This is just a review of the ideas that I learned during Data Analysis with Python

For this purpose, I am going to use the data: automobile_data

Part 1: Data-Wrangling

In [1]:

```
# importing all necessary modules
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
```

In [2]:

```
mydata = pd.read_csv('raw_automobile_data.csv')
```

In [3]:

```
mydata.head()
```

Out[3]:

	Unnamed: 0	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	er loc
0	0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	
1	1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	
2	2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	
3	3	2	164.0	audi	gas	std	four	sedan	fwd	
4	4	2	164.0	audi	gas	std	four	sedan	4wd	

5 rows × 27 columns

In [15]:

```
mydata.columns
```

Out[15]:

```
Index(['Unnamed: 0', 'symboling', 'normalized-losses', 'make', 'fuel
-type',
       'aspiration', 'num-of-doors', 'body-style', 'drive-wheels',
       'engine-location', 'wheel-base', 'length', 'width', 'height',
       'curb-weight', 'engine-type', 'num-of-cylinders', 'engine-siz
e',
       'fuel-system', 'bore', 'stroke', 'compression-ratio', 'horsep
ower',
       'peak-rpm', 'city-mpg', 'highway-mpg', 'price'],
      dtype='object')
```

In [4]:

```
df=mydata.drop(['Unnamed: 0'], axis = 1)
```

In [5]:

```
df.head()
```

Out[5]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee ba
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94
3	2	164.0	audi	gas	std	four	sedan	fwd	front	99
4	2	164.0	audi	gas	std	four	sedan	4wd	front	99

5 rows × 26 columns

Identify and handle missing values

Using NaN for: '?' and blank

In [6]:

```
# replace "?" to NaN
df.replace("?", np.nan, inplace = True)
df.head(5)
```

Out[6]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee ba
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94
3	2	164.0	audi	gas	std	four	sedan	fwd	front	99
4	2	164.0	audi	gas	std	four	sedan	4wd	front	99

5 rows × 26 columns

Checking for missing values:

In [7]:

```
missing_data = df.isnull()
missing_data.head(5)
```

Out[7]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	..
0	False	True	False	False	False	False	False	False	False	False	..
1	False	True	False	False	False	False	False	False	False	False	..
2	False	True	False	False	False	False	False	False	False	False	..
3	False	False	False	False	False	False	False	False	False	False	..
4	False	False	False	False	False	False	False	False	False	False	..

5 rows × 26 columns

Count missing values in each column

In [8]:

```
for column in missing_data.columns.values.tolist():
    print(column)
    print (missing_data[column].value_counts())
    print(' ')
```

```
symboling
False    205
Name: symboling, dtype: int64

normalized-losses
False    164
True      41
Name: normalized-losses, dtype: int64

make
False    205
Name: make, dtype: int64

fuel-type
False    205
Name: fuel-type, dtype: int64

aspiration
False    205
Name: aspiration, dtype: int64

num-of-doors
False    203
True       2
Name: num-of-doors, dtype: int64

body-style
False    205
Name: body-style, dtype: int64

drive-wheels
False    205
Name: drive-wheels, dtype: int64

engine-location
False    205
Name: engine-location, dtype: int64

wheel-base
False    205
```

```
Name: wheel-base, dtype: int64

length
False    205
Name: length, dtype: int64

width
False    205
Name: width, dtype: int64

height
False    205
Name: height, dtype: int64

curb-weight
False    205
Name: curb-weight, dtype: int64

engine-type
False    205
Name: engine-type, dtype: int64

num-of-cylinders
False    205
Name: num-of-cylinders, dtype: int64

engine-size
False    205
Name: engine-size, dtype: int64

fuel-system
False    205
Name: fuel-system, dtype: int64

bore
False    201
True       4
Name: bore, dtype: int64

stroke
False    201
True       4
Name: stroke, dtype: int64

compression-ratio
False    205
Name: compression-ratio, dtype: int64

horsepower
False    203
```

```
True       2
Name: horsepower, dtype: int64

peak-rpm
False    203
True       2
Name: peak-rpm, dtype: int64

city-mpg
False    205
Name: city-mpg, dtype: int64

highway-mpg
False    205
Name: highway-mpg, dtype: int64

price
False    201
True       4
Name: price, dtype: int64
```

We see that there are missing values as follows

- There are 41 missing values in normalized-losses
- There are 2 missing values in num-of-doors
- There are 4 missing values in bore
- There are 4 missing values in stroke
- There are 2 missing values in horsepower
- There are 2 missing values in peak-rpm
- There are 4 missing values in price

Deal with missing data

How to deal with missing data? What can we do?

1. drop data
    - drop the whole row
    - drop the whole column
2. replace data
    - replace it by mean
    - replace it by frequency
    - replace it based on other functions

Idea:

Here we want to predict car price. So can do the following:

Replace by mean for the numeric columns

- "normalized-losses": 41 missing data, replace them with mean
- "stroke": 4 missing data, replace them with mean
- "bore": 4 missing data, replace them with mean
- "horsepower": 2 missing data, replace them with mean
- "peak-rpm": 2 missing data, replace them with mean

Replace by highest frequency for categorical column (replace by mostly repeated item)

- "num-of-doors": 2 missing data, replace them with "four". Since four doors is most frequent, it is most likely to occur

Drop the whole row for the column to be predicted (do not use approximated values)

- "price": 4 missing data, simply delete the whole row

Calculate the average of the column and Replace "NaN" by mean value in the column

"normalized-losses"

In [9]:

```
avg_norm_loss = df['normalized-losses'].astype("float").mean()
print("Average of normalized-losses:", avg_norm_loss)
```

```
Average of normalized-losses: 122.0
```

In [10]:

```
df["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

"bore"

In [11]:

```
avg_bore = df['bore'].astype('float').mean()
avg_bore
```

Out[11]:

```
3.3297512437810957
```

In [12]:

```
df['bore'].replace(np.nan, avg_bore, inplace = True)
```

"stroke"

In [13]:

```
avg_stroke =df['stroke'].astype('float').mean()
avg_stroke
```

Out[13]:

```
3.2554228855721337
```

In [14]:

```
df['stroke'].replace(np.nan, avg_stroke, inplace = True)
```

"horsepower"

In [15]:

```
avg_horsepower = df['horsepower'].mean()
avg_horsepower
```

Out[15]:

```
104.25615763546799
```

In [16]:

```
df['horsepower'].replace(np.nan, avg_horsepower, inplace = True)
```

"peak-rpm"

In [17]:

```
avg_peak_rpm = df['peak-rpm'].mean()
avg_peak_rpm
```

Out[17]:

```
5125.369458128079
```

In [18]:

```
df['peak-rpm'].replace(np.nan, avg_peak_rpm, inplace= True)
```

"num-of-doors"

In [19]:

```
df['num-of-doors'].replace(np.nan, "four", inplace = True)
```

Price: dropping all rows of price with nan values

In [20]:

```
#simply drop whole row with NaN in "price" column
df.dropna(subset=["price"], axis=0, inplace=True)
```

In [21]:

```
# reset index, because we droped two rows
df.reset_index(drop=True, inplace=True)
```

In [22]:

```
df.head()
```

Out[22]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee ba
0	3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88
1	3	122.0	alfa-romero	gas	std	two	convertible	rwd	front	88
2	1	122.0	alfa-romero	gas	std	two	hatchback	rwd	front	94
3	2	164.0	audi	gas	std	four	sedan	fwd	front	99
4	2	164.0	audi	gas	std	four	sedan	4wd	front	99

5 rows × 26 columns

Correcting data format

Lets list the data types for each column

In [23]:

```
df.dtypes
```

Out[23]:

```
symboling              int64
normalized-losses    float64
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower           float64
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
dtype: object
```

'bore' and 'stroke' variables are numerical values that describe the engines, so we should expect them to be of the type 'float' or 'int'; however, they are shown as type 'object'. We have to convert data types into a proper format for each column using the "astype()" method.

Changing data format

In [24]:

```
df[["bore", "stroke"]] = df[["bore", "stroke"]].astype("float")
df[["normalized-losses"]] = df[["normalized-losses"]].astype("int")
df[["price"]] = df[["price"]].astype("float")
df[["peak-rpm"]] = df[["peak-rpm"]].astype("float")
```

In [25]:

```
df.dtypes
```

Out[25]:

```
symboling              int64
normalized-losses      int64
make                  object
fuel-type             object
aspiration            object
num-of-doors          object
body-style            object
drive-wheels          object
engine-location       object
wheel-base           float64
length               float64
width                float64
height               float64
curb-weight            int64
engine-type           object
num-of-cylinders      object
engine-size            int64
fuel-system           object
bore                 float64
stroke               float64
compression-ratio    float64
horsepower           float64
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
dtype: object
```

Data Standardization

Standardization is the process of transforming data into a common format which allows the researcher to make the meaningful comparison

In [26]:

```
# Convert mpg to L/100km by mathematical operation (235 divided by mpg)
df['city-L/100km'] = 235/df["city-mpg"]

# check your transformed data
df.head()
```

Out[26]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whe ba
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94
3	2	164	audi	gas	std	four	sedan	fwd	front	99
4	2	164	audi	gas	std	four	sedan	4wd	front	99

5 rows × 27 columns

Data Normalization

Normalization is the process of transforming values of several variables into a similar range. Typical normalizations include scaling the variable so the variable average is 0, scaling the variable so the variance is 1, or scaling variable so the variable values range from 0 to 1

In [27]:

```
# Normalizing columns "length", "width" and "height"
# replace (original value) by (original value)/(maximum value)
df['length'] = df['length']/df['length'].max()
df['width'] = df['width']/df['width'].max()
df['height'] = df['height']/df['height'].max()
```

In [28]:

```
df.head()
```

Out[28]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee ba
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94
3	2	164	audi	gas	std	four	sedan	fwd	front	99
4	2	164	audi	gas	std	four	sedan	4wd	front	99

5 rows × 27 columns

Binning

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins', for grouped analysis.

In our dataset, "horsepower" is a real valued variable ranging from 48 to 288, it has 57 unique values. What if we only care about the price difference between cars with high horsepower, medium horsepower, and little horsepower (3 types)? Can we rearrange them into three 'bins' to simplify analysis?

We will use the Pandas method 'cut' to segment the 'horsepower' column into 3 bins

In [29]:

```
### Creating Bins
bins = np.linspace(min(df["horsepower"]), max(df["horsepower"]), 4)
bins
```

Out[29]:

```
array([ 48.        , 119.33333333, 190.66666667, 262.        ])
```

In [30]:

```
# naming bins
group_names = ['Low', 'Medium', 'High']
```

In [31]:

```
# Changing Data type
df["horsepower"]=df["horsepower"].astype(int, copy=True)
```

In [32]:

```
# Putting into bins
df['horsepower-binned'] = pd.cut(df['horsepower'], bins, labels=group_names, inc
lude_lowest=True )
df[['horsepower','horsepower-binned']].head(20)
```

Out[32]:

	horsepower	horsepower-binned
0	111	Low
1	111	Low
2	154	Medium
3	102	Low
4	115	Low
5	110	Low
6	110	Low
7	110	Low
8	140	Medium
9	101	Low
10	101	Low
11	121	Medium
12	121	Medium
13	121	Medium
14	182	Medium
15	182	Medium
16	182	Medium
17	48	Low
18	70	Low
19	70	Low

In [33]:

```
df["horsepower-binned"].value_counts()
```

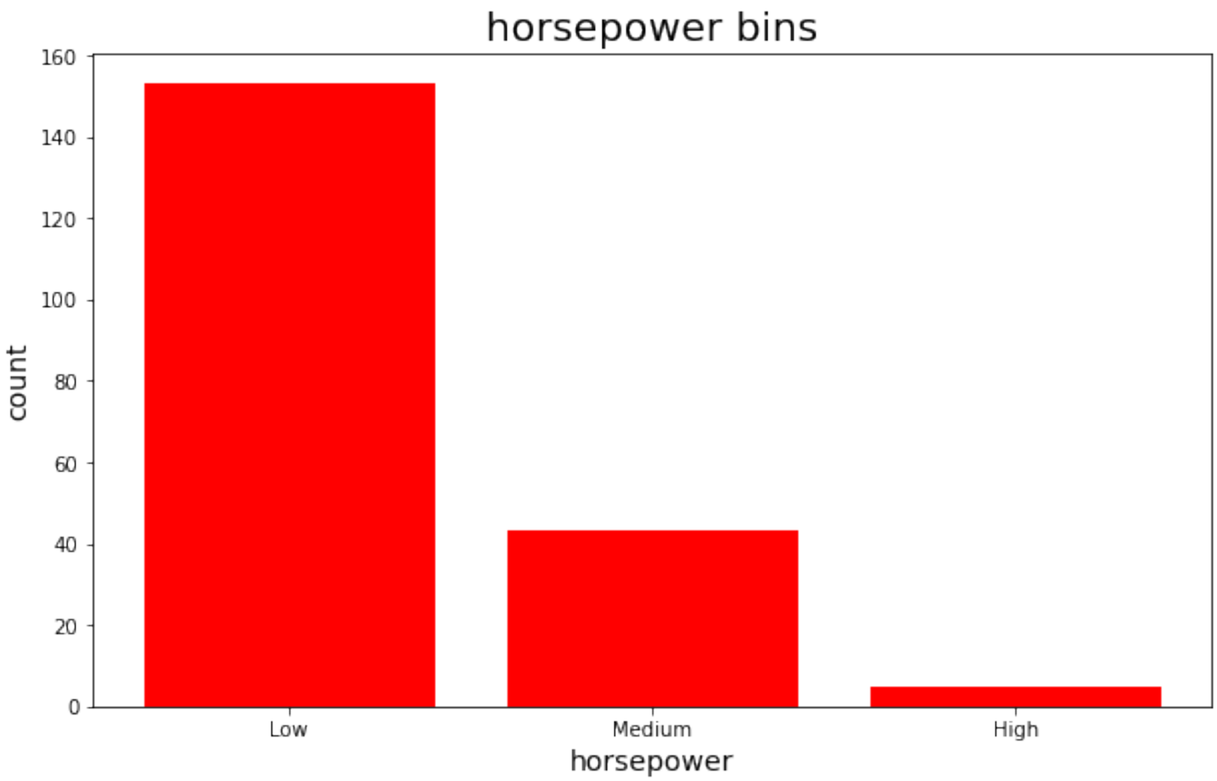
Out[33]:

```
Low        153
Medium      43
High         5
Name: horsepower-binned, dtype: int64
```

Bar chart

In [34]:

```
plt.figure(figsize=(10,6))
plt.bar(group_names, df["horsepower-binned"].value_counts(), color = 'r')
plt.xlabel("horsepower", fontsize = 14)
plt.ylabel("count", fontsize = 14)
plt.title("horsepower bins", fontsize = 20)
plt.show()
```



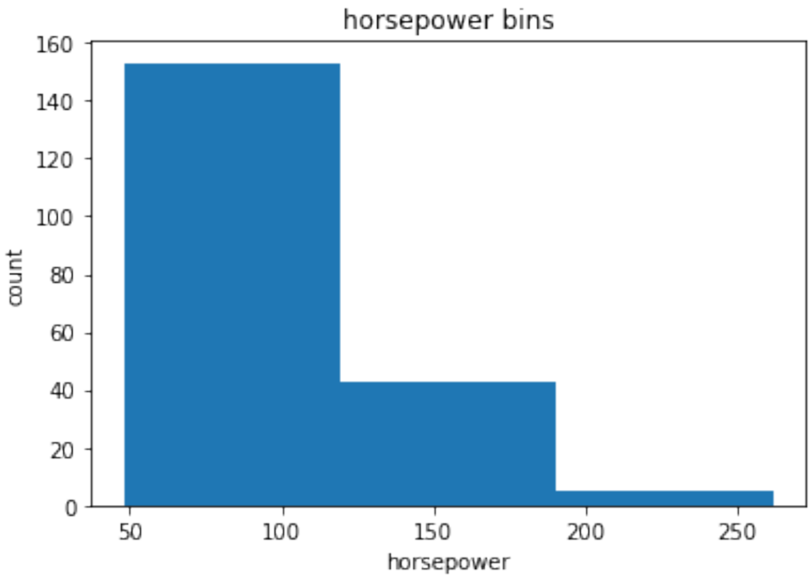
Histogram

In [35]:

```
# draw historgram of attribute "horsepower" with bins = 3
plt.hist(df["horsepower"], bins = 3)
plt.xlabel("horsepower")
plt.ylabel("count")
plt.title("horsepower bins")
```

Out[35]:

```
Text(0.5, 1.0, 'horsepower bins')
```



Indicator variable (or dummy variable)

An indicator variable (or dummy variable) is a numerical variable used to label categories. They are called 'dummies' because the numbers themselves don't have inherent meaning.

We can use categorical variables for regression analysis in the later modules. For example, We see the column "fuel-type" has two unique values, "gas" or "diesel". Regression doesn't understand words, only numbers. To use this attribute in regression analysis, we convert "fuel-type" into indicator variables 0 and 1.

Get indicator variables and assign it to data frame "dummy_variable_1"

In [36]:

```
dummy_variable_1 = pd.get_dummies(df["fuel-type"])
dummy_variable_1.head()
```

Out[36]:

	diesel	gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

In [37]:

```
#change column names for clarity
dummy_variable_1.rename(columns={'gas':'fuel-type-gas', 'diesel':'fuel-type-dies
el'}, inplace=True)
dummy_variable_1.head()
```

Out[37]:

	fuel-type-diesel	fuel-type-gas
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1

In [38]:

```
df.head()
```

Out[38]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	whee ba
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94
3	2	164	audi	gas	std	four	sedan	fwd	front	99
4	2	164	audi	gas	std	four	sedan	4wd	front	99

5 rows × 28 columns

Part 2: Exploratory-Data-Analysis

How to choose the right visualization method?

When visualizing individual variables, it is important to first understand what type of variable you are dealing with. This will help us find the right visualization method for that variable.

Also, we can calculate the correlation between variables of type "int64" or "float64" using the method "corr":

In [39]:

```
df.corr()
```

Out[39]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	
stroke	-0.008153	0.055045	0.158018	0.123952	0.188822	-0.060663	0.167438	
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	
horsepower	0.075810	0.217300	0.371178	0.579795	0.615056	-0.087001	0.757981	
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-
city-mpg	-0.035527	-0.225016	-0.470606	-0.665192	-0.633531	-0.049800	-0.749543	-
highway-mpg	0.036233	-0.181877	-0.543304	-0.698142	-0.680635	-0.104812	-0.794889	-
price	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	
city-L/100km	0.066171	0.238567	0.476153	0.657373	0.673363	0.003811	0.785353	

Regression Plot:

In order to start understanding the (linear) relationship between an individual variable and the price. We can do this by using "regplot", which plots the scatterplot plus the fitted regression line for the data.

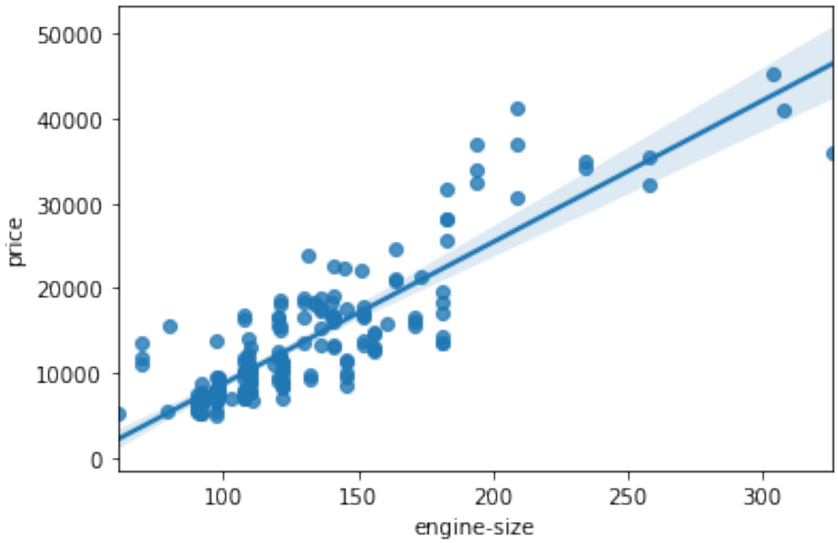
Let's find the scatterplot of "engine-size" and "price"

In [40]:

```
#Engine size as potential predictor variable of price
sns.regplot(x = "engine-size", y ="price", data=df)
plt.show
```

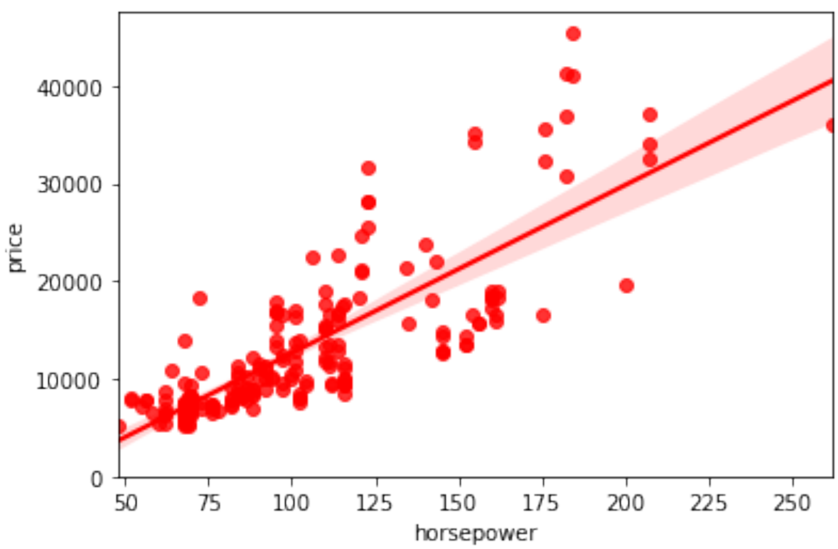
Out[40]:

```
<function matplotlib.pyplot.show(*args, **kw)>
```



In [41]:

```
# Engine size as potential predictor variable of price
sns.regplot(x="horsepower", y="price", data=df, color ='r')
plt.ylim(0,)
plt.show()
```



We can examine the correlation between 'engine-size', 'horsepower, and 'price' and see:

In [78]:

```
df[["engine-size", "horsepower", "price"]].corr()
```

Out[78]:

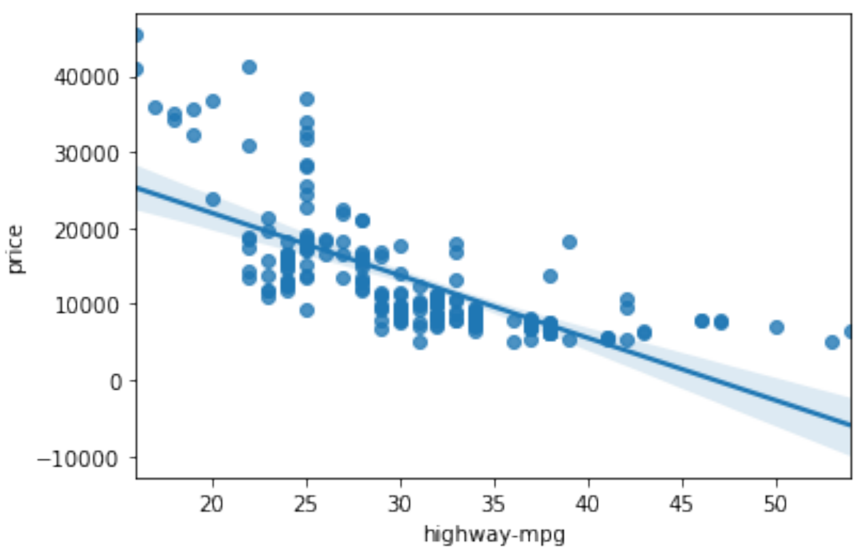
	engine-size	horsepower	price
engine-size	1.000000	0.822668	0.872335
horsepower	0.822668	1.000000	0.809607
price	0.872335	0.809607	1.000000

In [42]:

```
sns.regplot(x="highway-mpg", y="price", data=df)
```

Out[42]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe02e8d9700>
```



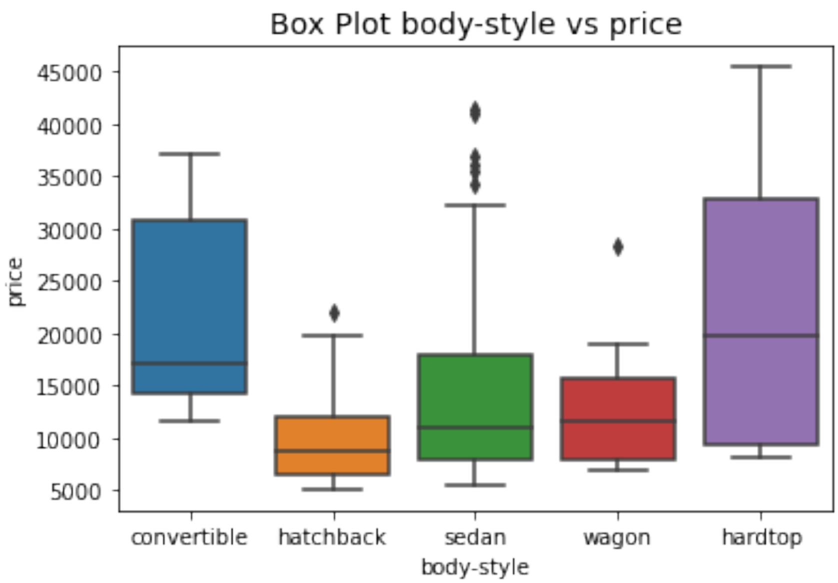
Categorical variables

The categorical variables can have the type "object" or "int64". A good way to visualize categorical variables is by using boxplots.

