Python Visualization Dashboards with Plotly's Dash Library

Udemy course by Jose Marcial Portilla

Contents

How to Use this Document:

Underlined text usually indicates a hyperlink, either to an external website or to a location within this document. Click once on the text to see the link, then click on the link to jump there. External links should open in a separate browser tab. For example, click here to jump to the heading above.

The Table of Contents at the top of this document offers similar navigation.

Lectures:

Plotly Basics

Plotly Basics Overview

This section compares Plotly to matplotlib using the same data to show the interactivity of plotly in the browser.

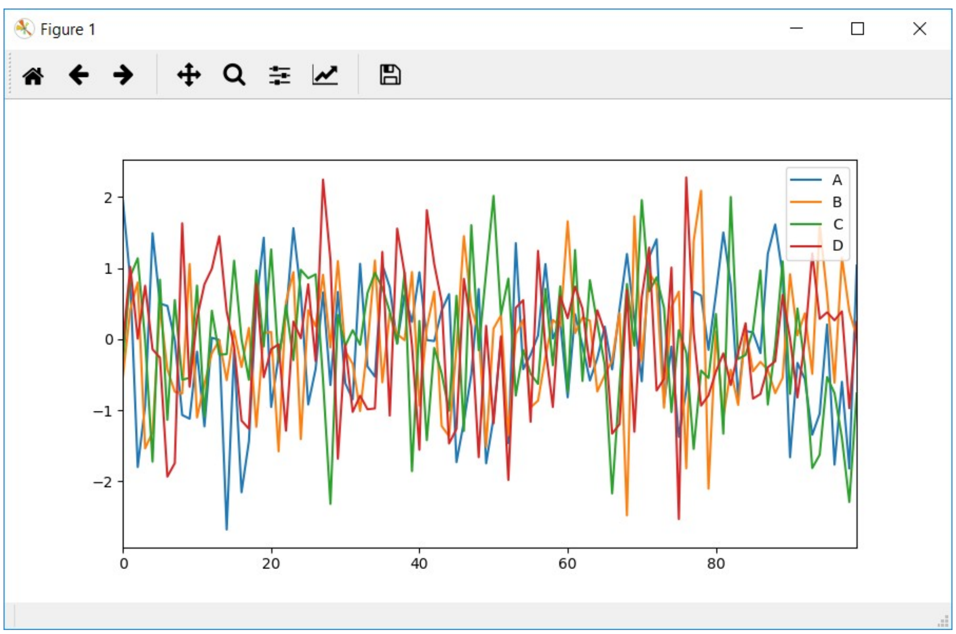
The first example provides a static matplotlib plot of four lines (called *traces*) drawn from random samples.

Create a file named **basic1.py** and add the following code:

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

# create fake data:
df = pd.DataFrame(np.random.randn(100,4),columns='A B C D'.split())
df.plot()
plt.show()
```

At the terminal run **python basic1.py**. A separate plot window should appear:



- There is no interactivity offered here, it is just a static image.
- You can save this image as a .png file if you want.
- Close the plot window to close the script.

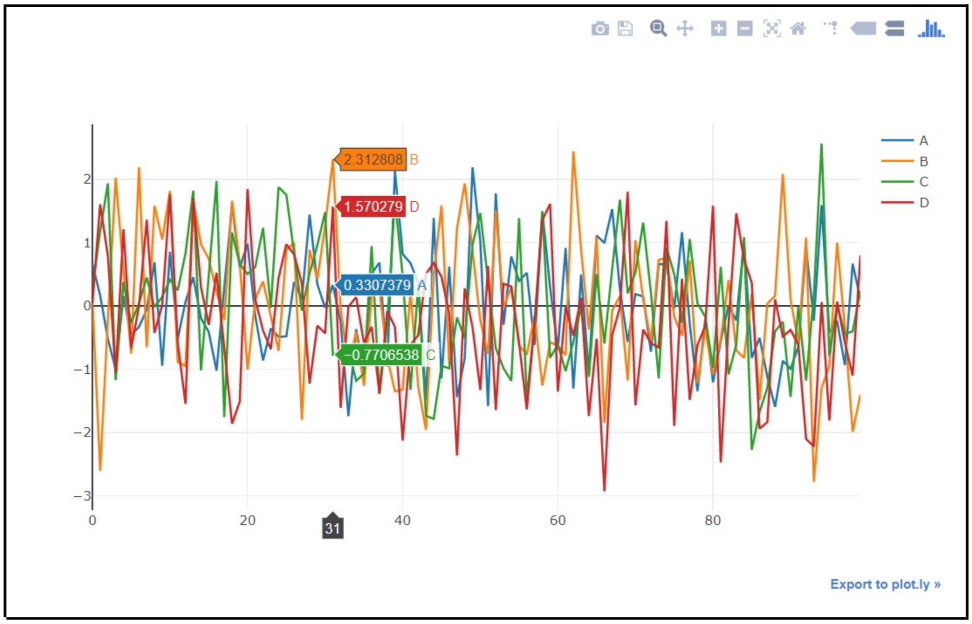
Next we'll build a Plotly plot with similar data. Create a new file called **basic2.py** and add the following code:

```
import numpy as np
import pandas as pd
import plotly.offline as pyo

# create fake data:
df = pd.DataFrame(np.random.randn(100,4),columns='A B C D'.split())
pyo.plot([{
    'x': df.index,
    'y': df[col],
    'name': col
} for col in df.columns])
```

- We've assigned the alias `pyo` to the `plotly.offline` import, to distinguish it from `import plotly.plotly as py` as shown in most online examples. Plotly offers online hosting from their website for those who set up an account with them. Throughout this course we will create offline plotly graphs and run them locally.
- **basic2.py** uses a *list comprehension* to build a trace for each column in the DataFrame. This technique is covered in more detail later.

Run this script at the terminal. A browser should open automatically and you should see something like this:



- Hover over data points in the graph to reveal specific information.
- Note that clicking on a trace (in the legend, a *trace* represents one of the displayed datasets A, B, C or D) removes it from the rest, and double-clicking a trace isolates it. Double-click again to redisplay the other traces.
- If you look to the directory where basic2.py was saved, you should see a new file named **temp-plot.html**. Plotly created this file, and this is what's showing in the browser. We'll show later how adding a **filename='something-else.html'** argument lets you change the name of the file (useful when working with multiple plots). Re-running a given script replaces earlier copies of the file.
- You can also save this plot to a static .png image file if you want.

Plots vs. Charts - we seem to use these terms interchangeably. We'll say things like "a bubble chart is a particular kind of scatter plot". The only real difference is that charts use some kind of symbol to represent the data.
From https://en.wikipedia.org/wiki/Chart:
"A chart is a graphical representation of data, in which the data is represented by symbols, such as bars in a bar chart, lines in a line chart, or slices in a pie chart. A chart can represent tabular numeric data, functions or some kinds of qualitative structure and provides different info."

Scatter Plots

A basic scatter plot maps a distribution of data points along an x- and y-axis. To illustrate, we'll take a random sample of 100 coordinate pairs, but we'll seed NumPy's random number generator so that everyone receives the same "random" sample.

Create a file named **scatter1.py** and add the following code:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

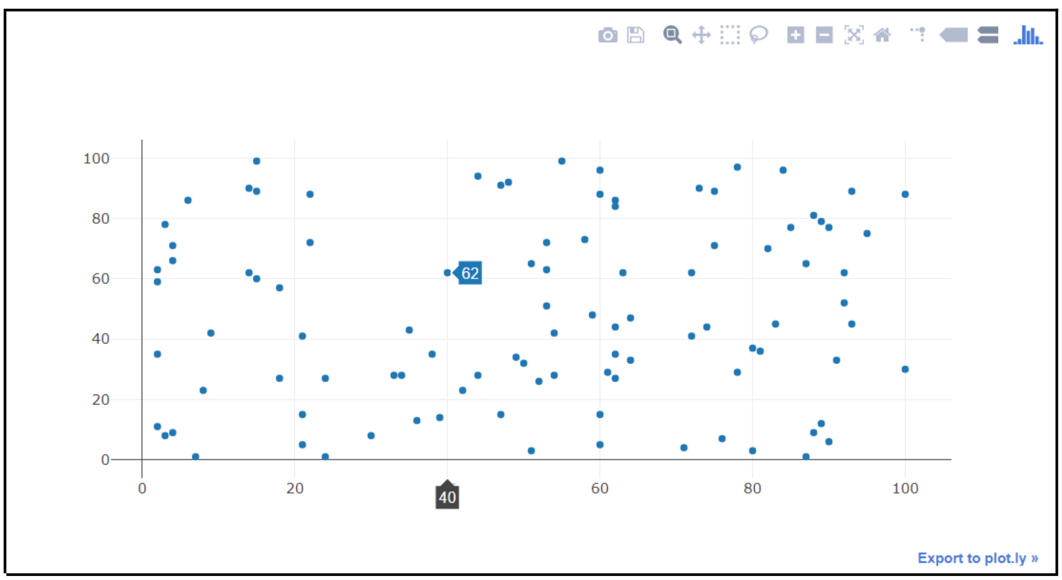
np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
)]

pyo.plot(data, filename='scatter1.html')
```

- **scatter1.py** plots 100 random coordinate pairs. By seeding the random number generator, we can reproduce the same plot each time the script is run.
- Now is a good time to mention that random number generators are algorithmic and not really random - and should never be used for cybersecurity! This explains why we can set seed values to obtain the same results.

Run the script and you should see:



- You'll notice that the plot has no title and no axis labels. To add them we'll use the graph_objs *Layout* module to add features to our graph.
- You may also notice that when you move the cursor across the graph, information is displayed about points on the graph. However, if more than one point occurs on the same vertical, you'll see that only one of the points has data displayed! Fortunately, this can be fixed by adding another parameter inside the layout.

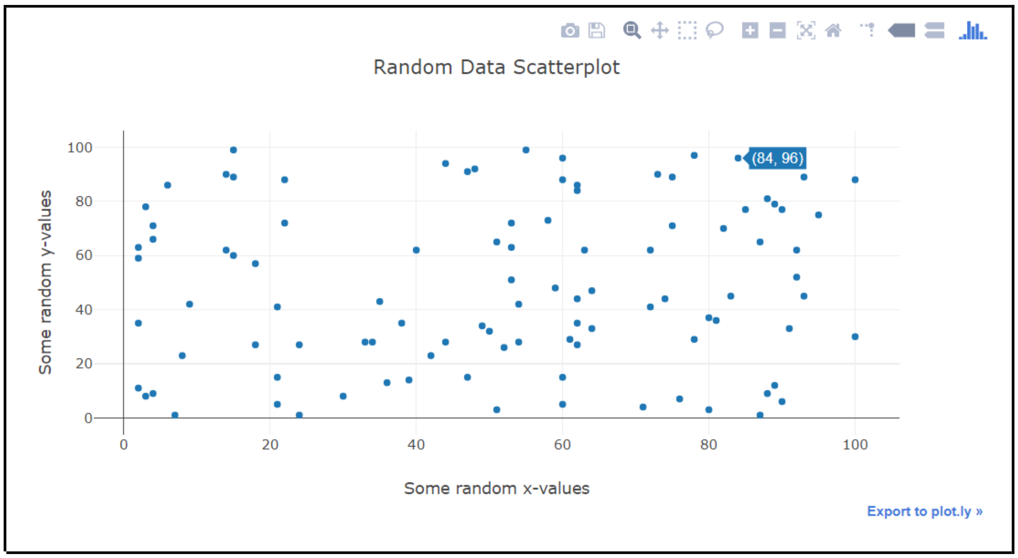
Make a duplicate of scatter1.py and name it **scatter2.py**. Add the following code (shown in bold):

```
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
)]
layout = go.Layout(
    title = 'Random Data Scatterplot', # Graph title
    xaxis = dict(title = 'Some random x-values'), # x-axis label
    yaxis = dict(title = 'Some random y-values'), # y-axis label
    hovermode ='closest' # handles multiple points landing on the same vertical
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='scatter2.html')
```

- **scatter2.py** plots the same points as scatter1, but adds a *Layout* layer which includes a title, axis labels, and fixes the hover issue. Notice how we bundled both the data and the layout inside a *Figure*, and had plotly graph the figure as HTML.

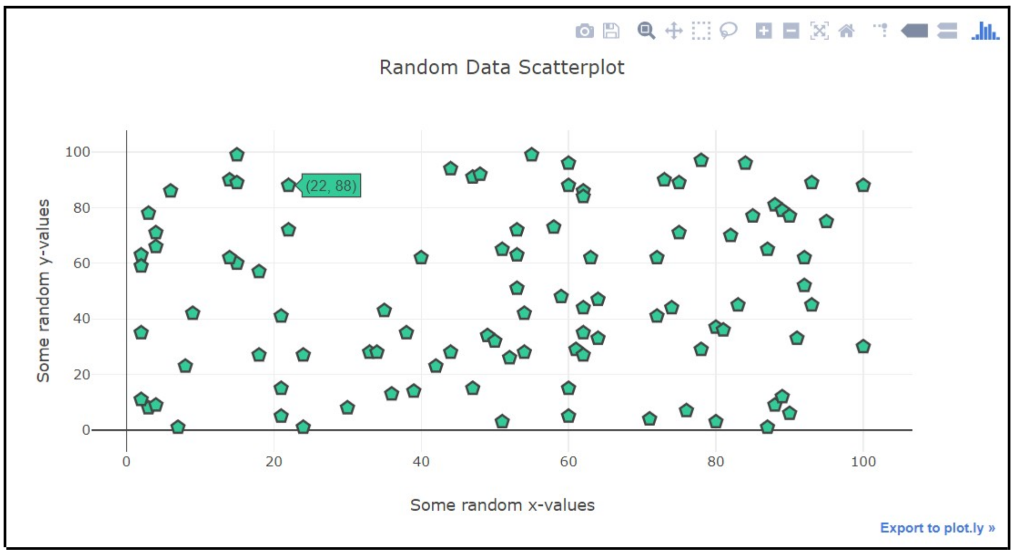


There's a lot you can do in Plotly to customize the appearance of the graph. **scatter3.py** is the same as scatter2, except we've added some style to the marker. We changed its color, size, shape, and added a line around it:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
    marker = dict(      # change the marker style
        size = 12,
        color = 'rgb(51,204,153)',
        symbol = 'pentagon',
        line = dict(
            width = 2,
        )
    )
)]
layout = go.Layout(
    title = 'Random Data Scatterplot', # Graph title
    xaxis = dict(title = 'Some random x-values'), # x-axis label
    yaxis = dict(title = 'Some random y-values'), # y-axis label
    hovermode ='closest' # handles multiple points landing on the same vertical
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='scatter3.html')
```



For more information on how you can customize your graphs, visit https://plot.ly/python/reference/#scatter

Resources: https://plot.ly/python/line-and-scatter/ and https://plot.ly/python/reference/#scatter

Line Charts

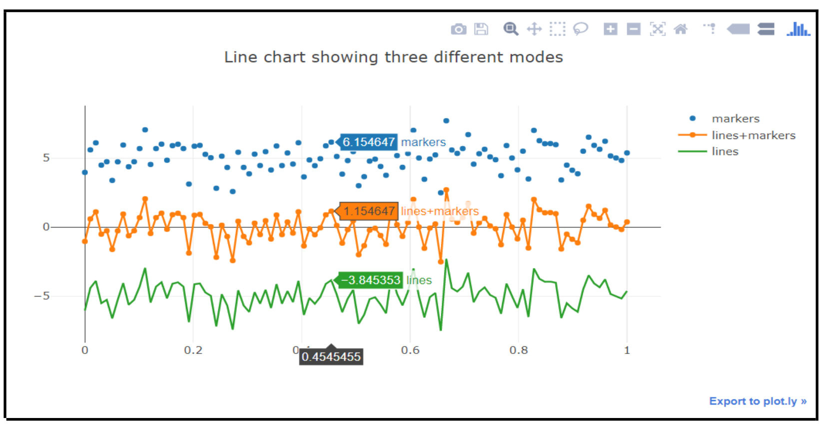
Line charts are little more than scatter plots that have only one data point per x-value, and (optionally) a line connecting the markers. To illustrate this, we'll take another random sample of data that is evenly distributed along the x-axis.

line1.py makes three copies of the same random dataset. Each set becomes a *trace*, that is, an independent set of data that appears on our graph.

```
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(56)
x_values = np.linspace(0, 1, 100) # 100 evenly spaced values
y_values = np.random.randn(100)   # 100 random values

# Create traces
trace0 = go.Scatter(
    x = x_values,
    y = y_values+5,
    mode = 'markers',
    name = 'markers'
)
trace1 = go.Scatter(
    x = x_values,
    y = y_values,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = x_values,
    y = y_values-5,
    mode = 'lines',
    name = 'lines'
)
data = [trace0, trace1, trace2]  # assign traces to data
layout = go.Layout(
    title = 'Line chart showing three different modes'
)
fig = go.Figure(data=data,layout=layout)
pyo.plot(fig, filename='line1.html')
```



- Note that each trace is assigned a name (`markers, lines+markers, lines`). Names appear in the legend to the upper right (similar to the A B C D names we saw in our first plotly example) and as hover text.

line2.py takes some online data and makes a stacked series of line charts. For this exercise we imported a dataset from the U.S. Census Bureau and whittled it down to a small file named **population.csv.** This file is stored in a neighboring folder called **/data**:

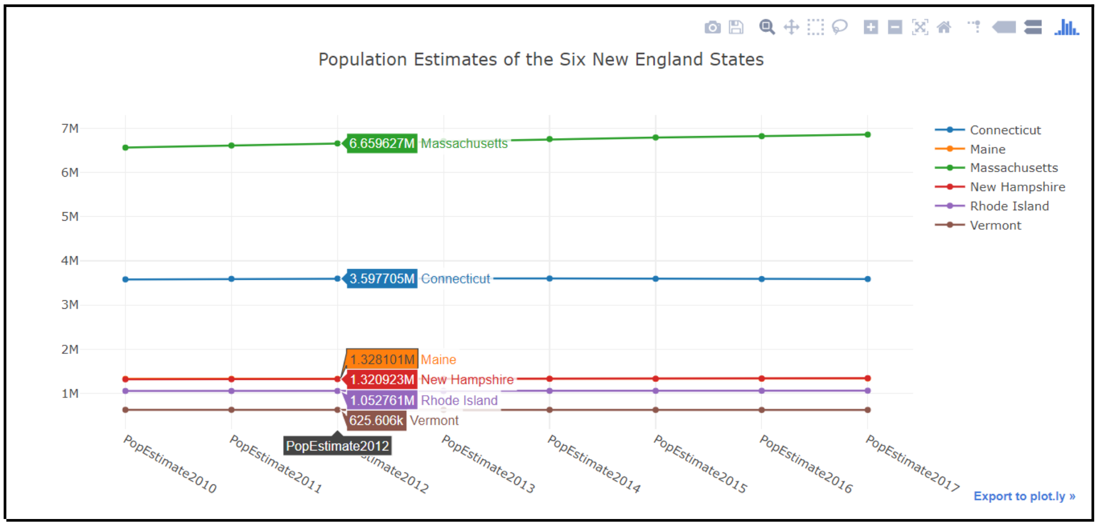
```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# read a .csv file into a pandas DataFrame:
df = pd.read_csv('../data/population.csv', index_col=0)

# create traces
traces = [go.Scatter(
    x = df.columns,
    y = df.loc[name],
```

```
    mode = 'markers+lines',
    name = name
) for name in df.index]

layout = go.Layout(
    title = 'Population Estimates of the Six New England States'
)
fig = go.Figure(data=traces,layout=layout)
pyo.plot(fig, filename='line2.html')
```



- To grab a file from a neighboring directory we use `pd.read_csv('../data/filename.csv')`
- We pass in the argument `index_col=0` to avoid having pandas add a numerical index to our data. This is described in more detail in the section Data Manipulation with Pandas.
- Similar to **basic2.py**, we use a list comprehension to extract traces from the DataFrame.
- An interesting thing about this plot is that the populations of Maine and New Hampshire are nearly equivalent, and you don't see this until you hover over the red line. If you click on New Hampshire in the legend, Maine’s orange line is revealed.

Resources: https://plot.ly/python/line-charts/

Data source: https://www.census.gov/data/datasets/2017/demo/popest/nation-total.html#ds
https://www2.census.gov/programs-surveys/popest/datasets/2010-2017/national/totals/nst-est2017-alldata.csv

Bar Charts

Bar Charts plot different categories along the x-axis, and numerical values along the y-axis. Categories are compared by looking at the height of each bar. For this reason, it’s important that the y-axis always start at zero, to avoid any visual misrepresentations.

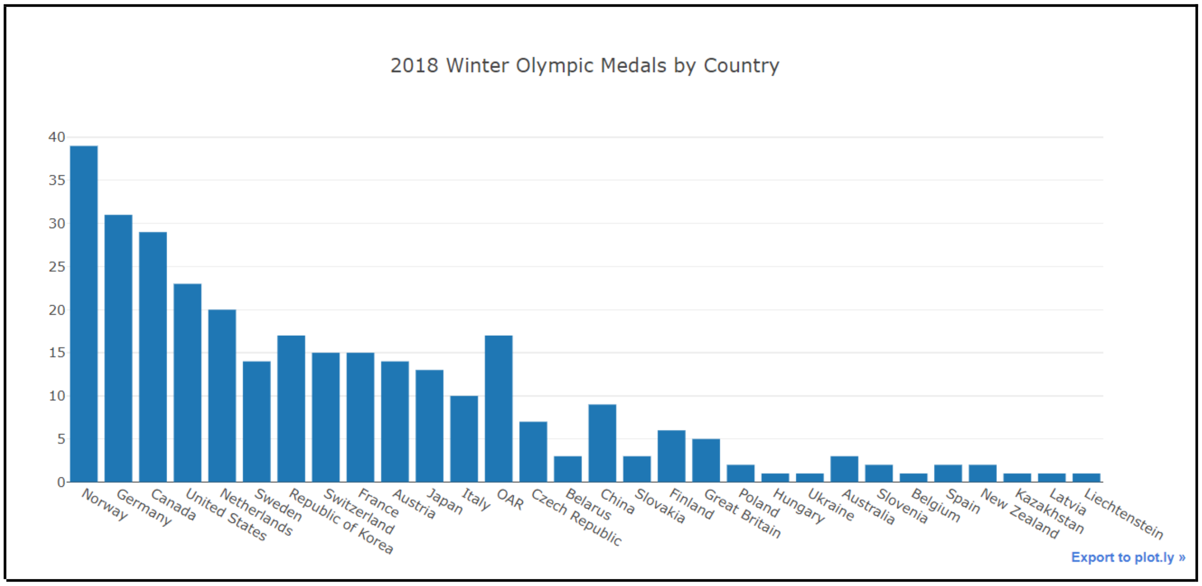
This section starts with a very simple, monochromatic bar chart showing the number of medals won by countries in the 2018 Winter Olympics in PyeongChang, South Korea.
We added a .csv file to the ../data folder called **2018WinterOlympics.csv**, and we plot the data with **bar1.py**:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2018WinterOlympics.csv')

data = [go.Bar(
    x=df['NOC'],  # NOC stands for National Olympic Committee
    y=df['Total']
)]
layout = go.Layout(
    title='2018 Winter Olympic Medals by Country'
)
fig = go.Figure(data=data, layout=layout)
```

```
pyo.plot(fig, filename='bar1.html')
```



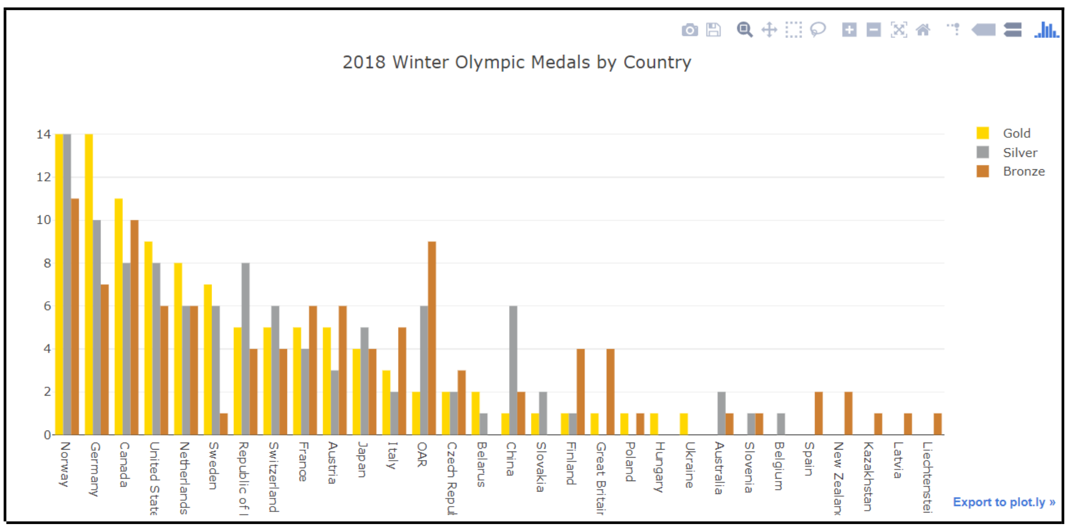
- Note that Country names fall under the NOC column - NOC stands for National Olympic Committee.
- We should mention that OAR stands for “Olympic Athletes from Russia”. Russia was banned from these Olympic games, but some athletes were invited to compete.
- Countries are ranked in scoring order from left to right, and yet some countries like South Korea earned more medals than countries that scored higher, like Sweden. We find out why on the next two plots.

Let's take a look at the types of medals earned by each country, Gold, Silver and Bronze with **bar2.py**:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2018WinterOlympics.csv')

trace1 = go.Bar(
    x=df['NOC'],  # NOC stands for National Olympic Committee
    y=df['Gold'],
    name = 'Gold',
    marker=dict(color='#FFD700') # set the marker color to gold
)
trace2 = go.Bar(
    x=df['NOC'],
    y=df['Silver'],
    name='Silver',
    marker=dict(color='#9EA0A1') # set the marker color to silver
)
trace3 = go.Bar(
    x=df['NOC'],
    y=df['Bronze'],
    name='Bronze',
    marker=dict(color='#CD7F32') # set the marker color to bronze
)
data = [trace1, trace2, trace3]
layout = go.Layout(
    title='2018 Winter Olympic Medals by Country'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bar2.html')
```



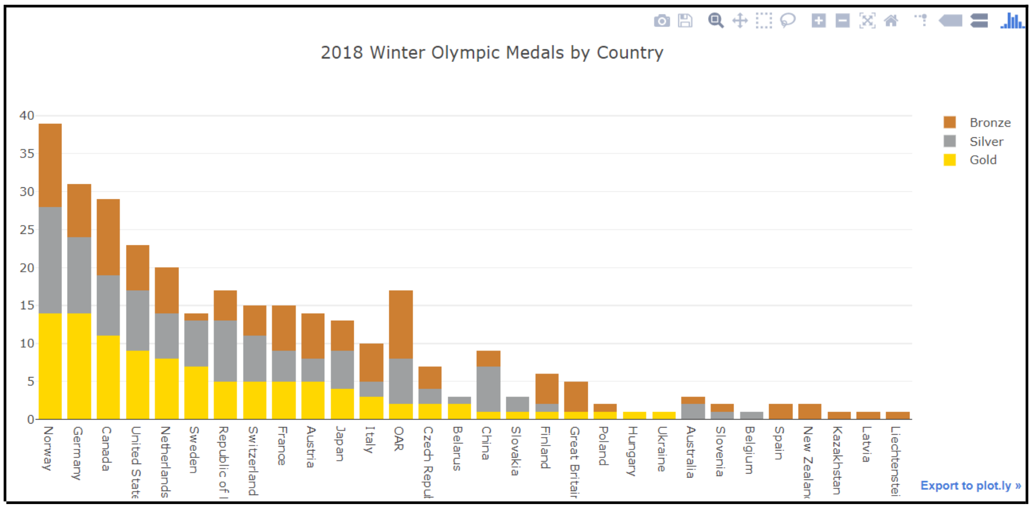
- This places Gold/Silver/Bronze side-by-side in a **Grouped Bar Chart.**
- We've added **customized colors** to each trace.
- However, in this example it's hard to see the effect that different medals have on overall score. In the next example we'll stack the bars.

bar3.py does a **Stacked Bar Chart**. Note the addition of `barmode='stack'` in the *layout* section.

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2018WinterOlympics.csv')

trace1 = go.Bar(
    x=df['NOC'],  # NOC stands for National Olympic Committee
    y=df['Gold'],
    name = 'Gold',
    marker=dict(color='#FFD700') # set the marker color to gold
)
trace2 = go.Bar(
    x=df['NOC'],
    y=df['Silver'],
    name='Silver',
    marker=dict(color='#9EA0A1') # set the marker color to silver
)
trace3 = go.Bar(
    x=df['NOC'],
    y=df['Bronze'],
    name='Bronze',
    marker=dict(color='#CD7F32') # set the marker color to bronze
)
data = [trace1, trace2, trace3]
layout = go.Layout(
    title='2018 Winter Olympic Medals by Country',
    barmode='stack'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bar3.html')
```



- Because Gold is placed at the bottom, you can see now why Sweden outscored South Korea!

Resources: https://plot.ly/python/bar-charts/ and https://plot.ly/python/reference/#bar

Data sources: http://time.com/5143796/winter-olympic-medals-by-country-2018/ and https://www.pyeongchang2018.com/en/game-time/results/OWG2018/en/general/medal-standings.htm

Bubble Charts

Bubble charts are simply scatter plots with the added feature that the size of the marker can be set by the data.

