Author: Aditya Porwal

Pandas

As described at https://pandas.pydata.org

> pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Resources

1. Ch 5-6 in Python for Data Analysis, 2nd Ed, Wes McKinney (UCalgary library and https://github.com/wesm/pydata-book)
2. Ch 3 in Python Data Science Handbook, Jake VanderPlas (Ucalgary library and https://github.com/jakevdp/PythonDataScienceHandbook)

Let's explore some of the features.

First, import Pandas, and Numpy as a good companion.

In [ ]:
```
import numpy as np
import pandas as pd
```

Create pandas DataFrames

There are several ways to create Pandas DataFrames, most notably from reading a csv (comma separated values file). DataFrames are 'spreadsheets' in Python. We will often use `df` as a variable name for a DataFrame.

If data is not stored in a file, a DataFrame can be created from a dictionary of lists

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
            'year': [2000, 2001, 2002, 2001, 2002, 2003],
            'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```
where dictionary keys become column headers.

An alternative is to create from a numpy array and set column headers seperatly:

In [ ]:
```
# From a numpy array
df = pd.DataFrame( np.arange(20).reshape(5,4), columns=['alpha', 'beta', 'gamma', 'delta'])
df
```

Out[ ]:

	alpha	beta	gamma	delta
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15
4	16	17	18	19

In [ ]:

```
# checking its type
type(df)
```

Out[ ]:

```
pandas.core.frame.DataFrame
```

Indexing

Accessing data in Dataframes is done by rows and columns, either index or label based.

In [ ]:

```
# select a column
df['alpha']
```

Out[ ]:

```
0     0
1     4
2     8
3    12
4    16
Name: alpha, dtype: int32
```

In [ ]:

```
# select two columns
df[['alpha', 'gamma']]
```

Out[ ]:

	alpha	gamma
0	0	2
1	4	6
2	8	10
3	12	14
4	16	18

In [ ]:

```
# select rows
df.iloc[:2]
```

Out[ ]:

	alpha	beta	gamma	delta
0	0	1	2	3
1	4	5	6	7

In [ ]:

```
# select rows and columns
df.iloc[:2, :2]
```

Out[ ]:

	alpha	beta
0	0	1
1	4	5

In [ ]:

```
# select rows and columns, mixed
df.loc[:2, ['alpha', 'beta']]
```

Out[ ]:

	alpha	beta
0	0	1
1	4	5
2	8	9

DataFrame math

Similar to Numpy, DataFrames support direct math

In [ ]:

```
# direct math
df2 = (9/5) * df + 32
df2
```

Out[ ]:

	alpha	beta	gamma	delta
0	32.0	33.8	35.6	37.4
1	39.2	41.0	42.8	44.6
2	46.4	48.2	50.0	51.8
3	53.6	55.4	57.2	59.0
4	60.8	62.6	64.4	66.2

In [ ]:

```
# add two dataframes of same shape
df + df2
```

Out[ ]:

	alpha	beta	gamma	delta
0	32.0	34.8	37.6	40.4
1	43.2	46.0	48.8	51.6
2	54.4	57.2	60.0	62.8
3	65.6	68.4	71.2	74.0
4	76.8	79.6	82.4	85.2

In [ ]:

```
# map a function to each column
f = lambda x: x.max() - x.min()

df.apply(f)
```

Out[ ]:
```
alpha    16
beta     16
gamma    16
delta    16
dtype: int32
```

DataFrame manipulation

Adding and deleting columns, as well as changing entries is similar to Python dictionaries.

Note that most DataFrame methods do not change the DataFrame directly, but return a new DataFrame. It is always good to check how the method you are invoking behaves.

In [ ]:
```
# add a column
df['epsilon'] = ['low', 'medium', 'low', 'high', 'high']
df
```

Out[ ]:

	alpha	beta	gamma	delta	epsilon
0	0	1	2	3	low
1	4	5	6	7	medium
2	8	9	10	11	low
3	12	13	14	15	high
4	16	17	18	19	high

In [ ]:
```
# What is the size?
df.shape
```

Out[ ]:
```
(5, 5)
```

In [ ]:
```
# delete column
df_dropped = df.drop(columns=['gamma'])
df_dropped
```

Out[ ]:

	alpha	beta	delta	epsilon
0	0	1	3	low
1	4	5	7	medium
2	8	9	11	low
3	12	13	15	high
4	16	17	19	high

In [ ]:
```
# the original dataframe is unaffected
df
```

Out[ ]:

	alpha	beta	gamma	delta	epsilon
0	0	1	2	3	low
1	4	5	6	7	medium
2	8	9	10	11	low
3	12	13	14	15	high
4	16	17	18	19	high

Let's create a copy and assign new values to the first column:

In [ ]:

```
df_copy = df.copy()
df_copy['alpha'] = 20
print(df)
print(df_copy)
```

```
   alpha  beta  gamma  delta epsilon
0      0     1      2      3     low
1      4     5      6      7  medium
2      8     9     10     11     low
3     12    13     14     15    high
4     16    17     18     19    high
   alpha  beta  gamma  delta epsilon
0     20     1      2      3     low
1     20     5      6      7  medium
2     20     9     10     11     low
3     20    13     14     15    high
4     20    17     18     19    high
```

DataFrames can be sorted by column:

In [ ]:

```
# sorting values
df.sort_values(by='epsilon')
```

Out[ ]:

	alpha	beta	gamma	delta	epsilon
3	12	13	14	15	high
4	16	17	18	19	high
0	0	1	2	3	low
2	8	9	10	11	low
1	4	5	6	7	medium

Load data from file

Most often data will come from somewhere, often csv files, and using `pd.read_csv()` will allow smooth creation of DataFrames.

Loading auto-mpg.data:

In [ ]:

```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'mode
data = pd.read_csv('C:/Users/imaro/OneDrive/Desktop/ENSF 611/Lab 0/auto-mpg.data', names=column_n
data.head()
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

After loading data, it is good practice to check what we have. Usually, the sequences is:

1. Check dimension
2. Peek at the first rows
3. Get info on data types and missing values
4. Summarize columns

In [ ]:

```
# Check dimension (rows, columns)
data.shape
```

Out[ ]:

```
(398, 9)
```

In [ ]:

```
# Peek at the last rows
data.tail()
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
393	27.0	4	140.0	86.00	2790.0	15.6	82	1	ford mustang gl
394	44.0	4	97.0	52.00	2130.0	24.6	82	2	vw pickup
395	32.0	4	135.0	84.00	2295.0	11.6	82	1	dodge rampage
396	28.0	4	120.0	79.00	2625.0	18.6	82	1	ford ranger
397	31.0	4	119.0	82.00	2720.0	19.4	82	1	chevy s-10

In [ ]:

```
# Column names are
data.columns
```

Out[ ]:

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'model year', 'origin', 'car name'],
      dtype='object')
```

In [ ]:

```
# Get info on data types and missing values
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    398 non-null    object 
 4   weight        398 non-null    float64
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64  
 7   origin        398 non-null    int64  
 8   car name      398 non-null    object 
dtypes: float64(4), int64(3), object(2)
memory usage: 28.1+ KB
```

Summarize values

What is the mean, std, min, max in each column?

In [ ]:
```
data.mean(numeric_only=True)
```

Out[ ]:
```
mpg               23.514573
cylinders          5.454774
displacement     193.425879
weight          2970.424623
acceleration      15.568090
model year        76.010050
origin             1.572864
dtype: float64
```

In [ ]:
```
# where are the other columns? Check data types
data.dtypes
```

Out[ ]:
```
mpg             float64
cylinders         int64
displacement    float64
horsepower       object
weight          float64
acceleration    float64
model year        int64
origin            int64
car name         object
dtype: object
```

Notice that many columns are of type object, which is not a number. Maybe this has to do with missing values? We know from peeking at the first rows that there are '?' values in there. Let's replace these with the string NaN for not-a-number.

In [ ]:
```
# replace '?' with 'NaN'
data = data.replace({'?': 'NaN'})
data.head()
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino

Pandas knows that 'NaN' probably means that numbers are missing. Now we can convert the data type from object to float

In [ ]:

```
# convert dtypes
data["horsepower"] = data["horsepower"].astype('float')
data.dtypes
```

Out[ ]:

```
mpg             float64
cylinders         int64
displacement    float64
horsepower      float64
weight          float64
acceleration    float64
model year        int64
origin            int64
car name         object
dtype: object
```

We could have loaded the data with the `na_values` argument to indicate that '?' means missing number:

In [ ]:

```
data = pd.read_csv('C:/Users/imaro/OneDrive/Desktop/ENSF 611/Lab 0/auto-mpg.data', names=column_
```

This worked nicely. Now we can describe all columns, meaning printing basic statistics. Note that by default Pandas ignores NaN, whereas Numpy does not.

