



# Datamics präsentiert: Python Visualisierung von Dashboards mit Plotly's Dash Bibliothek

---

© Datamics

# Inhalt

SO VERWENDEST DU DAS DOKUMENT:	7
LEKTIONEN	7
<b>Plotly Grundlagen</b>	<b>7</b>
Plotly Grundlagen Überblick	7
Streudiagramme	9
Liniendiagramme	13
Balkendiagramme	16
Blasendiagramme	19
Kastendiagramme (Box Plots)	21
Histogramme	25
Histogramme - BONUS Beispiel	29
Verteilungsdiagramme	31
Heatmaps	34
Übungen: Plotly Grundlagen	38
EX1-Scatterplot.py	38
Ex2-Linechart.py	38
Ex3-Barchart.py	38
Ex4-Bubblechart.py	38
Ex5-Boxplot.py	38
Ex6-Histogram.py	38
Ex7-Distplot.py	38
Ex8-Heatmap.py	38
Lösungen: Plotly Grundlagen	39
<b>Dash Basics - Layout</b>	<b>40</b>
Einführung: Dash Grundlagen	40
Dash Layout	40
Ein einfaches Plotly-Diagramm mit Dash zu Dashboard konvertieren	45
Übung: Erstelle ein einfaches Dashboard	46
Lösung: Erstelle ein einfaches Dashboard	47
Dash Komponenten	49
HTML Komponenten	49
Core Komponenten	51
Markdown	53
Nutzung von Help() mit Dash	54
Schreiben von Help() in HTML:	55
<b>Dash - Interaktive Komponenten</b>	<b>56</b>
Interactive Komponenten Übersicht	56
Verbinden von Komponenten mit Callbacks	57
Einen Callback zu einer Komponente hinzufügen	57
Zwei Komponenten mit Callbacks verbinden	59

In Bezug auf Stil	60
In Bezug auf Konnektivität	60
Multiple Inputs	61
Multiple Outputs	65
Übung: Interaktive Komponenten	69
Lösung: Interaktive Komponenten	69
Callbacks mit Dash State kontrollieren	70
<b>Interaktion mit Visualisierungen</b>	<b>73</b>
Einleitung zu Interaktion mit Visualisierungen	73
Hover Over Daten (mit der Maus drüberfahren)	73
Click Data	78
Selected Data (Datenauswahl)	79
Diagramme und Interaktionen updaten	84
Kodieren von Meilenstein-Projekten	91
Einführung zum Thema Live Updating	91
Einfaches Beispiel zum Thema Live Updating	92
<b>Bereitstellung</b>	<b>98</b>
Einführung zur Bereitstellung von Apps	98
App Authorisierung	98
App in Heroku bereitstellen	100
STEP 1 – Installieren von Heroku und Git	100
STEP 2 – Installieren von virtualenv	102
STEP 3 – Erstelle einen Entwicklungsordner (Development Folder)	102
STEP 4 – Initialisierung von Git	102
STEP 5 (WINDOWS) – Erstellen, Aktivieren und Bestücken von virtualenv	102
STEP 5 (macOS/Linux) - Erstellen, Aktivieren und Bestücken von virtualenv	103
STEP 6 – Dem Entwicklungsordner Dateien hinzufügen	103
appy1.py	103
.gitignore	104
Procfile	104
requirements.txt	104
STEP 7 - Log onto your Heroku Account	105
STEP 8 - Initialize Heroku, add files to Git, and Deploy	105
STEP 9 - Visit Your App on the Web!	105
STEP 10 - Update Your App	106
FEHLERSUCHE	106
<b>APPENDIX I - BEISPIELCODES:</b>	<b>107</b>
<b>Plotly Grundlagen</b>	<b>107</b>
Plotly Grundlagen Überblick	107
basic1.py	107
basic2.py	107
Streudiagramme	108
scatter1.py	108
scatter2.py	108



scatter3.py	109
Liniendiagramme	110
line1.py	110
line2.py	111
line3.py	112
Balkendiagramme	113
bar1.py	113
bar2.py	114
bar3.py	115
Blasendiagramme	116
bubble1.py	116
bubble2.py	117
Kastendiagramme (Box Plots)	118
box1.py	118
box2.py	118
box3.py	119
Histogramme	120
hist1.py	120
hist2.py	120
hist3.py	121
hist4.py	121
histBONUS.py	122
Verteilungsdiagramme	123
dist1.py	123
dist2.py	123
dist3.py	124
Heatmaps	124
heat1.py	124
heat2.py	125
heat3.py	125
heat4.py	126
Lösungen: Plotly Grundlagen	127
Sol1-Scatterplot.py	127
Eine Anmerkung zur Liniendiagramm Übung:	128
Sol2a-Linechart.py	129
Sol2b-Linechart.py	130
Sol3a-Barchart.py	131
Sol3b-Barchart.py	132
Sol4-Bubblechart.py	133
Sol5-Boxplot.py	134
Sol6-Histogram.py	135
Sol7-Distplot.py	136
Sol8-Heatmap.py	137

APPENDIX II – DASH CORE KOMPONENTEN	138
Dropdown	138
Slider (Schieberegler)	138



RangeSlider	139
Input	139
Textfeld	139
Checklists	140
Radio Buttons (Auswahlfelder)	140
Button	141
DatePickerSingle	141
DatePickerRange	142
Markdown	142
Graphen	142
Noch in Entwicklung	142
Interaktive Tabellen	142
Upload Komponente	142
Tabs	143

## APPENDIX III – ZUSÄTZLICHE QUELLEN (ENGLISCH) 143

Plotly User Guide for Python	143
Plotly Python Figure Reference	143
• Scatter	143
• ScatterGL	143
• Bar	143
• Box	143
• Pie	143
• Area	143
• Heatmap	143
• Contour	143
• Histogram	143
• Histogram 2D	143
• Histogram 2D Contour	143
• OHLC	143
• Candlestick	143
• Table	143
3D Charts:	143
• Scatter3D	143
• Surface	143
• Mesh	143
Maps:	143
• Scatter Geo	143
• Choropleth	143
• Scatter Mapbox	143
Weiterführende Charts:	143
• Carpet	143
• Scatter Carpet	143
• Contour Carpet	143
• Parallel Coordinates	143
• Scatter Ternary	143
• Sankey	143



Dash User Guide	144
Dash Tutorial	144
• Part 1 - Installation	144
• Part 2 - Dash Layout	144
• Part 3 - Basic Callbacks	144
• Part 4 - Dash State	144
• Part 5 - Interactive Graphing and Crossfiltering	144
• Part 6 - Sharing Data Between Callbacks	144
Dash HTML Components	144
Dash Core Components Gallery	144
• Dropdown	144
• Slider	144
• RangeSlider	144
• Input	144
• Textarea	144
• Checklist	144
• Radio Items	144
• DatePickerSingle	144
• DatePickerRange	144
• Markdown	144
• Buttons	144
• Graphen	144



## So verwendest du das Dokument:

Unterstrichener Text weist in der Regel auf einen Hyperlink zu einer externen Website oder zu einem Ort innerhalb dieses Dokuments hin.

Klicke einmal auf den Text, um den Link anzuzeigen, und klicke dann auf den Link, um dorthin zu springen. Externe Links sollten in einem separaten Browser-Tab geöffnet werden. Klicke zum Beispiel [hier](#), um zur Überschrift zu springen.

Das Inhaltsverzeichnis oben in diesem Dokument bietet eine ähnliche Navigation.

## Lektionen

### Plotly Grundlagen

#### Plotly Grundlagen Überblick

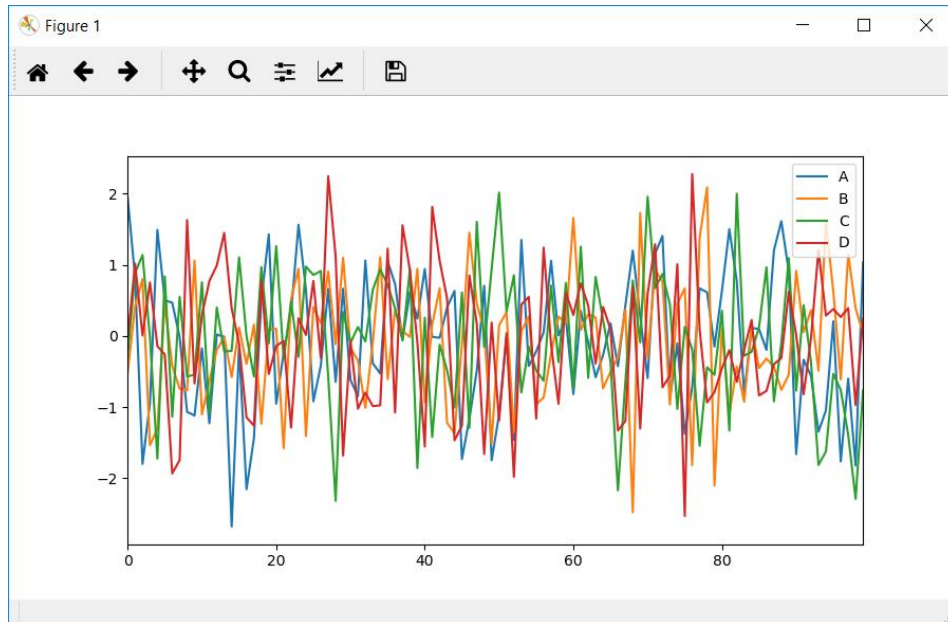
In diesem Abschnitt wird [Plotly](#) mit [matplotlib](#) verglichen, wobei die gleichen Daten verwendet werden, um die Interaktivität von Plotly im Browser anzuzeigen. Das erste Beispiel enthält ein statisches Matplotlib-Diagramm mit vier Linien (als *traces* bezeichnet), die auf Basis der Stichproben gezeichnet werden.

Erstelle eine Datei mit dem Namen **basic1.py** und füge den folgenden Code hinzu:

```
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()
```

Führe am Rechner **python basic1.py** aus. Ein separates Diagrammfenster sollte angezeigt werden:



- Hier wird keine Interaktivität angeboten, es handelt sich lediglich um ein statisches Bild.
- Du könntest dieses Bild bei Bedarf als PNG-Datei speichern.
- Schließe das Plotfenster, um das Skript zu schließen.

Als Nächstes erstellen wir ein Plotly-Diagramm mit ähnlichen Daten. Erstelle dafür eine neue Datei mit dem Namen **basic2.py** und füge den folgenden Code hinzu:

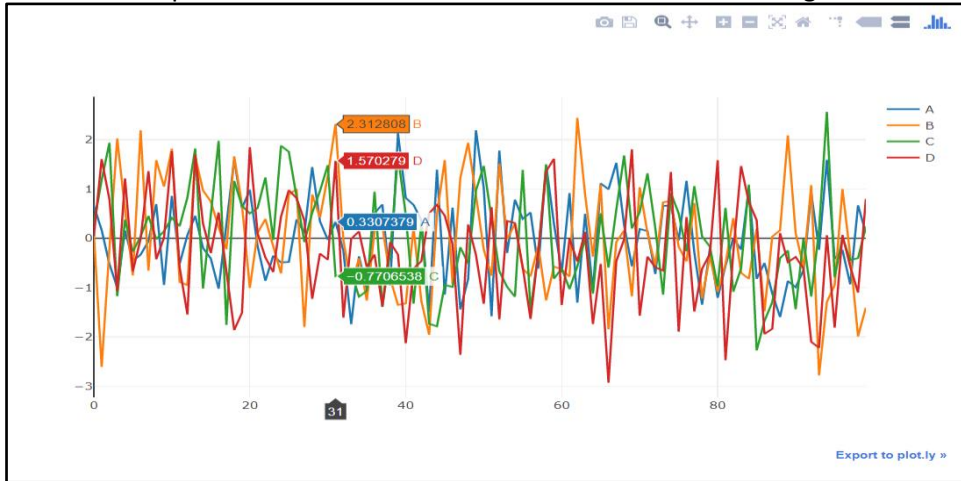
```
import numpy as np
import pandas as pd
import plotly.offline as pyo

# 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])
```

- Wir haben den Alias **pyo** dem `plotly.offline`-Import zugewiesen, um ihn von `import plotly.plotly as py` zu unterscheiden, wie in den meisten Online-Beispielen gezeigt wird. Plotly bietet ein Online-Hosting auf ihrer Website für diejenigen, die ein Account bei ihnen einrichten.
- Im Verlauf dieses Kurses erstellen wir Offline-Diagramme und führen diese lokal aus.
- **basic2.py** verwendet eine *vollständige Liste (List comprehension)*, um für jede Spalte im DataFrame eine Linie zu erstellen. Diese Technik wird später ausführlicher behandelt.



Führe das Skript am Rechner aus. Es sollte sich nun automatisch folgendes Browserfenster öffnen:



- Bewege den Mauszeiger über die Datenpunkte, um die Informationen anzuzeigen.
- Wenn du einmal auf eine Linie klickst (in der Legende steht eine Linie für eine der angezeigten Datensätze A, B, C oder D) wird diese aus dem Diagramm entfernt, durch Doppelklicken auf eine Linie wird diese einzeln angezeigt. Bei erneutem Doppelklicken werden die anderen Linien wieder hinzugefügt.
- Wenn du in das Verzeichnis schaust, in dem **basic2.py** gespeichert wurde, sollte eine neue Datei mit dem Namen **temp-plot.html** angezeigt werden. Plotly erstellt diese Datei, die im Browser angezeigt wird. Wir zeigen dir später noch, wie man durch Hinzufügen eines Arguments `filename = 'something-else.html'` den Namen der Datei ändern kann (nützlich bei der Arbeit mit mehreren Plots). Durch erneutes Ausführen eines bestimmten Skripts werden frühere Kopien der Datei ersetzt.
- Du kannst dieses Diagramm auch in einer statischen .png-Bilddatei speichern, wenn du möchtest.

**Diagramme vs. Graphen** (Plots vs. Charts) - wir scheinen diese Begriffe gleich zu verwenden. Wir sagen Dinge wie "ein Blasengraph ist eine bestimmte Art von Streudiagramm". Der einzige echte Unterschied besteht darin, dass Graphen eine Art Symbole verwenden, um die Daten darzustellen.

Von <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." („Ein Graph ist eine grafische Darstellung von Daten, in der die Daten durch Symbole dargestellt werden, z. B. Balken in einem Balkendiagramm, Linien in einem Liniendiagramm oder Segmente in einem Kreisdiagramm. Ein Graph kann tabellarische numerische Daten, Funktionen oder bestimmte Arten qualitativer Struktur darstellen und bietet verschiedene Informationen.“)

## Streudiagramme

Ein einfaches Streudiagramm bildet eine Verteilung von Datenpunkten entlang einer X- und Y-Achse ab. Zur Veranschaulichung nehmen wir eine Zufallsstichprobe von 100 Koordinatenpaaren, dafür verwenden wir Startwerte im NumPy-Zufallsgenerator, so dass jeder die gleiche „Zufallsstichprobe“ erhält.

Erstelle eine Datei mit dem Namen **scatter1.py** und füge den folgenden Code hinzu:

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

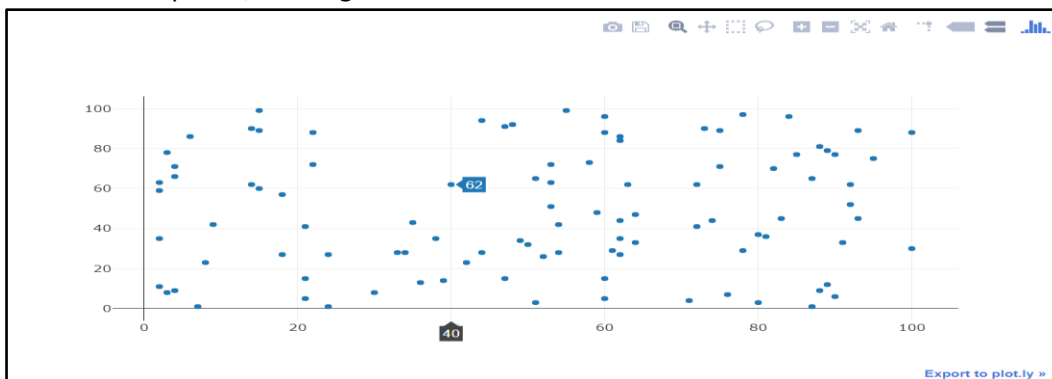
np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

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

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

- **scatter1.py** stellt 100 zufällige Koordinatenpaare dar. Durch den Zufallsgenerators können wir jedes Mal, wenn das Skript ausgeführt wird, dieselbe Darstellung reproduzieren.
- Jetzt ist vermutlich ein guter Zeitpunkt, um zu erwähnen, dass Zufallsgeneratoren algorithmisch und nicht wirklich zufällig sind und niemals in der Cybersicherheit verwendet werden sollten! Dies erklärt, warum wir Startwerte setzen können, um dieselben Ergebnisse zu erzielen.

Führe das Skript aus, um Folgendes zu sehen:



- Du wirst feststellen, dass das Diagramm keinen Titel und keine Achsenbeschriftungen hat. Um sie hinzuzufügen, verwenden wir das *Layout*-Modul **graph\_objs**, um unser Diagramm mit ein paar Funktionen auszustatten.
- Beim Bewegen des Cursors über die Grafik werden Informationen zu einzelnen Punkten angezeigt. Wenn sich jedoch mehr als ein Punkt in derselben Vertikalen befindet, werden nur Daten zu einem der Punkte angezeigt! Glücklicherweise kann dies durch Hinzufügen eines weiteren Parameters im Layout behoben werden.

Erstellen eine Kopie von scatter1.py und nenne sie **scatter2.py**. Füge den folgenden Code hinzu (fettgedruckt):

```
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)

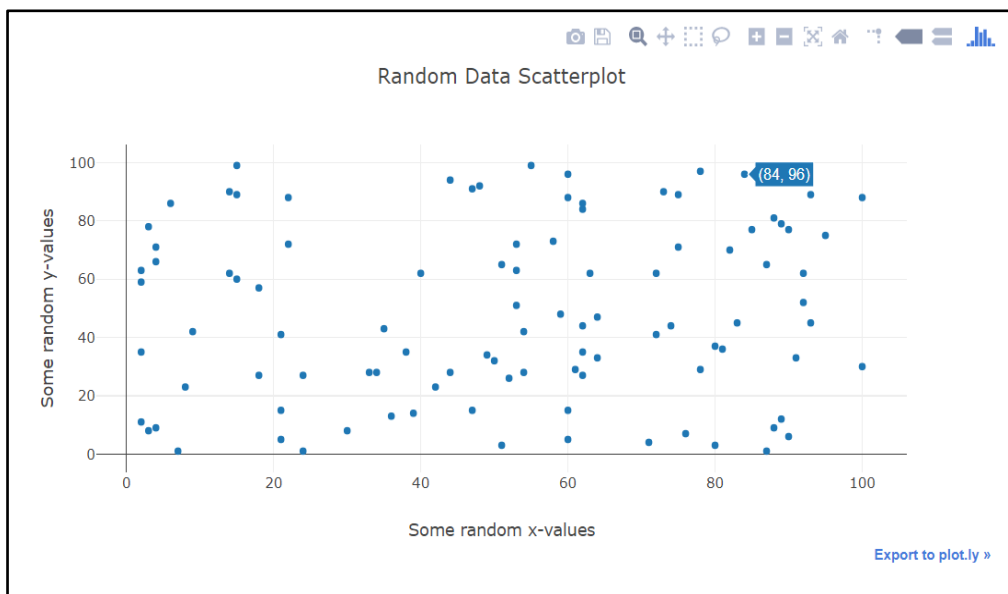
data = [go.Scatter(
    x = random_x,
    y = random_y,
    mode = 'markers',
)]

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** stellt die gleichen Punkte wie scatter1 dar, fügt jedoch eine *Layout*-Ebene hinzu, die einen Titel und Achsenbeschriftungen enthält, und behebt das Anzeigeproblem beim Darüberfahren mit der Maus. Schau dir nun an, wie wir sowohl die Daten als auch das Layout in einer Darstellung gebündelt haben und als HTML grafisch abbilden können.



In Plotly kannst du eine Menge Anpassungen vornehmen, um die Darstellung des Diagramms zu verändern.

**scatter3.py** ist das gleiche wie scatter2, nur dass wir den Datenpunkten einen Stil geben. Wir haben Farbe, Größe und Form geändert und einen Rahmen hinzugefügt:

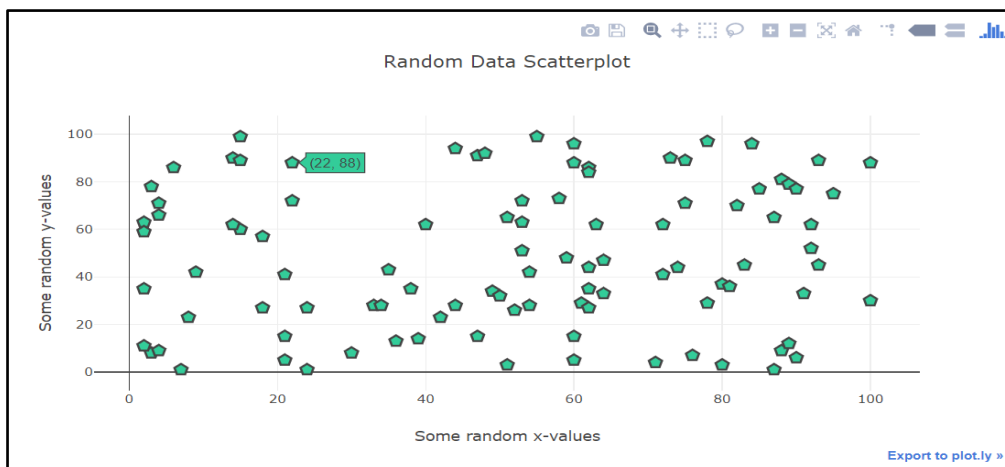
```
import plotly.offline as pyo
import plotly.graph_objs as go
import numpy as np

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')
```



Mehr Informationen zur Personalisierung von Graphen findest du hier: <https://plot.ly/python/reference/#scatter>

Quellen: <https://plot.ly/python/line-and-scatter/> und <https://plot.ly/python/reference/#scatter>

## Liniendiagramme

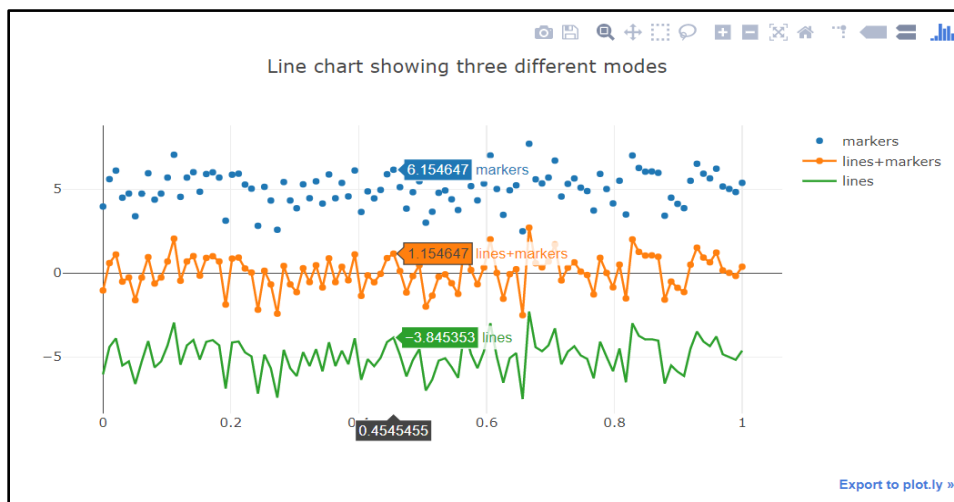
Liniendiagramme sind ein bisschen mehr als Streudiagramme, die nur einen Datenpunkt pro x-Wert haben und die Daten durch eine Linie verbunden sind (optional). Um dies zu veranschaulichen, nehmen wir eine weitere zufällige Auswahl von Daten, die gleichmäßig entlang der x-Achse verteilt sind.

**line1.py** erstellt drei Kopien desselben Zufallsdatensatzes. Jeder Satz wird zu einer Spur, d.h. zu einem unabhängigen Datensatz, der in unserem Diagramm angezeigt wird.