For this exercise we look at the **mpg.csv** dataset, a collection of 399 vehicles manufactured from 1970 to 1982. When brought into a DataFrame, the first ten records look like this:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
5	15.0	8	429.0	198	4341	10.0	70	1	ford galaxie 500
6	14.0	8	454.0	220	4354	9.0	70	1	chevrolet impala
7	14.0	8	440.0	215	4312	8.5	70	1	plymouth fury iii
8	14.0	8	455.0	225	4425	10.0	70	1	pontiac catalina
9	15.0	8	390.0	190	3850	8.5	70	1	amc ambassador dpl

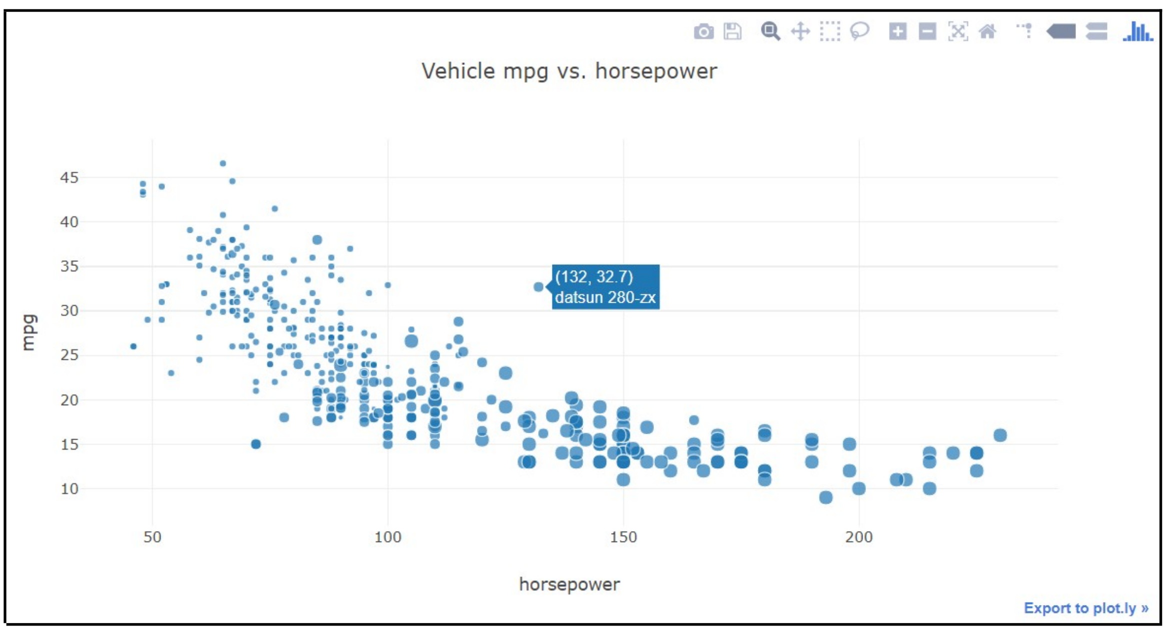
bubble1.py compares mpg to horsepower. The size of the marker is set by the number of cylinders.

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Scatter(           # start with a normal scatter plot
    x=df['horsepower'],
    y=df['mpg'],
    text=df['name'],
    mode='markers',
    marker=dict(size=1.5*df['cylinders']) # set the marker size
)]

layout = go.Layout(
    title='Vehicle mpg vs. horsepower',
    xaxis = dict(title = 'horsepower'), # x-axis label
    yaxis = dict(title = 'mpg'),        # y-axis label
    hovermode='closest'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bubble1.html')
```



- The graph shows a definite relationship between high horsepower and low mpg, and also displays a trend toward higher horsepower with greater numbers of cylinders (note that displacement is not factored here).
- We added **text** to each marker to show the name of the vehicle on hover.

- We added **hovermode='closest'** to the Layout - otherwise, *only the bottom-most marker is described if several markers appear on the same vertical x-value.*
- It is worth noting that bubble charts and scatter plots suffer a potential limitation, should more than one data point land on the same spot. A bubble may be a shade darker, but it's hard to tell that that multiple data points could be obscured. This limitation is addressed in the Dash section **Selected Data**, **select2.py** file, showing the "density" of similar looking scatter plots.
- **bubble2.py** is just like bubble1, except we show how to add multiple fields to the hover text. Since one of the fields was numeric (model_year), we first added a column to the DataFrame converting it to text, then another column that combined it with Name. This last column is used for the hover text.

Resources: https://plot.ly/python/bubble-charts/ and https://plot.ly/python/reference/#scatter

Data Sources: https://gist.github.com/omarish/5687264

Box Plots

At times it's important to determine if two samples of data belong to the same population. Box plots are great for this! The shape of a box plot (also called a box-and-whisker-plot) doesn't depend on aggregations like sample mean. Rather, the plot represents the true shape of the data. Also, depending on how the whiskers are constructed, box plots are useful for identifying *true* outliers of a data set. While some visualizations might arbitrarily discard the "top and bottom 5%" as outliers, a box plot identifies those points that lie far from the median *compared to the rest of the data*.

To construct the plot:

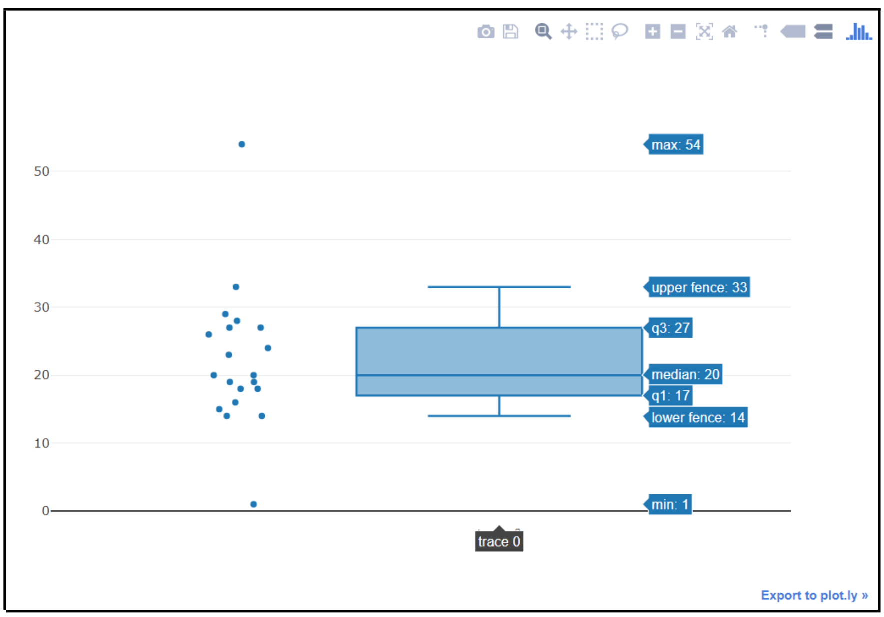
- First mark the median value (usually with a line segment). This sets the *location* of the distribution.
- Construct a box to contain all the inner-quartile values.
- Next draw the *whiskers*. There are several ways this is done, but usually you start from a distance one box-length out from the side of the box, and then come inward until you reach the first data point.
- Finally, plot any remaining points outside the whiskers as *outliers*.

box1.py takes a set of twenty points and plots them, showing one outlier:

```
import plotly.offline as pyo
import plotly.graph_objs as go

# set up an array of 20 data points, with 20 as the median value
y = [1,14,14,15,16,18,18,19,19,20,20,23,24,26,27,27,28,29,33,54]

data = [
    go.Box(
        y=y,
        boxpoints='all', # display the original data points
        jitter=0.3,      # spread them out so they all appear
        pointpos=-1.8    # offset them to the left of the box
    )
]
pyo.plot(data, filename='box1.html')
```



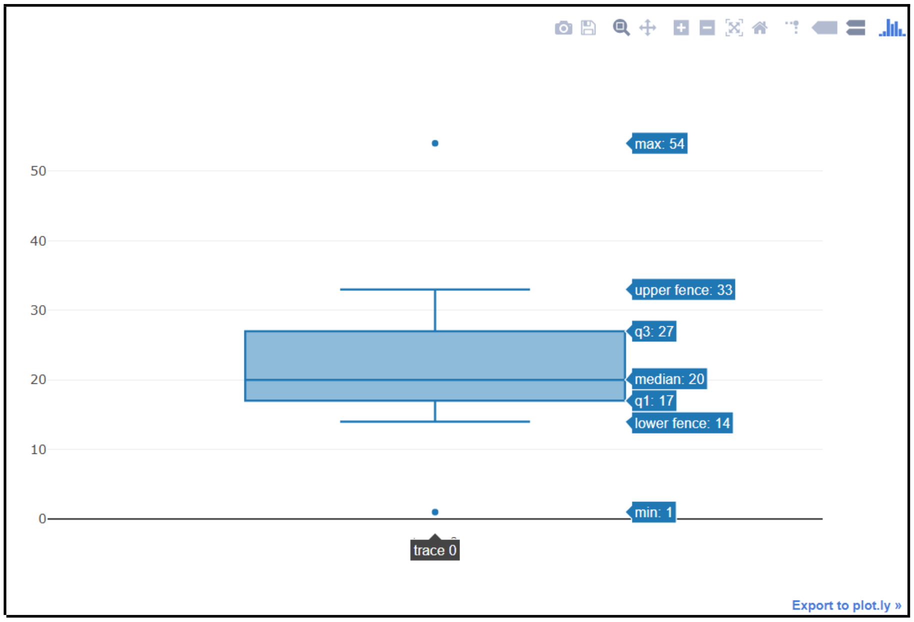
- Because we offset the data points to the left, the outlier doesn't appear over the box plot itself.

box2.py shows what a box plot would look like with displayed outliers:

```
import plotly.offline as pyo
import plotly.graph_objs as go

# set up an array of 20 data points, with 20 as the median value
y = [1,14,14,15,16,18,18,19,19,20,20,23,24,26,27,27,28,29,33,54]
```

```
data = [
    go.Box(
        y=y,
        boxpoints='outliers' # display only outlying data points
    )
]
pyo.plot(data, filename='box2.html')
```



The Quintus Curtius Snodgrass Letters

As a forensic example of applied statistics, there was a famous case where Mark Twain was accused of being a Confederate deserter during the Civil War, and the evidence given were ten essays published in the *New Orleans Daily Crescent* under the name Quintus Curtius Snodgrass. In 1963 Claude Brinegar published an article in the *Journal of the American Statistical Association* where he uses word frequencies and a chi-squared test to show that the essays were almost certainly *not* Twain's.

Brinegar's Abstract:

> "Mark Twain is widely credited with the authorship of 10 letters published in 1861 in the *New Orleans Daily Crescent*. The adventures described in these letters, which are signed "Quintus Curtius Snodgrass," provide the historical basis of a main part of Twain's presumed role in the Civil War. This study applies an old, though little used statistical test of authorship - a word-length frequency test - to show that Twain almost certainly did not write these 10 letters. The statistical analysis includes a visual comparison of several word-length frequency distributions and applications of the χ^2 and two-sample *t* tests."

The following table shows relative frequencies of three-letter-words from the Snodgrass letters, and from samples of Twain's known works. Rather than run them through complex calculations, let's make box plots!

Snodgrass	Twain		Snodgrass	Twain
.209	.225		.207	.229
.205	.262		.224	.235
.196	.217		.223	.217
.210	.240		.220	
.202	.230		.201	

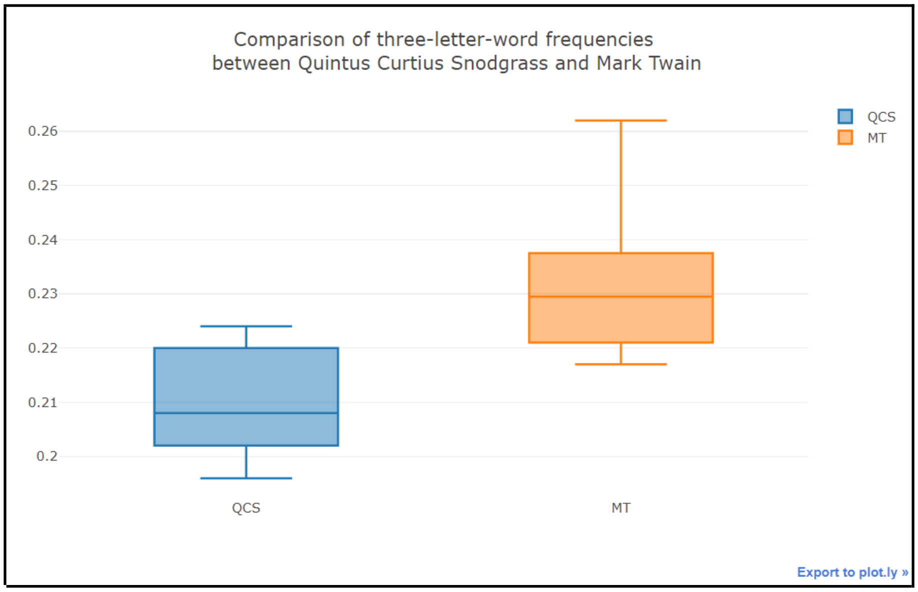
Citation: Brinegar, C., "Mark Twain and the Quintus Curtius Snodgrass Letters: A Statistical Test of Authorship", Journal. American Statistical Association, 1963, 58 (301): 85-96.

box3.py compares the two datasets side-by-side:

```
import plotly.offline as pyo
import plotly.graph_objs as go

snodgrass = [.209,.205,.196,.210,.202,.207,.224,.223,.220,.201]
twain = [.225,.262,.217,.240,.230,.229,.235,.217]

data = [
    go.Box(
        y=snodgrass,
        name='QCS'
    ),
    go.Box(
        y=twain,
        name='MT'
    )
]
layout = go.Layout(
    title = 'Comparison of three-letter-word frequencies<br>\
    between Quintus Curtius Snodgrass and Mark Twain'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='box3.html')
```



- As you can see from the plots, there's barely any overlap!
  The 10 Quintus Curtius Snodgrass letters were very likely *not* written by Mark Twain.

Resources: https://plot.ly/python/box-plots/, https://plot.ly/python/reference/#box and https://help.plot.ly/what-is-a-box-plot/

Data sources: https://www.math.utah.edu/~treiberg/M3074TwainEg.pdf
https://keepingupwiththequants.weebly.com/qcs-letters.html
https://www.jstor.org/stable/2282956?seq=1#page_scan_tab_contents

Histograms

Histograms are one of the most frequently used (and abused) visualizations. While they're great for showing which range of values has a greater frequency, it's hard to tell *how much* greater. And when converted to 3D, as seen in many flashy magazine articles, perspective can be completely distorted.

Still, if you're just starting your analysis and you want a quick peek at the data, histograms are a handy tool.

We should point out that while they look similar, histograms differ from bar charts in two important ways. First, histograms plot a numerical value along the x-axis - something that can be measured. Bar charts put categories along the x-axis, like the countries competing in the Olympics in our previous example. Second, unlike bar charts, the height of a histogram bar does not indicate frequency - rather, it's the *volume* of the bar (height x width) that does. The width of a histogram bar is determined by *binning*; since the x-axis usually displays a continuous range of values, like time or temperature, each vertical bar represents a range of values.

Also, while bar charts usually have space between bars, histograms generally have no space between adjacent bars.

For this section we'll revisit the mpg dataset. Let's take a look at a frequency distribution of mpg values from our set of 1970's era vehicles.

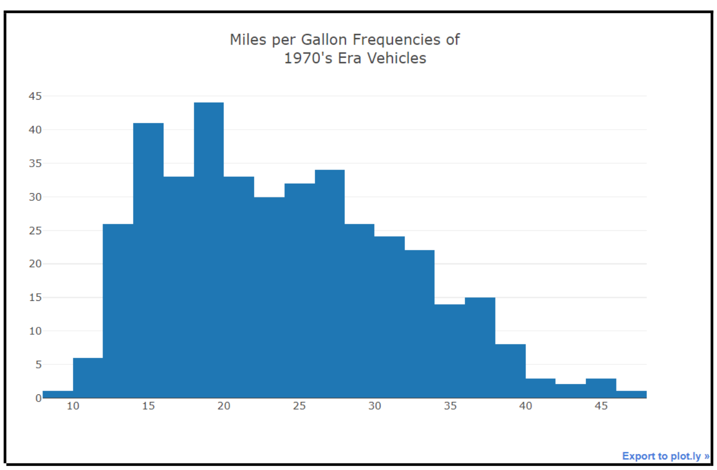
hist1.py takes plotly's default settings:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Histogram(
    x=df['mpg']
)]

layout = go.Layout(
    title="Miles per Gallon Frequencies of<br>\
    1970's Era Vehicles"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='basic_histogram.html')
```



- Note that each bin has a width of 2. The first bin spans 8 to 9.9, the last one from 48 to 49.9.

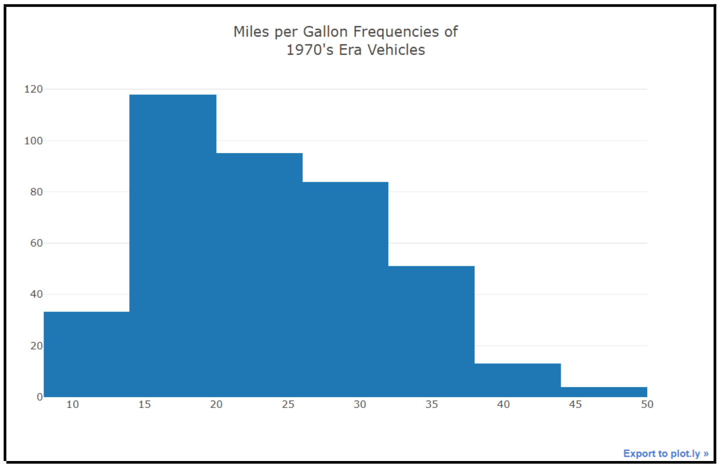
hist2.py sets bins wider, to 6. (since 50-8=42, seven equally spaced bins makes sense)

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Histogram(
    x=df['mpg'],
```

```
    xbins=dict(start=8,end=50,size=6),
)]
layout = go.Layout(
    title="Miles per Gallon Frequencies of<br>\
    1970's Era Vehicles"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='wide_histogram.html')
```



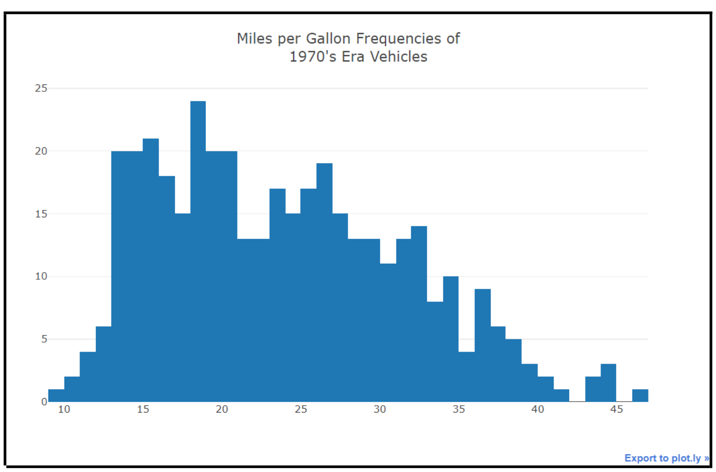
hist3.py sets bins to just 1:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Histogram(
    x=df['mpg'],
    xbins=dict(start=8,end=50,size=1),
)]

layout = go.Layout(
    title="Miles per Gallon Frequencies of<br>\
    1970's Era Vehicles"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='narrow_histogram.html')
```



- After comparing all three plots, it looks like plotly's default settings were a good choice for this dataset!

Our next example shows how to overlay two histograms, assign an opacity value, and compare two sets of data.

The data we'll use comes from a Cardiac Arrhythmia Database at https://archive.ics.uci.edu/ml/datasets/arrhythmia

We've stripped all but three columns and selected 420 records. The columns are 'Age', 'Sex' and 'Height'. For 'Sex', 0=male and 1=female, and height is measured in centimeters.

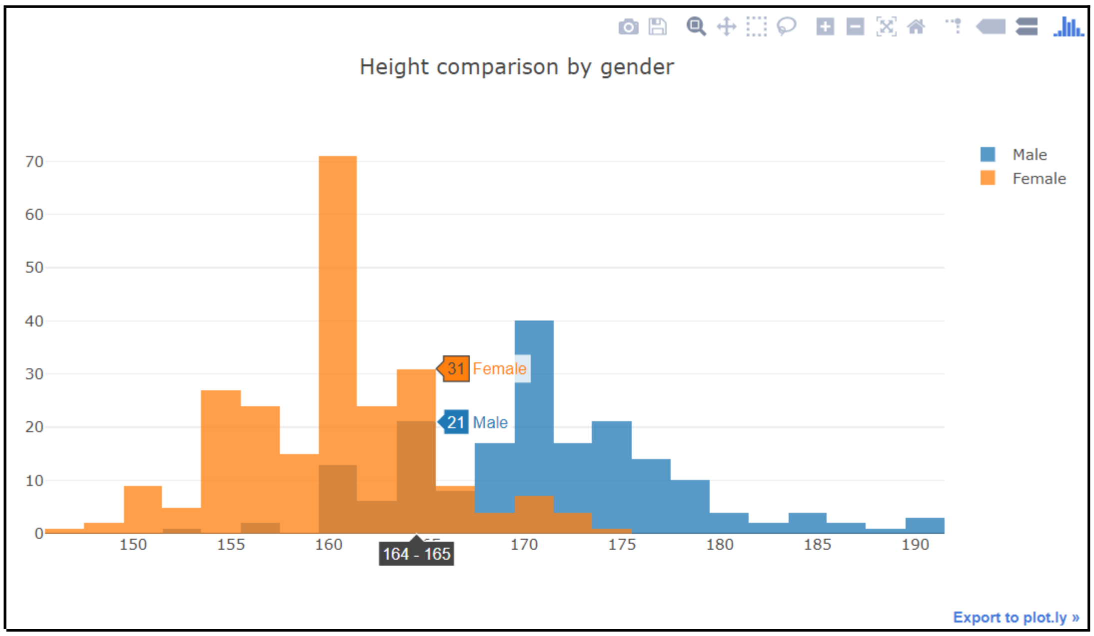
Create a file called **hist4.py** and add the following code:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/arrhythmia.csv')

data = [go.Histogram(
    x=df[df['Sex']==0]['Height'],
    opacity=0.75,
    name='Male'
),
go.Histogram(
    x=df[df['Sex']==1]['Height'],
    opacity=0.75,
    name='Female'
)]

layout = go.Layout(
    barmode='overlay',
    title="Height comparison by gender"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='basic_histogram2.html')
```



Now each trace has its own color, and opacity allows us to see each one independently.

For more information visit: https://plot.ly/python/histograms/ and https://plot.ly/python/reference/#histogram

Histograms - BONUS Example

What if the dataset itself contains frequency data? Histograms count the number of occurrences within one column. If you want to base your x-values on one column, but sum the *values* from another column, you need to use a bar chart. Let's try an example!

Fremont Bridge in Seattle, Washington has a pedestrian/bicycle path on either side. Cyclists on the eastern side generally travel north over the bridge, and south on the western side. The city installed sensors to record the number of bicycles that cross the bridge each day.

Images: http://sdotblog.seattle.gov/2016/02/25/how-does-that-bike-counter-work-at-the-fremont-bridge-and-who-named-fremont/

A nice time series dataset is available from https://data.seattle.gov/Transportation/Grouped-by-Hour/7mre-hcut, offering a 5+ year history (Oct-2012-Feb-2018) of the number of bikes that have crossed the bridge on either side each hour.

For this exercise, we'll take in a .csv file created from the source data, and then do some data wrangling:

- we want to change a text-based date column to datetime
- this lets us extract the time component into a separate column
- from this we build a new DataFrame from a groubpy of time, summing the number of bicycles in the eastern and western sides of the bridge

histBONUS.py performs these operations and plots the result:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/FremontBridgeBicycles.csv')

# Convert the "Date" text column to a Datetime series:
df['Date'] = pd.to_datetime(df['Date'])

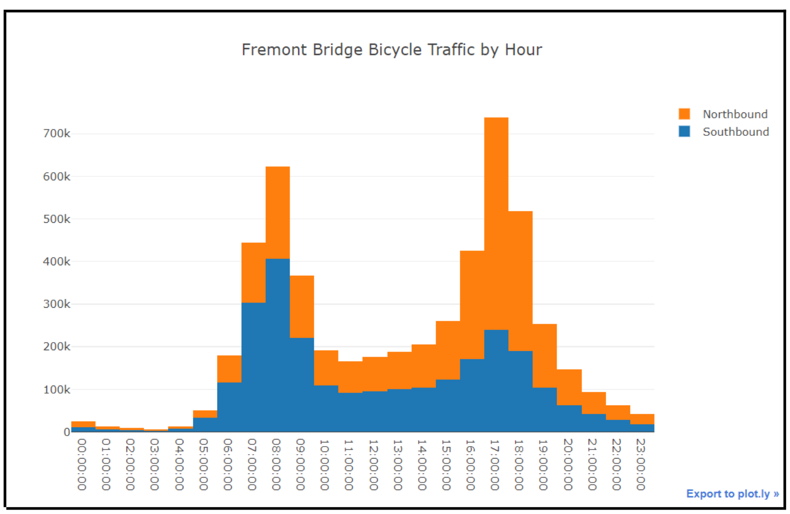
# Add a column to hold the hour:
df['Hour']=df['Date'].dt.time

# Let pandas perform the aggregation
df2 = df.groupby('Hour').sum()

trace1 = go.Bar(
    x=df2.index,
    y=df2['Fremont Bridge West Sidewalk'],
    name="Southbound",
    width=1  # eliminates space between adjacent bars
)
trace2 = go.Bar(
    x=df2.index,
    y=df2['Fremont Bridge East Sidewalk'],
    name="Northbound",
    width=1
)
data = [trace1, trace2]

layout = go.Layout(
    title='Fremont Bridge Bicycle Traffic by Hour',
    barmode='stack'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='fremont_bridge.html')
```

- The plot stacks two traces. It shows nicely how traffic Southbound is highest during the morning commute, while Northbound cyclists dominate the evening commute.
- We set the width to 1 so that adjacent bars touch, similar to a histogram.



Distplots

Distribution Plots, or Displots, typically layer three plots on top of one another. The first is a histogram, where each data point is placed inside a bin of similar values. The second is a rug plot - marks are placed along the x-axis for every data point, which lets you see the distribution of values inside each bin. Lastly, Displots often include a "kernel density estimate", or KDE line that tries to describes the shape of the distribution.

Because KDEs use computations to derive the shape of the line - using too large a bandwidth gives a line without enough detail, and too small a bandwidth can yield an unhelpful, jagged line - we say that we *plot* a histogram, but we *fit* a KDE line to the plot.

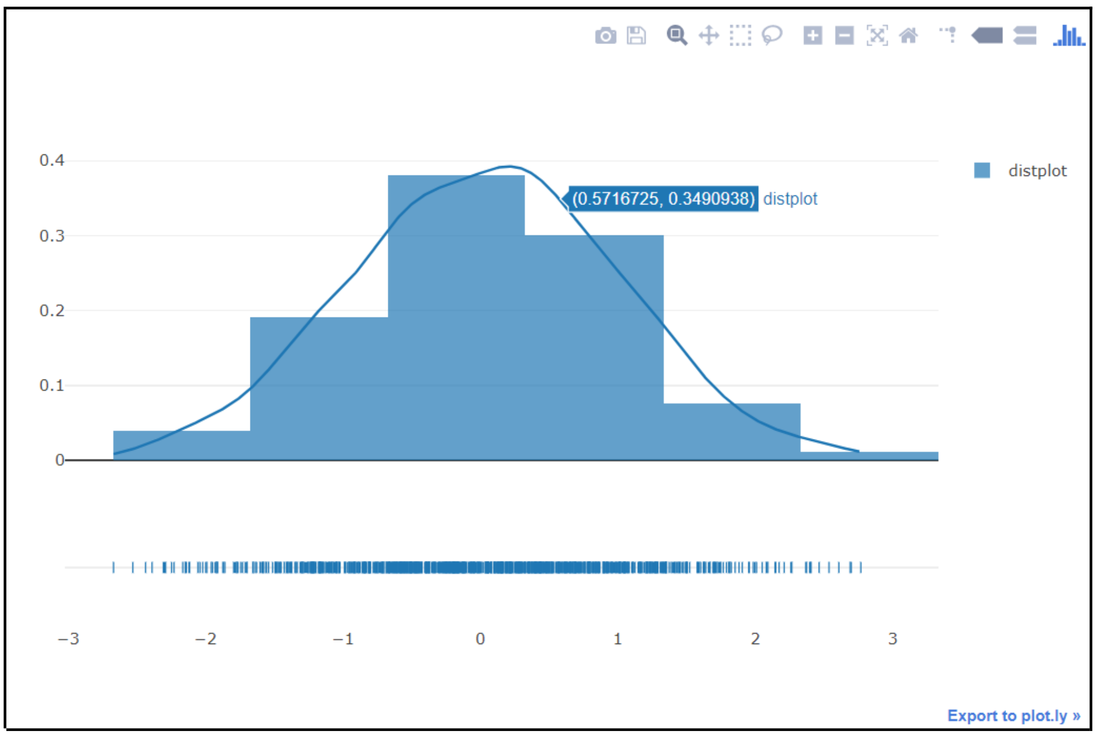
We obtain distplots from Plotly's **Figure Factory** module in place of Graph Objects.

dist1.py shows a basic distplot formed from 1000 random values:

```
import plotly.offline as pyo
import plotly.figure_factory as ff
import numpy as np

x = np.random.randn(1000)
hist_data = [x]
group_labels = ['distplot']

fig = ff.create_distplot(hist_data, group_labels)
pyo.plot(fig, filename='basic_distplot.html')
```



- Note that distplots display *relative* frequencies, not actual ones. The total area under the plot is equal to 1.
- By convention we use the label `hist_data` in place of `data`, as a reminder that this forms the histogram portion of the plot.

A random number generator will never show a perfectly normal (Gaussian) distribution - but the higher the number of data points, the closer you'll get. To demonstrate this we'll plot four relatively small samples side-by-side.

