In [1]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Loading Data

In [3]:
```
#Loading Data into data frame
path = "https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/Cog
cars = pd.read_csv(path,header = None)
```

In [4]:
```
#first five rows of the Data
cars.head()
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...	130	mpfi	3.47	2.68	9.0	111
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...	152	mpfi	2.68	3.47	9.0	154
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...	109	mpfi	3.19	3.40	10.0	102
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...	136	mpfi	3.19	3.40	8.0	115

5 rows × 26 columns

In [5]:
```
#last five rows of the Data
cars.tail()
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	...	16	17	18	19	20	21
200	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	9.5	114
201	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	8.7	16(
202	-1	95	volvo	gas	std	four	sedan	rwd	front	109.1	...	173	mpfi	3.58	2.87	8.8	134
203	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	109.1	...	145	idi	3.01	3.40	23.0	10(
204	-1	95	volvo	gas	turbo	four	sedan	rwd	front	109.1	...	141	mpfi	3.78	3.15	9.5	114

5 rows × 26 columns

Add Headers

Take a look at our dataset; pandas automatically set the header by an integer from 0.

To better describe our data we can introduce a header, this information is available at: [https://archive.ics.uci.edu/ml/datasets/Automobile (https://archive.ics.uci.edu/ml/datasets/Automobile)](https://archive.ics.uci.edu/ml/datasets/Automobile)

Thus, we have to add headers manually.

Firstly, we create a list "headers" that include all column names in order. Then, we use dataframe.columns = headers to replace the headers by the list we created.

In [6]:
```
# create headers list
headers = ["symboling","normalized-losses","make","fuel-type","aspiration", "
         "drive-wheels","engine-location","wheel-base", "length","width","hei
         "num-of-cylinders", "engine-size","fuel-system","bore","stroke","com
         "peak-rpm","city-mpg","highway-mpg","price"]
print("headers\n", headers)
```

```
headers
 ['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration', 'num-of-
doors', 'body-style', 'drive-wheels', 'engine-location', 'wheel-base', 'lengt
h', 'width', 'height', 'curb-weight', 'engine-type', 'num-of-cylinders', 'engin
e-size', 'fuel-system', 'bore', 'stroke', 'compression-ratio', 'horsepower', 'p
eak-rpm', 'city-mpg', 'highway-mpg', 'price']
```

In [7]:
```
#assign headers to columns
cars.columns = headers

```

In [8]:
```
cars.head()
```

Out[8]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...

5 rows × 26 columns

we can drop missing values along the column "price" as follows

In [9]:
```
cars.dropna(subset=["price"], axis=0)
```

Out[9]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	w
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	
2	1	?	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	
5	2	?	audi	gas	std	two	sedan	fwd	front	
6	1	158	audi	gas	std	four	sedan	fwd	front	
7	1	?	audi	gas	std	four	wagon	fwd	front	
8	1	158	audi	gas	turbo	four	sedan	fwd	front	
9	0	?	audi	gas	turbo	two	hatchback	4wd	front	
10	2	192	bmw	gas	std	two	sedan	rwd	front	
11	0	192	bmw	gas	std	four	sedan	rwd	front	
12	0	188	bmw	gas	std	two	sedan	rwd	front	
13	0	188	bmw	gas	std	four	sedan	rwd	front	
14	1	?	bmw	gas	std	four	sedan	rwd	front	
15	0	?	bmw	gas	std	four	sedan	rwd	front	
16	0	?	bmw	gas	std	two	sedan	rwd	front	
17	0	?	bmw	gas	std	four	sedan	rwd	front	
18	2	121	chevrolet	gas	std	two	hatchback	fwd	front	
19	1	98	chevrolet	gas	std	two	hatchback	fwd	front	
20	0	81	chevrolet	gas	std	four	sedan	fwd	front	
21	1	118	dodge	gas	std	two	hatchback	fwd	front	
22	1	118	dodge	gas	std	two	hatchback	fwd	front	
23	1	118	dodge	gas	turbo	two	hatchback	fwd	front	
24	1	148	dodge	gas	std	four	hatchback	fwd	front	
25	1	148	dodge	gas	std	four	sedan	fwd	front	
26	1	148	dodge	gas	std	four	sedan	fwd	front	
27	1	148	dodge	gas	turbo	?	sedan	fwd	front	
28	-1	110	dodge	gas	std	four	wagon	fwd	front	
29	3	145	dodge	gas	turbo	two	hatchback	fwd	front	
...	...	...	...	...	...	...	...	...	...	
175	-1	65	toyota	gas	std	four	hatchback	fwd	front	
176	-1	65	toyota	gas	std	four	sedan	fwd	front	

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	w
177	-1	65	toyota	gas	std	four	hatchback	fwd	front	
178	3	197	toyota	gas	std	two	hatchback	rwd	front	
179	3	197	toyota	gas	std	two	hatchback	rwd	front	
180	-1	90	toyota	gas	std	four	sedan	rwd	front	
181	-1	?	toyota	gas	std	four	wagon	rwd	front	
182	2	122	volkswagen	diesel	std	two	sedan	fwd	front	
183	2	122	volkswagen	gas	std	two	sedan	fwd	front	
184	2	94	volkswagen	diesel	std	four	sedan	fwd	front	
185	2	94	volkswagen	gas	std	four	sedan	fwd	front	
186	2	94	volkswagen	gas	std	four	sedan	fwd	front	
187	2	94	volkswagen	diesel	turbo	four	sedan	fwd	front	
188	2	94	volkswagen	gas	std	four	sedan	fwd	front	
189	3	?	volkswagen	gas	std	two	convertible	fwd	front	
190	3	256	volkswagen	gas	std	two	hatchback	fwd	front	
191	0	?	volkswagen	gas	std	four	sedan	fwd	front	
192	0	?	volkswagen	diesel	turbo	four	sedan	fwd	front	
193	0	?	volkswagen	gas	std	four	wagon	fwd	front	
194	-2	103	volvo	gas	std	four	sedan	rwd	front	
195	-1	74	volvo	gas	std	four	wagon	rwd	front	
196	-2	103	volvo	gas	std	four	sedan	rwd	front	
197	-1	74	volvo	gas	std	four	wagon	rwd	front	
198	-2	103	volvo	gas	turbo	four	sedan	rwd	front	
199	-1	74	volvo	gas	turbo	four	wagon	rwd	front	
200	-1	95	volvo	gas	std	four	sedan	rwd	front	
201	-1	95	volvo	gas	turbo	four	sedan	rwd	front	
202	-1	95	volvo	gas	std	four	sedan	rwd	front	
203	-1	95	volvo	diesel	turbo	four	sedan	rwd	front	
204	-1	95	volvo	gas	turbo	four	sedan	rwd	front	

205 rows × 26 columns

Now, we have successfully read the raw dataset and add the correct headers into the data frame.

In [10]:
```
cars.columns
```

Out[10]:
```
Index(['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-size', 'fuel-system', 'bore', 'stroke',
       'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg',
       'highway-mpg', 'price'],
      dtype='object')
```

In [11]:
```
#to save the file in local directory with suitable columns.
cars.to_csv("cars.csv",index=False)
```

In [12]:
```
cars.shape
```

Out[12]:
```
(205, 26)
```

Data Wrangling

In [13]:
```
#check the first five rows of data:
cars.head()
```

Out[13]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	.
0	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
1	3	?	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
2	1	?	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	.

5 rows × 26 columns

As we can see, several question marks appeared in the dataframe; those are missing values which may hinder our further analysis.
So, how do we identify all those missing values and deal with them?

How to work with missing data?

Steps for working with missing data:

1. dentify missing data
2. deal with missing data
3. correct data format

Identify and handle missing values

Identify missing values

Convert "?" to NaN
In the car dataset, missing data comes with the question mark "?". We replace "?" with NaN (Not a Number), which is Python's default missing value marker, for reasons of computational speed and convenience. Here we use the function:

```
.replace(A, B, inplace = True)
```

to replace A by B

In [14]:
```
# replace "?" to NaN
cars.replace("?", np.nan, inplace = True)
cars.head(5)
```

Out[14]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	.
0	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
1	3	NaN	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
2	1	NaN	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	.

5 rows × 26 columns

In [15]:
```
missing_data = cars.isnull()
missing_data.head(5)
```

Out[15]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...	er
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

Using a for loop in Python, we can quickly figure out the number of missing values in each column. As mentioned above, "True" represents a missing value, "False" means the value is present in the dataset. In the body of the for loop the method ".value_counts()" counts the number of "True" values.

In [16]:
```
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
```

Based on the summary above, each column has 205 rows of data, seven columns containing missing data:

1. "normalized-losses": 41 missing data
2. "num-of-doors": 2 missing data
3. "bore": 4 missing data
4. "stroke" : 4 missing data
5. "horsepower": 2 missing data
6. "peak-rpm": 2 missing data
7. "price": 4 missing data

Deal with missing data

How to deal with missing data?

1. drop data
   a. drop the whole row
   b. drop the whole column
2. replace data
   a. replace it by mean

b. replace it by frequency
c. replace it based on other functions

Whole columns should be dropped only if most entries in the column are empty. In our dataset, none of the columns are empty enough to drop entirely. We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. We will apply each method to many different columns:

Replace by mean:

- "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 frequency:

- "num-of-doors": 2 missing data, replace them with "four".
  - Reason: 84% sedans is four doors. Since four doors is most frequent, it is most likely to occur

Drop the whole row:

- "price": 4 missing data, simply delete the whole row
  - Reason: price is what we want to predict. Any data entry without price data cannot be used for prediction; therefore any row now without price data is not useful to us

Calculate the average of the column "normalized-lossed"

In [17]:
```
avg_norm_loss = cars["normalized-losses"].astype("float").mean(axis=0)
print("Average of normalized-losses:", avg_norm_loss)
```

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

Replace "NaN" by mean value in "normalized-losses" column

In [18]:
```
cars["normalized-losses"].replace(np.nan, avg_norm_loss, inplace=True)
```

In [19]:
```
cars["normalized-losses"].head()
```

Out[19]:
```
0    122
1    122
2    122
3    164
4    164
Name: normalized-losses, dtype: object
```

Now the missing values in the column "normalized-losses" has been replaced by the mean

Now the missing values in the column "normalized-losses" has been replaced by the mean of the column.

Dealing with the missing value in "num-of-doors"

In [20]:
```
cars["num-of-doors"].value_counts()

```

Out[20]:
```
four    114
two      89
Name: num-of-doors, dtype: int64
```

In [21]:
```
#maximum freq. in num-of-doors
cars["num-of-doors"].value_counts().idxmax()
```

Out[21]:
```
'four'
```

It is clear that four is the most frequent num-of-doors in cars so replace the missing value with the four in the "num-of-doors" column

In [22]:
```
cars["num-of-doors"].replace(np.nan,"four",inplace=True)
```

In [23]:
```
cars["num-of-doors"].value_counts()
```

Out[23]:
```
four    116
two      89
Name: num-of-doors, dtype: int64
```

Dealing with the missing value in "bore"

In [24]:
```
cars["bore"].head()
```

Out[24]:
```
0    3.47
1    3.47
2    2.68
3    3.19
4    3.19
Name: bore, dtype: object
```

Calculate the mean value for 'bore' column

In [25]:
```
avg_bore=cars['bore'].astype('float').mean(axis=0)
print("Average of bore:", avg_bore)
```

```
Average of bore: 3.3297512437810957
```

Replace NaN by mean value

In [26]:
```
cars["bore"].replace(np.nan, avg_bore, inplace=True)
```

Dealing with the missing value in "stroke"

In [27]:
```
cars["stroke"].head()
```

Out[27]:
```
0    2.68
1    2.68
2    3.47
3    3.40
4    3.40
Name: stroke, dtype: object
```

Calculate the mean value for 'bore' column

In [28]:
```
avg_stroke=cars['stroke'].astype('float').mean(axis=0)
print("Average of stroke:", avg_stroke)
```

```
Average of stroke: 3.2554228855721337
```

Replace NaN by mean value

In [29]:
```
cars["stroke"].replace(np.nan, avg_bore, inplace=True)
```

Dealing with the missing value in "horsepower"

In [30]:
```
cars["horsepower"].head()
```

Out[30]:
```
0    111
1    111
2    154
3    102
4    115
Name: horsepower, dtype: object
```

Calculate the mean value for 'horsepower' column

In [31]:
```
avg_horsepower=cars['horsepower'].astype('float').mean(axis=0)
print("Average of horsepower:", avg_horsepower)
```

```
Average of horsepower: 104.25615763546799
```

Replace NaN by mean value

In [32]:
```
cars["horsepower"].replace(np.nan,avg_horsepower,inplace=True)
```

Dealing with the missing value in "peak-rpm"

In [33]:
```
cars["peak-rpm"].head()
```

Out[33]:
```
0    5000
1    5000
2    5000
3    5500
4    5500
Name: peak-rpm, dtype: object
```

Calculate the mean value for 'peak-rpm' column

In [34]:
```
avg_peak_rpm = cars["peak-rpm"].astype('float').mean(axis=0)
print("Average of peak-rpm:", avg_peak_rpm)

```

```
Average of peak-rpm: 5125.369458128079
```

Replace NaN by mean value

In [35]:
```
cars["peak-rpm"].replace(np.nan,avg_peak_rpm,inplace=True)
```

Dealing with the missing value in "price"

In [36]:
```
cars["price"].head()
```

Out[36]:
```
0    13495
1    16500
2    16500
3    13950
4    17450
Name: price, dtype: object
```

Since the price is what we want to predict so any empty entry is of no use. Hence, Drop the entire row which contain null value.