In [ ]:

```
data.describe() # ignores NaN
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin
count	398.000000	398.000000	398.000000	392.000000	398.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.469388	2970.424623	15.568090	76.010050	1.572864
std	7.815984	1.701004	104.269838	38.491160	846.841774	2.757689	3.697627	0.802055
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000	1.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000	1.000000
50%	23.000000	4.000000	148.500000	93.500000	2803.500000	15.500000	76.000000	1.000000
75%	29.000000	8.000000	262.000000	126.000000	3608.000000	17.175000	79.000000	2.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000	3.000000

We could be interested by these statistics in every origin. To get these, we first group values by origin, then ask for the description.

In [ ]:
```
data.groupby(by='origin').describe().weight
```

Out[ ]:

origin	count	mean	std	min	25%	50%	75%	max
1	249.0	3361.931727	794.792506	1800.0	2720.00	3365.0	4054.00	5140.0
2	70.0	2423.300000	490.043191	1825.0	2067.25	2240.0	2769.75	3820.0
3	79.0	2221.227848	320.497248	1613.0	1985.00	2155.0	2412.50	2930.0

Find NaNs

How many NaNs in each column?

We can ask which entries are null, which produces a boolean array

In [ ]:
```
data.isnull()
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...
393	False	False	False	False	False	False	False	False	False
394	False	False	False	False	False	False	False	False	False
395	False	False	False	False	False	False	False	False	False
396	False	False	False	False	False	False	False	False	False
397	False	False	False	False	False	False	False	False	False

398 rows × 9 columns

Applying `sum()` to this boolean array will count the number of `True` values in each column

In [ ]:
```
data.isnull().sum()
```

Out[ ]:
```
mpg             0
cylinders       0
displacement    0
horsepower      6
weight          0
acceleration    0
model year      0
origin          0
car name        0
dtype: int64
```

We get complementary information from `info()`

In [ ]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    392 non-null    float64
 4   weight        398 non-null    float64
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64  
 7   origin        398 non-null    int64  
 8   car name      398 non-null    object 
dtypes: float64(5), int64(3), object(1)
memory usage: 28.1+ KB
```

We can fill (replace) these missing values, for example with the minimum value in each column

In [ ]:
```
data = data.fillna(data.mean())
data.describe()
```

Out[ ]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin
count	398.000000	398.000000	398.000000	398.000000	398.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	103.587940	2970.424623	15.568090	76.010050	1.572864
std	7.815984	1.701004	104.269838	38.859575	846.841774	2.757689	3.697627	0.802055
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000	1.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000	1.000000
50%	23.000000	4.000000	148.500000	92.000000	2803.500000	15.500000	76.000000	1.000000
75%	29.000000	8.000000	262.000000	125.000000	3608.000000	17.175000	79.000000	2.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000	3.000000

Count unique values (a histogram)

We finish off, with our good friend the histogram

In [ ]:
```
data['origin'].value_counts()
```

Out[ ]:
```
1    249
3     79
2     70
Name: origin, dtype: int64
```

Visualization

Author: Aditya Porwal

Topics

1. Matplotlib core framework
2. Pandas plot()
3. Seaborn statistical visualization
4. (not covered) Grammar of graphics (ggplot2 see plotnine)
5. (not covered) Interactive plotting

Resources

1. Ch 9 in Python for Data Analysis, 2nd Ed, Wes McKinney (UCalgary library and https://github.com/wesm/pydata-book)
2. Ch 4 in Python Data Science Handbook, Jake VanderPlas (Ucalgary library and https://github.com/jakevdp/PythonDataScienceHandbook)
3. Fundamentals of Data Visualization, Claus O. Wilke (Ucalgary library and https://serialmentor.com/dataviz/index.html)
4. Overview by Jake VanderPlas https://www.youtube.com/watch?v=FytuB8nFHPQ

Matplotlib

> Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Matplotlib tries to make easy things easy and hard things possible.
For simple plotting the pyplot module provides a MATLAB-like interface

https://matplotlib.org

Importing matplotlib looks like this

In [ ]:
```
%matplotlib inline

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

Two interfaces

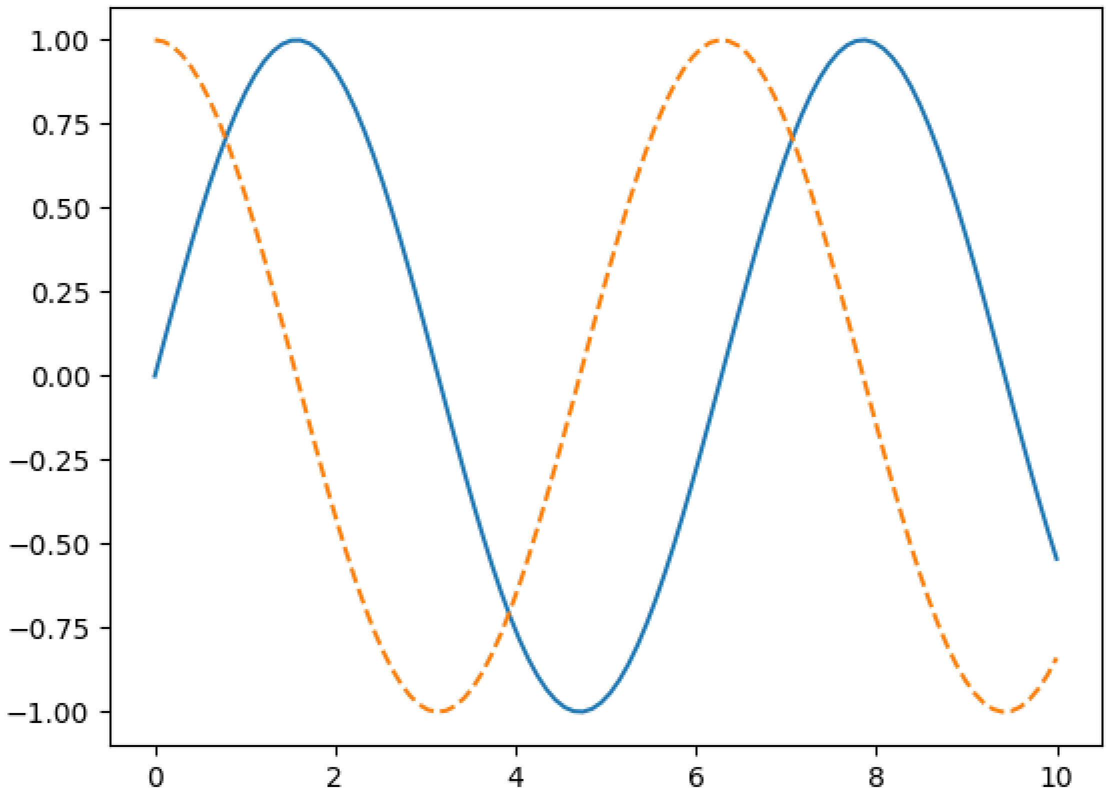
There are two ways to interact with Matplot lib: a Matlab style and an object oriented style interface.

See Ch 4 in Python Data Science Handbook, Jake VanderPlas

- Two Interfaces for the Price of One, pp. 222
- Matplotlib Gotchas, pp. 232

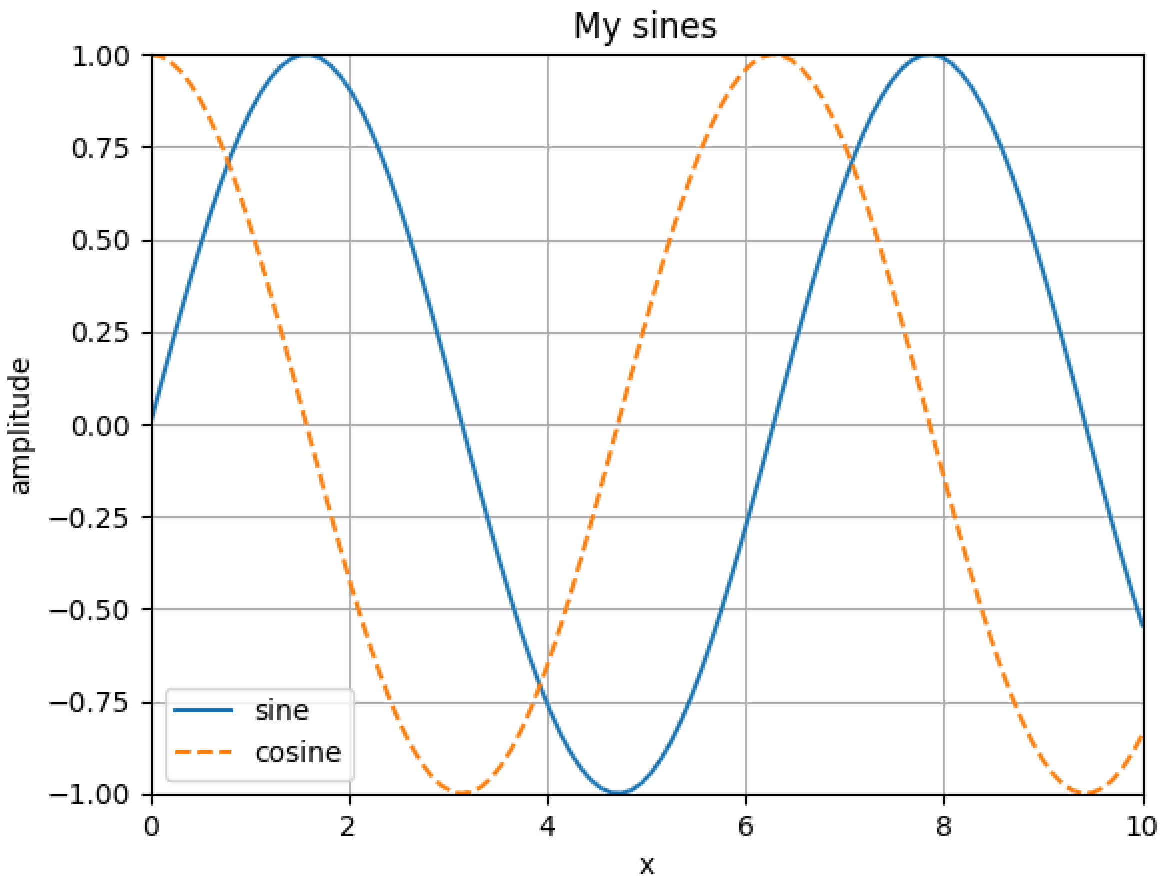
Matlab style interface

In [ ]:
```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--');
```



Adding decorations to the plot is done by repeatatly calling functions on the imported `plt` module. All calls within the cell will be applied to the current figure and axes.