```
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')
```



- Achte darauf, dass jeder Spur ein Name zugewiesen wird (hier: marker / lines + markers / lines). Namen erscheinen in der Legende oben rechts (ähnlich den A-B-C-D-Namen, die wir in unserem ersten Plot-Beispiel gesehen haben) und als Hover-Text (wenn man mit der Maus darüberfährt).

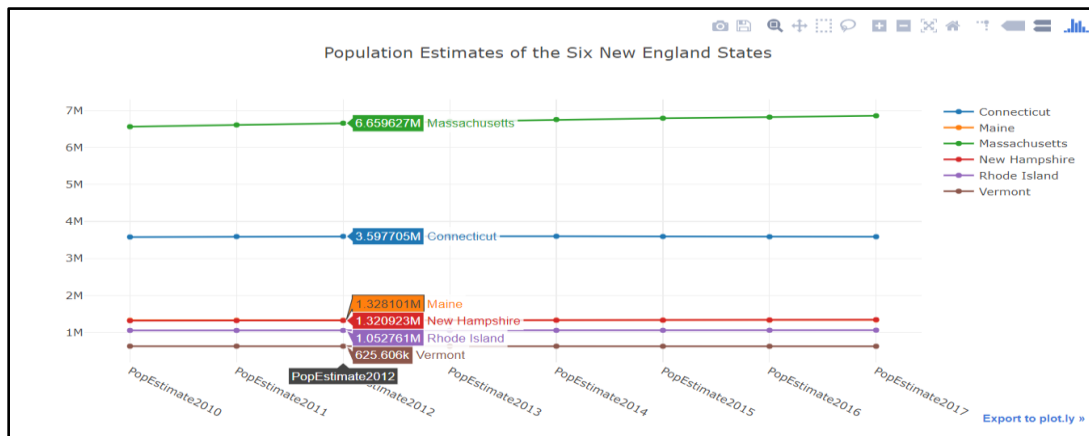
**line2.py** nimmt einige Online-Daten und erstellt eine Reihe von Liniendiagrammen. Für diese Übung haben wir einen Datensatz aus dem US Census Bureau importiert und in eine kleine Datei mit dem Namen **population.csv** umgewandelt. Diese Datei wird in einem Ordner mit dem Namen **/data** gespeichert:

```
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')
```



- Um eine Datei aus einem benachbarten Verzeichnis zu holen, verwenden wir `pd.read_csv('.../data/filename.csv')`
- Wir überspringen das Argument `index_col=0` um zu vermeiden, dass Pandas einen numerischen Index zu unseren Daten hinzufügt.  
Dies wird im Abschnitt Datenmanipulation mit Pandas genauer beschrieben.
- Ähnlich wie **basic2.py** verwenden wir ein Listenverständnis (list comprehension), um einzelne Spalten aus dem DataFrame zu extrahieren.
- Interessant an diesem Plot ist, dass die Populationen von Maine und New Hampshire fast gleich sind und wir das erst sehen, wenn wir über die rote Linie fahren. Wenn man in Legende auf New Hampshire klickt, wird die orangefarbene Linie von Maine erst sichtbar.

Quelle: <https://plot.ly/python/line-charts/>

Datenquelle: <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>

## Balkendiagramme

Balkendiagramme zeigen verschiedene Kategorien entlang der X-Achse und numerische Werte entlang der Y-Achse. Kategorien werden anhand der Höhe der einzelnen Balken verglichen. Aus diesem Grund ist es wichtig, dass die Y-Achse immer bei Null beginnt, um visuelle Falschdarstellungen zu vermeiden.

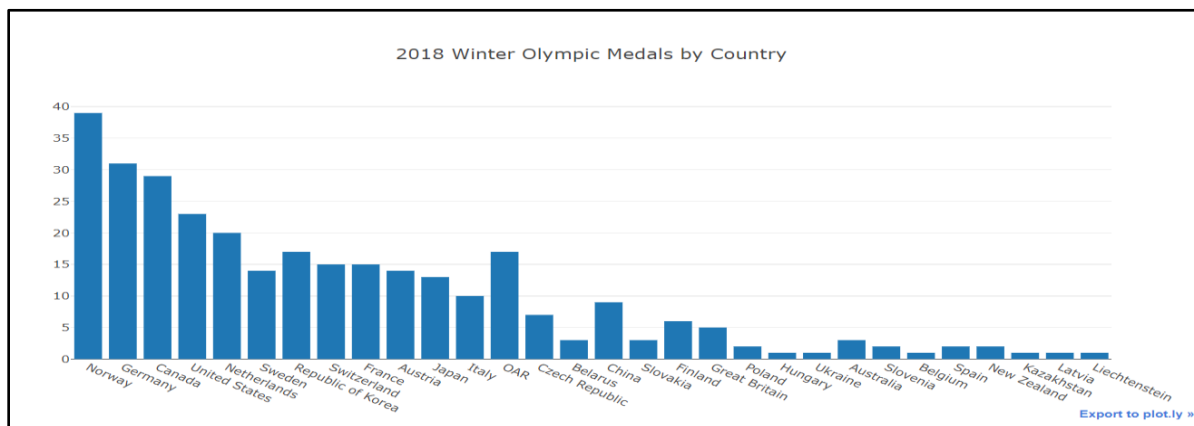
Dieser Abschnitt beginnt mit einem einfachen, einfarbigen Balkendiagramm, das die Anzahl der, von den Ländern bei den Olympischen Winterspielen 2018 in PyeongChang, Südkorea, gewonnenen Medaillen zeigt.

Wir haben dem Ordner `.../data` eine `.csv`-Datei mit dem Namen **2018WinterOlympics.csv** hinzugefügt und die Daten mit **bar1.py** dargestellt:

```
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')
```



- Beachte, dass die Ländernamen unter der Spalte NOC stehen - NOC steht für National Olympic Committee.
- Wir sollten noch erwähnen, dass OAR für „Olympic Athletes from Russia“ steht. Russland wurde von diesen Olympischen Spielen ausgeschlossen, einige Athleten wurden jedoch zum Wettbewerb zugelassen.
- Die Länder sind in der Rangfolge von links nach rechts geordnet, aber einige Länder wie Südkorea erzielten trotzdem mehr Medaillen als Länder mit einem scheinbar höheren Wert, wie Schweden. Warum, erfahren wir in den nächsten beiden Plots.

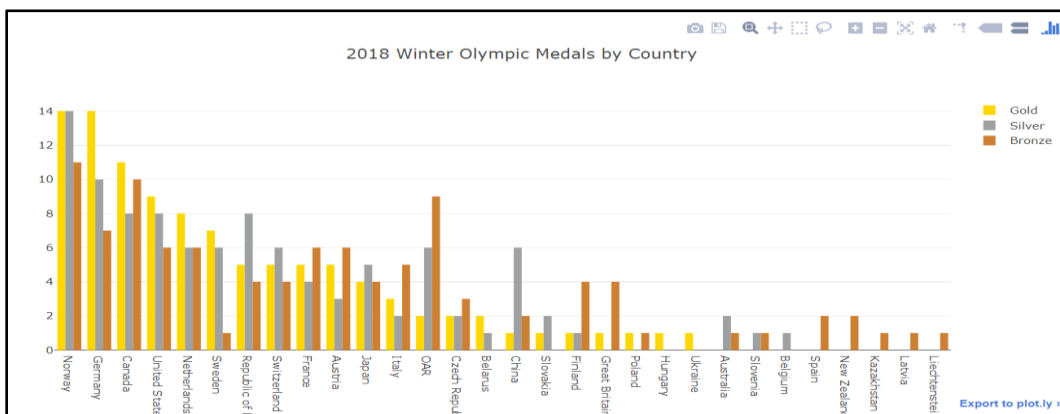


Werfen wir einen Blick auf die Medaillenarten, die jedes Land verdient hat: Gold, Silber und Bronze mit **bar2.py**:

```
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')
```



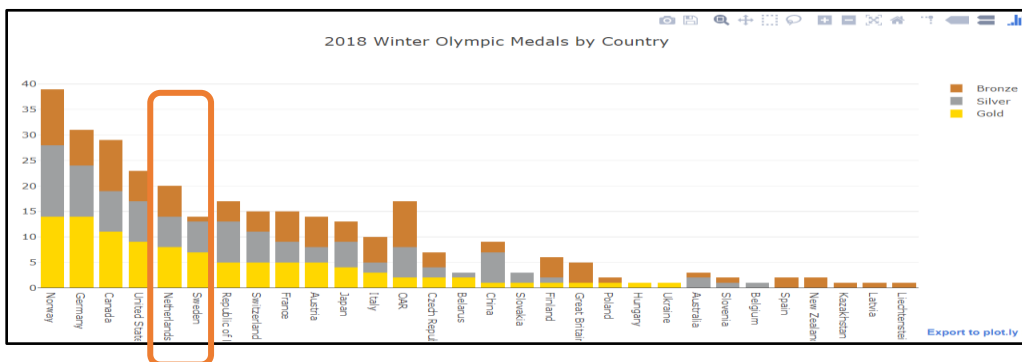
- Hier wird nun Gold / Silber / Bronze nebeneinander in einem gruppierten Balkendiagramm angezeigt.
- Wir haben jeder Spur die passende Farbe zugewiesen.
- In diesem Beispiel ist jedoch schwer zu erkennen, welche Auswirkungen unterschiedliche Medaillen auf die Gesamtpunktzahl haben. Im nächsten Beispiel stapeln wir die Balken deshalb.

**bar3.py** erstellt ein **gestapeltes Balkendiagramm**. Achte auf den Zusatz `barmode='stack'` im *Layout*-Bereich:

```
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')
```



- Da Gold ganz unten steht, können wir jetzt sehen, warum Schweden Südkorea übertroffen hat!

Quellen: <https://plot.ly/python/bar-charts/> und <https://plot.ly/python/reference/#bar>

Datenquellen: <http://time.com/5143796/winter-olympic-medals-by-country-2018/> und <https://www.pyeongchang2018.com/en/game-time/results/OWG2018/en/general/medal-standings.htm>

## Blasendiagramme

Blasendiagramme sind Streudiagramme mit der zusätzlichen Funktion, die Größe der Datenpunkte durch die Daten darzustellen.

Für diese Übung betrachten wir den **mpg.csv**-Datensatz (Meilen pro Gallone), eine Sammlung von 399 Fahrzeugen, die zwischen 1970 und 1982 hergestellt wurden. Beim Einfügen in einen DataFrame sehen die ersten zehn Datensätze folgendermaßen aus:

	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** vergleicht mpg mit PS. Die Größe der Blase wird durch die Anzahl der Zylinder festgelegt:

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

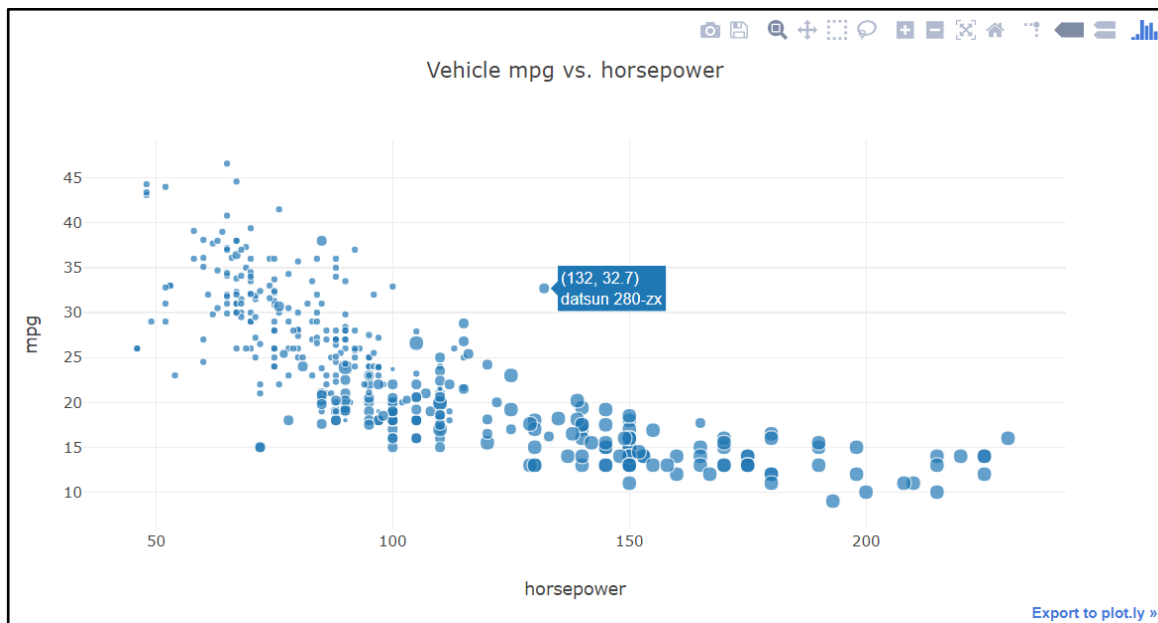
df = pd.read_csv('../data/mpg.csv')

data = [go.Scatter(
    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')
```





- Die Grafik zeigt eine eindeutige Beziehung zwischen einer hohen Leistung und einem niedrigen mpg-Wert (hoher Verbrauch) und zeigt auch einen Trend zu einer höheren Leistung mit einer größeren Anzahl von Zylindern (beachte, dass die Verschiebung hier nicht berücksichtigt wird).
- Wir haben jeder Blase **Text** hinzugefügt, um den Namen des Fahrzeugs beim mit der Maus Überfahren anzuzeigen
- Wir haben **hovermode='closest'** zum Layout hinzugefügt. Andernfalls wird nur die unterste Blase beschrieben, wenn mehrere Markierungen auf demselben vertikalen x-Wert erscheinen.
- Es ist erwähnenswert, dass Blasendiagramme und Streudiagramme potenziell eingeschränkt sind, falls mehr als ein Datenpunkt an derselben Stelle landen sollte. Eine Blase wird dann zwar etwas dunkler, aber es ist schwer zu sagen, ob mehrere Datenpunkte verdeckt werden. Diese Einschränkung wird in der Dash-Sektion "Ausgewählte Daten" **select2.py** adressiert, die die "Dichte" von ähnlich aussehenden Streudiagrammen anzeigt.
- **bubble2.py** ist gleich wie bubble1, nur dass hier gezeigt wird, wie man dem „Hover“-Text mehrere Felder hinzufügen kann. Da eines der Felder numerisch war (model\_year), haben wir zunächst eine Spalte zum DataFrame hinzugefügt, die ihn in Text konvertiert, und dann eine weitere Spalte, um diesem einen Namen zuzuweisen. Diese letzte Spalte wird für den Hover-Text verwendet.

Quellen: <https://plot.ly/python/bubble-charts/> und <https://plot.ly/python/reference/#scatter>

Datenquelle: <https://gist.github.com/omarish/5687264>

## Kastendiagramme (Box Plots)

Manchmal ist es wichtig zu bestimmen, ob zwei Stichproben zur gleichen Grundgesamtheit gehören. Box-Plots sind dafür großartig geeignet! Die Form eines Kastendiagramms (auch Box-und-Whisker-Diagramm genannt) hängt nicht von Aggregationen wie dem Stichprobenmittelwert ab. Die Darstellung repräsentiert vielmehr die wahre Form der Daten. Je nachdem, wie die Whisker aufgebaut sind, dienen Kastendiagramme dazu, echte Ausreißer eines Datensatzes zu identifizieren. Während einige Visualisierungen die oberen und unteren 5% willkürlich als Ausreißer identifizieren, werden in einem Boxdiagramm die Punkte angegeben, die im *Vergleich zu den übrigen Daten* weiter vom Mittelwert entfernt liegen.

Um das Diagramm zu erstellen:

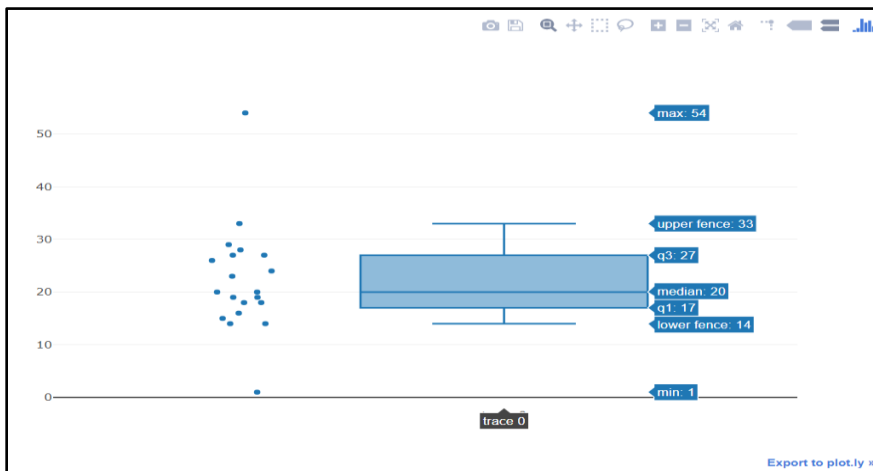
- Markieren wir zuerst den Mittelwert (normalerweise mit einem Liniensegment). Dadurch wird der *Ort* der Verteilung festgelegt.
- Konstruieren wir einen Kasten, der alle Werte innerhalb des Quartils enthält.
- Als nächstes zeichnen wir die *Whisker*. Es gibt mehrere Möglichkeiten, dies zu tun, aber in der Regel beginnen wir damit, eine Kastenlänge außerhalb des gezeichneten Kastens zu markieren und bewegen uns dann nach innen, bis wir den ersten Datenpunkt erreichen.
- Die verbleibenden Punkte außerhalb der Whisker sind damit die *Ausreißer*.

**box1.py** nimmt einen Satz von zwanzig Punkten, zeichnet sie auf und zeigt einen Ausreißer:

```
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')
```



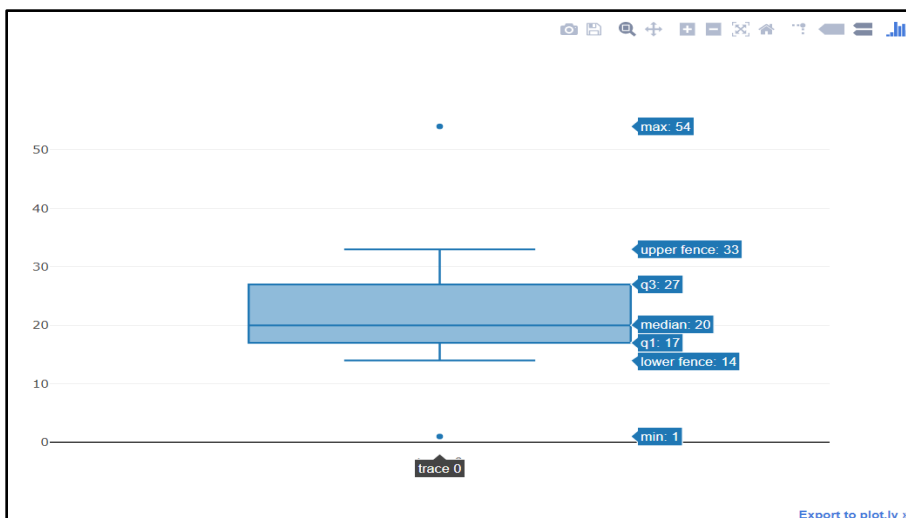
- Da wir die Datenpunkte nach links verschieben, wird der Ausreißer nicht über dem Kastendiagramm selbst angezeigt.

**box2.py** zeigt, wie ein Kastendiagramm mit angezeigten Ausreißern aussehen würde:

```
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')
```



## Die Quintus Curtius Snodgrass Briefe

Als forensisches Beispiel für angewandte Statistiken gab es einen berühmten Fall, in dem Mark Twain beschuldigt wurde, während des Bürgerkriegs ein Deserteur gewesen zu sein. Die Beweise dafür waren zehn Essays, die im *New Orleans Daily Crescent* unter dem Namen Quintus Curtius Snodgrass veröffentlicht wurden. Claude Brinegar veröffentlichte 1963 einen Artikel im *Journal of American Statistical Association*, in dem er Wortfrequenzen und einen Chi-Quadrat-Test verwendete, um zu zeigen, dass die Aufsätze fast mit Sicherheit nicht Twains waren.

### 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  $\chi^2$  and two-sample t tests."*

„Mark Twain ist weithin als Autor von 10 Briefen bekannt, die 1861 im *New Orleans Daily Crescent* veröffentlicht wurden. Die in diesen Briefen beschriebenen Abenteuer, die mit „Quintus Curtius Snodgrass“ signiert sind, bilden die historische Grundlage für einen Hauptteil von Twains vermuteter Rolle im Bürgerkrieg. Diese Studie wendet einen alten, wenn auch wenig genutzten statistischen Test der Urheberschaft an - einen Wortlängen-Frequenztest -, um zu zeigen, dass Twain diese 10 Briefe fast sicher nicht geschrieben hat. Die statistische Analyse beinhaltet einen visuellen Vergleich mehrerer Wortlängen-Häufigkeitsverteilungen und die Anwendungen der  $\chi^2$  und Zwei-Stichproben-t-Tests. “

Die folgende Tabelle zeigt die relative Häufigkeit von Wörtern mit drei Buchstaben aus den Briefen von Snodgrass und aus Beispielen von Twains bekannten Werken. Statt komplexe Berechnungen durchzuführen, erstellen wir Box-Plots!

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

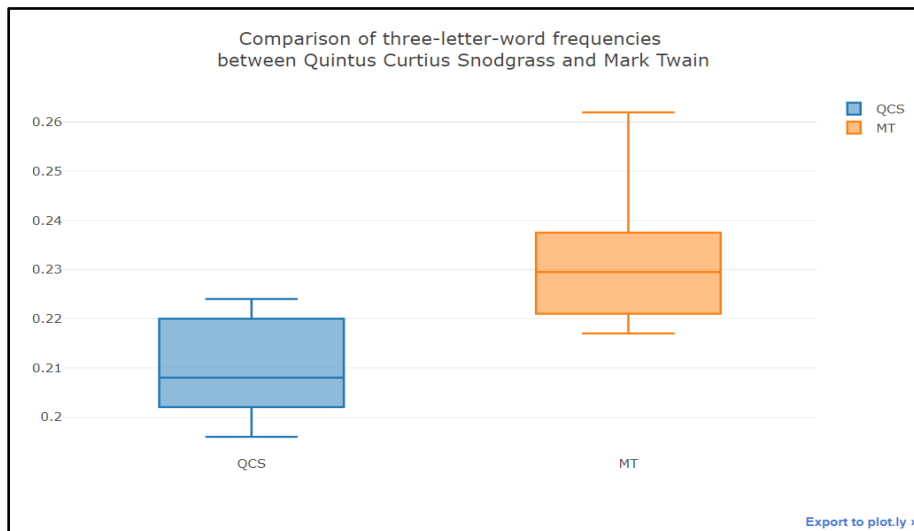
Zitat: Brinegar, C., "Mark Twain and the Quintus Curtius Snodgrass Letters: A Statistical Test of Authorship", *Journal. American Statistical Association*, 1963, 58 (301): 85-96.

**box3.py** vergleicht die beiden Datensätze miteinander:

```
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')
```



- Wie man an den Kästen sehen kann, gibt es kaum Überlappungen! Die 10 Quintus Curtius Snodgrass-Briefe wurden somit höchstwahrscheinlich nicht von Mark Twain geschrieben.

Quellen: <https://plot.ly/python/box-plots/>, <https://plot.ly/python/reference/#box> und <https://help.plot.ly/what-is-a-box-plot/>

Datenquellen: <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](https://www.jstor.org/stable/2282956?seq=1#page_scan_tab_contents)



## Histogramme

Histogramme sind eine der am häufigsten (missbräuchlich) verwendeten Darstellungen. Sie sind zwar sehr gut um zu zeigen, welcher Wertebereich *am häufigsten* auftritt, aber es ist schwer zu sagen, wie *häufig*. Und bei der Konvertierung in 3D, wie in vielen modernen Zeitschriftenartikeln, kann zudem die Perspektive völlig verzerrt werden.

Wenn du jedoch gerade mit deiner Analyse beginnst und einen groben Überblick über die Daten möchtest, sind Histogramme ein praktisches Werkzeug.

Wir sollten darauf hinweisen, dass sich Histogramme, obwohl sie ähnlich aussehen, in zweierlei Hinsicht von Balkendiagrammen unterscheiden. Zunächst zeichnen Histogramme einen numerischen Wert entlang der x-Achse auf - etwas, das gemessen werden kann. Balkendiagramme setzen Kategorien entlang der X-Achse, wie die Länder, der Olympischen Spiele in unserem vorherigen Beispiel. Zweitens gibt nicht allein die Höhe eines Histogrammbalkens im Gegensatz zu Balkendiagrammen die Häufigkeit an, sondern das Volumen des Balkens (Höhe x Breite). Die Breite eines Histogrammbalkens wird durch *Gruppierung* bestimmt; da die x-Achse normalerweise einen kontinuierlichen Wertebereich wie Zeit oder Temperatur anzeigt, repräsentiert jeder vertikale Balken einen gruppierten Wertebereich.

Während Balkendiagramme normalerweise einen Abstand zwischen den Balken haben, haben Histogramme im Allgemeinen keinen Abstand zwischen den benachbarten Balken.

Für diesen Abschnitt nutzen wir das mpg-Dataset erneut. Werfen wir einen Blick auf die Häufigkeitsverteilung der mpg-Werte unserer Fahrzeuge aus den 1970er-Jahren.

**hist1.py** verwendet die Standardeinstellungen von plotly:

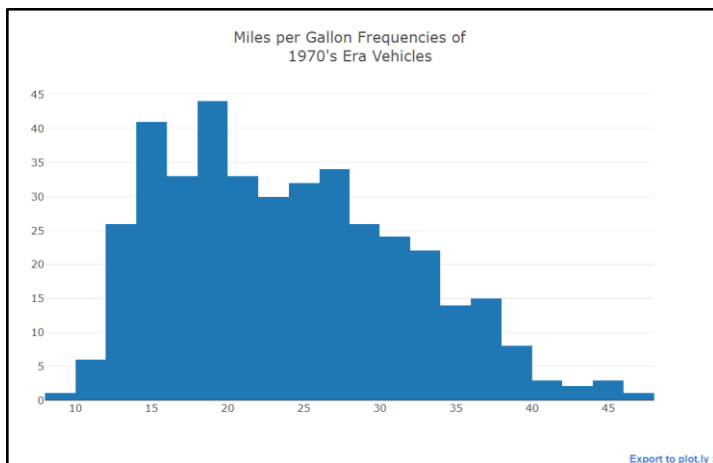
```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

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')
```



- Beachte, dass jeder „Balken“ eine Breite von 2 hat. Der erste Balken umfasst 8 bis 9,9, der letzte von 48 bis 49,9.

