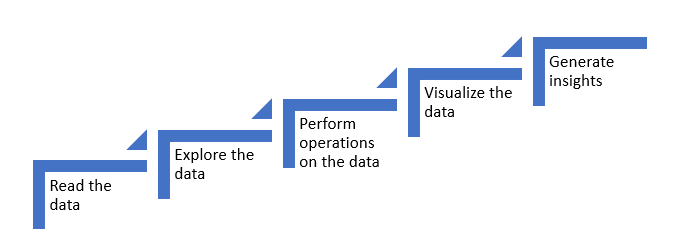
Introduction to Pandas

Pandas is an open-source library for real world data analysis in python. It is built on top of Numpy. Using Pandas, data can be cleaned, transformed, manipulated, and analyzed. It is suited for different kinds of data including tabular as in a SQL table or a Excel spreadsheets, time series data, observational or statistical datasets.

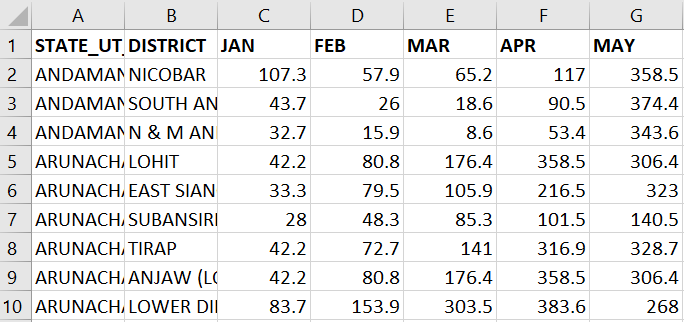
The steps involved to perform data analysis using Pandas are as follows:



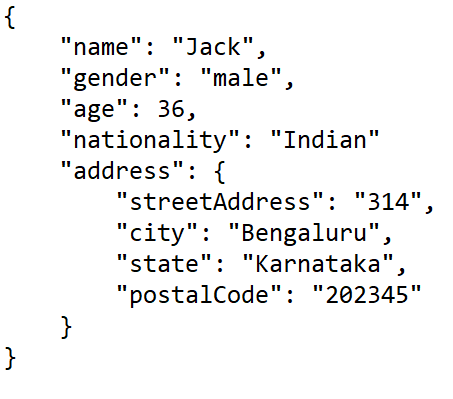
**Reading the data**

The first step is to read the data. There are multiple formats in which data can be obtained such as '.csv', '.json', '.xlsx' etc.

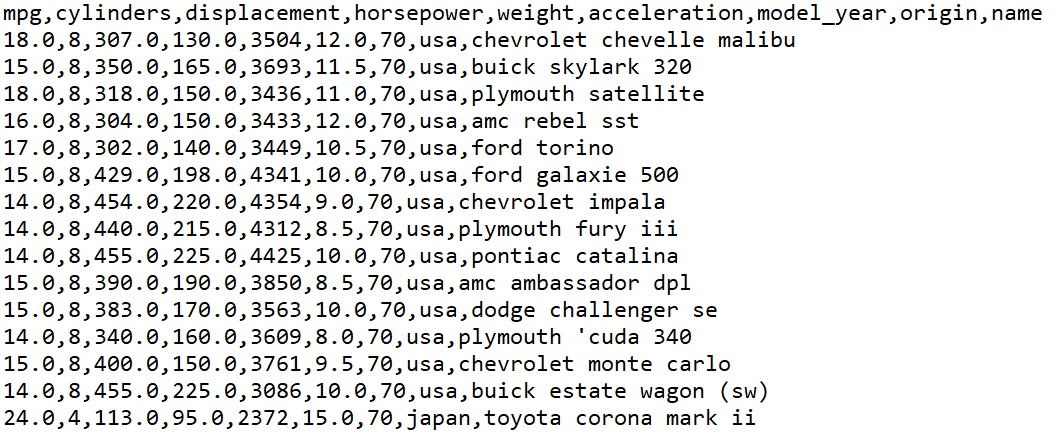
Below are the examples:

**Example of an excel file:** 

#### Example of a json (javascript object notation) file:



#### Example of a csv (comma separated values) file:



## Exploring the data

The next step is to explore the data. Exploring data helps to:

* know the shape(number of rows and columns) of the data
* understand the nature of the data by obtaining subsets of the data
* identify missing values and treat them accordingly
* get insights about the data using descriptive statistics

## Performing operations on the data

Some of the operations supported by pandas for data manipulation are as follows:

* Grouping operations
* Sorting operations
* Masking operations
* Merging operations
* Concatenating operations

## Visualizing data

The next step is to visualize the data to get a clear picture of various relationshipsamong the data. The following plots can help visualize the data:

* Scatter plot
* Box plot
* Bar plot
* Histogram and many more

## Generating Insights

All the above steps help generating insights about our data.