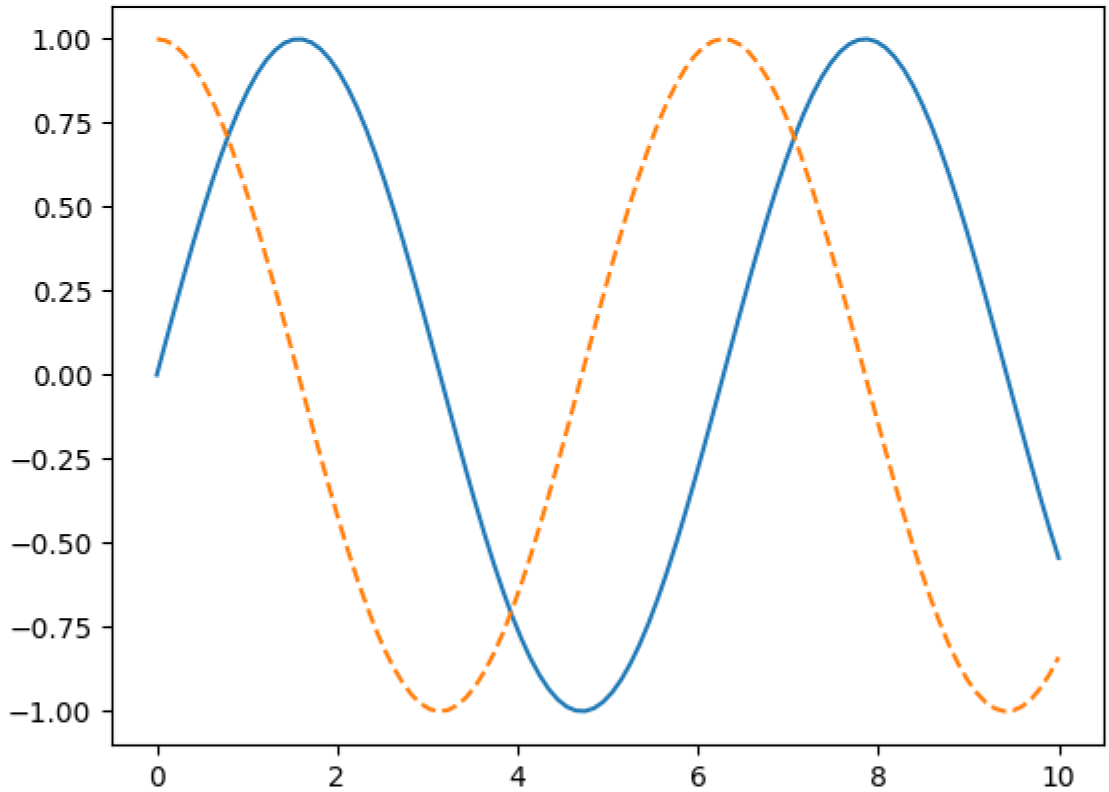
In [ ]:
```
plt.plot(x, np.sin(x), '-', label='sine')
plt.plot(x, np.cos(x), '--', label ='cosine')

plt.xlim([0, 10])
plt.ylim([-1, 1])
plt.xlabel('x')
plt.ylabel('amplitude')
plt.title('My sines')
plt.grid()
plt.legend();
```



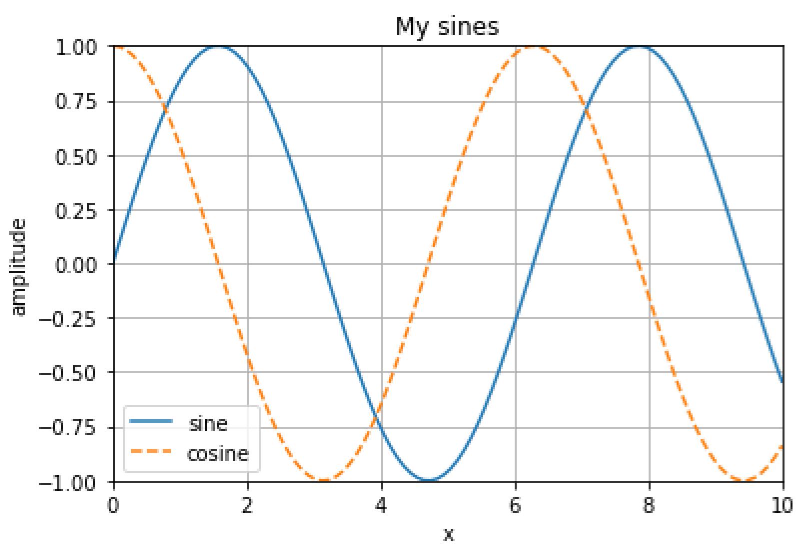
Object oriented interface

With this interface, you first create a figure and an axes object, then call their methods to change the plot.

In [ ]:
```
fig = plt.figure()
ax = plt.axes()
ax.plot(x, np.sin(x), '-')
ax.plot(x, np.cos(x), '--');
```




```
In [ ]: fig = plt.figure()
        ax = plt.axes()
        ax.plot(x, np.sin(x), '-', label='sine')
        ax.plot(x, np.cos(x), '--', label ='cosine')

        ax.set(xlim=[0, 10], ylim=[-1, 1],
               xlabel='x', ylabel='amplitude',
               title='My sines');
        ax.grid()
        ax.legend();
```