In [37]:
```
# simply drop whole row with NaN in "price" column
cars.dropna(subset=["price"], axis=0, inplace=True)

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

In [47]:
```
cars.head()
```

Out[47]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	.
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	.

5 rows × 26 columns

Now, we obtain the dataset with no missing values.

Correct data format

In Pandas, we use

.dtype() to check the data type

.astype() to change the data type

Lets list the data types for each column

In [38]:
```
cars.dtypes
```

Out[38]:
```
symboling              int64
normalized-losses     object
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                  object
stroke                object
compression-ratio    float64
horsepower            object
peak-rpm              object
city-mpg               int64
highway-mpg            int64
price                 object
dtype: object
```

As we can see above, some columns are not of the correct data type. Numerical variables should have type 'float' or 'int', and variables with strings such as categories should have type 'object'. For example, '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.

Convert data types to proper format

In [39]:
```
cars[["bore", "stroke"]] = cars[["bore", "stroke"]].astype("float")
cars[["normalized-losses"]] = cars[["normalized-losses"]].astype("int")
cars[["price"]] =cars[["price"]].astype("float")
cars[["peak-rpm"]] = cars[["peak-rpm"]].astype("float")
```

Let us list the columns after the conversion

In [40]:
```
cars.dtypes
```

Out[40]:
```
symboling              int64
normalized-losses      int32
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            object
peak-rpm             float64
city-mpg               int64
highway-mpg            int64
price                float64
dtype: object
```

Now, we finally obtain the cleaned dataset with no missing values and all data in its proper format.

Data Standardization

Data is usually collected from different agencies with different formats. (Data Standardization is also a term for a particular type of data normalization, where we subtract the mean and divide by the standard deviation)

Example

Transform mpg to L/100km:

In our dataset, the fuel consumption columns "city-mpg" and "highway-mpg" are represented by mpg (miles per gallon) unit. Assume we are developing an application in a country that accept the fuel consumption with L/100km standard

We will need to apply **data transformation** to transform mpg into L/100km?

The formula for unit conversion is

L/100km = 235 / mpg

We can do many mathematical operations directly in Pandas.

In [41]:
```
cars.head()
```

Out[41]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...

5 rows × 26 columns

In [42]:
```
# Convert mpg to L/100km by mathematical operation (235 divided by mpg)
cars['city-L/100km'] = 235/cars["city-mpg"]

# check transformed data
cars.head()
```

Out[42]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...

5 rows × 27 columns

In [43]:
```
#now drop the column("city-mpg") as we already created a standard version as
cars.drop(columns=['city-mpg'],inplace=True)
```

In [44]:
```
cars.head()
```

Out[44]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...

5 rows × 26 columns

In [45]:
```
# transform mpg to L/100km by mathematical operation (235 divided by mpg)
cars["highway-mpg"] = 235/cars["highway-mpg"]

# rename column name from "highway-mpg" to "highway-L/100km"
cars.rename(columns={"highway-mpg": "highway-L/100km"},inplace= True)

# check your transformed data
cars.head()

```

Out[45]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...

5 rows × 26 columns

Data Normalization

let's say we want to scale the columns "length", "width" and "height"

Target:would like to Normalize those variables so their value ranges from 0 to 1.

Approach: replace original value by (original value)/(maximum value)

In [47]:
```
cars[["length","width","height"]].head()
```

Out[47]:

	length	width	height
0	168.8	64.1	48.8
1	168.8	64.1	48.8
2	171.2	65.5	52.4
3	176.6	66.2	54.3
4	176.6	66.4	54.3

In [48]:
```
# replace (original value) by (original value)/(maximum value)
cars['length'] = cars['length']/cars['length'].max()
cars['width'] = cars['width']/cars['width'].max()
cars['height'] = cars['height']/cars['height'].max()
# show the scaled columns
cars[["length","width","height"]].head()
```

Out[48]:

	length	width	height
0	0.811148	0.890278	0.816054
1	0.811148	0.890278	0.816054
2	0.822681	0.909722	0.876254
3	0.848630	0.919444	0.908027
4	0.848630	0.922222	0.908027

Binning

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

Convert data to correct format

In [49]:
```
cars["horsepower"]=cars["horsepower"].astype(int, copy=True)
```

In [50]:
```
cars["horsepower"].dtypes
```

Out[50]: dtype('int32')

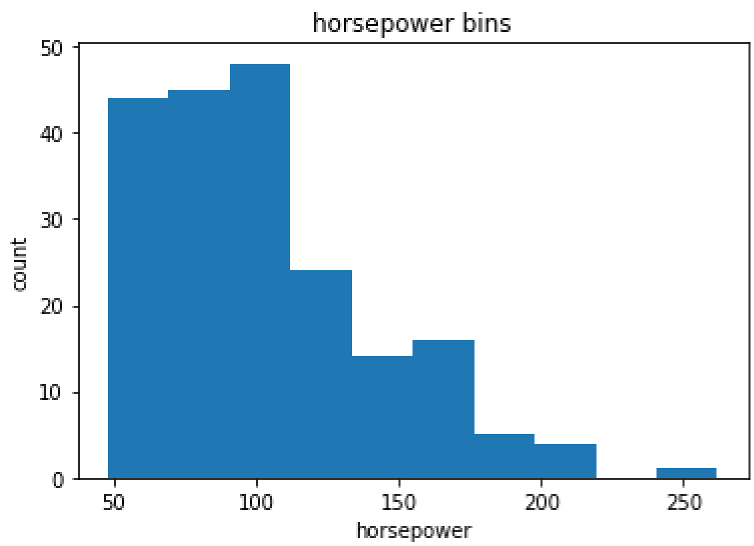
Lets plot the histogram of horspower, to see what the distribution of horsepower looks like.

In [51]:

```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
plt.pyplot.hist(cars["horsepower"])

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

Out[51]:

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



We would like 3 bins of equal size bandwidth so we use numpy's `linspace(start_value, end_value, numbers_generated` function.

Since we want to include the minimum value of horsepower we want to set start_value=min(df["horsepower"]).

Since we want to include the maximum value of horsepower we want to set end_value=max(df["horsepower"]).

Since we are building 3 bins of equal length, there should be 4 dividers, so numbers_generated=4.

We build a bin array, with a minimum value to a maximum value, with bandwidth calculated above. The bins will be values used to determine when one bin ends and another begins.

In [52]:

```
bins = np.linspace(min(cars["horsepower"]), max(cars["horsepower"]), 4)
bins
```

Out[52]:

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

We set group names:

In [53]:
```
group_names = ['Low', 'Medium', 'High']
```

We apply the function "cut" the determine what each value of "df['horsepower']" belongs to.

In [54]:
```
cars['horsepower-binned'] = pd.cut(cars['horsepower'], bins, labels=group_nam
cars[['horsepower','horsepower-binned']].head(20)
```

Out[54]:

	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

Lets see the number of vehicles in each bin.

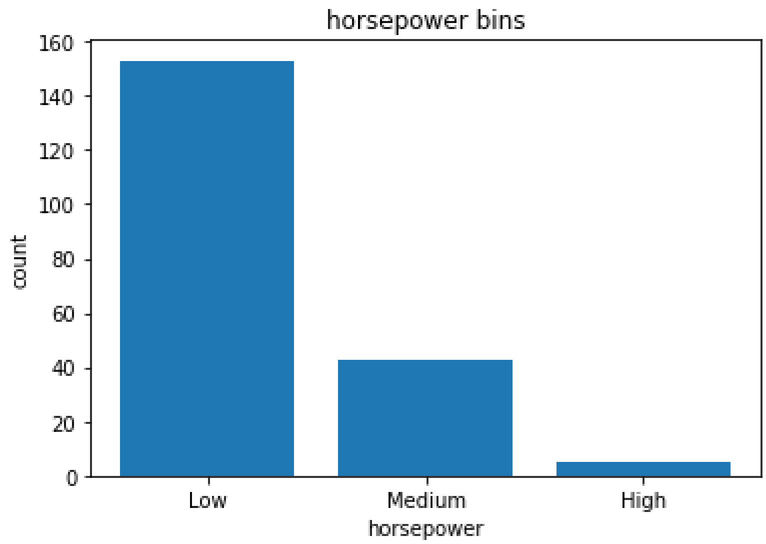
In [55]:
```
cars["horsepower-binned"].value_counts()
```

Out[55]:
```
Low       153
Medium     43
High        5
Name: horsepower-binned, dtype: int64
```

Lets plot the distribution of each bin.

In [56]:
```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot
pyplot.bar(group_names, cars["horsepower-binned"].value_counts())

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

Out[56]: Text(0.5, 1.0, 'horsepower bins')



In [57]:
```
cars.head()
```

Out[57]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	...
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	...
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	...
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	...
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	...

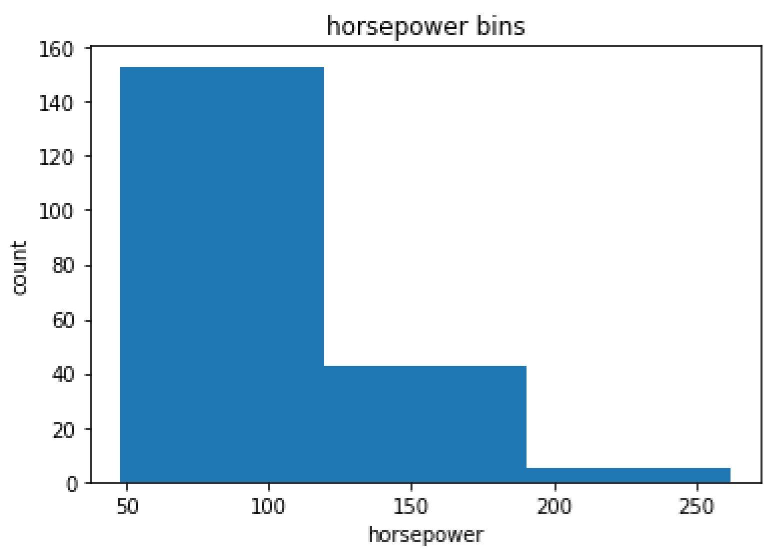
5 rows × 27 columns

Bins visualization

Normally, a histogram is used to visualize the distribution of bins we created above.

In [58]:
```
%matplotlib inline
import matplotlib as plt
from matplotlib import pyplot


# draw historgram of attribute "horsepower" with bins = 3
plt.pyplot.hist(cars["horsepower"], bins = 3)

# set x/y labels and plot title
plt.pyplot.xlabel("horsepower")
plt.pyplot.ylabel("count")
plt.pyplot.title("horsepower bins")
```

Out[58]:
```
Text(0.5, 1.0, 'horsepower bins')
```



The plot above shows the binning result for attribute "horsepower".

Indicator variable (or dummy variable)

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.

We will use the panda's method 'get_dummies' to assign numerical values to different categories of fuel type.

In [59]:
```
cars.columns
```

Out[59]:
```
Index(['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-size', 'fuel-system', 'bore', 'stroke',
       'compression-ratio', 'horsepower', 'peak-rpm', 'highway-L/100km',
       'price', 'city-L/100km', 'horsepower-binned'],
      dtype='object')
```

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

In [60]:
```
dummy_variable_1 = pd.get_dummies(cars["fuel-type"])
dummy_variable_1.head()
```

Out[60]:

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

We now have the value 0 to represent "gas" and 1 to represent "diesel" in the column "fuel-type". We will now insert this column back into our original dataset.

In [61]:
```
cars.head()

```

Out[61]:

	symboling	normalized-losses	make	fuel-type	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	.
0	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
1	3	122	alfa-romero	gas	std	two	convertible	rwd	front	88.6	.
2	1	122	alfa-romero	gas	std	two	hatchback	rwd	front	94.5	.
3	2	164	audi	gas	std	four	sedan	fwd	front	99.8	.
4	2	164	audi	gas	std	four	sedan	4wd	front	99.4	.

5 rows × 27 columns

In [62]:
```
# merge data frame "cars" and "dummy_variable_1"
cars = pd.concat([cars, dummy_variable_1], axis=1)

# drop original column "fuel-type" from "cars"
cars.drop("fuel-type", axis = 1, inplace=True)
```

In [63]:
```
cars.head()
```

Out[63]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	lengt
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.81114
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.81114
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.82268
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.84863
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.84863

5 rows × 28 columns

The last two columns are now the indicator variable representation of the fuel-type variable. It's all 0s and 1s now.

In [64]:
```
cars.rename(columns={'gas':'fuel-type-gas', 'diesel':'fuel-type-diesel'}, inp
cars.head()
```

Out[64]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	lengt
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.81114
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.81114
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.82268
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.84863
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.84863

5 rows × 28 columns

In [65]:

```
# get indicator variables of aspiration and assign it to data frame "dummy_va
dummy_variable_2 = pd.get_dummies(cars['aspiration'])

# change column names for clarity
dummy_variable_2.rename(columns={'std':'aspiration-std', 'turbo': 'aspiration

# show first 5 instances of data frame "dummy_variable_1"
dummy_variable_2.head()
```

Out[65]:

	aspiration-std	aspiration-turbo
0	1	0
1	1	0
2	1	0
3	1	0
4	1	0

In [66]:

```

#merge the new dataframe to the original datafram
cars = pd.concat([cars, dummy_variable_2], axis=1)

cars.head()
```

Out[66]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	lengt
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.81114
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.81114
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.82268
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.84863
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.84863

5 rows × 30 columns

In [67]:
```
# drop original column "aspiration" from "df"
cars.drop('aspiration', axis = 1, inplace=True)
cars.head()
```

Out[67]:

	symboling	normalized-losses	make	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	width
0	3	122	alfa-romero	two	convertible	rwd	front	88.6	0.811148	0.890278
1	3	122	alfa-romero	two	convertible	rwd	front	88.6	0.811148	0.890278
2	1	122	alfa-romero	two	hatchback	rwd	front	94.5	0.822681	0.909722
3	2	164	audi	four	sedan	fwd	front	99.8	0.848630	0.919444
4	2	164	audi	four	sedan	4wd	front	99.4	0.848630	0.922222

5 rows × 29 columns

save the new csv

In [68]:
```
cars.to_csv('clean_cars.csv')
```

Exploratory-Data-Analysis

Analyzing Individual Feature Patterns using Visualization

To install seaborn we use the pip which is the python package manager.

In [69]:
```
%%capture
! pip install seaborn
```

In [70]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Finding Correlation

In [71]:

```
cars.corr()
```

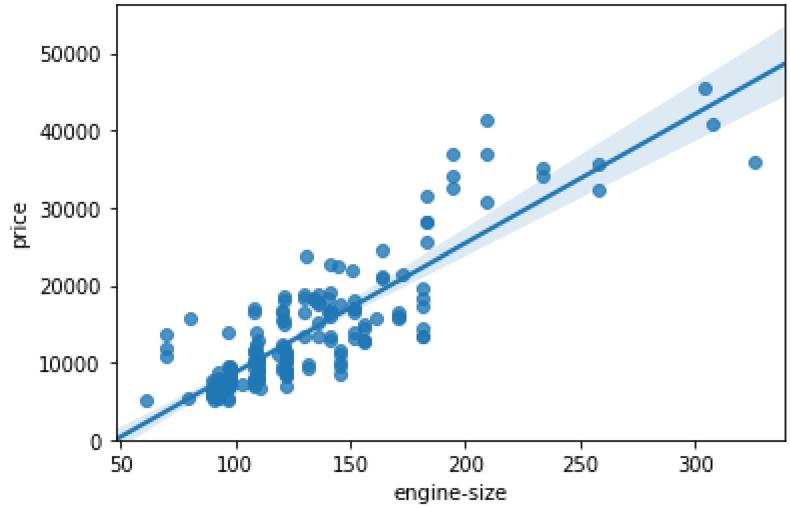
Out[71]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engi
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-0.110
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	0.112
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	0.572
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	0.685
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	0.729
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	0.074
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	0.849
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	1.000
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	0.572
stroke	-0.000059	0.059131	0.155225	0.121904	0.188301	-0.068633	0.166038	0.199
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	0.028
horsepower	0.075810	0.217300	0.371178	0.579795	0.615056	-0.087001	0.757981	0.822
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-0.256
highway-L/100km	-0.029807	0.181189	0.577576	0.707108	0.736728	0.084301	0.836921	0.783
price	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	0.872
city-L/100km	0.066171	0.238567	0.476153	0.657373	0.673363	0.003811	0.785353	0.745
fuel-type-diesel	-0.196735	-0.101546	0.307237	0.211187	0.244356	0.281578	0.221046	0.070
fuel-type-gas	0.196735	0.101546	-0.307237	-0.211187	-0.244356	-0.281578	-0.221046	-0.070
aspiration-std	0.054615	0.006911	-0.256889	-0.230085	-0.305732	-0.090336	-0.321955	-0.110
aspiration-turbo	-0.054615	-0.006911	0.256889	0.230085	0.305732	0.090336	0.321955	0.110

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

In [72]:
```
# Engine size as potential predictor variable of price
sns.regplot(x="engine-size", y="price", data=cars)
plt.ylim(0,)
```

Out[72]: (0, 56269.964617255086)



As the engine-size goes up, the price goes up: this indicates a positive direct correlation between these two variables. Engine size seems like a pretty good predictor of price since the regression line is almost a perfect diagonal line.

We can examine the correlation between 'engine-size' and 'price'

In [73]:
```
cars[["engine-size", "price"]].corr()
```

Out[73]:

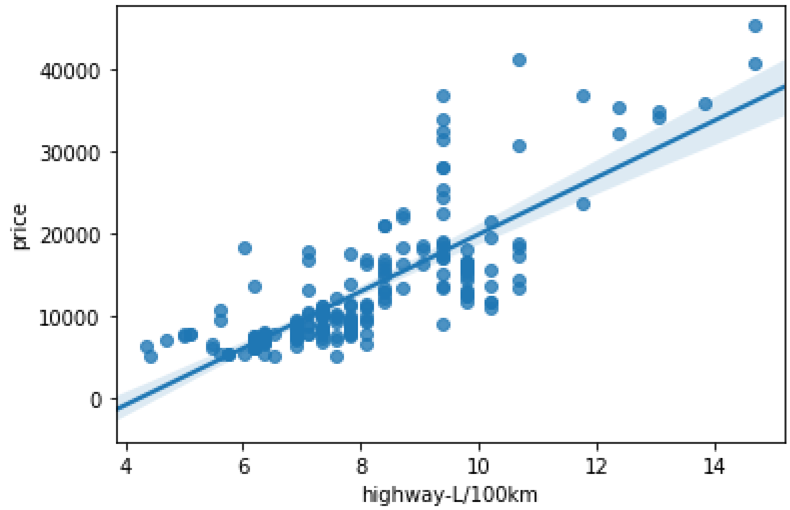
	engine-size	price
engine-size	1.000000	0.872335
price	0.872335	1.000000

it's approximately 0.87

Highway mpg is a potential predictor variable of price

In [74]:
```
sns.regplot(x="highway-L/100km", y="price", data=cars)
```

Out[74]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2172b8f3dd8>
```



As the highway-L/100km goes up, the price goes up: this indicates an positive relationship between these two variables. highway-L/100km could potentially be a predictor of price.

We can examine the correlation between 'highway-mpg' and 'price'

In [75]:
```
cars[['highway-L/100km', 'price']].corr()
```

Out[75]:

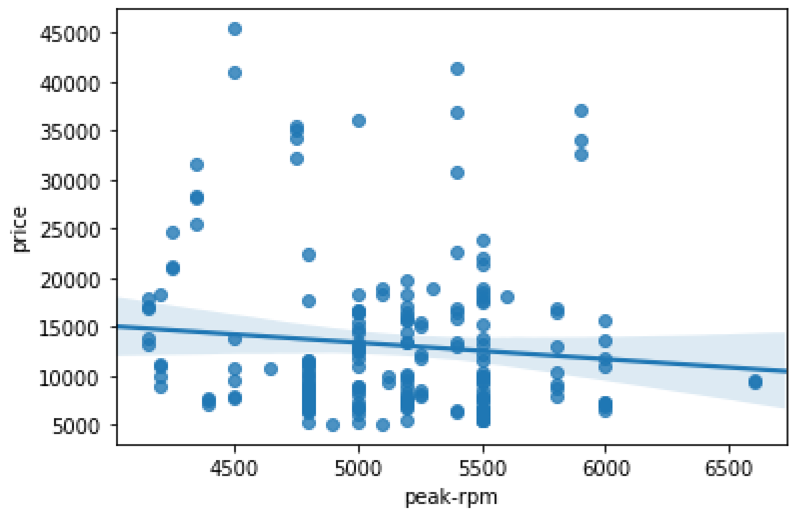
	highway-L/100km	price
highway-L/100km	1.000000	0.801118
price	0.801118	1.000000

it's approximately 0.801118

Let's see if "Peak-rpm" as a predictor variable of "price".

In [76]:
```
sns.regplot(x="peak-rpm", y="price", data=cars)
```

Out[76]: <matplotlib.axes._subplots.AxesSubplot at 0x2172b9601d0>



Peak rpm does not seem like a good predictor of the price at all since the regression line is close to horizontal. Also, the data points are very scattered and far from the fitted line, showing lots of variability. Therefore it's it is not a reliable variable.

We can examine the correlation between 'peak-rpm' and 'price'

In [77]:
```
cars[['peak-rpm','price']].corr()
```

Out[77]:

	peak-rpm	price
peak-rpm	1.000000	-0.101616
price	-0.101616	1.000000

it's approximately -0.101616

Let's see if "stroke" as a predictor variable of "price".

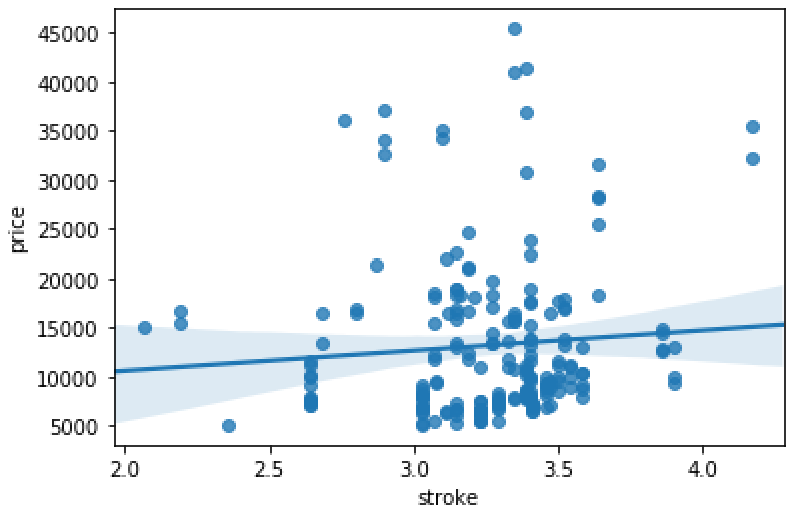
In [78]:
```
cars[['stroke','price']].corr()
```

Out[78]:

	stroke	price
stroke	1.000000	0.082116
price	0.082116	1.000000

In [79]:
```
sns.regplot(x='stroke',y='price',data=cars)
```

Out[79]: <matplotlib.axes._subplots.AxesSubplot at 0x2172b9ce2b0>



There is a weak correlation between the variable 'stroke' and 'price.' as such regression will not work well.

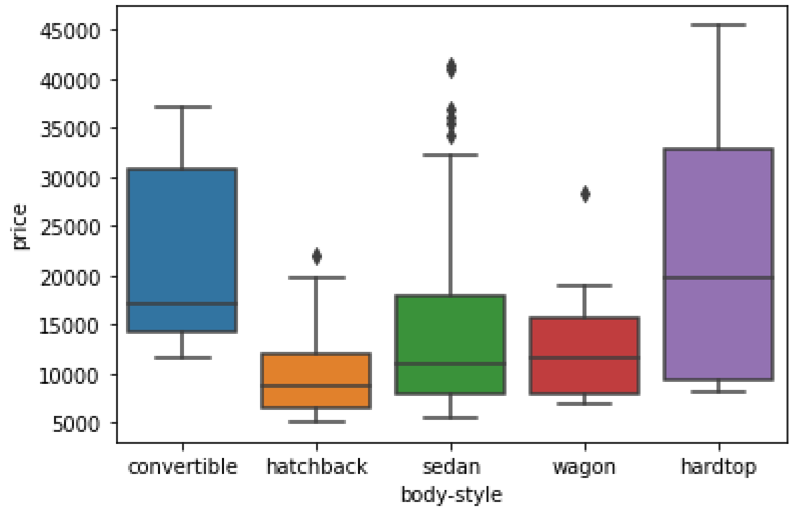
Categorical variables

These are variables that describe a 'characteristic' of a data unit, and are selected from a small group of categories. 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 [80]:
```
sns.boxplot(x="body-style", y="price", data=cars)
```

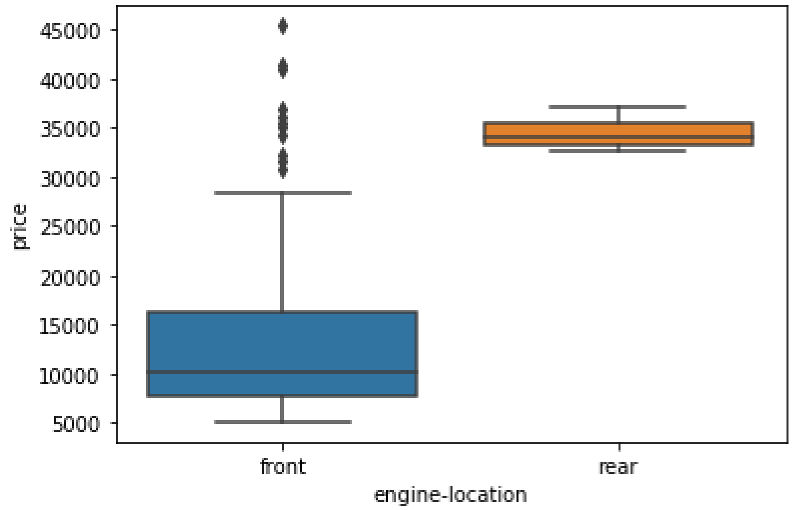
Out[80]: <matplotlib.axes._subplots.AxesSubplot at 0x2172ba110f0>



We see that the distributions of price between the different body-style categories have a significant overlap, and so body-style would not be a good predictor of price. Let's examine engine "engine-location" and "price":

In [81]:
```
sns.boxplot(x="engine-location", y="price", data=cars)
```

Out[81]: <matplotlib.axes._subplots.AxesSubplot at 0x2172ba91080>

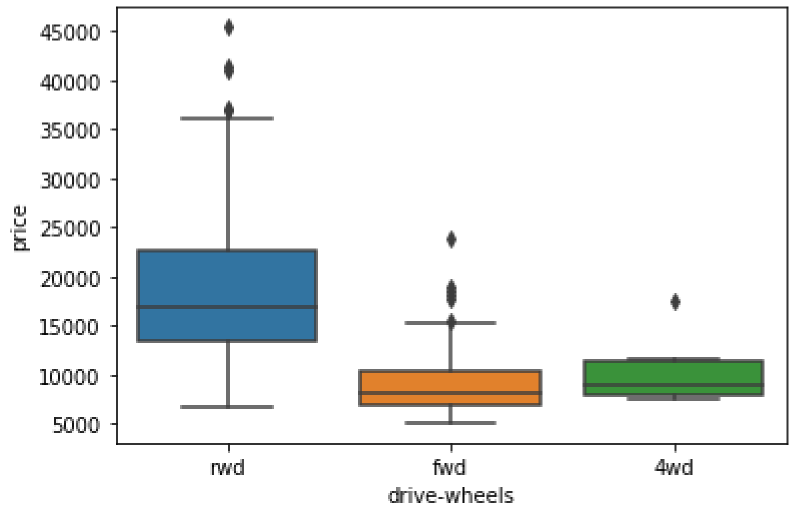


Here we see that the distribution of price between these two engine-location categories, front and rear, are distinct enough to take engine-location as a potential good predictor of price.