Pandas is one of the most popular data wrangling and analysis tools because it:

* has the capability to load huge sizes of data easily
* provides us with extremely streamlined forms of data representation
* can handle heterogenous data, has extensive set of data manipulation features and makes data flexible and customizable

Introduction to Pandas objects

To get started with Pandas, Numpy and Pandas needs to be imported as shown below:

1. *#Importing libraries*
2. *#python library for numerical and scientific computing. pandas is built on top of numpy*
3. import numpy as np
4. *#importing pandas*
5. import pandas as pd

In a nutshell, Pandas objects are advanced versions of NumPy structured arrays in which the rows and columns are identified with labels instead of simple integer indices.

The basic data structures of Pandas are Series and DataFrame.

Series is one dimensional labelled array. It supports different datatypes like integer, float, string etc. Let us understand more about series with the following example.

Consider the scenario where marks of students are given as shown in the following table:

|  |  |
| --- | --- |
| Student ID | Marks |
| 1 | 78 |
| 2 | 92 |
| 3 | 36 |
| 4 | 64 |
| 5 | 89 |

The pandas series object can be used to represent this data in a meaningful manner. Series is created using the following syntax:

**Syntax:**

**pd.Series(data, index, dtype)**

data – It can be a list, a list of lists or even a dictionary.

             index – The index can be explicitly defined for different valuesif required.

             dtype – This represents the data type used in the series (optional parameter).

1. series = pd.Series(data = [78, 92, 36, 64, 89])
2. series

As shown in the above output, the series object provides the values along with their index attributes.

**Series.values** provides the values.

1. series.values

**Series.index** provides the index.

1. series.index

### Accessing data in series:

Data can be accessed by the associated index using [ ].

1. series[1]

### Slicing a series:

1. series[1:3]

By default, series creates an integer index. The custom index can also be defined.

For example, consider the following table containing car details:

|  |  |
| --- | --- |
| Car Name | Car Price |
| Swift | 700000 |
| Jazz | 800000 |
| Civic | 1600000 |
| Altis | 1800000 |
| Gallardo | 30000000 |

A Pandas series can be created using the following syntax:

1. data = pd.Series(data = [700000, 800000, 1600000, 1800000, 30000000], index = ['Swift', 'Jazz', 'Civic', 'Altis', 'Gallardo'])
2. data

Values can be accessed as:

1. data['Swift']
2. data['Jazz': 'Gallardo']

In this case, observations are that the output starts from Jazz and goes till Gallardo(inclusive). This is the fundamental difference between implicit and explicit indexing.

**Series can also be viewed as a specialized dictionary where the keys act as index and corresponding values act as values.**

Let us create a series out of the dictionary data structure.

1. *#Using dictionary to create a series*
2. car\_price\_dict = {'Swift': 700000,
3. 'Jazz' : 800000,
4. 'Civic' : 1600000,
5. 'Altis' : 1800000,
6. 'Gallardo': 30000000
7. }
8. car\_price = pd.Series(car\_price\_dict)
9. car\_price

A series gives a useful way to view and manipulate one dimensional data. But when data is present in rows and columns, it becomes necessary to make use of the Pandas DataFrame object. A DataFrame is a collection of series where each series represents a column from a table.

For example, consider the following table containing car details:

|  |  |  |
| --- | --- | --- |
| Car Name | Car Price | Car Manufacturer |
| Swift | 700000 | Maruti |
| Jazz | 800000 | Honda |
| Civic | 1600000 | Honda |
| Altis | 1800000 | Toyota |
| Gallardo | 30000000 | Lamborghini |

Let us create two series from two dictionaries - one containing car name and price and the other with car name and manufacturer.

1. *#Creating a car price series with a dictionary*
2. car\_price\_dict = {'Swift': 700000,
3. 'Jazz' : 800000,
4. 'Civic' : 1600000,
5. 'Altis' : 1800000,
6. 'Gallardo': 30000000
7. }
8. car\_price = pd.Series(car\_price\_dict)
9. *# Creating the car manufacturer series with a dictionary*
10. car\_man\_dict = {'Swift' : 'Maruti',
11. 'Jazz' : 'Honda',
12. 'Civic' : 'Honda',
13. 'Altis' : 'Toyota',
14. 'Gallardo' : 'Lamborghini'}
15. car\_man = pd.Series(car\_man\_dict)
16. print(car\_price)
17. print(car\_man)

Let us create a Dataframe object using the series objects as shown below:

**Syntax:**

**pd.DataFrame(data, index, columns)**

data - data can contain Series or list-like objects. If data is a dictionary, column order follows insertion-order.

               index- index for dataframe that is created. By default, it will be RangeIndex(0, 1, 2, …, n) if no explicit index is provided

               columns-  If data contains column labels, it will use the same . Else, default to RangeIndex(0, 1, 2, …, n).