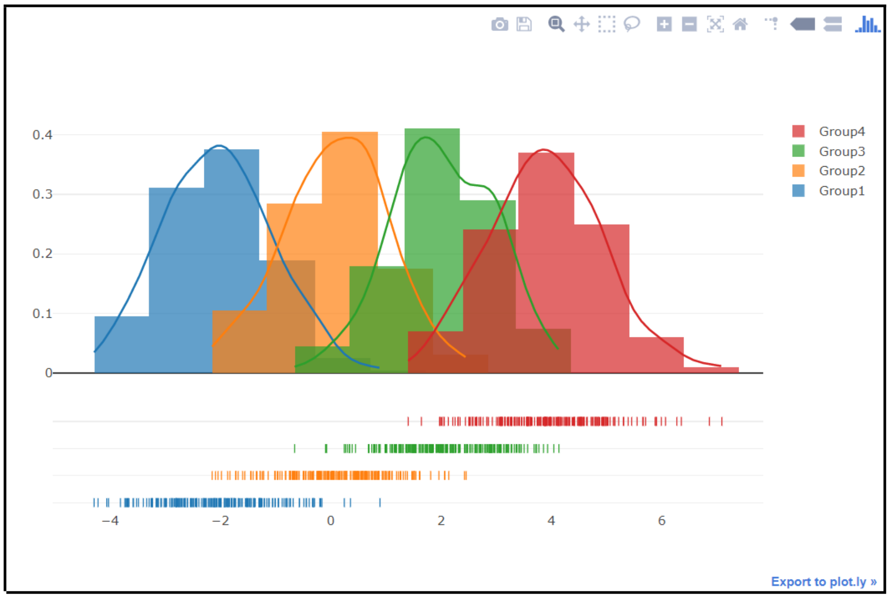
dist2.py compares four similar plots, each drawn from a different set of 200 random numbers:

```
import plotly.offline as pyo
import plotly.figure_factory as ff
import numpy as np

x1 = np.random.randn(200)-2
x2 = np.random.randn(200)
x3 = np.random.randn(200)+2
x4 = np.random.randn(200)+4

hist_data = [x1,x2,x3,x4]
group_labels = ['Group1','Group2','Group3','Group4']

fig = ff.create_distplot(hist_data, group_labels)
pyo.plot(fig, filename='multiset_distplot.html')
```



- A normal distribution would show an even, symmetric bell curve. These generally do not.

Distplots are not very informative on small datasets.

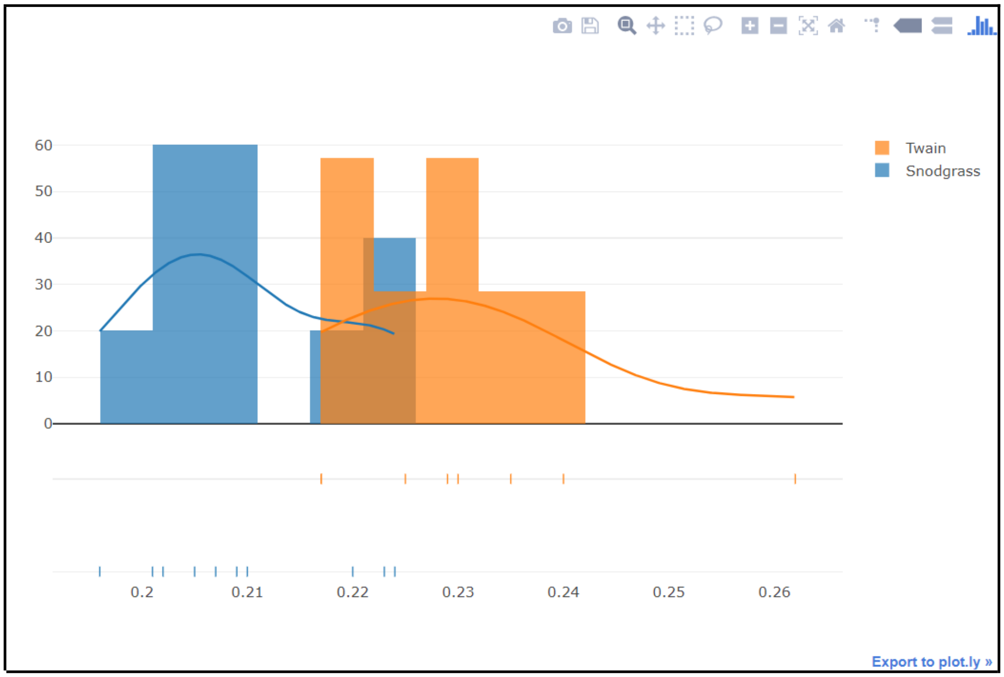
dist3.py goes back to our Mark Twain example, and plots two groups of only 10 and 8 points, respectively.

```
import plotly.offline as pyo
import plotly.figure_factory as ff

snodgrass = [.209,.205,.196,.210,.202,.207,.224,.223,.220,.201]
twain = [.225,.262,.217,.240,.230,.229,.235,.217]

hist_data = [snodgrass,twain]
group_labels = ['Snodgrass','Twain']

fig = ff.create_distplot(hist_data, group_labels, bin_size=[.005,.005])
pyo.plot(fig, filename='SnodgrassTwainDistplot.html')
```



- We set the bin_size to .005, and the results are confusing at best.
- Box plots were clearly a better choice here!

Resources: https://plot.ly/python/distplot/ and https://seaborn.pydata.org/tutorial/distributions.html

Heatmaps

In their simplest forms, Bar Charts, Box Plots, Histograms and Distplots help visualize “univariate distributions” - that is, the frequency of only one variable across a range of values or categories.

Heatmaps provide a “multivariate” plot by adding a third dimension - color - to the marker. This is somewhat similar to changing the size of the marker in our bubble plots.

For these examples we take temperature data for the same one-week period in 2010 from three US weather stations: Santa Barbara, California, Yuma, Arizona, and Sitka, Alaska. The raw data was obtained from the U.S. Climate Reference Network (USCRN) website, specifically https://www1.ncdc.noaa.gov/pub/data/uscrn/products/hourly02/2010/ .

From this we whittled down the data to three columns (date, time, avg temp), added a column for “day”, and removed all but one week’s worth of recordings (June 1st - 7th). The resulting files are SantaBarbaraCA.csv, YumaAZ.csv and SitkaAK.csv.

For starters, let’s create basic heatmaps for each dataset (these are, **heat2.py** and **heat3.py**) and accept plotly’s default parameters. Note that temperatures are given in degrees Celsius.

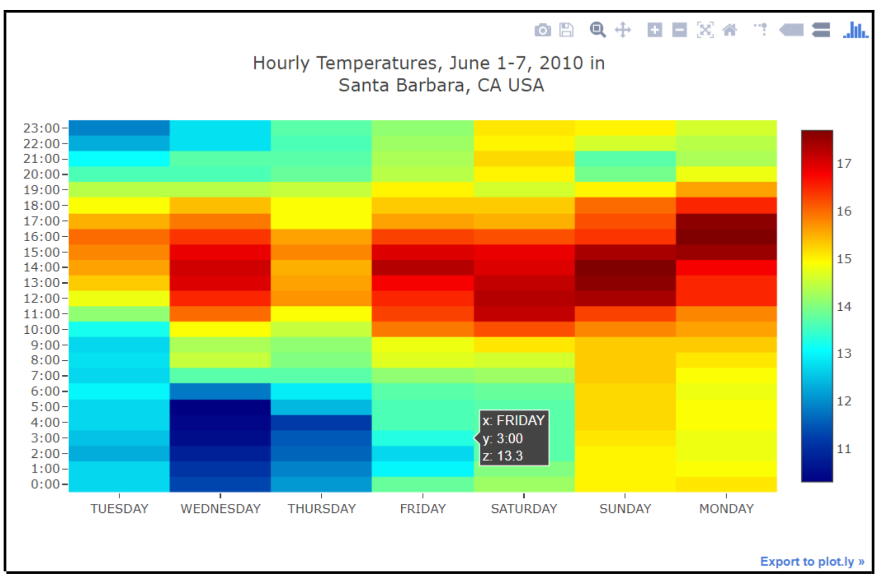
heat1.py creates a heatmap from SantaBarbaraCA.csv:

```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

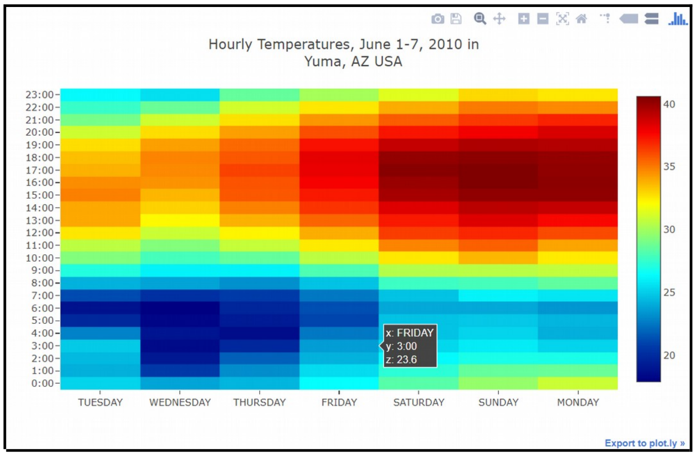
df = pd.read_csv('../data/2010SantaBarbaraCA.csv')

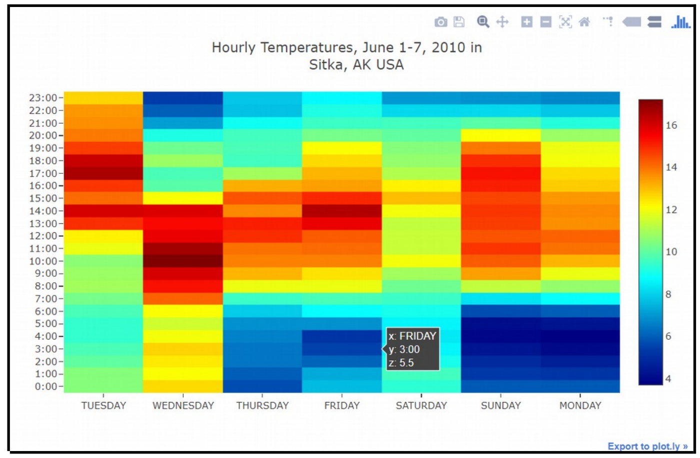
data = [go.Heatmap(
    x=df['DAY'],
    y=df['LST_TIME'],
    z=df['T_HR_AVG'].values.tolist(),
    colorscale='Jet'
)]

layout = go.Layout(
    title='Hourly Temperatures, June 1-7, 2010 in<br>\
    Santa Barbara, CA USA'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='Santa_Barbara.html')
```



heat2.py and **heat3.py** create similar heatmaps from YumaAZ.csv and SitkaAK.csv respectively:





- Although all three heatmaps appear fairly similar (warm during the day, cold at night), the temperature ranges are quite different for each one.

For **heat4.py**, several things happen:

i. We import plotly's *tools* module to create a figure with subplots.
ii. We add *zmin* and *zmax* values to each trace.
Looking at the raw data, the lowest temperature recorded in Sitka, Alaska was 3.7°C, and the highest recorded temperature in Yuma, Arizona was 40.6°C (that's 105° Fahrenheit!).
Using this range we set the min to 5 and the max to 40.
iii. In the subplot section we set `shared_yaxes = True`. This makes the hour markers appear on the left side only, and not beside each plot.
iv. Lastly, instead of combining *data* and *layout* into a Figure as we've done in the past, we access the layout directly using `fig['layout'].update()`.

heat4.py

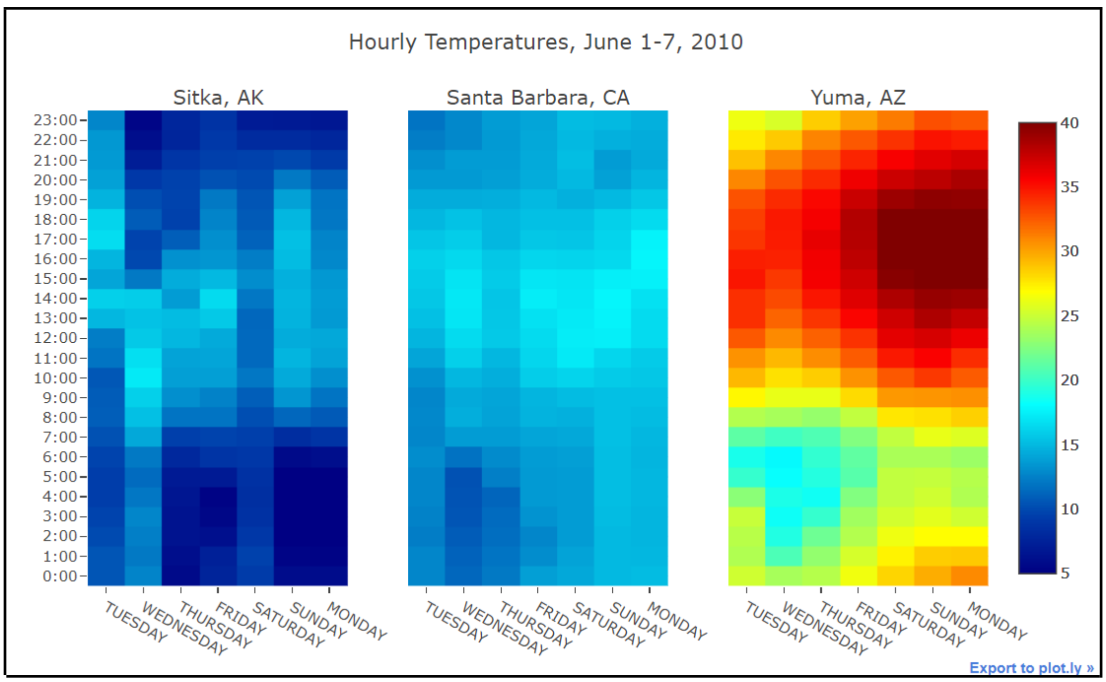
```
import plotly.offline as pyo
import plotly.graph_objs as go
from plotly import tools
import pandas as pd

df1 = pd.read_csv('../data/2010SitkaAK.csv')
df2 = pd.read_csv('../data/2010SantaBarbaraCA.csv')
df3 = pd.read_csv('../data/2010YumaAZ.csv')

trace1 = go.Heatmap(
    x=df1['DAY'],
    y=df1['LST_TIME'],
    z=df1['T_HR_AVG'].values.tolist(),
    colorscale='Jet',
    zmin = 5, zmax = 40 # add max/min color values to make each plot consistent
)
trace2 = go.Heatmap(
    x=df2['DAY'],
    y=df2['LST_TIME'],
    z=df2['T_HR_AVG'].values.tolist(),
    colorscale='Jet',
    zmin = 5, zmax = 40
)
trace3 = go.Heatmap(
    x=df3['DAY'],
    y=df3['LST_TIME'],
    z=df3['T_HR_AVG'].values.tolist(),
    colorscale='Jet',
    zmin = 5, zmax = 40
)

fig = tools.make_subplots(rows=1, cols=3,
    subplot_titles=('Sitka, AK','Santa Barbara, CA', 'Yuma, AZ'),
    shared_yaxes = True,  # this makes the hours appear only on the left
)
fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)
fig.append_trace(trace3, 1, 3)

fig['layout'].update(      # access the layout directly!
    title='Hourly Temperatures, June 1-7, 2010'
)
pyo.plot(fig, filename='AllThree.html')
```



With this final plot we see data from three different regions, using the same scale, side-by-side for comparison. Not bad!

Resources: https://plot.ly/python/heatmaps/

Data source: https://www1.ncdc.noaa.gov/pub/data/uscrn/products/hourly02/2010/

Exercises: Plotly Basics

Ex1-Scatterplot.py

Objective: Create a scatterplot of 1000 random data points.
x-axis values should come from a normal distribution using np.random.randn(1000).
y-axis values should come from a uniform distribution over [0,1) using np.random.rand(1000)

Ex2-Linechart.py

Objective: Using the file 2010YumaAZ.csv, develop a Line Chart that plots seven days worth of temperature data on one graph. You can use a list comprehension to assign each day to its own trace.

See https://pandas.pydata.org/pandas-docs/stable/generated/pandas.unique.html
for help on finding unique values with pandas

Ex3-Barchart.py

Objective: Create a stacked bar chart from the file ../data/mocksurvey.csv. Note that questions appear in the index (and should be used for the x-axis), while responses appear as column labels.
Extra Credit: make a horizontal bar chart!

See https://plot.ly/python/horizontal-bar-charts/ for extra credit help.

Ex4-Bubblechart.py

Objective: Create a bubble chart that compares three other features from the mpg.csv dataset.
Fields include: 'mpg', 'cylinders', 'displacement' 'horsepower', 'weight', 'acceleration', 'model_year', 'origin', 'name'

Ex5-Boxplot.py

Objective: Make a DataFrame sing the Abalone dataset (../data/abalone.csv). Take two independent random samples of different sizes from the 'rings' field. HINT: np.random.choice(df['rings'],10,replace=False) takes 10 random values
Use box plots to show that the samples do derive from the same population.

Ex6-Histogram.py

Objective: Create a histogram that plots the 'length' field from the Abalone dataset (../data/abalone.csv).
Set the range from 0 to 1, with a bin size of 0.02

Ex7-Distplot.py

Objective: Using the iris dataset, develop a Distplot that compares the petal lengths of each class.
File: '../data/iris.csv'
Fields: 'sepal_length','sepal_width','petal_length','petal_width','class'
Classes: 'Iris-setosa','Iris-versicolor','Iris-virginica'

Ex8-Heatmap.py

Objective: Using the "flights" dataset available from Python's Seaborn module (see https://seaborn.pydata.org/generated/seaborn.heatmap.html) create a heatmap with the following parameters:
x-axis="year"
y-axis="month"
z-axis(color)="passengers"

Plotly Basics Exercise Solutions

Refer to the Code section for exercise solutions

Dash Basics - Layout

Introduction to Dash Basics

If you haven't already done so, follow the installation instructions in Lecture 4.
As a quick review, the Dash installation steps are:

```
pip install dash==0.21.0                    # The core dash backend
pip install dash-renderer==0.11.3           # The dash front-end
pip install dash-html-components==0.9.0     # HTML components
pip install dash-core-components==0.21.2    # Supercharged components
pip install plotly --upgrade          # Plotly graphing library used in examples
```

Dash apps are composed of two parts. The first part is the **layout** of the app and it describes what the application looks like. The second part describes the **interactivity** of the application.

The good news is that you don't need to know any HTML or CSS to use Dash. Most html tags are provided as Python classes. For example, typing **html.H1(children='Hello Dash')** into your Dash script results in the HTML element **<h1>Hello Dash</h1>**.

Dash offers two distinct component libraries. The code above comes from the **dash_html_components** library which has a component for every HTML tag, like the first-level heading **H1**. Another library, **dash_core_components**, offers higher-level, interactive components that are generated with JavaScript, HTML, and CSS through the React.js library.

Dash components - be they html or core - are described entirely through keyword attributes. Dash is *declarative*: you will primarily describe your application through these attributes.

Dash Layout

Let's create a simple HTML page that displays a bar chart. Create a file called **layout1.py** and enter the following:

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()

app.layout = html.Div(children=[
    html.H1(children='Hello Dash'),
    html.Div(children='Dash: A web application framework for Python.'),

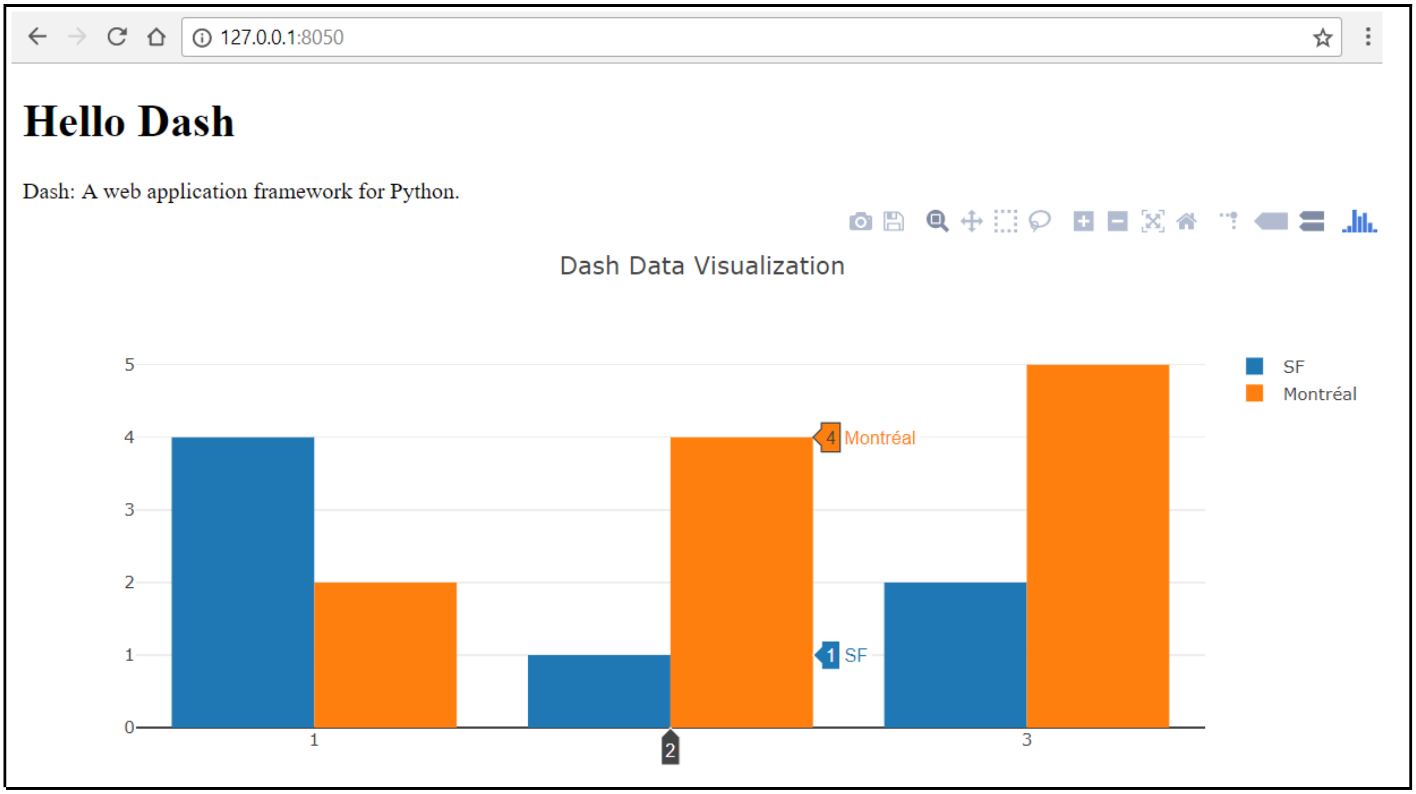
    dcc.Graph(
        id='example-graph',
        figure={
            'data': [
                {'x': [1, 2, 3], 'y': [4, 1, 2], 'type': 'bar', 'name': 'SF'},
                {'x': [1, 2, 3], 'y': [2, 4, 5], 'type': 'bar', 'name': u'Montréal'},
            ],
            'layout': {
                'title': 'Dash Data Visualization'
            }
        }
    )
])

if __name__ == '__main__':
    app.run_server()
```

Run the app with

```
$ python layout1.py
...Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
```

and visit http://127.0.0.1:8050/ in your web browser. You should see a page that looks like this:



Note: the interactive portions only appear when your cursor hovers over a bar.

TROUBLESHOOTING: Some text editors do not properly encode utf. If you receive an error message that states

```
File "app.py", line 17
SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xe9 in position 5: invalid continuation byte
```

the problem is likely with the extended Unicode character in `u'Montréal'`. Change this to a regular `e` instead. Save the file and try running it again as shown above.

The steps we took in this quick example are as follows:

1. `# -*- coding: utf-8 -*-`
   This specifies the encoding for the Python file. See [PEP 0263 - Defining Python Source Code Encodings]() for details.

2. ```
   import dash
   import dash_core_components as dcc
   import dash_html_components as html
   ```
   We import Dash and both of its component libraries.

3. `app = dash.Dash()`
   We launch a Dash application. “app” is just a convenient name for our Dash instance.
   ```

4. `app.layout = html.Div(children=[`
   ```
       html.H1(children='Hello Dash'),
       html.Div(children='Dash: A web application framework for Python.'),
   ```
   Here we start to define the application layout.
   `H1` and `Div` are component attributes that map to corresponding HTML tags.
   `H1` we've seen; it creates a level one heading. `Div` creates a `<div>` tag which is like an HTML container.
   `children` is a property of HTML components (we'll use this keyword later when we add interactivity to our dashboards). By default this is the first property listed, so we don't really need to add `children=` to our code.

5. `dcc.Graph(`
   ```
       id='example-graph',
       figure={
           'data': [
               {'x': [1, 2, 3], 'y': [4, 1, 2], 'type': 'bar', 'name': 'SF'},
               {'x': [1, 2, 3], 'y': [2, 4, 5], 'type': 'bar', 'name': u'Montréal'},
           ],
           'layout': {
               'title': 'Dash Data Visualization'
   ```
   This is all one core component!
   The 'data' and 'layout' keyword attributes should look familiar as they're taken directly from Plotly.
   Graph components have a `figure` property in place of `children`. This is the same figure used in Plotly.

6. `if __name__ == '__main__':`
   ```
       app.run_server()
   ```
   This last section launches a local server *only if* **layout1.py** is run as a script.
   If we import this file into another program, this line of code is ignored.

It's important to note that, unlike Plotly, **layout1.py** is an active script that requires a local web server running in the background. If you should make changes to **layout1.py** that prevent it from running properly then the terminal will display an error and shut down the server.

Dash uses Flask as its server back end. You can pass **debug=True** into the server call to enable some diagnostic services (you wouldn't want to do this in production!). The code would look like this:

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

Before we move on, let's make some changes to the HTML on our page. Open a new file called **layout2.py** and copy the contents of layout1 to layout2. (You can simply duplicate **layout1.py** if you want).

Next, insert the code shown below in black (original code is shown in blue):

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()

colors = {
    'background': '#111111',
    'text': '#7FDBFF'
}

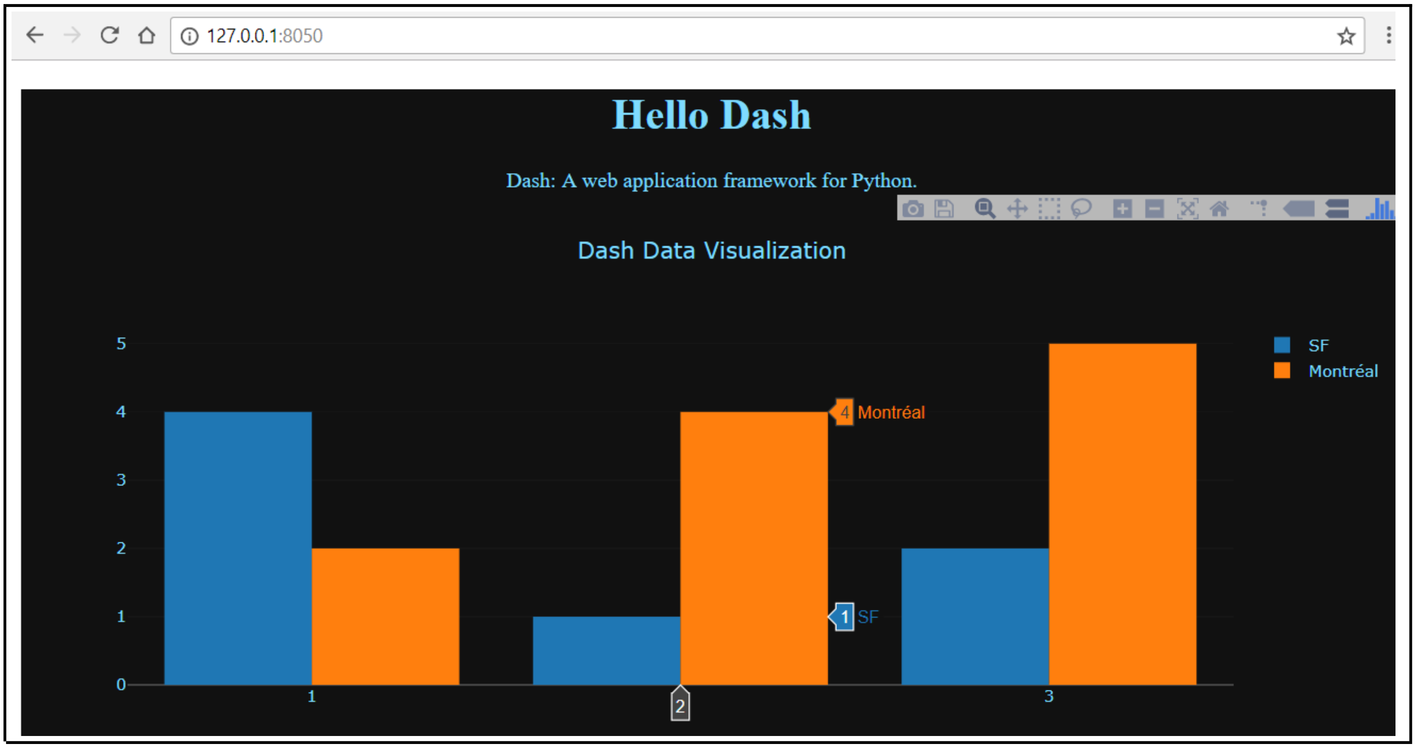
app.layout = html.Div(children=[
    html.H1(
        children='Hello Dash',
        style={
            'textAlign': 'center',
            'color': colors['text']
        }
    ),

    html.Div(
        children='Dash: A web application framework for Python.',
        style={
            'textAlign': 'center',
            'color': colors['text']
        }
    ),

    dcc.Graph(
        id='example-graph',
        figure={
            'data': [
                {'x': [1, 2, 3], 'y': [4, 1, 2], 'type': 'bar', 'name': 'SF'},
                {'x': [1, 2, 3], 'y': [2, 4, 5], 'type': 'bar', 'name': u'Montréal'},
            ],
            'layout': {
                'plot_bgcolor': colors['background'],
                'paper_bgcolor': colors['background'],
                'font': {
                    'color': colors['text']
                },
                'title': 'Dash Data Visualization'
            }
        }
    )],
    style={'backgroundColor': colors['background']}
)

if __name__ == '__main__':
    app.run_server()
```

In this version we add a dictionary of color styles. These are referenced in the style properties added to each component. Run **`python layout2.py`** in the terminal and you should see this page:



In this example, we modified the inline styles of the **html.Div** and **html.H1** components with the **style** property. `html.H1('Hello Dash', style={'textAlign':'center', 'color':'#7FDFF'})` is rendered in the Dash application as `<h1 style="text-align:center; color:#7FDFF">Hello Dash</h1>`.

There are a few important differences between the **dash_html_components** and the HTML attributes:

1. The **style** property in HTML is a semicolon-separated string. In Dash, you can just supply a dictionary.
2. The keys in the **style** dictionary are camelCased. So, instead of **text-align**, it's **textAlign**.
3. The HTML **class** attribute is **className** in Dash. We'll see this in upcoming examples.
4. The children of an HTML tag are specified through the **children** keyword argument.
   By convention, this is always the *first* argument and so it is often omitted.
   `html.H1(children='Hello Dash')` is the same as `html.H1('Hello Dash')`.

That's it! You've just created your first dashboard! Up next, we'll convert a simple Plotly plot to Dash.

