ANSWER KEY
Internal Assessment Test 3 – Jan , 2022

Sub:	Data Analytics using Python						Sub Code:	20MCA31
Date:	25/01//2022	Duration:	90 min's	Max Marks:	50	Sem:	III	Branch:	MCA

1 **What are the Steps in Visualization ? Explain with an example**

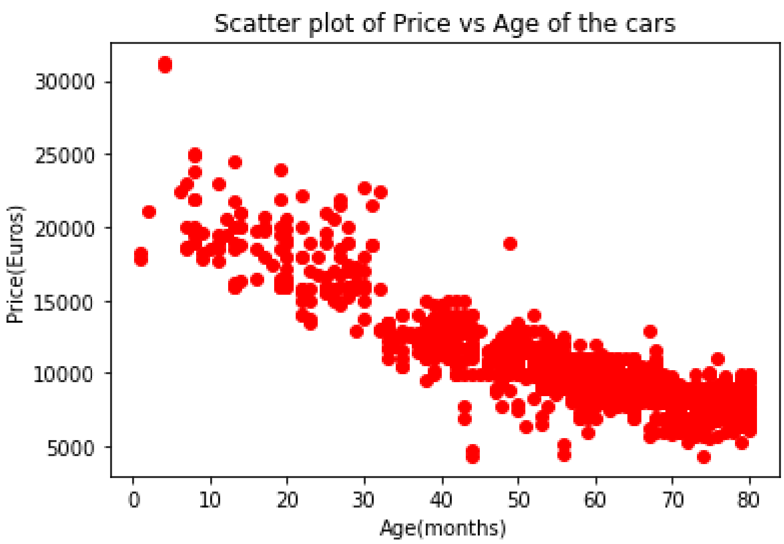
Data Visualization :Steps involved in plotting a graph

- Define the x-axis and corresponding y-axis values as lists.
- Plot them on canvas using .plot() function.
- Give a name to x-axis and y-axis using .xlabel() and .ylabel() functions.
- Give a title to your plot using .title() function.
- Finally, view the plot, using .show() function.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
cars_data = pd.read_csv('Toyota.csv', index_col=0,na_values=["??","????"])

#removing missing values from the dataframe
cars_data.dropna(axis=0, inplace=True)
```

Scatter Plot

```
plt.scatter(cars_data['Age'],cars_data['Price'],c='red')
plt.title('Scatter plot of Price vs Age of the cars')
plt.xlabel('Age(months)')
plt.ylabel('Price(Euros)')
plt.show()
```

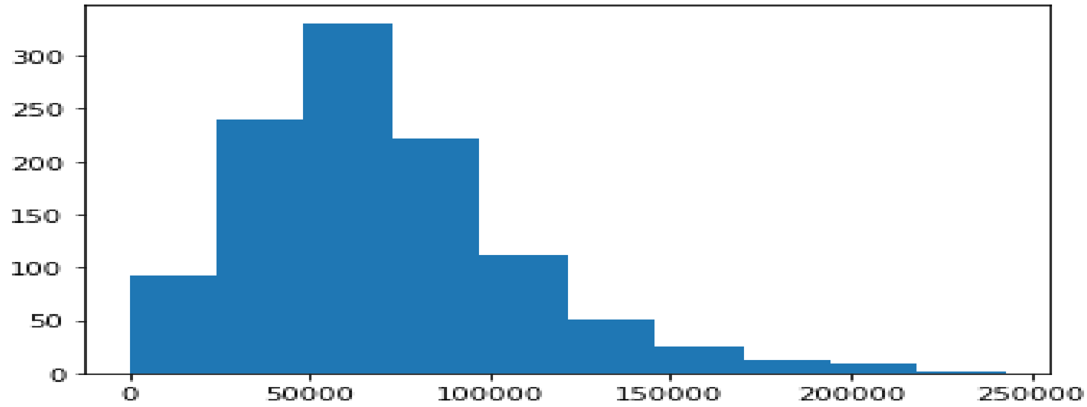


Histogram

