate (or marginal) distribution of each on separate axes

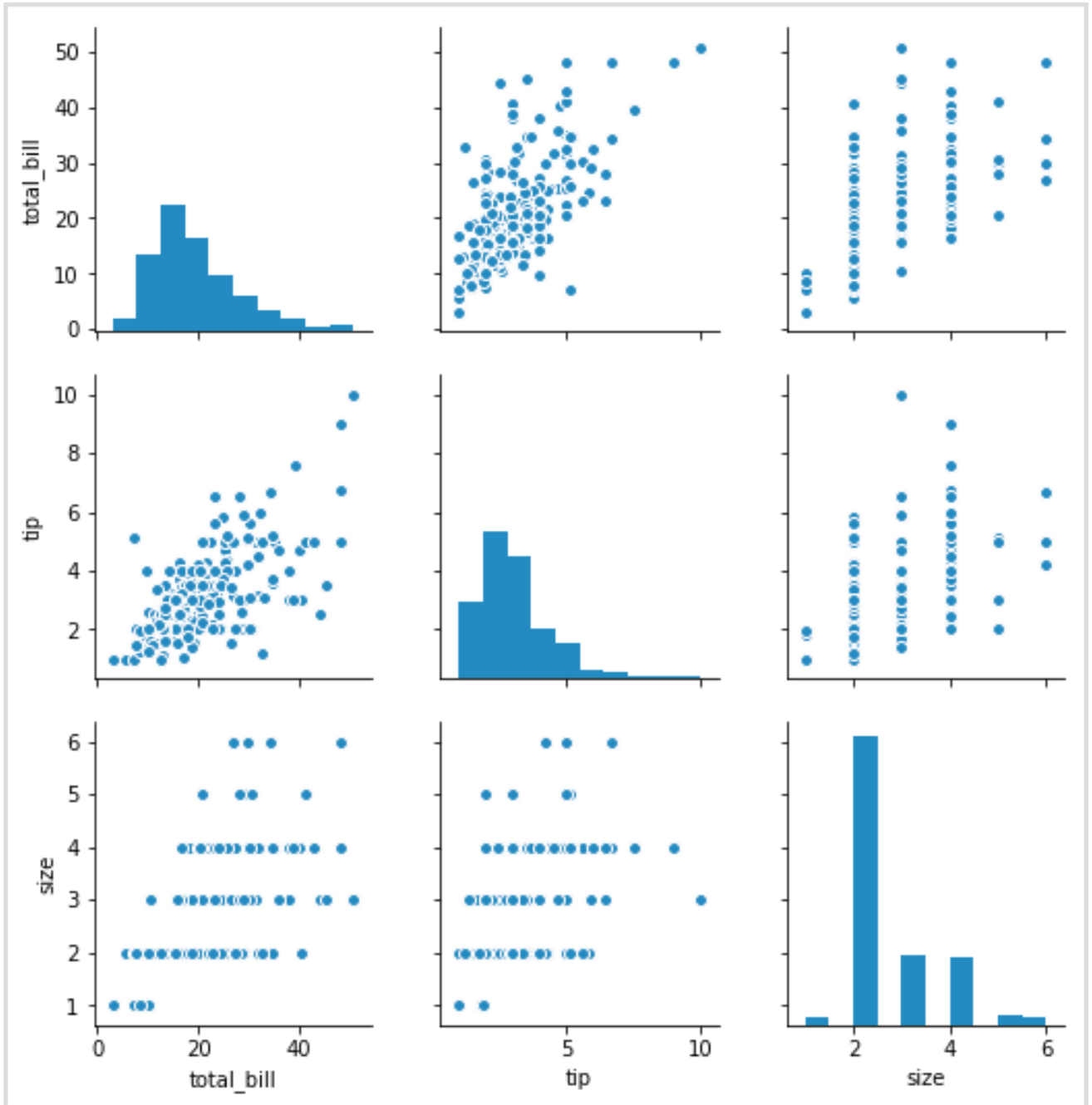
```
>>>sns.jointplot(x= 'total_bill', y= 'tip', data=tips)
>>>plt.show()
```