Save to file

With the figure object at hand, we can save to file

```
In [ ]: fig.savefig('sines.pdf')
        #!ls *.pdf
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

Plotting with pandas

> We use the standard convention for referencing the matplotlib API ... We provide the basics in pandas to easily create decent looking plots.

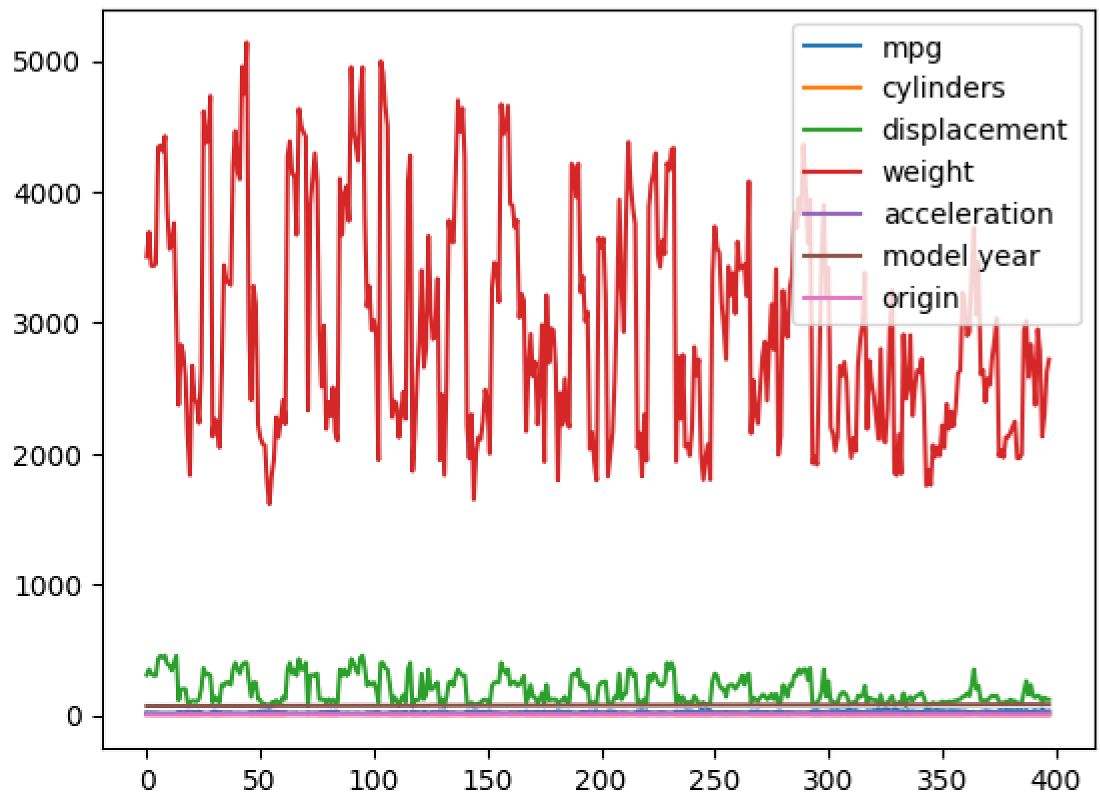
https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

Loading the auto-mpg dataset.

In [ ]:
```
column_names = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'mode
data = pd.read_csv('C:/Users/imaro/OneDrive/Desktop/ENSF 611/Lab 0/auto-mpg.data', names=column_n
```

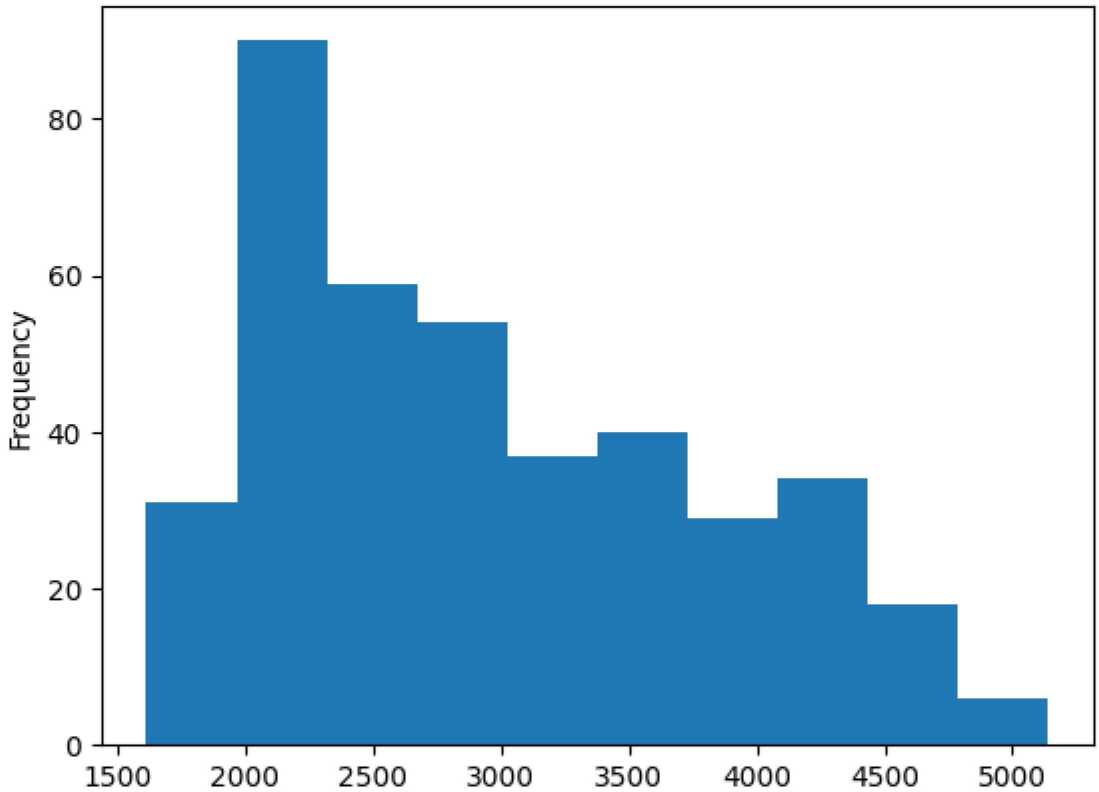
Plotting all columns, works, but does not provide a lot of insight.

In [ ]:
```
data.plot();
```



Let's look at the weight distribution (a histogram)

In [ ]:
```
data['weight'].plot.hist();
```



How many origins do we have?

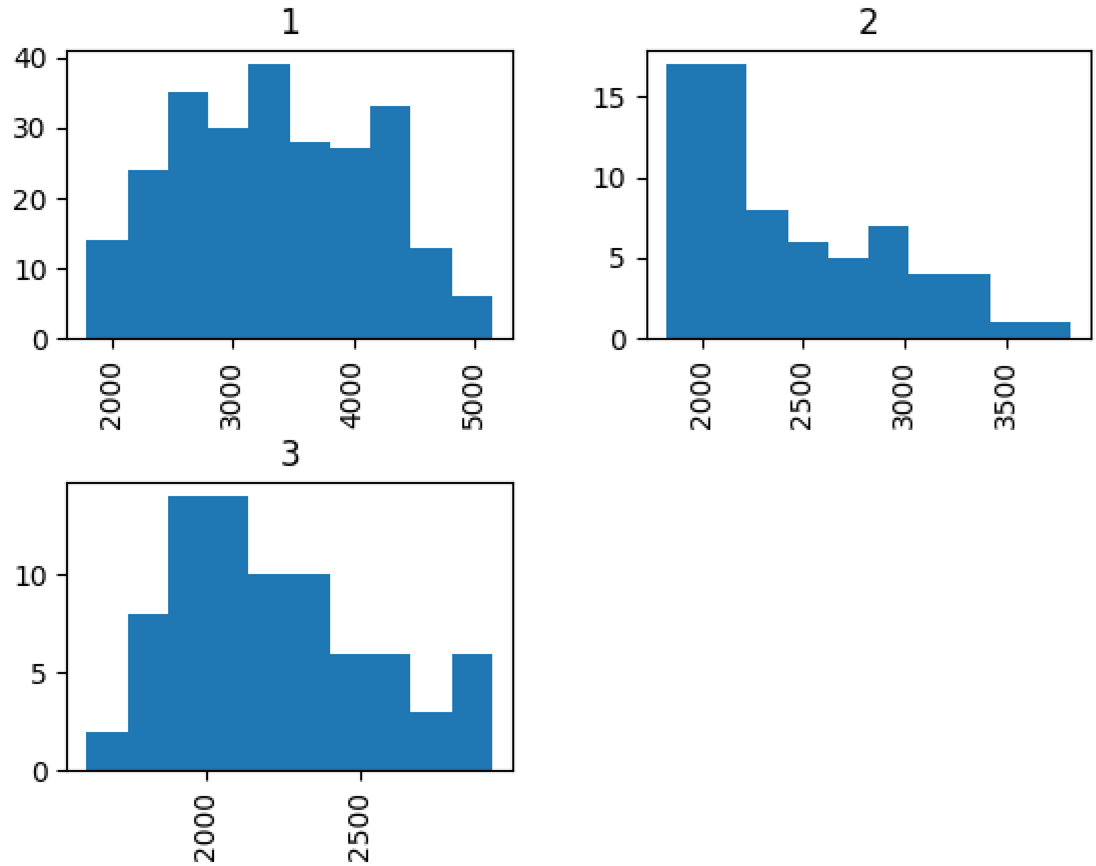
In [ ]:
```