Let's look at the relationship between "body-style" and "price".

In [43]:

```
#relationship between "body-style" and "price".
sns.boxplot(x="body-style", y="price", data=df)
plt.title('Box Plot body-style vs price', fontsize = 14)
plt.show()
```

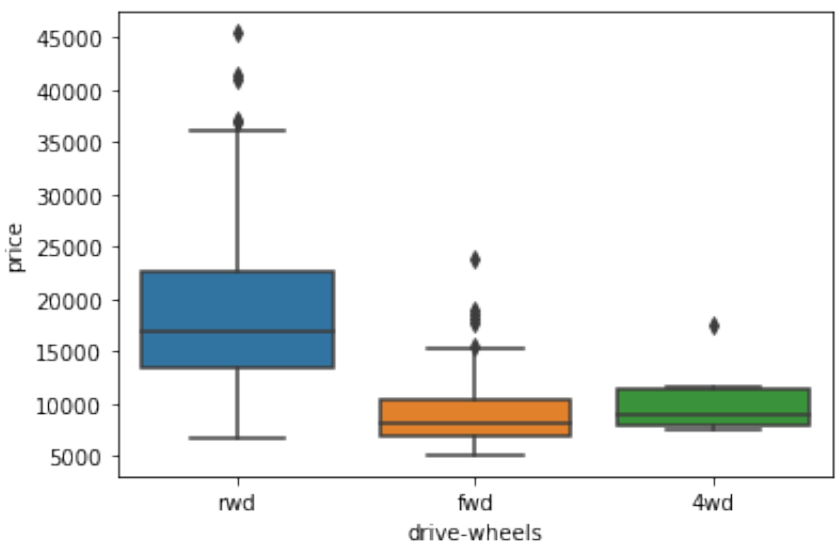


In [44]:

```
# drive-wheels
sns.boxplot(x="drive-wheels", y="price", data=df)
```

Out[44]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe02eaa9c40>
```



Descriptive Statistical Analysis

The describe function automatically computes basic statistics for all continuous variables. Any NaN values are automatically skipped in these statistics.

In [45]:

```
df.describe()
```

Out[45]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight
count	201.000000	201.00000	201.000000	201.000000	201.000000	201.000000	201.000000
mean	0.840796	122.00000	98.797015	0.837102	0.915126	0.899108	2555.666667
std	1.254802	31.99625	6.066366	0.059213	0.029187	0.040933	517.296727
min	-2.000000	65.00000	86.600000	0.678039	0.837500	0.799331	1488.000000
25%	0.000000	101.00000	94.500000	0.801538	0.890278	0.869565	2169.000000
50%	1.000000	122.00000	97.000000	0.832292	0.909722	0.904682	2414.000000
75%	2.000000	137.00000	102.400000	0.881788	0.925000	0.928094	2926.000000
max	3.000000	256.00000	120.900000	1.000000	1.000000	1.000000	4066.000000

Value Counts

Value-counts is a good way of understanding how many units of each characteristic/variable we have. We can apply the "value_counts" method on the column 'drive-wheels'.

In [46]:

```
df['drive-wheels'].value_counts()
```

Out[46]:

```
fwd    118
rwd     75
4wd      8
Name: drive-wheels, dtype: int64
```

In [47]:

```
df['drive-wheels'].value_counts().to_frame()
```

Out[47]:

	drive-wheels
fwd	118
rwd	75
4wd	8

In [48]:

```
df['engine-location'].value_counts().to_frame()
```

Out[48]:

	engine-location
front	198
rear	3

Grouping

The "groupby" method groups data by different categories. The data is grouped based on one or several variables and analysis is performed on the individual groups.

In [49]:

```
# getting category
df['drive-wheels'].unique()
```

Out[49]:

```
array(['rwd', 'fwd', '4wd'], dtype=object)
```

In [50]:

```
df_group = df[['drive-wheels','body-style','price']]
df_group.head()
```

Out[50]:

	drive-wheels	body-style	price
0	rwd	convertible	13495.0
1	rwd	convertible	16500.0
2	rwd	hatchback	16500.0
3	fwd	sedan	13950.0
4	4wd	sedan	17450.0

We can then calculate the average price for each of the different categories of data.

In [51]:

```
#Grouping by drive-wheels:
df_group_mean1 = df_group.groupby(['drive-wheels'],as_index=False).mean()
df_group_mean1
```

Out[51]:

	drive-wheels	price
0	4wd	10241.000000
1	fwd	9244.779661
2	rwd	19757.613333

In [52]:

```
#Grouping by body-style:
df_group_mean2 = df_group.groupby(['body-style'],as_index=False).mean()
df_group_mean2
```

Out[52]:

	body-style	price
0	convertible	21890.500000
1	hardtop	22208.500000
2	hatchback	9957.441176
3	sedan	14459.755319
4	wagon	12371.960000

From our data, it seems rear-wheel drive vehicles are, on average, the most expensive, while 4-wheel and front-wheel are approximately the same in price.

In [53]:

```
# grouping by more than 1 variable
df_gptest = df[['drive-wheels','body-style','price']]
grouped_test1 = df_gptest.groupby(['drive-wheels','body-style'],as_index=False).
mean()
grouped_test1
```

Out[53]:

	drive-wheels	body-style	price
0	4wd	hatchback	7603.000000
1	4wd	sedan	12647.333333
2	4wd	wagon	9095.750000
3	fwd	convertible	11595.000000
4	fwd	hardtop	8249.000000
5	fwd	hatchback	8396.387755
6	fwd	sedan	9811.800000
7	fwd	wagon	9997.333333
8	rwd	convertible	23949.600000
9	rwd	hardtop	24202.714286
10	rwd	hatchback	14337.777778
11	rwd	sedan	21711.833333
12	rwd	wagon	16994.222222

Heat Map: Variables: Drive Wheels and Body Style vs Price

Pivot Table

In [54]:

```
grouped_pivot = grouped_test1.pivot(index='drive-wheels',columns='body-style')
grouped_pivot
```

Out[54]:

	price				
body-style	**convertible**	**hardtop**	**hatchback**	**sedan**	**wagon**
drive-wheels					
4wd	NaN	NaN	7603.000000	12647.333333	9095.750000
fwd	11595.0	8249.000000	8396.387755	9811.800000	9997.333333
rwd	23949.6	24202.714286	14337.777778	21711.833333	16994.222222

In [55]:

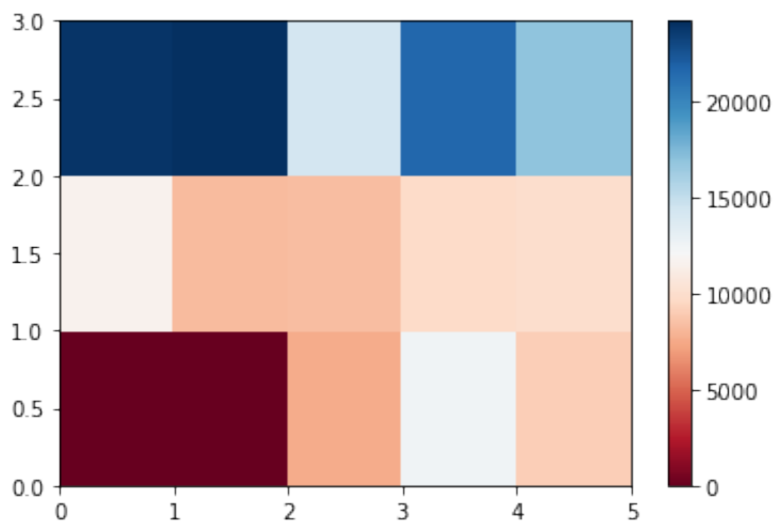
```
# fill NaN by 0
#fill missing values with 0 in grouped_pivot
grouped_pivot = grouped_pivot.fillna(0)
grouped_pivot
```

Out[55]:

	price				
body-style	**convertible**	**hardtop**	**hatchback**	**sedan**	**wagon**
drive-wheels					
4wd	0.0	0.000000	7603.000000	12647.333333	9095.750000
fwd	11595.0	8249.000000	8396.387755	9811.800000	9997.333333
rwd	23949.6	24202.714286	14337.777778	21711.833333	16994.222222

In [56]:

```
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```



In [57]:

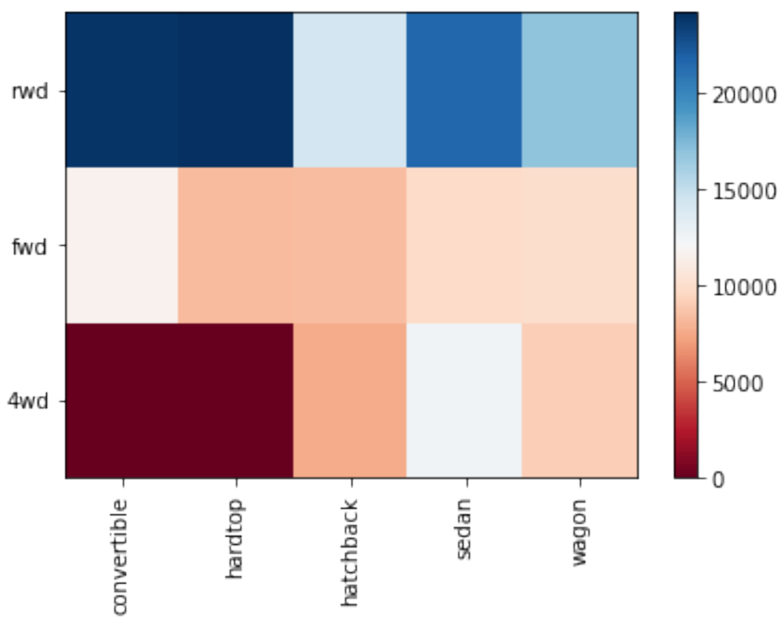
```
fig, ax = plt.subplots()
im = ax.pcolor(grouped_pivot, cmap='RdBu')

#label names
row_labels = grouped_pivot.columns.levels[1]
col_labels = grouped_pivot.index

#move ticks and labels to the center
ax.set_xticks(np.arange(grouped_pivot.shape[1]) + 0.5, minor=False)
ax.set_yticks(np.arange(grouped_pivot.shape[0]) + 0.5, minor=False)

#insert labels
ax.set_xticklabels(row_labels, minor=False)
ax.set_yticklabels(col_labels, minor=False)

#rotate label if too long
plt.xticks(rotation=90)
fig.colorbar(im)
plt.show()
```



The main question we want to answer in this module, is "What are the main characteristics which have the most impact on the car price?".

To get a better measure of the important characteristics, we look at the correlation of these variables with the car price, in other words: how is the car price dependent on this variable?

Correlation and Causation

In [58]:

```
df.corr()
```

Out[58]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	
stroke	-0.008153	0.055045	0.158018	0.123952	0.188822	-0.060663	0.167438	
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	
horsepower	0.075810	0.217300	0.371178	0.579795	0.615056	-0.087001	0.757981	
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-
city-mpg	-0.035527	-0.225016	-0.470606	-0.665192	-0.633531	-0.049800	-0.749543	-
highway-mpg	0.036233	-0.181877	-0.543304	-0.698142	-0.680635	-0.104812	-0.794889	-
price	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	
city-L/100km	0.066171	0.238567	0.476153	0.657373	0.673363	0.003811	0.785353	

sometimes we would like to know the significant of the correlation estimate

By convention, when the

- p-value is $<$ 0.001: we say there is strong evidence that the correlation is significant.
- p-value is $<$ 0.05: there is moderate evidence that the correlation is significant.
- p-value is $<$ 0.1: there is weak evidence that the correlation is significant.
- p-value is $>$ 0.1: there is no evidence that the correlation is significant.

We can obtain this information using "stats" module in the "scipy" library.

In [59]:

```
from scipy import stats
```

wheel-base VS price

In [60]:

```
pearson_coef, p_value = stats.pearsonr(df['wheel-base'], df['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-value o
f P =", p_value)
```

```
The Pearson Correlation Coefficient is 0.584641822265508  with a P-v
alue of P = 8.076488270733218e-20
```

Conclusion:

Since the p-value is < 0.001, the correlation between wheel-base and price is statistically significant, although the linear relationship isn't extremely strong (~0.585)

horsepower VS price

In [61]:

```
pearson_coef, p_value = stats.pearsonr(df['horsepower'], df['price'])
pearson_coef, p_value
```

Out[61]:

```
(0.8096068016571054, 6.273536270650504e-48)
```

Conclusion:

Since the p-value is < 0.001, the correlation between horsepower and price is statistically significant, and the linear relationship is quite strong (~0.809, close to 1)

Length vs Price

In [62]:

```
pearson_coef, p_value = stats.pearsonr(df['length'], df['price'])
pearson_coef, p_value
```

Out[62]:

```
(0.6906283804483639, 8.016477466159328e-30)
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between length and price is statistically significant, and the linear relationship is moderately strong (~0.691).

Width vs Price

In [63]:

```
pearson_coef, p_value = stats.pearsonr(df['width'], df['price'])
pearson_coef, p_value
```

Out[63]:

```
(0.7512653440522673, 9.200335510481646e-38)
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between width and price is statistically significant, and the linear relationship is quite strong (~0.751).

Curb-weight vs Price

In [64]:

```
pearson_coef, p_value = stats.pearsonr(df['curb-weight'], df['price'])
pearson_coef, p_value
```

Out[64]:

```
(0.8344145257702844, 2.189577238893878e-53)
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between curb-weight and price is statistically significant, and the linear relationship is quite strong (~0.834).

Engine-size vs Price

In [65]:

```
pearson_coef, p_value = stats.pearsonr(df['engine-size'], df['price'])
pearson_coef, p_value
```

Out[65]:

```
(0.8723351674455185, 9.265491622198389e-64)
```

Conclusion:

Since the p-value is < 0.001, the correlation between engine-size and price is statistically significant, and the linear relationship is very strong (~0.872).

Bore vs Price

In [66]:

```
pearson_coef, p_value = stats.pearsonr(df['bore'], df['price'])
pearson_coef, p_value
```

Out[66]:

```
(0.5431553832626602, 8.049189483935489e-17)
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between bore and price is statistically significant, but the linear relationship is only moderate (~0.521).

City-mpg vs Price

In [67]:

```
pearson_coef, p_value = stats.pearsonr(df['city-mpg'], df['price'])
pearson_coef, p_value
```

Out[67]:

```
(-0.6865710067844678, 2.321132065567641e-29)
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between city-mpg and price is statistically significant, and the coefficient of ~ -0.687 shows that the relationship is negative and moderately strong.

Highway-mpg vs Price

In [68]:

```
pearson_coef, p_value = stats.pearsonr(df['highway-mpg'], df['price'])
pearson_coef, p_value
```

Out[68]:

```
(-0.704692265058953, 1.7495471144476358e-31)
```

Conclusion:

Since the p-value is < 0.001, the correlation between highway-mpg and price is statistically significant, and the coefficient of ~ -0.705 shows that the relationship is negative and moderately strong.

ANOVA: Analysis of Variance

The Analysis of Variance (ANOVA) is a statistical method used to test whether there are significant differences between the means of two or more groups. ANOVA returns two parameters:

F-test score: ANOVA assumes the means of all groups are the same, calculates how much the actual means deviate from the assumption, and reports it as the F-test score. A larger score means there is a larger difference between the means.

P-value: P-value tells how statistically significant is our calculated score value.

If our price variable is strongly correlated with the variable we are analyzing, expect ANOVA to return asizeable F-test score and a small p-value.

Drive Wheels

Since ANOVA analyzes the difference between different groups of the same variable, the groupby function will come in handy. Because the ANOVA algorithm averages the data automatically, we do not need to take the average before hand.

Let's see if different types 'drive-wheels' impact 'price', we group the data.

In [69]:

```
grouped_test2=df_gptest[['drive-wheels', 'price']].groupby(['drive-wheels'])
grouped_test2.head(2)
```

Out[69]:

	drive-wheels	price
0	rwd	13495.0
1	rwd	16500.0
3	fwd	13950.0
4	4wd	17450.0
5	fwd	15250.0
136	4wd	7603.0

We can obtain the values of the method group using the method "get_group".

In [70]:

```
grouped_test2.get_group('4wd')['price']
```

Out[70]:

```
4      17450.0
136     7603.0
140     9233.0
141    11259.0
144     8013.0
145    11694.0
150     7898.0
151     8778.0
Name: price, dtype: float64
```

In [71]:

```
#ANOVA
f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_t
est2.get_group('rwd')['price'], grouped_test2.get_group('4wd')['price'])
print( "ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 67.95406500780399 , P = 3.3945443577151245e-23
```

fwd and rwd

In [72]:

```
f_val, p_val = stats.f_oneway(grouped_test2.get_group('fwd')['price'], grouped_t
est2.get_group('rwd')['price'])
print( "ANOVA results: F=", f_val, ", P =", p_val )
```