1. cars = pd.DataFrame({'Price': car\_price , 'Manufacturer' : car\_man})
2. cars

The output shows the Dataframe containing multiple columns. The car names act as the indices and ‘Price’ and ‘Manufacturer’ act as the columns or 'features' of this small dataset.

To access individual features, the following code can be used:

1. cars['Price']
2. cars['Manufacturer']

There are different approaches to create a DataFrame such as:

### ****1. From a single series object****

A DataFrame is a collection of Series objects, and a single-column DataFrame can be constructed from a single Series:

1. *#Using dictionary to create a series*
2. car\_price\_dict = {'Swift': 700000,
3. 'Jazz' : 800000,
4. 'Civic' : 1600000,
5. 'Altis' : 1800000,
6. 'Gallardo': 30000000
7. }
8. car\_price = pd.Series(car\_price\_dict)
9. car\_price
10. *#Creating a DataFrame from car\_price Series*
11. pd.DataFrame(car\_price, columns=['Car Price'])

### ****2. From a list of dictionaries****

Consider the following data of marks for four students.

|  |  |
| --- | --- |
| Name | Marks |
| Subodh | 28 |
| Ram | 27 |
| Abdul | 26 |
| John | 28 |

Following list of dictionaries can be used:

1. data = [{'Name': 'Subodh', 'Marks': 28},
2. {'Name': 'Ram', 'Marks': 27},
3. {'Name': 'Abdul', 'Marks': 26},
4. {'Name': 'John', 'Marks': 28}]
5. pd.DataFrame(data)

**Suppose there is a following table to be represented as a dataframe ?**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Subject** | **Subodh** | **Ram** | **Abdul** | **John** |
| **Mathematics** | 20 | 25 | **Not appeared** | **Not appeared** |
| **Physics** | **Not appeared** | **Not Appeared** | 29 | 24 |

1. pd.DataFrame([{'Subodh':20, 'Ram':25},
2. {'Abdul':29, 'John':24}],
3. index = ['Mathematics', 'Physics'])

Each dictionary element in the list is taken as a row . Index is representing different subjects.

Note: NaN(Not a Number) represents missing values.

### ****3. From a dictionary of series objects****

A DataFrame can be constructed from a dictionary of Series objects:

1. *#Using dictionary to create a series*
2. car\_price\_dict = {'Swift': 700000,
3. 'Jazz' : 800000,
4. 'Civic' : 1600000,
5. 'Altis' : 1800000,
6. 'Gallardo': 30000000
7. }
8. car\_price = pd.Series(car\_price\_dict)
9. car\_man\_dict = {'Swift' : 'Maruti',
10. 'Jazz' : 'Honda',
11. 'Civic' : 'Honda',
12. 'Altis' : 'Toyota',
13. 'Gallardo' : 'Lamborghini'}
14. car\_man = pd.Series(car\_man\_dict)
15. cars = pd.DataFrame({'Price': car\_price , 'Manufacturer' : car\_man})
16. cars

### ****4. From an existing file****

In most real world scenarios, the data is in different file formats like csv, xlsx, json etc. Pandas supports reading the data from these files. Below is an example of creating a DataFrame from a json file.

Click [here](https://lex.infosysapps.com/apis/authContent/content-store/Infosys/Infosys_Ltd/Public/lex_auth_013242329180684288492/web-hosted/assets/example1626702607322.json) to download the json file used in the demo.

1. data\_json = pd.read\_json('example.json',)
2. data\_json

### The axis keyword

One of the important parameters used while performing operations on DataFrames is 'axis'. Axis takes two values: 0 and 1.

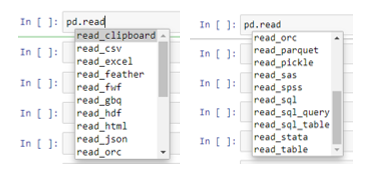
axis = 0 represents row specific operations.

axis = 1 represents column specific operations.

Working with datasets

### Reading the data from XYZ custom cars

Pandas can read a variety of files. For example, a table of fixed width formatted lines (read\_fwf), excel sheets (read\_excel), html files (read\_html), json files (read\_json) etc.