data.origin.value_counts()
```

Out[ ]:
```
1    249
3     79
2     70
Name: origin, dtype: int64
```

Notice that we accessed the origin column with dot notation. This can be done whenever the column name is 'nice' enough to be a python variable name.

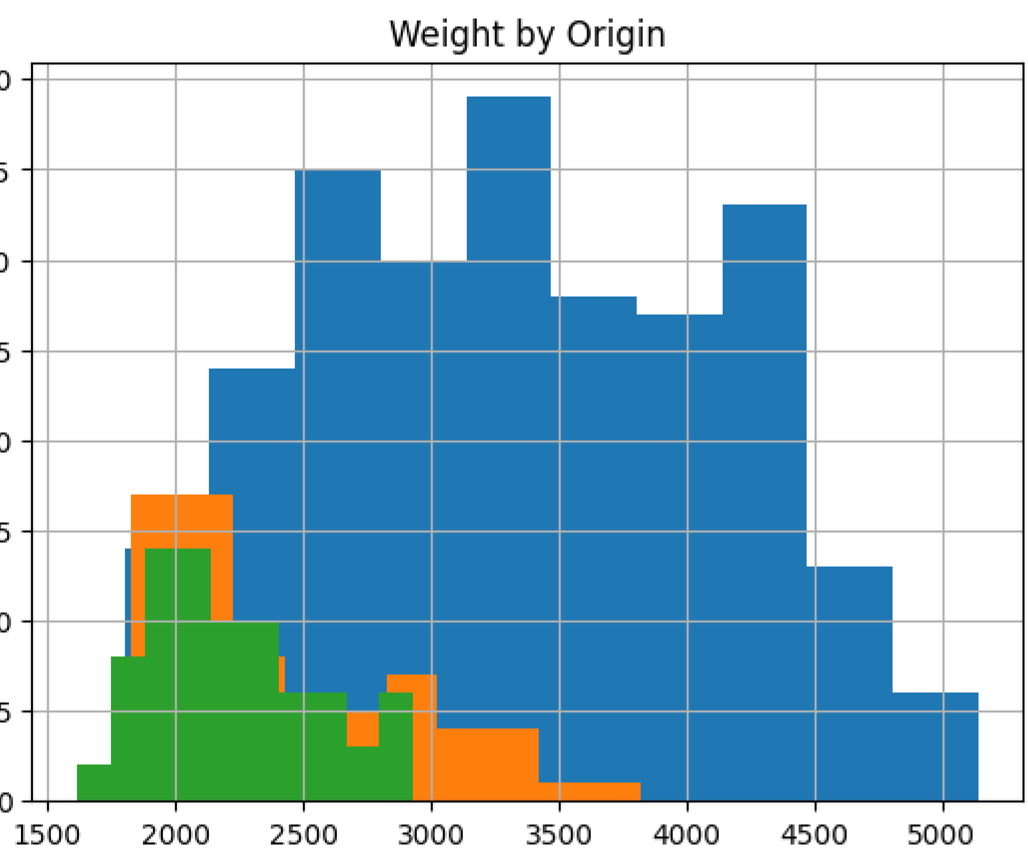
Do we have similar origin for different weights?
Plotting histograms for each origin side beside directly form the dataframe:

In [ ]:
```
axs = data.hist(column='weight', by='origin')
```



To format this plot, we can work on the axes (array) that is returned by the plot call. We use Matplotlib object oriented interface methods to do this

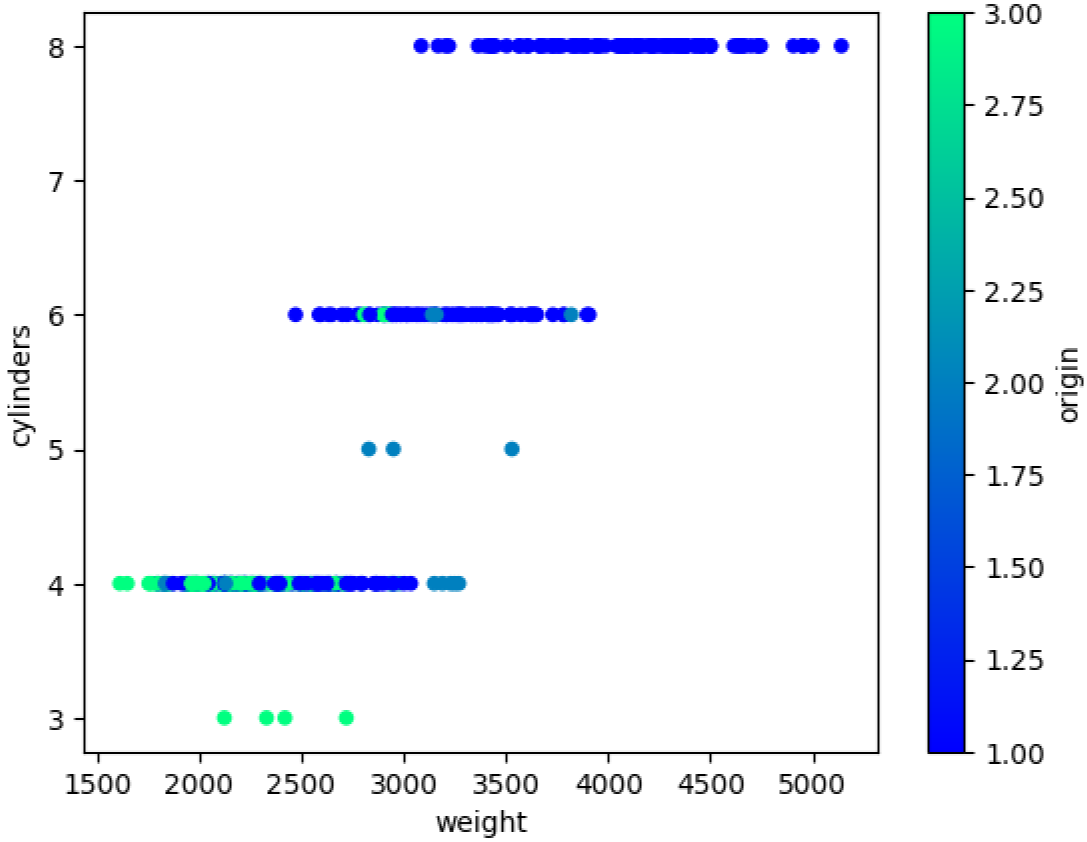
In [ ]:
```
# Using the histogram function from matplotlib instead of Pandas histogram
hist_weight = data["weight"].groupby(data['origin'])
axs = hist_weight.hist();
plt.title("Weight by Origin");
```



Is weight and cylinders correlated? Maybe it is different for different countries?

Let's have a look with a scatter plot.