**hist2.py** setzt die Balkenbreite auf 6. (weil  $50-8 = 42$  sind sieben Balken mit gleicher Breite sinnvoll)

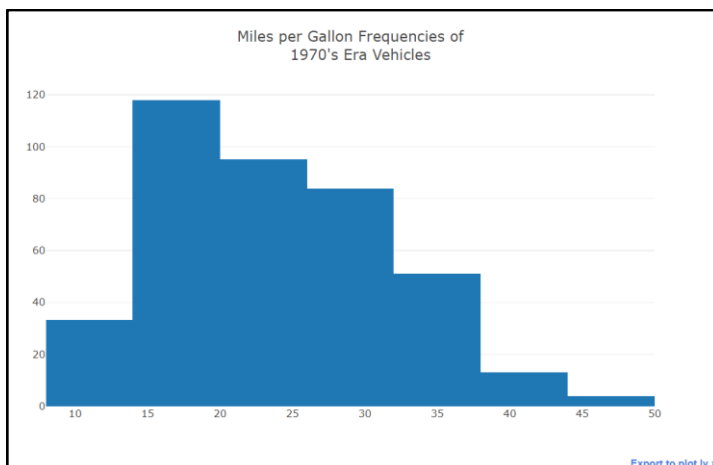
```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

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** bildet eine Balkenbreite von 1 ab:

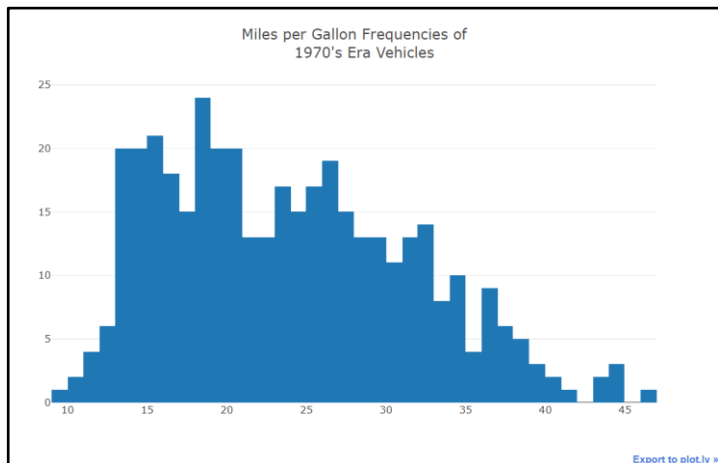
```
import plotly.offline as pyo
import plotly.graph_objs as go
import pandas as pd

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')
```



- Nachdem wir alle drei Diagramme verglichen haben, scheint es, dass die Standardeinstellungen von plotly eine gute Wahl für diesen Datensatz waren.

Das nächste Beispiel zeigt, wie zwei Histogramme übereinandergelegt, ein Deckkraftwert zugewiesen und damit zwei Datensätze verglichen werden können.

Die Daten, die wir verwenden, stammen aus einer Datenbank für Herzrhythmusstörungen unter <https://archive.ics.uci.edu/ml/datasets/arrhythmia>

Wir haben alle, bis auf drei Spalten, entfernt und 420 Datensätze ausgewählt. Die Spalten sind "Alter", "Geschlecht" und "Größe". Für „Geschlecht“ ist 0 = männlich und 1 = weiblich und die Größe wird in Zentimeter gemessen.

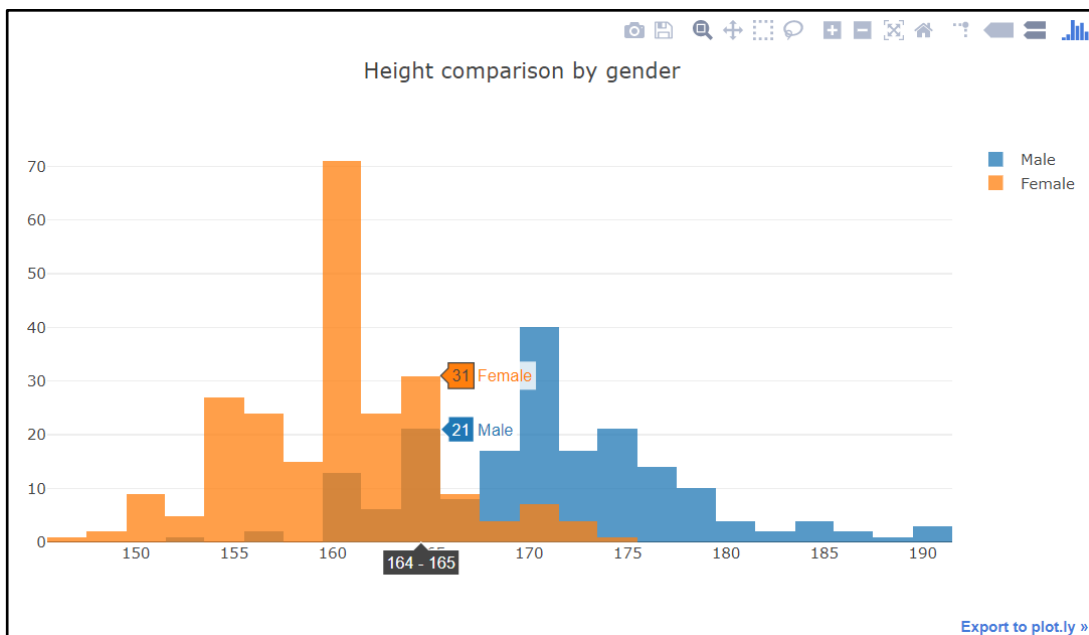
Erstelle eine Datei namens **hist4.py** und füge den folgenden Code hinzu:

```
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')
```



Jetzt hat jede Spur ihre eigene Farbe, und die Deckkraft ermöglicht es, jede Spur unabhängig voneinander zu betrachten.

Mehr Informationen bekommst du auf:

<https://plot.ly/python/histograms/> und <https://plot.ly/python/reference/#histogram>



## Histogramme - BONUS Beispiel

Was ist, wenn der Datensatz selbst Häufigkeitsdaten enthält? Histogramme zählen die Anzahl der Vorkommen in einer Spalte. Wenn du x-Werte auf einer Spalte basieren möchtest, aber die Werte aus einer anderen Spalte summieren möchtest, solltest du ein Balkendiagramm verwenden. Versuchen wir uns an einem Beispiel!

Die **Fremont Bridge** in Seattle, Washington, hat auf beiden Seiten einen Fußgänger-/Fahrradweg. Radfahrer fahren auf der Ostseite im Allgemeinen nach Norden über die Brücke und auf der Westseite nach Süden. Die Stadt installierte Sensoren zur Zählung der Fahrräder, die jeden Tag die Brücke überqueren.

Bilder:



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

Ein schönes Zeitreihen-Dataset ist unter <https://data.seattle.gov/Transportation/Grouped-by-Hour/7mre-hcut> verfügbar und bietet eine über 5-jährige Aufzeichnung (Okt-2012-Feb-2018) der Anzahl der Fahrräder, die die Brücke auf jeder Seite überquert haben.

Für diese Übung erstellen wir eine CSV-Datei, die aus den Quelldaten erstellt wurde, und führen dann einige Änderungen durch:

- wir möchten eine textbasierte Datumsspalte in ein Datum ändern
- Dadurch können wir die Zeitkomponente in eine separate Spalte extrahieren
- Daraus erstellen wir mit groupby einen neuen DataFrame, der die Anzahl der Fahrräder auf der östlichen und westlichen Seite der Brücke summiert

**histBONUS.py** führt diese Aktionen aus und zeigt das Ergebnis an:

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

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

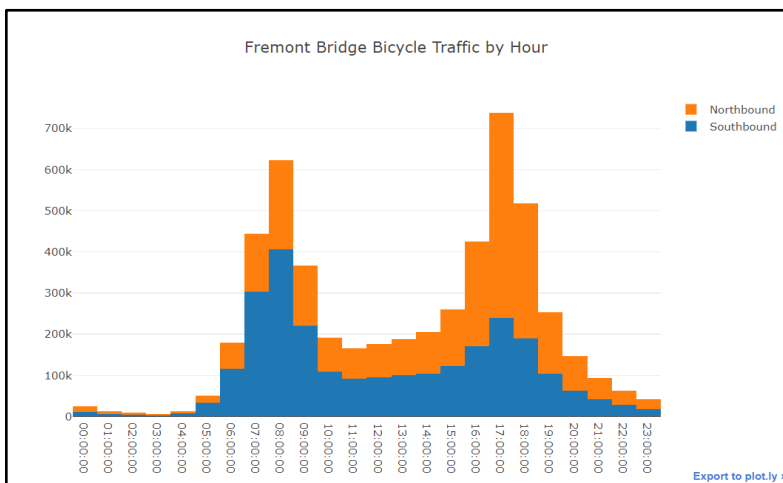
# Convert the "Date" text column to a Datetime series:
df['Date'] = pd.to_datetime(df['Date'])

# 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')
```



- Die Grafik stapelt zwei Spuren. Es zeigt sehr schön, wie der Verkehr in Richtung Süden während des morgendlichen Pendels am höchsten ist, während Radfahrer im Norden den abendlichen Pendelverkehr dominieren.
- Wir setzen die Breite auf 1, sodass sich benachbarte Balken berühren, ähnlich wie bei einem Histogramm.



## Verteilungsdiagramme

Verteilungsdiagramme (oder Displots genannt) stellen normalerweise drei sich überlagernde Diagramme dar. Das erste ist ein **Histogramm**, bei dem jeder Datenpunkt in einem „Balken“ mit ähnlichen Werten gruppiert wird. Das zweite ist ein **Rug-Plot** (sogenanntes Teppichdiagramm), bei dem jeder Datenpunkt entlang der x-Achse platziert wird, sodass die Verteilung der Werte in jedem Balken anzeigen werden kann. Schließlich enthalten Verteilungsdiagramme häufig eine **"Kerndichteschätzung"** oder eine KDE-Linie (kernel density estimate), die versucht, die Form der Verteilung zu beschreiben.

KDEs verwenden Berechnungen, um die Form der Linie abzuleiten. Wenn su eine zu große Bandbreite verwendest, erhältst du eine Linie ohne genügend Details, und eine zu kleine Bandbreite kann eine wenig hilfreiche, gezackte Linie ergeben. Wir sagen, dass wir ein Histogramm *zeichnen*, aber KDE-Linie an ein Diagramm *anpassen*.

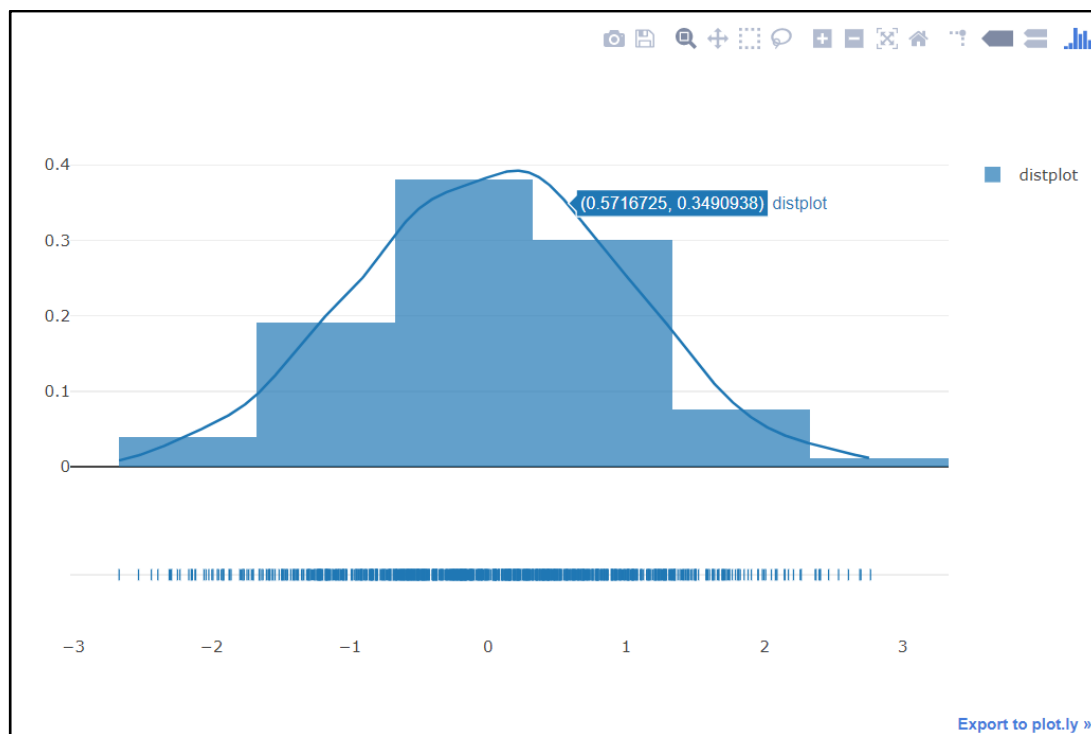
Wir erhalten Verteilungsdiagramme von Plotlys **Figure Factory**-Modul anstelle von **Graph Objects**.

**dist1.py** zeigt ein einfaches Verteilungsdiagramm aus 1000 zufälligen Werten:

```
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')
```



- Beachte, dass Verteilungsdiagramme *relative* Häufigkeiten anzeigen, nicht tatsächliche. Die Gesamtfläche unter dem Diagramm beträgt 1.
- Konventionell verwenden wir die Bezeichnung `hist_data` anstelle von `data`, um daran zu erinnern, dass diese den Histogramm-Anteil des Diagramms bildet.

Ein Zufallsgenerator zeigt niemals eine vollkommen normale Verteilung (Gaußsche Verteilung) - aber je größer die Anzahl der Datenpunkte ist, desto näher kommen man dieser. Um das zu demonstrieren, werden wir vier relativ kleine Proben nebeneinanderstellen.

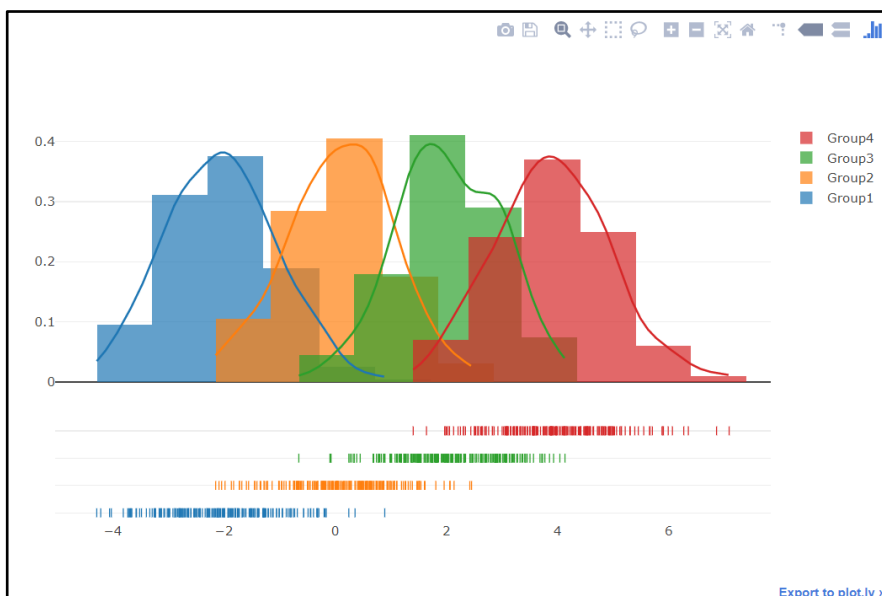
**dist2.py** vergleicht vier ähnliche Diagramme, die jeweils aus einem anderen Satz von 200 Zufallszahlen stammen:

```
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')
```



- Eine Normalverteilung würde eine gerade, symmetrische Glockenkurve zeigen. Diese ist hier nicht der Fall. Verteilungsdiagramme sind daher für kleine Datensätze nicht sehr informativ. Wie das nächste Beispiel zeigt.



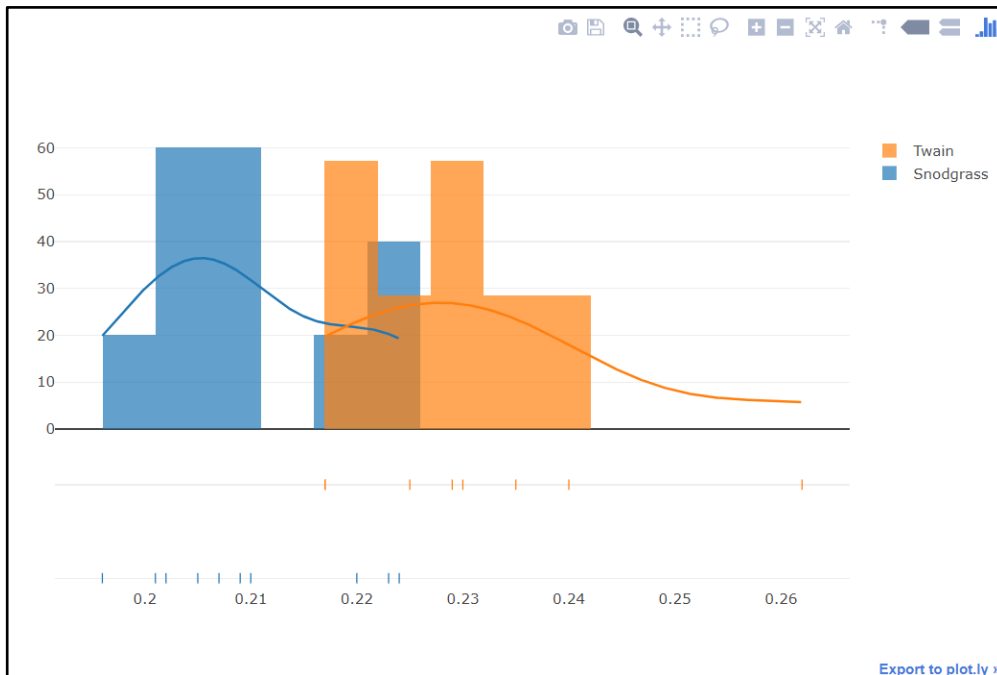
**dist3.py** führt uns zum Beispiel von Mark Twain zurück und bildet zwei Gruppen von nur 10 bzw. 8 Punkten ab.

```
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')
```



- Wir setzen **bin\_size** auf .005, und die Ergebnisse sind bestenfalls verwirrend.
- Box-Plots (Kastendiagramme) waren hier eindeutig die bessere Wahl!

Quellen: <https://plot.ly/python/distplot/> und <https://seaborn.pydata.org/tutorial/distributions.html>

## Heatmaps

Balkendiagramme, Box-Plots, Histogramme und Verteilungsdiagramme (Distplots) helfen, "univariate Verteilungen" in ihrer einfachsten Form zu visualisieren. Das heißt, die Häufigkeit von nur einer Variablen über einen Bereich von Werten oder Kategorien.

Heatmaps bieten eine "multivariate" Darstellung, indem den Datenpunkten eine dritte Dimension - Farbe - hinzugefügt wird. Dies ähnelt der Änderung der Größe der Datenblasen in unseren Blasendiagrammen.

Für diese Beispiele beziehen wir Temperaturdaten für den gleichen Zeitraum von einer Woche im Jahr 2010 von drei US-amerikanischen Wetterstationen: Santa Barbara (Kalifornien), Yuma (Arizona) und Sitka (Alaska). Die Rohdaten wurden von der Website des US-amerikanischen *Climate Reference Network* (USCRN) abgerufen:

<https://www1.ncdc.noaa.gov/pub/data/uscrn/products/hourly02/2010/>

Wir haben die Daten auf drei Spalten (Datum, Uhrzeit, Durchschnittstemperatur) reduziert, eine Spalte für „Tag“ hinzugefügt und alle Aufzeichnungen mit Ausnahme dieser einen Woche (1. bis 7. Juni) entfernt. Die resultierenden Dateien sind **SantaBarbaraCA.csv**, **YumaAZ.csv** und **SitkaAK.csv**.

Für den Anfang erstellen wir einfache Heatmaps für jeden Datensatz (**heat1.py**, **heat2.py** und **heat3.py**) und nehmen die Standardparameter von plotly. Die Temperaturen sind in Grad Celsius angegeben.

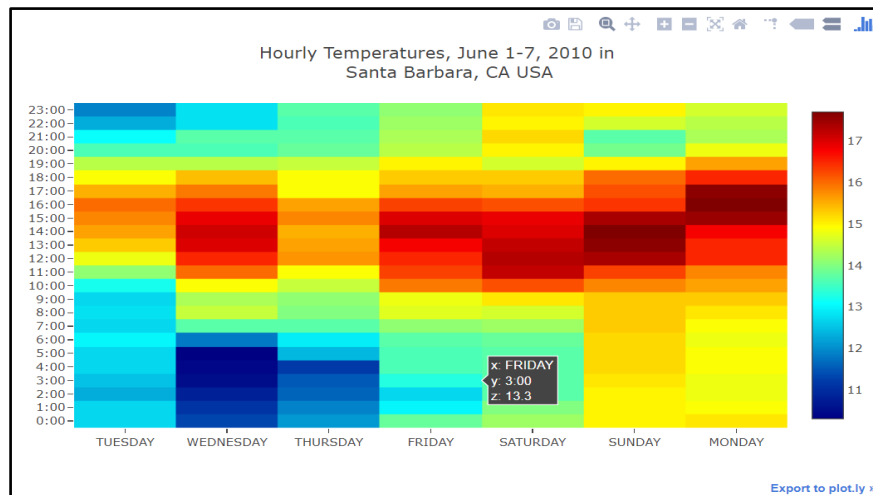
**heat1.py** erstellt eine Heatmap aus **SantaBarbaraCA.csv**:

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

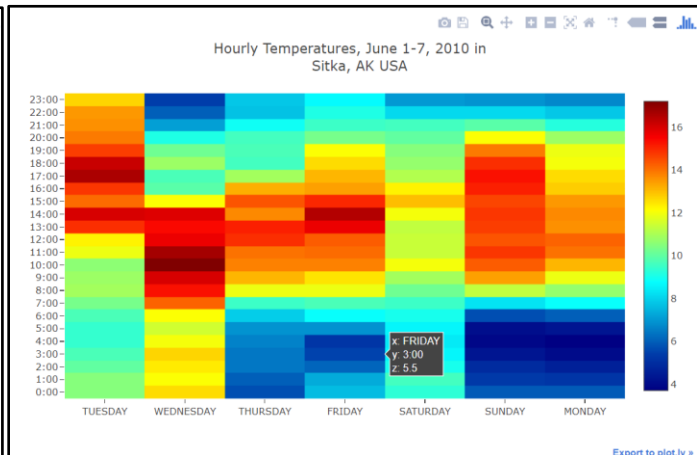
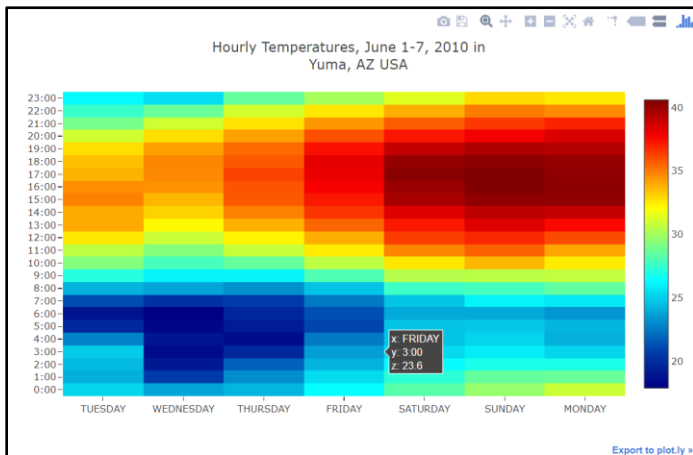
df = pd.read_csv('../data/2010SantaBarbaraCA.csv')

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** und **heat3.py** bilden ähnliche Heatmaps YumaAZ.csv und SitkaAK.csv ab:



Obwohl alle drei Heatmaps ziemlich ähnlich aussehen (warm am Tag, kalt in der Nacht) sind die Temperaturbereiche jeweils recht unterschiedlich.

Bei **heat4.py** passieren mehrere Dinge:

Wir importieren das plotly-Werkzeugmodul (*tools*), um eine Darstellung mit subplots zu erstellen.

Wir fügen jeder Spur *zmin*- und *zmax*-Werte hinzu.

Betrachtet man die Rohdaten, so betrug die niedrigste in Sitka, Alaska gemessene Temperatur 3,7°C, und die höchste gemessene Temperatur in Yuma, Arizona, betrug 40,6°C (105° Fahrenheit).

Für diesen Bereich setzen wir min auf 5 und max auf 40.

In der Unteransicht setzen wir `shared_yaxes = True`. Dadurch werden die Stundenmarkierungen nur auf der linken Seite und nicht neben jeder Darstellung angezeigt.

Anstatt Daten und Layout in einer Grafik umständlich zu kombinieren, wie wir es in der Vergangenheit getan haben, greifen wir direkt mit `fig['layout'].update()` auf das Layout zu.

## heat4.py

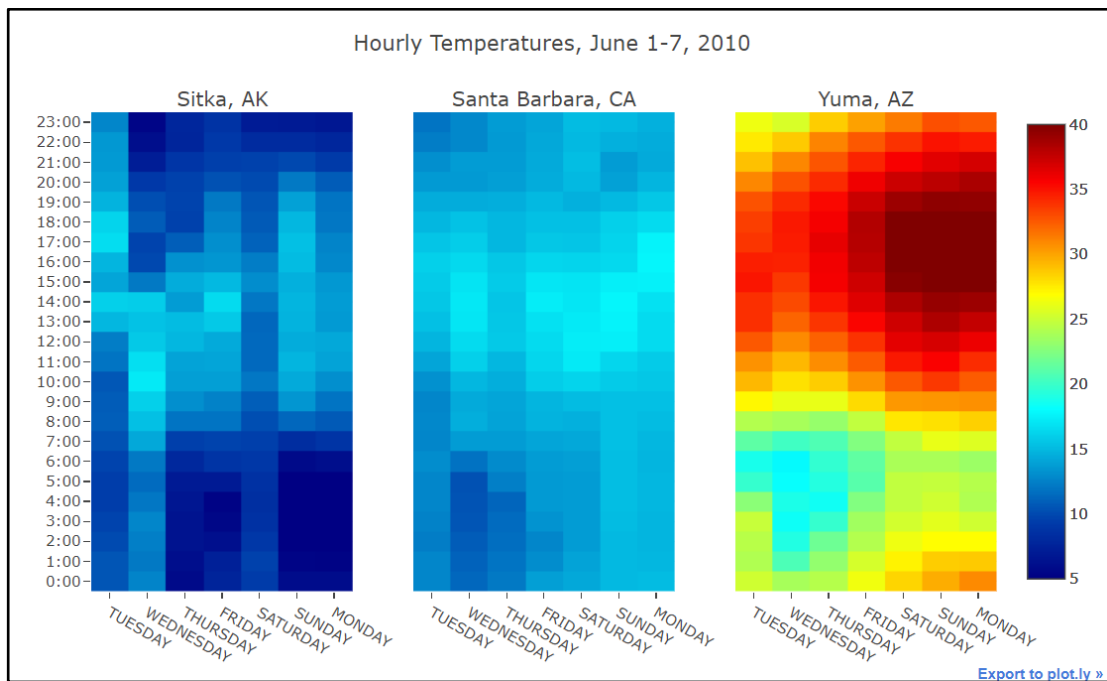
```
import plotly.offline as pyo
import plotly.graph_objs as go
from plotly import tools
import pandas as pd

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')
```



Bei diesem letzten Diagramm sehen wir Daten aus drei verschiedenen Regionen, wobei zum Vergleich die gleiche Skala nebeneinander verwendet wird.