Converting Simple Plotly Plot to Dashboard with Dash

For this exercise we'll go back to our scatter3.py script. This involved a scatter plot of 100 random data points. We'll seed the random number generator so that everyone sees the same result.

Open a new file, and name it **plotly1.py**. Enter the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import plotly.graph_objs as go
import numpy as np

app = dash.Dash()

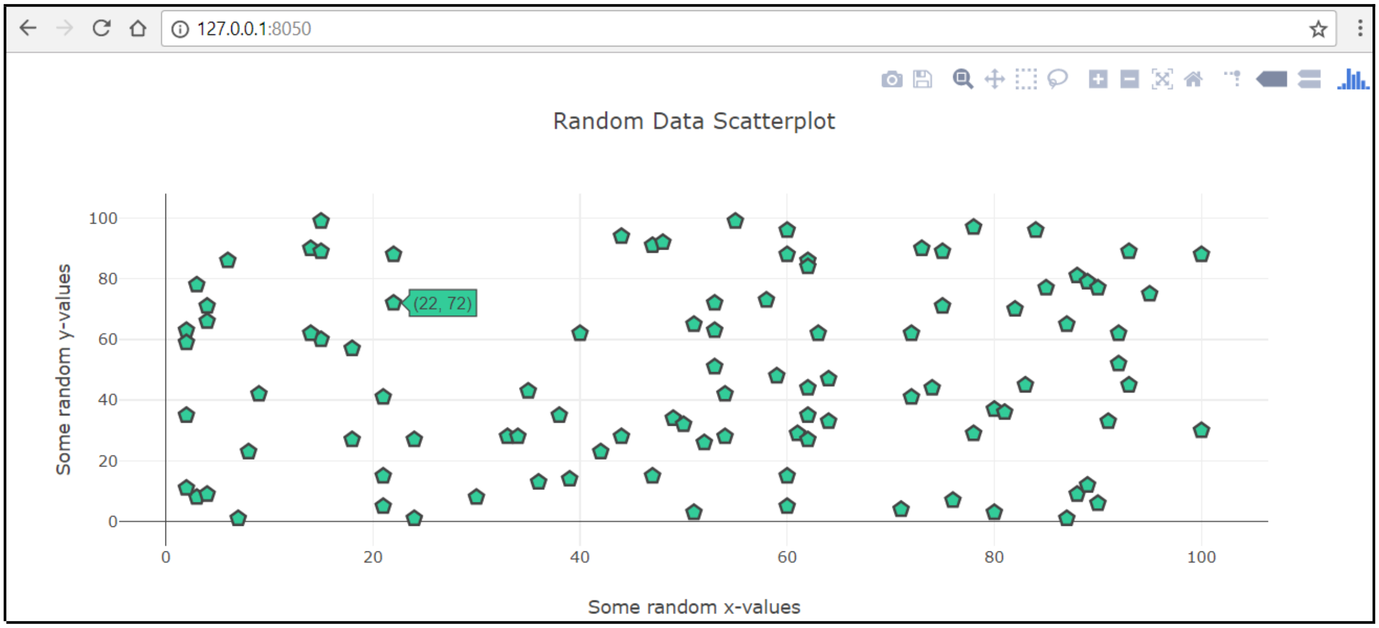
np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

app.layout = html.Div([
    dcc.Graph(
        id='scatter3',
        figure={
            'data': [
                go.Scatter(
                    x = random_x,
                    y = random_y,
                    mode = 'markers',
                    marker = {
                        'size': 12,
                        'color': 'rgb(51,204,153)',
                        'symbol': 'pentagon',
                        'line': {'width': 2}
                        }
                )
            ],
            'layout': go.Layout(
                title = 'Random Data Scatterplot',
                xaxis = {'title': 'Some random x-values'},
                yaxis = {'title': 'Some random y-values'},
                hovermode='closest'
            )
        }
    )
])

if __name__ == '__main__':
    app.run_server()
```

As you can see, much of this is the same code as was used in **scatter3.py**. In Dash, the **dash_core_components** library includes a component called **Graph**, which renders interactive data visualizations using Plotly's JavaScript graphing library. In fact, the **figure** argument in the **dcc.Graph** component is the same **figure** argument that is used by Plotly.

Once you have saved the file, run **python plotly1.py** in the terminal. Open your browser again to http://127.0.0.1:8050/ and the following page should appear:



Exercise: Create a Simple Dashboard

For this exercise we will create a dashboard similar to the scatterplot above, only this time we'll import some data.

Old Faithful is a cone geyser located in Yellowstone National Park in Wyoming, United States. Since 2000 its intervals have varied from 44 to 125 minutes between eruptions, with an average of about 90-92 minutes.

It is not possible to predict more than one eruption in advance. Old Faithful is currently bimodal. It has two eruption durations, either long (over 4 minutes) or more rarely short (about 2-1/2 minutes). Short eruptions lead to an interval of just over an hour and long eruptions lead to an interval of about 1-1/2 hours.

For this exercise, build a dashboard that imports **OldFaithful.csv** from the data directory, and displays a scatterplot.

The dataset is comprised of 3 fields (D,Y,X) where
D = date of recordings in month (in August),
X = duration of the current eruption in minutes (to nearest 0.1 minute),
Y = waiting time until the next eruption in minutes (to nearest minute).

Image: Eruption of Old Faithful in 1948
https://en.wikipedia.org/wiki/Old_Faithful#/media/File:OldFaithful1948.jpg

Simple Dashboard Exercise Solution

This is our suggested solution:

```
# Perform imports here:
import dash
import dash_core_components as dcc
import dash_html_components as html
import plotly.graph_objs as go
import pandas as pd

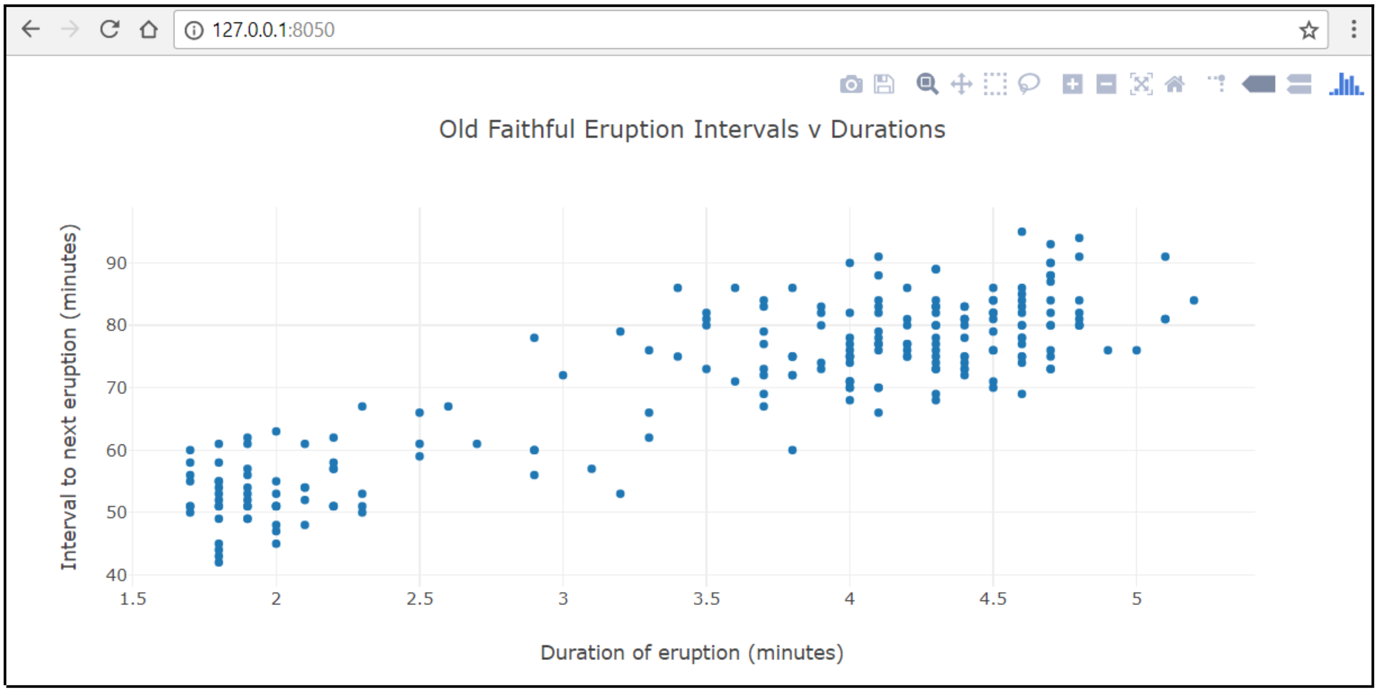
# Launch the application:
app = dash.Dash()

# Create a DataFrame from the .csv file:
df = pd.read_csv('../data/OldFaithful.csv')

# Create a Dash layout that contains a Graph component:
app.layout = html.Div([
    dcc.Graph(
        id='old_faithful',
        figure={
            'data': [
                go.Scatter(
                    x = df['X'],
                    y = df['Y'],
                    mode = 'markers'
                )
            ],
            'layout': go.Layout(
                title = 'Old Faithful Eruption Intervals v Durations',
                xaxis = {'title': 'Duration of eruption (minutes)'},
                yaxis = {'title': 'Interval to next eruption (minutes)'},
                hovermode='closest'
            )
        }
    )
])

# Add the server clause:
if __name__ == '__main__':
    app.run_server()
```

If all goes well, your finished dashboard should open a page like this:



Which shows a clear correlation between an eruption's duration and the expected wait to the next eruption!

Dash Components

Dash components are provided by two libraries: **dash_html_components** which we usually abbreviate to **html**, and **dash_core_components**, usually abbreviated to **dcc**. Normally, **html** components describe the layout of the page, including placement and alignment of different graphs. **dcc** components describe the individual graphs themselves.

HTML Components

For a description of Dash's HTML components, visit https://dash.plot.ly/dash-html-components

Common components include:

● **html.Div**([section])	applies CSS to section of page	
● **html.P**()	paragraph	
● **html.H1**('text')	heading (level 1)	
● **html.Label**('text')	label	<label> </label>

HTML elements and Dash classes are mostly the same but there are a few key differences:

- The **style** property is a dictionary
- Properties in the style dictionary are camelCased
- The *class* key is renamed as **className**
- Style properties in pixel units can be supplied as just numbers without the *px* unit

Let's take a look at an example.

```
import dash_html_components as html
html.Div([
    html.Div('Example Div', style={'color': 'blue', 'fontSize': 14}),
    html.P('Example P', className='my-class', id='my-p-element')
], style={'marginBottom': 50, 'marginTop': 25})
```

That dash code will render this HTML markup:

```
<div style="margin-bottom: 50px; margin-top: 25px;">

    <div style="color: blue; font-size: 14px">
        Example Div
    </div>

    <p class="my-class", id="my-p-element">
        Example P
    </p>

</div>
```

To provide an example of how **dash_html_components** can be laid out on a page, create a file called **HTMLComponents.py** and add the following code:

```
import dash
import dash_html_components as html

app = dash.Dash()

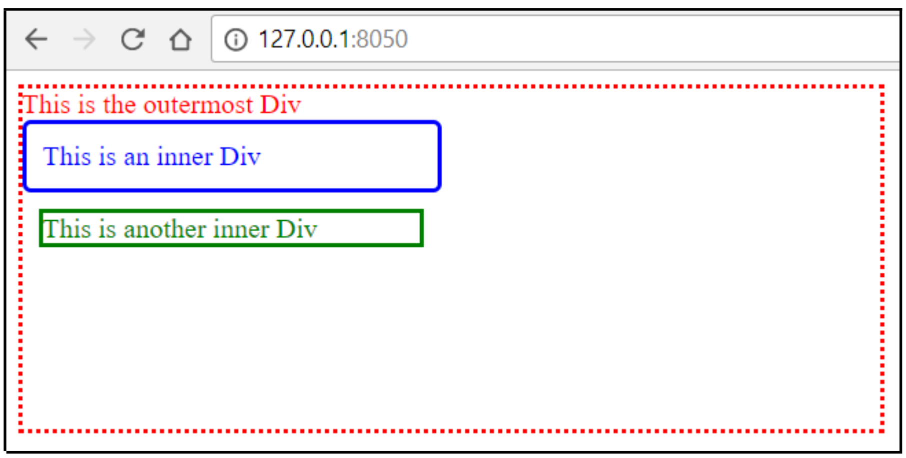
app.layout = html.Div([
    'This is the outermost Div',
    html.Div(
        'This is an inner Div',
        style={'color':'blue', 'border':'2px blue solid', 'borderRadius':5,
        'padding':10, 'width':220}
    ),
    html.Div(
        'This is another inner Div',
        style={'color':'green', 'border':'2px green solid',
        'margin':10, 'width':220}
    ),
],
# this styles the outermost Div:
style={'width':500, 'height':200, 'color':'red', 'border':'2px red dotted'})

if __name__ == '__main__':
    app.run_server()
```

Note that `'border':'2px blue solid'` is shorthand for
`'borderWidth':2, 'borderColor':'blue', 'borderStyle':'solid'`

`'borderRadius':5` needs to be listed separately.

Now run the script and open a browser to http://127.0.0.1:8050/ You should see something like this:



Note how style can be individually applied to each Div, providing color, borders, padding and margins.

Core Components

For a complete description of Dash's core components, visit https://dash.plot.ly/dash-core-components

Here we describe a few useful tools.

Create a file called **CoreComponents.py** and add the following code.
Keep this file handy - you may want to add components to it that you find useful!

CoreComponents.py

```
import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()

app.layout = html.Div([

    # DROPDOWN https://dash.plot.ly/dash-core-components/dropdown
    html.Label('Dropdown'),
    dcc.Dropdown(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': 'Montréal', 'value': 'MTL'},
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        value='MTL'
    ),

    html.Label('Multi-Select Dropdown'),
    dcc.Dropdown(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': u'Montréal', 'value': 'MTL'},
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        value=['MTL', 'SF'],
        multi=True
    ),

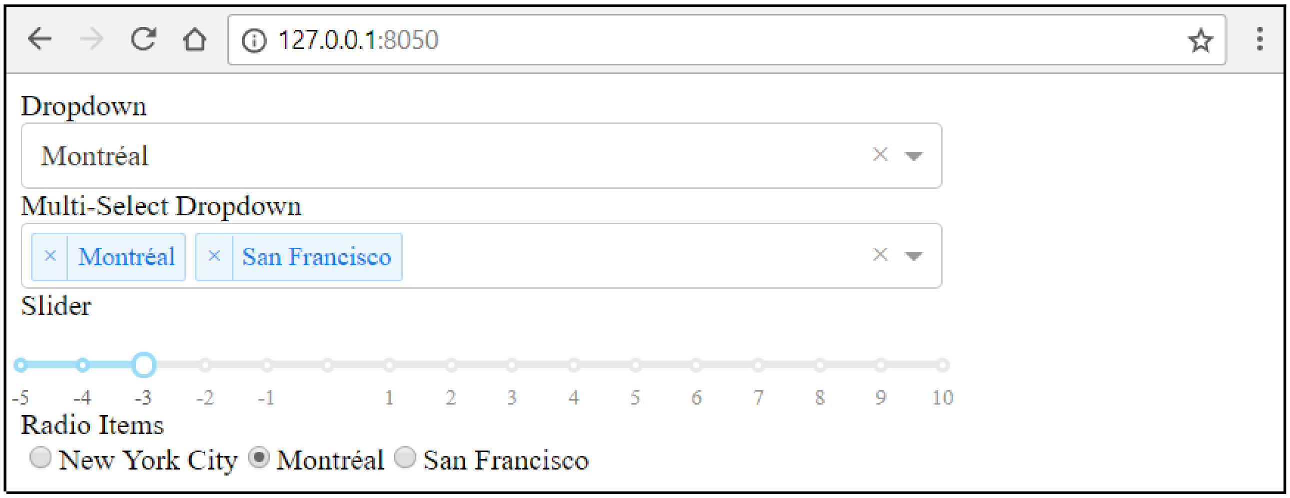
    # SLIDER https://dash.plot.ly/dash-core-components/slider
    html.Label('Slider'),
    html.P(
    dcc.Slider(
        min=-5,
        max=10,
        step=0.5,
        marks={i: i for i in range(-5,11)},
        value=-3
    )),

    # RADIO ITEMS https://dash.plot.ly/dash-core-components/radioitems
    html.Label('Radio Items'),
    dcc.RadioItems(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': 'Montréal', 'value': 'MTL'},
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        value='MTL'
    )
], style={'width': '50%'})

if __name__ == '__main__':
    app.run_server()
```

- We put the Slider inside an html paragraph **html.P()** to prevent the radio buttons beneath it from overwriting the slider marks.

Run this script and you should see the following:



Here we've shown only three built-in components:

- Dropdown
- Slider
- Radio Items

Many others are available, such as:

- RangeSlider
- Input
- Textarea
- Checklist (horizontal and vertical checkboxes)
- DatePickerSingle
- DatePickerRange

These components don't really do anything until we involve Dash's interactive features, like **callbacks**.

Before we get there, let's investigate the **Markdown** component (a shortcut for writing HTML text), and Dash's **Help()** method.

Markdown

While Dash exposes HTML through the **dash_html_components** library, it can be tedious to write your copy in HTML. For writing blocks of text, you can use the **Markdown** component in the **dash_core_components** library.

Create a file called **markdown.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash()

markdown_text = '''
### Dash and Markdown

Dash apps can be written in Markdown.
Dash uses the [CommonMark](http://commonmark.org/)specification of Markdown.

Check out their [60 Second Markdown Tutorial](http://commonmark.org/help/)
if this is your first introduction to Markdown!

Markdown includes syntax for things like **bold text** and *italics*,
[links](http://commonmark.org/help), inline `code` snippets, lists,
quotes, and more.
'''

app.layout = html.Div([
    dcc.Markdown(children=markdown_text)
])

if __name__ == '__main__':
    app.run_server()
```

Run the program at the terminal, open a browser to http://127.0.0.1:8050/ and you should see the following page:

Notice how in the code three hash marks ### translates to an <h3> tag on the page.

Notice also that the line break between “Dash apps can be written in Markdown.” and “Dash uses the [CommonMark](http://commonmark.org/) specification of Markdown.” is ignored.
To start a new paragraph on the page requires a blank line.

For more information, visit https://dash.plot.ly/dash-core-components/markdown

Using Help() with Dash

Dash components are declarative: every configurable aspect of these components is set during instantiation as a keyword argument. Call **help** in your Python console on any of the components to learn more about a component and its available arguments.

```
>>> import dash
>>> import dash_core_components as dcc
>>> help(dcc.Dropdown)
class Dropdown(dash.development.base_component.Component)
|   A Dropdown component.
|   Dropdown is an interactive dropdown element for selecting one or more
|   items.
|   The values and labels of the dropdown items are specified in the `options`
|   property and the selected item(s) are specified with the `value` property.
|
|   Use a dropdown when you have many options (more than 5) or when you are
|   constrained for space. Otherwise, you can use RadioItems or a Checklist,
|   which have the benefit of showing the users all of the items at once.
|
|   Keyword arguments:
|   - id (string; optional)
|   - className (string; optional)
|   - disabled (boolean; optional): If true, the option is disabled
-- More --
```

Hit <space> to see more content on this topic.

```
>>> import dash_html_components as html
>>> help(html.Div)
Help on class Div in module builtins:

class Div(dash.development.base_component.Component)
 |   A Div component.
 |
 |
 |   Keyword arguments:
 |   - children (optional): The children of this component
 |   - id (optional): The ID of this component, used to identify dash components
 |   in callbacks. The ID needs to be unique across all of the
 |   components in an app.
 |   - n_clicks (optional): An integer that represents the number of times
 |   that this element has been clicked on.
 |   - key (optional): A unique identifier for the component, used to improve
 |   performance by React.js while rendering components
 |   See https://reactjs.org/docs/lists-and-keys.html for more info
 |   - accessKey (optional): Defines a keyboard shortcut to activate or add focus to the element.
 |   - className (optional): Often used with CSS to style elements with common properties.
 |   - contentEditable (optional): Indicates whether the element's content is editable.
 |   - contextMenu (optional): Defines the ID of a <menu> element which will serve as the
-- More --
```

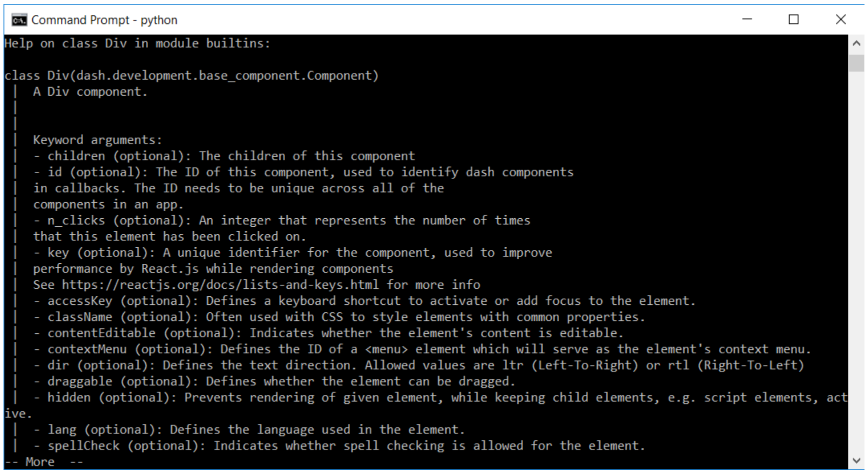
Hit <space> to see more content on this topic.

Writing Help() to HTML:

As an alternative to reading a plain-text help file in the console, you can write it out to an .html file using *pydoc*.

At the terminal (*not* in Python or any IDE) type **pydoc -w dash_html_components.Div**
This creates a file called **dash_html_components.Div.html** in the same directory, which can be viewed in the browser. This works for any Dash component!

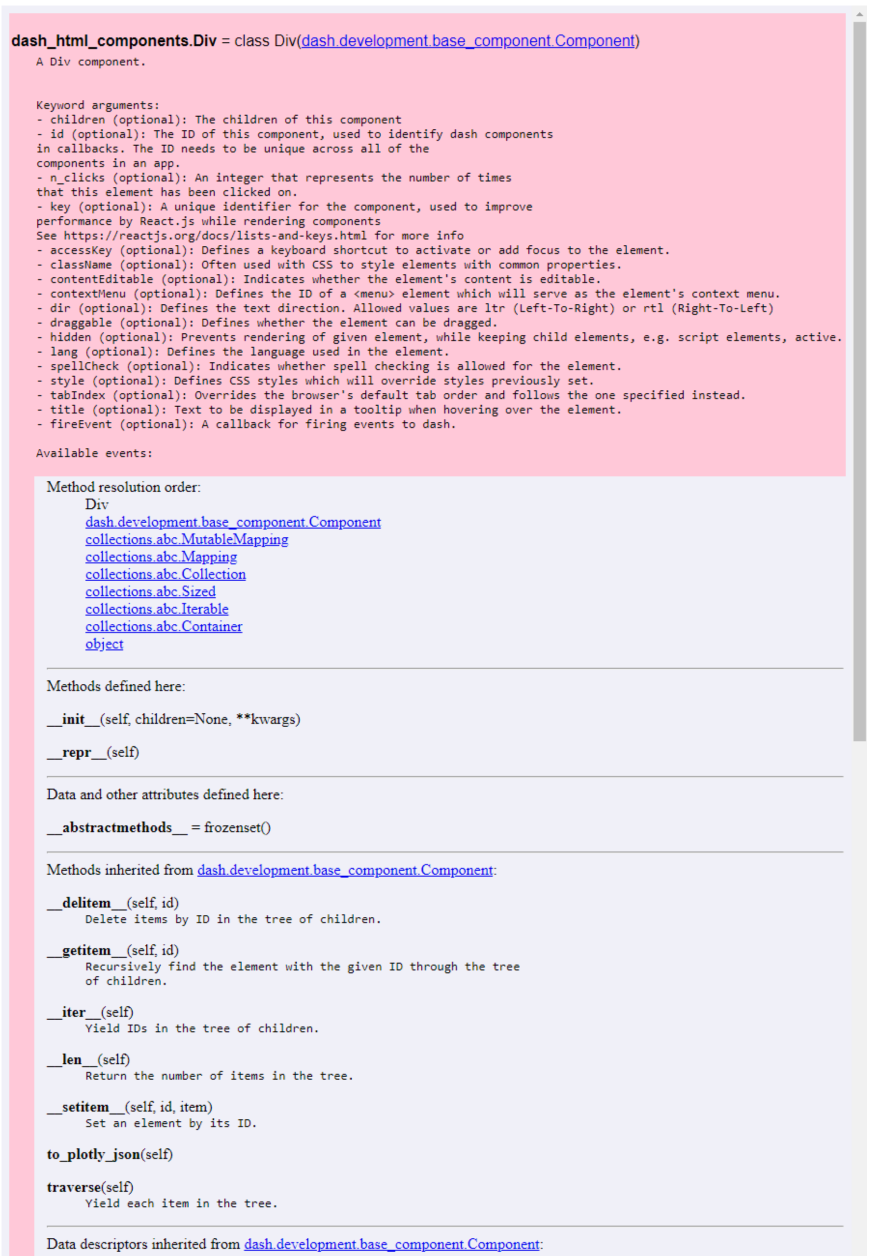
This is what help looks like in the Python console:



```
Command Prompt - python
Help on class Div in module builtins:

class Div(dash.development.base_component.Component)
 |  A Div component.
 |
 |
 |  Keyword arguments:
 |  - children (optional): The children of this component
 |  - id (optional): The ID of this component, used to identify dash components
 |  in callbacks. The ID needs to be unique across all of the
 |  components in an app.
 |  - n_clicks (optional): An integer that represents the number of times
 |  that this element has been clicked on.
 |  - key (optional): A unique identifier for the component, used to improve
 |  performance by React.js while rendering components
 |  See https://reactjs.org/docs/lists-and-keys.html for more info
 |  - accessKey (optional): Defines a keyboard shortcut to activate or add focus to the element.
 |  - className (optional): Often used with CSS to style elements with common properties.
 |  - contentEditable (optional): Indicates whether the element's content is editable.
 |  - contextMenu (optional): Defines the ID of a <menu> element which will serve as the element's context menu.
 |  - dir (optional): Defines the text direction. Allowed values are ltr (Left-To-Right) or rtl (Right-To-Left)
 |  - draggable (optional): Defines whether the element can be dragged.
 |  - hidden (optional): Prevents rendering of given element, while keeping child elements, e.g. script elements, act
ive.
 |  - lang (optional): Defines the language used in the element.
 |  - spellCheck (optional): Indicates whether spell checking is allowed for the element.
-- More  --
```

This is the same file seen in the browser:



```
dash_html_components.Div = class Div(dash.development.base_component.Component)
    A Div component.


    Keyword arguments:
    - children (optional): The children of this component
    - id (optional): The ID of this component, used to identify dash components
    in callbacks. The ID needs to be unique across all of the
    components in an app.
    - n_clicks (optional): An integer that represents the number of times
    that this element has been clicked on.
    - key (optional): A unique identifier for the component, used to improve
    performance by React.js while rendering components
    See https://reactjs.org/docs/lists-and-keys.html for more info
    - accessKey (optional): Defines a keyboard shortcut to activate or add focus to the element.
    - className (optional): Often used with CSS to style elements with common properties.
    - contentEditable (optional): Indicates whether the element's content is editable.
    - contextMenu (optional): Defines the ID of a <menu> element which will serve as the element's context menu.
    - dir (optional): Defines the text direction. Allowed values are ltr (Left-To-Right) or rtl (Right-To-Left)
    - draggable (optional): Defines whether the element can be dragged.
    - hidden (optional): Prevents rendering of given element, while keeping child elements, e.g. script elements, active.
    - lang (optional): Defines the language used in the element.
    - spellCheck (optional): Indicates whether spell checking is allowed for the element.
    - style (optional): Defines CSS styles which will override styles previously set.
    - tabIndex (optional): Overrides the browser's default tab order and follows the one specified instead.
    - title (optional): Text to be displayed in a tooltip when hovering over the element.
    - fireEvent (optional): A callback for firing events to dash.

    Available events:

    Method resolution order:
        Div
        dash.development.base_component.Component
        collections.abc.MutableMapping
        collections.abc.Mapping
        collections.abc.Collection
        collections.abc.Sized
        collections.abc.Iterable
        collections.abc.Container
        object

    Methods defined here:

    __init__(self, children=None, **kwargs)

    __repr__(self)

    Data and other attributes defined here:

    __abstractmethods__ = frozenset()

    Methods inherited from dash.development.base_component.Component:

    __delitem__(self, id)
        Delete items by ID in the tree of children.

    __getitem__(self, id)
        Recursively find the element with the given ID through the tree
        of children.

    __iter__(self)
        Yield IDs in the tree of children.

    __len__(self)
        Return the number of items in the tree.

    __setitem__(self, id, item)
        Set an element by its ID.

    to_plotly_json(self)

    traverse(self)
        Yield each item in the tree.

    Data descriptors inherited from dash.development.base_component.Component:
```

Dash - Interactive Components

Interactive Components Overview

The first part of this tutorial covered the **layout** of Dash apps:

- The **layout** of a Dash app describes what the app looks like. It is a hierarchical tree of components.
- The **dash_html_components** library provides classes for all of the HTML tags and the keyword arguments describe the HTML attributes like style, className, and id.
- The **dash_core_components** library generates higher-level components like controls and graphs.

The second part of the tutorial describes how to make your Dash apps interactive.
Let's get started with a simple example.

Connecting Components with Callbacks

Adding a callback to one component

For this exercise we'll add a **callback** to an input box, and display the data being entered as an immediate output on the same screen.

Create a file called **callback1.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

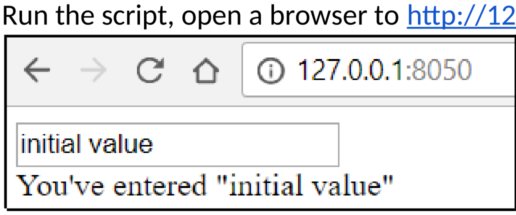
app = dash.Dash()

app.layout = html.Div([
    dcc.Input(id='my-id', value='initial value', type='text'),
    html.Div(id='my-div')
])

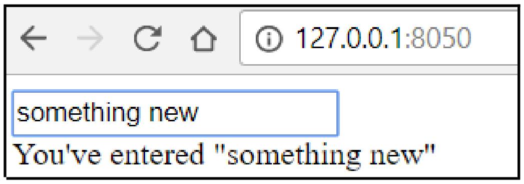
@app.callback(
    Output(component_id='my-div', component_property='children'),
    [Input(component_id='my-id', component_property='value')]
)
def update_output_div(input_value):
    return 'You\'ve entered "{}"'.format(input_value)

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ and you should see:



Now type something into the input box. Immediately you should see the output change to reflect the input!