```
In [ ]: data.plot.scatter('weight', 'cylinders', c='origin', colormap='winter');
```

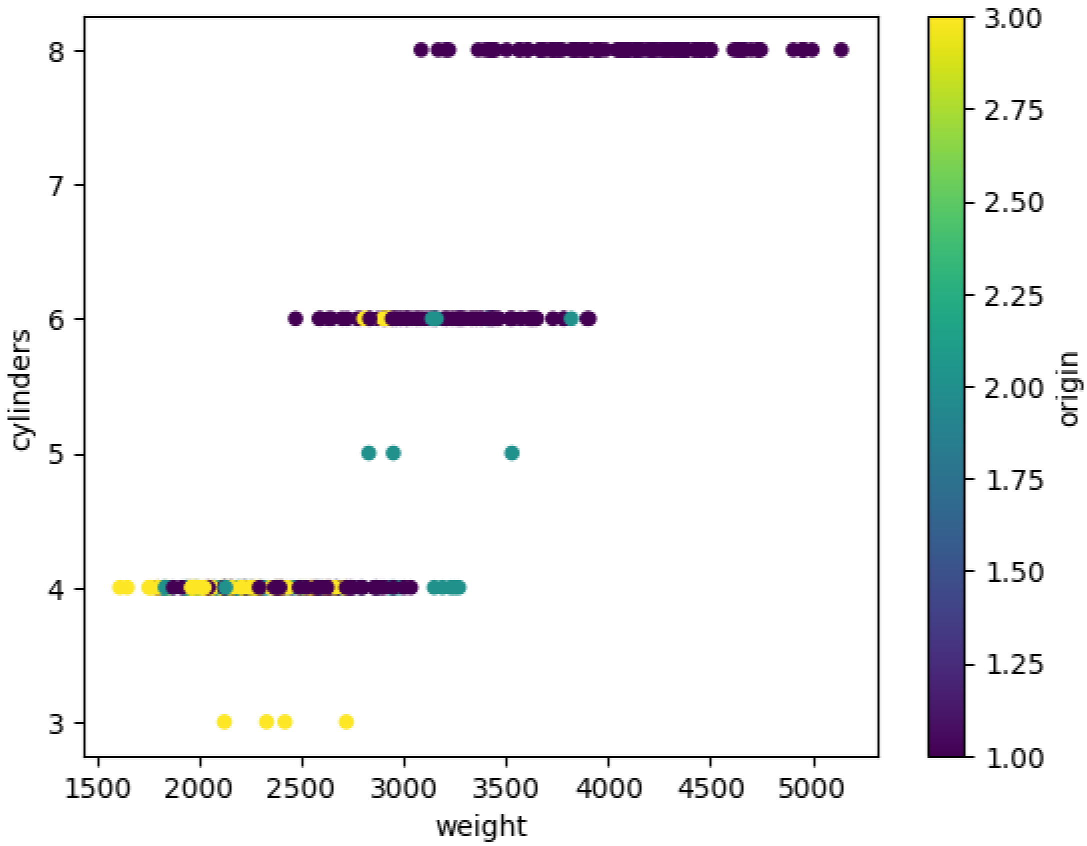


According to:
https://stackoverflow.com/questions/43578976/pandas-missing-x-tick-labels

the missing x-labels are a pandas bug.

Workaraound is to create axes prior to calling plot

```
In [ ]: fig, ax = plt.subplots()
        data.plot.scatter('weight', 'cylinders', c='origin', colormap='viridis', ax=ax);
```



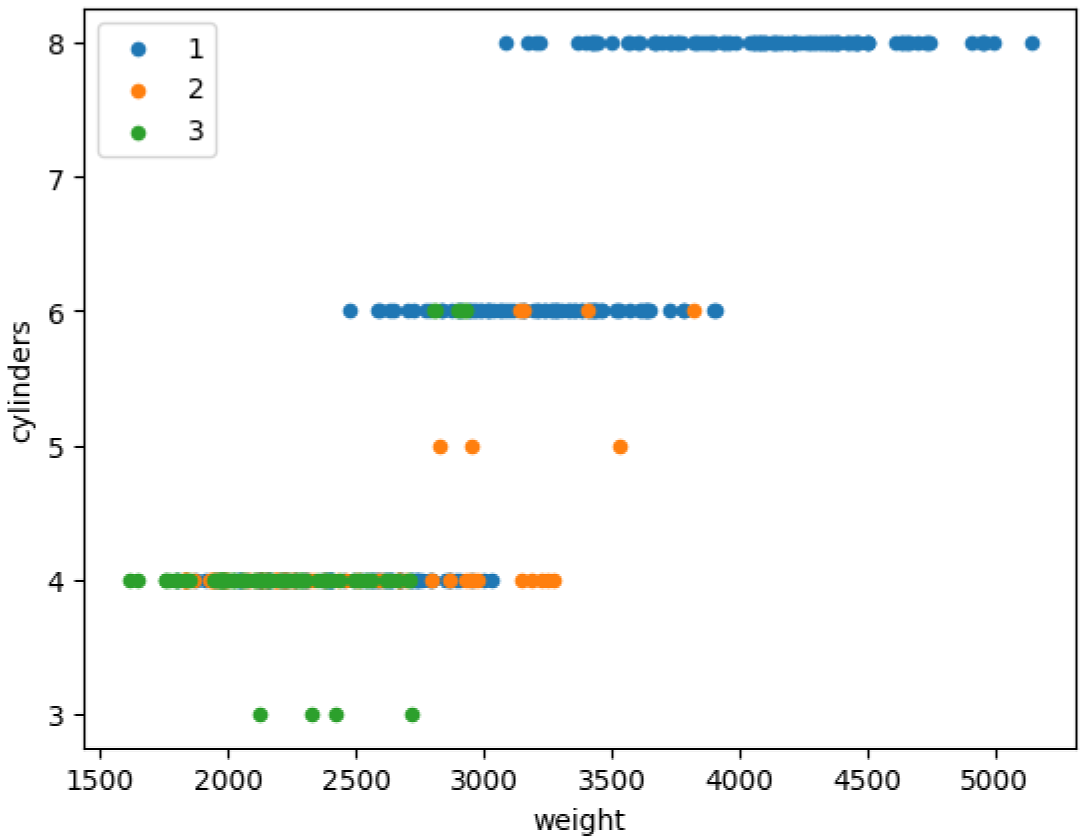
We know origin is categorical.

One way to avoid the colorbar is to loop over the categories and assign colors based on the category.

See: https://stackoverflow.com/questions/26139423/plot-different-color-for-different-categorical-levels-using-matplotlib

In [ ]:

```
colors = {1: 'tab:blue', 2: 'tab:orange', 3: 'tab:green'}
fig, ax = plt.subplots()
for key, group in data.groupby(by='origin'):
    group.plot.scatter('weight', 'cylinders', c=colors[key], label=key, ax=ax);
```



Seaborn

> Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

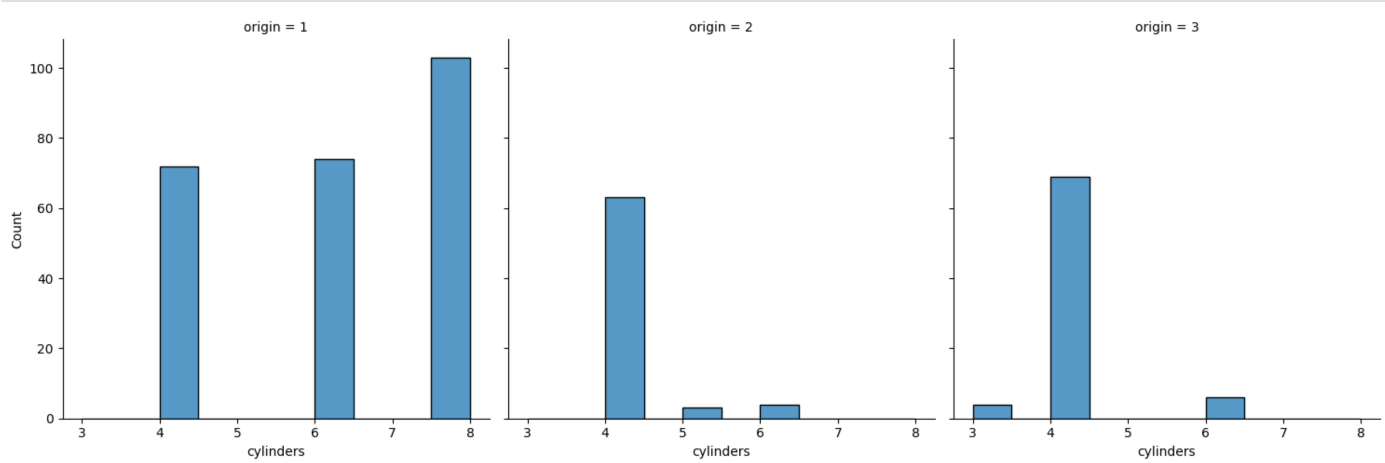
http://seaborn.pydata.org/index.html

Seaborn is usually imported as `sns`

In [ ]:
```
import seaborn as sns
```

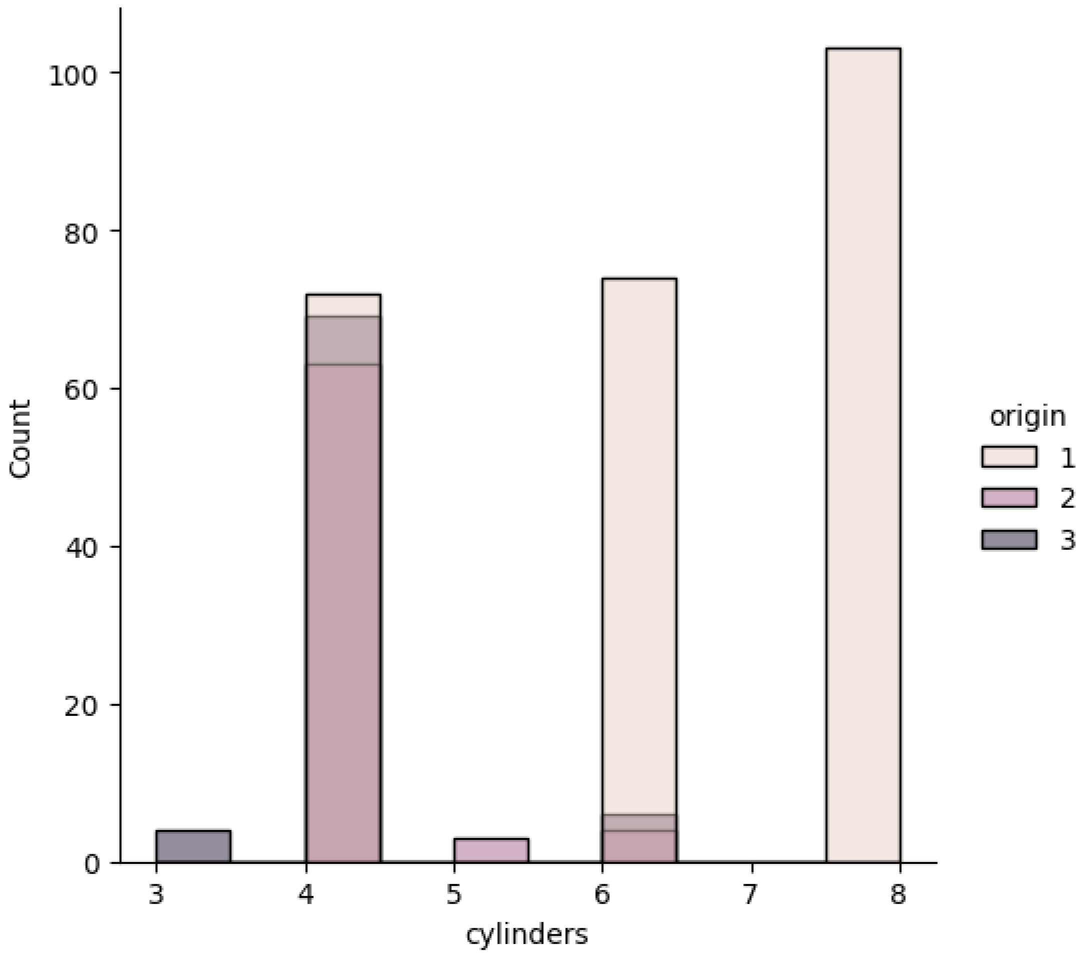
Let's re-create the histograms by gender with seaborn with the figure level `displot()` function.

In [ ]:
```
# Use gender to split age into columns
sns.displot(x='cylinders', col='origin', data=data);
```



We can display the counts in the same plot, one on top of the other.

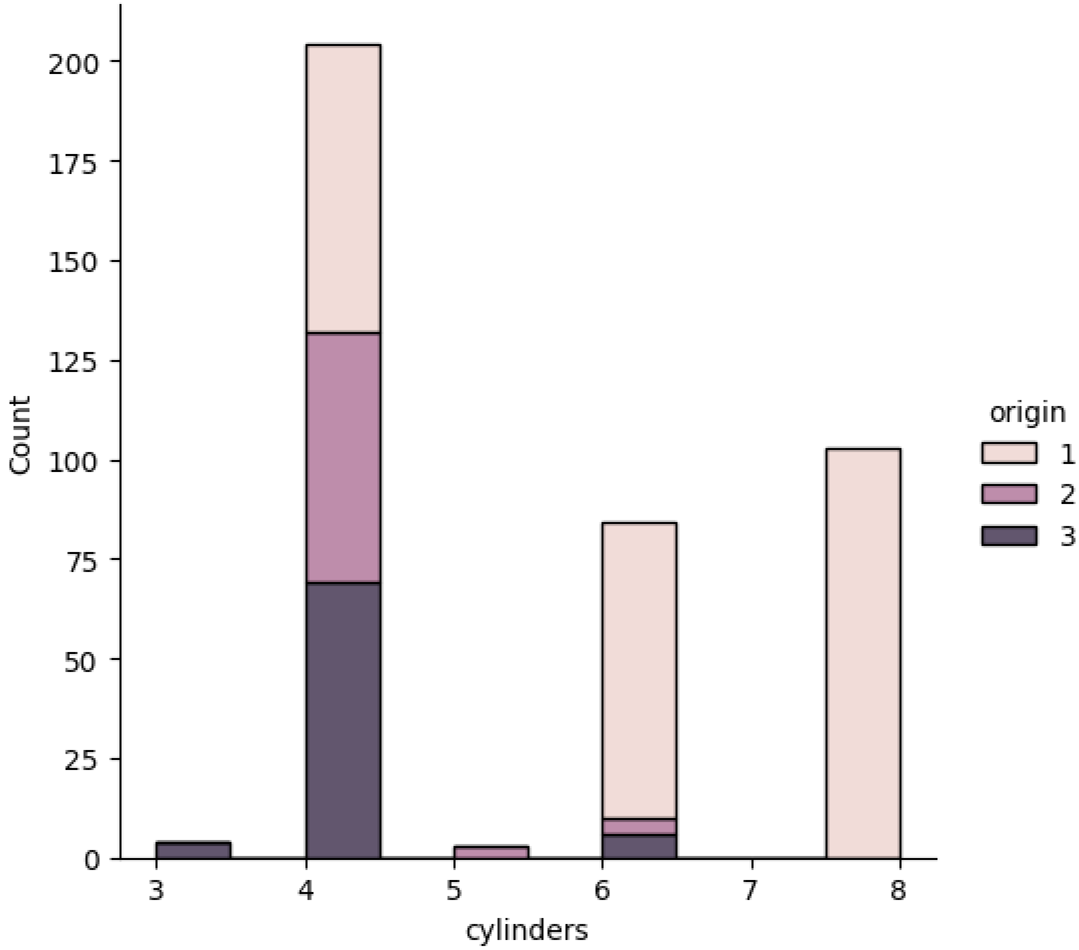
In [ ]:
```
# Use origin to color (hue) in the same plot
sns.displot(x='cylinders', hue='origin', data=data);
```



To have an idea of the split between the origins, we can stack the counts, adding up to total.

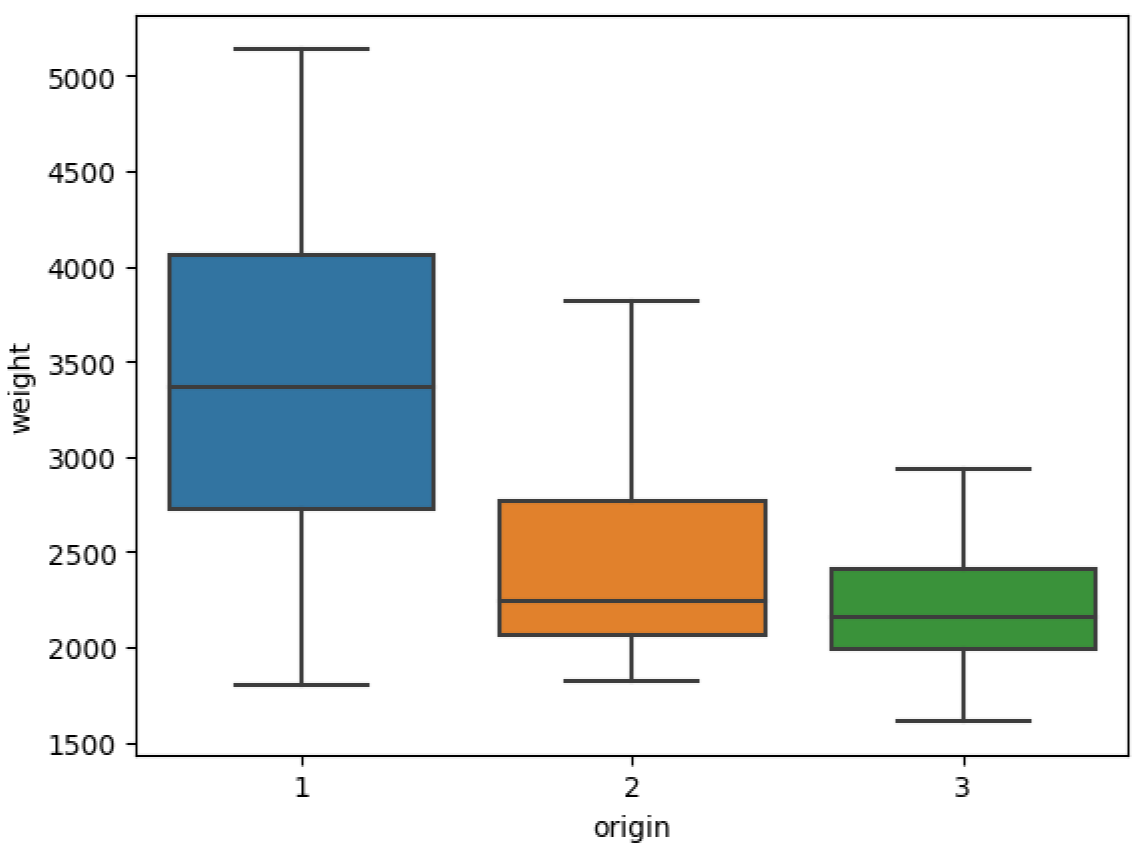
In [ ]:
```
sns.displot(x='cylinders', hue='origin', data=data, multiple='stack');
```