```
plt.hist(cars_data['KM'])
```

Out[3]:

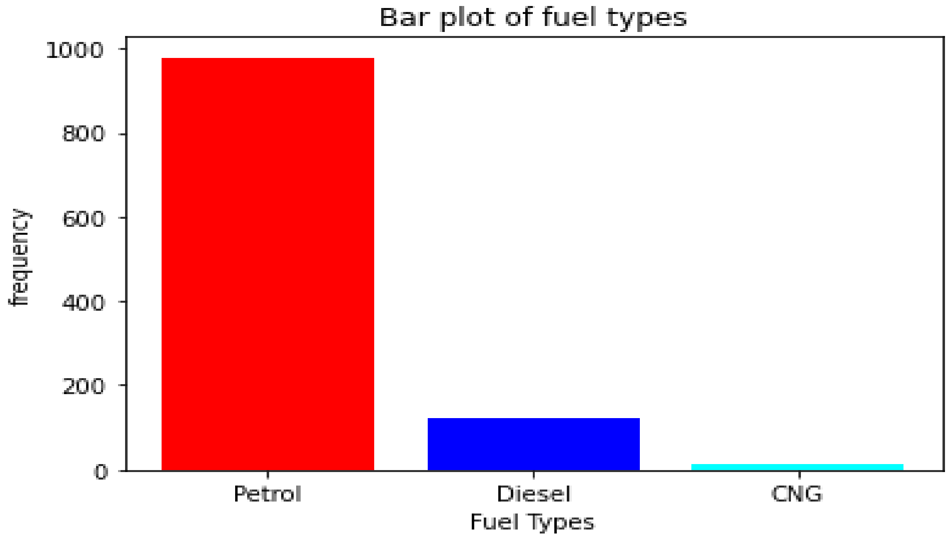
```
(array([ 92., 239., 331., 222., 111.,  51.,  25.,  13.,  10.,   2.]),
 array([1.000000e+00, 2.430090e+04, 4.860080e+04, 7.290070e+04,
        9.720060e+04, 1.215005e+05, 1.458004e+05, 1.701003e+05,
        1.944002e+05, 2.187001e+05, 2.430000e+05]),
 <BarContainer object of 10 artists>)
```



Bar Plot (for categorical variables)

In [5]:

```
counts=[979, 120, 12]
fuelType = ('Petrol', 'Diesel', 'CNG')
index = np.arange(len(fuelType))