Nicht schlecht!

Quelle: <https://plot.ly/python/heatmaps/>

Datenquellen: <https://www1.ncdc.noaa.gov/pub/data/uscrn/products/hourly02/2010/>

## Übungen: Plotly Grundlagen

### EX1-Scatterplot.py

Ziel: Erstelle ein Streudiagramm aus 1000 zufälligen Datenpunkten.

Die Werte für die X-Achse sollten aus einer Normalverteilung kommen, mit `np.random.randn(1000)`.

y-Achsen-Werte sollten aus einer uniformen Verteilung über [0,1] kommen, mit `np.random.rand(1000)`.

### Ex2-Linechart.py

Ziel: Entwickle mit der Datei **2010YumaAZ.csv** ein Liniendiagramm, in dem die Temperaturdaten für sieben Tage in einer Grafik dargestellt werden. Du kannst ein Listenverständnis (list comprehension) verwenden, um jeden Tag einer eigenen Linie zuzuordnen.

Siehe <https://pandas.pydata.org/pandas-docs/stable/generated/pandas.unique.html> für Hilfe bei der Suche nach eindeutigen Werten mit Pandas

### Ex3-Barchart.py

Ziel: Erstelle ein gestapeltes Balkendiagramm aus der Datei **../data/mocksurvey.csv**. Beachte dabei, dass Fragen im Index angezeigt werden (und für die X-Achse verwendet werden sollten), während Antworten als Spaltenbeschriftungen angezeigt werden.

Extra Credit: Erstelle ein horizontales Balkendiagramm!

Weitere Informationen dazu erhältst du unter <https://plot.ly/python/horizontal-bar-charts/>

### Ex4-Bubblechart.py

Ziel: Erstelle ein Blasendiagramm, das drei Eigenschaften aus dem Datensatz **mpg.csv** vergleicht. Sie umfassen:

'mpg', 'Zylinder', 'Verdrängung' 'PS', 'Gewicht', 'Beschleunigung', 'Modell\_Jahr', 'Ursprung', 'Name'  
( 'mpg', 'cylinders', 'displacement' 'horsepower', 'weight', 'acceleration', 'model\_year', 'origin', 'name' )

### Ex5-Boxplot.py

Ziel: Erstelle einen DataFrame für das Abalone-Dataset (**../data/abalone.csv**). Verwende zwei unabhängige Stichproben unterschiedlicher Größe aus dem Feld "rings". HINWEIS: `np.random.choice(df['rings'],10,replace=False)` nimmt 10 zufällige Werte

Verwende ein Boxdiagramme, um zu zeigen, dass die Stichproben aus derselben Grundgesamtheit stammen.

### Ex6-Histogram.py

Ziel: Erstelle ein Histogramm, das das „Längen“-Feld aus dem Abalone-Dataset (**../data/abalone.csv**) darstellt.

Lege den Bereich von 0 bis 1 mit einer Balkengröße von 0,02 fest.

### Ex7-Distplot.py

Ziel: Entwickle mithilfe des Iris-Datasets ein Verteilungsdiagramm, das die Längen der Blütenblätter jeder Klasse vergleicht.

Datei: **../data/iris.csv**


Felder: 'sepal\_length', 'sepal\_width', 'petal\_length', 'petal\_width', 'class'

Klassen: 'Iris-setosa', 'Iris-versicolor', 'Iris-virginica'

### Ex8-Heatmap.py

Zielsetzung: Verwende das Dataset "flights" des Seaborn-Moduls von Python (um eine Heatmap mit folgenden Parametern zu erstellen, siehe <https://seaborn.pydata.org/generated/seaborn.heatmap.html>):

x-Achse = "Jahr"



y-Achse = "Monat"

Z-Achse (Farbe) = 'passengers' ("Passagiere")

## Lösungen: Plotly Grundlagen

Die Lösungen der Übungsaufgaben finden sich im Abschnitt [Code](#).



## Dash Basics - Layout

### Einführung: Dash Grundlagen

Wenn du es noch nicht getan hast, befolge nun die Installationsanweisungen in Lektion 4. In Kürze, die Dash-Installationsschritte lauten wie folgt:

```
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 bestehen aus zwei Teilen. Der erste Teil ist das **Layout** und beschreibt, wie die Anwendung aussieht. Der zweite Teil beschreibt die **Interaktivität** der Anwendung.

Die gute Nachricht ist, dass zur Verwendung von Dash kein HTML- oder CSS-Kenntnisse erforderlich sind. Die meisten HTML-Tags werden als Python-Klassen bereitgestellt. Wenn du beispielsweise `html.H1(children='Hello Dash')` in dein Dash-Skript eingibst, wird das HTML-Element `<h1>Hello Dash</h1>` angezeigt.

Dash bietet zwei verschiedene Komponentenbibliotheken. Der obige Code stammt aus `dash_html_components`, die für jedes HTML-Tag eine Komponente enthält, wie die Überschrift `H1` der ersten Ebene. Eine weitere Bibliothek, `dash_core_components`, bietet übergeordnete, interaktive Komponenten, die mit JavaScript, HTML und CSS über die React.js-Bibliothek generiert werden.

Dash-Komponenten - egal, ob HTML oder core - werden vollständig durch Keyword-Attribute (Schlüsselwort) beschrieben. Dash ist *deklarativ*: man beschreibt die Anwendung hauptsächlich anhand dieser Attribute.

### Dash Layout

Erstelle eine einfache HTML-Seite mit Balkendiagramm. Erstelle eine Datei **layout1.py** und gebe Folgendes ein:

```
# -*- 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'
            }
        }
    )
])
```

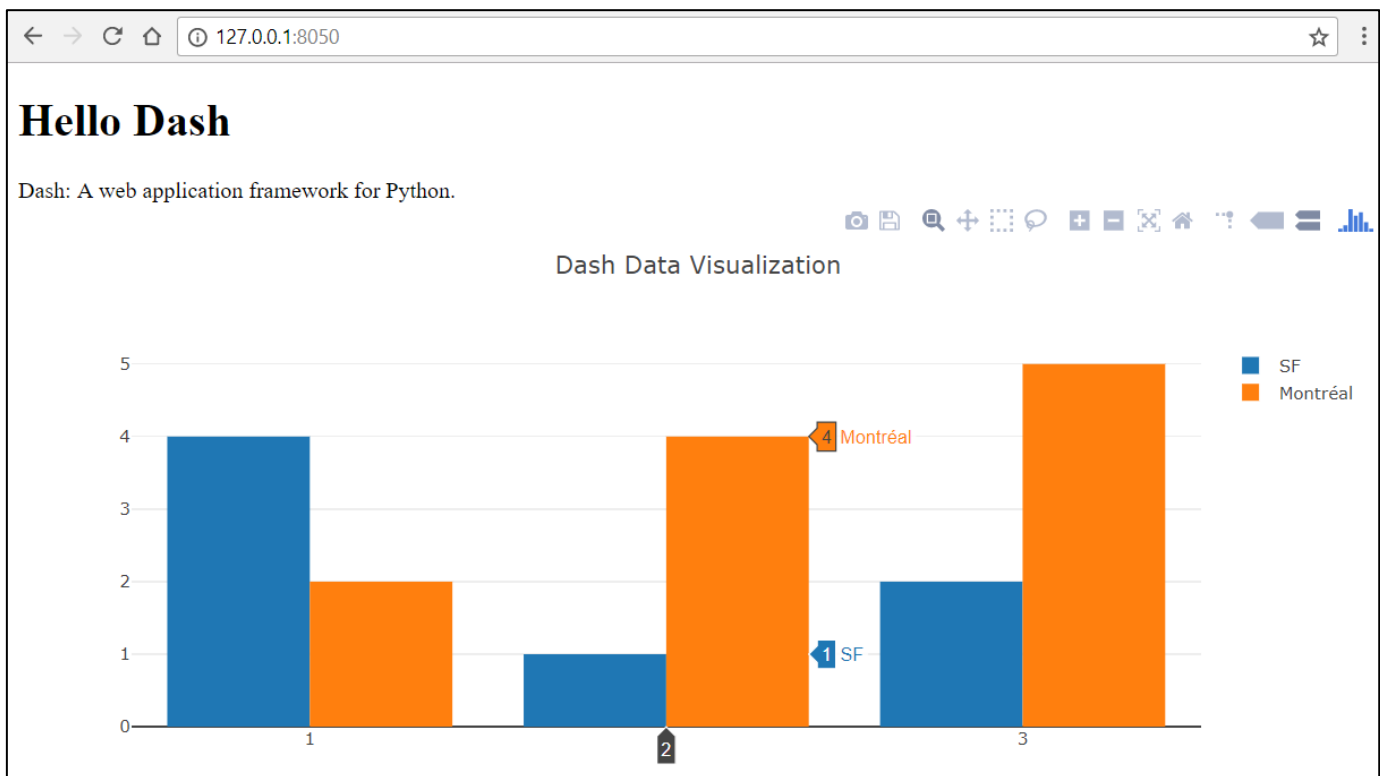


```
)  
])  
  
if __name__ == '__main__':  
    app.run_server()
```

Starte die App mit

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

und gehe auf <http://127.0.0.1:8050/> in deinem Webbrowser. Hier sollte folgende Seite erscheinen:



Hinweis: Die interaktiven Teile werden nur angezeigt, wenn sich der Cursor über einer der Balken befindet.

**FEHLERSUCHE:** Einige Texteditoren codieren utf nicht richtig. Wenn Sie eine Fehlermeldung erhalten, die besagt

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

ist das Problem wahrscheinlich mit dem erweiterten Unicode-Zeichen in u'Montréal' verbunden. Ändere dies an der Stelle in ein normales e.

Speichere die Datei und führe sie wie oben beschrieben erneut aus.

Die Schritte aus diesem kurzen Beispiel sind folgende:

1. `# -*- coding: utf-8 -*-`

Dies gibt die Kodierung für die Python-Datei an. Siehe [PEP 0263 - Defining Python Source Code Encodings](#) für weitere Details.

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

Wir importieren Dash und seine beiden Komponentenbibliotheken.

3. `app = dash.Dash()`

Wir starten eine Dash-Anwendung. "App" ist nur ein praktischer Name für unsere Dash-Instanz.

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

Hier beginnen wir mit der Definition des Applikationslayouts.

H1 und Div sind Komponentenattribute, die entsprechenden HTML-Tags zugeordnet sind.

H1 haben wir schon gesehen; es erstellt eine Überschrift der ersten Ebene.

Div erstellt einen `<div>` Tag, der einem HTML-Container ähnelt.

children ist eine Eigenschaft (property) von HTML-Komponenten (dieses Keyword wird später verwendet, wenn wir unser Dashboards interaktiv gestalten. Standardmäßig ist dies die erste Eigenschaft, die aufgeführt wird. Daher müssen wir unserem Code nicht extra `children=` hinzufügen.

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'`  
    `}`  
`)`

Das ist alles eine Core-Komponente! Die Schlüsselwortattribute "data" und "layout" sollten bekannt sein, da sie direkt aus Plotly stammen. Diagrammkomponenten haben die Eigenschaft `figure` anstatt `children`. Das ist die gleiche Kennung wie in Plotly.

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

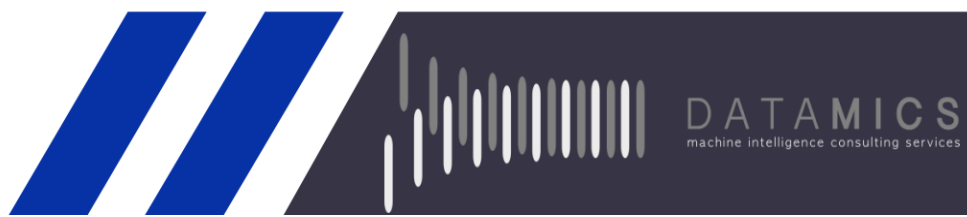
Dieser letzte Abschnitt startet einen lokalen Server *nur*, wenn **layout1.py** als Skript ausgeführt wird.

Wenn wir diese Datei in ein anderes Programm importieren, wird diese Codezeile ignoriert.

Beachte auch, dass **Layout1.py** im Gegensatz zu Plotly ein aktives Skript ist, für das ein lokaler Webserver im Hintergrund laufen muss. Wenn du Änderungen an **layout1.py** vornehmen solltest, die verhindern, dass es ordnungsgemäß ausgeführt werden kann, zeigt es eine Fehlermeldung an und fährt den Server herunter.

Dash verwendet **Flask** als Server-Backend. Du kannst **debug=True** an den Serveraufruf übergeben, um einige Diagnosedienste zu aktivieren (Bitte nicht in der Produktivumgebung!). Der Code würde so aussehen:

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



Bevor wir fortfahren, nehmen wir einige Änderungen am HTML-Code auf unserer Seite vor. Öffne eine neue Datei mit dem Namen **layout2.py** und kopiere den Inhalt von layout1 nach layout2. (Du kannst auch **layout1.py** einfach kopieren, wenn du möchtest).

Als nächstes füge den unten gezeigten Code in Schwarz ein (der Originalcode wird in Blau angezeigt):

```
# -*- 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'
}

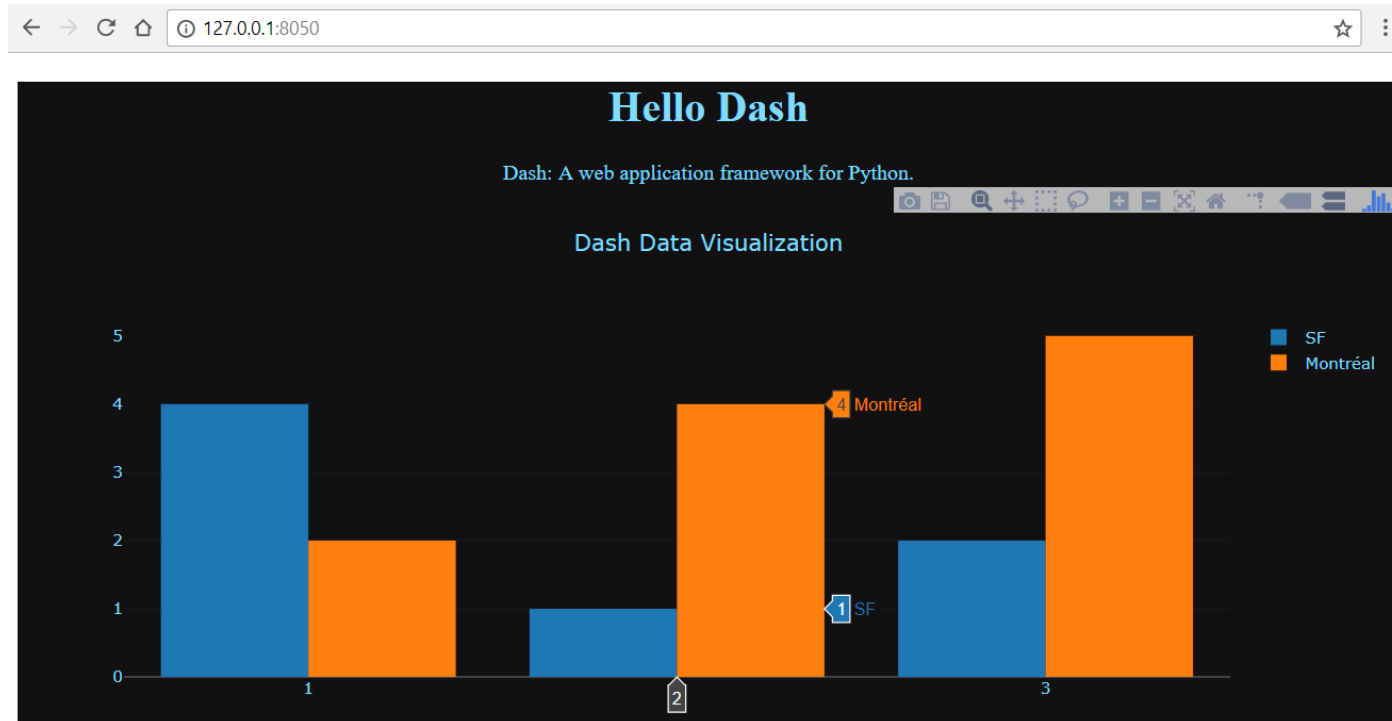
app.layout = html.Div(children=[
    html.H1(
        children='Hello Dash',
        style={
            'textAlign': 'center',
            'color': colors['text']
        }
    ),

    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 dieser Version fügen wir ein Dictionary mit Farbstilen hinzu. Diese werden in den, jeder Komponente hinzugefügten Stileigenschaften, referenziert. Führe am Rechner `python layout2.py` aus, Folgendes erscheint:



In diesem Beispiel haben wir die Inline-Stile der Komponenten `html.Div` und `html.H1` mit der `style`-Eigenschaft geändert.

`html.H1('Hello Dash', style={'textAlign':'center', 'color':'#7FDFDF'})` wird in der Dash-Anwendung als `<h1 style="text-align:center; color:#7FDFDF">Hello Dash</h1>` angezeigt.

Es gibt einige wichtige Unterschiede zwischen den Attributen `dash_html_components` und den HTML-Attributen:

1. Die `style` property in HTML ist eine durch Semikolons getrennte Zeichenfolge. In Dash können Sie einfach ein Dictionary angeben.
2. Die Keys im `style` dictionary sind `camelCased`. Statt `text-align` ist es `textAlign` (Text ausrichten).
3. Die HTML-Klasse (`class`) ist in Dash Klassenname (`className`). Wir werden das in den nächsten Beispielen sehen.
4. Die untergeordneten Elemente (children) eines HTML-Tags werden über das untergeordnete Schlüsselwortargument angegeben (`children` keyword argument). Konventionell ist dies immer das *erste* Argument und wird daher oft weggelassen.  
`html.H1(children='Hello Dash')` ist das gleiche wie `html.H1('Hello Dash')`.

Das wars schon! Du hast gerade dein erstes Dashboard erstellt. Als Nächstes konvertieren wir ein einfaches Plotly-Diagramm in Dash.

## Ein einfaches Plotly-Diagramm mit Dash zu Dashboard konvertieren

Für diese Übung kehren wir zu unserem `scatter3.py`-Skript zurück. Dies beinhaltete ein Streudiagramm von 100 zufälligen Datenpunkten. Wir lassen den Zufallsgenerator laufen, damit alle das gleiche Ergebnis sehen.

Öffne nun eine neue Datei und nenne sie **plotly1.py**. Gebe anschließend den folgenden Code ein:

```
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()

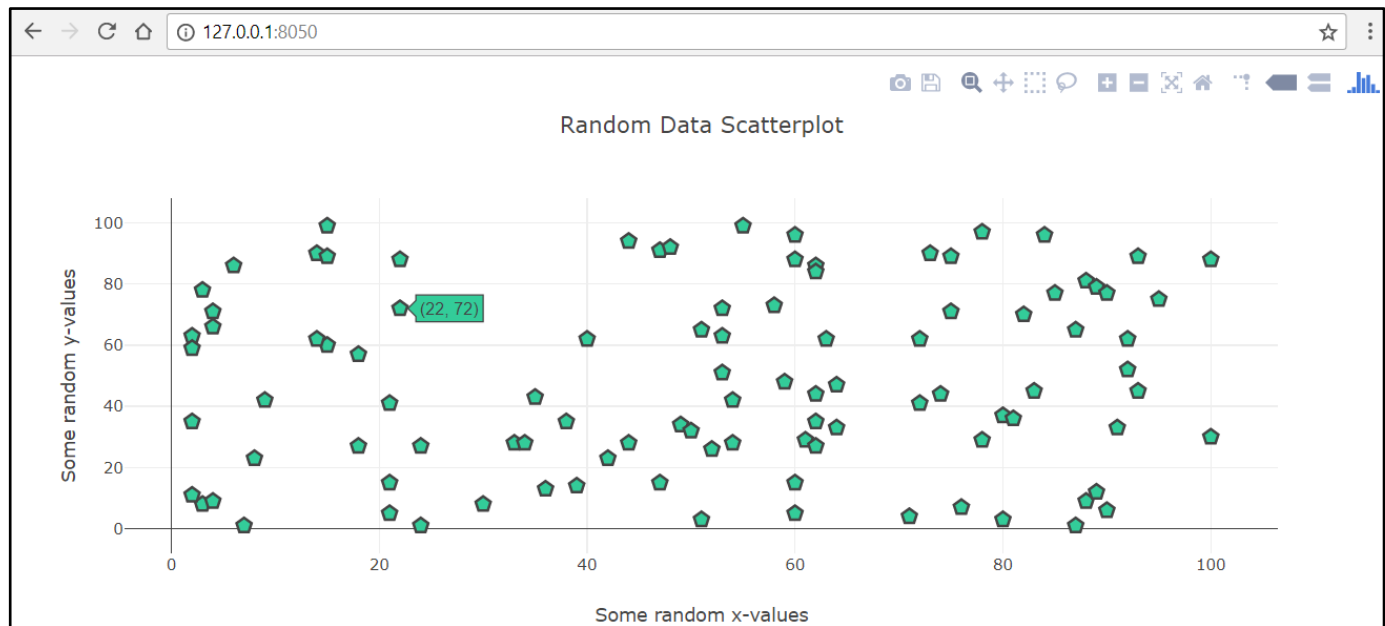
np.random.seed(42)
random_x = np.random.randint(1,101,100)
random_y = np.random.randint(1,101,100)

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()
```

Wie du siehst, handelt es sich bei dem Großteil davon um den gleichen Code wie in **scatter3.py**. In Dash enthält die Bibliothek **dash\_core\_components** eine Komponente namens **Graph**, die interaktive Datenvisualisierungen mithilfe der JavaScript-Grafikbibliothek von Plotly darstellt. Tatsächlich ist das Argument **figure** in der Komponente **dcc.Graph** dasselbe **figure** Argument, das auch von Plotly verwendet wird.

Nachdem du die Datei gespeichert hast, führe **python plotly1.py** an deinem Rechner aus. Öffne den Browser erneut unter <http://127.0.0.1:8050/>. Nun sollte folgende Seite erscheinen:



## Übung: Erstelle ein einfaches Dashboard

Für diese Übung erstellen wir ein Dashboard ähnlich dem oben genannten Streudiagramm. Nur dieses Mal importieren wir einige Daten.

**Old Faithful** ist ein Kegel-Geysir im Yellowstone National Park in Wyoming, Seit dem Jahr 2000 waren die Intervalle zwischen den Ausbrüchen zwischen 125 Minuten lang, mit durchschnittlich 90 - 92 Minuten. Es ist nicht möglich, Voraus mehr als einen Ausbruch vorherzusagen. Old Faithful ist derzeit Er hat zwei Ausbruchslängen, entweder eine lange (über 4 Minuten) oder seltener eine kurze Dauer (etwa 2 ½ Minuten). Kurze Ausbrüche führen zu Intervall von ca. einer Stunde und lange Ausbrüche zu einem Intervall von 1/2 Stunden.

Erstelle für diese Übung ein Dashboard, das **OldFaithful.csv** aus dem Datenverzeichnis importiert und ein Streudiagramm anzeigt.

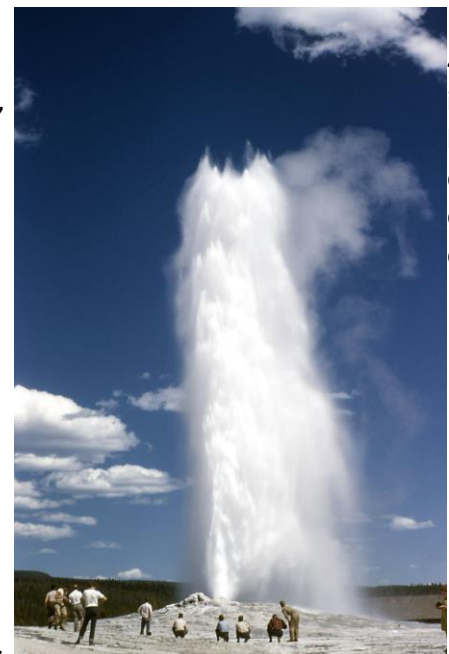
Der Datensatz besteht aus 3 Feldern (D, Y, X), wobei

D = Aufnahmedatum im Monat (im August),

X = Dauer des aktuellen Ausbruchs in Minuten (bis 0,1 Minuten)

Y = Wartezeit bis zum nächsten Ausbruch in Minuten (zur nächsten Minute).