```
c:\Users\imaro\miniconda3\envs\ensf-ml\lib\site-packages\seaborn\distributions.py:269: FutureWar
ning: In a future version, `df.iloc[:, i] = newvals` will attempt to set the values inplace inst
ead of always setting a new array. To retain the old behavior, use either `df[df.columns[i]] = n
ewvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  baselines.iloc[:, cols] = (curves
```



We can look at the differences in weight with a boxplot too

In [ ]:
```
sns.boxplot(x='origin', y='weight', data=data);
```

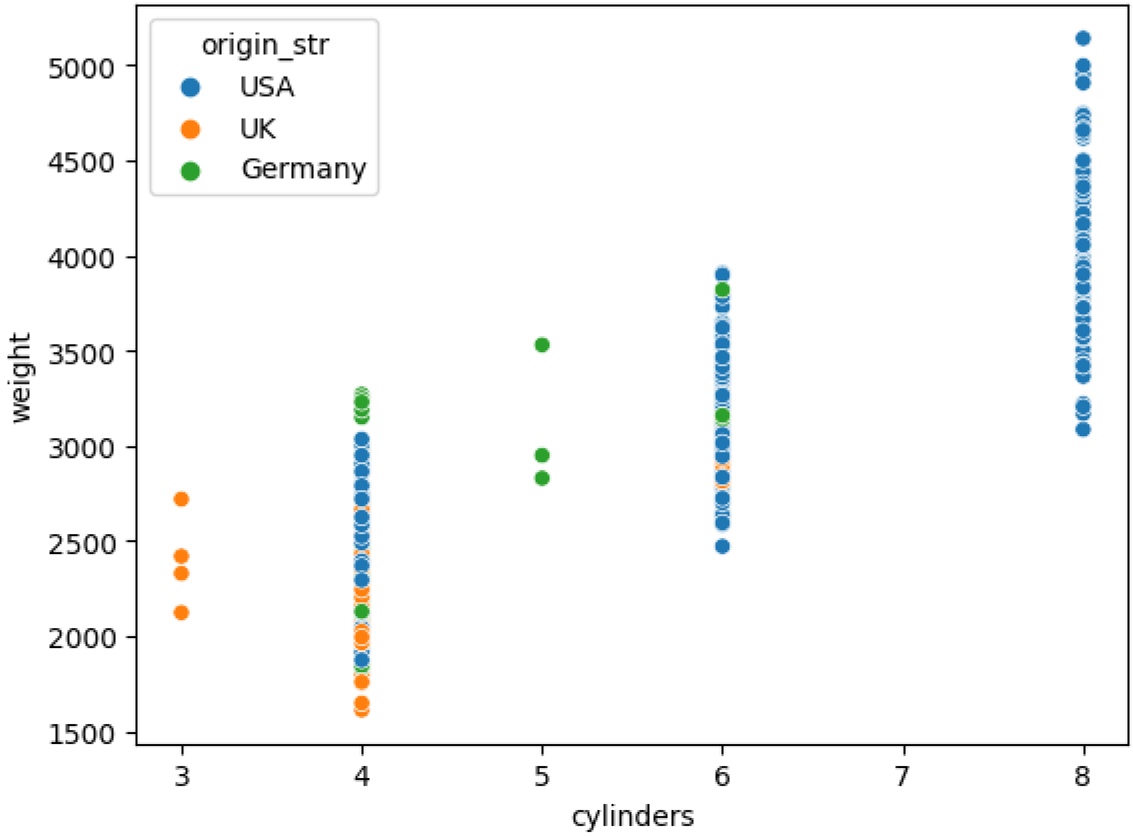


Let's re-create the scatter plot to see if cylinders and weight are correlated by origin.

To make the legend show strings we will create a origin string column with USA, Germany, and UK strings rather than 1, 2, and 3.

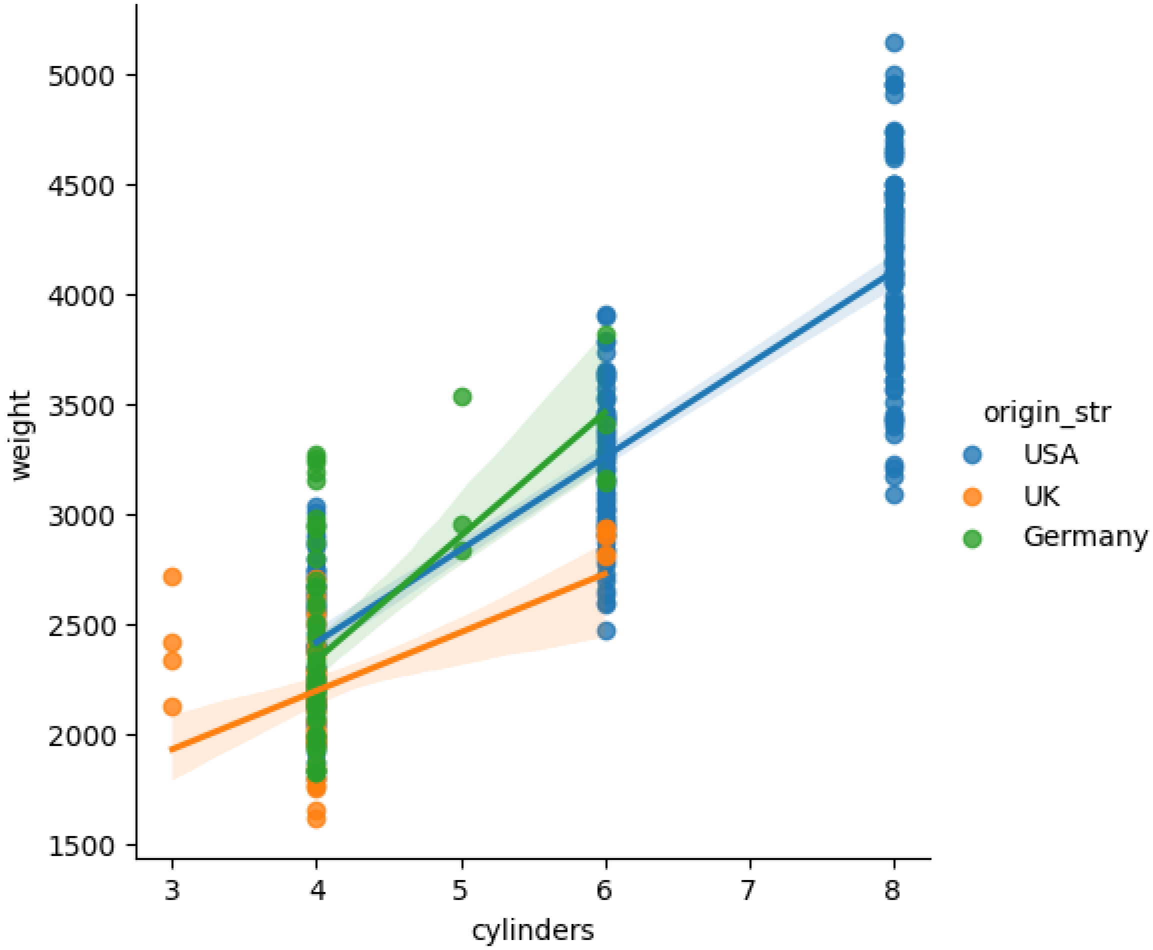
In [ ]:
```
data['origin_str'] = data['origin'].replace([1, 2, 3], ['USA', 'Germany', 'UK'])
```

In [ ]:
```
ax = sns.scatterplot(x='cylinders', y='weight', data=data, hue='origin_str')
```



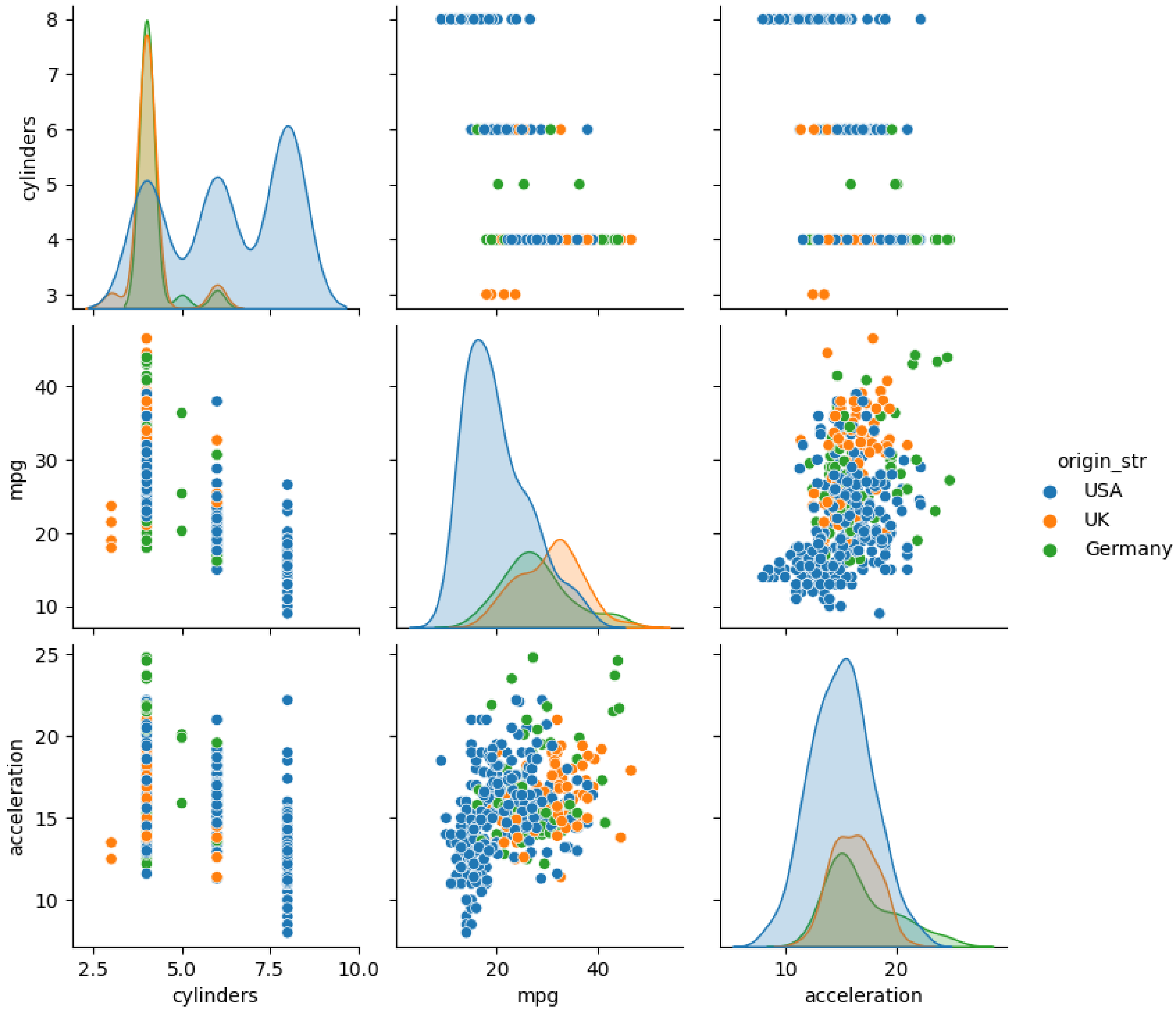
Adding a regression line helps with visualizing the relationship

In [ ]:
```
ax = sns.lmplot(x='cylinders', y='weight', data=data, hue='origin_str')
```



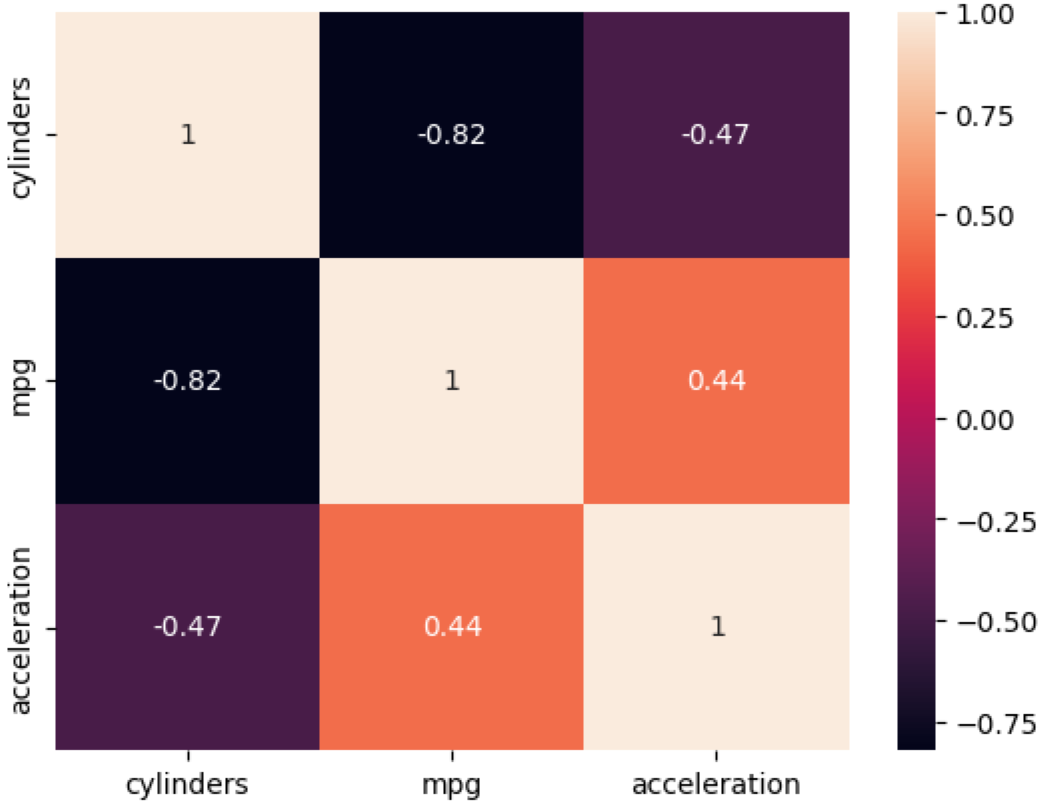
Maybe there are other correlations in the data set. Pairplot is a great way to get an overview

In [ ]:
```
sns.pairplot(data, vars=['cylinders', 'mpg', 'acceleration'], hue='origin_str');
```



As an alternative, we can visualize the correlation matrix as a heatmap

In [ ]:
```
g = sns.heatmap(data[['cylinders', 'mpg', 'acceleration']].corr(method='spearman'),
                annot=True)
```


cylinders
mpg
acceleration
1
-0.82
-0.47
-0.82
1
0.44
-0.47
0.44
1
cylinders
mpg
acceleration
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75