Introduction

In this lab we will build out functions to help us plot visualizations in lessons going forward.

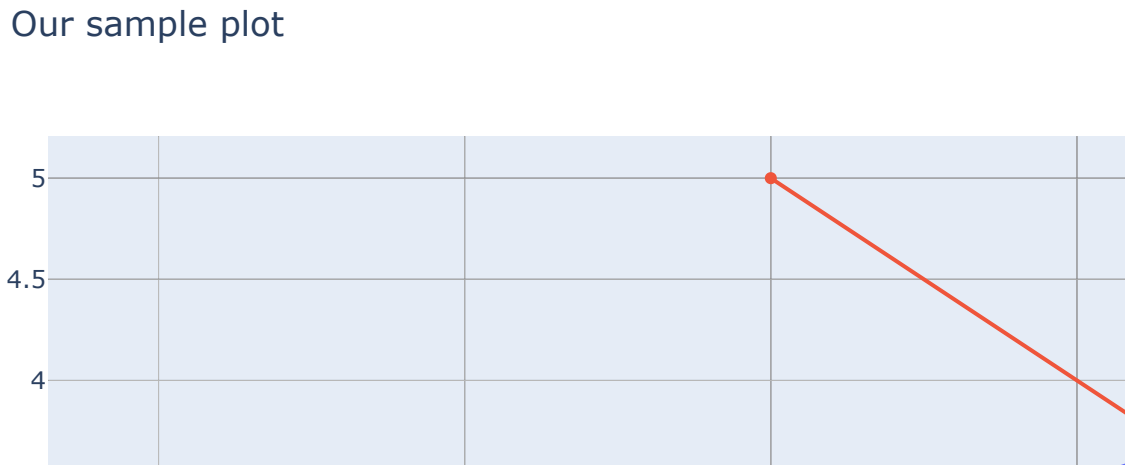
The Setup

Let's start by providing a function called `plot` which plots our data.

In [1]:
```
import plotly
from plotly.offline import iplot, init_notebook_mode

init_notebook_mode(connected=True)

def plot(figure):
    plotly.offline.iplot(figure)
```

To see our plot on the screen, we provide our `plot` function a dictionary. The dictionary has a key of `data` which points to a list of traces. Let's see it!

In [2]:
```
sample_trace = {'x': [1, 2, 3], 'y': [2, 3, 4]}
other_sample_trace = {'x': [2, 3, 4], 'y': [5, 3, 4]}
sample_figure = {'data': [sample_trace, other_sample_trace], 'layout': {'ti
plot(sample_figure)
```



Ok, now that our `plot` function works, we need an easy way to create the following:

- traces
- figures
- layouts

Let's take these one by one. We'll start with a `build_trace` function that easily creates traces.

A function to create traces

build_trace

Write a `build_trace` function that can take in data that comes in the following format:

In [3]:
```
data = [{'x': 1, 'y': 1}, {'x': 3, 'y': 2}, {'x': 2, 'y': 5}]
```

And returns data like the commented out dictionary below:

```
data = [{'x': 1, 'y': 1}, {'x': 3, 'y': 2}, {'x': 2, 'y': 5}]
build_trace(data)
# {'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'dat
a'}
```

So `build_trace` that takes in a list of data points as arguments and returns a dictionary with a key of `x` that points to a list of x values, and a key of `y` that points to a list of y values.

> **Note:** Look at the parameters provided for `build_trace` . The arguments `mode = 'markers'` and `name = 'data'` may seem scary since we haven't seem them before. No need to worry! These are **default arguments**.

> If no argument for `mode` or `name` is provided when we call the `build_trace` function, Python will automatically set these parameters to the value provided, which, in this case would be 'markers' for the mode and 'data' for the name.

In [4]:
```
def build_trace(data, mode = 'markers', name = 'data'):
    trace_dict = dict()
    data_x_list = []
    data_y_list = []
    for i in range(0,len(data)):
        #print("x:"+str(data[i]['x'])+", y:"+str(data[i]['y']))
        data_x_list.append(data[i]['x'])
        data_y_list.append(data[i]['y'])
    #print(data_x_list)
    trace_dict.update({'x':data_x_list,'y':data_y_list,'mode':mode,'name':n
    print(trace_dict)
    return trace_dict
    pass
```

So by default, if we just call `build_trace(data)` without specifying either a mode or a name, the function will automatically set these parameters to 'markers' and 'data' respectively.