Let's break down what's happening here:

1. We set up our **dcc.Input** in the usual way, except that we assigned an **id** to it, and added another Div after it with an assigned **id** (`'my-id'` and `'my-div'` respectively)

2. **app.callback** is called as a *decorator* function over `update_output_div`. The "inputs" and "outputs" of our application interface are described declaratively through the **app.callback** decorator.
   For more on Python decorators visit https://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators

3. Inside **@app.callback**, **Output** and **Input** are abbreviated forms of **dash.dependencies.Output** and **dash.dependencies.Input**. Note how we imported them from `dash.dependencies` by name.

4. In Dash, the inputs and outputs of our application are simply the properties of a particular component. In this example, our input is the **"value"** property of the component that has the ID `"my-id"`. Our output is the **"children"** property of the component with the ID `"my-div"`.

5. Whenever an input property changes, the function that the callback decorator wraps will get called automatically. Dash provides the function with the new value of the input property as an input argument and Dash updates the property of the output component with whatever was returned by the function.

6. The **component_id** and **component_property** keywords inside **Output** and **Input** are optional (there are only two arguments for each of those objects). We included them here for clarity but we'll omit them from here on out for brevity and readability.

7. Don't confuse the **dash.dependencies.Input** object inside **app.callback** from the **dash_core_components.Input** object inside **app.layout**. The former is just used in these callbacks and the latter is an actual component.

8. Notice how we don't set a value for the **children** property of the **my-div** component in the **layout**. When the Dash app starts, it automatically calls all of the callbacks with the initial values of the input components in order to populate the initial state of the output components. In this example, if you specified something like `html.Div(id='my-div', children='Hello world')`, it would get overwritten when the app starts.

It's sort of like programming with Microsoft Excel: whenever an input cell changes, all of the cells that depend on that cell will get updated automatically. This is called "Reactive Programming".

Remember how every component was described entirely through its set of keyword arguments? Those properties are important now. With Dash interactivity, we can dynamically update any of those properties through a callback function. Frequently we'll update the **children** of an **html** component to display new text or the **figure** of a **dcc.Graph** component to display new data, but we could also update the **style** of a component or even the available **options** of a **dcc.Dropdown** component!

Connecting two components with callbacks

The next example comes from the Dash online tutorial, and it's fairly complex. It introduces some layout features we haven't seen before, like a logarithmic x-axis. The goal is to have an interactive **Slider** update a **Graph** on the same page. We will use a Dash dataset available online at
https://raw.githubusercontent.com/plotly/datasets/master/gapminderDataFiveYear.csv

Create a file called **callback2.py** and add the following code:

```
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/gapminderDataFiveYear.csv')

app = dash.Dash()


# https://dash.plot.ly/dash-core-components/dropdown
# We need to construct a dictionary of dropdown values for the years
year_options = []
for year in df['year'].unique():
    year_options.append({'label':str(year),'value':year})

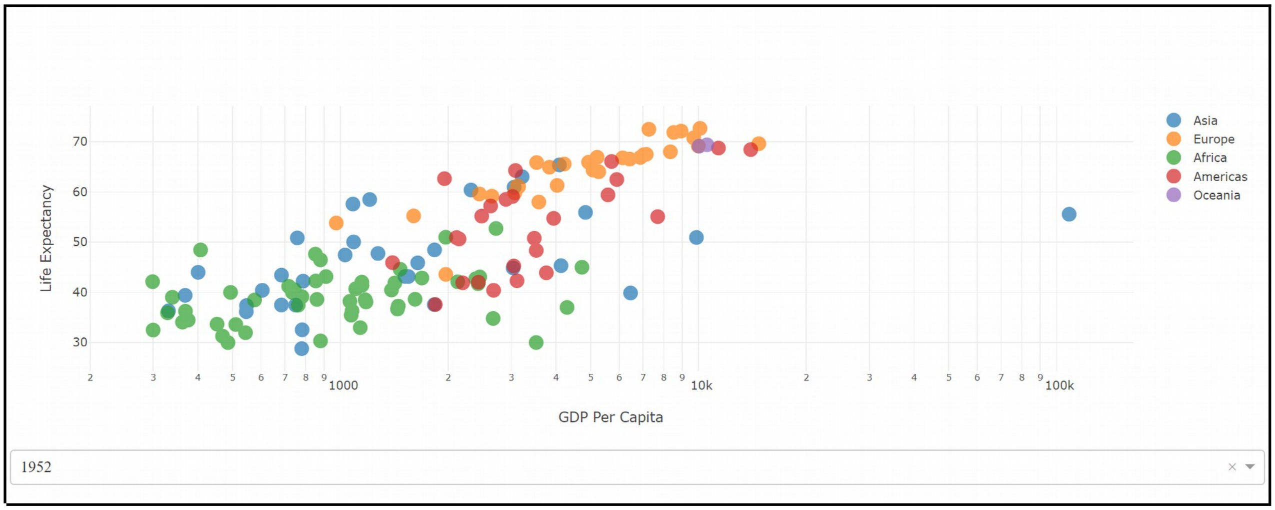
app.layout = html.Div([
    dcc.Graph(id='graph-with-slider'),
    dcc.Dropdown(id='year-picker',options=year_options,value=df['year'].min())
])

@app.callback(Output('graph-with-slider', 'figure'),
              [Input('year-picker', 'value')])
def update_figure(selected_year):
    filtered_df = df[df['year'] == selected_year]
    traces = []
    for continent_name in filtered_df['continent'].unique():
        df_by_continent = filtered_df[filtered_df['continent'] == continent_name]
        traces.append(go.Scatter(
            x=df_by_continent['gdpPercap'],
            y=df_by_continent['lifeExp'],
            text=df_by_continent['country'],
            mode='markers',
            opacity=0.7,
            marker={'size': 15},
            name=continent_name
        ))

    return {
        'data': traces,
        'layout': go.Layout(
            xaxis={'type': 'log', 'title': 'GDP Per Capita'},
            yaxis={'title': 'Life Expectancy'},
            hovermode='closest'
        )
    }

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ and you should see:



You can hover over any data point to reveal its Country, Continent and axis data. More importantly, you can use the dropdown to change the displayed graph!

Concerning style:

Before we discuss the connectivity, let's look at some of the style choices made:

- the x-axis is logarithmic, becoming denser as values increase
- we use the pandas `.unique()` method to extract the years for the dropdown (similar to our Plotly Linechart exercise!)

Concerning connectivity:

In this example, the **"value"** property of the **Dropdown** is the *input* of the app and the *output* of the app is the **"figure"** property of the **Graph**. Whenever the **value** of the **Dropdown** changes, Dash calls the callback function **update_figure** with the new value. The function filters the DataFrame with this new value, constructs a **figure** object, and returns it to the Dash application.

There are a few nice patterns in this example:

- We're using the Pandas library for importing and filtering datasets in memory.
- We load our DataFrame at the start of the app: `df = pd.read_csv('...')`. This DataFrame `df` is in the global state of the app and can be read inside the callback functions.
- Loading data into memory can be expensive. By loading querying data at the start of the app instead of inside the callback functions, we ensure that this operation is only done when the app server starts. When a user visits the app or interacts with the app, that data (the `df`) is already in memory. If possible, expensive initialization (like downloading or querying data) should be done in the global scope of the app instead of within the callback functions.
- The callback does not modify the original data, it just creates copies of the dataframe by filtered through pandas filters. This is important: your callbacks should never mutate variables outside of their scope. If your callbacks modify global state, then one user's session might affect the next user's session and when the app is deployed on multiple processes or threads, those modifications will not be shared across sessions.

Multiple Inputs

Input parameters are passed to the callback decorator as a list. For this reason, we can include multiple inputs in our dashboard to affect the same output through a callback function. For this example we'll use the mpg.csv dataset to show two input components - both dropdowns - will let us set the x-axis and y-axis features from our dataset.

Create a file called **callback3.py** and add the following:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd

app = dash.Dash()

df = pd.read_csv('../data/mpg.csv')

features = df.columns

app.layout = html.Div([
    html.Div([
        html.Div([
            dcc.Dropdown(
                id='xaxis',
                options=[{'label': i, 'value': i} for i in features],
                value='displacement'
            )
        ],
        style={'width': '48%', 'display': 'inline-block'}),

        html.Div([
            dcc.Dropdown(
                id='yaxis',
                options=[{'label': i, 'value': i} for i in features],
                value='acceleration'
            )
        ],style={'width': '48%', 'float': 'right', 'display': 'inline-block'})
    ]),

    dcc.Graph(id='feature-graphic')
], style={'padding':10})

@app.callback(
    Output('feature-graphic', 'figure'),
    [Input('xaxis', 'value'),
     Input('yaxis', 'value')])
def update_graph(xaxis_name, yaxis_name):
    return {
        'data': [go.Scatter(
            x=df[xaxis_name],
            y=df[yaxis_name],
            text=df['name'],
            mode='markers',
            marker={
                'size': 15,
                'opacity': 0.5,
                'line': {'width': 0.5, 'color': 'white'}
            }
        )],
        'layout': go.Layout(
            xaxis={'title': xaxis_name},
            yaxis={'title': yaxis_name},
            margin={'l': 40, 'b': 40, 't': 10, 'r': 0},
            hovermode='closest'
        )
    }

if __name__ == '__main__':
    app.run_server()
```

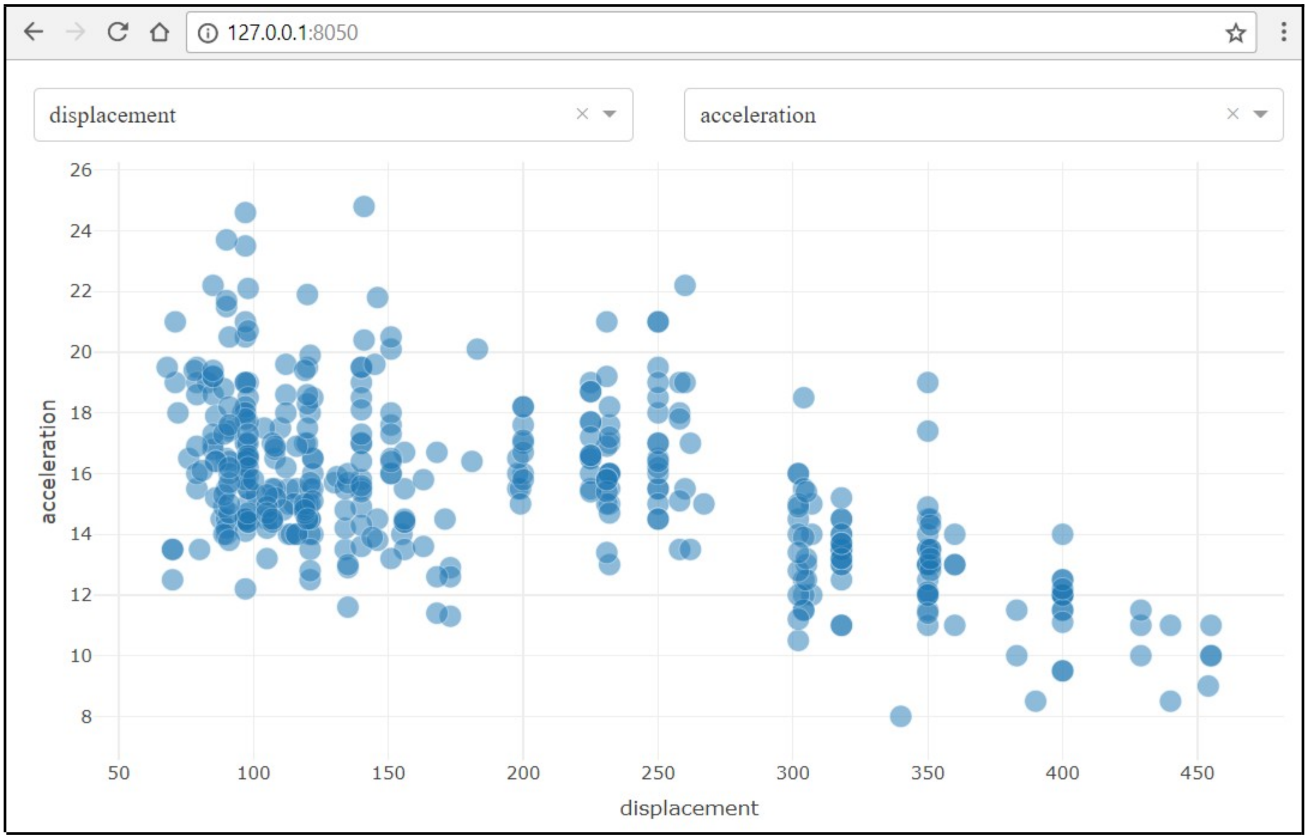
Let's look at what happened here:

- We set a variable *features* equal to the column names in our dataset. An alternative would be to set it to a recurring value in one dataset column. Note that setting this variable is optional - we could just as easily pass `df.columns` wherever *features* is used.
- Nothing new has happened in the **layout** section. Inside a Div we set our two dropdown components, followed

by our **Graph**.

- Notice, though, that **app.callback** now has *two* Input parameters, one for each dropdown.
- Other than two inputs, however, the returning update is relatively straightforward. We set up a Scatter plot with our x- and y-axes.

Run the script, open a browser to http://127.0.0.1:8050/ and you should see:



You can change either dropdown entry and immediately the x-axis and y-axis features change!

As a quick formatting choice, what if we wanted our features to appear capitalized? Even though our dataset column name is "displacement", how do we make "Displacement" appear on our graph both in the dropdown list and the axis title? This is actually a quick fix:

```
...
app.layout = html.Div([
    html.Div([
        html.Div([
            dcc.Dropdown(
                id='xaxis',
                options=[{'label': i.title(), 'value': i} for i in features],
                value='displacement'
...
        html.Div([
            dcc.Dropdown(
                id='yaxis',
                options=[{'label': i.title(), 'value': i} for i in features],
                value='acceleration'
            )
...
def update_graph(xaxis_name, yaxis_name):
...
        'layout': go.Layout(
            xaxis={'title': xaxis_name.title()},
            yaxis={'title': yaxis_name.title()},
            margin={'l': 40, 'b': 40, 't': 10, 'r': 0},
            hovermode='closest'
...
```

Which results in a graph like this:

For another example of multiple inputs, visit the Dash documentation at https://dash.plot.ly/getting-started-part-2. This shows not only dropdown lists but also radio buttons and a slider used as simultaneous input choices on the same graph.

Multiple Outputs

Each Dash callback function can only update a single Output property. In the above examples we show how to pass multiple inputs inside an **Input** list parameter. To update multiple **Outputs**, just write multiple functions.

For this example, we'll set up two sets of radio buttons, and two separate output areas.
Next, we'll add a third output that's determined by the *combination* of radio buttons selected!

Create a file called **callback4.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import pandas as pd

app = dash.Dash()

df = pd.read_csv('../data/wheels.csv')

app.layout = html.Div([
    dcc.RadioItems(
        id='wheels',
        options=[{'label': i, 'value': i} for i in df['wheels'].unique()],
        value=1
    ),
    html.Div(id='wheels-output'),

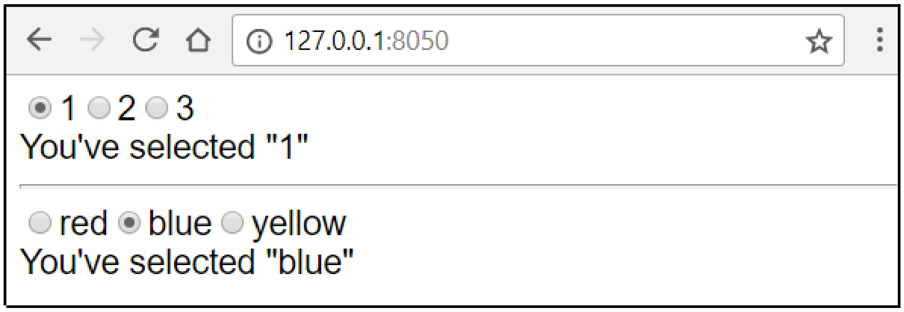
    html.Hr(),  # add a horizontal rule
    dcc.RadioItems(
        id='colors',
        options=[{'label': i, 'value': i} for i in df['color'].unique()],
        value='blue'
    ),
    html.Div(id='colors-output')
], style={'fontFamily':'helvetica', 'fontSize':18})
```

```
@app.callback(
    Output('wheels-output', 'children'),
    [Input('wheels', 'value')])
def callback_a(wheels_value):
    return 'You\'ve selected "{}"'.format(wheels_value)

@app.callback(
    Output('colors-output', 'children'),
    [Input('colors', 'value')])
def callback_b(colors_value):
    return 'You\'ve selected "{}"'.format(colors_value)

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ and you should see:



Changing either selection affects an independent output!

Resources: https://dash.plot.ly/getting-started-part-2

Let's expand this example, and have an output be determined by *both* inputs.
Make a duplicate of callback4.py and name it **callback5.py**. Add the following code (shown in bold).

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import pandas as pd
import base64

app = dash.Dash()

df = pd.read_csv('../data/wheels.csv')

def encode_image(image_file):
    encoded = base64.b64encode(open(image_file, 'rb').read())
    return 'data:image/png;base64,{}'.format(encoded.decode())

app.layout = html.Div([
    dcc.RadioItems(
        id='wheels',
        options=[{'label': i, 'value': i} for i in df['wheels'].unique()],
        value=1
    ),
    html.Div(id='wheels-output'),

    html.Hr(),  # add a horizontal rule
    dcc.RadioItems(
        id='colors',
        options=[{'label': i, 'value': i} for i in df['color'].unique()],
        value='blue'
    ),
    html.Div(id='colors-output'),
    html.Img(id='display-image', src='children', height=300)
], style={'fontFamily':'helvetica', 'fontSize':18})

@app.callback(
    Output('wheels-output', 'children'),
    [Input('wheels', 'value')])
def callback_a(wheels_value):
    return 'You\'ve selected "{}"'.format(wheels_value)

@app.callback(
    Output('colors-output', 'children'),
    [Input('colors', 'value')])
def callback_b(colors_value):
    return 'You\'ve selected "{}"'.format(colors_value)

@app.callback(
    Output('display-image', 'src'),
    [Input('wheels', 'value'),
     Input('colors', 'value')])
def callback_image(wheel, color):
    path = '../data/images/'
    return encode_image(path+df[(df['wheels']==wheel) & \
    (df['color']==color)]['image'].values[0])

if __name__ == '__main__':
    app.run_server()
```

Now when you run the script, the default values of **1** and **blue** display an image of a blue unicycle. Change *either* input to change the displayed image!

A couple of interesting techniques were introduced here:

- As of this writing, Dash doesn't serve up static files gracefully. To display images stored on the hard drive requires a conversion to base64. For this we defined a conversion function named "**encode_image**" and then used it inside our callback function.
- For our **Output**, **'display-image'** is the component ID, and **'src'** is the component_property we're affecting.
- We used pandas to obtain the name of our image file from the dataset using conditional selection. Note that the table only includes the filename, not the PATH. For this we set our own path variable inside the callback function. This way, we can modify our script to fit any other file structure.
- As of this writing, **html.Img** takes a **height=** argument, but not an **alt=** for providing alternate text in the event an image can't be retrieved.

Exercise: Interactive Components

For this exercise we want to take two or more integer inputs, and output their product. Be creative! You can use radio buttons, dropdowns, even a RangeSlider to obtain two input values. Use a callback to return the product of the two values. Don't forget to assign IDs to each component. Good luck!

Interactive Components Exercise Solution

For our suggested solution we chose a RangeSlider to obtain our two values. Note that RangeSliders return both values as a single list:

```
# Perform imports here:
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

# Launch the application:
app = dash.Dash()

# Create a Dash layout that contains input components
# and at least one output. Assign IDs to each component:
app.layout = html.Div([
    dcc.RangeSlider(        # this is the input
        id='range-slider',
        min=-5,
        max=6,
        marks={i:str(i) for i in range(-5, 7)},
        value=[-3, 4]
    ),
    html.H1(id='product')  # this is the output
], style={'width':'50%'})

# Create a Dash callback:
@app.callback(
    Output('product', 'children'),
    [Input('range-slider', 'value')])
def update_value(value_list):
    return value_list[0]*value_list[1]

# Add the server clause:
if __name__ == '__main__':
    app.run_server()
```

- **range(min, max+1)** won't work here. It has to be hardcoded unless min & max are defined outside of **layout**.

Controlling Callbacks with Dash State

In the previous interactive examples we've seen how inputs immediately affect outputs. As soon as values are entered, the page updates to reflect any changes.

What if we wanted to wait before displaying the page? What if we wanted time to enter a series of changes before submitting them? This is where **dash.dependencies.State** comes in. Dash offers the ability to store saved changes, and send them back on command. Consider this very basic example of Input/Output with a callback:

callback6.py

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

app = dash.Dash()

app.layout = html.Div([
    dcc.Input(
        id='number-in',
        value=1,
        style={'fontSize':28}
    ),
    html.H1(id='number-out')
])

@app.callback(
    Output('number-out', 'children'),
    [Input('number-in', 'value')])
def output(number):
    return number

if __name__ == '__main__':
    app.run_server()
```

As soon as you type characters into the Input box, they appear below as an HTML header.

Now let's add a Submit button, and store characters until the button is pressed:

callback6a.py (additional code is shown in **bold**)

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output, State

app = dash.Dash()

app.layout = html.Div([
    dcc.Input(
        id='number-in',
        value=1,
        style={'fontSize':28}
    ),
    html.Button(
        id='submit-button',
        n_clicks=0,
        children='Submit',
        style={'fontSize':28}
    ),
    html.H1(id='number-out')
])

@app.callback(
    Output('number-out', 'children'),
    [Input('submit-button', 'n_clicks')],
    [State('number-in', 'value')])
def output(n_clicks, number):
    return number

if __name__ == '__main__':
    app.run_server()
```

Now our Input is the action of clicking the **html.Button** element. The value typed into the Input box is stored inside of State, and is not passed to our Output until the Input registers a button click!

So what is **n_clicks**? It turns out, this stores the number of clicks that have occurred during the session. We can show this as part of our output if we want:

callback6b.py

```
...

@app.callback(
    Output('number-out', 'children'),
    [Input('submit-button', 'n_clicks')],
    [State('number-in', 'value')])
def output(n_clicks, number):
    return '{} displayed after {} clicks!'.format(number,n_clicks)

if __name__ == '__main__':
    app.run_server()
```

Each time you submit a new value, the page also reports the number of times the button has been clicked! It should be noted that *any* HTML element can be assigned an 'n_clicks' property.

Resources: https://dash.plot.ly/state

Interacting with Visualizations

Introduction to Interacting with Visualizations

The first part of this tutorial covered the **layout** of Dash apps:

- The **layout** of a Dash app describes what the app looks like. It is a hierarchical tree of components.
- The **dash_html_components** library provides classes for all of the HTML tags and the keyword arguments describe the HTML attributes like style, className, and id.
- The **dash_core_components** library generates higher-level components like controls and graphs.

The second part covered **callbacks**:

- The **dash.dependencies.Input** and **dash.dependencies.Output** components constantly monitor the page, and update an output display, graph, or other page content as needed.
- Dash supports multiple inputs and multiple outputs.
- You can hold onto input data using **dash.dependencies.State**, and have it submitted on demand using a Button or other html element.

In this next section, we revisit **dash_core_components.Graph**, and take a deep dive back into Plotly charts.

Resources: https://dash.plot.ly/interactive-graphing

Hover Over Data

Recall back in **Converting Simple Plotly Plot to Dashboard with Dash**, we displayed a Scatter plot comprised of random data points. When the cursor hovers over an individual point, the data for that point (the x-axis and y-axis values) are displayed as text.

Here we'll show how simply hovering over a data point can immediately affect another part of the figure!

We'll start by building a 3x3 scatterplot from our wheels.csv file. Recall that there are 3 x-axis values (red, yellow, blue) and 3 y-axis values (1,2,3).

Next we'll add a callback that takes in 'hoverData', and displays that data to the screen as a JSON object.

Create a file called **hover1.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd
import json

app = dash.Dash()

df = pd.read_csv('../data/wheels.csv')

app.layout = html.Div([
    html.Div([
    dcc.Graph(
        id='wheels-plot',
        figure={
            'data': [
                go.Scatter(
                    x = df['color'],
                    y = df['wheels'],
                    dy = 1,
                    mode = 'markers',
                    marker = {
                        'size': 12,
                        'color': 'rgb(51,204,153)',
                        'line': {'width': 2}
                        }
                )
            ],
            'layout': go.Layout(
                title = 'Wheels & Colors Scatterplot',
                xaxis = {'title': 'Color'},
                yaxis = {'title': '# of Wheels','nticks':3},
                hovermode='closest'
            )
        }
    )], style={'width':'30%', 'float':'left'}),

    html.Div([
    html.Pre(id='hover-data', style={'paddingTop':35})
    ], style={'width':'30%'})
])

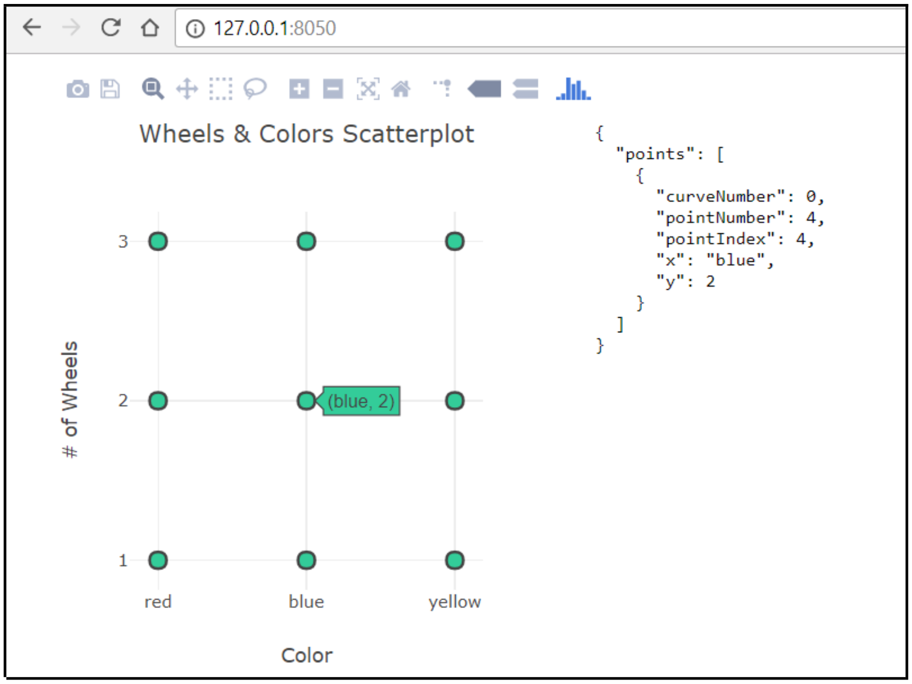
@app.callback(
    Output('hover-data', 'children'),
    [Input('wheels-plot', 'hoverData')])
def callback_image(hoverData):
    return json.dumps(hoverData, indent=2)

if __name__ == '__main__':
    app.run_server()
```

Some things of note:

- we **import json** so that we can display the captured hoverData as a **json.dumps** object.
- we label our output box **'hover-data'** only for convenience - this could be anything.
- Our input from 'wheels-plot' captures **'hoverData'** and we then pass **hoverData** into our callback function. *These* tags are important!
- We display the hoverData inside an **html <pre>** tag, which allows for pre-formatting (which we didn't use) and displays the contents in a fixed-width font, preserving spaces and line breaks.
- We added **`'nticks':3`** to the y-axis **layout** property. Without it the ticks would be [1, 1.5, 2, 2.5, 3]

Run the script, open a browser to http://127.0.0.1:8050/ and you should see something like this:



- Note that the initial state of the JSON output is "null", and only changes once a point is hovered over.

So how do we make use of hoverData? We do so through a series of dictionary calls!
Let's extract the x- and y-axis values of a data point, and retrieve the associated image file.

Make a duplicate of hover1.py and call it **hover2.py**. Add the following code (shown in bold).