Bild: Eruption des Old Faithful in 1948 [https://en.wikipedia.org/wiki/Old\\_Faithful#/media/File:OldFaithful1948.jpg](https://en.wikipedia.org/wiki/Old_Faithful#/media/File:OldFaithful1948.jpg)



USA.  
44 und  
im  
bimodal.  
etwas  
einem  
etwa 1



## Lösung: Erstelle ein einfaches Dashboard

Dies ist unser Lösungsvorschlag

```
# 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()

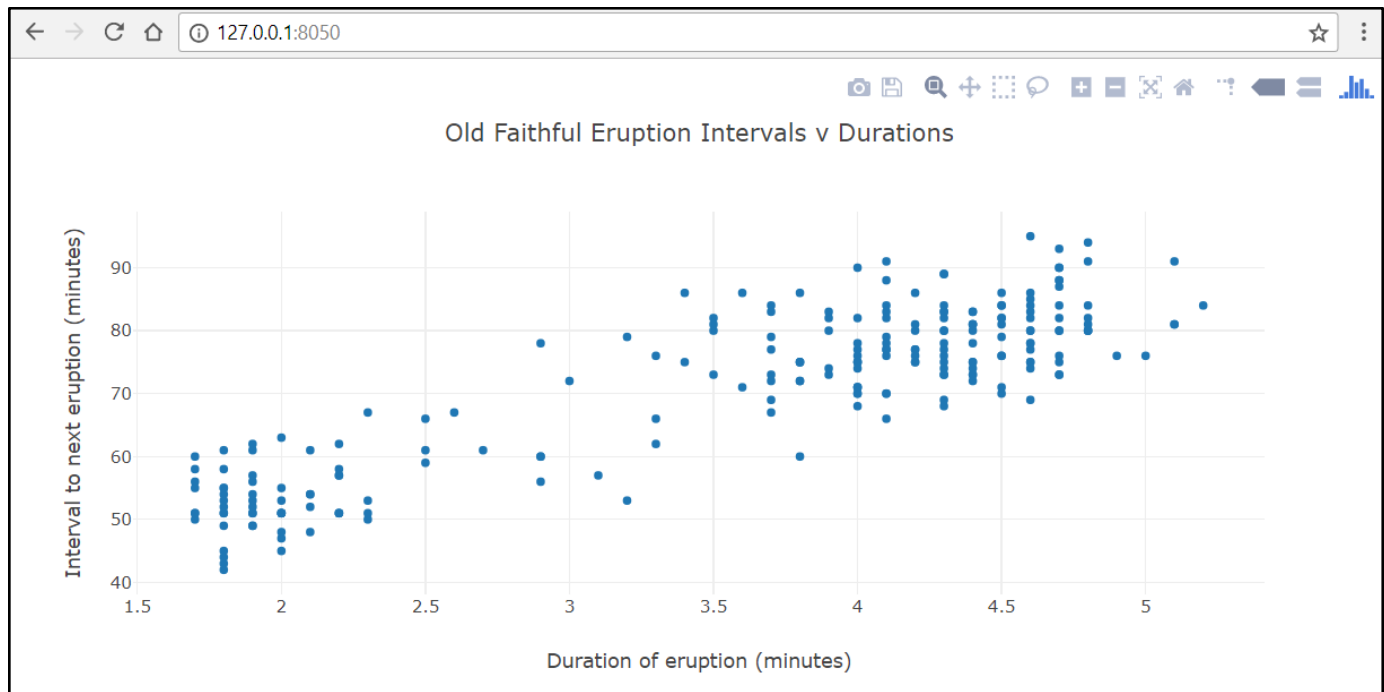
# Create a DataFrame from the .csv file:
df = pd.read_csv('../data/OldFaithful.csv')

# 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()
```



Wenn alles klappt, dann sollte dir dein fertiges Dashboard folgende Seite öffnen:





## Dash Komponenten

Dash-Komponenten werden von zwei Bibliotheken bereitgestellt: **dash\_html\_components**, die wir normalerweise als **html** abkürzen, und **dash\_core\_components**, normalerweise als **dcc** abgekürzt. Normalerweise beschreiben **HTML**-Komponenten das Layout der Seite, einschließlich der Anordnung und Ausrichtung verschiedener Diagramme. **dcc**-Komponenten beschreiben die einzelnen Diagramme selbst.

### HTML Komponenten

Eine Beschreibung der HTML-Komponenten von Dash findest du unter: <https://dash.plot.ly/dash-html-components>

Gängige Komponenten sind:

- |                                 |                                |   |
|---------------------------------|--------------------------------|---|
| • <b>html.Div</b> ([ section ]) | applies CSS to section of page | <code>&lt;div&gt; &lt;/div&gt;</code>     |
| • <b>html.P</b> ()              | paragraph                      | <code>&lt;p&gt; &lt;/p&gt;</code>         |
| • <b>html.H1</b> ('text')       | heading (level 1)              | <code>&lt;h1&gt; &lt;/h1&gt;</code>       |
| • <b>html.Label</b> ('text')    | label                          | <code>&lt;label&gt; &lt;/label&gt;</code> |

HTML-Elemente und Dash-Klassen sind im Wesentlichen identisch, es gibt jedoch einige wesentliche Unterschiede:

- Die **style** property ist ein dictionary
- Eigenschaften im style dictionary sind camelCased
- Die Klasse (*class*) wird in **className** umbenannt
- Style properties in Pixeleinheiten können nur als Zahlen ohne die Einheit *px* angegeben werden

Lass uns ein Beispiel anschauen:

```
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})
```

Dieser Dash-Code führt zu diesem HTML Code:

```
<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>
```

Erstelle nun bitte eine Datei mit dem Namen **HTMLComponents.py**, und füge den folgenden Code hinzu, um ein Gefühl dafür zu bekommen, wie **dash\_html\_components** auf einer Seite angeordnet werden kann:

```
import dash
import dash_html_components as html

app = dash.Dash()

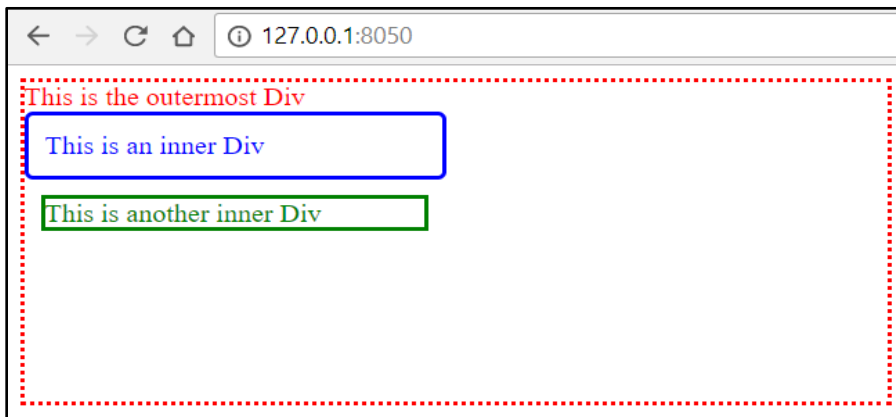
app.layout = html.Div([
    'This is the outermost Div',
    html.Div(
        'This is an inner Div',
        style={'color':'blue', 'border':'2px blue solid', 'borderRadius':5,
              'padding':10, 'width':220}
    ),
    html.Div(
        'This is another inner Div',
        style={'color':'green', '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()
```

Man sieht nun dass **'border':'2px blue solid'** die Kurzform von **'borderWidth':2, 'borderColor':'blue', 'borderStyle':'solid'**

**'borderRadius':5** muss separate aufgeführt werden

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Man sieht schön, wie der Stil für jedes Div individuell angewendet werden kann, indem Farbe, Rahmen, Füllfarbe und Ränder bereitgestellt werden.

## Core Komponenten

Eine vollständige Beschreibung der Core-Komponenten von Dash findest du unter <https://dash.plot.ly/dash-core-components>

Hier beschreiben wir einige nützliche Werkzeuge.

Erstelle eine Datei mit dem Namen **CoreComponents.py** und füge den folgenden Code hinzu.

Halte diese Datei bereit - Vielleicht möchtest du Komponenten hinzufügen, die du für nützlich hältst!

### 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': '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()

```

Wir fügen den Schieberegler (Slider) in einen HTML-Absatz ein **html.P()**, um zu verhindern, dass die darunter liegenden Optionsfelder (Radio Buttons) die Slider-Markierungen überschreiben.

Führe das Skript aus:

Hier haben wir drei eingebaute Komponenten:

- [Dropdown](#)
- [Slider](#)
- [Radio Items](#)

Es gibt jedoch noch viele andere, wie:

- [RangeSlider](#)
- [Input](#)
- [Textarea](#)
- [Checklist](#) (horizontale and vertikale Checkboxes)
- [DatePickerSingle](#)
- [DatePickerRange](#)

Diese Komponenten tun nicht wirklich was, bis wir die interaktiven Funktionen von Dash wie zum Beispiel [Callbacks](#) verwenden.

Bevor wir dorthin gelangen, untersuchen wir die [Markdown](#)-Komponente (ein Shortcut (Verknüpfung) zum Schreiben von HTML-Text) und die [Help\(\)](#)-Methode von Dash.

## Markdown

Während Dash HTML über die Bibliothek [dash\\_html\\_components](#) verfügbar macht, kann es mühsam sein, deine Kopie in HTML zu schreiben. Zum Schreiben von Textblöcken kannst du die [Markdown](#)-Komponente in der Bibliothek [dash\\_core\\_components](#) verwenden.

Erstelle eine Datei namens **markdown.py** und füge den folgenden Code ein:

```
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.

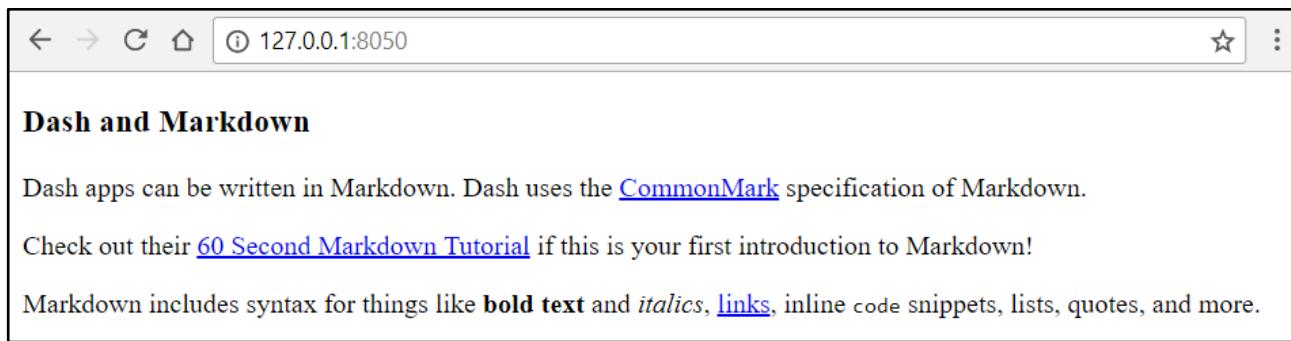
Check out their [60 Second Markdown Tutorial](http://commonmark.org/help/)
if this is your first introduction to Markdown!

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()
```

Nun kannst du das Programm ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Hier siehst du wie die 3 Hashtags `###` auf der Seite in ein `<h3>` Tag übersetzt werden.

Beachte auch, dass der Zeilenumbruch zwischen „Dash-Apps kann in Markdown geschrieben werden (“Dash apps can be written in Markdown.”) und „Dash verwendet die Markdown-Spezifizierung [CommonMark] (<http://commonmark.org/>)“ (“Dash uses the [CommonMark](<http://commonmark.org/>) specification of Markdown.”) ignoriert wird. Um einen neuen Absatz auf der Seite zu beginnen, ist eine leere Zeile erforderlich.

Mehr Infos dazu findest du hier: <https://dash.plot.ly/dash-core-components/markdown>

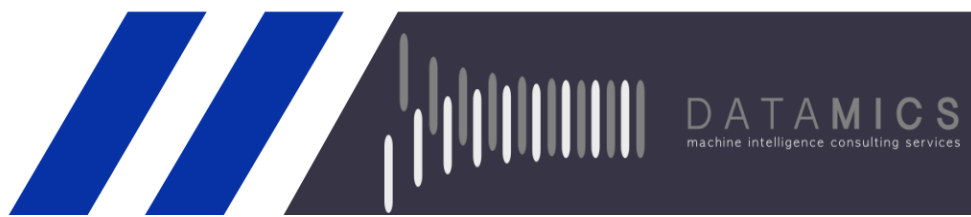
## Nutzung von Help() mit Dash

Dash-Komponenten sind deklarativ: Jeder konfigurierbare Aspekt dieser Komponenten wird während der Instanziierung als Keyword-Argument festgelegt. Rufe in der Python-Bedienung `help` (Hilfe) zu einer der Komponenten auf, um mehr über eine Komponente und ihre verfügbaren Argumente zu erfahren.

```
>>> 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.

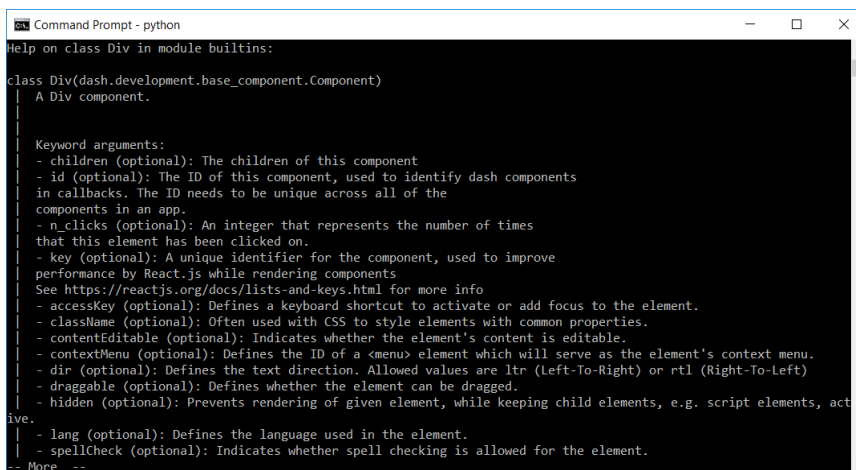
## Schreiben von Help() in HTML:

Als Alternative zum Lesen einer Volltext-Hilfedatei in der Bedienleiste kannst du mit *pydoc* in eine .html-Datei schreiben.

Geben Sie am Bildschirm (nicht in Python oder einer IDE) **pydoc -w dash\_html\_components.Div** ein.

Dadurch wird im selben Verzeichnis eine Datei mit dem Namen **dash\_html\_components.Div.html** erstellt, die im Browser angezeigt werden kann. Dies funktioniert für jede Dash-Komponente!

So sieht **help** in der Python-Bedienung aus:



```

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, active.
|   - lang (optional): Defines the language used in the element.
|   - spellCheck (optional): Indicates whether spell checking is allowed for the element.
-- More --

```

Dies ist die gleiche Datei, die im Browser angezeigt wird:

```
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 - Interaktive Komponenten

### Interactive Komponenten Übersicht

Der erste Teil dieses Tutorials behandelte das Layout von Dash-Apps:

- Das **Layout** einer Dash-App beschreibt, wie die App aussieht. Es ist ein hierarchischer Baum von Komponenten.
- Die Bibliothek **dash\_html\_components** enthält Klassen für alle HTML-Tags. Die Keyword-Argumente beschreiben die HTML-Attribute wie style, className und id.
- Die Bibliothek **dash\_core\_components** generiert übergeordnete Komponenten wie Steuerelemente und Diagramme.

Im zweiten Teil des Kurses wird beschrieben, wie die Dash-App interaktiv gemacht werden kann.



Beginnen wir mit einem einfachen Beispiel.

## Verbinden von Komponenten mit Callbacks

### Einen Callback zu einer Komponente hinzufügen

In dieser Übung fügen wir einem Eingabefeld einen Callback hinzu und zeigen die eingegebenen Daten als sofortige Ausgabe auf demselben Bildschirm an.

Erstelle eine Datei mit dem Namen **callback1.py** und füge den folgenden Code hinzu:

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

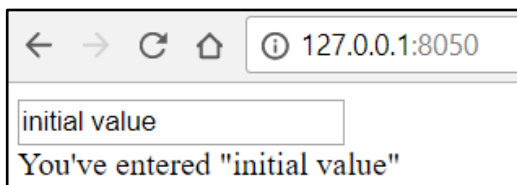
app = dash.Dash()

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()
```


Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Gib nun etwas in das Eingabefeld ein. Sofort sollte sich die Ausgabe ändern und die Eingabe anzeigen!



Lass uns durch die einzelnen Schritte gehen, die hier passiert sind:

- 
1. Wir richten unser **dcc.Input** auf die übliche Weise ein, außer dass wir ihm eine **id** zugewiesen haben und danach eine weitere Div mit einer zugewiesenen **id** hinzufügen (jeweils `'my-id'` und `'my-div'`).
  2. **app.callback** wird als *decorator*-Funktion über `update_output_div` aufgerufen. Die "Inputs" und "Outputs" unserer Anwendungsoberfläche werden deklarativ durch den **app.callback**-Dekorator beschrieben.  
**Weitere Informationen zu Python-Dekoratoren findest du unter:**  
[https://en.wikipedia.org/wiki/Python\\_syntax\\_and\\_semantics#Decorators](https://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators)
  3. In **@app.callback** sind **Output** und **Input** abgekürzte Formen von **dash.dependencies.Output** und **dash.dependencies.Input**. Wir haben sie nach ihrem Namen aus `dash.dependencies` importiert haben.
  4. In Dash sind die Ein- und Ausgaben unserer Anwendung lediglich die Eigenschaften einer bestimmten Komponente. In diesem Beispiel ist unsere Eingabe die Eigenschaft **"value"** der Komponente, die die ID `"my-id"` hat. Unsere Ausgabe ist die **"children"**-Eigenschaft der Komponente mit der ID `"my-div"`.
  5. Wenn sich eine Input-Eigenschaft ändert, wird die Funktion, die der Callback-Dekorator einschließt, automatisch aufgerufen. Dash stellt der Funktion den neuen Wert der Input-Eigenschaft als Eingabeargument bereit, und Dash aktualisiert ebenfalls die Eigenschaft der Output-Komponente mit dem, was von der Funktion zurückgegeben wurde.
  6. Die Keywords **component\_id** und **component\_property** innerhalb von **Output** und **Input** sind optional (für jedes dieser Objekte gibt es nur zwei Argumente). Wir haben sie aus Gründen der Klarheit hier aufgenommen, aber wir werden sie hier aus Gründen der Kürze und Lesbarkeit, von hier an, weglassen.
  7. Verwechsle bitte nicht das **dash.dependencies.Input**-Objekt in **app.callback** mit dem **dash\_core\_components.Input**-Objekt in **app.layout**. Ersteres wird nur in den Callbacks verwendet, und letzteres ist eine tatsächliche Komponente.
  8. Beachte auch, dass wir im Layout keinen Wert für die untergeordnete (**children**) Eigenschaft der **my-div**-Komponente festlegen. Beim Start der Dash-App werden automatisch alle Callbacks mit den Anfangswerten der Input-Komponenten aufgerufen, um den Anfangsstatus der Output-Komponenten aufzufüllen. Wenn du in diesem Beispiel etwas wie `html.Div(id='my-div', children='Hello world')` angegeben hast, wird es beim Start der App überschrieben.

Es ist wie mit dem Programmieren mit Microsoft Excel: Wenn sich eine Eingabezelle ändert, werden alle von dieser Zelle abhängigen Zellen automatisch aktualisiert. Dies wird als "reaktive Programmierung" bezeichnet.

Erinnerst du dich daran, wie jede Komponente vollständig durch ihre Keyword-Argumente beschrieben wurde? Diese Eigenschaften (properties) sind jetzt wichtig. Mit der Dash-Interaktivität können wir diese Eigenschaften über eine Callback-Funktion dynamisch aktualisieren. Häufig aktualisieren wir die **untergeordneten (children)** Elemente einer **HTML**-Komponente, um einen neuen Text anzuzeigen oder die **Abbildung** einer **dcc.Graph**-Komponente, um neue Daten anzuzeigen. Wir können jedoch auch den Stil (**style**) einer Komponente oder sogar die verfügbaren **Optionen** einer **dcc.Dropdown**-Komponente aktualisieren !

## Zwei Komponenten mit Callbacks verbinden

Das nächste Beispiel stammt aus dem Online-Tutorial von Dash und ist ziemlich komplex. Es führt einige Layout-Funktionen ein, die wir bisher noch nicht gesehen haben, wie eine logarithmische X-Achse. Ziel ist es, dass ein interaktiver **Slider** einen **Graphen** auf derselben Seite aktualisiert. Wir werden einen Dash-Datensatz verwenden, der online verfügbar ist:

<https://raw.githubusercontent.com/plotly/datasets/master/gapminderDataFiveYear.csv>

Erstelle eine Datei mit dem Namen **callback2.py** und füge den folgenden Code hinzu:

```
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'},
```



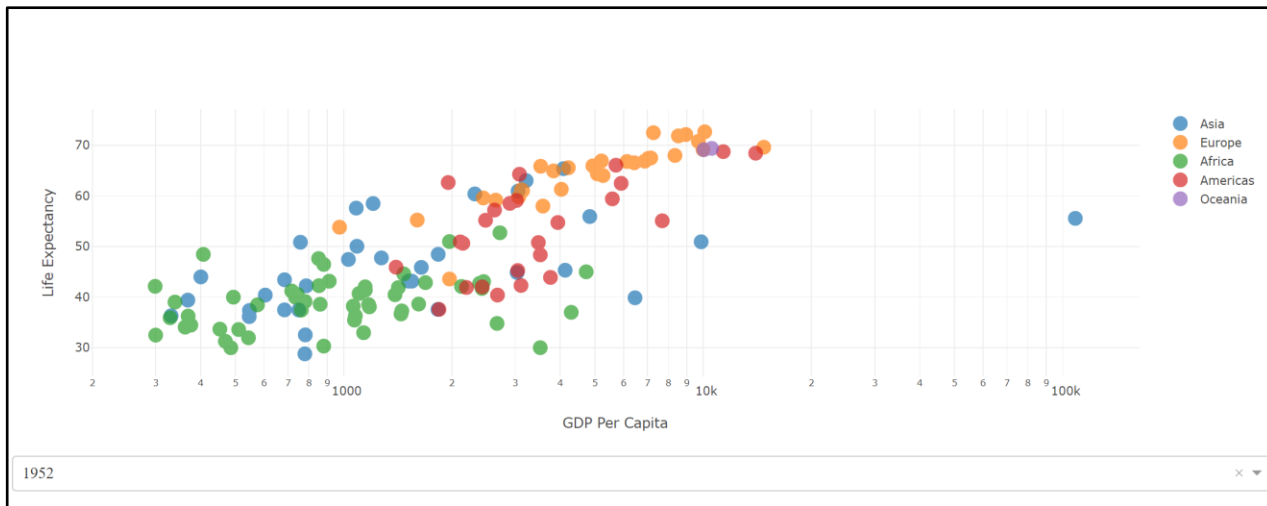
```

        yaxis={'title': 'Life Expectancy'},
        hovermode='closest'
    )
}

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

```

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Du kannst den Mauszeiger über jeden Datenpunkt bewegen, um die Daten zu Land, Kontinent und Achsendaten anzuzeigen. Noch wichtiger ist, du kannst die Dropdown-Liste verwenden, um die angezeigte Grafik zu verändern!

### In Bezug auf Stil

Bevor wir die Verbindung (connectivity) besprechen, betrachten wir einige der ausgewählten Stiloptionen:

- Die X-Achse ist logarithmisch und wird mit zunehmenden Werten dichter
- Wir verwenden die Pandas `.unique()` -Methode, um die Jahre für das Dropdown zu extrahieren (ähnlich unserer Plotly [Liniendiagramm-Übung!](#)).

### In Bezug auf Konnektivität

In diesem Beispiel ist die Eigenschaft "**value**" des **Dropdown**-Menüs der *Input* der App und der *Output* der App ist die Eigenschaft "**figure**" des **Diagramms**. Immer wenn sich der Wert des Dropdown-Menüs ändert, ruft Dash die Callback-Funktion **update\_figure** mit dem neuen Wert auf. Die Funktion filtert den DataFrame mit diesem neuen Wert, erstellt ein **figure**-Objekt und gibt es an die Dash-Anwendung zurück.

In diesem Beispiel gibt es einige schöne Vorlagen:

- Wir verwenden die Pandas-Bibliothek zum Importieren und Filtern von Datensätzen im Speicher.
- Wir laden unseren DataFrame zu Beginn der App: `df = pd.read_csv('...')`. Dieser DataFrame-`df` befindet sich im globalen Status der App und kann innerhalb der Callback-Funktionen gelesen werden.

- Das Laden von Daten in den Speicher kann teuer sein. Durch das Laden von Abfragedaten beim Start der App anstelle der Callback-Funktionen stellen wir sicher, dass diese Operation nur beim Start des App-Servers ausgeführt wird. Wenn ein Benutzer die App besucht oder mit der App interagiert, befinden sich diese Daten (df) bereits im Speicher. Nach Möglichkeit sollten teure Initialisierungen (wie das Herunterladen oder Abfragen von Daten) im globalen Anwendungsbereich der App und nicht innerhalb der Callback-Funktionen ausgeführt werden.
- Der Callback ändert die Originaldaten nicht, sondern erstellt lediglich Kopien des DatenFrames, die durch Pandas-Filter gefiltert werden. Das ist wichtig: die Callbacks dürfen niemals Variablen außerhalb des Gültigkeitsbereichs verändern. Wenn deine Callbacks den globalen Status ändern, kann sich die Sitzung eines Benutzers auf die Sitzung des nächsten Benutzers auswirken. Wenn die App auf mehreren Prozessen oder Threads bereitgestellt wird, können diese Änderungen nicht in allen Sitzungen angezeigt werden.

## Multiple Inputs

Eingabeparameter werden als Liste an den Callback-Dekorator übergeben. Aus diesem Grund können wir mehrere Inputs in unser Dashboard aufnehmen um denselben Output über eine Callback-Funktion zu beeinflussen. In diesem Beispiel verwenden wir das mpg.csv-Dataset, um zwei Eingabekomponenten anzuzeigen - beides Dropdown-Listen -, sodass wir die X- und Y-Achsen-Features aus unserem Dataset festlegen können.