plt.bar(index, counts,color=['red','blue', 'cyan'])
plt.title('Bar plot of fuel types')
plt.xlabel('Fuel Types')
plt.ylabel('frequency')
plt.xticks(index, fuelType, rotation = 0)
plt.show()
```



2 **What do you mean by Normalization and Standardization? Write any five differences between them**

Feature scaling is one of the most important data preprocessing step in machine learning. Algorithms that compute the distance between the features are biased towards numerically larger values if the data is not scaled.
Tree-based algorithms are fairly insensitive to the scale of the features. Also, feature scaling helps machine learning, and deep learning algorithms train and converge faster.

There are some feature scaling techniques such as Normalization and Standardization that are the most popular and at the same time, the most confusing ones.

Let's resolve that confusion.

Normalization or Min-Max Scaling is used to transform features to be on a similar scale. The new point is calculated as:

X_new = (X - X_min)/(X_max - X_min)

This scales the range to [0, 1] or sometimes [-1, 1]. Geometrically speaking, transformation squishes the n-dimensional data into an n-dimensional unit hypercube. Normalization is useful when there are no outliers as it cannot cope up with them. Usually, we would scale age and not incomes because only a few people have high incomes but the age is close to uniform.

Standardization or Z-Score Normalization is the transformation of features by subtracting from mean and dividing by standard deviation. This is often called as Z-score.

X_new = (X - mean)/Std

Standardization can be helpful in cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Geometrically speaking, it translates the data to the mean vector of original data to the origin and squishes or expands the points if std is 1 respectively. We can see that we are just changing mean and standard deviation to a standard normal distribution which is still normal thus the shape of the distribution is not affected.

Standardization does not get affected by outliers because there is no predefined range of transformed features.

Difference between Normalization and Standardization

S.NO.	Normalization	Standardization
1.	Minimum and maximum value of features are used for scaling	Mean and standard deviation is used for scaling.
2.	It is used when features are of different scales.	It is used when we want to ensure zero mean and unit standard deviation.
3.	Scales values between [0, 1] or [-1, 1].	It is not bounded to a certain range.
4.	It is really affected by outliers.	It is much less affected by outliers.
5.	Scikit-Learn provides a transformer called `MinMaxScaler` for Normalization.	Scikit-Learn provides a transformer called `StandardScaler` for standardization.
6.	This transformation squishes the n-dimensional data into an n-dimensional unit hypercube.	It translates the data to the mean vector of original data to the origin and squishes or expands.
7.	It is useful when we don't know about the distribution	It is useful when the feature distribution is Normal or Gaussian.
8.	It is a often called as Scaling Normalization	It is a often called as Z-Score Normalization.

3 **Write the importance of self and __init__() method. Give an example**

i)Importance of 'self'

- Explicit reference to refer the current object, i.e the object which invoked the method
- Used to create and initialize instance variables of a class i.e it creates the attribute for the class
- 'self' reference must be used as a first parameter in all instance methods of a class otherwise

the methods are known as simply “class methods”

- Moreover, “self” is not a keyword and has no special meaning in Python. We can use any name in that place. However, it is recommended not to use any name other than “self” (merely a convention and for readability)

ii) __init__() method

- __init__() method helps in initializing the variable during the creation of object. Hence, it is also called as ‘initializer method’ or Default Constructor.

Instance Methods and Static Methods
Instance / Regular methods require an instance (self) as the first argument and when the method is invoked (bound), self is automatically passed to the method

- Static methods are functions which do not require

instance but are part of class definitions.
When to use Static and Instance methods?

- Instance methods Useful when method needs access to the values that are specific to the instance and needs to call other methods that have access to instance specific values.

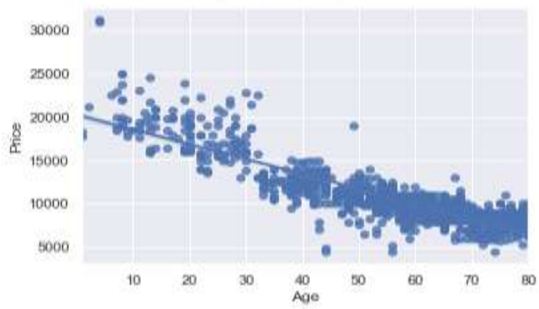
4 **Write a Python program to demonstrate Data Visualization using Seaborn**

```
In [ ]: ##11.Write a Python program to demonstrate Data Visualization using Seaborn.
```

Scatter Plot

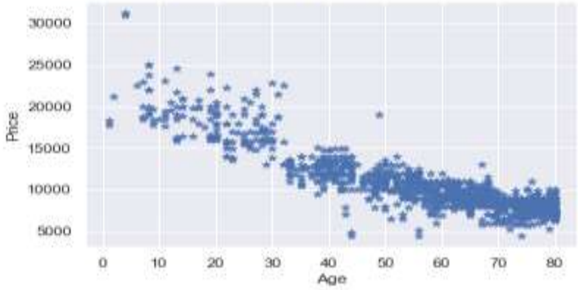
```
In [5]:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# scatter plot of Price and age with the regeression fitline
cars_data = pd.read_csv('Toyota.csv', index_col=0, na_values=["??","????"])
cars_data.dropna(axis=0, inplace= True)
sns.set(style="darkgrid")
sns.regplot(x=cars_data['Age'],y= cars_data['Price'])
```