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd
import base64

app = dash.Dash()

df = pd.read_csv('../data/wheels.csv')

def encode_image(image_file):
    encoded = base64.b64encode(open(image_file, 'rb').read())
    return 'data:image/png;base64,{}'.format(encoded.decode())

app.layout = html.Div([
    html.Div([
    dcc.Graph(
        id='wheels-plot',
        figure={
            'data': [
                go.Scatter(
                    x = df['color'],
                    y = df['wheels'],
                    dy = 1,
                    mode = 'markers',
                    marker = {
                        'size': 12,
                        'color': 'rgb(51,204,153)',
                        'line': {'width': 2}
                        }
                )
            ],
            'layout': go.Layout(
                title = 'Wheels & Colors Scatterplot',
                xaxis = {'title': 'Color'},
                yaxis = {'title': '# of Wheels','nticks':3},
                hovermode='closest'
            )
        }
    )], style={'width':'30%', 'float':'left'}),

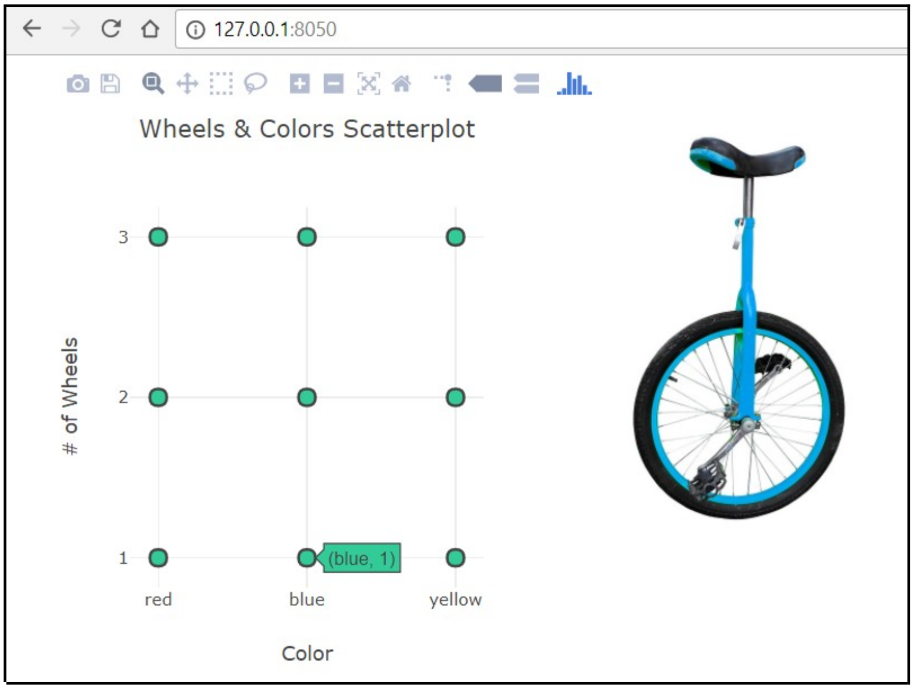
    html.Div([
    html.Img(id='hover-image', src='children', height=300)
    ], style={'paddingTop':35})
])

@app.callback(
    Output('hover-image', 'src'),
    [Input('wheels-plot', 'hoverData')])
def callback_image(hoverData):
    wheel=hoverData['points'][0]['y']
    color=hoverData['points'][0]['x']
    path = '../data/images/'
    return encode_image(path+df[(df['wheels']==wheel) & \
    (df['color']==color)]['image'].values[0])

if __name__ == '__main__':
    app.run_server()
```

The sections in blue bold are merely for handling images (recall that we have to convert files to base64 first).
Note how we use **`hoverData['points'][0]['y']`** to obtain the y-axis value.
We feed that into Pandas to retrieve the corresponding image file.

Run the script, open a browser to http://127.0.0.1:8050/, hover over any of the data points and you should see something like this:


127.0.0.1:8050
Wheels & Colors Scatterplot
# of Wheels
3
2
1
(blue, 1)
red
blue
yellow
Color

Click Data

Click Data is handled nearly the same way as Hover Data - it's simply an attribute of the graph that can be accessed using dictionary calls.

Make a duplicate of hover2.py and name it **click1.py**. Changes are shown in bold:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd
import base64

app = dash.Dash()

df = pd.read_csv('../data/wheels.csv')

def encode_image(image_file):
    encoded = base64.b64encode(open(image_file, 'rb').read())
    return 'data:image/png;base64,{}'.format(encoded.decode())

app.layout = html.Div([
    html.Div([
    dcc.Graph(
        id='wheels-plot',
        figure={
            'data': [
                go.Scatter(
                    x = df['color'],
                    y = df['wheels'],
                    dy = 1,
                    mode = 'markers',
                    marker = {
                        'size': 12,
                        'color': 'rgb(51,204,153)',
                        'line': {'width': 2}
                        }
                )
            ],
            'layout': go.Layout(
                title = 'Wheels & Colors Scatterplot',
                xaxis = {'title': 'Color'},
                yaxis = {'title': '# of Wheels','nticks':3},
                hovermode='closest'
            )
        }
   )], style={'width':'30%', 'float':'left'}),

    html.Div([
    html.Img(id='click-image', src='children', height=300)
    ], style={'paddingTop':35})
])

@app.callback(
    Output('click-image', 'src'),
    [Input('wheels-plot', 'clickData')])
def callback_image(clickData):
    wheel=clickData['points'][0]['y']
    color=clickData['points'][0]['x']
    path = '../data/images/'
    return encode_image(path+df[(df['wheels']==wheel) & \
    (df['color']==color)]['image'].values[0])

if __name__ == '__main__':
    app.run_server()
```

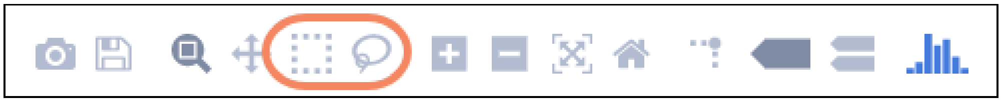
Really only two things changed:

- We changed the output box ID to **'click-image'**, although this wasn't really necessary.
- Instead of **hoverData**, we're passing **clickData** to our callback function.

Now when you run the script, images appear on the screen as data points are clicked on instead of hovered over. That's it! Everything else - including the dictionary call to obtain our x- and y-axis values - remains the same.

Selected Data

Selection Data makes use of the lasso or rectangle tool in the graph's menu bar:



and selected points in the graph.

To see what this looks like, duplicate hover1.py from Section 37, and name it **select1.py**. Changes are shown in bold:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd
import json

app = dash.Dash()

df = pd.read_csv('../data/wheels.csv')

app.layout = html.Div([
    html.Div([
    dcc.Graph(
        id='wheels-plot',
        figure={
            'data': [
                go.Scatter(
                    x = df['color'],
                    y = df['wheels'],
                    dy = 1,
                    mode = 'markers',
                    marker = {
                        'size': 12,
                        'color': 'rgb(51,204,153)',
                        'line': {'width': 2}
                        }
                )
            ],
            'layout': go.Layout(
                title = 'Wheels & Colors Scatterplot',
                xaxis = {'title': 'Color'},
                yaxis = {'title': '# of Wheels','nticks':3},
                hovermode='closest'
            )
        }
   )], style={'width':'30%', 'display':'inline-block'}),

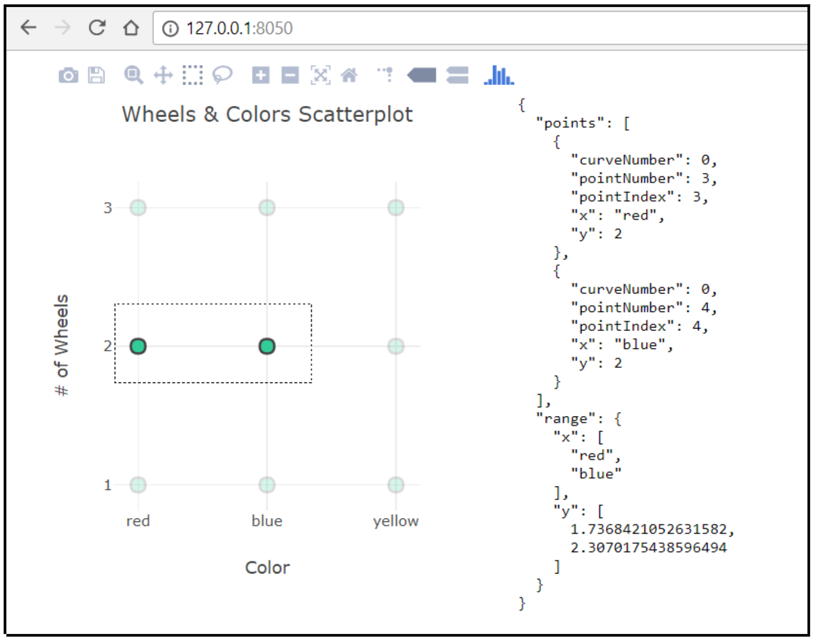
    html.Div([
    html.Pre(id='selection', style={'paddingTop':25})
    ], style={'width':'30%', 'display':'inline-block', 'verticalAlign':'top'})
])

@app.callback(
    Output('selection', 'children'),
    [Input('wheels-plot', 'selectedData')])
def callback_image(selectedData):
    return json.dumps(selectedData, indent=2)

if __name__ == '__main__':
    app.run_server()
```

Here we changed the Input parameter to `'selectedData'`.
Shown in blue, we also changed the Div styles from `'float':'left'` to `'display':'inline-block'`.
This prevents the longer JSON output from pushing the graph downward or wrapping underneath it.

Run the script, open a browser to http://127.0.0.1:8050/ , and use the lasso and rectangle selection tools in the graph menu bar to select groups of data points. You should see something like this:



The returning dictionary has a key for 'points' and another key for either 'range' or 'lassoPoints'.
Points data is similar to what we saw above for hover and click, only this time the list contains a dictionary for every encircled point.
Range data contains 'x' and 'y' axis boundaries for the selection box itself.
LassoPoints can be a fairly long list. These are the (x,y) coordinate pairs that define the selection boundary.

Let's put this to use! One problem we find with scatter plots is it can be difficult to identify overlapping data points. Setting opacity helps (two points occupying the same space will be darker than one point alone), but not foolproof. For this example we'll make an artificial dataset, plot points, and use Selected Data to determine the *density* of points in a given region of the plot.

Create a file named **select2.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import numpy as np
import pandas as pd

app = dash.Dash()

# create x and y arrays
np.random.seed(10)          # for reproducible results
x1 = np.linspace(0.1,5,50)  # left half
x2 = np.linspace(5.1,10,50) # right half
y = np.random.randint(0,50,50) # 50 random points

# create three "half DataFrames"
df1 = pd.DataFrame({'x': x1, 'y': y})
df2 = pd.DataFrame({'x': x1, 'y': y})
df3 = pd.DataFrame({'x': x2, 'y': y})

# combine them into one DataFrame (df1 and df2 points overlap!)
df = pd.concat([df1,df2,df3])

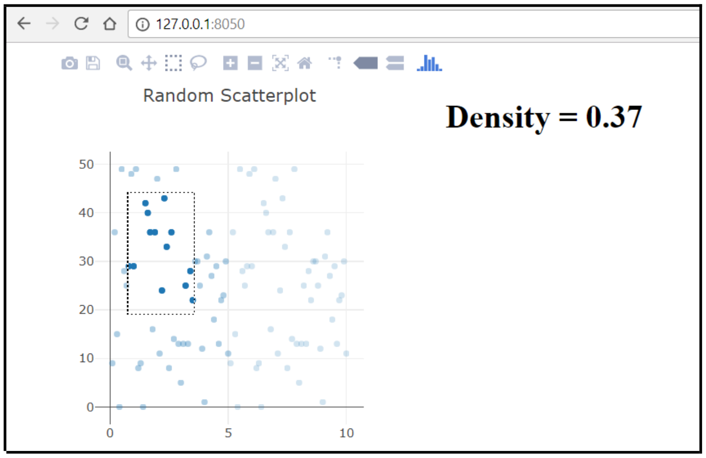
app.layout = html.Div([
    html.Div([
    dcc.Graph(
        id='plot',
        figure={
            'data': [
                go.Scatter(
                    x = df['x'],
                    y = df['y'],
                    mode = 'markers'
                )
            ],
            'layout': go.Layout(
                title = 'Random Scatterplot',
                hovermode='closest'
            )
        }
    )], style={'width':'30%', 'display':'inline-block'}),

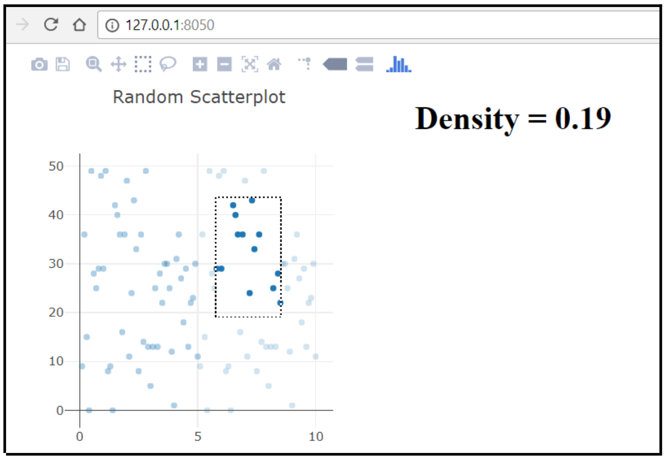
    html.Div([
    html.H1(id='density', style={'paddingTop':25})
    ], style={'width':'30%', 'display':'inline-block', 'verticalAlign':'top'})
])

@app.callback(
    Output('density', 'children'),
    [Input('plot', 'selectedData')])
def find_density(selectedData):
    pts = len(selectedData['points'])
    rng_or_lp = list(selectedData.keys())
    rng_or_lp.remove('points')
    max_x = max(selectedData[rng_or_lp[0]]['x'])
    min_x = min(selectedData[rng_or_lp[0]]['x'])
    max_y = max(selectedData[rng_or_lp[0]]['y'])
    min_y = min(selectedData[rng_or_lp[0]]['y'])
    area = (max_x-min_x)*(max_y-min_y)
    d = pts/area
    return 'Density = {:.2f}'.format(d)

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ , and use the lasso and rectangle selection tools in the graph menu bar to select groups of data points on *either side* of the graph. You should see something like this:





Everything we did here resembles the JSON output script, except for finding the density. Because Selected Data returns either a “range” key or a “lassoPoints” key depending on the tool used, we had to get creative with how we mined the size of the selection. Note that lassos will always have overstated areas, since essentially we’re just building a box around the min and max “x” and “y” values of the blob.

In this example, the points on the left half of the plot are doubled up (wherever you see a point, there are actually two overlapping points). The right half of the plot is occupied by single points. Thus, the calculated density is twice as high on the left as on the right for similar selections of points.

If you’re curious what the JSON output looks like for this chart, run the included **select2a.py** file that’s included in the course materials.

Updating Graphs on Interactions

So far in this section on *Interacting with Visualizations*, we've only used Hover, Click and Select to display new data on the screen. In this next part we show how to apply these tools to one graph, and have them trigger changes to other graphs in the same dashboard.

For this exercise we revisit the mpg.csv dataset since it has a convenient number of data points we can hover over. To set up a useful scatter plot we'll want to spread the points out along the x-axis. Model Year is a good feature, but we'll add an artificial "jitter" to the data so that points don't all line up along distinct verticals.

To the right of our scatter plot we'll create a line plot that represents the acceleration of a selected vehicle. The steeper the line, the quicker the acceleration. We'll remove the x- and y-axis ticks - all we want is for the line to show relative comparisons.

Some math: recall that the dataset has a column for acceleration that represents the time in seconds to go from zero to sixty miles per hour. To translate this into slope we'll use the following formula:

$$acceleration = \frac{\Delta v}{t} = \frac{\Delta miles\ per\ minute}{¿\ minutes} = \frac{(60\ miles\ per\ hour)/(60\ minutes/hour)}{(¿\ seconds)/(60\ seconds/minute)} = \frac{60}{¿\ seconds}$$

Therefore, the greater the number of seconds, the slower the acceleration, and the flatter the slope.

Create a file named **updating1.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import plotly.graph_objs as go
import pandas as pd
from numpy import random

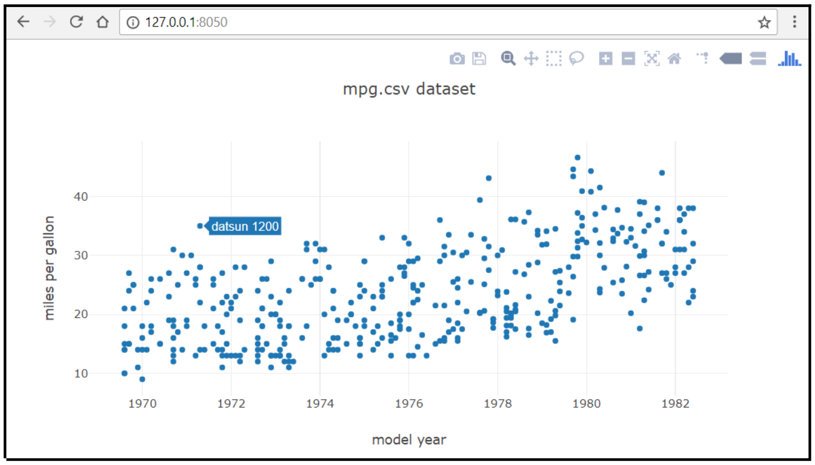
app = dash.Dash()

df = pd.read_csv('../data/mpg.csv')
# Add a random "jitter" to model_year to spread out the plot
df['year'] = random.randint(-4,5,len(df))*0.10 + df['model_year']

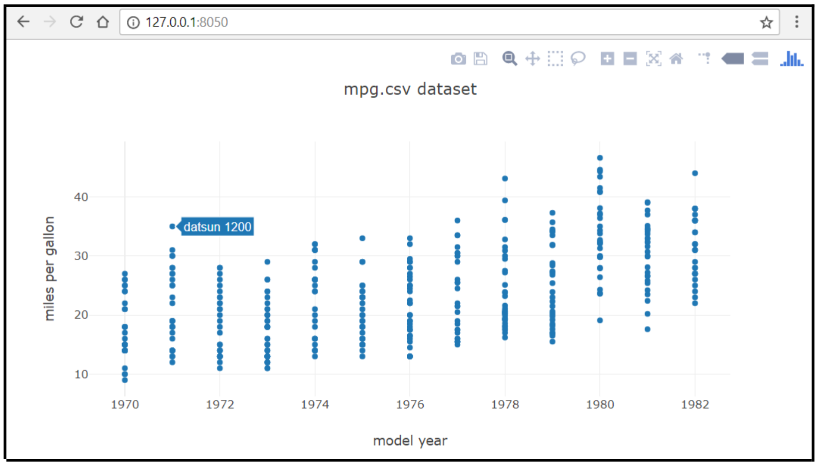
app.layout = html.Div([
    dcc.Graph(
        id='mpg_scatter',
        figure={
            'data': [go.Scatter(
                x = df['year']+1900,  # our "jittered" data
                y = df['mpg'],
                text = df['name'],
                hoverinfo = 'text',
                mode = 'markers'
            )],
            'layout': go.Layout(
                title = 'mpg.csv dataset',
                xaxis = {'title': 'model year'},
                yaxis = {'title': 'miles per gallon'},
                hovermode='closest'
            )
        }
    )
])

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ , and you should see something like:



We used random values for our jitter, so yours may look slightly different.
If we hadn't added the jitter, the graph would have looked like this:



Next, we'll add a line graph representing acceleration, and tie it back to our scatter plot with hoverData.
Copy updating1.py and name the new file **updating2.py**. Add the following code (shown in bold):

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd
from numpy import random

app = dash.Dash()

df = pd.read_csv('../data/mpg.csv')

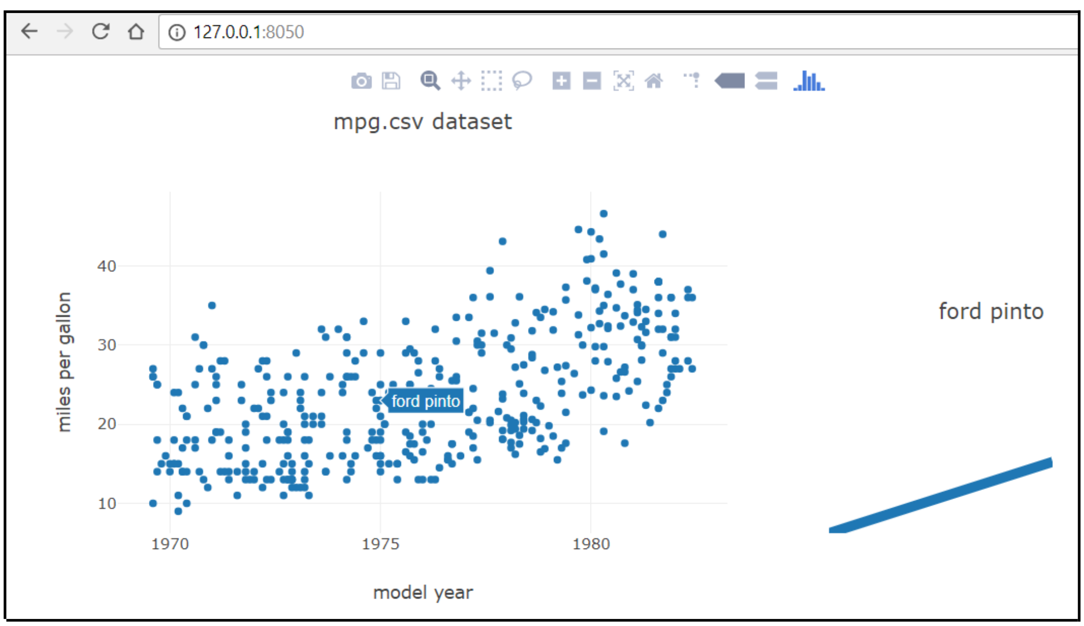
# Add a random "jitter" to model_year to spread out the plot
df['year'] = df['model_year'] + random.randint(-4,5,len(df))*0.10

app.layout = html.Div([
    html.Div([   # this Div contains our scatter plot
    dcc.Graph(
        id='mpg_scatter',
        figure={
            'data': [go.Scatter(
                x = df['year']+1900,  # our "jittered" data
                y = df['mpg'],
                text = df['name'],
                hoverinfo = 'text',
                mode = 'markers'
            )],
            'layout': go.Layout(
                title = 'mpg.csv dataset',
                xaxis = {'title': 'model year'},
                yaxis = {'title': 'miles per gallon'},
                hovermode='closest'
            )
```

```
            }
        # add style to the Div to make room for our output graph
        )], style={'width':'50%','display':'inline-block'}),
        # add a new Div for our output graph
        html.Div([  # this Div contains our output graph
        dcc.Graph(
            id='mpg_line',
            figure={
                'data': [go.Scatter(
                    x = [0,1],
                    y = [0,1],
                    mode = 'lines'
                )],
                'layout': go.Layout(
                    title = 'acceleration',
                    margin = {'l':0}
                )
            }
        )
        ], style={'width':'20%', 'height':'50%','display':'inline-block'})
])
# add a callback
@app.callback(
    Output('mpg_line', 'figure'),
    [Input('mpg_scatter', 'hoverData')])
def callback_graph(hoverData):
    v_index = hoverData['points'][0]['pointIndex']
    fig = {
        'data': [go.Scatter(
            x = [0,1],
            y = [0,60/df.iloc[v_index]['acceleration']],
            mode='lines',
            line={'width':2*df.iloc[v_index]['cylinders']}
        )],
        'layout': go.Layout(
            title = df.iloc[v_index]['name'],
            xaxis = {'visible':False},
            yaxis = {'visible':False, 'range':[0,60/df['acceleration'].min()]},
            margin = {'l':0},
            height = 300
        )
    }
    return fig

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ , and you should see something like:



As you hover over different vehicles, the graph on the right changes pitch (higher for quicker cars), and thickness depending on the number of cylinders.

Let's add one more feature, and have vehicle statistics appear as a **dcc.Markdown** element.

Copy updating2.py and name the new file **updating3.py**. Add the following code (shown in bold):

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import pandas as pd
from numpy import random

app = dash.Dash()

df = pd.read_csv('../data/mpg.csv')

# Add a random "jitter" to model_year to spread out the plot
df['year'] = df['model_year'] + random.randint(-4,5,len(df))*0.10

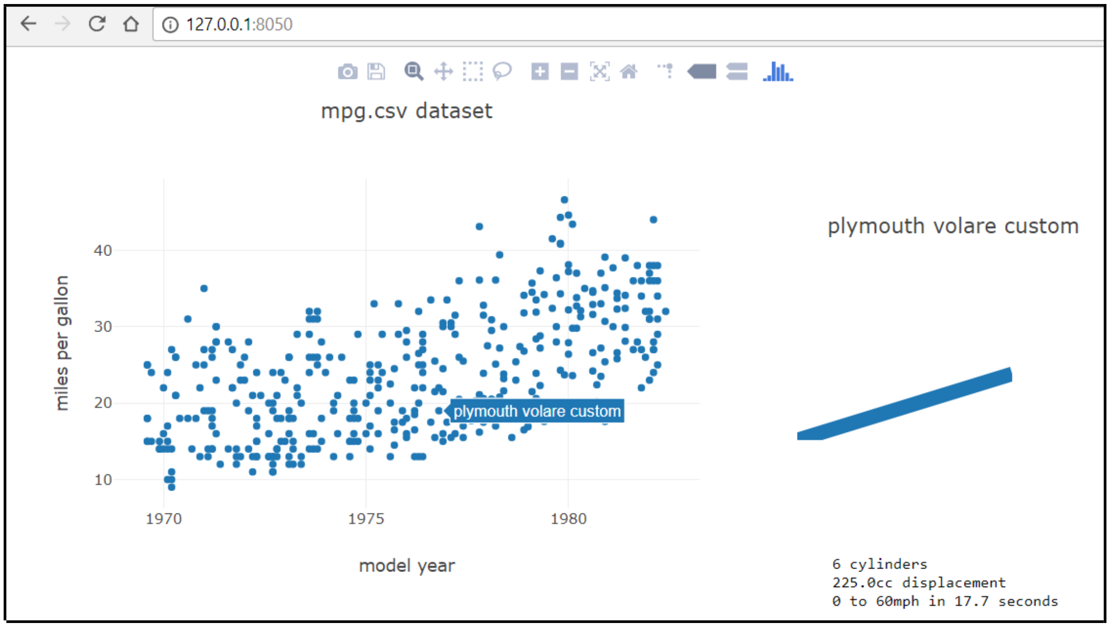
app.layout = html.Div([
    html.Div([   # this Div contains our scatter plot
    dcc.Graph(
        id='mpg_scatter',
        figure={
            'data': [go.Scatter(
                x = df['year']+1900,  # our "jittered" data
                y = df['mpg'],
                text = df['name'],
                hoverinfo = 'text',
                mode = 'markers'
            )],
            'layout': go.Layout(
                title = 'mpg.csv dataset',
                xaxis = {'title': 'model year'},
                yaxis = {'title': 'miles per gallon'},
                hovermode='closest'
            )
        }
    )], style={'width':'50%','display':'inline-block'}),
    html.Div([  # this Div contains our output graph and vehicle stats
    dcc.Graph(
        id='mpg_line',
        figure={
            'data': [go.Scatter(
                x = [0,1],
                y = [0,1],
                mode = 'lines'
            )],
            'layout': go.Layout(
                title = 'acceleration',
                margin = {'l':0}
            )
        }
    ),
    # add a Markdown section
    dcc.Markdown(
        id='mpg_stats'
    )
    ], style={'width':'20%', 'height':'50%','display':'inline-block'})
])

@app.callback(
    Output('mpg_line', 'figure'),
    [Input('mpg_scatter', 'hoverData')])
def callback_graph(hoverData):
    v_index = hoverData['points'][0]['pointIndex']
    fig = {
        'data': [go.Scatter(
            x = [0,1],
            y = [0,60/df.iloc[v_index]['acceleration']],
            mode='lines',
            line={'width':2*df.iloc[v_index]['cylinders']}
        )],
        'layout': go.Layout(
            title = df.iloc[v_index]['name'],
            xaxis = {'visible':False},
            yaxis = {'visible':False, 'range':[0,60/df['acceleration'].min()]},
            margin = {'l':0},
            height = 300
        )
    }
    return fig
# add a second callback for our Markdown
@app.callback(
    Output('mpg_stats', 'children'),
    [Input('mpg_scatter', 'hoverData')])
def callback_stats(hoverData):
    v_index = hoverData['points'][0]['pointIndex']
```

```
    stats = """
        {} cylinders
        {}cc displacement
        0 to 60mph in {} seconds
        """.format(df.iloc[v_index]['cylinders'],
            df.iloc[v_index]['displacement'],
            df.iloc[v_index]['acceleration'])
    return stats

if __name__ == '__main__':
    app.run_server()
```

Run the script, open a browser to http://127.0.0.1:8050/ , and you should see something like:



That's it! Now we've used hover to dynamically change another graph on the same page, and populate a Markdown section at the same time.

Code Along Milestone Project

We offer a culminating project here. Some new material may be covered, so we encourage research into pandas, plotly, and the Dash documentation.

This project develops a Stock Ticker Dashboard that allows the user to either enter a ticker symbol into an input box, or to select item(s) from a dropdown list, and uses pandas_datareader to look up and display stock data on a graph. The final project will include a DatePicker to set the start and end dates for the graph:

Refer to the separate Milestone Project Google Doc for details.

Introduction to Live Updating

So far we've shown a lot of ways to work with static data. For convenience we have provided the .csv files themselves, but in most cases we could just as easily have programmed the source websites into our graphs.

But what if the information on the web is constantly changing? For this section we can't supply a .csv file because our source data, https://www.flightradar24.com updates its information every 8 seconds!

This section introduces the **dash_core_components.Interval** component. Instead of waiting for some user interaction to update the page, Interval lets you update components in your application every few seconds or minutes.

Resources: https://dash.plot.ly/live-updates

Simple Live Updating Example

Before we invoke the dcc.Interval component, let's consider an update on page load. From the Dash documentation: "By default, Dash apps store the **app.layout** in memory. This ensures that the **layout** is only computed once, when the app starts. If you set **app.layout** to a function, then you can serve a dynamic layout on every page load."

To demonstrate, create a file called **layoutupdate0.py** and add the following:

```
import dash
import dash_html_components as html

app = dash.Dash()

crash_free = 0
crash_free += 1

app.layout = html.H1('Crash free for {} refreshes'.format(crash_free))

if __name__ == '__main__':
    app.run_server()
```

Run the script to display the page, and then refresh the page several times. Note that the layout doesn't change.

Copy layoutupdate0.py to a new file **layoutupdate1.py** and add the following code (shown in bold):

```
import dash
import dash_html_components as html

app = dash.Dash()

crash_free = 0
def refresh_layout():
    global crash_free
    crash_free += 1
    return html.H1('Crash free for {} refreshes'.format(crash_free))

app.layout = refresh_layout

if __name__ == '__main__':
    app.run_server()
```

Run the script. Now you should see that refreshing the page *does* update the layout.

Now it's time to make the page refresh at regular intervals automatically.

Copy layoutupdate1.py to a new file **layoutupdate2.py** and add the following code (shown in bold):

```
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output

app = dash.Dash()

app.layout = html.Div([
    html.H1(id='live-update-text'),
    dcc.Interval(
        id='interval-component',
        interval=2000, # 2000 milliseconds = 2 seconds
        n_intervals=0
    )
])

@app.callback(Output('live-update-text', 'children'),
              [Input('interval-component', 'n_intervals')])
def update_layout(n):
    return 'Crash free for {} refreshes'.format(n)

if __name__ == '__main__':
    app.run_server()
```

Here we're using a callback **Input** (**dcc.Interval**) to trigger a callback **Output** (our **html.H1** tag) at regular intervals.

Run the script, and the layout should update automatically every 2 seconds!

Remember, the IDs we assign our Input and Output elements are arbitrary (**'live-update-text'** and **'interval-component'** in this case). However, the property names we use are important. We want to input the **'n_intervals'** property of the **dcc.Interval** component, and in this situation we want to return a **'children'** property to our **html.H1** component (here it's the string that will be become the Header text).

In this next example we'll scrape a website that updates every eight seconds. The site https://www.flightradar24.com receives flight data from around the world and continually updates its page by plotting real time flight data on top of Google maps.