In [5]:
```
data = [{'x': 1, 'y': 1}, {'x': 3, 'y': 2}, {'x': 2, 'y': 5}]
build_trace(data)
# {'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'data'}
```

```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'data'}
```

Out[5]:
```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'data'}
```

If we want our `build_trace` function to take a different mode arguement, we add a second argument when we call the function which will overwrite the mode's default argument.

In [6]:
```
build_trace(data, 'scatter')
# {'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'scatter', 'name': 'data'}
```

```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'scatter', 'name': 'data'}
```

Out[6]:
```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'scatter', 'name': 'data'}
```

We could do the same thing with the name of the plot. This is useful for when we have more than one trace in the same plot.

> Order matters. The value passed through as the `name` argument, should correspond to the value of the `name` key in our returned dictionary.

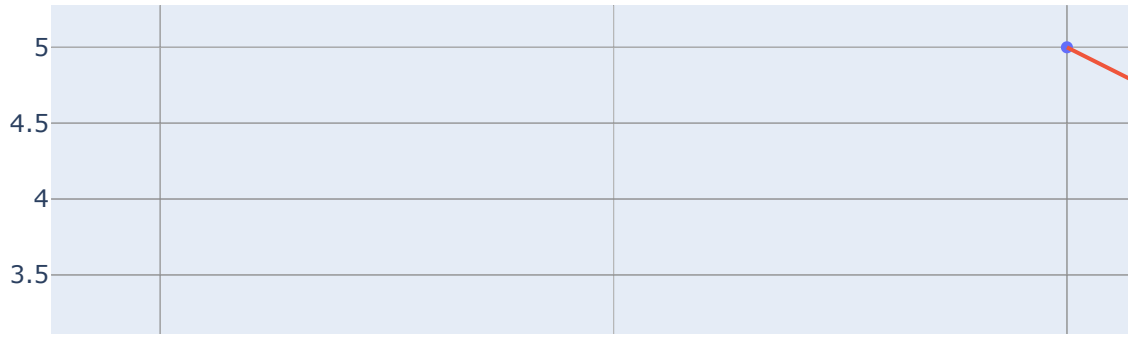
In [7]:
```
build_trace(data, 'markers', 'sample plot')
# {'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'sample plot'
```

```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'sample plo
t'}
```

Out[7]:
```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'sample plo
t'}
```

In [8]:
```
trace0 = build_trace(data)

trace0 = build_trace(data, 'markers')
trace1 = build_trace(data, 'lines', 'my_trace')
plot({'data':[trace0, trace1]})
```

```
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'data'}
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'markers', 'name': 'data'}
{'x': [1, 3, 2], 'y': [1, 2, 5], 'mode': 'lines', 'name': 'my_trace'}
```



trace_values

Now let's write another function to create a trace called `trace_values`. It works just like our `build_trace` function, except that it takes in a list of x_values and a list of y_values and returns our trace dictionary. We will use default argument again here in the same manner as before.

In [9]:
```
def trace_values(x_values, y_values, mode = 'markers', name="data"):
    trace_dict = {}
    trace_dict.update({'x':x_values,'y':y_values,'mode':mode,'name':name})

    return trace_dict
    pass
```

In [10]:
```
trace_values([1, 2, 3], [2, 4, 5])
# {'x': [1, 2, 3], 'y': [2, 4, 5], 'mode': 'markers', 'name': 'data'}
```

Out[10]:
```
{'x': [1, 2, 3], 'y': [2, 4, 5], 'mode': 'markers', 'name': 'data'}
```

Now let's try to build a line trace with our newly defined `trace_values` function. We will set `mode` to 'lines' and the `name` of our trace to 'line trace'.