```
Out[5]: <AxesSubplot:xlabel='Age', ylabel='Price'>
```



In [6]:
```
# scatter plot of Price and age without the regeression fitline
sns.regplot(x=cars_data['Age'],y= cars_data['Price'],marker='*' ,fit_reg=False)
```

Out[6]: <AxesSubplot:xlabel='Age', ylabel='Price'>



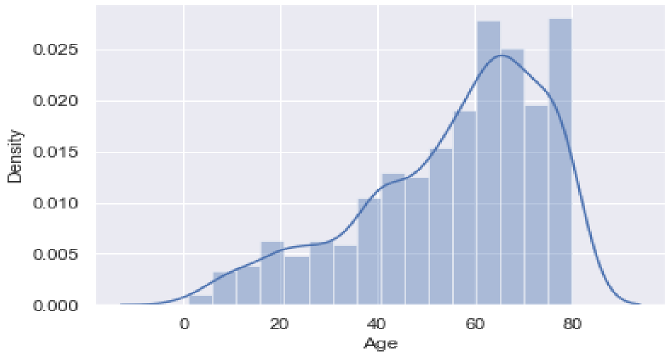
Histogram

Histogram with default kernel density estimate

In [7]:
```
#distribution of the variable 'Age'
sns.distplot(cars_data['Age'])
```

```
d:\Users\jvaku\anaconda3\envs\tflow\lib\site-packages\seaborn\distributions.py:2557
use either `displot` (a figure-level function with similar flexibility) or `histplo
  warnings.warn(msg, FutureWarning)
```

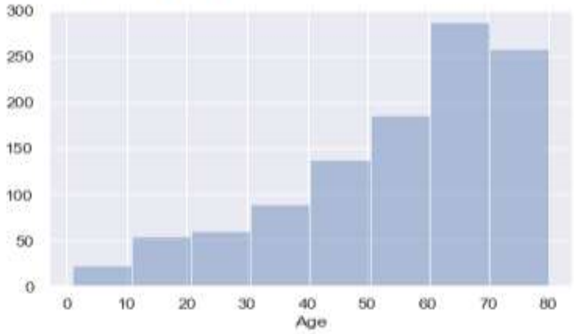
Out[7]: <AxesSubplot:xlabel='Age', ylabel='Density'>



In [8]:
```
sns.distplot(cars_data['Age'],kde=False, bins = 8)
```