Let's examine "drive-wheels" and "price".

In [82]:
```
# drive-wheels
sns.boxplot(x="drive-wheels", y="price", data=cars)
```

Out[82]: <matplotlib.axes._subplots.AxesSubplot at 0x2172bb0bda0>



Here we see that the distribution of price between the different drive-wheels categories differs; as such drive-wheels could potentially be a predictor of price.

Descriptive Statistical Analysis

Let's first take a look at the variables by utilizing a description method.

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

This will show:

- the count of that variable
- the mean
- the standard deviation (std)
- the minimum value
- the IQR (Interquartile Range: 25%, 50% and 75%)
- the maximum value

In [83]:
```
cars.describe()
```

Out[83]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	e
count	201.000000	201.00000	201.000000	201.000000	201.000000	201.000000	201.000000	201.(
mean	0.840796	122.00000	98.797015	0.837102	0.915126	0.899108	2555.666667	126.8
std	1.254802	31.99625	6.066366	0.059213	0.029187	0.040933	517.296727	41.5
min	-2.000000	65.00000	86.600000	0.678039	0.837500	0.799331	1488.000000	61.(
25%	0.000000	101.00000	94.500000	0.801538	0.890278	0.869565	2169.000000	98.(
50%	1.000000	122.00000	97.000000	0.832292	0.909722	0.904682	2414.000000	120.(
75%	2.000000	137.00000	102.400000	0.881788	0.925000	0.928094	2926.000000	141.(
max	3.000000	256.00000	120.900000	1.000000	1.000000	1.000000	4066.000000	326.(

The default setting of "describe" skips variables of type object. We can apply the method "describe" on the variables of type 'object' as follows:

In [84]:
```
cars.describe(include=['object'])
```

Out[84]:

	make	num-of-doors	body-style	drive-wheels	engine-location	engine-type	num-of-cylinders	fuel-system
count	201	201	201	201	201	201	201	201
unique	22	2	5	3	2	6	7	8
top	toyota	four	sedan	fwd	front	ohc	four	mpfi
freq	32	115	94	118	198	145	157	92

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'. Don't forget the method "value_counts" only works on Pandas series, not Pandas Dataframes. As a result, we only include one bracket "df['drive-wheels']" not two brackets "df[['drive-wheels']]".

In [85]:
```
cars['drive-wheels'].value_counts()
```

Out[85]:
```
fwd    118
rwd     75
4wd      8
Name: drive-wheels, dtype: int64
```

We can convert the series to a Dataframe as follows :

In [86]:
```
cars['drive-wheels'].value_counts().to_frame()
```

Out[86]:

	drive-wheels
fwd	118
rwd	75
4wd	8

Let's repeat the above steps but save the results to the dataframe "drive_wheels_counts" and rename the column 'drive-wheels' to 'value_counts'.

In [87]:
```
drive_wheels_counts = cars['drive-wheels'].value_counts().to_frame()
drive_wheels_counts.rename(columns={'drive-wheels': 'value_counts'}, inplace=
drive_wheels_counts
```

Out[87]:

	value_counts
fwd	118
rwd	75
4wd	8

Now let's rename the index to 'drive-wheels':

In [88]:
```
drive_wheels_counts.index.name = 'drive-wheels'
drive_wheels_counts
```

Out[88]:

	value_counts
drive-wheels	
fwd	118
rwd	75
4wd	8

We can repeat the above process for the variable 'engine-location'.

In [89]:
```
# engine-location as variable
engine_loc_counts = cars['engine-location'].value_counts().to_frame()
engine_loc_counts.rename(columns={'engine-location': 'value_counts'}, inplace
engine_loc_counts.index.name = 'engine-location'
engine_loc_counts.head(10)
```

Out[89]:

	value_counts
engine-location	
front	198
rear	3

Examining the value counts of the engine location would not be a good predictor variable for the price. This is because we only have three cars with a rear engine and 198 with an engine in the front, this result is skewed. Thus, we are not able to draw any conclusions about the engine location.

Basics of 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.

For example, let's group by the variable "drive-wheels". We see that there are 3 different categories of drive wheels.

In [90]:
```
cars['drive-wheels'].unique()
```

Out[90]:
```
array(['rwd', 'fwd', '4wd'], dtype=object)
```

If we want to know, on average, which type of drive wheel is most valuable, we can group "drive-wheels" and then average them.

We can select the columns 'drive-wheels', 'body-style' and 'price', then assign it to the variable "cars_group_one".

In [91]:
```
cars_group_one = cars[['drive-wheels','body-style','price']]
```

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

In [92]:
```
# grouping results
cars_group_one =cars_group_one.groupby(['drive-wheels'],as_index=False).mean(
cars_group_one
```

Out[92]:

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

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.

let's group by both 'drive-wheels' and 'body-style'. This groups the dataframe by the unique combinations 'drive-wheels' and 'body-style'. We can store the results in the variable 'grouped_test1'.

In [93]:
```
# grouping results
df_gptest = cars[['drive-wheels','body-style','price']]
grouped_test1 = df_gptest.groupby(['drive-wheels','body-style'],as_index=Fals
grouped_test1
```

Out[93]:

	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

This grouped data is much easier to visualize when it is made into a pivot table. A pivot table is like an Excel spreadsheet, with one variable along the column and another along the row. We can convert the dataframe to a pivot table using the method "pivot " to create a pivot table from the groups.

In this case, we will leave the drive-wheel variable as the rows of the table, and pivot body-style to become the columns of the table:

In [94]:
```
grouped_pivot = grouped_test1.pivot(index='drive-wheels',columns='body-style'
grouped_pivot
```

Out[94]:

					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

Often, we won't have data for some of the pivot cells. We can fill these missing cells with the value 0, but any other value could potentially be used as well.

In [95]:
```
grouped_pivot = grouped_pivot.fillna(0) #fill missing values with 0
grouped_pivot
```

Out[95]:

	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 [96]:
```
df_gptest2 = cars[['body-style','price']]
grouped_test_bodystyle = df_gptest2.groupby(['body-style'],as_index= False).m
grouped_test_bodystyle
```

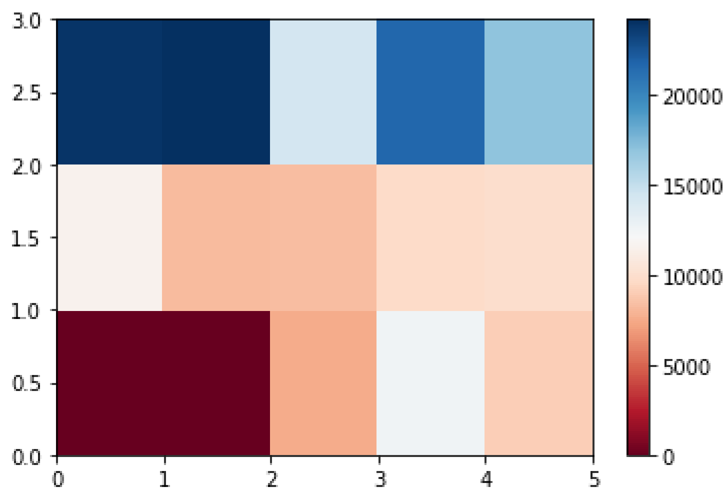
Out[96]:

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

Variables: Drive Wheels and Body Style vs Price

Let's use a heat map to visualize the relationship between Body Style vs Price.

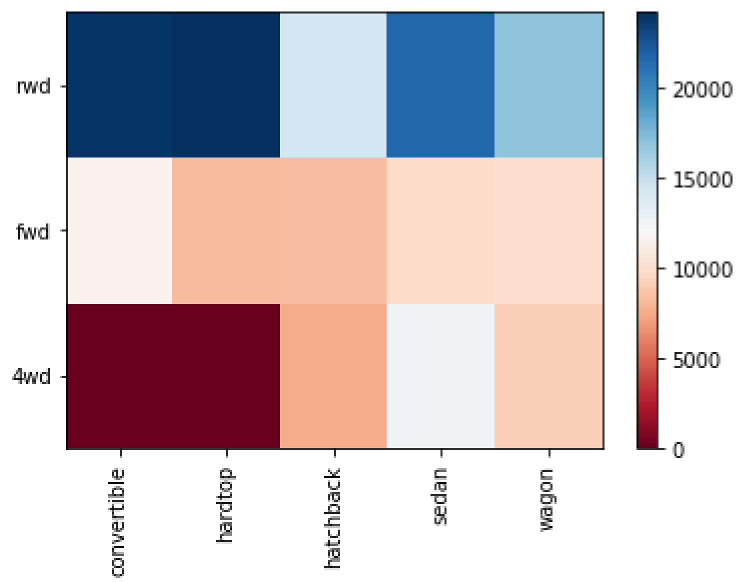
In [97]:
```
#use the grouped results
plt.pcolor(grouped_pivot, cmap='RdBu')
plt.colorbar()
plt.show()
```



The heatmap plots the target variable (price) proportional to colour with respect to the variables 'drive-wheel' and 'body-style' in the vertical and horizontal axis respectively. This allows us to 'drive-wheel' and 'body-style' in the vertical and horizontal axis respectively. This allows us to visualize how the price is related to 'drive-wheel' and 'body-style'.

The default labels convey no useful information to us. Let's change that:

In [98]:
```python
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()
```



Correlation and Causation

Correlation: a measure of the extent of interdependence between variables.

Causation: the relationship between cause and effect between two variables.

It is important to know the difference between these two and that correlation does not imply causation. Determining correlation is much simpler the determining causation as causation may require independent experimentation.

Pearson Correlation

The Pearson Correlation measures the linear dependence between two variables X and Y.

The resulting coefficient is a value between -1 and 1 inclusive, where:

- **1**: Total positive linear correlation.
- **0**: No linear correlation, the two variables most likely do not affect each other.
- **-1**: Total negative linear correlation.

Pearson Correlation is the default method of the function "corr". Like before we can calculate the Pearson Correlation of the of the 'int64' or 'float64' variables.

In [99]:
```
cars.corr()
```

Out[99]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engi s
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-0.110
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	0.112
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	0.572
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	0.685
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	0.729
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	0.074
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	0.849
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	1.000
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	0.572
stroke	-0.000059	0.059131	0.155225	0.121904	0.188301	-0.068633	0.166038	0.199
compression-ratio	-0.182196	-0.114713	0.250313	0.159733	0.189867	0.259737	0.156433	0.028
horsepower	0.075810	0.217300	0.371178	0.579795	0.615056	-0.087001	0.757981	0.822
peak-rpm	0.279740	0.239543	-0.360305	-0.285970	-0.245800	-0.309974	-0.279361	-0.256
highway-L/100km	-0.029807	0.181189	0.577576	0.707108	0.736728	0.084301	0.836921	0.783
price	-0.082391	0.133999	0.584642	0.690628	0.751265	0.135486	0.834415	0.872
city-L/100km	0.066171	0.238567	0.476153	0.657373	0.673363	0.003811	0.785353	0.745
fuel-type-diesel	-0.196735	-0.101546	0.307237	0.211187	0.244356	0.281578	0.221046	0.070
fuel-type-gas	0.196735	0.101546	-0.307237	-0.211187	-0.244356	-0.281578	-0.221046	-0.070
aspiration-std	0.054615	0.006911	-0.256889	-0.230085	-0.305732	-0.090336	-0.321955	-0.110
aspiration-turbo	-0.054615	-0.006911	0.256889	0.230085	0.305732	0.090336	0.321955	0.110

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

P-value:

What is this P-value? The P-value is the probability value that the correlation between these two variables is statistically significant. Normally, we choose a significance level of 0.05, which means that we are 95% confident that the correlation between the variables is significant.

By convention, when the

- p-value is $<$ 0.001: we say there is strong evidence that the correlation is significant.
- the p-value is $<$ 0.05: there is moderate evidence that the correlation is significant.
- the p-value is $<$ 0.1: there is weak evidence that the correlation is significant.
- the 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 [100]:
```
from scipy import stats
```

Wheel-base vs Price

Let's calculate the Pearson Correlation Coefficient and P-value of 'wheel-base' and 'price'.

In [101]:
```
pearson_coef, p_value = stats.pearsonr(cars['wheel-base'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.5846418222655081  with a P-value of P
= 8.076488270732955e-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

Let's calculate the Pearson Correlation Coefficient and P-value of 'horsepower' and 'price'.

In [102]:
```
pearson_coef, p_value = stats.pearsonr(cars['horsepower'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.8096068016571052  with a P-value of P
=  6.273536270651004e-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

Let's calculate the Pearson Correlation Coefficient and P-value of 'length' and 'price'.