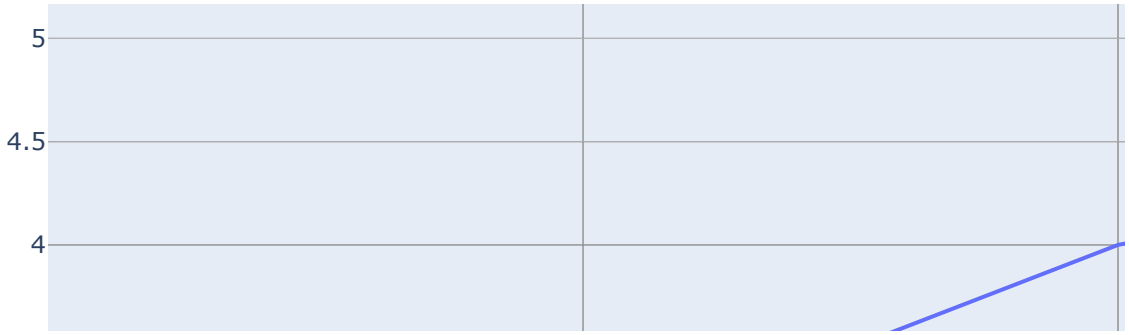
In [11]:
```
trace_values([1, 2, 3], [2, 4, 5], 'lines', 'line trace')
# {'x': [1, 2, 3], 'y': [2, 4, 5], 'mode': 'lines', 'name': 'line trace'}
```

Out[11]:
```
{'x': [1, 2, 3], 'y': [2, 4, 5], 'mode': 'lines', 'name': 'line trace'}
```

From there, we can use our `trace_values` function to plot our chart.

> Uncomment and run the code below

In [12]:
```
trace2 = trace_values([1, 2, 3], [2, 4, 5], 'lines', 'line trace')
plot({'data': [trace2]})
```

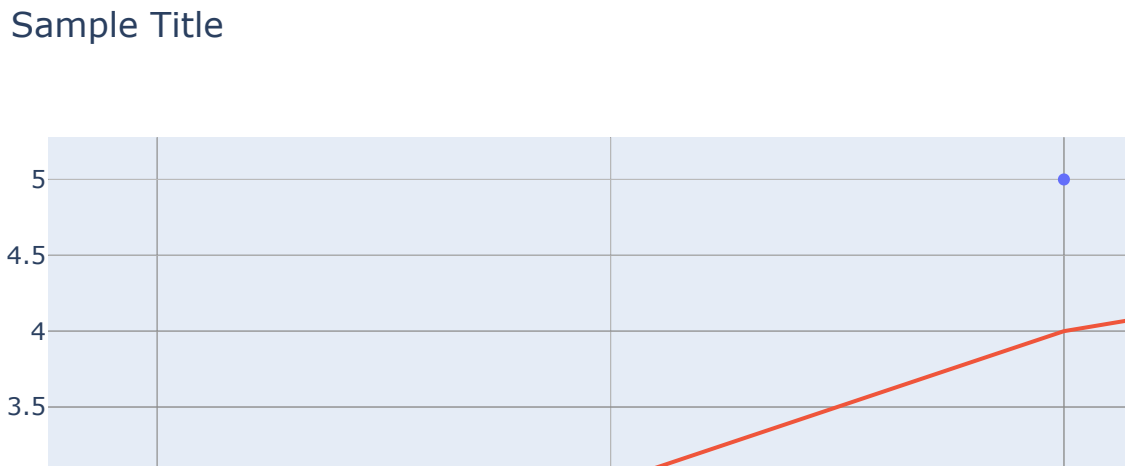


Creating layouts

Ok, now that we have built some functions to create traces, let's write a function to create a layout. Remember that our layout also can be passed to our plot function.

> Uncomment and run the code below.

In [13]:
```
plot({'data': [trace0, trace2], 'layout': {'title': 'Sample Title'}})
```



Our `layout` function should return a dictionary, just as it's defined in the above plot. We'll start by returning an empty dictionary then below we'll walk through building out the rest of the function.