The data we care about is only going to be the total number of active flights worldwide. This is shown in the upper left corner of the screen, right next to the number of flights contained in the current view. It's worth noting that flightradar24 data arrives from a number of sources, including radar stations (ADS-B, FLARM, MLAT, FAA) as well as estimated numbers.

It would be nice to be able to scrape the opening page and grab this data. The script would look something like this:

```
import bs4, requests
res = requests.get('https://www.flightradar24.com', headers={'User-Agent': 'Mozilla/5.0'})
soup = bs4.BeautifulSoup(res.text,'lxml')
soup.select('#statTotal')
```

Unfortunately, most of the data displayed on flightradar24's page is derived from JavaScript calls!

Fortunately, we can still handle this with a little JSON parsing. If you're curious where the url we're about to use came from, simply inspect the `#statTotal` element in developer tools, open Network, and take a look at the various JavaScript calls that are going on.

Open a new file and name it **liveupdating1.py**. Add the following code:

```
import dash
import dash_html_components as html
import requests

url = "https://data-live.flightradar24.com/zones/fcgi/feed.js?faa=1\
       &mlat=1&flarm=1&adsb=1&gnd=1&air=1&vehicles=1&estimated=1&stats=1"

# A fake header is necessary to access the site
res = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
data = res.json()
counter = 0
for element in data["stats"]["total"]:
    counter += data["stats"]["total"][element]

app = dash.Dash()

app.layout = html.Div([
    html.Div([
        html.Iframe(src = 'https://www.flightradar24.com', height = 500, width = 1200)
    ]),

    html.Div([
    html.Pre('Active flights worldwide: {}'.format(counter))
    ])
])

if __name__ == '__main__':
    app.run_server()
```

Here we've embedded the flightradar24 website itself into our own page, followed by the counter value obtained via web scraping! Note that if you refresh the page, the counter value doesn't change. Once set by our script, that value remains until the script is halted and restarted.

Next, let's add a **dcc.Interval** component.

Make a duplicate of liveupdating1.py and name it **liveupdating2.py**. Add the following code (shown in bold):

```
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output
import requests

app = dash.Dash()

app.layout = html.Div([
    html.Div([
        html.Iframe(src = 'https://www.flightradar24.com', height = 500, width = 1200)
    ]),

    html.Div([
    html.Pre(
        id='counter_text',
        children='Active flights worldwide:'
    ),
    dcc.Interval(
        id='interval-component',
        interval=6000, # 6000 milliseconds = 6 seconds
        n_intervals=0
    )])
])

@app.callback(Output('counter_text', 'children'),
              [Input('interval-component', 'n_intervals')])
def update_layout(n):
    url = "https://data-live.flightradar24.com/zones/fcgi/feed.js?faa=1\
           &mlat=1&flarm=1&adsb=1&gnd=1&air=1&vehicles=1&estimated=1&stats=1"
    # A fake header is necessary to access the site:
    res = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
    data = res.json()
    counter = 0
    for element in data["stats"]["total"]:
        counter += data["stats"]["total"][element]
    return 'Active flights worldwide: {}'.format(counter)

if __name__ == '__main__':
    app.run_server()
```

Note that we simply moved the url request section to inside the update_layout function definition.

Run the script, and you'll notice that the flight total updates every six seconds. It won't be in perfect sync with flightradar24, but it will be close.

Finally, let's plot the incoming data.

Make a duplicate of liveupdating2.py and name it **liveupdating3.py**. Add the following code (shown in bold):

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import requests

app = dash.Dash()

app.layout = html.Div([
    html.Div([
        html.Iframe(src = 'https://www.flightradar24.com', height = 500, width = 1200)
    ]),

    html.Div([
    html.Pre(
        id='counter_text',
        children='Active flights worldwide:'
    ),
    dcc.Graph(id='live-update-graph',style={'width':1200}),
    dcc.Interval(
        id='interval-component',
        interval=6000, # 6000 milliseconds = 6 seconds
        n_intervals=0
    )])
])
counter_list = []

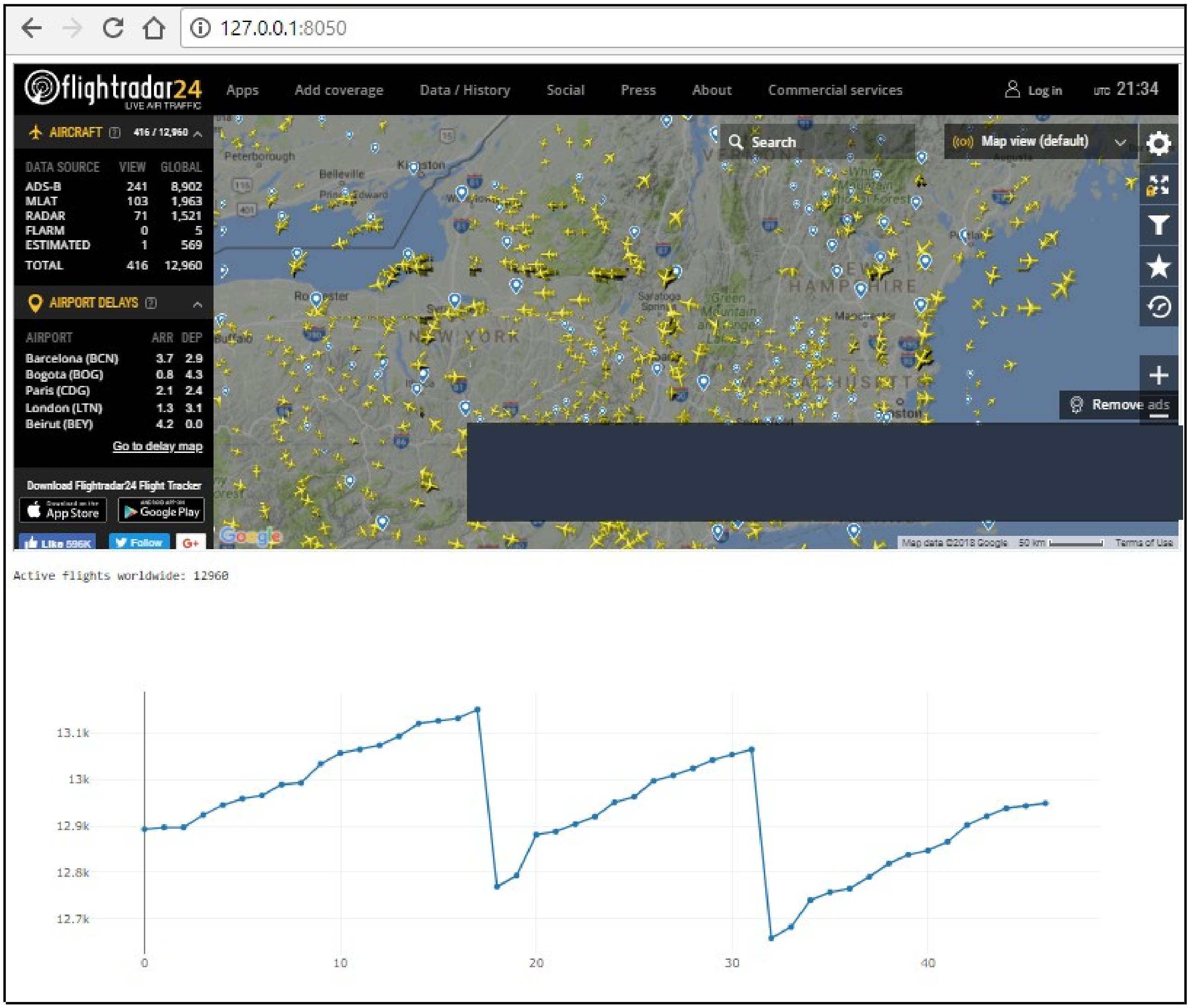
@app.callback(Output('counter_text', 'children'),
              [Input('interval-component', 'n_intervals')])
def update_layout(n):
    url = "https://data-live.flightradar24.com/zones/fcgi/feed.js?faa=1\
          &mlat=1&flarm=1&adsb=1&gnd=1&air=1&vehicles=1&estimated=1&stats=1"
    # A fake header is necessary to access the site:
    res = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
    data = res.json()
    counter = 0
    for element in data["stats"]["total"]:
        counter += data["stats"]["total"][element]
    counter_list.append(counter)
    return 'Active flights worldwide: {}'.format(counter)

@app.callback(Output('live-update-graph','figure'),
              [Input('interval-component', 'n_intervals')])
def update_graph(n):
    fig = go.Figure(
        data = [go.Scatter(
        x = list(range(len(counter_list))),
        y = counter_list,
        mode='lines+markers'
        )])
    return fig

if __name__ == '__main__':
    app.run_server()
```

Run the script, and now we have a constantly updating line chart beneath the website!
Notice we haven't done anything with datetime. This graph simply plots the data we've stored since the page was opened, letting us see the trend in the number of active flights worldwide.

After awhile, your page may look something like this:


127.0.0.1:8050
flightradar24
LIVE AIR TRAFFIC
Apps
Add coverage
Data / History
Social
Press
About
Commercial services
Log in
UTC 21:34
AIRCRAFT 416 / 12,960
DATA SOURCE VIEW GLOBAL
ADS-B 241 8,902
MLAT 103 1,963
RADAR 71 1,521
FLARM 0 5
ESTIMATED 1 569
TOTAL 416 12,960
AIRPORT DELAYS
AIRPORT ARR DEP
Barcelona (BCN) 3.7 2.9
Bogota (BOG) 0.8 4.3
Paris (CDG) 2.1 2.4
London (LTN) 1.3 3.1
Beirut (BEY) 4.2 0.0
Go to delay map
Download Flightradar24 Flight Tracker
App Store
Google Play
Search
Map view (default)
Remove ads
NEW YORK
NEW HAMPSHIRE
Active flights worldwide: 12960
13.1k
13k
12.9k
12.8k
12.7k
0
10
20
30
40

Good job!

Deployment

Introduction to Deploying Apps

In this section we'll look at the final phase of dashboard development - deployment! We show how to deploy your app on Heroku, and how to add a user authentication to your app so that only invited guests can view its contents.

Before deploying your app, you may decide to add user authentication (username and password).

App Authorization

From the Dash documentation:

Authentication for dash apps is provided through a separate **dash-auth** package.
dash-auth provides two methods of authentication: **HTTP Basic Auth** and **Plotly OAuth**.

HTTP Basic Auth is one of the simplest forms of authentication on the web. As a Dash developer, you hardcode a set of usernames and passwords in your code and send those usernames and passwords to your viewers. There are a few limitations to HTTP Basic Auth:

- Users can not log out of applications
- You are responsible for sending the usernames and passwords to your viewers over a secure channel
- Your viewers can not create their own account and cannot change their password
- You are responsible for safely storing the username and password pairs in your code.

Plotly OAuth provides authentication through your online Plotly account or through your company's Plotly On-Premise server. As a Dash developer, this requires a paid Plotly subscription. Here's where you can subscribe to Plotly Cloud, and here's where you can contact us about Plotly On-Premise. The viewers of your app will need a Plotly account but they do not need to upgrade to a paid subscription.

Plotly OAuth allows you to share your apps with other users who have Plotly accounts. With Plotly On-Premise, this includes sharing apps through the integrated LDAP system. Apps that you have saved will appear in your list of files at https://plot.ly/organize and you can manage the permissions of the apps there. Viewers create and manage their own accounts.

HTTP Basic Auth will be sufficient for our purposes. To add authentication to your app, first make sure that both **dash** and **dash-auth** are installed on your system:

```
$ pip install dash
$ pip install dash-auth
```

Next, pick an app from earlier in the course that you would like to deploy. We're going to use the solution to our Interactive Components exercise since it's a fairly short script (it returns the product of two values submitted by a range slider).
Create a new file called **auth1.py** and add the following code:

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

app = dash.Dash()

app.layout = html.Div([
    dcc.RangeSlider(
        id='range-slider',
        min=-5,
        max=6,
        marks={i:str(i) for i in range(-5, 7)},
```

```
        value=[-3, 4]
    ),
    html.H1(id='product')  # this is the output
], style={'width':'50%'})

@app.callback(
    Output('product', 'children'),
    [Input('range-slider', 'value')])
def update_value(value_list):
    return value_list[0]*value_list[1]

if __name__ == '__main__':
    app.run_server()
```

Run the script just to make sure it works, then add the following code (shown in bold):

```
import dash
import dash_auth
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

USERNAME_PASSWORD_PAIRS = [
    ['JamesBond', '007'],['LouisArmstrong', 'satchmo']
]

app = dash.Dash()
auth = dash_auth.BasicAuth(app,USERNAME_PASSWORD_PAIRS)

app.layout = html.Div([
    dcc.RangeSlider(
        id='range-slider',
        min=-5,
        max=6,
        marks={i:str(i) for i in range(-5, 7)},
        value=[-3, 4]
    ),
    html.H1(id='product')  # this is the output
], style={'width':'50%'})

@app.callback(
    Output('product', 'children'),
    [Input('range-slider', 'value')])
def update_value(value_list):
    return value_list[0]*value_list[1]

if __name__ == '__main__':
    app.run_server()
```

That's it! Run the script, open a browser to http://127.0.0.1:8050/ , and you should see be prompted for a username and password before the app will load. We should point out a couple of things:

- The username is case sensitive. `JamesBond` will work, but `jamesbond` will not.
- In production, you should store your USERNAME_PASSWORD_PAIRS in a separate file or database, and not inside your source code as we have it.
- The field name is arbitrary; we used USERNAME_PASSWORD_PAIRS but you can name yours anything you want so long as the same name is passed into **dash_auth.BasicAuth**.

Resources: https://dash.plot.ly/authentication

Deploying App to Heroku

Every Dash script so far has used `app.run_server()` to launch the app. By default the app runs on **localhost**, and you can only see it on your own machine.

The good news is that Dash uses Flask as its web framework, so anywhere you can deploy Flask, you can deploy Dash. While there are many options out there including Digital Ocean, PythonAnywhere, Google Cloud, Amazon Web Services, Azure, etc., we'll walk through an app deployment on Heroku.

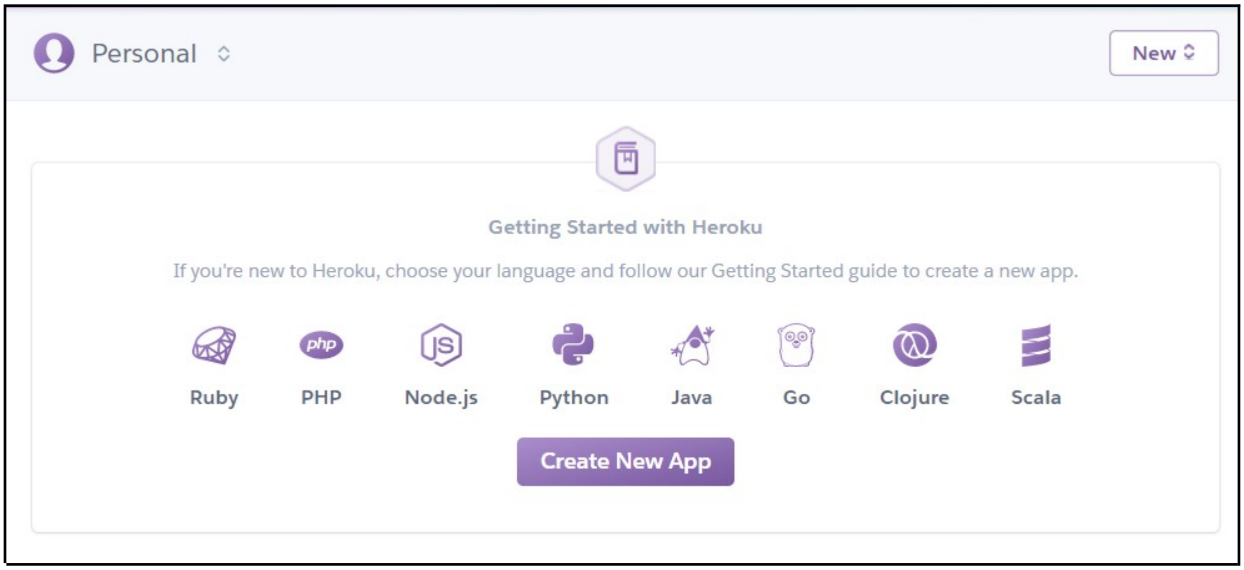
For more information on deploying Flask apps visit http://flask.pocoo.org/docs/0.12/deploying/
For more on Heroku visit https://devcenter.heroku.com/articles/getting-started-with-python#introduction

STEP 1 - Install Heroku and Git

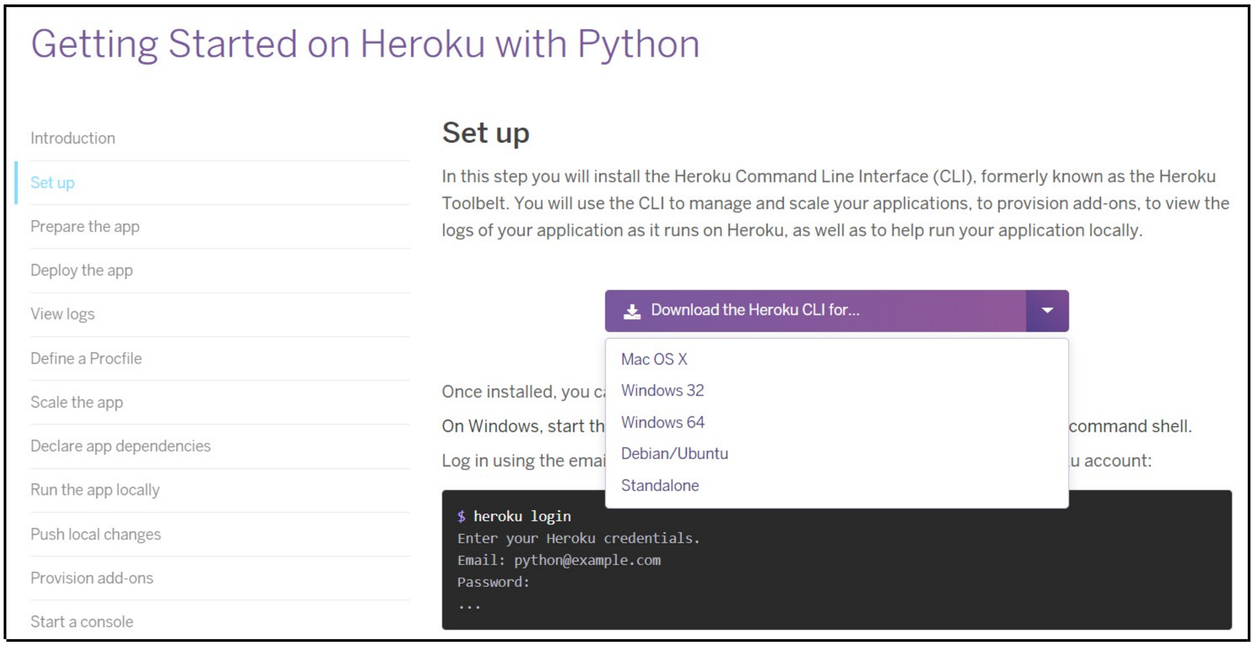
Heroku is a cloud platform that lets users deploy apps on the web.
Git is a version control system that will let you keep a local copy of your app for development, and enable you to push changes from your development copy to the deployed version stored at Heroku.

1. Open a **Heroku** account. Free accounts are available at https://signup.heroku.com/dc
   Follow the instructions to obtain a username and password. Write them down!

2. Log into your Heroku account. It should take you to https://dashboard.heroku.com/apps



3. Click on **Python**. On the next screen select **Set Up**. An option should appear to download the **Heroku Command Line Interface (CLI)**. Choose your operating system from the dropdown list and follow the instructions to install the utility. You should have the option to install **Git** as well.



4. If **git** was *not* installed with Heroku CLI, you can download it directly from https://git-scm.com/downloads and follow the instructions for your operating system.

STEP 2 - Install virtualenv

5. Install **virtualenv** if you don't already have it by typing `pip install virtualenv` at your terminal. Virtualenv allows you to create virtual environments for your app that house Python and all the dependencies your app requires. This includes specific version of plotly, dash, and other libraries that you know will work. As new updates become available, they won't break your app until you've had a chance to test them first!

STEP 3 - Create a Development Folder

6. Create a new folder for your project. This will house the "development" copy of your app:

```
C:\>mkdir my_dash_app
C:\>cd my_dash_app
```

STEP 4 - Initialize Git

7. Initialize an empty git repository:

```
C:\my_dash_app>git init
Initialized empty Git repository in C:/my_dash_app/.git/
```

STEP 5 (WINDOWS) - Create, Activate and Populate a virtualenv

see below for macOS/Linux instructions!

8. Create a virtual environment. We're calling ours "venv" but you can use any name you want:
```
C:\my_dash_app>python -m virtualenv venv
```

9. Activate the virtual environment:
```
C:\my_dash_app>.\venv\Scripts\activate
```

10. Install dash and any desired dependencies into your virtual environment
```
(venv) C:\my_dash_app>pip install dash
(venv) C:\my_dash_app>pip install dash-auth
(venv) C:\my_dash_app>pip install dash-renderer
(venv) C:\my_dash_app>pip install dash-core-components
(venv) C:\my_dash_app>pip install dash-html-components
(venv) C:\my_dash_app>pip install plotly
```
(requirement may be satisfied, see below)

At the time of this writing, `pip install dash` installs:
```
Flask-0.12.2 Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1 certifi-2018.1.18
chardet-3.0.4 click-6.7 dash-0.21.0 decorator-4.2.1 flask-compress-1.4.0
idna-2.6 ipython-genutils-0.2.0 itsdangerous-0.24 jsonschema-2.6.0 jupyter-
core-4.4.0 nbformat-4.4.0 plotly-2.5.1 pytz-2018.4 requests-2.18.4 six-
1.11.0 traitlets-4.3.2 urllib3-1.22
```

11. Install a new dependency **gunicorn** for deploying the app:
```
(venv) C:\my_dash_app>pip install gunicorn
```

STEP 5 (macOS/Linux) - Create, Activate and Populate a virtualenv

8. Create a virtual environment. We're calling ours "venv" but you can use any name you want:
```
$ python3 -m python3 -m virtualenv venv
```

9. Activate the virtual environment:
```
$ source venv/bin/activate
```

10. Install dash and any desired dependencies into your virtual environment
```
$ pip install dash
$ pip install dash-auth
$ pip install dash-renderer
$ pip install dash-core-components
$ pip install dash-html-components
$ pip install plotly
```
(requirement may be satisfied, see above)

11. Install a new dependency gunicorn for deploying the app:
```
$ pip install gunicorn
```

STEP 6 - Add Files to the Development Folder

The following files need to be added:

app1.py	a Dash application
.gitignore	used by git, identifies files that *won't* be pushed to production
Procfile	used for deployment
requirements.txt	describes your Python dependencies, can be created automatically

app1.py

Copy the file used in the Basic Authorization section (or any file you'd like to deploy) and add the following code, shown in bold:

```
import dash
import dash_auth
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

USERNAME_PASSWORD_PAIRS = [
    ['JamesBond', '007'],['LouisArmstrong', 'satchmo']
]

app = dash.Dash()
auth = dash_auth.BasicAuth(app,USERNAME_PASSWORD_PAIRS)
server = app.server

app.layout = html.Div([
    dcc.RangeSlider(
        id='range-slider',
        min=-5,
        max=6,
        marks={i:str(i) for i in range(-5, 7)},
        value=[-3, 4]
    ),
    html.H1(id='product')  # this is the output
], style={'width':'50%'})

@app.callback(
    Output('product', 'children'),
    [Input('range-slider', 'value')])
def update_value(value_list):
    return value_list[0]*value_list[1]

if __name__ == '__main__':
    app.run_server()
```

.gitignore

```
venv
*.pyc
.DS_Store
.env
```

Procfile

```
web: gunicorn app1:server
```

`app1` refers to the filename of our application (app1.py) and `server` refers to the variable **server** inside that file.

requirements.txt

This can be automatically generated by running `pip freeze > requirements.txt` at the terminal. Make sure to do it from inside the development folder with the virtual environment activated.

```
(venv) C:\my_dash_app>pip freeze > requirements.txt
```

Results in a file that looks something like this:

```
certifi==2018.1.18
chardet==3.0.4
click==6.7
dash==0.21.0
dash-auth==0.1.0
dash-core-components==0.22.1
dash-html-components==0.10.0
dash-renderer==0.12.1
decorator==4.2.1
Flask==0.12.2
Flask-Compress==1.4.0
Flask-SeaSurf==0.2.2
gunicorn==19.7.1
idna==2.6
ipython-genutils==0.2.0
itsdangerous==0.24
Jinja2==2.10
jsonschema==2.6.0
jupyter-core==4.4.0
MarkupSafe==1.0
nbformat==4.4.0
plotly==2.5.1
pytz==2018.4
requests==2.18.4
retrying==1.3.3
six==1.11.0
traitlets==4.3.2
urllib3==1.22
Werkzeug==0.14.1
```

STEP 6 - Log onto your Heroku Account

At the terminal, login using the credentials you established in **STEP1**:

```
(venv) C:\my_dash_app>heroku login
Enter your Heroku credentials:
Email: my.name@somewhere.com
Password: ********
Logged in as my.name@somewhere.com
```

STEP 7 - Initialize Heroku, add files to Git, and Deploy

```
(venv) C:\my_dash_app>heroku create my-dash-app
```

You have to change ***my-dash-app*** to a unique name. The name must start with a letter and can only contain lowercase letters, numbers, and dashes.

```
(venv) C:\my_dash_app>git add .
```

Note the period at the end. This adds all files to git (except those listed in .gitignore)

```
(venv) C:\my_dash_app>git commit -m "Initial launch"
```

Every git commit should include a brief descriptive comment. Depending on your operating system, this comment may require double-quotes (not single-quotes).

```
(venv) C:\my_dash_app>git push heroku master
```

This deploys your current code to Heroku. The first time you push may take awhile as it has to set up Python and all your dependencies on the remote server.

```
(venv) C:\my_dash_app>heroku ps:scale web=1
Scaling dynos... done, now running web at 1:Free
```

This runs the app with a 1 heroku "dyno"

STEP 8 - Visit Your App on the Web!