Erstelle eine Datei mit dem Namen **callback3.py** und füge den folgenden Code hinzu:

```
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'})
    ], style={'width': '100%', 'display': 'flex', 'flex-direction: row'})
```



```

    ]),

    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'
        )
    }

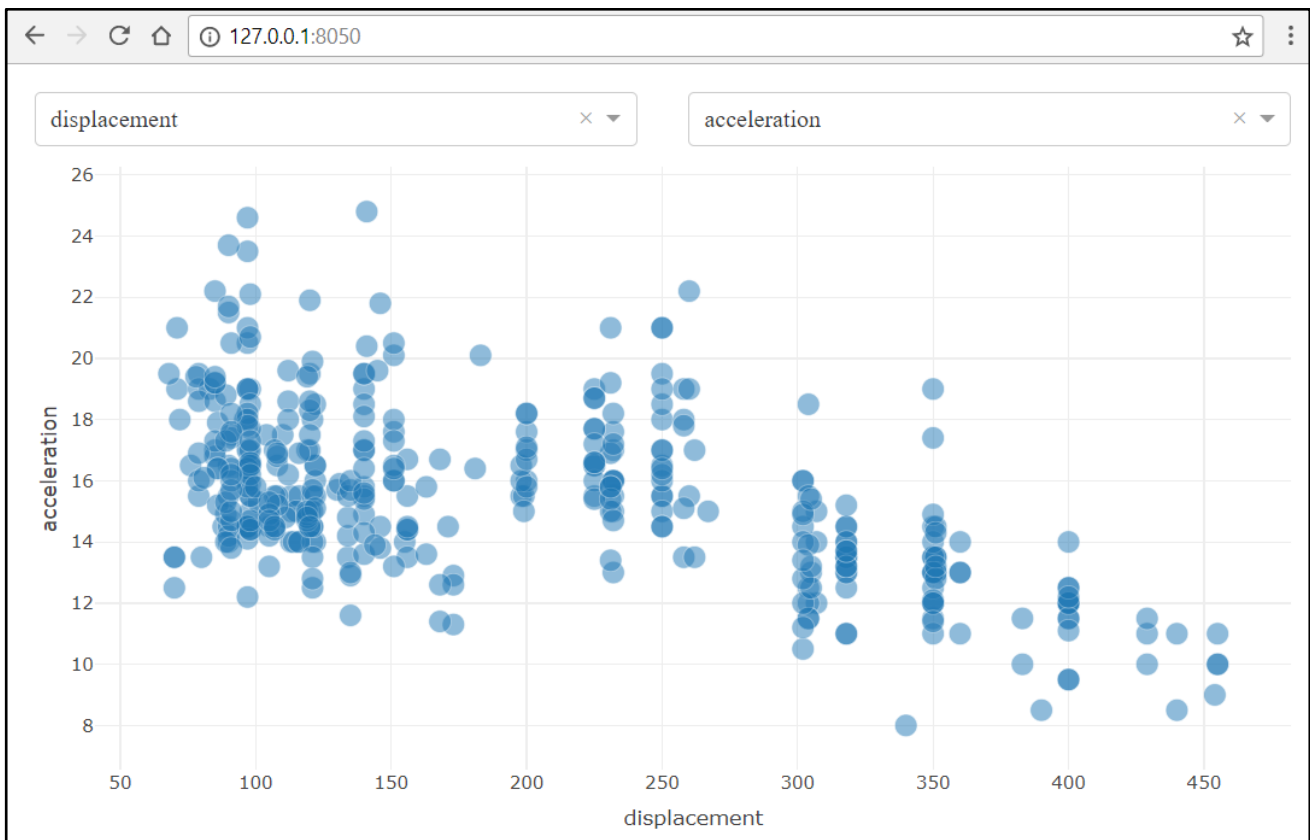
if __name__ == '__main__':
    app.run_server()

```

Schauen wir uns an, was hier passiert ist:

- Wir setzen eine Variable Merkmale (*features*) gleich mit den entsprechend den Spaltennamen in unserem Datensatz. Eine Alternative wäre, es in einer Datensatzspalte auf einen wiederkehrenden Wert festzulegen. Beachte, dass das Setzen dieser Variablen optional ist - wir können `df.columns` genauso einfach ausführen, wo auch immer *features* verwendet wird.
- Im **Layout**bereich ist nichts Neues passiert. In einem Div legen wir unsere zwei Dropdown-Komponenten fest, gefolgt von unserem **Graphen**.
- Beachte jedoch, dass **app.callback** jetzt *zwei* Eingabeparameter hat, einen für jedes Dropdown-Menü.
- Abgesehen von zwei Inputs ist das zurückkehrende Update jedoch relativ unkompliziert. Wir erstellen ein Streudiagramm mit unseren X- und Y-Achsen.

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Du kannst nun einen der Dropdown-Einträge ändern und sofort ändern sich die X- und Y-Achsen-Merkmale!

Als schnelle Formatierungsoption: Was wäre, wenn unsere Funktionen großgeschrieben werden sollten? Obwohl der Name der Dataset-Spalte kleingeschrieben ist und „displacement“ lautet, wie wird in unserer Grafik sowohl in der Dropdown-Liste als auch im Achsentitel „Displacement“ daraus? Da gibt es eine schnelle Lösung:

```
...
app.layout = html.Div([
    html.Div([
        html.Div([
            dcc.Dropdown(
                id='xaxis',
                options=[{'label': i.title(), 'value': i} for i in features],
                value='displacement'
            )
        ])
        ...

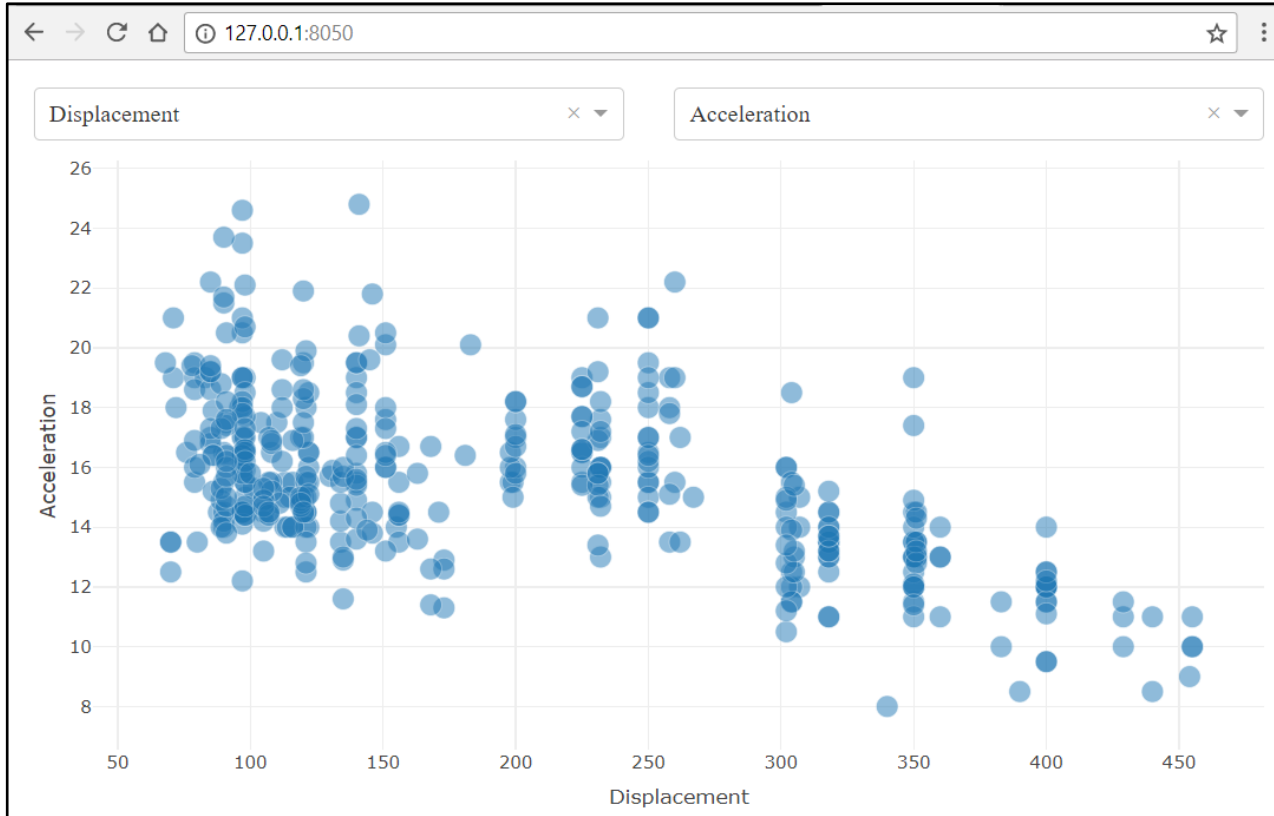
        html.Div([
            dcc.Dropdown(
                id='yaxis',
                options=[{'label': i.title(), 'value': i} for i in features],
                value='acceleration'
            )
        ])
    ])
    ...
def update_graph(xaxis_name, yaxis_name):
    ...
    'layout': go.Layout(
        xaxis={'title': xaxis_name.title()},
```



```
yaxis={'title': yaxis_name.title()},  
margin={'l': 40, 'b': 40, 't': 10, 'r': 0},  
hovermode='closest'
```

...

Was zu dieser Darstellung führt:



Ein anderes Beispiel für mehrere Inputs findest du in der Dash-Dokumentation unter <https://dash.plot.ly/getting-started-part-2>. Hier werden nicht nur Dropdown-Listen angezeigt, sondern auch Optionsfelder (Radio Buttons) und Schieberegler, die als Auswahlmöglichkeiten gleichzeitig für die gleiche Grafik verwendet werden können.



## Multiple Outputs

Jede Dash-Callback-Funktion kann nur eine einzige Output-Eigenschaft aktualisieren. In den obigen Beispielen zeigen wir dir, wie du mehrere **Inputs** innerhalb eines Eingabelistenparameters übergibst. Um mehrere **Outputs** zu aktualisieren, schreiben wir einfach mehrere Funktionen.

In diesem Beispiel richten wir zwei Sets mit Radio Buttons und zwei separate Output-Bereiche ein.

Als Nächstes fügen wir einen dritten Output hinzu, der durch die *Kombination* der ausgewählten Optionsfelder bestimmt wird.

Erstelle eine Datei mit dem Namen **callback4.py** und füge den folgenden Code hinzu:

```
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})

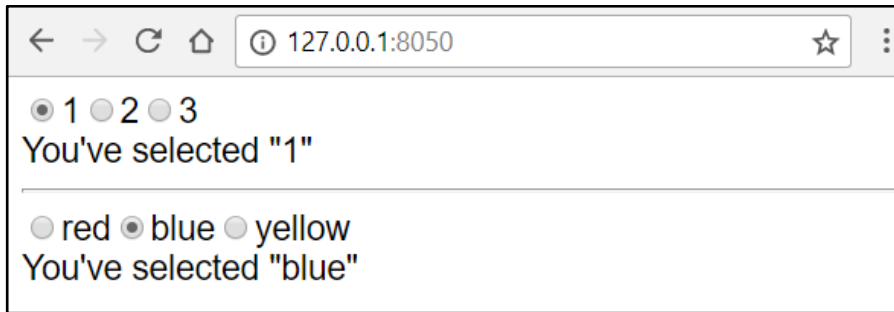
@app.callback(
    Output('wheels-output', 'children'),
    [Input('wheels', 'value')])
def callback_a(wheels_value):
    return 'You\'ve selected {}'.format(wheels_value)

@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()
```



Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



The screenshot shows a web browser window with the address bar set to `127.0.0.1:8050`. The page content includes two sets of radio buttons. The first set has three options: `1`, `2`, and `3`, with `1` selected. Below this, the text `You've selected "1"` is displayed. The second set has three options: `red`, `blue`, and `yellow`, with `blue` selected. Below this, the text `You've selected "blue"` is displayed.

Das Ändern einer Auswahl wirkt sich auf einen unabhängige Output aus!

Quelle: <https://dash.plot.ly/getting-started-part-2>

Lass uns dieses Beispiel erweitern und den Output von beiden Inputs abhängig machen.  
Kopiere **callback4.py** und nenne die Datei **callback5.py**. Füge den folgenden Code hinzu (fett dargestellt):

```
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()
```



Wenn du nun das Skript ausführst, wird bei den Standardwerten **1** und **blue** ein Bild eines blauen Einrads angezeigt. Ändert man nun den Input, ändert sich auch das angezeigte Bild!

Hier wurden einige interessante Techniken vorgestellt:

- Zum jetzigen Zeitpunkt liefert Dash statische Dateien nicht anstandslos. Um auf der Festplatte gespeicherte Bilder anzuzeigen, ist eine Konvertierung in base64 erforderlich. Hierfür haben wir eine Konvertierungsfunktion mit dem Namen "**encode\_image**" definiert und diese dann in unserer Callback-Funktion verwendet.
- Für unseren **Output**, ist "**display-image**" die Komponenten-ID und "**src**" die Komponente-Eigenschaft (`component_property`), die wir beeinflussen.
- Wir haben Pandas verwendet, um den Namen unserer Bilddatei aus dem Datensatz mithilfe einer bedingten Auswahl (`conditional selection`) zu erhalten. Beachte bitte, dass die Tabelle nur den Dateinamen enthält, nicht den Pfad. Dafür setzen wir innerhalb der Callback-Funktion unsere eigene Pfadvariable. Auf diese Weise können wir unser Skript an jede andere Dateistruktur anpassen.
- Zum jetzigen Zeitpunkt benötigt **html.img** ein **height** = Argument, aber kein **alt** =, um einen alternativen Text bereitzustellen, falls ein Bild nicht abgerufen werden kann.

## Übung: Interaktive Komponenten

Für diese Übung möchten wir zwei oder mehr integrale Inputs verwenden und ihr Produkt ausgeben. Sei dabei ruhig kreativ! Du kannst dazu Radio Buttons, Dropdown-Listen und sogar einen Schieberegler verwenden, um zwei Inputwerte zu erhalten. Verwenden Sie einen Callback, um das Produkt der beiden Werte zurückzugeben. Vergiss dabei nicht, jeder Komponente IDs zuzuweisen. Viel Glück!

## Lösung: Interaktive Komponenten

Für unsere vorgeschlagene Lösung haben wir einen Bereichs-Schieberegler (RangeSlider) ausgewählt, um unsere beiden Werte zu erhalten. Wichtig ist zu wissen, dass RangeSlider beide Werte als eine einzige Liste zurückgeben:

```
# 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()
```

• Bereich (min, max + 1) funktioniert hier nicht. Es muss hart codiert sein, es sei denn, min & max werden außerhalb des **Layouts** definiert.



## Callbacks mit Dash State kontrollieren

In den vorherigen interaktiven Beispielen haben wir gesehen, wie sich Eingaben sofort auf Outputs auswirken. Sobald Werte eingegeben werden, wird die Seite aktualisiert, um die Änderungen anzuzeigen.

Was wäre, wenn wir warten wollten, bevor die Seite angezeigt wird? Was wäre, wenn wir Zeit für eine Reihe von Änderungen hätten, bevor wir sie einreichen? An dieser Stelle setzt [dash.dependencies.State](#) ein. Dash bietet die Möglichkeit, Änderungen zu speichern und auf Befehl zurückzusenden. Betrachten wir dieses sehr einfache Beispiel für Input/Output mit einem 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()
```

Sobald du ein Zeichen in das Eingabefeld eingibst, wird dies unten als HTML-Header angezeigt.

Nun fügen wir eine Schaltfläche "Senden" hinzu und speichern Zeichen, bis die Schaltfläche gedrückt wird:

**callback6a.py** (der zusätzliche Code ist **fettgedruckt**)

```
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()
```

Jetzt ist unsere Eingabe das Klicken auf das Element **html.Button**. Der in das Eingabefeld eingegebene Wert wird innerhalb von State gespeichert und nicht an unseren Output übergeben, bis der Input einen Schaltflächenklick registriert!

Was ist also **n\_clicks**? Es stellt die Anzahl der Klicks, die während der Sitzung aufgetreten sind fest und speichert sie. Wir können dies als Teil unseres Outputs anzeigen, wenn wir wollen:


**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()
```





Jedes Mal, wenn du einen neuen Wert übermittelst, wird auf der Seite nun angezeigt, wie oft auf die Schaltfläche geklickt wurde!

Es ist interessant, dass *jedem* HTML-Element eine 'n\_clicks'-Eigenschaft zugewiesen werden kann.

Quelle: <https://dash.plot.ly/state>



# Interaktion mit Visualisierungen

## Einleitung zu Interaktion mit Visualisierungen

Der erste Teil dieses Tutorials behandelte das **Layout** von Dash-Apps:

- Das **Layout** einer Dash-App beschreibt, wie die App aussieht. Es ist ein hierarchischer Baum von Komponenten.
- Die Bibliothek **dash\_html\_components** enthält Klassen für alle HTML-Tags. Die Keyword-Argumente beschreiben die HTML-Attribute wie `style`, `className` und `id`.
- Die Bibliothek **dash\_core\_components** generiert übergeordnete Komponenten wie Steuerelemente und Diagramme.

Der zweite Teil behandelte Callbacks:

- Die Komponenten **dash.dependencies.Input** und **dash.dependencies.Output** monitoren die Seite ständig und aktualisieren bei Bedarf eine Output-Anzeige, -Diagramm oder andere Seiteninhalte.
- Dash unterstützt multiple Inputs und Outputs.
- Du kannst Input-Daten mithilfe von **dash.dependencies.State** anhalten und bei Bedarf mit einem Button oder einem anderen HTML-Element senden.

In diesem nächsten Abschnitt gehen wir nochmals auf **dash\_core\_components.Graph** ein steigen tiefer in Plotly-Diagramme ein.

Quelle: <https://dash.plot.ly/interactive-graphing>

## Hover Over Daten (mit der Maus drüberfahren)

Überfliege nochmals kurz „[Ein einfaches Plotly-Diagramm zu einem Dashboard konvertieren mit Dash](#)“. Wir haben ein Streudiagramm angezeigt, das aus zufälligen Datenpunkten besteht. Wenn sich der Cursor über einem einzelnen Punkt befindet, werden die Daten für diesen Punkt (die Werte für die X-Achse und die Y-Achse) als Text angezeigt.

Hier zeigen wir dir, wie einfach ein Schweben über einen Datenpunkt einen anderen Teil der Darstellung sofort beeinflussen kann!

Wir beginnen mit dem Erstellen eines 3x3-Streudiagramm aus der Datei **wheels.csv**. Es sei daran erinnert, dass es 3 x-Achsenwerte gibt (Rot, Gelb, Blau) und 3 Y-Achsen-Werte (1,2,3).

Als Nächstes fügen wir einen Callback hinzu, der **'hoverData'** übernimmt und diese Daten als **JSON**-Objekt auf dem Bildschirm anzeigt.

Erstelle eine Datei mit dem Namen **hover1.py** und füge den folgenden Code hinzu:

```
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)
```

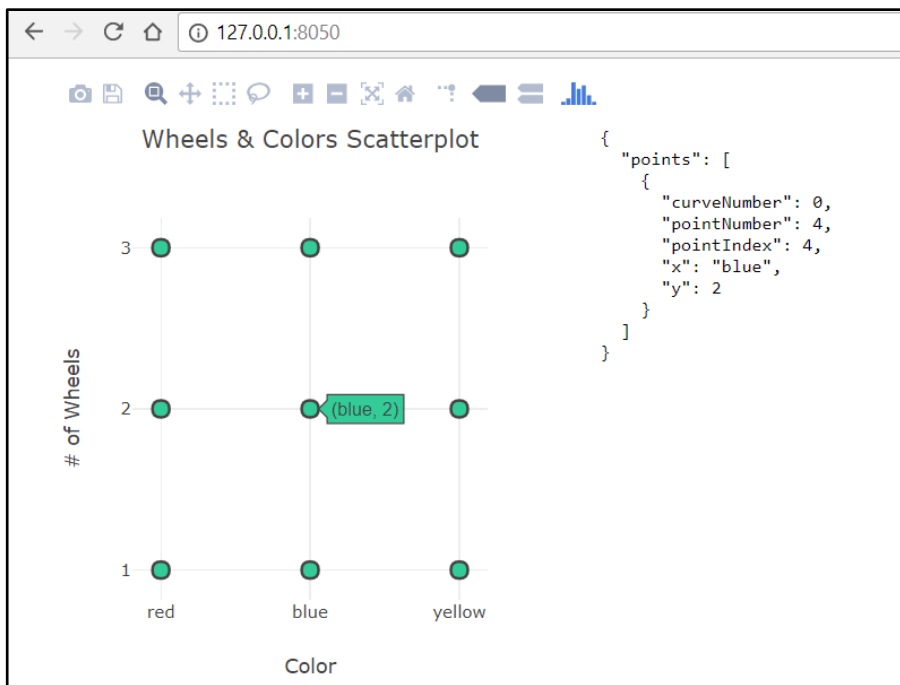


```
if __name__ == '__main__':  
    app.run_server()
```

Einige wichtige Dinge:

- wir importieren json (**import json**), damit wir die erfassten hoverData als **json.dumps**-Objekt anzeigen können.
- Wir bezeichnen unser Ausgabefeld als "**hover-data**" - dies kann jedoch alles sein.
- Unser Input von 'wheels-plot' erfasst '**hoverData**' und wir übergeben **hoverData** dann in unsere Callback-Funktion.  
*Diese Tags sind wichtig!*
- Wir zeigen die hoverData innerhalb eines **html <pre>**-Tags an, das eine Vorformatierung ermöglicht (die wir nicht verwendet haben) und den Inhalt in einer Schriftart mit fester Breite darstellt, wobei Leerzeichen und Zeilenumbrüche erhalten bleiben.
- Wir haben '**nticks**': 3 zur **Layout**-Eigenschaft der y-Achse hinzugefügt. Ohne sie wären die Ticks [1, 1.5, 2, 2.5, 3]

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



- Beachte bitte, dass der Anfangsstatus der JSON-Ausgabe „null“ ist und sich erst ändert, wenn ein Punkt mit der Maus überfahren wird.

Aber wie nutzen wir hoverData? Wir machen dies durch eine Reihe von Aufrufen des Dictionary!  
Lass uns die X- und Y-Achsenwerte eines Datenpunkts extrahieren und die zugehörige Bilddatei abrufen.

Kopiere **hover1.py** und nenne die Datei **hover2.py**. Füge den folgenden Code hinzu (fett dargestellt):

```
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])

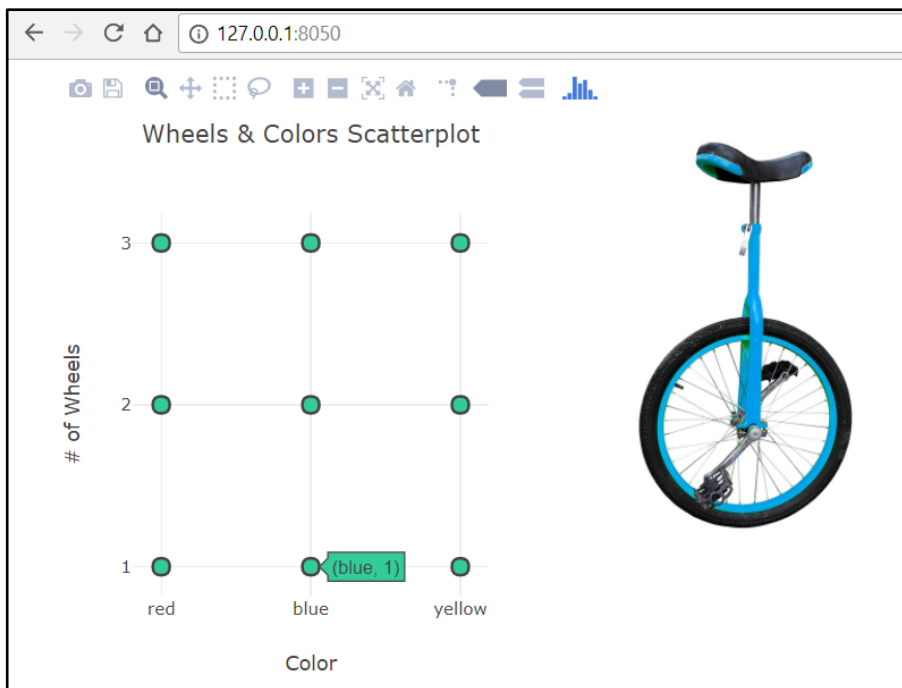
if __name__ == '__main__':
    app.run_server()

```

Die blau markierten Abschnitte dienen lediglich dem Handling von Bildern (erinnerst du dich daran, dass wir zuerst Dateien in base64 konvertieren mussten?).

Schau, wie wir `hoverData['points'][0]['y']` verwenden, um den Wert der y-Achse zu erhalten. Wir geben das in Pandas ein, um die entsprechende Bilddatei abzurufen.

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> , beim Überfahren der einzelnen Datenpunkte (1, blau) Folgendes sehen:



## Click Data

Klick-Daten (Click Data) wird fast genauso gehandhabt wie Hover Data - es handelt sich lediglich um ein Attribut des Diagramms, auf das über Verzeichnisaufrufe zugegriffen werden kann.

Kopiere **hover2.py** und nenne die Datei **click1.py**. Füge den folgenden Code hinzu (Änderungen sind **fett** dargestellt):

```
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()
```

Wirklich nur zwei Dinge haben sich geändert:

Wir haben die ID des Ausgabefelds in "**click-image**" geändert, obwohl dies nicht unbedingt erforderlich war.

Anstelle von **hoverData** übergeben wir **clickData** an unsere Callback-Funktion.

Wenn du nun das Skript ausführst, werden die Bilder angezeigt, wenn Datenpunkte angeklickt werden anstatt darüber zu fahren.

Das wars schon! Alles andere - einschließlich des Dictionary-Aufrufs um unserer X- und Y-Achsen-Werte zu erhalten- bleibt gleich.

## Selected Data (Datenauswahl)

Die Datenauswahl verwende das Lasso oder Rechteckwerkzeug in der Menüleiste des Graphen:



und ausgewählte Punkte in der Grafik.