In [103]:
```
pearson_coef, p_value = stats.pearsonr(cars['length'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.6906283804483642  with a P-value of P
=  8.016477466158713e-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

Let's calculate the Pearson Correlation Coefficient and P-value of 'width' and 'price':

In [104]:
```
pearson_coef, p_value = stats.pearsonr(cars['width'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.7512653440522673  with a P-value of P
= 9.20033551048166e-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

Let's calculate the Pearson Correlation Coefficient and P-value of 'curb-weight' and 'price':

In [105]:
```
pearson_coef, p_value = stats.pearsonr(cars['curb-weight'], cars['price'])
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-val
```

```
The Pearson Correlation Coefficient is 0.8344145257702846  with a P-value of P
=  2.1895772388936997e-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

Let's calculate the Pearson Correlation Coefficient and P-value of 'engine-size' and 'price':

In [106]:
```
pearson_coef, p_value = stats.pearsonr(cars['engine-size'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.8723351674455185  with a P-value of P
= 9.265491622197996e-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

Let's calculate the Pearson Correlation Coefficient and P-value of 'bore' and 'price':

In [107]:
```
pearson_coef, p_value = stats.pearsonr(cars['bore'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.5431553832626602  with a P-value of P
=   8.0491894839353643e-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-L/100km vs Price

In [108]:
```
pearson_coef, p_value = stats.pearsonr(cars['city-L/100km'], cars['price'])
print("The Pearson Correlation Coefficient is", pearson_coef, " with a P-valu
```

```
The Pearson Correlation Coefficient is 0.7898975136626942  with a P-value of P
=  3.903106400939940 5e-44
```

Conclusion:

Since the p-value is $<$ 0.001, the correlation between city-L/100km and price is statistically significant, and the coefficient of ~ 0.789 shows that the relationship is moderately strong.

Highway-L/100km vs Price

In [109]:
```
pearson_coef, p_value = stats.pearsonr(cars['highway-L/100km'], cars['price']
print( "The Pearson Correlation Coefficient is", pearson_coef, " with a P-val
```

```
The Pearson Correlation Coefficient is 0.8011176263981975  with a P-value of P
=  3.0467845810412534e-46
```

Conclusion:

Since the p-value is < 0.001, the correlation between highway-L/100km and price is statistically significant, and the coefficient of ~ 0.801 shows that the relationship is moderately strong.

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-L/100km
- Highway-L/100km
- Wheel-base
- Bore

Categorical variables:

- Drive-wheels

As we now move into building machine learning models to automate our analysis, feeding the model with variables that meaningfully affect our target variable will improve our model's prediction performance.

Model Development

we will develop several models that will predict the price of the car using the variables or features. This is just an estimate but should give us an objective idea of how much the car should cost.

Linear Regression

Lets load the modules for linear regression

In [183]:
```
from sklearn.linear_model import LinearRegression
```

Create the linear regression object

In [184]:
```
lm = LinearRegression()
lm
```

Out[184]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

How could Highway-L/100km help us predict car price?

we want to look at how highway-mpg can help us predict car price. Using simple linear regression, we will create a linear function with "highway-mpg" as the predictor variable and the "price" as the response variable.

In [185]:
```
X = cars[['highway-L/100km']]
Y = cars['price']
```

Fit the linear model using highway-L/100km.

In [186]:
```
lm.fit(X,Y)
```

Out[186]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

We can output a prediction

In [114]:
```
Yhat=lm.predict(X)
Yhat[0:5]
```

Out[114]:
```
array([15485.52737455, 15485.52737455, 16643.34931414, 12475.19033163,
       22327.2024721 ])
```

In [187]:
```
lm.score(X,Y)
```

Out[187]:
```
0.6417894513258818
```

What is the value of the intercept (a)?

In [115]:
```
lm.intercept_
```

Out[115]:
```
-14617.843054664598
```

What is the value of the Slope (b)?

In [116]:
```
lm.coef_
```

Out[116]:
```
array([3458.68511314])
```

What is the final estimated linear model we get?

As we saw above, we should get a final linear model with the structure:

$$Yhat = a + bX$$

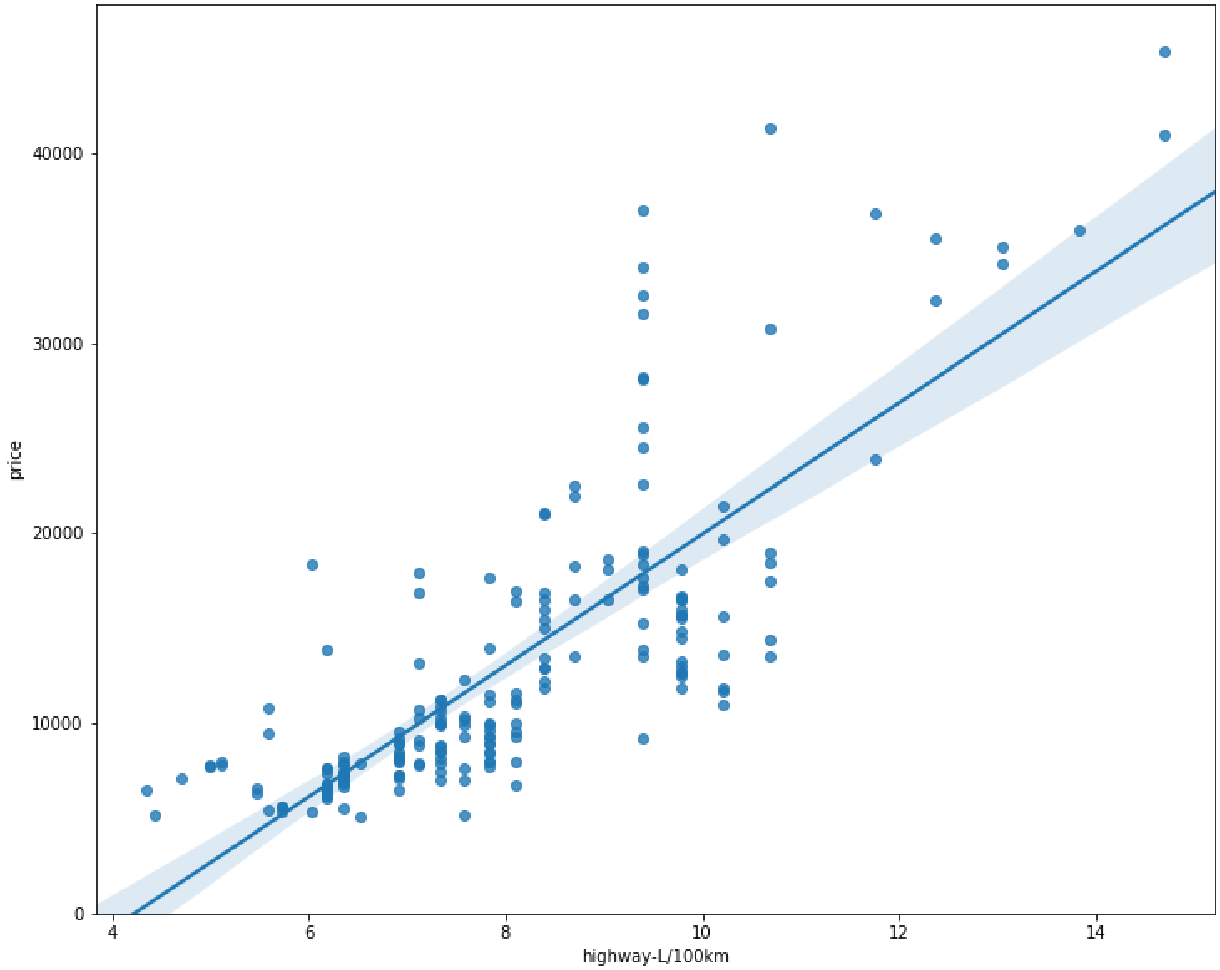
Plugging in the actual values we get:

price = -14617.84 + 3458.68 x **highway-L/100km**

Model Evaluation using Visualization

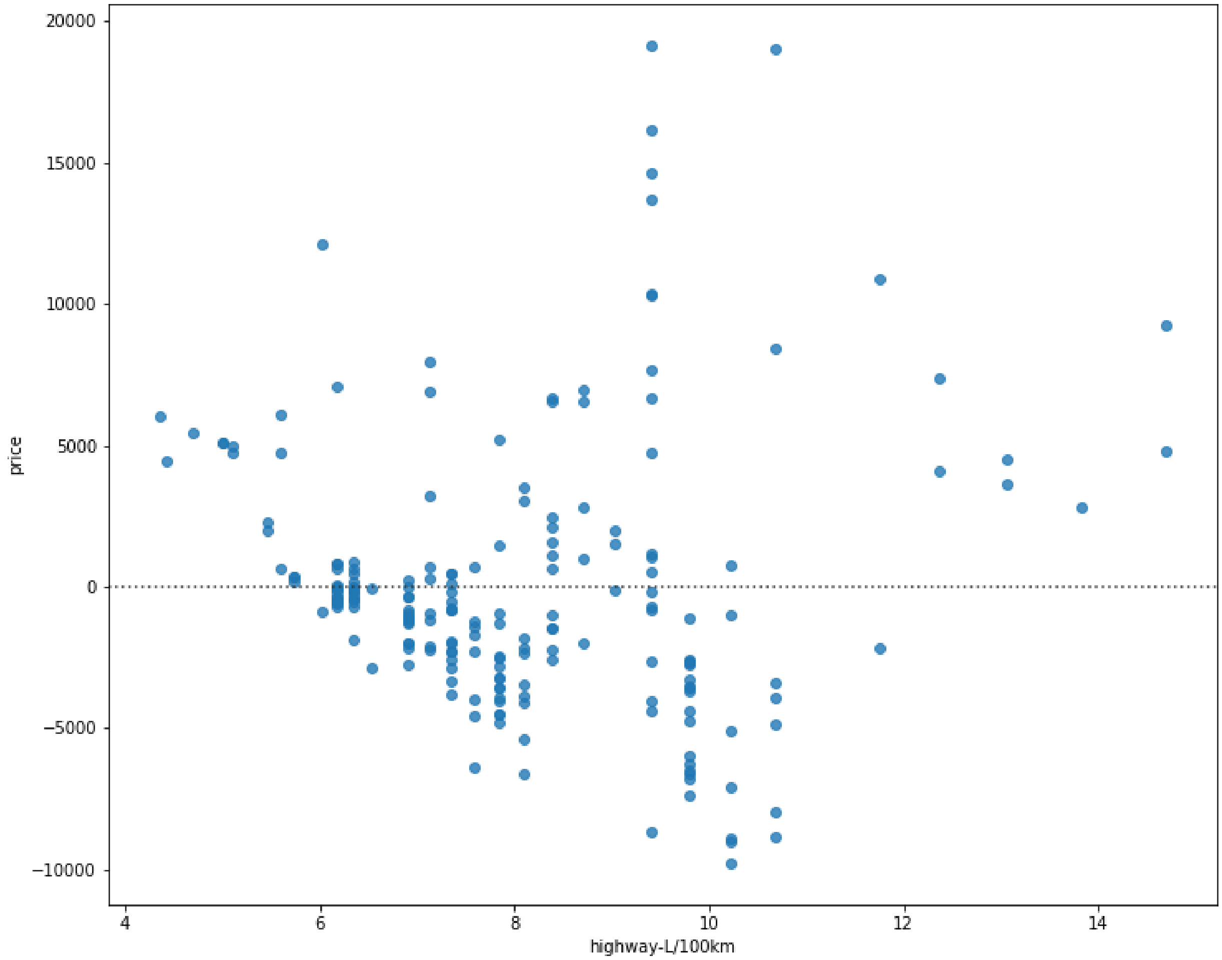
In [117]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="highway-L/100km", y="price", data=cars)
plt.ylim(0,)
```

Out[117]: (0, 47827.10608389057)



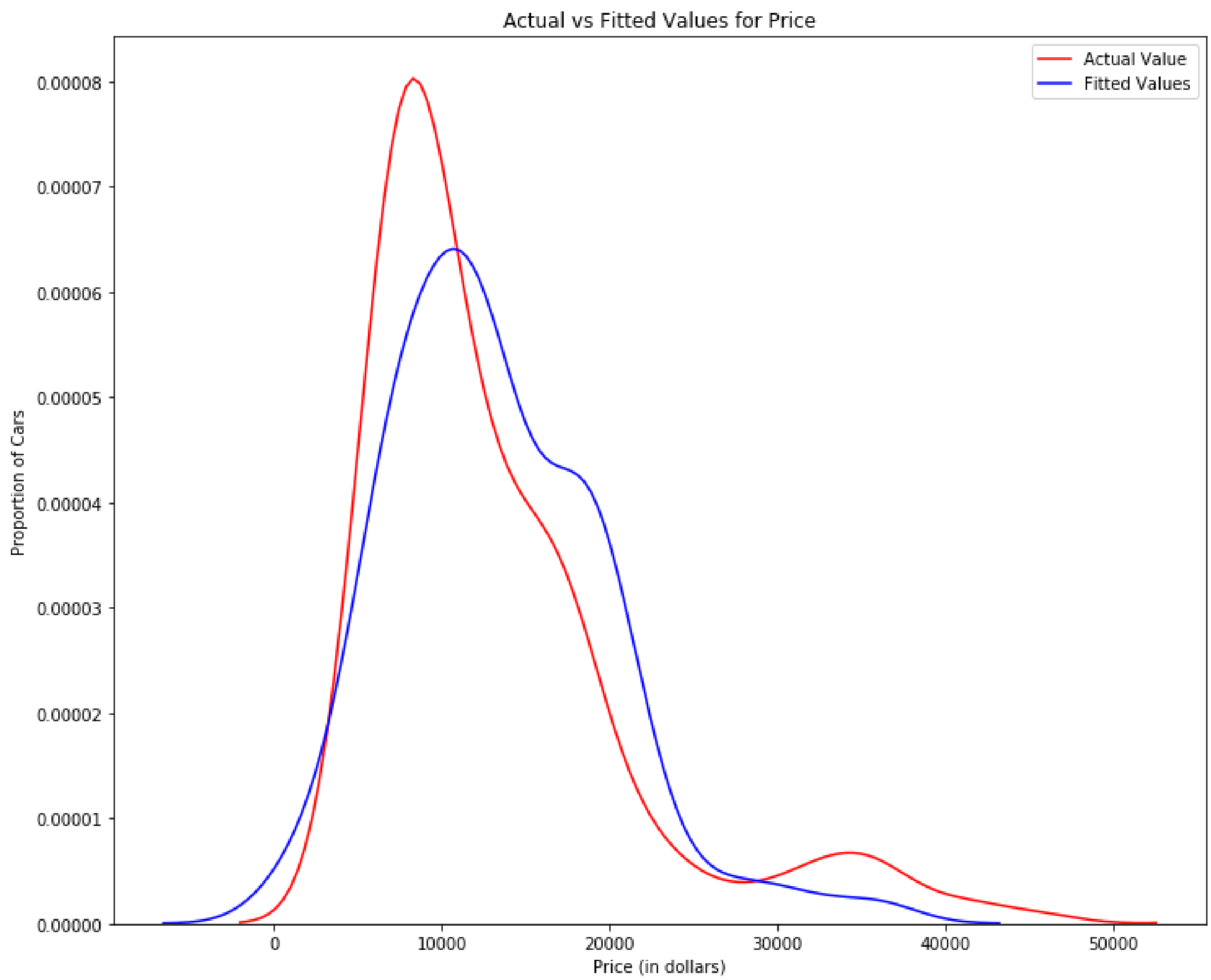
We can see from this plot that price is positively correlated to highway-L/100km, since the regression slope is positive. Since data is too far off from the line, this linear model might not be the best model for this data.