In [14]:
```
def layout(x_range = None, y_range = None, options = {}):
    layout_dict = dict()
    if (isinstance(x_range,list) and isinstance(y_range,list) and options):
        layout_dict.update({'xaxis':{'range':x_range},'yaxis':{'range':y_ra
    elif (isinstance(x_range,list) and isinstance(y_range,list) and not opt
        layout_dict.update({'xaxis':{'range':x_range},'yaxis':{'range':y_ra
    elif (isinstance(x_range,list) and not isinstance(y_range,list) and not
        layout_dict.update({'xaxis':{'range':x_range}})
    elif (not isinstance(x_range,list) and isinstance(y_range,list) and not
        layout_dict.update({'yaxis':{'range':y_range}})
    elif (isinstance(x_range,list) and not isinstance(y_range,list) and opt
        layout_dict.update({'xaxis':{'range':x_range},'options':options})
    elif (not isinstance(x_range,list) and isinstance(y_range,list) and opt
        layout_dict.update({'yaxis':{'range':y_range},'options':options})
    elif (not isinstance(x_range,list) and not isinstance(y_range,list) and
        layout_dict.update({'options':options})
    else:
        layout_dict = {}
    return layout_dict
    pass
```

In [15]:
```
layout()
# {}
```

Out[15]:
```
{}
```

Setting the xaxis and yaxis range

Oftentimes in building a layout, we want an easy way to set the range for the `x` and `y` axis. To set a range in the x-axis of 1 through 4 and a range of the y-axis of 2 through 5, we return a layout of the following structure.

```
{'xaxis': {'range': [1, 4]}, 'yaxis': {'range': [2, 5]}}
```

Let's start with adding functionality to the `layout()` function so it can set the range for the x-axis. (**Hint**: Google search Python's built-in `isinstance()` and `update()` functions.)

Add an argument of x_range returns a dictionary with a range set on the xaxis.

In [16]:
```
layout([1, 4])
# {'xaxis': {'range': [1, 4]}}
```

Out[16]:
```
{'xaxis': {'range': [1, 4]}}
```

We want to ensure that when an x_range is not provided, an empty dictionary is still returned.

```
layout()
# {}
```

The `x_range` should be a default argument that sets `x_range` to `None`. Then, only add a key of xaxis to the dictionary layout when the `x_range` does not equal `None`.

In [17]:
```
layout() # {}
```

Out[17]:
```
{}
```

Now let's provide the same functionality for the `y_range` . When the `y_range` is provided we add a key of `yaxis` which points to a dictionary that expresses the y-axis range.

In [18]:
```
layout([1, 3], [4, 5])
# {'xaxis': {'range': [1, 3]}, 'yaxis': {'range': [4, 5]}}
```

Out[18]:
```
{'xaxis': {'range': [1, 3]}, 'yaxis': {'range': [4, 5]}}
```

Adding layout options

Now have the final argument of our layout function be options. The `options` argument should by default point to a dictionary. And whatever is provided as pointing to the options argument should be updated into the returned dictionary.

In [19]:
```
layout(options = {'title': 'foo'})
```

Out[19]:
```
{'options': {'title': 'foo'}}
```

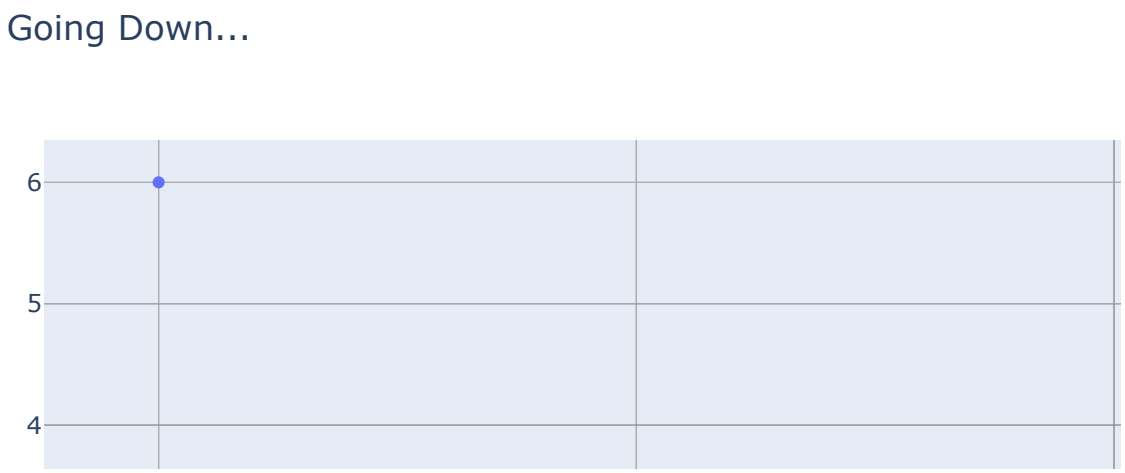
In [20]:
```
layout([1, 3], options = {'title': 'chart'})

# {'xaxis': {'range': [1, 3]}, 'title': 'chart'}
```

Out[20]:
```
{'xaxis': {'range': [1, 3]}, 'options': {'title': 'chart'}}
```

Ok, now let's see this `layout` function in action.