Um zu sehen, wie das aussieht, kopieren wir **hover1.py** und nennen es **select1.py**. Änderungen sind **fett** dargestellt:

```
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()

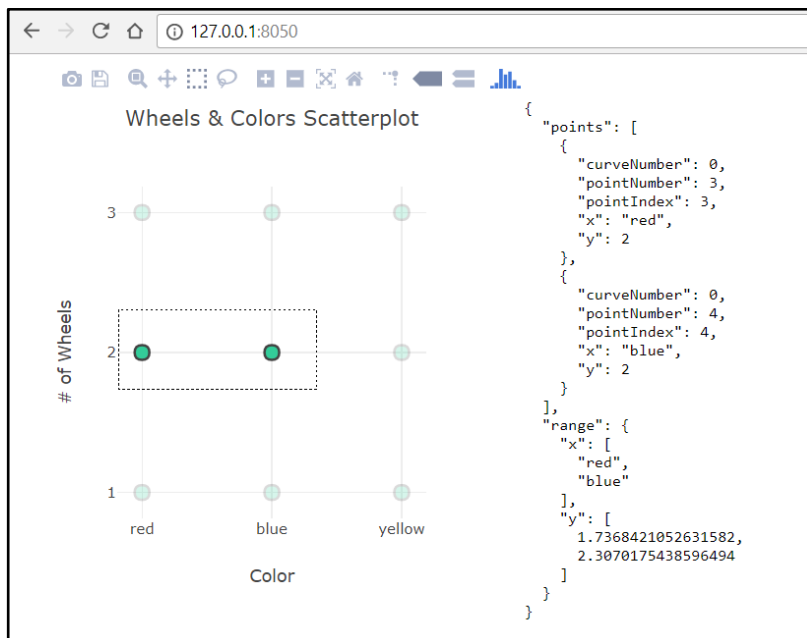
```

Hier haben wir den Input-Parameter in '**selectedData**' geändert.

Blau dargestellt, haben wir auch die Div-Stile von '**float': 'left**' in '**display': 'inline-block**' geändert. Dadurch wird verhindert, dass der längere JSON-Output die Grafik nach unten drückt oder darunter packt.

Führe das Skript aus, öffne einen Browser unter <http://127.0.0.1:8050/> und verwende die Lasso- und Rechteckauswahlwerkzeuge in der Diagrammenüleiste, um Gruppen von Datenpunkten auszuwählen. Du solltest so etwas angezeigt bekommen:





Das zurückkehrende Dictionary hat einen Schlüssel für 'points' und einen weiteren Schlüssel für entweder 'range' oder 'lassoPoints'.

**Punkt**daten sind ähnlich wie das, was wir oben für Hover und Klicken gesehen haben, nur diesmal enthält die Liste ein Dictionary für jeden eingekreisten Punkt.

**Bereichs**daten enthalten 'x' und 'y' Achsbegrenzungen für die Auswahlbox selbst.

**LassoPoints** können eine ziemlich lange Liste sein. Dies sind die (x, y) Koordinatenpaare, die die Auswahlgrenze definieren.

Lass uns das nun nutzen! Ein Problem, das wir bei Streudiagrammen finden, ist, dass es schwierig sein kann, überlappende Datenpunkte zu identifizieren. Die Einstellung der Deckkraft hilft (zwei Punkte, die den gleichen Platz einnehmen, werden dunkler als ein Punkt alleine), aber es ist nicht narrensicher. In diesem Beispiel erstellen wir ein künstliches Dataset, zeichnen Punkte auf und verwenden ausgewählte Daten, um die *Dichte* von Punkten in einem bestimmten Bereich des Diagramms zu bestimmen.

Erstelle eine Datei mit dem Namen **select2.py** und füge den folgenden Code hinzu:

```

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'}),

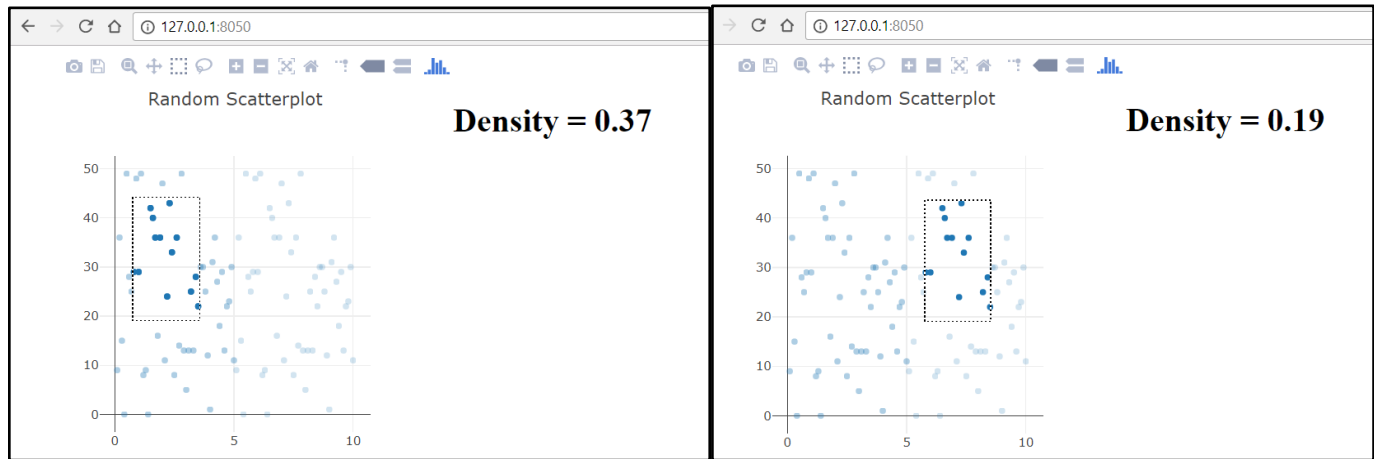
    html.Div([
        html.H1(id='density', style={'paddingTop':25})
    ], style={'width':'30%', 'display':'inline-block', 'verticalAlign':'top'})
])

@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()

```

Führe das Skript aus, öffne einen Browser unter <http://127.0.0.1:8050/> und verwende die Lasso- und Rechteckauswahlwerkzeuge in der Diagrammenüleiste, um Gruppen von Datenpunkten auf *beiden Seiten* des Diagramms auszuwählen. Du solltest so etwas angezeigt bekommen:



Alles, was wir hier gemacht haben, ähnelt dem JSON-Outputskript, mit Ausnahme der Ermittlung der Dichte. Da Selected Data je nach verwendetem Werkzeug entweder einen „Range“-Schlüssel oder einen „lassoPoint“-Schlüssel zurückgibt, mussten wir hinsichtlich der Selektionsgröße kreativ werden. Beachte bitte auch, dass Lassos immer überstehende Bereiche haben werden, da wir im Wesentlichen eine Box um die minimalen und maximalen Werte für "x" und "y" des Blobs (Datenklumpen) erstellen.

In diesem Beispiel werden die Punkte in der linken Hälfte der Grafik verdoppelt (wo immer du einen Punkt siehst, sind es tatsächlich zwei überlappende Punkte). Die rechte Hälfte der Grafik ist mit Einzelpunkten belegt. Daher ist die berechnete Dichte links, für eine ähnliche Auswahl von Punkten, doppelt so hoch wie rechts.

Wenn du wissen möchtest, wie der JSON-Output für dieses Diagramm aussieht, führe die enthaltene Datei **select2a.py** aus, die im Kursmaterial enthalten ist.

## Diagramme und Interaktionen updaten

Im Abschnitt zu [Interaktion mit Visualisierungen](#) haben wir bisher nur Hover, Klicken und Auswählen verwendet, um neue Daten anzuzeigen. Im nächsten Abschnitt wird gezeigt, wie diese Werkzeuge auf ein Diagramm angewendet werden und damit Änderungen an einem anderen Diagramm im selben Dashboard ausgelöst werden können.

Für diese Übung nehmen wir erneut das **mpg.csv**-Dataset, da es über eine praktische Anzahl von Datenpunkten verfügt, über die wir mit dem Curser gleiten können.

Um ein nützliches Streudiagramm einzurichten, möchten wir die Punkte entlang der x-Achse verteilen. Das Modelljahr ist eine gute Funktion, aber wir fügen den Daten ein künstliches "Jitter" (Zittern) hinzu, sodass die Punkte sich nicht alle entlang einer vertikalen Linien ausrichten.

Rechts von unserem Streudiagramm erstellen wir ein Liniendiagramm, das die Beschleunigung eines ausgewählten Fahrzeugs darstellt. Je steiler die Linie, desto schneller die Beschleunigung. Wir entfernen die X- und Y-Achsentexte - wir möchten, dass die Linie nur relative Vergleiche anzeigt.

Ein bisschen Mathe: erinnerst du dich daran, dass das Dataset eine Spalte für die Beschleunigung enthält, die die Zeit in Sekunden angibt, um von null auf sechzig Meilen pro Stunde zu gehen? Um dies in eine Steigung umzuwandeln, verwenden wir die folgende Formel:

$$\begin{aligned} \text{acceleration} &= \frac{\Delta v}{t} = \frac{\Delta \text{miles per minute}}{\# \text{ minutes}} = \frac{(60 \text{ miles per hour}) / (60 \text{ minutes/hour})}{(\# \text{ seconds}) / (60 \text{ seconds/minute})} \\ &= \frac{60}{\# \text{ seconds}} \end{aligned}$$

Je höher die Anzahl der Sekunden ist, desto langsamer ist die Beschleunigung und desto flacher ist die Steigung.

Erstelle eine Datei mit dem Namen **updating1.py** und füge den folgenden Code hinzu:

```
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(
```



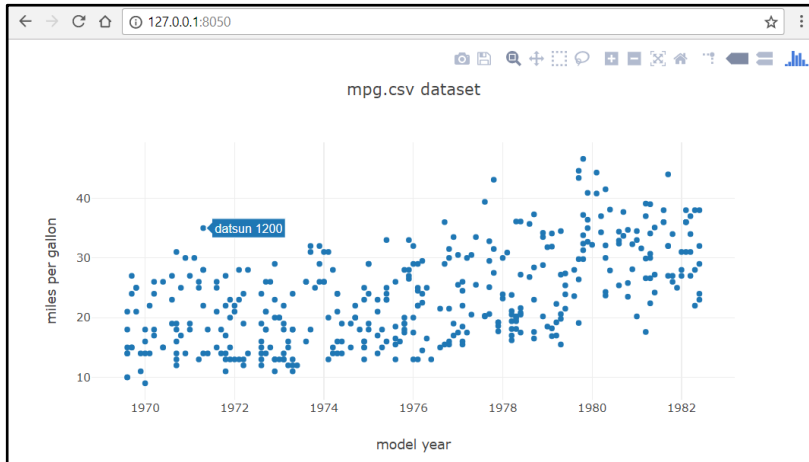
```

        title = 'mpg.csv dataset',
        xaxis = {'title': 'model year'},
        yaxis = {'title': 'miles per gallon'},
        hovermode='closest'
    )
}
)
))

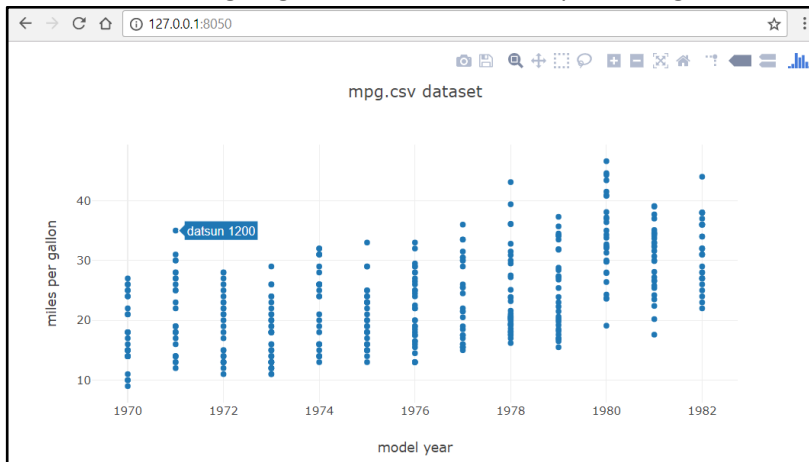
if __name__ == '__main__':
    app.run_server()

```

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Wir haben für unser „Jittern“ zufällige Werte verwendet, so dass dein etwas anders aussehen kann. Wenn wir das Zittern nicht hinzugefügt hätten, hätte der Graph so ausgesehen:



Als Nächstes fügen wir ein Liniendiagramm hinzu, das die Beschleunigung darstellt, und binden es mit `hoverData` an unser Streudiagramm.

Kopiere **updating1.py** und nenne die Datei **updating2.py**. Füge den folgenden Code hinzu (**fett** dargestellt):

```

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}
                )
            }
        ),
        # add a callback
        @app.callback(
            Output('mpg_line', 'figure'),
            [Input('mpg_scatter', 'hoverData')])
        def callback_graph(hoverData):

```



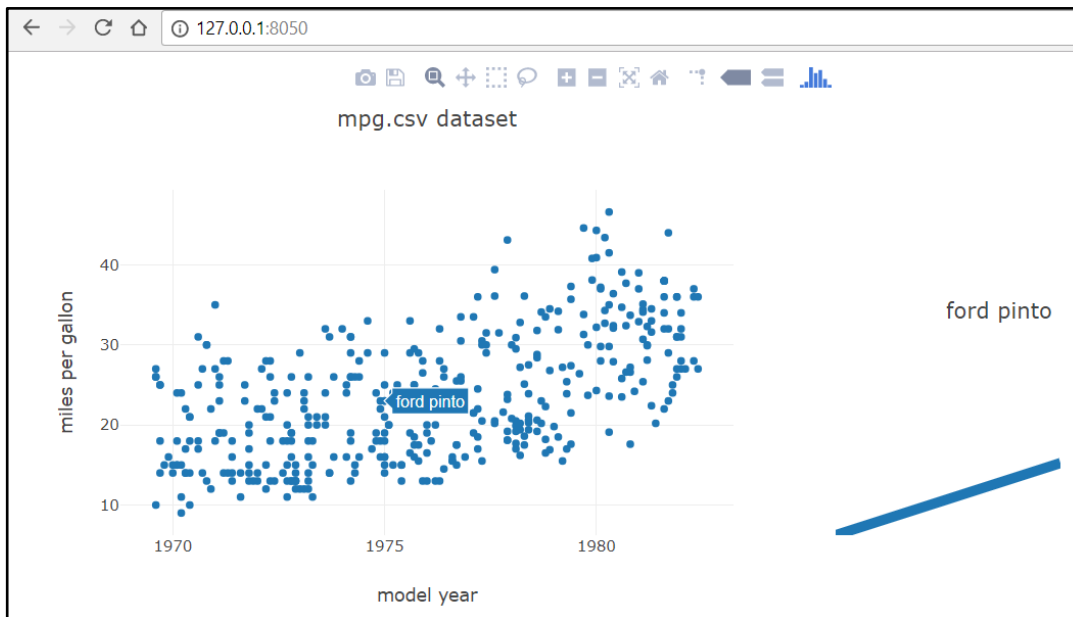
```

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()

```

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Wenn du nun mit der Maus über verschiedene Fahrzeuge fährst, ändert die Grafik rechts die Position (höher für schnellere Autos) und die Dicke abhängig von der Anzahl der Zylinder.

Fügen wir eine weitere Funktion hinzu und lassen die Fahrzeugstatistik als **dcc.Markdown**-Element anzeigen.

Kopiere **updating2.py** und nenne die Datei **updating3.py**. Füge den folgenden Code hinzu (**fett dargestellt**):

```

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}
                )
            }
        )], style={'width':'50%', 'display':'inline-block'}),
    # add a Markdown section
    dcc.Markdown(
        id='mpg_stats'
    ),
    ], style={'width':'20%', 'height':'50%', 'display':'inline-block'})
])

@app.callback(
    Output('mpg_line', 'figure'),

```





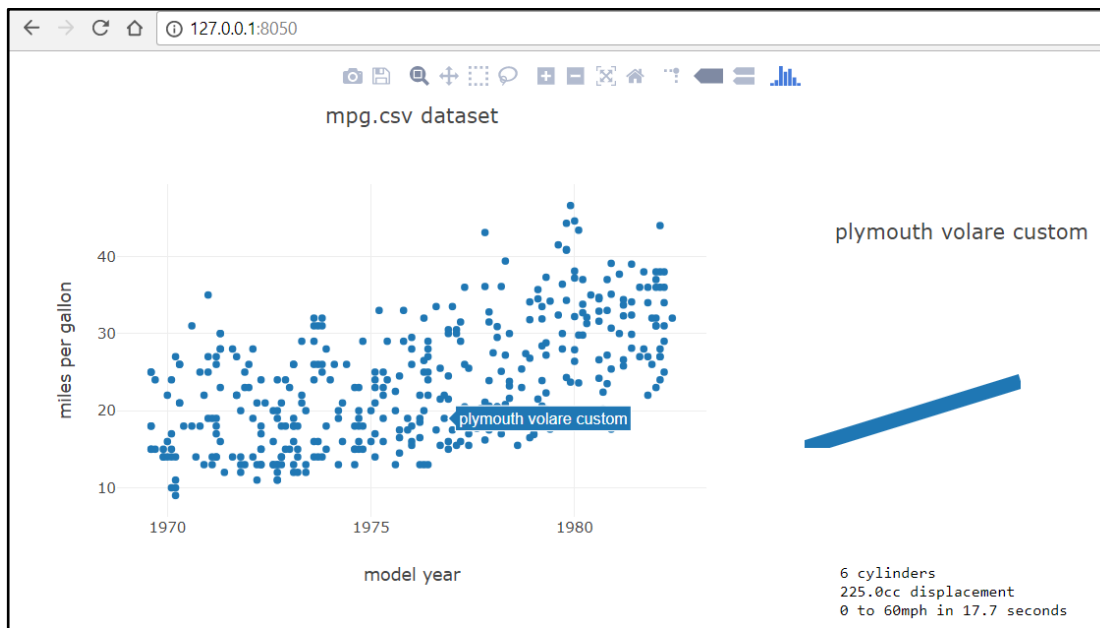
```

[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()

```

Nun kannst du das Skript ausführen und im Browser unter <http://127.0.0.1:8050/> folgendes sehen:



Das wars schon! Jetzt haben wir „Hover“ verwendet, um ein anderes Diagramm auf derselben Seite dynamisch zu ändern und gleichzeitig einen Markdown-Bereich eingepflegt.

## Kodieren von Meilenstein-Projekten

Wir bieten hier ein vollständiges Abschlussprojekt. Möglicherweise werden einige neue Inhalte behandelt. Daher empfehlen wir die Erforschung der Pandas, Plotly und der Dash-Dokumentationen.

In diesem Projekt wird ein Börsenticker-Dashboard entwickelt, mit dem der Benutzer entweder ein Aktienkürzel in ein Eingabefeld eingeben oder Elemente aus einer Dropdown-Liste auswählen kann. Mit `pandas_datareader` können die Börsendaten in einem Diagramm angezeigt werden. Das finale Projekt enthält einen `DatePicker`, um das Start- und Enddatum für das Diagramm festzulegen:



Weitere Informationen findest du in dem separaten Google Doc Meilensteinprojekt.

## Einführung zum Thema Live Updating

Bisher haben wir viele Möglichkeiten gezeigt, mit statischen Daten zu arbeiten. Der Einfachheit halber haben wir die .csv-Dateien selbst bereitgestellt, aber in den meisten Fällen hätten wir die Quellwebsites genauso gut in unsere Diagramme einbinden können.

Was aber, wenn sich die Informationen im Internet ständig ändern? In diesem Abschnitt können wir keine CSV-Datei bereitstellen, da unsere Quelldaten (<https://www.flightradar24.com>) alle 8 Sekunden aktualisiert werden!

In diesem Abschnitt wird die Komponente `dash_core_components.Interval` vorgestellt. Anstatt auf eine Benutzerinteraktion zu warten, um die Seite zu aktualisieren, kann man die Anwendung mit der Intervall-Komponente alle paar Sekunden oder Minuten aktualisieren.

Quelle: <https://dash.plot.ly/live-updates>

## Einfaches Beispiel zum Thema Live Updating

Bevor wir die dcc.Interval-Komponente aufrufen, betrachten wir ein Update beim Laden der Seite. Aus der Dash-Dokumentation:

„Standardmäßig speichern Dash-Apps das **app.layout**. Dadurch wird sichergestellt, dass das **Layout** beim Start der App nur einmal berechnet wird. Wenn du **app.layout** auf eine Funktion setzt, kannst du bei jedem Laden der Seite ein dynamisches Layout bereitstellen.“

Um dies zu demonstrieren, erstelle eine Datei **layoutupdate0.py** und füge folgenden Code hinzu:

```
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()
```

Nun kannst du das Skript ausführen um die Seite anzuzeigen und mehrmals aktualisieren. Du wirst feststellen, dass sich **nichts ändert**.

Kopiere **layoutupdate0.py** und nenne die Datei **layoutupdate1.py**. Füge den folgenden Code hinzu (**fett dargestellt**):

```
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()
```

Nun kannst du das Skript ausführen um die Seite anzuzeigen und mehrmals aktualisieren. Du wirst feststellen, dass sich das Layout nun **ändert**.

Jetzt ist es an der Zeit, die Seite in regelmäßigen Abständen automatisch zu aktualisieren.

Kopiere **layoutupdate1.py** und nenne die Datei **layoutupdate2.py**. Füge den folgenden Code hinzu (**fett dargestellt**):

```
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()

```

Hier verwenden wir einen Callback-Input (**dcc.Interval**), um in regelmäßigen Abständen eine Callback-Output (unser **html.H1**-Tag) auszulösen.

Führe nun das Skript aus. Das Layout sollte automatisch alle 2 Sekunden aktualisiert werden!

Denk daran, dass die IDs, die wir unseren Input- und Output-Elementen zuweisen, willkürlich sind (in diesem Fall **'live-update-text'** und **'intervall-component'**). Die verwendeten Property-Namen sind jedoch wichtig. Wir möchten die **'n\_intervals'**-Eigenschaft der **dcc.Interval**-Komponente eingeben, und in dieser Situation möchten wir eine **'children'**-Eigenschaft an unsere **html.H1**-Komponente zurückgeben (hier wird der String zum Headertext).

Im nächsten Beispiel werden wir eine Website heranziehen, die alle acht Sekunden aktualisiert wird. Die Website <https://www.flightradar24.com> empfängt Flugdaten aus der ganzen Welt und aktualisiert ihre Seite ständig, indem Echtzeitflugdaten auf Google Maps angezeigt werden.

Die Daten, die uns wichtig sind, werden nur die Gesamtzahl der aktiven Flüge weltweit sein. Dies wird in der oberen linken Ecke des Bildschirms angezeigt, direkt neben der Anzahl der Flüge in der aktuellen Ansicht. Es ist erwähnenswert, dass flightradar24-Daten aus einer Reihe von Quellen stammen, einschließlich Radarstationen (ADS-B, FLARM, MLAT, FAA) sowie Schätzungen.

Es wäre schön, wenn wir die Startseite aufrufen und diese Daten direkt so übernehmen könnten. Das Skript würde ungefähr so aussehen:

```

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')

```

Leider kommen die meisten auf der flightradar24 Seite angezeigten Daten von JavaScript-Aufrufen!

Glücklicherweise können wir das immer noch mit ein bisschen JSON-Analyse bewältigen. Wenn du wissen möchtest, woher die URL kommt, die wir verwenden möchten, überprüfe einfach das `#statTotal` -Element in den Entwicklertools, öffne das Netzwerk und sieh dir die verschiedenen JavaScript-Aufrufe an.

Erstelle eine Datei **liveupdating1.py** und füge folgenden Code hinzu:

```
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()
```

Hier haben wir die flightradar24-Website selbst in unsere eigene Seite eingebettet, gefolgt vom, durch Web-Scraping erhaltenen, Zählerwert! Wenn du die Seite aktualisierst, ändert sich der Zählerwert nicht. Sobald durch unser Skript gesetzt, bleibt dieser Wert erhalten, bis das Skript angehalten und neu gestartet wird.

Als Nächstes fügen wir eine **dcc.Interval**-Komponente hinzu.