In [118]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(cars['highway-L/100km'], cars['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.

In [119]:
```
plt.figure(figsize=(width, height))


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


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.

How could city-L/100km help us predict car price?

In [120]:
```
lm1 = LinearRegression()
lm1

```

Out[120]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

we want to look at how city-L/100km can help us predict car price. Using simple linear regression, we will create a linear function with "highway-mpg" as the predictor variable and the "price" as the response variable.

In [121]:
```
X = cars[['city-L/100km']]
Y = cars['price']
```

Fit the linear model using highway-L/100km.

In [122]:
```
lm1.fit(X,Y)
```

Out[122]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

We can output a prediction

In [123]:
```
Yhat=lm1.predict(X)
Yhat[0:5]
```

Out[123]:
```
array([16293.8802044 , 16293.8802044 , 19211.26183196, 12829.48952166,
       20913.06778137])
```

In [124]:
```
print("intercept:",lm1.intercept_,"slope:",lm1.coef_)
```

```
intercept: -11421.245257455565 slope: [2476.67078595]
```

Estimated Linear model:

$$Yhat = a + bX$$

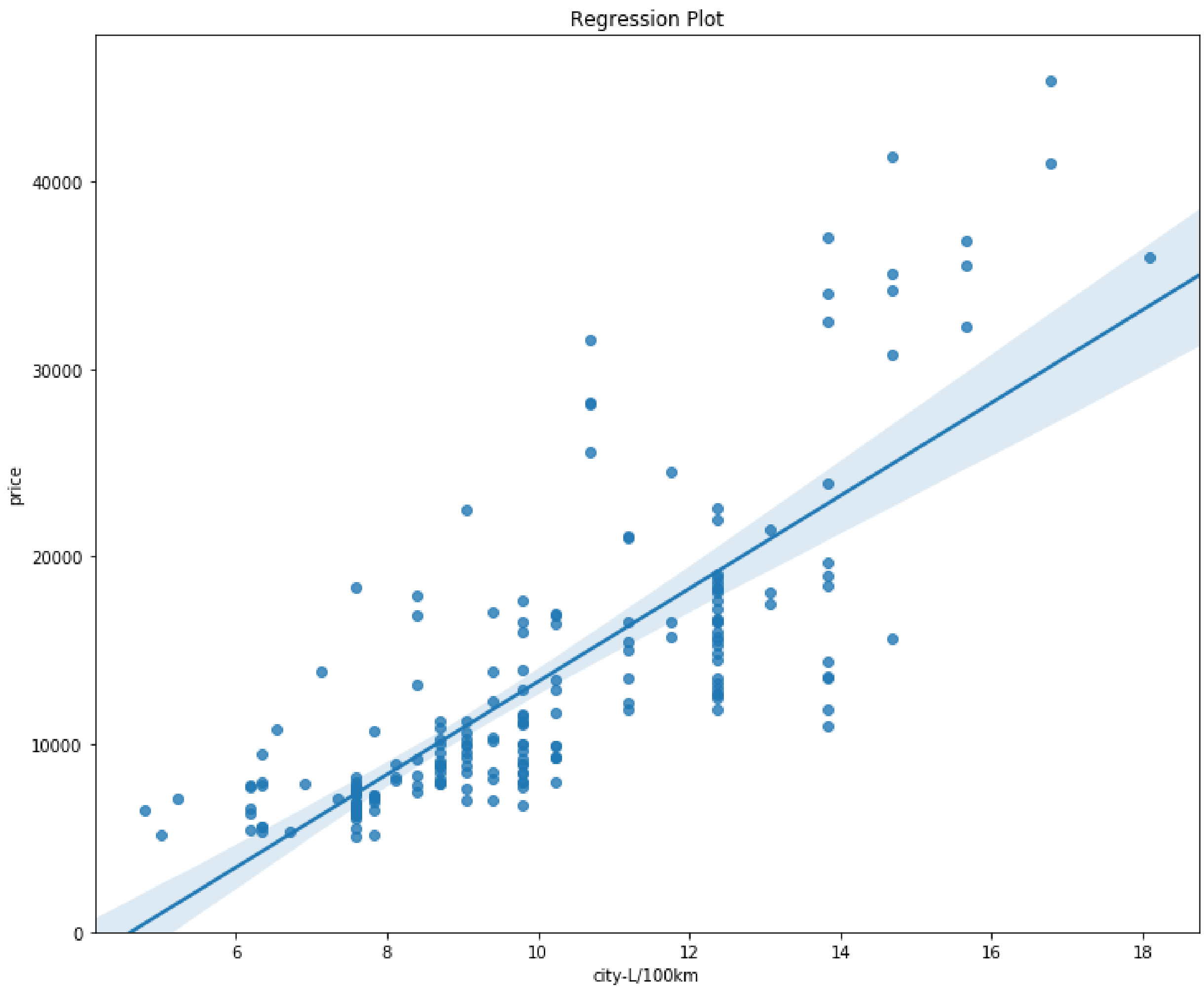
Plugging in the actual values we get:

price = -11421.24 + 2476.67 x **city-L/100km**

Model Evaluation using Visualization

In [125]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="city-L/100km", y="price", data=cars)
plt.ylim(0,)
plt.title("Regression Plot")
```

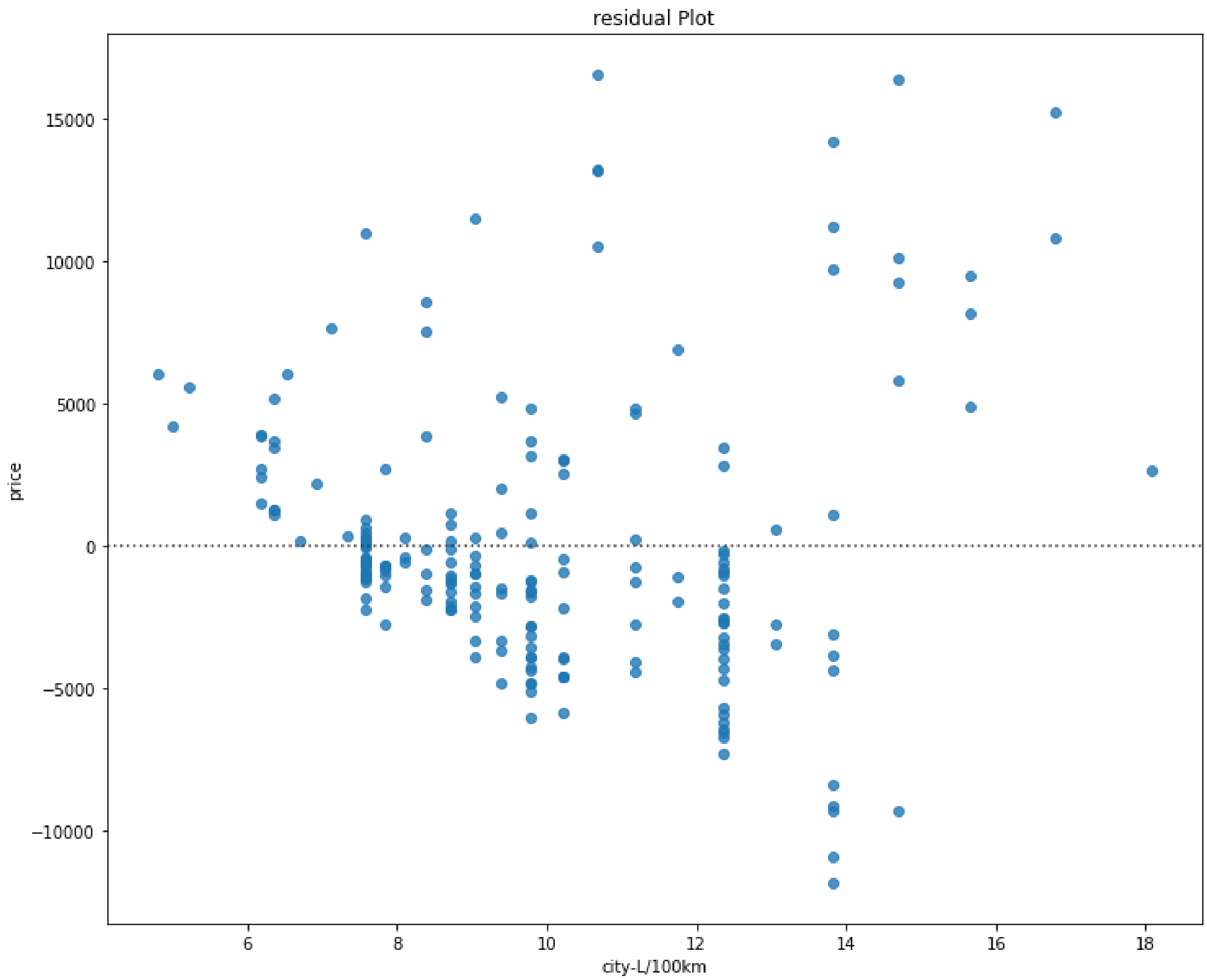
Out[125]: Text(0.5, 1.0, 'Regression Plot')



We can see from this plot that price is positively correlated to city-L/100km, since the regression slope is positive. Since data is too far off from the line, this linear model might not be the best model for this data.

In [126]:

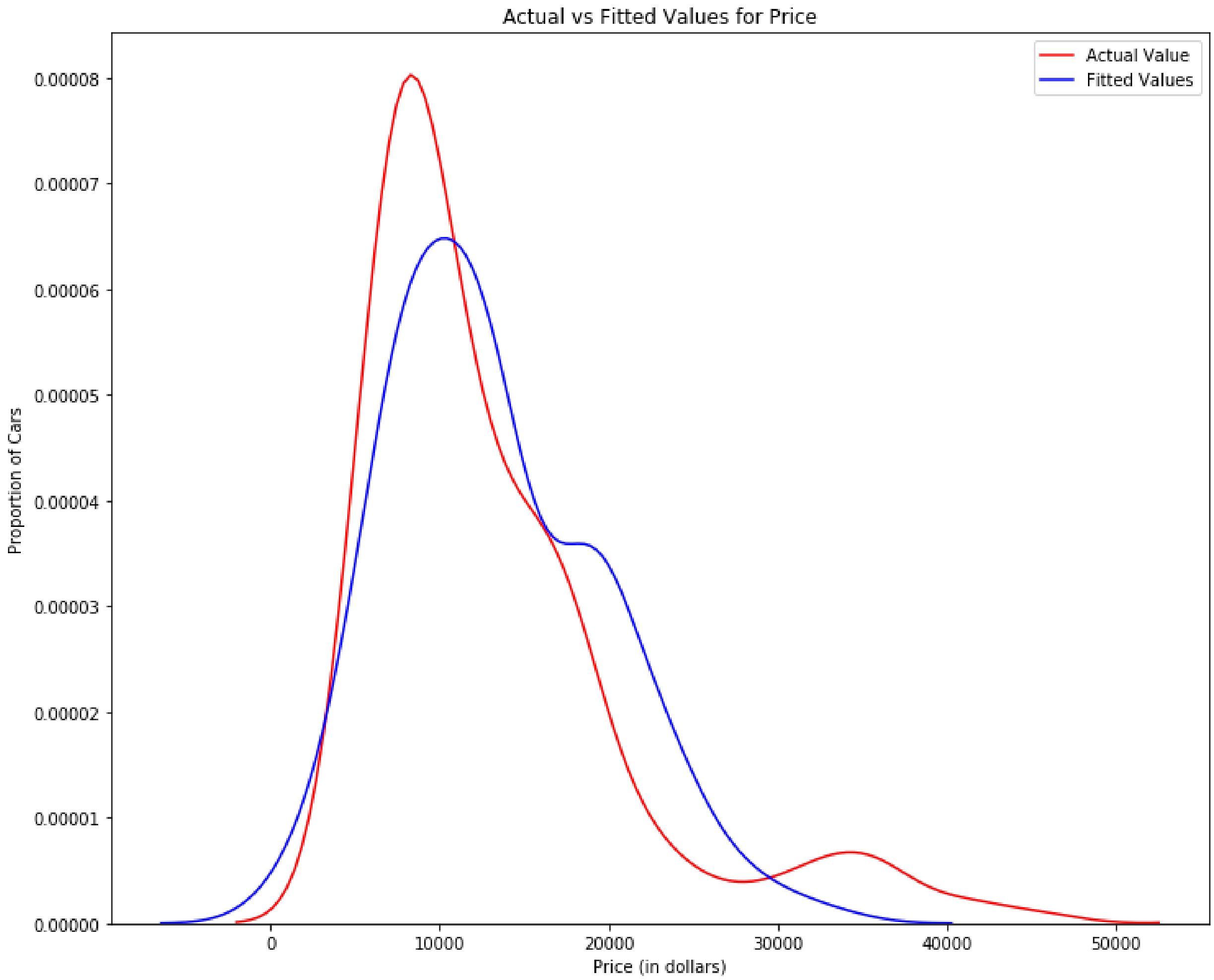
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(cars['city-L/100km'], cars['price'])
plt.title("residual Plot")
plt.show()
```



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.

In [127]:

```
plt.figure(figsize=(width, height))


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


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.

How could Engine-size help us predict car price?