```
d:\Users\jvaku\anaconda3\envs\tflow\lib\site-packages\seaborn\distributions.py:2557: Fut
use either `displot` (a figure-level function with similar flexibility) or `histplot` (a
  warnings.warn(msg, FutureWarning)
```

Out[8]: <AxesSubplot:xlabel='Age'>

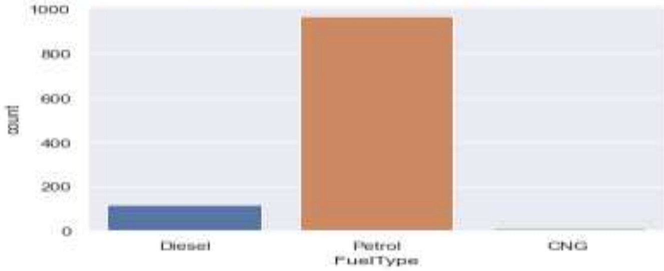


Bar Plot

Bar Plot

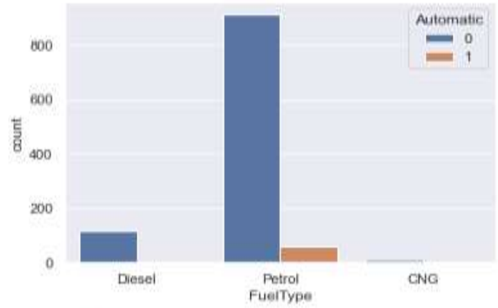
In [9]:
```
# frequency distribution of categorical variable 'fuelType'
sns.countplot(x="FuelType", data=cars_data)
```

Out[9]: <AxesSubplot:xlabel='FuelType', ylabel='count'>



In [10]:
```
#grouped bar plot of FuelType and Automatic
sns.countplot(x="FuelType", data=cars_data, hue="Automatic")
```

Out[10]: <AxesSubplot:xlabel='FuelType', ylabel='count'>

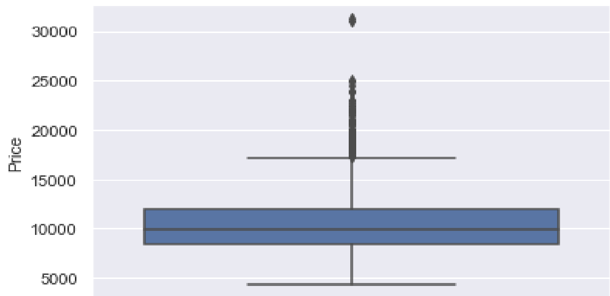


Box and Whiskers plot

(For numerical variable)

In [11]:
```
sns.boxplot(y=cars_data["Price"])
```

Out[11]: <AxesSubplot:ylabel='Price'>



5 Explain the terms histogram, binning and density.

A simple histogram can be a great first step in understanding a dataset. Earlier, we saw a preview of Matplotlib's histogram function which creates a basic histogram in one line, once the normal boiler-plate imports are done:

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

data = np.random.randn(1000)
plt.hist(data);
```

The `hist()` function has many options to tune both the calculation and the display; here's an example of a more customized histogram:

```
plt.hist(data, bins=30, normed=True, alpha=0.5,
         histtype='stepfilled', color='steelblue',
         edgecolor='none');
```

The `plt.hist` docstring has more information on other customization options available. I find this combination of `histtype='stepfilled'` along with some transparency `alpha` to be very useful when comparing histograms of several distributions:

```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
x3 = np.random.normal(3, 2, 1000)

kwargs = dict(histtype='stepfilled', alpha=0.3, normed=True, bins=40)

plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
plt.hist(x3, **kwargs);
```

If you would like to simply compute the histogram (that is, count the number of points in a given bin) and not display it, the `np.histogram()` function is available:

```
counts, bin_edges = np.histogram(data, bins=5)
print(counts)
[ 12 190 468 301  29]
```

Two-Dimensional Histograms and Binnings

Just as we create histograms in one dimension by dividing the number-line into bins, we can also create histograms in two-dimensions by dividing points among two-dimensional bins. We'll take a brief look at several ways to do this here. We'll start by defining some data—an `x` and `y` array drawn from a multivariate Gaussian distribution:

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

`plt.hist2d`: Two-dimensional histogram

One straightforward way to plot a two-dimensional histogram is to use Matplotlib's `plt.hist2d` function:

```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```

Just as with `plt.hist`, `plt.hist2d` has a number of extra options to fine-tune the plot and the binning, which are nicely outlined in the function docstring. Further, just as `plt.hist` has a counterpart in `np.histogram`, `plt.hist2d` has a counterpart in `np.histogram2d`, which can be used as follows:

```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

For the generalization of this histogram binning in dimensions higher than two, see the `np.histogramdd` function.