```
ANOVA results: F= 130.5533160959111 , P = 2.2355306355677845e-23
```

4wd and rwd

In [73]:

```
f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_t
est2.get_group('rwd')['price'])
print( "ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 8.580681368924756 , P = 0.004411492211225333
```

4wd and fwd

In [74]:

```
f_val, p_val = stats.f_oneway(grouped_test2.get_group('4wd')['price'], grouped_t
est2.get_group('fwd')['price'])
print("ANOVA results: F=", f_val, ", P =", p_val)
```

```
ANOVA results: F= 0.665465750252303 , P = 0.41620116697845666
```

Conclusion: Important Variables

We now have a better idea of what our data looks like and which variables are important to take into account when predicting the car price. We have narrowed it down to the following variables:

Continuous numerical variables:

- Length
- Width
- Curb-weight
- Engine-size
- Horsepower
- City-mpg
- Highway-mpg
- Wheel-base
- Bore

Categorical variables:

- Drive-wheels

Part 3: Model Development

We are working with the data:

In [75]:

```
df
```

Out[75]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	w
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	
3	2	164	audi	gas	std	four	sedan	fwd	front	
4	2	164	audi	gas	std	four	sedan	4wd	front	
...	...	...	...	...	...	...	...	...	...	
196	-1	95	volvo	gas	std	four	sedan	rwd	front	
197	-1	95	volvo	gas	turbo	four	sedan	rwd	front	
198	-1	95	volvo	gas	std	four	sedan	rwd	front	
199	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	
200	-1	95	volvo	gas	turbo	four	sedan	rwd	front	

201 rows × 28 columns

Linear Regression

In [76]:

```
from sklearn.linear_model import LinearRegression
```

In [77]:

```
lm = LinearRegression()
lm
```

Out[77]:

```
LinearRegression()
```

In [78]:

```
X = df[['highway-mpg']]
Y = df['price']
```

In [79]:

```
lm.fit(X,Y)
```

Out[79]:

```
LinearRegression()
```

In [80]:

```
Yhat=lm.predict(X)
Yhat[0:5]
```

Out[80]:

```
array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085
,
       20345.17153508])
```

In [81]:

```
# y- intercept
lm.intercept_
```

Out[81]:

```
38423.305858157415
```

In [82]:

```
# slope
lm.coef_
```

Out[82]:

```
array([-821.73337832])
```

linear model

price = 38423.31 - 821.73 x highway-mpg

Multiple Linear Regression

From the previous section we know that other good predictors of price could be:

- Horsepower
- Curb-weight
- Engine-size
- Highway-mpg

Let's develop a model using these variables as the predictor variables.

In [83]:

```
Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]
```

In [84]:

```
# Fit the linear model using the four above-mentioned variables.
lm.fit(Z, df['price'])
```

Out[84]:

```
LinearRegression()
```

In [85]:

```
# intercept
lm.intercept_
```

Out[85]:

```
-15811.863767729246
```

In [86]:

```
# coefficients(slope)
lm.coef_
```

Out[86]:

```
array([53.53022809,  4.70805253, 81.51280006, 36.1593925 ])
```

linear model

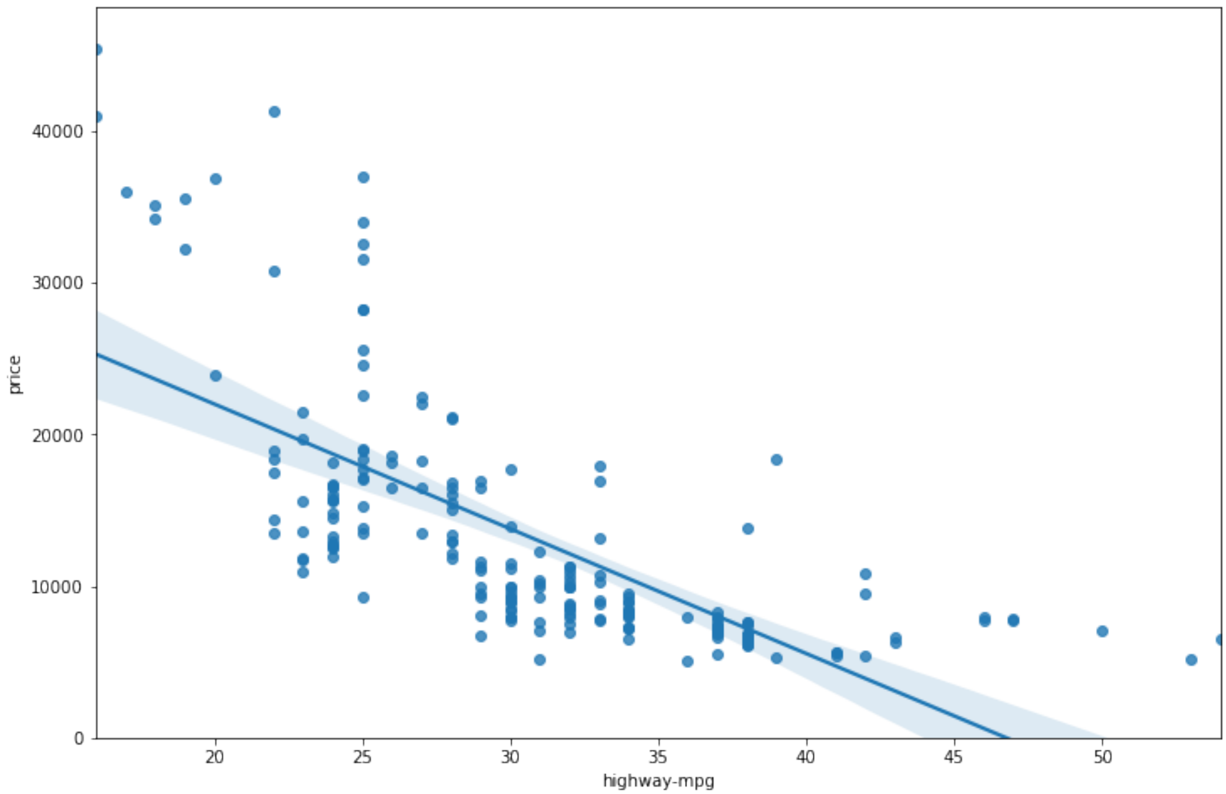
Price = -15678.742628061467 + 52.65851272 x horsepower + 4.69878948 x curb-weight + 81.95906216 x engine-size + 33.58258185 x highway-mpg

Model Evaluation using Visualization

Regression Plot

In [89]:

```
import seaborn as sns
%matplotlib inline
plt.figure(figsize=(12, 8))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
plt.show()
```



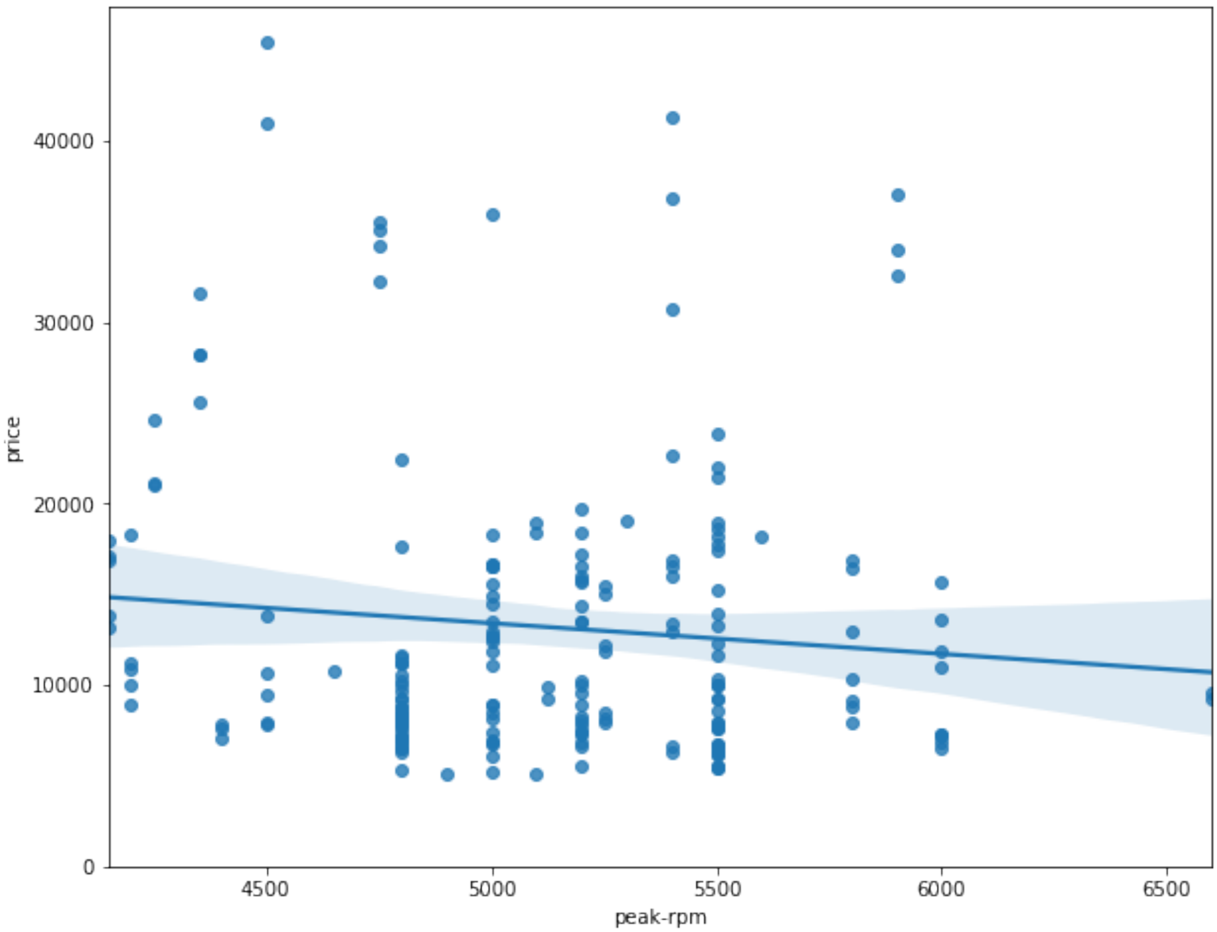
We can see from this plot that price is negatively correlated to highway-mpg,

In [90]:

```
plt.figure(figsize=(10, 8))
sns.regplot(x="peak-rpm", y="price", data=df)
plt.ylim(0,)
```

Out[90]:

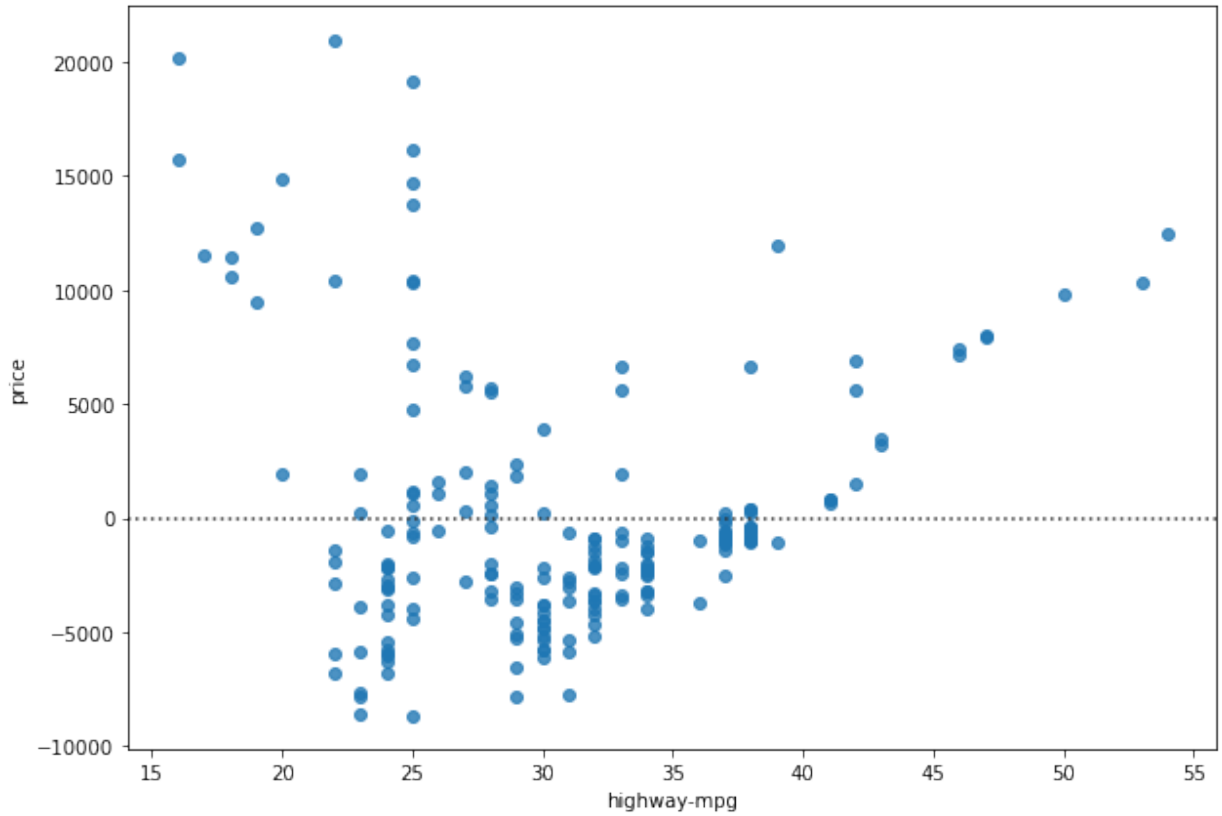
```
(0.0, 47414.1)
```



Residual Plot

In [92]:

```
plt.figure(figsize=(10, 7))
sns.residplot(df['highway-mpg'], df['price'])
plt.show()
```



We can see from this residual plot that the residuals are not randomly spread around the x-axis, which leads us to believe that maybe a non-linear model is more appropriate for this data.

Multiple Linear Regression

How do we visualize a model for Multiple Linear Regression? This gets a bit more complicated because you can't visualize it with regression or residual plot. One way to look at the fit of the model is by looking at the distribution plot:

Distribution plot:

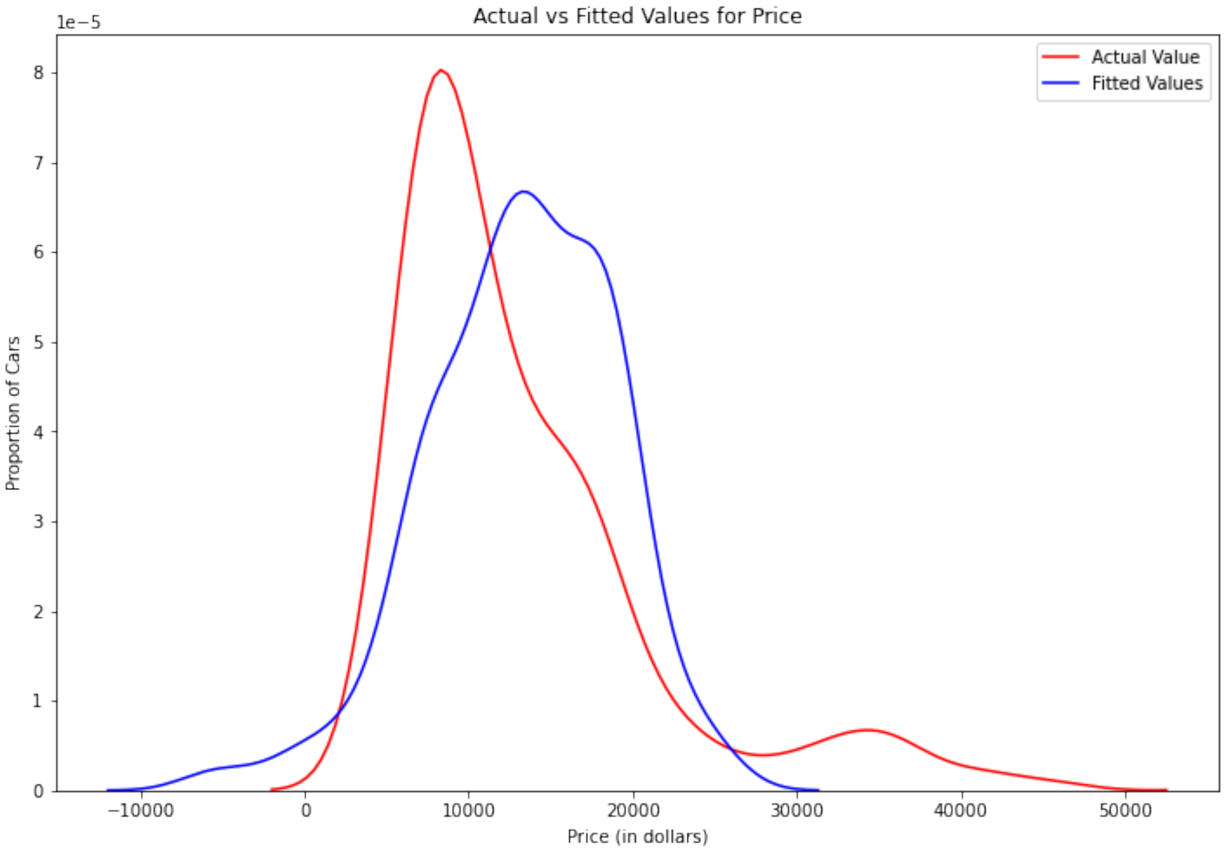
We can look at the distribution of the fitted values that result from the model and compare it to the distribution of the actual values.

In [99]:

```
Y_hat = lm.predict(Z)
plt.figure(figsize=(12, 8))

ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
ax1 = sns.distplot(Yhat, hist=False, color="b", label="Fitted Values")

plt.title('Actual vs Fitted Values for Price')
plt.xlabel('Price (in dollars)')
plt.ylabel('Proportion of Cars')
plt.show()
plt.close()
```



We can see that the fitted values are reasonably close to the actual values, since the two distributions overlap a bit. However, there is definitely some room for improvement.

Polynomial Regression and Pipelines

Polynomial Regression

We saw earlier that a linear model did not provide the best fit while using highway-mpg as the predictor variable. Let's see if we can try fitting a polynomial model to the data instead.

We will use the following function to plot the data:

In [100]:

```
def PlotPolly(model, independent_variable, dependent_variabble, Name):
    x_new = np.linspace(15, 55, 100)
    y_new = model(x_new)
    plt.plot(independent_variable, dependent_variabble, '.', x_new, y_new, '-')
    plt.title('Polynomial Fit with Matplotlib for Price ~ Length')
    ax = plt.gca()
    ax.set_facecolor((0.898, 0.898, 0.898))
    fig = plt.gcf()
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')
    plt.show()
    plt.close()
```

In [101]:

```
x = df['highway-mpg']
y = df['price']
```

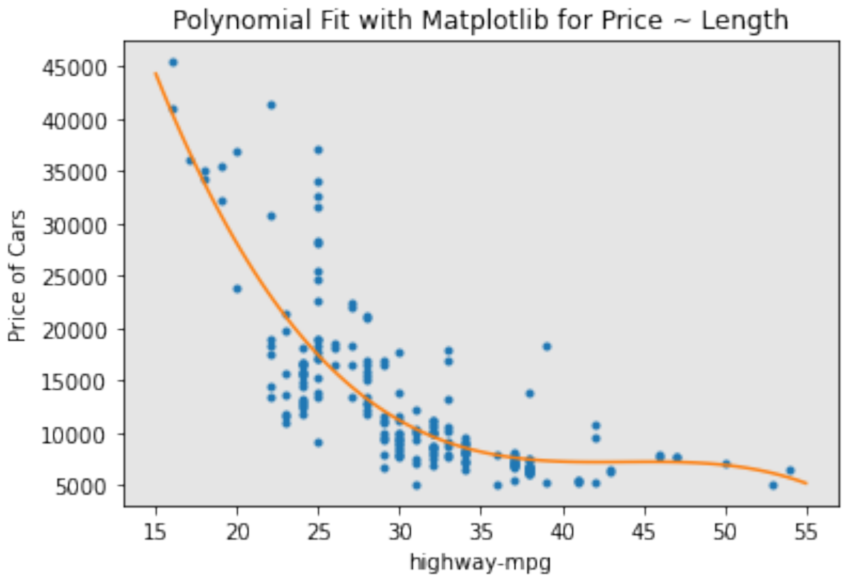
In [102]:

```
# Here we use a polynomial of the 3rd order (cubic)
f = np.polyfit(x, y, 3)
p = np.poly1d(f)
print(p)
```

```
        3         2
-1.557 x + 204.8 x - 8965 x + 1.379e+05
```

In [103]:

```
PlotPolly(p, x, y, 'highway-mpg')
```



In [104]:

```
np.polyfit(x, y, 3)
```

Out[104]:

```
array([-1.55663829e+00,  2.04754306e+02, -8.96543312e+03,  1.3792359
4e+05])
```

We can already see from plotting that this polynomial model performs better than the linear model. This is because the generated polynomial function "hits" more of the data points.

Pipeline

Data Pipelines simplify the steps of processing the data. We use the module Pipeline to create a pipeline.

We also use StandardScaler as a step in our pipeline.

In [109]:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
```

In [110]:

```
Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include_bia
s=False)), ('model',LinearRegression())]
```

In [111]:

```
pipe=Pipeline(Input)
pipe
```

Out[111]:

```
Pipeline(steps=[('scale', StandardScaler()),
                ('polynomial', PolynomialFeatures(include_bias=False
)),
                ('model', LinearRegression())])
```

In [112]:

```
pipe.fit(Z,y)
```

Out[112]:

```
Pipeline(steps=[('scale', StandardScaler()),
                ('polynomial', PolynomialFeatures(include_bias=False
)),
                ('model', LinearRegression())])
```

In [114]:

```
ypipe=pipe.predict(Z)
ypipe[0:4]
```

Out[114]:

```
array([13102.93329646, 13102.93329646, 18226.43450275, 10391.0918395
5])
```

Measures for In-Sample Evaluation

Two very important measures that are often used in Statistics to determine the accuracy of a model are:

- R^2 or R-squared
- Mean Squared Error (MSE)

Model 1: Simple Linear Regression

In [115]:

```
#highway_mpg_fit
lm.fit(X, Y)
# Find the R^2
print('The R-square is: ', lm.score(X, Y))
```

```
The R-square is:  0.4965911884339176
```

In [116]:

```
Yhat=lm.predict(X)
```

In [117]:

```
from sklearn.metrics import mean_squared_error
```

In [118]:

```
mse = mean_squared_error(df['price'], Yhat)
print('The mean square error of price and predicted value is: ', mse)
```

```
The mean square error of price and predicted value is:  31635042.944
639888
```

Model 2: Multiple Linear Regression

In [119]:

```
# fit the model
lm.fit(Z, df['price'])
# Find the R^2
print('The R-square is: ', lm.score(Z, df['price']))
```

```
The R-square is:  0.8093732522175299
```

In [120]:

```
Y_predict_multifit = lm.predict(Z)
```

In [124]:

```
MSE = mean_squared_error(df['price'], Y_predict_multifit)
MSE
```

Out[124]:

```
11979300.349818882
```

Model 3: Polynomial Fit

In [125]:

```
from sklearn.metrics import r2_score
```

In [126]:

```
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
```

```
The R-square value is:  0.674194666390652
```

In [127]:

```
MSE = mean_squared_error(df['price'], p(x))
MSE
```

Out[127]:

```
20474146.426361218
```

Decision Making: Determining a Good Model Fit

Let's take a look at the values for the different models.

Simple Linear Regression:

Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.49659118843391759
- MSE: 3.16 x 10^7

Multiple Linear Regression:

Using Horsepower, Curb-weight, Engine-size, and Highway-mpg as Predictor Variables of Price.

- R-squared: 0.80896354913783497
- MSE: 1.2 x 10^7

Polynomial Fit:

Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.6741946663906514
- MSE: 2.05 x 10^7

Conclusion:

Comparing these three models, we conclude that the MLR model is the best model to be able to predict price from our dataset. This result makes sense, since we have 27 variables in total, and we know that more than one of those variables are potential predictors of the final car price.

Prediction

We have

In [128]:

```
Y_hat = lm.predict(Z)
```

In [134]:

```
Z
```

Out[134]:

	horsepower	curb-weight	engine-size	highway-mpg
0	111	2548	130	27
1	111	2548	130	27
2	154	2823	152	26
3	102	2337	109	30
4	115	2824	136	22
...	...	...	...	...
196	114	2952	141	28
197	160	3049	141	25
198	134	3012	173	23
199	106	3217	145	27
200	114	3062	141	25

201 rows × 4 columns

In [152]:

```
df
```

Out[152]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	w
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	
3	2	164	audi	gas	std	four	sedan	fwd	front	
4	2	164	audi	gas	std	four	sedan	4wd	front	
...	...	...	...	...	...	...	...	...	...	
196	-1	95	volvo	gas	std	four	sedan	rwd	front	
197	-1	95	volvo	gas	turbo	four	sedan	rwd	front	
198	-1	95	volvo	gas	std	four	sedan	rwd	front	
199	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	
200	-1	95	volvo	gas	turbo	four	sedan	rwd	front	

201 rows × 28 columns

In [153]:

```
Y_hat[0:5] #first five prices
```

Out[153]:

```
array([13699.07700462, 13699.07700462, 19052.71346719, 10620.6152440
4,
       15520.90025344])
```

In [161]:

```
Y_hat[198:201] # last three prices
```

Out[161]:

```
array([20475.22145759, 17803.80500849, 17103.9287056 ])
```

In [ ]:

```

```