Kopiere **liveupdating1.py** und nenne die Datei **liveupdating2.py**. Füge den folgenden Code hinzu (**fett** dargestellt):

```
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"

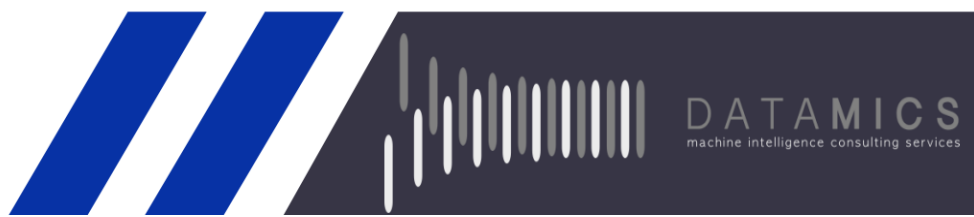
    # A fake header is necessary to access the site:
    res = requests.get(url, headers={'User-Agent': 'Mozilla/5.0'})
    data = res.json()
    counter = 0
    for element in data["stats"]["total"]:
        counter += data["stats"]["total"][element]
    return 'Active flights worldwide: {}'.format(counter)

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

Du wirst feststellen dass wir den Abschnitt URL-Aufforderung einfach in die Funktionsdefinition **update\_layout** verschoben haben.

Führe nun das Skript aus und du wirst feststellen, dass die gesamten Flüge alle sechs Sekunden aktualisiert werden. Es ist nicht perfekt mit flightradar24 synchronisiert, aber wir sind nahe dran.

Lass uns nun die eingehenden Daten darstellen.



Kopiere **liveupdating2.py** und nenne die Datei **liveupdating3.py**. Füge den folgenden Code hinzu (**fett dargestellt**):

```
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

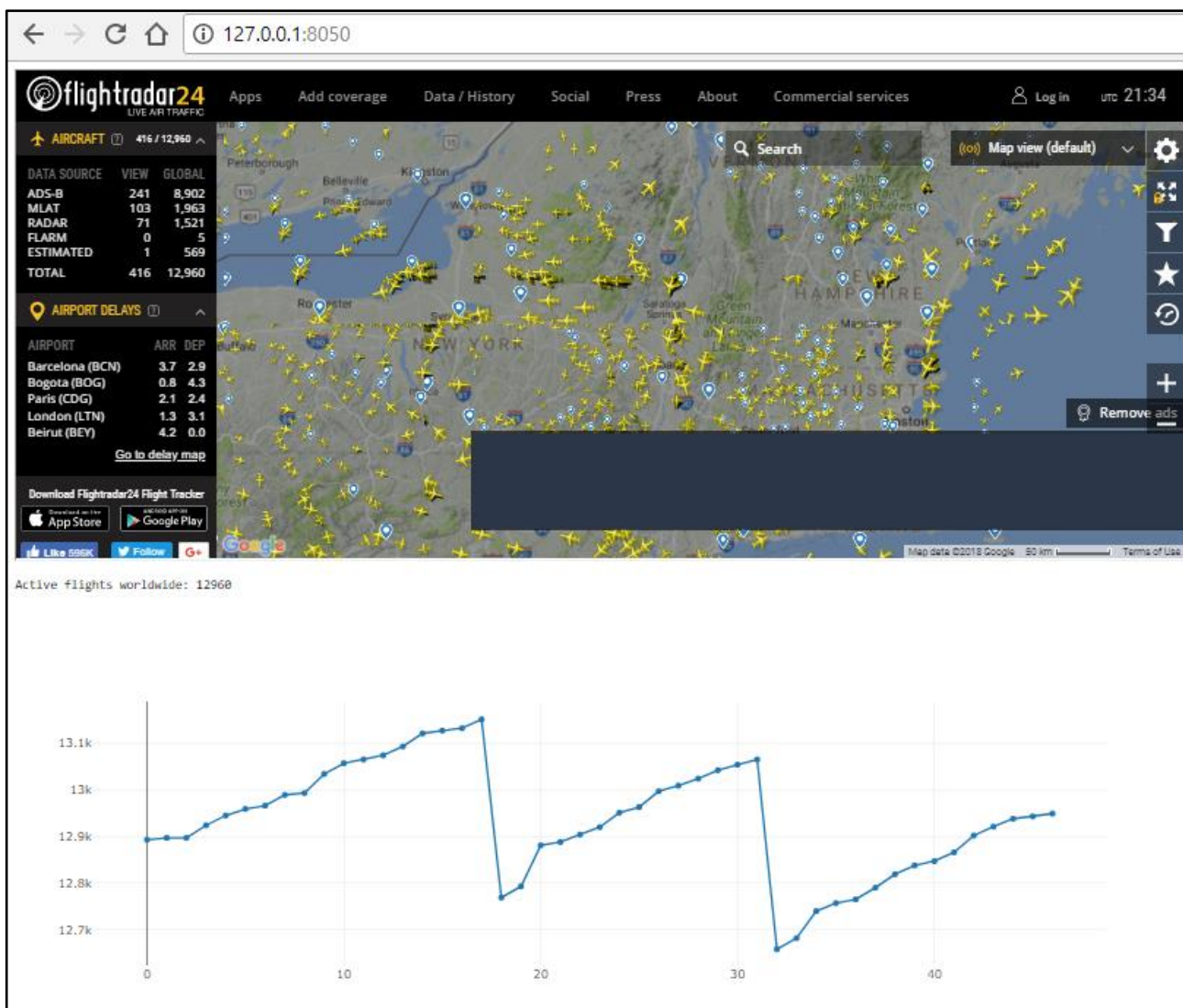
if __name__ == '__main__':
    app.run_server()

```

Führe das Skript nun aus! jetzt haben wir ein ständig aktualisiertes Liniendiagramm unterhalb der Website!

Du siehst, mit datetime haben wir nichts unternommen. Diese Grafik zeigt einfach die Daten, die wir seit dem Öffnen der Seite gespeichert haben, und zeigt uns damit den Trend der Anzahl der aktiven Flüge weltweit.

Nach einiger Zeit sieht deine Seite etwa so aus:



Super gemacht!

## Bereitstellung

### Einführung zur Bereitstellung von Apps

In diesem Abschnitt betrachten wir die letzte Phase der Dashboard-Entwicklung – die Bereitstellung. Wir zeigen, wie du deine App auf Heroku bereitstellen kannst und wie du deiner App eine Benutzerauthentifizierung hinzufügst, sodass der Inhalt nur eingeladenen Gäste/Usern angezeigt wird.

Bevor du deine App bereitstellst, solltest du die Benutzerauthentifizierung (Benutzername und Kennwort) hinzufügen.

### App Authorisierung

Aus der Dash Dokumentation:

Die Authentifizierung für Dash-Apps erfolgt über ein separates [dash-auth](#)-Paket. [dash-auth](#) bietet zwei Authentifizierungsmethoden: **HTTP Basic Auth** und **Plotly OAuth**.

**HTTP Basic Auth** ist eine der einfachsten Formen der Authentifizierung im Web. Als Entwickler von Dash kodieren Sie eine Reihe von Benutzernamen und Kennwörtern in Ihrem Code und senden diese Benutzernamen und Kennwörter an Ihre User. Es gibt einige Einschränkungen für HTTP Basic Auth:

- Benutzer können sich nicht von Anwendungen abmelden
- Sie sind dafür verantwortlich, Benutzernamen und Kennwörter über einen sicheren Kanal an Ihre User zu senden
- Ihre User können kein eigenes Konto erstellen und ihr Kennwort nicht ändern
- Sie sind dafür verantwortlich, die Paaren aus Benutzername und Kennwort sicher in Ihrem Code zu speichern.

**Plotly OAuth** bietet Authentifizierung über Ihr Online-Plotly-Konto oder über den [Plotly On-Premise server](#) ihres Unternehmens. Als Dash-Entwickler ist hierfür ein kostenpflichtiges Plotly-Abonnement erforderlich. Hier können Sie Plotly Cloud abonnieren ([subscribe to Plotly Cloud](#)). Hier können Sie uns bezüglich Plotly On-Premise kontaktieren. Die Anwender Ihrer App benötigen ein Plotly-Konto, müssen jedoch kein Upgrade auf ein kostenpflichtiges Abonnement durchführen.

Mit Plotly OAuth können Sie Ihre Apps mit anderen Benutzern teilen, die über Plotly-Konten verfügen. Mit Plotly On-Premise umfasst dies auch das Teilen von Apps über das integrierte LDAP-System. Apps, die Sie gespeichert haben, werden in der Liste der Dateien unter <https://plot.ly/organize> angezeigt und Sie können die Berechtigungen der Apps dort verwalten. Viewer erstellen und verwalten ihre eigenen Konten.

**HTTP Basic Auth** reicht für unsere Zwecke völlig aus. Um deine App Authentifizierung hinzuzufügen, stelle zunächst sicher, dass sowohl **dash** als auch **dash-auth** in deinem System installiert sind:

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

Wähle als Nächstes eine App aus dem Kurs aus, die du bereitstellen möchtest. Wir werden die Lösung für unsere [Übung zu interaktiven Komponenten](#) verwenden, da es ein recht kurzes Skript ist (es gibt das Produkt zweier Werte zurück, die von einem Range-Schieberegler übermittelt wurden).

Erstelle eine neue Datei mit dem Namen **auth1.py** und füge folgenden Code hinzu:

```
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()
```

Führe das Skript aus, um sicherzustellen, dass es funktioniert, und füge dann folgenden Code hinzu (in **fett** dargestellt):

```
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()

```

Das wars schon! Führe das Skript aus, öffne den Browser unter <http://127.0.0.1:8050/>, und schon solltest du aufgefordert werden, einen Benutzernamen und ein Kennwort einzugeben, bevor die App geladen wird. Wir hier sollten ein paar Dinge hervorheben:

- Der Benutzername unterscheidet zwischen Groß- und Kleinschreibung. `JamesBond` wird funktionieren, aber `jamesbond` nicht.
- In der Produktivumgebung solltest du `USERNAME_PASSWORD_PAIRS` in einer separaten Datei oder Datenbank speichern und nicht in deinem Quellcode, wie wir es hier gemacht haben.
- Der Feldname ist beliebig. Wir haben `USERNAME_PASSWORD_PAIRS` verwendet, aber du kannst es beliebig benennen, solange derselbe Name in `dash_auth.BasicAuth` verwendet wird.

## App in Heroku bereitstellen

Bisher hat jedes Dash-Skript `app.run_server()` zum Starten der App verwendet. Die App wird standardmäßig auf **localhost** ausgeführt und kann nur auf dem eigenen Computer angezeigt werden.

Die gute Nachricht ist, Dash verwendet Flask als Web-Framework. Wenn du also Flask bereitstellen kannst, kannst du auch Dash bereitstellen. Während es viele Optionen gibt, darunter Digital Ocean, PythonAnywhere, Google Cloud, Amazon Web Services, Azure usw., werden wir eine App-Bereitstellung auf Heroku durchgehen.

Weitere Informationen zum Bereitstellen von Flask-Apps findest du unter <http://flask.pocoo.org/docs/0.12/deploying/>

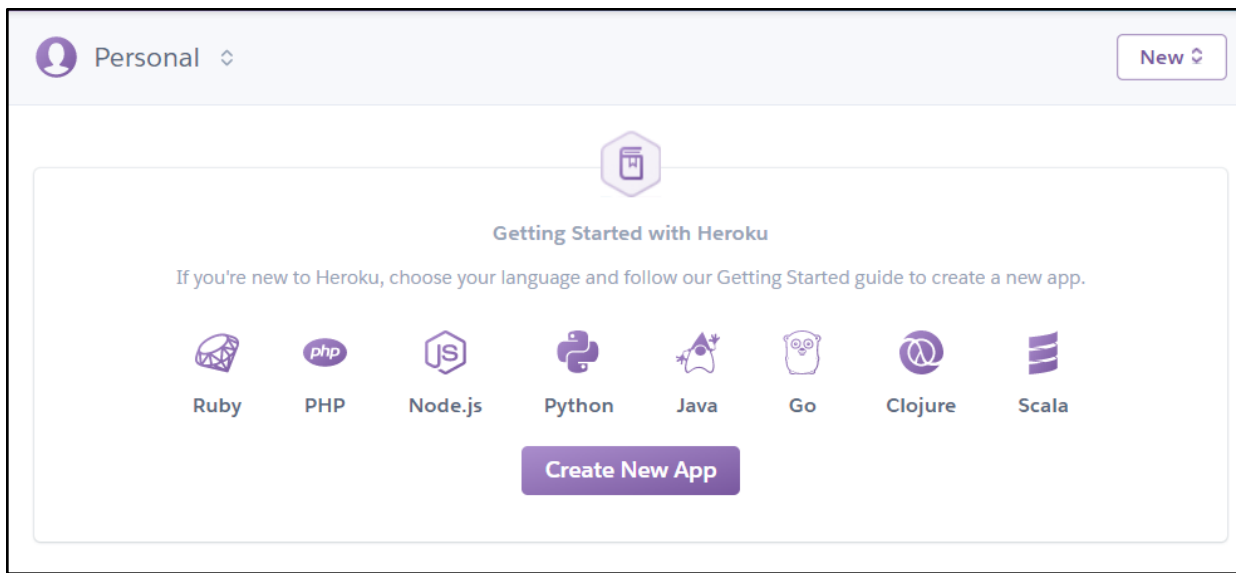
Weitere Informationen zu Heroku findest du unter <https://devcenter.heroku.com/articles/getting-started-with-python#introduction>

### STEP 1 – Installieren von Heroku und Git

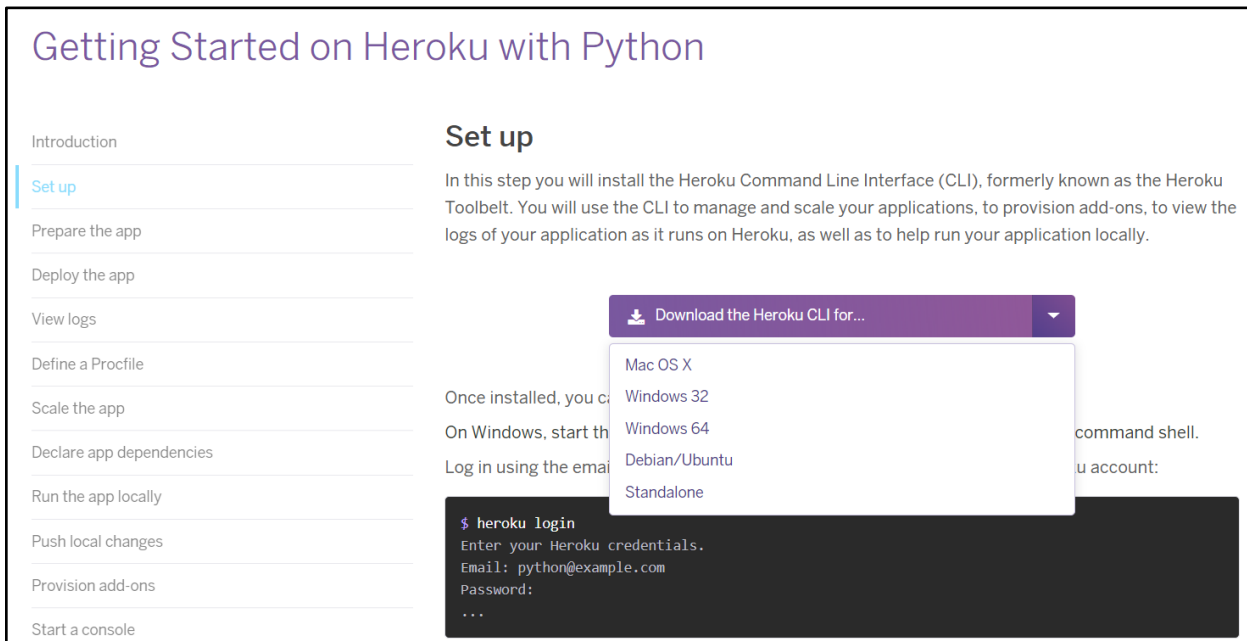
Heroku ist eine Cloud-Plattform, mit der Benutzer Apps im Web bereitstellen können.

Git ist ein System zur Versionskontrolle, mit dem du eine lokale Kopie deiner App für die Entwicklung halten und Änderungen von der Entwicklungskopie auf die bei Heroku gespeicherte bereitgestellte Version übertragen kannst.

1. Öffne einen **Heroku** Account. Free-Accounts sind verfügbar unter: <https://signup.heroku.com/dc>  
Folge den Anweisungen, um einen Benutzernamen und ein Passwort zu erhalten. Bitte schreib sie auf!
2. Melde dich bei deinem Heroku-Konto an. Es sollte dich hierhin führen <https://dashboard.heroku.com/apps>



3. Klicke nun auf **Python**. Wähle im nächsten Screen die Option **Set Up**. Es sollte eine Option angezeigt werden, um **Heroku Command Line Interface (CLI)** herunterzuladen. Wähle dein Betriebssystem aus der Dropdown-Liste aus und befolge dann die Anweisungen zur Installation des Dienstprogramms. Du solltest hier auch die Möglichkeit bekommen **Git** zu installieren.



4. Wenn **git** nicht mit Heroku CLI installiert wurde, kannst du es direkt von <https://git-scm.com/downloads> herunterladen und den Anweisungen für dein Betriebssystem folgen.

## STEP 2 – Installieren von virtualenv

5. Installiere nun bitte **virtualenv**, wenn du es noch nicht hast, indem du **pip install virtualenv** an deinem Rechner eingibst. Mit virtualenv kannst Sie virtuelle Umgebungen für deine App erstellen, die Python und alle Notwendigkeiten enthalten, die deine App benötigt. Dazu gehört eine bestimmte Version von Plotly, Dash und anderen Bibliotheken, von denen wir wissen, dass sie funktionieren. Sobald neue Updates verfügbar sind, wird deine App solange nicht angerührt, bis du sie mit den neuen Updates testen konntest!

## STEP 3 – Erstelle einen Entwicklungsordner (Development Folder)

6. Erstelle für dein Projekt einen neuen Ordner. Dieser wird die Entwicklungsumgebung deiner App enthalten:
- ```
C:\>mkdir my_dash_app  
C:\>cd my_dash_app
```

## STEP 4 – Initialisierung von Git

7. Initialisiere einen leeren git Ablageort
- ```
C:\my_dash_app>git init  
Initialized empty Git repository in C:/my_dash_app/.git/
```

## STEP 5 (WINDOWS) – Erstellen, Aktivieren und Bestücken von virtualenv

Siehe unten [macOS/Linux Anleitung](#)!

8. Erstelle eine virtuelle Umgebung. Wir nennen sie "venv", aber du kannst jeden beliebigen Namen verwenden:
- ```
C:\my_dash_app>python -m virtualenv venv
```
9. Aktiviere die virtuelle Umgebung:
- ```
C:\my_dash_app>.\venv\Scripts\activate
```
10. Installiere dash und alle gewünschten Abhängigkeiten in deiner virtuellen Umgebung:
- ```
(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 (Anforderung kann erfüllt sein, siehe unten)
```

Während wir das schreiben installiert **pip install dash**:

```
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. Installiere eine neue Abhängigkeit **gunicorn** zur Bereitstellung der App:
- ```
(venv) C:\my_dash_app>pip install gunicorn
```

## STEP 5 (macOS/Linux) - Erstellen, Aktivieren und Bestücken von virtualenv

8. Erstelle eine virtuelle Umgebung. Wir nennen sie "venv", aber du kannst jeden beliebigen Namen verwenden:

```
$ python3 -m python3 -m virtualenv venv
```

9. Aktiviere die virtuelle Umgebung:

```
$ source venv/bin/activate
```

10. Installiere dash und alle gewünschten Abhängigkeiten in deiner virtuellen Umgebung:

```
$ 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. Installiere eine neue Abhängigkeit **gunicorn** zur Bereitstellung der App:

```
$ pip install gunicorn
```

## STEP 6 – Dem Entwicklungsordner Dateien hinzufügen

Die folgenden Dateien müssen hinzugefügt werden:

- app1.py eine Dash-Anwendung
- .gitignore von git verwendet Dateien, die *nicht* in die Produktion verschoben werden
- Procfile für die Bereitstellung verwendet
- requirements.txt beschreibt deine Python-Abhängigkeiten und kann automatisch erstellt werden

app1.py

Kopiere die in "[App Autorisierung](#)" verwendete Datei (oder eine beliebige Datei, die du bereitstellen möchtest) und fügen den folgenden Code hinzu, der **fett** gedruckt ist:

```
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 bezieht sich auf den Dateinamen unserer Anwendung (app1.py) und server bezieht sich auf den variablen **Server** in dieser Datei.

### requirements.txt

Diese Datei kann automatisch generiert werden, in dem man `pip freeze > requirements.txt` anstößt. Stelle sicher, dass du dies aus dem Entwicklungsordner mit aktivierter virtueller Umgebung machst.

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

Das Ergebnisfile sieht ungefähr so aus:

```

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 7 - Log onto your Heroku Account

Melde dich am Rechner mit den in STEP1 festgelegten Zugangsdaten an:

```
(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 8 - Initialize Heroku, add files to Git, and Deploy

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

Du musst **my-dash-app** in einen eindeutigen Namen ändern. Der Name muss mit einem Buchstaben beginnen und darf nur Kleinbuchstaben, Zahlen und Bindestriche enthalten.

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

Beachte die Periode am Ende. Dies fügt alle Dateien zu git hinzu (außer die in .gitignore aufgelisteten).

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

Jedes **git commit** sollte einen kurzen beschreibenden Kommentar enthalten. Abhängig von deinem Betriebssystem erfordert dieser Kommentar möglicherweise doppelte Anführungszeichen (keine einfachen Anführungszeichen).

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

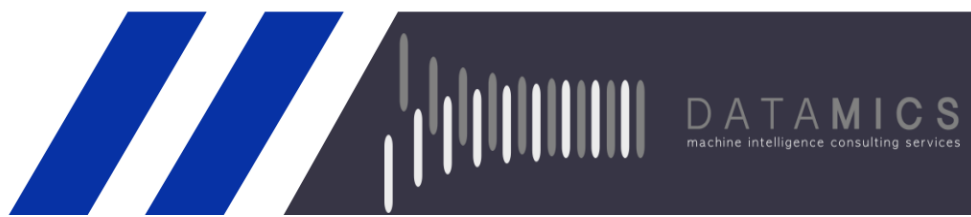
Dadurch wird dein aktueller Code für Heroku bereitgestellt. Das erste Mal, wenn du **push** ausführst, kann eine Weile dauern, da Python und alle Abhängigkeiten auf dem Remote-Server eingerichtet werden müssen.

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

Dadurch wird die App mit 1 "Heroku" Dyno ausgeführt.

## STEP 9 - Visit Your App on the Web!

Du solltest deine Seite hier sehen <https://my-dash-app.herokuapp.com>  
(ändere **my-dash-app** in den Namen deiner App)



## STEP 10 - Update Your App

Jedes Mal, wenn du Änderungen an deiner App vornimmst, deinem Repo neue Apps hinzufügst oder neue Bibliotheken installierst und / oder vorhandene Abhängigkeiten in deiner virtuellen Umgebung aktualisierst, möchtest du natürlich die neuesten Updates in Heroku veröffentlichen. Dies sind die grundlegenden Schritte:

Beim Installieren eines neuen Paketes:

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

Beim Updaten eines vorhandenen Paketes:

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

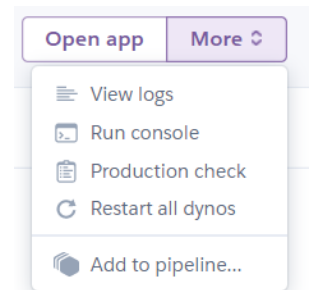
In allen Fällen:

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

## FEHLERSUCHE

Wenn du deine App auf Heroku nicht veröffentlichen kannst, folge dieser Checkliste:

- ☐ **app1.py** beinhaltet **server = app:server**  
Wenn nicht, füge diese Zeile hinzu, speichere die Datei, und führe dann Git add / commit / push aus
- ☐ **gunicorn** installiert, und in **requirements.txt** eingebettet  
Wenn nicht, führe **pip install gunicorn** aus, dann **pip freeze > requirements.txt**, und führe dann Git add / commit / push aus
- ☐ Wenn du es lokal nicht nachvollziehen kannst, besuche dein Heroku-Dashboard und klicken Sie auf **More / View logs** (Protokolle) anzeigen



Quelle: <https://dash.plot.ly/deployment>

## APPENDIX I - BEISPIELCODES:

### Plotly Grundlagen

#### Plotly Grundlagen Überblick

##### 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([{\br/>    'x': df.index,  
    'y': df[col],  
    'name': col  
} for col in df.columns])
```

## Streudiagramme

### 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')
```

## Liniendiagramme

### 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')
```



## Balkendiagramme

### 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')
```

## Blasendiagramme

### 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')
```

## Kastendiagramme (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')
```



## Histogramme

### 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>\n1970'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>\n1970'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>\n1970'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')
```

## Verteilungsdiagramme

### 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>\n    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>\nYuma, 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>\nSitka, 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')
```

## Lösungen: Plotly Grundlagen

[>> Hier geht's zu den Aufgaben <<](#)

### 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')
```

## Eine Anmerkung zur Liniendiagramm Übung:

Eine Schleife funktioniert nicht wie erwartet! Der 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)
```

hat jede *Zeile* als eigene Spur, nicht jeden Tag.

**Sol2a-Linechart.py** verwendet hart codierte Werte:

```
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)
```

...aber das ist keine ideale Lösung!

**Sol2b-Linechart.py** lässt Pandas den df['DAY'] filter:

```
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)
```

Das funktioniert!



## Sol2a-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.  
#####  
# 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

```
#####
# 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 KOMPONENTEN

<https://dash.plot.ly/dash-core-components>

Ein kurzer Überblick über die verfügbaren Core Komponenten:

### 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'
)
```

Bei einer “Single dropdown” Liste, setzt **value** Initialwert fest

```
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** erlaubt Mehrfachselektionen

### Slider (Schieberegler)

```
import dash_core_components as dcc

dcc.Slider(
    min=-5,
    max=10,
    step=0.5,
    value=-3,
)
```

Basic Schieberegler

```
import dash_core_components as dcc

dcc.Slider(
```



```
min=0,
max=9,
marks={i: 'Label {}'.format(i) for i in range(10)},
value=5,
)
```

Hier kann man Labels setzen (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=''
)
```

## Textfeld

```
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']
)
```

### Vertikale Liste

```
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'}
)
```

### Horizontale

## Radio Buttons (Auswahlfelder)

```
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'
)
```

### Vertikale Liste

```
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'
)
```



```
        value='MTL',
        labelStyle={'display': 'inline-block'}
    )
```

Horizontale

## 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)
```

Für mehr zu [dash.dependencies.State](#), gibt es das Tutorial auf [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.
''')
```

## Graphen

Die Komponente **Graph** hat den selben Syntax wie die Open-Source Bibliothek **plotly.py**. Sieh dir die Dokumente [plotly.py docs](https://plotly.com/python/) um mehr zu erfahren.

## Noch in Entwicklung


### Interaktive Tabellen

Die **dash\_html\_components** Bibliothek enthält alle HTML tags, was auch **Table**, **Tr**, und **Tbody** Tags beinhaltet, die verwendet werden können, um eine HTML Tabelle zu erstellen. Siehe „Erstelle dein erstes Dashboard“ Teil 1 als Beispiel.

Dash führt zurzeit eine interaktive Tabelle aus, die integrierte Filterung, Zeilenauswahl, Bearbeitung und Sortierung bietet. Prototypen dieser Komponente werden im Repository für [dash-table-experiments](#) entwickelt. Du kannst gerne an der Diskussion im [Dash Community Forum](#) teilnehmen.

## Upload Komponente





Mit der Komponente **dcc.Upload** können Benutzer Dateien per Drag & Drop oder über den systemeigenen Datei-Explorer des Systems in ihre App hochladen.

## 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](#).

Die Komponente **dcc.Tabs** ist derzeit im Prerelease-Kanal des Pakets **dash-core-components** verfügbar. Um es auszuprobieren, schau dir die Tab-Komponente [Pull Request on GitHub](#) an.

## APPENDIX III – ZUSÄTZLICHE QUELLEN (ENGLISCH)

### [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](#)

#### Weiterführende 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**  
Detaillierter in den Dokumenten [Dash State](#) beschrieben
- **Graphen**  
Detaillierter in den Dokumenten [Plotly Python](#) beschrieben