`plt.hexbin`: Hexagonal binnings

The two-dimensional histogram creates a tesselation of squares across the axes. Another natural shape for such a tesselation is the regular hexagon. For this purpose, Matplotlib provides the `plt.hexbin` routine, which will represents a two-dimensional dataset binned within a grid of hexagons:

```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```

`plt.hexbin` has a number of interesting options, including the ability to specify weights for each point, and to change the output in each bin to any NumPy aggregate (mean of weights, standard deviation of weights, etc.).

Kernel density estimation

Another common method of evaluating densities in multiple dimensions is *kernel density estimation* (KDE). This will be discussed more fully in [In-Depth: Kernel Density Estimation](), but for now we'll simply mention that KDE can be thought of as a way to "smear out" the points in space and add up the result to obtain a smooth function. One extremely quick and simple KDE implementation exists in the `scipy.stats` package. Here is a quick example of using the KDE on this data:

```
from scipy.stats import gaussian_kde

# fit an array of size [Ndim, Nsamples]
data = np.vstack([x, y])
kde = gaussian_kde(data)

# evaluate on a regular grid
xgrid = np.linspace(-3.5, 3.5, 40)
ygrid = np.linspace(-6, 6, 40)
Xgrid, Ygrid = np.meshgrid(xgrid, ygrid)
Z = kde.evaluate(np.vstack([Xgrid.ravel(), Ygrid.ravel()]))

# Plot the result as an image
plt.imshow(Z.reshape(Xgrid.shape),
           origin='lower', aspect='auto',
           extent=[-3.5, 3.5, -6, 6],
           cmap='Blues')
cb = plt.colorbar()
cb.set_label("density")
```

KDE has a smoothing length that effectively slides the knob between detail and smoothness (one example of the ubiquitous bias–variance trade-off). The literature on choosing an appropriate smoothing length is vast: `gaussian_kde` uses a rule-of-thumb to attempt to find a nearly optimal smoothing length for the input data.Other KDE implementations are available within the SciPy ecosystem, each with its own strengths and weaknesses; see, for example, `sklearn.neighbors.KernelDensity` and `statsmodels.nonparametric.kernel_density.KDEMultivariate`

6 **Explain with an example "The GroupBy object "-- aggregate, filter, transform, and apply**

The GroupBy object is a very flexible abstraction. In many ways, you can simply treat it as if it's a collection of DataFrames, and it does the difficult things under the hood. Let's see some examples using the Planets data.

Perhaps the most important operations made available by a GroupBy are *aggregate*, *filter*, *transform*, and *apply*. We'll discuss each of these more fully in "Aggregate, Filter, Transform, Apply", but before that let's introduce some of the other functionality that can be used with the basic GroupBy operation.

Aggregate, filter, transform, apply

The preceding discussion focused on aggregation for the combine operation, but there are more options available. In particular, GroupBy objects have aggregate(), filter(), transform(), and apply() methods that efficiently implement a variety of useful operations before combining the grouped data.
For the purpose of the following subsections, we'll use this DataFrame:

Code

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
               'data1': range(6),
               'data2': rng.randint(0, 10, 6)},
               columns = ['key', 'data1', 'data2'])
df
```

Out[18]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation

The aggregate() method allows for even more flexibility. It can take a string, a function, or a list thereof, and compute all the aggregates at once. Here is a quick example combining all these:

```
In [21]: df.groupby('key').aggregate(['min', np.median, max])
```

Out[21]:

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Filtering

A filtering operation allows you to drop data based on the group properties. For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

In [29]:
```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()", df.groupby('key').filter(filter_func))
```

'df'

"df.groupby('key').std()"

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

The filter function should return a Boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the result.

Transformation

While aggregation must return a reduced version of the data, transformation can return some transformed version of the full data to recombine. For such a transformation, the output is the same shape as the input. A common example is to center the data by subtracting the group-wise mean:

In [24]:
```
df.groupby('key').transform(lambda x: x - x.mean())
```

Out[24]:

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

The apply() method

The `apply()` method lets you apply an arbitrary function to the group results. The function should take a `DataFrame`, and return either a Pandas object (e.g., `DataFrame`, `Series`) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` that normalizes the first column by the sum of the second:

In [28]:
```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', df.groupby('key').apply(norm_by_data2))
```

'df'

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

7 **When to use Static and Instance methods? Explain with an example**

Instance / Regular methods require an instance (self) as the first argument and when the method is invoked (bound), self is automatically passed to the method

- Static methods are functions which do not require instance but are part of class definitions.
- Static mathods Useful when method does not need access to either the class variables or the instance variables.
- Instance methods Useful when method needs access to the values that are specific to the instance and needs to call other methods that have access to instance specific values.