In [172]:
```
lm2 = LinearRegression()
lm2
```

Out[172]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [173]:
```
X = cars[['engine-size']]
Y = cars['price']
```

In [174]:
```
lm2.fit(X,Y)
```

Out[174]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [175]:
```
yhat = lm2.predict(X)
yhat[0:5]
```

Out[175]:
```
array([13728.4631336 , 13728.4631336 , 17399.38347881, 10224.40280408,
       14729.62322775])
```

In [176]:
```
print("intercept:",lm2.intercept_,"slope:",lm2.coef_)
```

```
intercept: -7963.338906281049 slope: [166.86001569]
```

Estimated Linear model:

$$Yhat = a + bX$$

Plugging in the actual values we get:

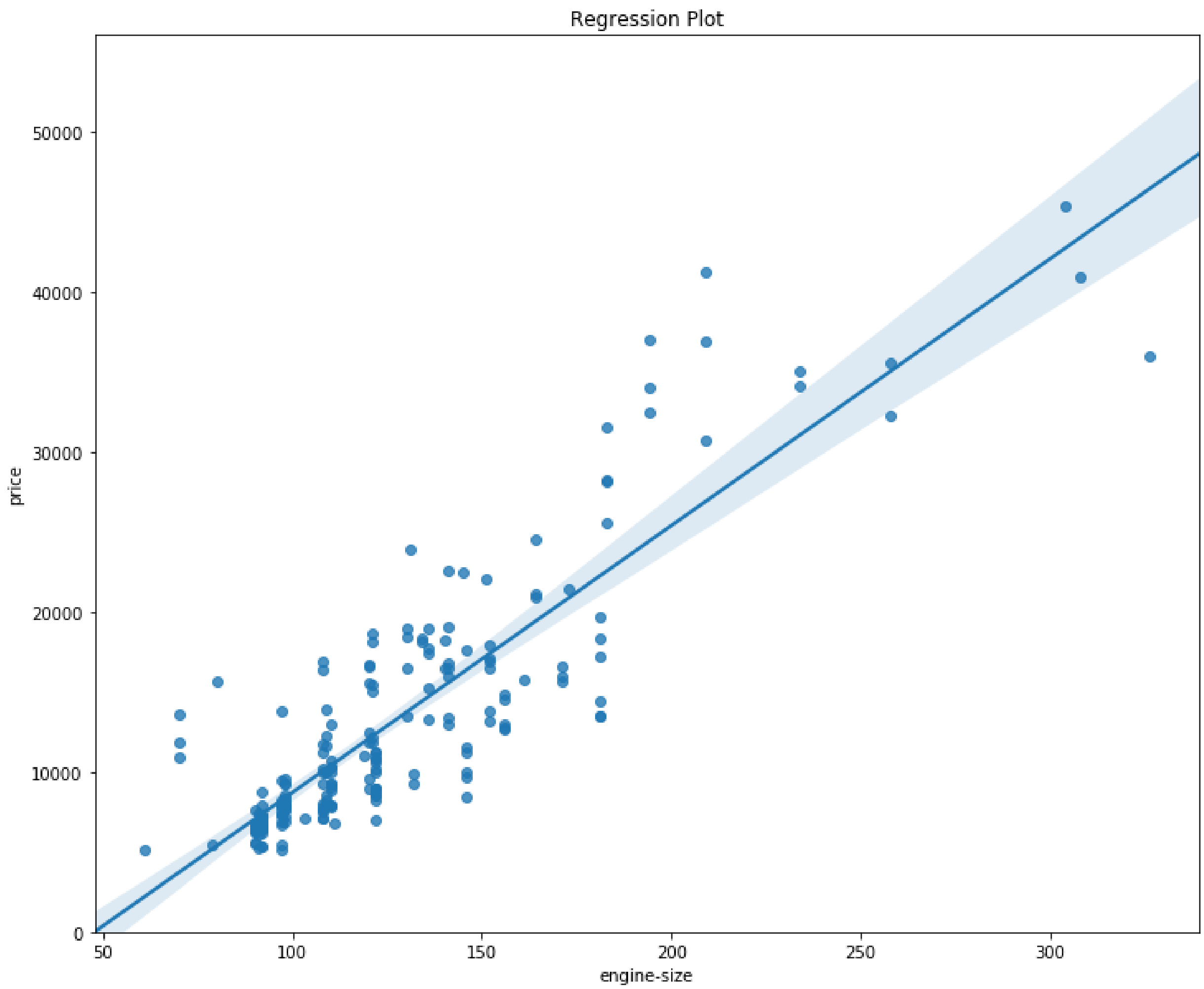
price = -7963.33 + 166.86 x **engine-size**

Model Evaluation using Visualization

In [134]:

```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.regplot(x="engine-size", y="price", data=cars)
plt.ylim(0,)
plt.title("Regression Plot")
```

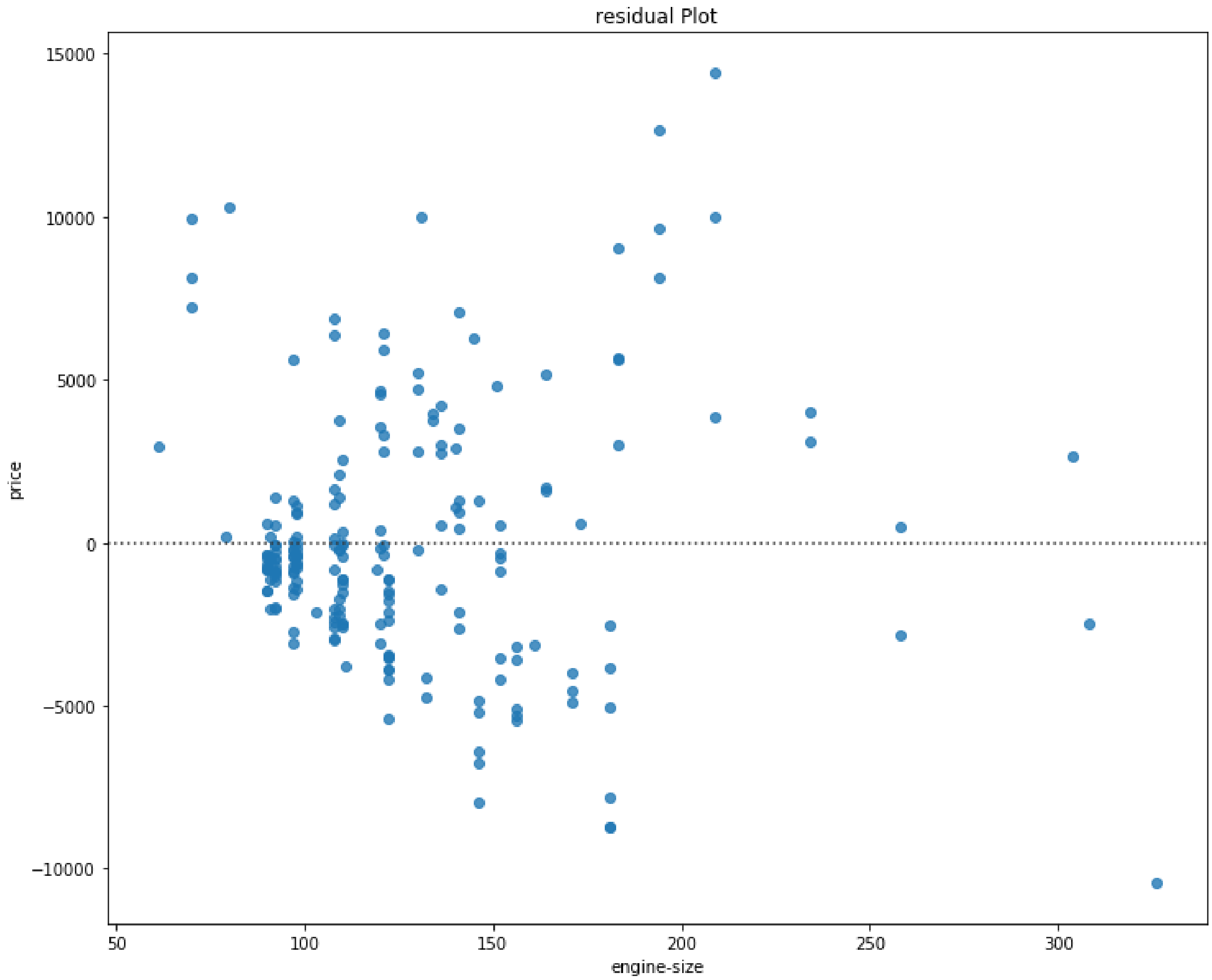
Out[134]: `Text(0.5, 1.0, 'Regression Plot')`



We can see from this plot that price is positively correlated to engine-size, since the regression slope is positive. Since data is too far off from the line, this linear model might not be the best model for this data.

In [135]:

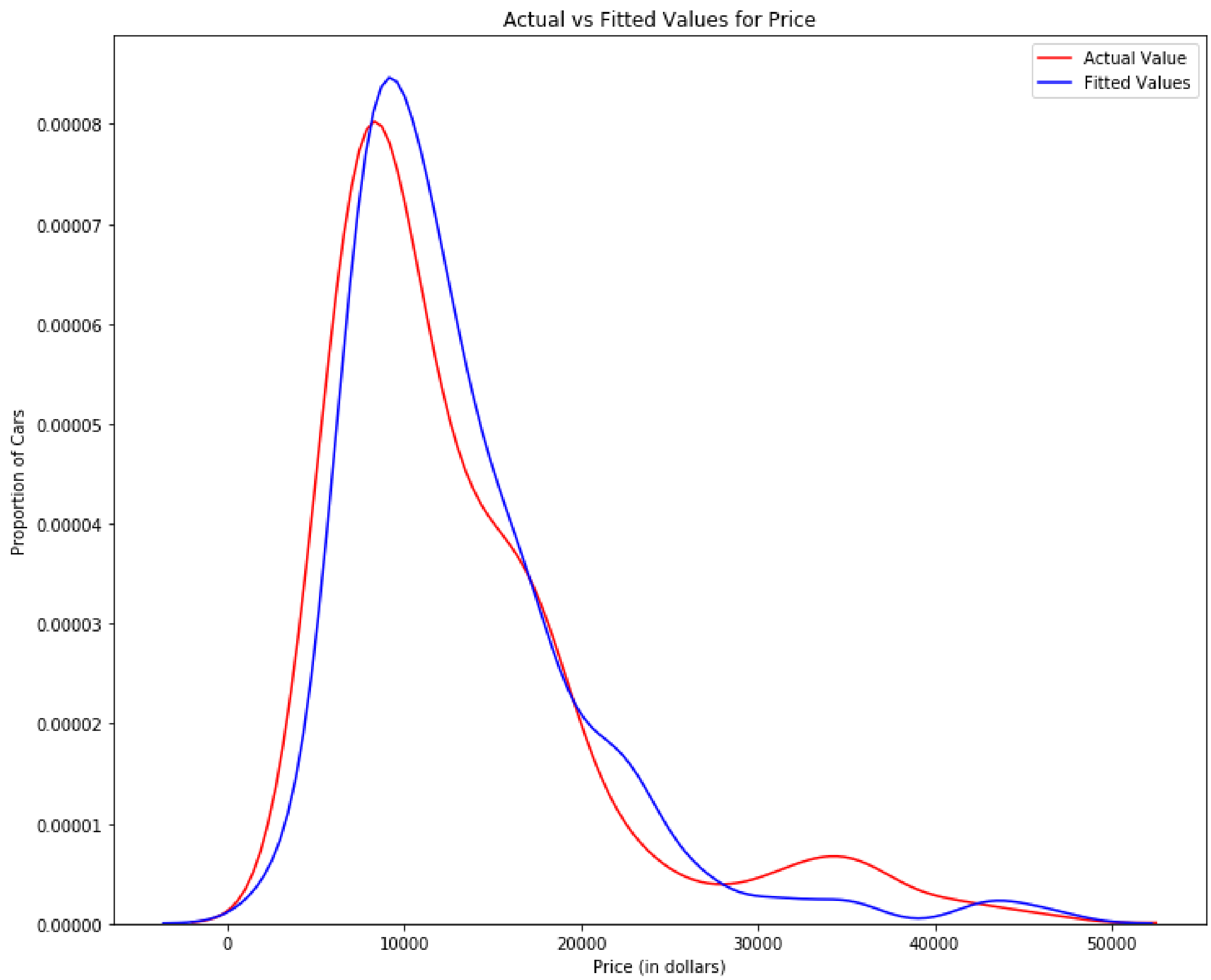
```
width = 12
height = 10
plt.figure(figsize=(width, height))
sns.residplot(cars['engine-size'], cars['price'])
plt.title("residual Plot")
plt.show()
```



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.