# Plotting multivariate distributions using pairplot()

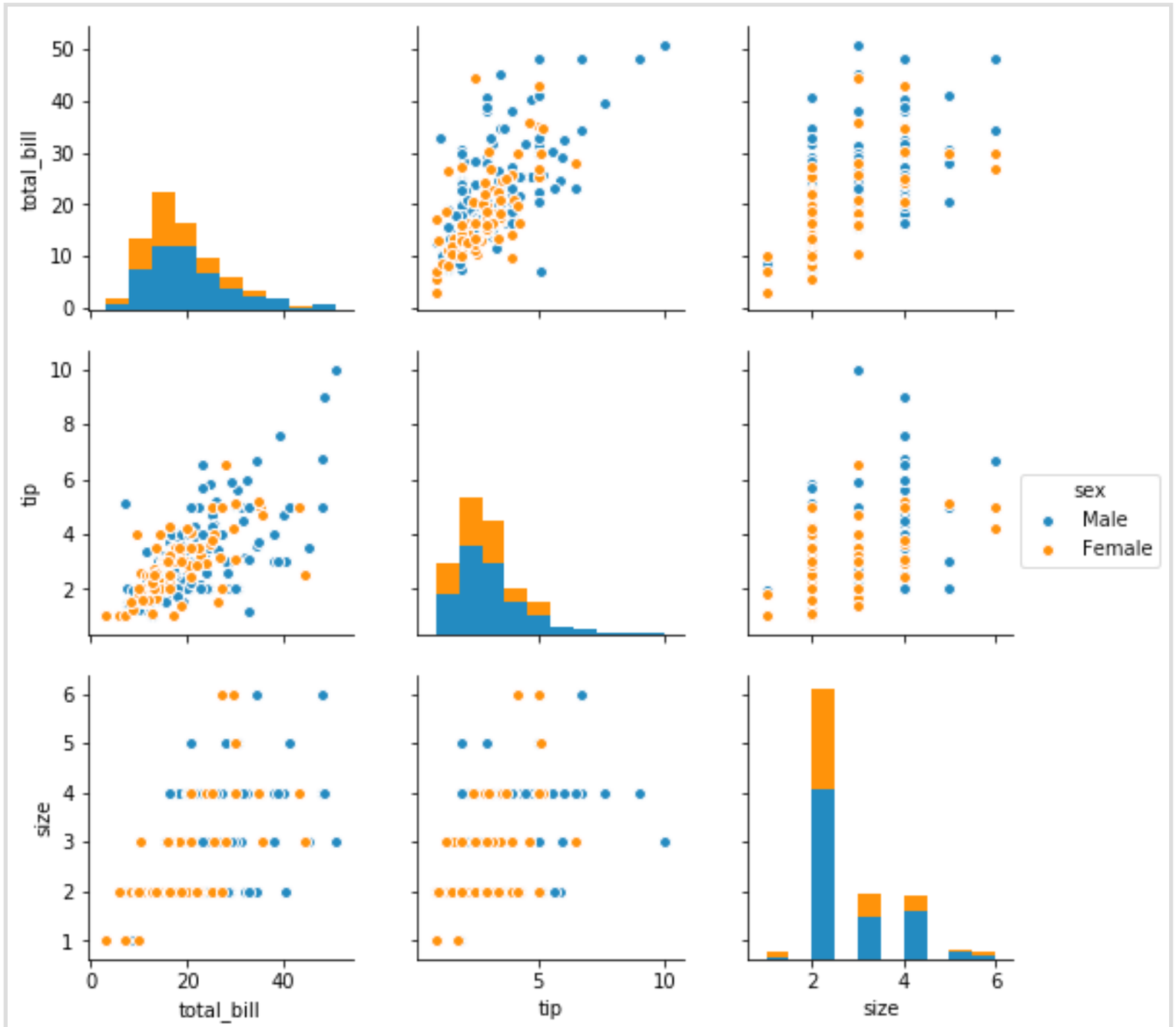
- **pairplot()** function will create a grid of Axes such that each variable in data will by shared in the y-axis across a single row and in the x-axis across a single column. The diagonal Axes are treated differently, drawing a plot to show the univariate distribution of the data for the variable in that column

```
>>>sns.pairplot(tips)

>>>plt.show()
```



# Using pairplot() with hue

```
>>>sns.pairplot(tips, hue='sex')
>>>plt.show()
```



# Plotting a correlation matrix using heatmap()

- **heatmap()** plots rectangular data as a color-encoded matrix
- **corr()** function computes the correlation matrix

```
# Considering only numeric values
>>>numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
>>>newdf = tips.select_dtypes(include=numerics)
# creating numeric correlation using pandas correlation function
>>>correlation=newdf.corr()
>>>sns.heatmap(correlation)
>>>plt.title('correlation plot')
>>>plt.show()
```

# Plotting a correlation matrix using heatmap()