	• Before init () method, the object is been already constructed
8	**Justify the need of Pivot Tables? Explain with and example**

We have seen how the GroupBy abstraction lets us explore relationships within a dataset.

A *pivot table* is a similar operation that is commonly seen in spreadsheets and other programs that operate on tabular data.

The pivot table takes simple column-wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data.

The difference between pivot tables and GroupBy : pivot tables as essentially a *multidimensional* version of GroupBy aggregation. That is, you split-apply-combine, but both the split and the combine happen across not a one-dimensional index, but across a two-dimensional grid.

Pivot Table Syntax

Here is the equivalent to the preceding operation using the pivot_table method of DataFrames:

```
In [5]: titanic.pivot_table('survived', index='sex', columns='class')
```

Out[5]:

class sex	First	Second	Third
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

his is eminently more readable than the groupby approach, and produces the same result. As you might expect of an early 20th-century transatlantic cruise, the survival gradient favors both women and higher classes. First-class women survived with near certainty (hi, Rose!), while only one in ten third-class men survived (sorry, Jack!).

Multi-level pivot tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the pd.cut function:

```
In [6]: age = pd.cut(titanic['age'], [0, 18, 80])
        titanic.pivot_table('survived', ['sex', age], 'class')
```

Out[6]:

sex	class age	First	Second	Third
female	**(0, 18]**	0.909091	1.000000	0.511628
	(18, 80]	0.972973	0.900000	0.423729
male	**(0, 18]**	0.800000	0.600000	0.215686
	(18, 80]	0.375000	0.071429	0.133663

9	List and Explain in detail different ways of creating contour plots

A contour plot can be created with the `plt.contour` function. It takes three arguments: a grid of *x* values, a grid of *y* values, and a grid of *z* values. The *x* and *y* values represent positions on the plot, and the *z* values will be represented by the contour levels. Perhaps the most straightforward way to prepare such data is to use the `np.meshgrid` function, which builds two-dimensional grids from one-dimensional arrays:

```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

Now let's look at this with a standard line-only contour plot:

```
plt.contour(X, Y, Z, colors='black');
```

Notice that by default when a single color is used, negative values are represented by dashed lines, and positive values by solid lines. Alternatively, the lines can be color-coded by specifying a colormap with the `cmap` argument. Here, we'll also specify that we want more lines to be drawn—20 equally spaced intervals within the data range:

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```

Here we chose the `RdGy` (short for *Red-Gray*) colormap, which is a good choice for centered data. Matplotlib has a wide range of colormaps available, which you can easily browse in IPython by doing a tab completion on the `plt.cm` module:

```
plt.cm.<TAB>
```

Our plot is looking nicer, but the spaces between the lines may be a bit distracting. We can change this by switching to a filled contour plot using the `plt.contourf()` function (notice the `f` at the end), which uses largely the same syntax as `plt.contour()`.

Additionally, we'll add a `plt.colorbar()` command, which automatically creates an additional axis with labeled color information for the plot:

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```

The colorbar makes it clear that the black regions are "peaks," while the red regions are "valleys."

One potential issue with this plot is that it is a bit "splotchy." That is, the color steps are discrete rather than continuous, which is not always what is desired. This could be remedied by setting the number of contours to a very high number, but this results in a rather inefficient plot: Matplotlib must render a new polygon for each step in the level. A better way to handle this is to use the `plt.imshow()` function, which interprets a two-dimensional grid of data as an image.

The following code shows this:

```
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
```

```
            cmap='RdGy')
plt.colorbar()
plt.axis(aspect='image');
```

There are a few potential gotchas with `imshow()`, however:

- `plt.imshow()` doesn't accept an *x* and *y* grid, so you must manually specify the *extent* [*xmin*, *xmax*, *ymin*, *ymax*] of the image on the plot.
- `plt.imshow()` by default follows the standard image array definition where the origin is in the upper left, not in the lower left as in most contour plots. This must be changed when showing gridded data.
- `plt.imshow()` will automatically adjust the axis aspect ratio to match the input data; this can be changed by setting, for example, `plt.axis(aspect='image')` to make *x* and *y* units match.

Finally, it can sometimes be useful to combine contour plots and image plots. For example, here we'll use a partially transparent background image (with transparency set via the `alpha` parameter) and overplot contours with labels on the contours themselves (using the `plt.clabel()` function):

```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)

plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
            cmap='RdGy', alpha=0.5)
plt.colorbar();
```

The combination of these three functions—`plt.contour`, `plt.contourf`, and `plt.imshow`—gives nearly limitless possibilities for displaying this sort of three-dimensional data within a two-dimensional plot.

10 Write a Pandas program to create a data frame with the test data , split the dataframe by school code and get mean, min, and max value of i) age ii) weight for each school.

Test Data:

	school	class	name	age	height	weight
S1	s001	V	Ram	12	173	35
S2	s002	V	Kiran	12	192	32
S3	s003	VI	Ryan	13	186	33
S4	s001	VI	Bhim	13	167	30
S5	s002	VI	Sita	14	151	31
S6	s004	V	Bhavana	12	159	32

Solution

```
In [1]: import pandas as pd
student_data = pd.DataFrame({
    'school_code': ['s001','s002','s003','s001','s002','s004'],
    'class': ['I', 'III', 'III', 'I', 'V', 'V'],
    'name': ['Alberto Franco','Gino Mcneill','Ryan Parkes', 'Eesha Hinton', 'Gino Mcneill', 'David Parkes'],
    'date_Of_Birth ': ['15/05/2002','17/05/2002','16/02/1999','25/09/1998','11/05/2002','15/09/1997'],
    'age': [12, 12, 13, 13, 14, 12],
    'height': [173, 192, 186, 167, 151, 159],
    'weight': [35, 32, 33, 30, 31, 32],
    'address': ['street1', 'street2', 'street3', 'street1', 'street2', 'street4']},
    index=['S1', 'S2', 'S3', 'S4', 'S5', 'S6'])

print("Original DataFrame:")
print(student_data)
```

```
Original DataFrame:
   school_code class            name date_Of_Birth   age  height  weight  \
S1        s001     I  Alberto Franco    15/05/2002   12     173      35
S2        s002   III    Gino Mcneill    17/05/2002   12     192      32
S3        s003   III     Ryan Parkes    16/02/1999   13     186      33
S4        s001     I    Eesha Hinton    25/09/1998   13     167      30
S5        s002     V    Gino Mcneill    11/05/2002   14     151      31
S6        s004     V    David Parkes    15/09/1997   12     159      32

   address
S1  street1
S2  street2
S3  street3
S4  street1
S5  street2
S6  street4
```

In [2]:
```
print('\nMean, min, and max value of age for each value of the school:')
grouped_single = student_data.groupby('school_code').agg({'age': ['mean', 'min', 'max']})
print(grouped_single)
```

```
Mean, min, and max value of age for each value of the school:
             age
            mean min max
school_code
s001        12.5  12  13
s002        13.0  12  14
s003        13.0  13  13
s004        12.0  12  12
```

In [3]:
```
print('\nMean, min, and max value of weight for each value of the school:')
grouped_single = student_data.groupby('school_code').agg({'weight': ['mean', 'min', 'max']})
print(grouped_single)
```

```
Mean, min, and max value of weight for each value of the school:
           weight
             mean min max
school_code
s001         32.5  30  35
s002         31.5  31  32
s003         33.0  33  33
s004         32.0  32  32
```