In [136]:

```
plt.figure(figsize=(width, height))


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


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 lot. However, there is definitely some room for improvement.

Multiple Linear Regression

In [137]:
```
cars.corr()
```

Out[137]:

	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	en
symboling	1.000000	0.466264	-0.535987	-0.365404	-0.242423	-0.550160	-0.233118	-0.1
normalized-losses	0.466264	1.000000	-0.056661	0.019424	0.086802	-0.373737	0.099404	0.1
wheel-base	-0.535987	-0.056661	1.000000	0.876024	0.814507	0.590742	0.782097	0.5
length	-0.365404	0.019424	0.876024	1.000000	0.857170	0.492063	0.880665	0.6
width	-0.242423	0.086802	0.814507	0.857170	1.000000	0.306002	0.866201	0.7
height	-0.550160	-0.373737	0.590742	0.492063	0.306002	1.000000	0.307581	0.0
curb-weight	-0.233118	0.099404	0.782097	0.880665	0.866201	0.307581	1.000000	0.8
engine-size	-0.110581	0.112360	0.572027	0.685025	0.729436	0.074694	0.849072	1.0
bore	-0.140019	-0.029862	0.493244	0.608971	0.544885	0.180449	0.644060	0.5
stroke	-0.000059	0.059131	0.155225	0.121904	0.188301	-0.068633	0.166038	0.1

Length Width Curb-weight Engine-size Horsepower City-L/100km Highway-L/100km Wheel-base Bore,,,,,,,Let's develop a model using these variables as the predictor variables.

In [195]:
```
z = cars[["curb-weight","engine-size","horsepower","highway-L/100km"]]
```

Fit the linear model using the four above-mentioned variables.

In [196]:
```
lm_m = LinearRegression()
lm_m
```

Out[196]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [197]:
```
lm_m.fit(z,cars["price"])
```

Out[197]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [198]:
```
yhat = lm_m.predict(z)
yhat[0:5]
```

Out[198]:
```
array([14055.08612634, 14055.08612634, 18638.8884263 , 10758.2803759 ,
       16670.07025864])
```

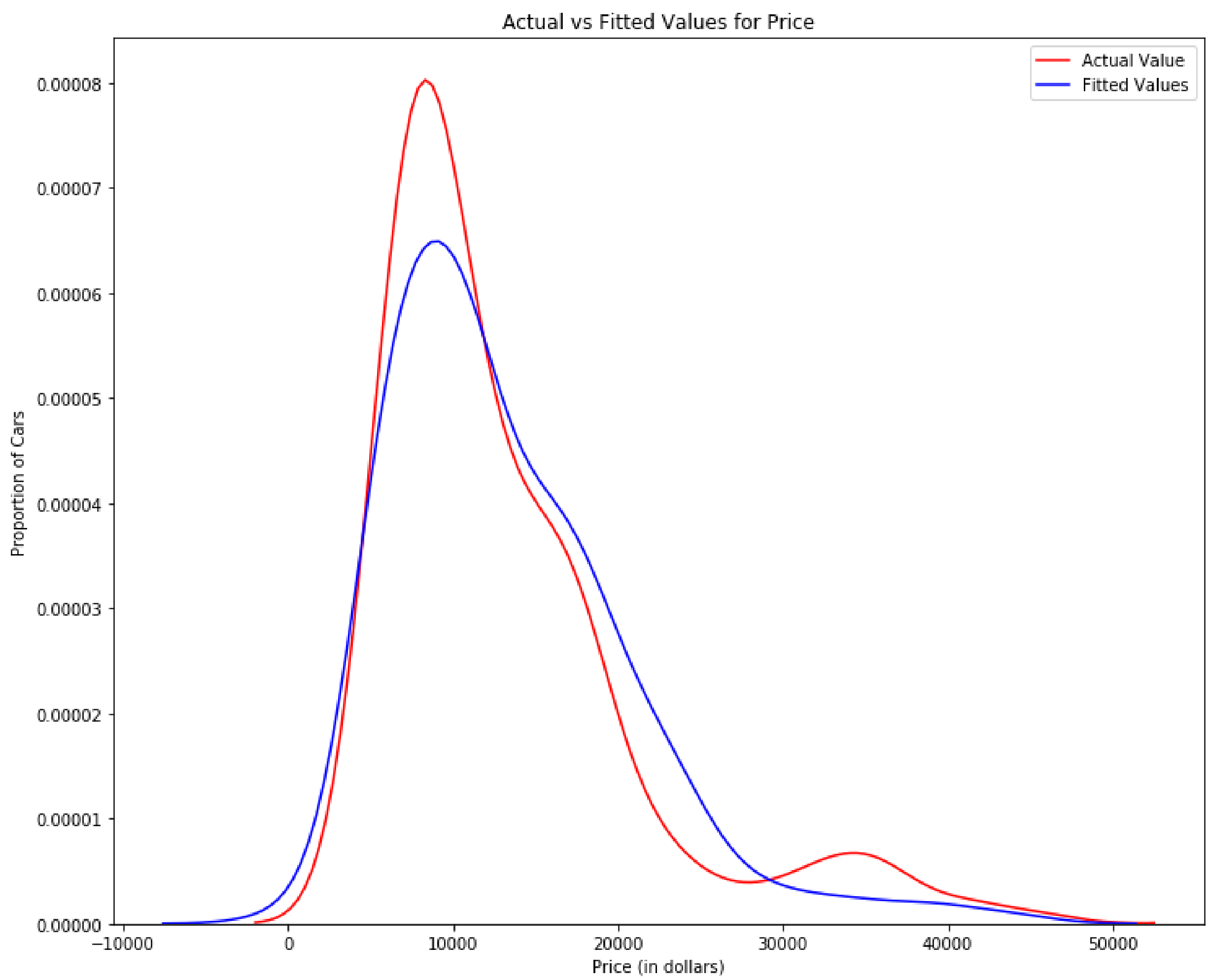
In [199]:
```
print("intercept:",lm_m.intercept_,"slope:",lm_m.coef_)
```

```
intercept: -14385.634549360078 slope: [  3.50038215  85.37370862  36.6377371  5
00.51979785]
```

In [200]:
```
lm_m.score(z,cars["price"])
```

Out[200]: 0.811811561534475

In [143]:
```
plt.figure(figsize=(width, height))


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


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 lot. However, there is definitely some room for improvement.

In [144]:
```
lm_m.score(z,cars["price"])
```

Out[144]: 0.811811561534475

Polynomial Regression

highway-L/100km(corr() with price = 0.801)

In [205]:
```
x = cars["highway-L/100km"]
y = cars["price"]

```

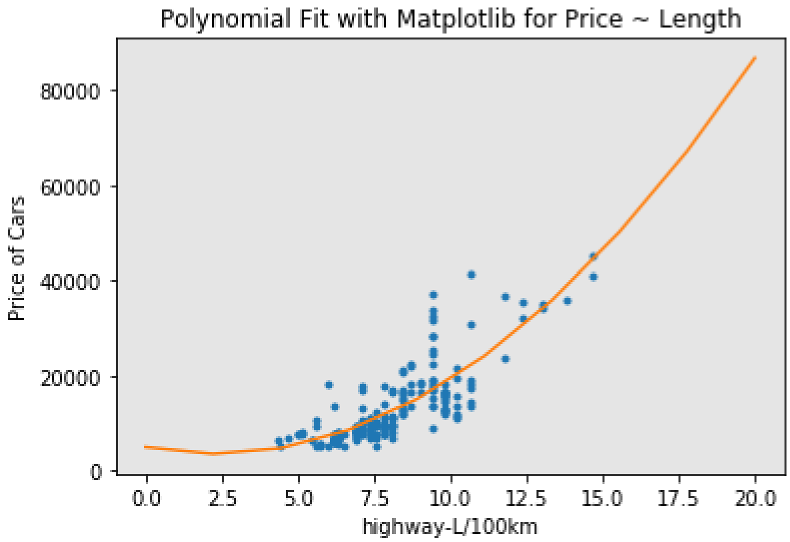
Let's fit the polynomial using the function **polyfit**, then use the function **poly1d** to display the polynomial function.

In [206]:
```
#We are using a polynomial of degree 2.
f = np.polyfit(x,y,2)
p = np.poly1d(f)
print(p)
```

```
       2
266.1 x - 1234 x + 5019
```

In [215]:
```
def PlotPolly(model, independent_variable, dependent_variabble, Name):
    x_new = np.linspace(0, 20, 10)
    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 [216]:
```
PlotPolly(p, x, y, 'highway-L/100km')
```



In [217]:
```
from sklearn.metrics import r2_score
```

In [218]:
```
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
```

```
The R-square value is:  0.6733365470678851
```

In [219]:
```
from sklearn.metrics import mean_squared_error
mean_squared_error(cars['price'], p(x))
```

Out[219]: 20528072.06493496

Performing a polynomial transform on multiple features:(Z)

In [150]:
```
from sklearn.preprocessing import PolynomialFeatures
```

We create a **PolynomialFeatures** object of degree 2:

In [151]:
```
pr=PolynomialFeatures(degree=2)
pr
```

Out[151]:
```
PolynomialFeatures(degree=2, include_bias=True, interaction_only=False,
                   order='C')
```

In [152]:
```
Z_pr=pr.fit_transform(z)
```

The original data is of 201 samples and 4 features

In [153]:
```
z.shape
```

Out[153]:
```
(201, 4)
```

In [154]:
```
Z_pr.shape
```

Out[154]:
```
(201, 15)
```

after the transformation, there 201 samples and 15 features

Pipeline

In [155]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

In [156]:
```
Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include_
```

In [157]:
```
pipe=Pipeline(Input)
pipe
```

Out[157]:
```
Pipeline(memory=None,
         steps=[('scale',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('polynomial',
                 PolynomialFeatures(degree=2, include_bias=False,
                                    interaction_only=False, order='C')),
                ('model',
                 LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                                  normalize=False))],
         verbose=False)
```

We can normalize the data, perform a transform and fit the model simultaneously.

In [158]:
```
pipe.fit(z,cars["price"])
```

Out[158]:
```
Pipeline(memory=None,
         steps=[('scale',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('polynomial',
                 PolynomialFeatures(degree=2, include_bias=False,
                                    interaction_only=False, order='C')),
                ('model',
                 LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                                  normalize=False))],
         verbose=False)
```

Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously

In [159]:
```
ypipe=pipe.predict(z)
ypipe[0:4]
```

Out[159]:
```
array([12395.58706934, 12395.58706934, 18422.45365339,  9979.7757077 ])
```

In [160]:
```
pipe.score(z,cars["price"])
```

Out[160]:
```
0.8531061720624729
```

Measures for In-Sample Evaluation

When evaluating our models, not only do we want to visualize the results, but we also want a quantitative measure to determine how accurate the model is.

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

- **R^2 / R-squared**
- **Mean Squared Error (MSE)**

R-squared

R squared, also known as the coefficient of determination, is a measure to indicate how close the data is to the fitted regression line.

The value of the R-squared is the percentage of variation of the response variable (y) that is explained by a linear model.

Mean Squared Error (MSE)

The Mean Squared Error measures the average of the squares of errors, that is, the difference between actual value (y) and the estimated value (ŷ).

Model 1: Simple Linear Regression

Let's calculate the R^2

In [189]:
```
#highway-L/100km_fit
lm.fit(X, Y)
# Find the R^2
print('The R-square is: ', lm.score(X, Y))
```

```
The R-square is:  0.6417894513258818
```

We can say that ~ 64.178% of the variation of the price is explained by this simple linear model "highway-L/100km_fit".

Let's calculate the MSE

In [190]:
```
Yhat=lm.predict(X)
print('The output of the first four predicted value is: ', Yhat[0:4])
```

```
The output of the first four predicted value is:  [15485.52737455 15485.5273745
5 16643.34931414 12475.19033163]
```

we compare the predicted results with the actual results

In [192]:
```
mse = mean_squared_error(cars['price'], Yhat)
print('The mean square error of price and predicted value is: ', mse)
```

```
The mean square error of price and predicted value is:  22510543.777085222
```

Model 2: Multiple Linear Regression

Let's calculate the R^2

In [202]:
```
# fit the model
lm_m.fit(z, cars['price'])
# Find the R^2
print('The R-square is: ', lm.score(z, cars['price']))
```

```
The R-square is:  0.811811561534475
```

We can say that ~ 81.181 % of the variation of price is explained by this multiple linear regression "multi_fit".

Let's calculate the MSE

In [203]:
```
Y_predict_multifit = lm_m.predict(z)
```

In [204]:
```
print('The mean square error of price and predicted value using multifit is:
      mean_squared_error(cars['price'], Y_predict_multifit))
```

```
The mean square error of price and predicted value using multifit is:  1182607
2.956532082
```

Model 3: Polynomial Fit

Let's calculate the R^2

In [220]:
```
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
```

```
The R-square value is:  0.6733365470678851
```

We can say that ~ 67.333 % of the variation of price is explained by this polynomial fit

We can also calculate the MSE:

In [221]:
```
mean_squared_error(cars["price"], p(x))
```

Out[221]:
```
20528072.06493496
```

Decision Making: Determining a Good Model Fit

Now that we have visualized the different models, and generated the R-squared and MSE values for the fits, how do we determine a good model fit?

- *What is a good R-squared value?*

When comparing models, **the model with the higher R-squared value is a better fit** for the data.

- *What is a good MSE?*

When comparing models, **the model with the smallest MSE value is a better fit** for the data.

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

Simple Linear Regression: Using Highway-L/100km as a Predictor Variable of Price.

- R-squared: 0.64
- MSE: 22510543.77

Multiple Linear Regression: Using Horsepower, Curb-weight, Engine-size, and Highway-L/100km as Predictor Variables of Price.

- R-squared: 0.81
- MSE: 11826072.95

Polynomial Fit: Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.67
- MSE: 20528072.06

Simple Linear Regression model (SLR) vs Multiple Linear Regression model (MLR)

To be able to compare the results of the MLR vs SLR models, we look at a combination of both the R-squared and MSE to make the best conclusion about the fit of the model.

- **MSE**The MSE of SLR is 22510543.77 while MLR has an MSE of 11826072.95. The MSE of MLR is much smaller.
- **R-squared**: In this case, we can also see that there is a big difference between the R-squared of the SLR and the R-squared of the MLR. The R-squared for the SLR (~0.64) is very small compared to the R-squared for the MLR (~0.81).

This R-squared in combination with the MSE show that MLR seems like the better model fit in this case, compared to SLR.

Simple Linear Model (SLR) vs Polynomial Fit

- **MSE**: We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.
- **R-squared**: The R-squared for the Polyfit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.

Since the Polynomial Fit resulted in a lower MSE and a higher R-squared, we can conclude that this was a better fit model than the simple linear regression for predicting Price with Highway-mpg as a predictor variable.

Multiple Linear Regression (MLR) vs Polynomial Fit

- **MSE**: The MSE for the MLR is smaller than the MSE for the Polynomial Fit.
- **R-squared**: The R-squared for the MLR is also much larger than for the Polynomial Fit.

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.

In [ ]:

```
1
```