You should be able to view your app at [https://**my-dash-app**.herokuapp.com](https://my-dash-app.herokuapp.com)
(changing **my-dash-app** to the name of your app)

STEP 9 - Update Your App

Any time you make changes to your app, add new apps to your repo, or install new libraries and/or upgrade existing dependencies in your virtual environment, you want to push the latest updates to Heroku. These are the basic steps:

If installing a new package:

```
$ pip install newdependency
$ pip freeze > requirements.txt
```

If updating an existing package:

```
$ pip install dependency --upgrade
$ pip freeze > requirements.txt
```

In all cases:

```
$ git status # view the changes (optional)
$ git add .  # add all the changes
$ git commit -m "a description of the changes"
$ git push heroku master
```

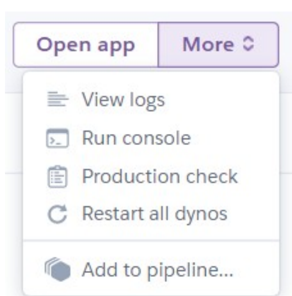
TROUBLESHOOTING

If your app won't launch on Heroku, follow this checklist:

☐ **app1.py** includes **server = app:server**
If not, add this line, save the file, then run Git add/commit/push

☐ **gunicorn** installed, and included in **requirements.txt**
If not, run **pip install gunicorn**, then **pip freeze > requirements.txt**, then run Git add/commit/push

☐ If unable to trace locally, visit your Heroku dashboard and click on **More** / **View logs**



Resources: https://dash.plot.ly/deployment

APPENDIX I - EXAMPLES CODE:

Plotly Basics

Plotly Basics Overview

basic1.py

```
#######
# This script creates a static matplotlib plot
######
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# create fake data:
df = pd.DataFrame(np.random.randn(100,4),columns='A B C D'.split())
df.plot()
plt.show()

#######
# At the terminal run:  python basic1.py
# Close the plot window to close the script
######
```

basic2.py

```
#######
# This script creates the same type of plot as basic1.py,
# but in Plotly. Note that it creates an .html file!
######
import numpy as np
import pandas as pd
import plotly.offline as pyo
import plotly.graph_objs as go

# create fake data:
df = pd.DataFrame(np.random.randn(100,4),columns='A B C D'.split())
pyo.plot([{
    'x': df.index,
    'y': df[col],
    'name': col
} for col in df.columns])
```

Scatter Plots

scatter1.py

```
#######
# This plots 100 random data points (set the seed to 42 to
# obtain the same points we do!) between 1 and 100 in both
# vertical and horizontal directions.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
)]

pyo.plot(data, filename='scatter1.html')
```

scatter2.py

```
#######
# This plots 100 random data points (set the seed to 42 to
# obtain the same points we do!) between 1 and 100 in both
# vertical and horizontal directions.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
)]
layout = go.Layout(
    title = 'Random Data Scatterplot', # Graph title
    xaxis = dict(title = 'Some random x-values'), # x-axis label
    yaxis = dict(title = 'Some random y-values'), # y-axis label
    hovermode ='closest' # handles multiple points landing on the same vertical
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='scatter2.html')
```

scatter3.py

```
#######
# This plots 100 random data points (set the seed to 42 to
# obtain the same points we do!) between 1 and 100 in both
# vertical and horizontal directions.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
    marker = dict(      # change the marker style
        size = 12,
        color = 'rgb(51,204,153)',
        symbol = 'pentagon',
        line = dict(
            width = 2,
        )
    )
)]
layout = go.Layout(
    title = 'Random Data Scatterplot', # Graph title
    xaxis = dict(title = 'Some random x-values'), # x-axis label
    yaxis = dict(title = 'Some random y-values'), # y-axis label
    hovermode ='closest' # handles multiple points landing on the same vertical
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='scatter3.html')
```

Line Charts

line1.py

```
#######
# This line chart displays the same data
# three different ways along the y-axis.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

np.random.seed(56)
x_values = np.linspace(0, 1, 100) # 100 evenly spaced values
y_values = np.random.randn(100)   # 100 random values

# create traces
trace0 = go.Scatter(
    x = x_values,
    y = y_values+5,
    mode = 'markers',
    name = 'markers'
)
trace1 = go.Scatter(
    x = x_values,
    y = y_values,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = x_values,
    y = y_values-5,
    mode = 'lines',
    name = 'lines'
)
data = [trace0, trace1, trace2]  # assign traces to data
layout = go.Layout(
    title = 'Line chart showing three different modes'
)
fig = go.Figure(data=data,layout=layout)
pyo.plot(fig, filename='line1.html')
```

line2.py

```
#######
# This line chart shows U.S. Census Bureau
# population data from six New England states.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# read a .csv file into a pandas DataFrame:
df = pd.read_csv('../data/population.csv', index_col=0)

# create traces
traces = [go.Scatter(
    x = df.columns,
    y = df.loc[name],
    mode = 'markers+lines',
    name = name
) for name in df.index]

layout = go.Layout(
    title = 'Population Estimates of the Six New England States'
)

fig = go.Figure(data=traces,layout=layout)
pyo.plot(fig, filename='line2.html')
```

line3.py

```
#######
# This line chart shows U.S. Census Bureau
# population data from six New England states.
# THIS PLOT USES PANDAS TO EXTRACT DESIRED DATA FROM THE SOURCE
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../sourcedata/nst-est2017-alldata.csv')
# Alternatively:
# df = pd.read_csv('https://www2.census.gov/programs-surveys/popest/datasets/
2010-2017/national/totals/nst-est2017-alldata.csv')

# grab just the six New England states:
df2 = df[df['DIVISION']=='1']
# set the index to state name:
df2.set_index('NAME', inplace=True)
# grab just the population columns:
df2 = df2[[col for col in df2.columns if col.startswith('POP')]]

traces=[go.Scatter(
    x = df2.columns,
    y = df2.loc[name],
    mode = 'markers+lines',
    name = name
) for name in df2.index]

layout = go.Layout(
    title = 'Population Estimates of the Six New England States'
)

fig = go.Figure(data=traces,layout=layout)
pyo.plot(fig, filename='line3.html')
```

Bar Charts

bar1.py

```
#######
# A basic bar chart showing the total number of
# 2018 Winter Olympics Medals won by Country.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# read a .csv file into a pandas DataFrame:
df = pd.read_csv('../data/2018WinterOlympics.csv')

data = [go.Bar(
    x=df['NOC'],  # NOC stands for National Olympic Committee
    y=df['Total']
)]
layout = go.Layout(
    title='2018 Winter Olympic Medals by Country'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bar1.html')
```

bar2.py

```
#######
# This is a grouped bar chart showing three traces
# (gold, silver and bronze medals won) for each country
# that competed in the 2018 Winter Olympics.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2018WinterOlympics.csv')

trace1 = go.Bar(
    x=df['NOC'],  # NOC stands for National Olympic Committee
    y=df['Gold'],
    name = 'Gold',
    marker=dict(color='#FFD700') # set the marker color to gold
)
trace2 = go.Bar(
    x=df['NOC'],
    y=df['Silver'],
    name='Silver',
    marker=dict(color='#9EA0A1') # set the marker color to silver
)
trace3 = go.Bar(
    x=df['NOC'],
    y=df['Bronze'],
    name='Bronze',
    marker=dict(color='#CD7F32') # set the marker color to bronze
)
data = [trace1, trace2, trace3]
layout = go.Layout(
    title='2018 Winter Olympic Medals by Country'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bar2.html')
```

bar3.py

```
#######
# This is a stacked bar chart showing three traces
# (gold, silver and bronze medals won) for each country
# that competed in the 2018 Winter Olympics.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2018WinterOlympics.csv')

trace1 = go.Bar(
    x=df['NOC'],  # NOC stands for National Olympic Committee
    y=df['Gold'],
    name = 'Gold',
    marker=dict(color='#FFD700') # set the marker color to gold
)
trace2 = go.Bar(
    x=df['NOC'],
    y=df['Silver'],
    name='Silver',
    marker=dict(color='#9EA0A1') # set the marker color to silver
)
trace3 = go.Bar(
    x=df['NOC'],
    y=df['Bronze'],
    name='Bronze',
    marker=dict(color='#CD7F32') # set the marker color to bronze
)
data = [trace1, trace2, trace3]
layout = go.Layout(
    title='2018 Winter Olympic Medals by Country',
    barmode='stack'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bar3.html')
```

Bubble Charts

bubble1.py

```
#######
# A bubble chart is simply a scatter plot
# with the added feature that the size of the
# marker can be set by the data.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Scatter(              # start with a normal scatter plot
            x=df['horsepower'],
            y=df['mpg'],
            text=df['name'],
            mode='markers',
            marker=dict(size=1.5*df['cylinders']) # set the marker size
    )]

layout = go.Layout(
    title='Vehicle mpg vs. horsepower',
    hovermode='closest'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bubble1.html')
```

bubble2.py

```
#######
# A bubble chart is simply a scatter plot
# with the added feature that the size of the
# marker can be set by the data.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

# Add columns to the DataFrame to convert model year to a string and
# then combine it with name so that hover text shows both:
df['text1']=pd.Series(df['model_year'],dtype=str)
df['text2']="'"+df['text1']+" "+df['name']

data = [go.Scatter(
            x=df['horsepower'],
            y=df['mpg'],
            text=df['text2'],  # use the new column for the hover text
            mode='markers',
            marker=dict(size=1.5*df['cylinders'])
    )]
layout = go.Layout(
    title='Vehicle mpg vs. horsepower',
    hovermode='closest'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='bubble2.html')
```

Box Plots

box1.py

```
#######
# This simple box plot places the box beside
# the original data points on the same graph.
######
import plotly.offline as pyo
import plotly.graph_objs as go

# set up an array of 20 data points, with 20 as the median value
y = [1,14,14,15,16,18,18,19,19,20,20,23,24,26,27,27,28,29,33,54]

data = [
    go.Box(
        y=y,
        boxpoints='all', # display the original data points
        jitter=0.3,      # spread them out so they all appear
        pointpos=-1.8    # offset them to the left of the box
    )
]
pyo.plot(data, filename='box1.html')
```

box2.py

```
#######
# This simple box plot displays outliers
# above and below the box.
######
import plotly.offline as pyo
import plotly.graph_objs as go

# set up an array of 20 data points, with 20 as the median value
y = [1,14,14,15,16,18,18,19,19,20,20,23,24,26,27,27,28,29,33,54]

data = [
    go.Box(
        y=y,
        boxpoints='outliers' # display only outlying data points
    )
]
pyo.plot(data, filename='box2.html')
```

box3.py

```
#######
# This plot compares sample distributions
# of three-letter-words in the works of
# Quintus Curtius Snodgrass and Mark Twain
######
import plotly.offline as pyo
import plotly.graph_objs as go

snodgrass = [.209,.205,.196,.210,.202,.207,.224,.223,.220,.201]
twain = [.225,.262,.217,.240,.230,.229,.235,.217]

data = [
    go.Box(
        y=snodgrass,
        name='QCS'
    ),
    go.Box(
        y=twain,
        name='MT'
    )
]
layout = go.Layout(
    title = 'Comparison of three-letter-word frequencies<br>\
    between Quintus Curtius Snodgrass and Mark Twain'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='box3.html')
```

Histograms

hist1.py

```
#######
# This histogram looks back at the mpg dataset
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Histogram(
    x=df['mpg']
)]

layout = go.Layout(
    title="Miles per Gallon Frequencies of<br>\
    1970's Era Vehicles"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='basic_histogram.html')
```

hist2.py

```
#######
# This histogram has wider bins than the previous hist1.py
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Histogram(
    x=df['mpg'],
    xbins=dict(start=8,end=50,size=6),
)]

layout = go.Layout(
    title="Miles per Gallon Frequencies of<br>\
    1970's Era Vehicles"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='wide_histogram.html')
```

hist3.py

```
#######
# This histogram has narrower bins than the previous hist1.py
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/mpg.csv')

data = [go.Histogram(
    x=df['mpg'],
    xbins=dict(start=8,end=50,size=1),
)]

layout = go.Layout(
    title="Miles per Gallon Frequencies of<br>\
    1970's Era Vehicles"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='narrow_histogram.html')
```

hist4.py

```
#######
# This histogram displays the number of Reddit button presses
# over the two months of their social experiment.
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/thebutton_presses.csv')

data = [go.Histogram(
    x=df['press time']
)]

layout = go.Layout(
    title="Number of presses per timeslot"
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='button_presses.html')
```

histBONUS.py

```
#######
# This bar chart mimics a histogram as the x-axis
# is a continuous time series, and the y-axis sums
# a frequency that is already part of the dataset
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/FremontBridgeBicycles.csv')

# Convert the "Date" text column to a Datetime series:
df['Date'] = pd.to_datetime(df['Date'])

# Add a column to hold the hour:
df['Hour']=df['Date'].dt.time

# Let pandas perform the aggregation
df2 = df.groupby('Hour').sum()

trace1 = go.Bar(
    x=df2.index,
    y=df2['Fremont Bridge West Sidewalk'],
    name="Southbound",
    width=1  # eliminates space between adjacent bars
)
trace2 = go.Bar(
    x=df2.index,
    y=df2['Fremont Bridge East Sidewalk'],
    name="Northbound",
    width=1
)
data = [trace1, trace2]

layout = go.Layout(
    title='Fremont Bridge Bicycle Traffic by Hour',
    barmode='stack'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='fremont_bridge.html')
```

Distplots

dist1.py

```
#######
# This distplot uses plotly's Figure Factory
# module in place of Graph Objects
######
import plotly.offline as pyo
import plotly.figure_factory as ff
import numpy as np

x = np.random.randn(1000)
hist_data = [x]
group_labels = ['distplot']

fig = ff.create_distplot(hist_data, group_labels)
pyo.plot(fig, filename='basic_distplot.html')
```

dist2.py

```
#######
# This distplot demonstrates that random samples
# seldom fit a "normal" distribution.
######
import plotly.offline as pyo
import plotly.figure_factory as ff
import numpy as np

x1 = np.random.randn(200)-2
x2 = np.random.randn(200)
x3 = np.random.randn(200)+2
x4 = np.random.randn(200)+4

hist_data = [x1,x2,x3,x4]
group_labels = ['Group1','Group2','Group3','Group4']

fig = ff.create_distplot(hist_data, group_labels)
pyo.plot(fig, filename='multiset_distplot.html')
```

dist3.py

```
#######
# This distplot looks back at the Mark Twain/
# Quintus Curtius Snodgrass data and tries
# to compare them.
######
import plotly.offline as pyo
import plotly.figure_factory as ff

snodgrass = [.209,.205,.196,.210,.202,.207,.224,.223,.220,.201]
twain = [.225,.262,.217,.240,.230,.229,.235,.217]

hist_data = [snodgrass,twain]
group_labels = ['Snodgrass','Twain']

fig = ff.create_distplot(hist_data, group_labels, bin_size=[.005,.005])
pyo.plot(fig, filename='SnodgrassTwainDistplot.html')
```

Heatmaps

heat1.py

```
#######
# Heatmap of temperatures for Santa Barbara, California
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2010SantaBarbaraCA.csv')

data = [go.Heatmap(
    x=df['DAY'],
    y=df['LST_TIME'],
    z=df['T_HR_AVG'].values.tolist(),
    colorscale='Jet'
)]

layout = go.Layout(
    title='Hourly Temperatures, June 1-7, 2010 in<br>\
    Santa Barbara, CA USA'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='Santa_Barbara.html')
```

heat2.py

```
#######
# Heatmap of temperatures for Yuma, Arizona
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2010YumaAZ.csv')

data = [go.Heatmap(
    x=df['DAY'],
    y=df['LST_TIME'],
    z=df['T_HR_AVG'].values.tolist(),
    colorscale='Jet'
)]

layout = go.Layout(
    title='Hourly Temperatures, June 1-7, 2010 in<br>\
    Yuma, AZ USA'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='Yuma.html')
```

heat3.py

```
#######
# Heatmap of temperatures for Sitka, Alaska
######
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

df = pd.read_csv('../data/2010SitkaAK.csv')

data = [go.Heatmap(
    x=df['DAY'],
    y=df['LST_TIME'],
    z=df['T_HR_AVG'].values.tolist(),
    colorscale='Jet'
)]

layout = go.Layout(
    title='Hourly Temperatures, June 1-7, 2010 in<br>\
    Sitka, AK USA'
)
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='Sitka.html')
```

heat4.py

```
#######
# Side-by-side heatmaps for Sitka, Alaska,
# Santa Barbara, California and Yuma, Arizona
# using a shared temperature scale.
######
import plotly.offline as pyo
import plotly.graph_objs as go
from plotly import tools
import pandas as pd

df1 = pd.read_csv('../data/2010SitkaAK.csv')
df2 = pd.read_csv('../data/2010SantaBarbaraCA.csv')
df3 = pd.read_csv('../data/2010YumaAZ.csv')

trace1 = go.Heatmap(
    x=df1['DAY'],
    y=df1['LST_TIME'],
    z=df1['T_HR_AVG'].values.tolist(),
    colorscale='Jet',
    zmin = 5, zmax = 40 # add max/min color values to make each plot
consistent
)
trace2 = go.Heatmap(
    x=df2['DAY'],
    y=df2['LST_TIME'],
    z=df2['T_HR_AVG'].values.tolist(),
    colorscale='Jet',
    zmin = 5, zmax = 40
)
trace3 = go.Heatmap(
    x=df3['DAY'],
    y=df3['LST_TIME'],
    z=df3['T_HR_AVG'].values.tolist(),
    colorscale='Jet',
    zmin = 5, zmax = 40
)

fig = tools.make_subplots(rows=1, cols=3,
    subplot_titles=('Sitka, AK','Santa Barbara, CA', 'Yuma, AZ'),
    shared_yaxes = True,  # this makes the hours appear only on the left
)
fig.append_trace(trace1, 1, 1)
fig.append_trace(trace2, 1, 2)
fig.append_trace(trace3, 1, 3)

fig['layout'].update(      # access the layout directly!
    title='Hourly Temperatures, June 1-7, 2010'
)
pyo.plot(fig, filename='AllThree.html')
```

Plotly Basics Exercise Solutions

[Return to Topic]

Sol1-Scatterplot.py

```
#######
# Objective: Create a scatterplot of 1000 random data points.
# x-axis values should come from a normal distribution using
# np.random.randn(1000)
# y-axis values should come from a uniform distribution over [0,1) using
# np.random.rand(1000)
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

# obtain x and y values:
```

```
random_x = np.random.randn(1000) # normal distribution
random_y = np.random.rand(1000)  # uniform distribution

# define a data variable
data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
)]

# define the layout, and include a title and axis labels
layout = go.Layout(
    title = 'Random Data Scatterplot',
    xaxis = dict(title = 'Normal distribution'),
    yaxis = dict(title = 'Uniform distribution'),
    hovermode ='closest'
)

# Create a fig from data and layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution1.html')
```

A Note About the Line Chart Exercise:

By itself, a for loop won't work as expected! The code

```
data = []
for day in df['DAY']:
    trace = go.Scatter(x=df['LST_TIME'],
                       y=df[df['DAY']==day]['T_HR_AVG'],
                       mode='lines',
                       name=day)
    data.append(trace)
```

has each *row* as its own trace, not each day.

Sol2a-Linechart.py uses hardcoded values:

```
data = []
days = ['TUESDAY','WEDNESDAY','THURSDAY','FRIDAY','SATURDAY','SUNDAY','MONDAY']
for day in days:
    trace = go.Scatter(x=df['LST_TIME'],
                       y=df[df['DAY']==day]['T_HR_AVG'],
                       mode='lines',
                       name=day)
    data.append(trace)
```

...but this is not an ideal solution!

Sol2b-Linechart.py lets Pandas filter the df['DAY'] field:

```
data = []
for day in df['DAY'].unique():
    trace = go.Scatter(x=df['LST_TIME'],
                       y=df[df['DAY']==day]['T_HR_AVG'],
                       mode='lines',
                       name=day)
    data.append(trace)
```

This works!

Sol2a-Linechart.py

```python
#######
# Objective: Using the file 2010YumaAZ.csv, develop a Line Chart
# that plots seven days worth of temperature data on one graph.
# You can use a list comprehension to assign each day to its own trace.
######
# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# Create a pandas DataFrame from 2010YumaAZ.csv
df = pd.read_csv('../data/2010YumaAZ.csv')
days = ['TUESDAY','WEDNESDAY','THURSDAY','FRIDAY','SATURDAY','SUNDAY','MONDAY']

# Use a for loop to create the traces for the seven days
# There are many ways to do this!

data = []

for day in days:
    trace = go.Scatter(x=df['LST_TIME'],
                       y=df[df['DAY']==day]['T_HR_AVG'],
                       mode='lines',
                       name=day)
    data.append(trace)

# Define the layout
layout = go.Layout(
    title='Daily temperatures from June 1-7, 2010 in Yuma, Arizona',
    hovermode='closest'
)

# Create a fig from data and layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution2a.html')
```

Sol2b-Linechart.py

```
#####################
## NOTE: ADVANCED SOLUTION THAT USES ONLY PURE DF CALLS
## THIS IS FOR MORE ADVANCED PANDAS USERS TO TAKE A LOOK AT! :)


#######
# Objective: Using the file 2010YumaAZ.csv, develop a Line Chart
# that plots seven days worth of temperature data on one graph.
# You can use a list comprehension to assign each day to its own trace.
######
# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# Create a pandas DataFrame from 2010YumaAZ.csv
df = pd.read_csv('../data/2010YumaAZ.csv')


# Define a data variable
data = [{
    'x': df['LST_TIME'],
    'y': df[df['DAY']==day]['T_HR_AVG'],
    'name': day
} for day in df['DAY'].unique()]

# Define the layout
layout = go.Layout(
    title='Daily temperatures from June 1-7, 2010 in Yuma, Arizona',
    hovermode='closest'
)

# Create a fig from data and layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution2b.html')
```

Sol3a-Barchart.py

```
#######
# Objective: Create a stacked bar chart from
# the file ../data/mocksurvey.csv. Note that questions appear in
# the index (and should be used for the x-axis), while responses
# appear as column labels.  Extra Credit: make a horizontal bar chart!
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# create a DataFrame from the .csv file:
df = pd.read_csv('../data/mocksurvey.csv',index_col=0)

# create traces using a list comprehension:
data = [go.Bar(
    x = df.index,
    y = df[response],
    name=response
) for response in df.columns]

# create a layout, remember to set the barmode here
layout = go.Layout(
    title='Mock Survey Results',
    barmode='stack'
)

# create a fig from data & layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution3a.html')
```

Sol3b-Barchart.py

```
#######
# Objective: Create a stacked bar chart from
# the file ../data/mocksurvey.csv. Note that questions appear in
# the index (and should be used for the x-axis), while responses
# appear as column labels.  Extra Credit: make a horizontal bar chart!
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# create a DataFrame from the .csv file:
df = pd.read_csv('../data/mocksurvey.csv',index_col=0)

# create traces using a list comprehension:
data = [go.Bar(
    y = df.index,     # reverse your x- and y-axis assignments
    x = df[response],
    orientation='h',  # this line makes it horizontal!
    name=response
) for response in df.columns]

# create a layout, remember to set the barmode here
layout = go.Layout(
    title='Mock Survey Results',
    barmode='stack'
)

# create a fig from data & layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution3a.html')
```

Sol4-Bubblechart.py

```
#######
# Objective: Create a bubble chart that compares three other features
# from the mpg.csv dataset. Fields include: 'mpg', 'cylinders', 'displacement'
# 'horsepower', 'weight', 'acceleration', 'model_year', 'origin', 'name'
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# create a DataFrame from the .csv file:
df = pd.read_csv('../data/mpg.csv')

# create data by choosing fields for x, y and marker size attributes
data = [go.Scatter(
    x=df['displacement'],
    y=df['acceleration'],
    text=df['name'],
    mode='markers',
    marker=dict(size=df['weight']/500)
)]

# create a layout with a title and axis labels
layout = go.Layout(
    title='Vehicle acceleration vs. displacement',
    xaxis = dict(title = 'displacement'),
    yaxis = dict(title = 'acceleration = seconds to reach 60mph'),
    hovermode='closest'
)

# create a fig from data & layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution4.html')
#######
# So what happened?? Why is the trend sloping downward?
# Remember that acceleration is the number of seconds to go from 0 to 60mph,
# so fewer seconds means faster acceleration!
######
```

Sol5-Boxplot.py

```
#######
# Objective: Make a DataFrame using the Abalone dataset (../data/abalone.csv).
# Take two independent random samples of different sizes from the 'rings' field.
# HINT: np.random.choice(df['rings'],10,replace=False) takes 10 random values
# Use box plots to show that the samples do derive from the same population.
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np
import pandas as pd

# create a DataFrame from the .csv file:
df = pd.read_csv('../data/abalone.csv')

# take two random samples of different sizes:
a = np.random.choice(df['rings'],30,replace=False)
b = np.random.choice(df['rings'],100,replace=False)

# create a data variable with two Box plots:
data = [
    go.Box(
        y=a,
        name='A'
    ),
    go.Box(
        y=b,
        name='B'
    )
]

# add a layout
layout = go.Layout(
    title = 'Comparison of two samples taken from the same population'
)

# create a fig from data & layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution5.html')
```

Sol6-Histogram.py

```
#######
# Objective: Create a histogram that plots the 'length' field
# from the Abalone dataset (../data/abalone.csv).
# Set the range from 0 to 1, with a bin size of 0.02
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

# create a DataFrame from the .csv file:
df = pd.read_csv('../data/abalone.csv')

# create a data variable:
data = [go.Histogram(
    x=df['length'],
    xbins=dict(start=0,end=1,size=.02),
)]

# add a layout
layout = go.Layout(
    title="Shell lengths from the Abalone dataset"
)

# create a fig from data & layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution6.html')
```

Sol7-Distplot.py

```
#######
# Objective: Using the iris dataset, develop a Distplot
# that compares the petal lengths of each class.
# File: '../data/iris.csv'
# Fields: 'sepal_length','sepal_width','petal_length','petal_width','class'
# Classes: 'Iris-setosa','Iris-versicolor','Iris-virginica'
######

# Perform imports here:
import plotly.offline as pyo
import plotly.figure_factory as ff
import pandas as pd

# create a DataFrame from the .csv file:
df = pd.read_csv('../data/iris.csv')

# Define the traces
trace0 = pd.DataFrame(df[df['class']=='Iris-setosa'])['petal_length']
trace1 = pd.DataFrame(df[df['class']=='Iris-versicolor'])['petal_length']
trace2 = pd.DataFrame(df[df['class']=='Iris-virginica'])['petal_length']

# Define a data variable
hist_data = [trace0, trace1, trace2]
group_labels = ['Iris Setosa','Iris Versicolor','Iris Virginica']

# Create a fig from data and layout, and plot the fig
fig = ff.create_distplot(hist_data, group_labels)
pyo.plot(fig, filename='solution7.html')

########
# Great! This shows that if given a flower with a petal length
# between 1-2cm, it is almost certainly an Iris Setosa!
######
```

Sol8-Heatmap.py

```python
#######
# Objective: Using the "flights" dataset available from Python's
# Seaborn module (see https://seaborn.pydata.org/generated/seaborn.heatmap.html)
# create a heatmap with the following parameters:
# x-axis="year"
# y-axis="month"
# z-axis(color)="passengers"
######

# Perform imports here:
import plotly.offline as pyo
import plotly.graph_objs as go

# Create a DataFrame from Seaborn "flights" data
import seaborn as sns
df = sns.load_dataset("flights")

# Define a data variable
data = [go.Heatmap(
    x=df['year'],
    y=df['month'],
    z=df['passengers'].values.tolist()
)]

# Define the layout
layout = go.Layout(
    title='Flights'
)
# Create a fig from data and layout, and plot the fig
fig = go.Figure(data=data, layout=layout)
pyo.plot(fig, filename='solution8.html')

#######
# Excellent! This shows two distinct trends - an increase in
# passengers flying over the years, and a greater number of
# passengers flying in the summer months.
######
```

APPENDIX II – DASH CORE COMPONENTS

https://dash.plot.ly/dash-core-components

A quick run-through of available core components:

Dropdown

```
import dash_core_components as dcc

dcc.Dropdown(
    options=[
    {'label': 'New York City', 'value': 'NYC'},
    {'label': 'Montréal', 'value': 'MTL'},
    {'label': 'San Francisco', 'value': 'SF'}
    ],
    value='MTL'
)
```

Single dropdown list, **value** sets initial displayed entry

```
import dash_core_components as dcc

dcc.Dropdown(
    options=[
    {'label': 'New York City', 'value': 'NYC'},
    {'label': 'Montréal', 'value': 'MTL'},
    {'label': 'San Francisco', 'value': 'SF'}
    ],
    multi=True,
    value="MTL"
)
```

multi permits multiple selection

Slider

```
import dash_core_components as dcc

dcc.Slider(
    min=-5,
    max=10,
    step=0.5,
    value=-3,
)
```

Basic slider

```
import dash_core_components as dcc

dcc.Slider(
    min=0,
    max=9,
    marks={i: 'Label {}'.format(i) for i in range(10)},
    value=5,
)
```

Lets you set labels as Label 0, Label 1, etc.

RangeSlider

```
import dash_core_components as dcc

dcc.RangeSlider(
    count=1,
    min=-5,
    max=10,
    step=0.5,
    value=[-3, 7]
)
```

```
import dash_core_components as dcc

dcc.RangeSlider(
```

```
    marks={i: 'Label {}'.format(i) for i in range(-5, 7)},
    min=-5,
    max=6,
    value=[-3, 4]
)
```

Input

```
import dash_core_components as dcc

dcc.Input(
    placeholder='Enter a value...',
    type='text',
    value=''
)
```

Textarea

```
import dash_core_components as dcc

dcc.Textarea(
    placeholder='Enter a value...',
    value='This is a TextArea component',
    style={'width': '100%'}
)
```

Checklists

```
import dash_core_components as dcc

dcc.Checklist(
    options=[
    {'label': 'New York City', 'value': 'NYC'},
    {'label': 'Montréal', 'value': 'MTL'},
    {'label': 'San Francisco', 'value': 'SF'}
    ],
    values=['MTL', 'SF']
)
```

Vertical list

```
import dash_core_components as dcc

dcc.Checklist(
    options=[
    {'label': 'New York City', 'value': 'NYC'},
    {'label': 'Montréal', 'value': 'MTL'},
    {'label': 'San Francisco', 'value': 'SF'}
    ],
    values=['MTL', 'SF'],
    labelStyle={'display': 'inline-block'}
)
```

Horizontal array

Radio Items

```
import dash_core_components as dcc

dcc.RadioItems(
    options=[
    {'label': 'New York City', 'value': 'NYC'},
    {'label': 'Montréal', 'value': 'MTL'},
    {'label': 'San Francisco', 'value': 'SF'}
    ],
    value='MTL'
)
```

Vertical list

```
import dash_core_components as dcc

dcc.RadioItems(
    options=[
    {'label': 'New York City', 'value': 'NYC'},
    {'label': 'Montréal', 'value': 'MTL'},
    {'label': 'San Francisco', 'value': 'SF'}
    ],
    value='MTL',
    labelStyle={'display': 'inline-block'}
)
```

Horizontal array

Button

```
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output, State

app = dash.Dash()
app.layout = html.Div([
    html.Div(dcc.Input(id='input-box', type='text')),
    html.Button('Submit', id='button'),
    html.Div(id='output-container-button',
        children='Enter a value and press submit')
])

@app.callback(
    Output('output-container-button', 'children'),
    [Input('button', 'n_clicks')],
    [State('input-box', 'value')])
def update_output(n_clicks, value):
    return 'The input value was "{}" and the button has been clicked {} times'.format(
        value,
        n_clicks
    )

if __name__ == '__main__':
    app.run_server(debug=True)
```

For more on **dash.dependencies.State**, see the tutorial on Dash State.

DatePickerSingle

```
import dash_core_components as dcc
from datetime import datetime as dt

dcc.DatePickerSingle(
    id='date-picker-single',
    date=dt(1997, 5, 10)
)
```

DatePickerRange

```
import dash_core_components as dcc
from datetime import datetime as dt

dcc.DatePickerRange(
    id='date-picker-range',
    start_date=dt(1997, 5, 3),
    end_date_placeholder_text='Select a date!'
)
```

Markdown

```
import dash_core_components as dcc

dcc.Markdown('''
#### Dash and Markdown

Dash supports [Markdown](http://commonmark.org/help).

Markdown is a simple way to write and format text.
It includes a syntax for things like **bold text** and *italics*,
[links](http://commonmark.org/help), inline `code` snippets, lists,
quotes, and more.
''')
```

Graphs

The **Graph** component shares the same syntax as the open-source **plotly.py** library. View the plotly.py docs to learn more.

Still in Development

Interactive Tables

The **dash_html_components** library exposes all of the HTML tags. This includes the **Table**, **Tr**, and **Tbody** tags that can be used to create an HTML table. See Create Your First Dash App, Part 1 for an example.
Dash is currently incubating an interactive table component that provides built-in filtering, row-selection, editing, and sorting. Prototypes of this component are being developed in the dash-table-experiments repository. Join the discussion in the Dash Community Forum.

Upload Component

The **dcc.Upload** component allows users to upload files into your app through drag-and-drop or the system's native file explorer.

Tabs

The **dcc.Tabs** component is currently available in the prerelease channel of the **dash-core-components** package. To try it out, see the tab component Pull Request on GitHub.

APPENDIX III - ADDITIONAL RESOURCES

Plotly User Guide for Python

Plotly Python Figure Reference

- Scatter
- ScatterGL
- Bar
- Box
- Pie
- Area
- Heatmap
- Contour
- Histogram
- Histogram 2D
- Histogram 2D Contour
- OHLC
- Candlestick
- Table

3D Charts:

- Scatter3D
- Surface
- Mesh

Maps:

- Scatter Geo
- Choropleth
- Scatter Mapbox

Advanced Charts:

- Carpet
- Scatter Carpet
- Contour Carpet
- Parallel Coordinates
- Scatter Ternary
- Sankey

Dash User Guide

Dash Tutorial

- Part 1 - Installation
- Part 2 - Dash Layout
- Part 3 - Basic Callbacks
- Part 4 - Dash State
- Part 5 - Interactive Graphing and Crossfiltering
- Part 6 - Sharing Data Between Callbacks

Dash HTML Components

Dash Core Components Gallery

- Dropdown
- Slider
- RangeSlider
- Input
- Textarea
- Checklist
- Radio Items
- DatePickerSingle
- DatePickerRange
- Markdown
- **Buttons**
  described in more detail under Dash State
- **Graph**
  described in more detail in the Plotly Python docs