In [21]:
```
another_trace = trace_values([1, 2, 3], [6, 3, 1])
print("another trace:"+str(another_trace))
another_layout = layout([-1, 4], [0, 7], {'title': 'Going Down...'})
print("another layout:"+str(another_layout))
plot({'data': [another_trace], 'layout': {'title': 'Going Down...'}})
#plot({'data': [another_trace], 'layout': another_layout})
```

```
another trace:{'x': [1, 2, 3], 'y': [6, 3, 1], 'mode': 'markers', 'name':
'data'}
another layout:{'xaxis': {'range': [-1, 4]}, 'yaxis': {'range': [0, 7]},
'options': {'title': 'Going Down...'}}
```



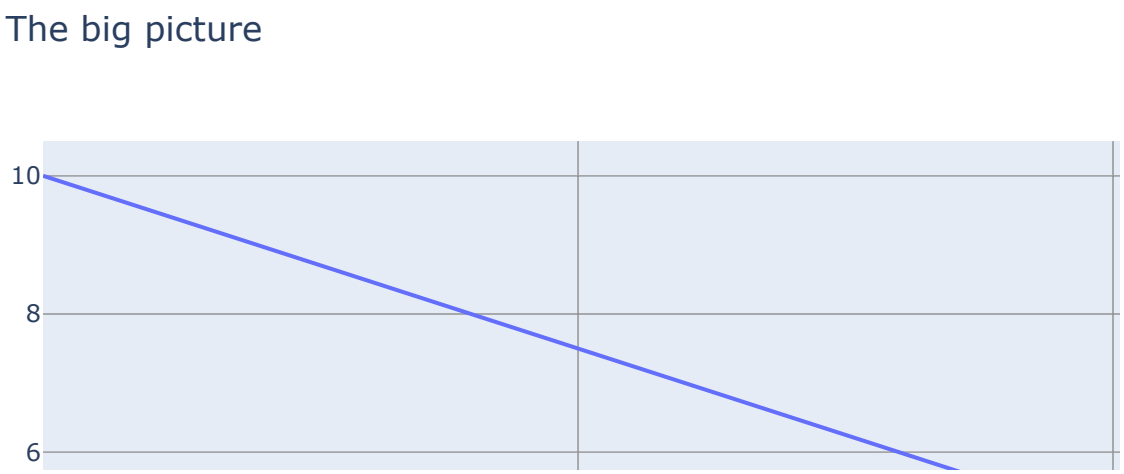
Finally, we'll modify the `plot` function for you so that it takes the data, and the layout as arguments.

In [124]:
```
#def plot(traces = [], layout = {}):
#if not isinstance(traces, list): raise TypeError('first argument must be a
#if not isinstance(layout, dict): raise TypeError('second argument must be
#plotly.offline.iplot({'data': traces, 'layout': layout})
```

Uncomment the below code to see the updated `plot` function in action.

In [22]:
```
trace4 = trace_values([4, 5, 6], [10, 5, 1], mode = 'lines')
print("trace4:"+str(trace4))
last_layout = layout(options = {'title': 'The big picture'})
print("last layout:"+str(last_layout))
plot({'data': [trace4], 'layout': {'title': 'The big picture'}})
#plot([trace4], last_layout)
```

```
trace4:{'x': [4, 5, 6], 'y': [10, 5, 1], 'mode': 'lines', 'name': 'data'}
last layout:{'options': {'title': 'The big picture'}}
```



Summary

In this lab, we built out some methods so that we can easily create graphs going forward. We'll make good use of them in the lessons to come, as well as write new methods to help us easily display information in our charts.