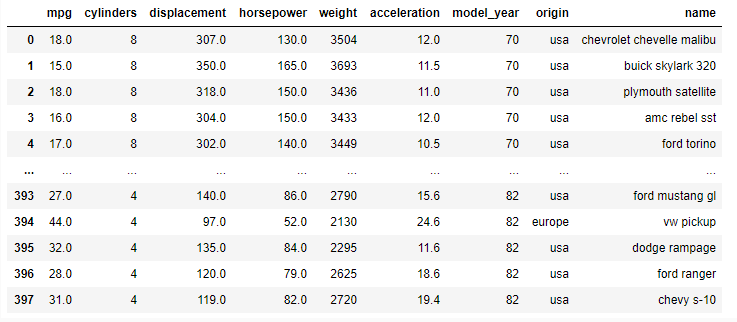
XYZ custom cars data is given in a csv format. This data is imported to a pandas DataFrame as shown below.

**Syntax:**

**pd.read\_csv(filepath)**

 filepath - storage path of the file

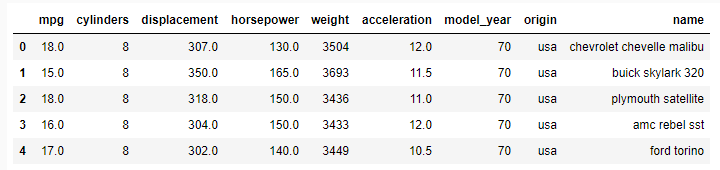
1. import pandas as pd
2. import numpy as np
3. df = pd.read\_csv('auto\_mpg.csv')
4. df



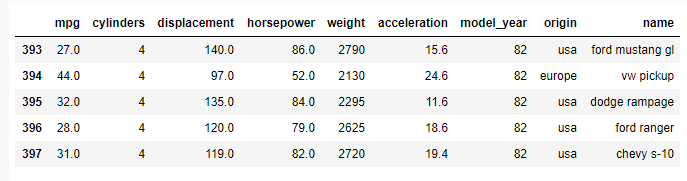
**1. Head and Tail**

To view the first few rows or the last few rows, the functions that can be used are: df.head() and df.tail() respectively. If the number of rows to be viewed is not passed, then, the head and tail functions provides five rows by default.

1. df.head()



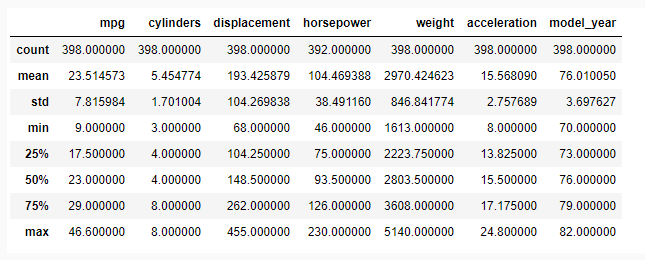
df.tail()



**2. Describe**

The describe function can be used to generate a quick summary of data statistics.

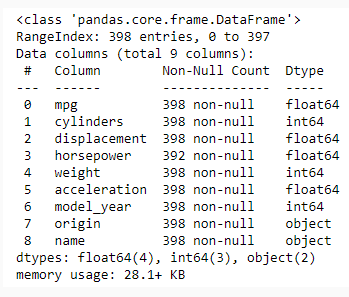
1. df.describe()



**3. Info**

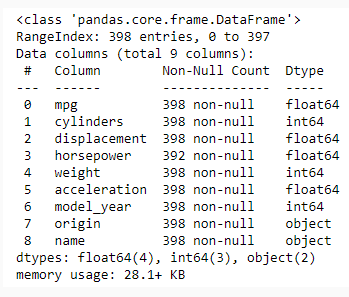
To know about the datatypes and number of rows containing null values for respective columns, the info() function can be used.

1. df.info()

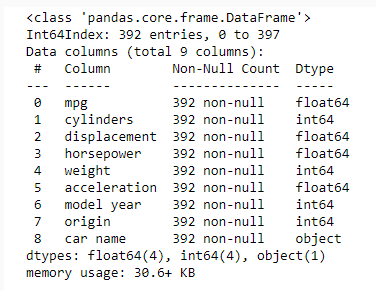


**4. Dropping null values**

It can be observed that the ‘horsepower’ attribute has some null values. The easiest approach is to remove the rows with any null values. This can be achieved using dropna() function.



1. df.dropna(inplace = True)
2. df.info()



After dropping the rows with null horsepower values, it can be observed that the number of rows has been reduced to 392.

**Note:**

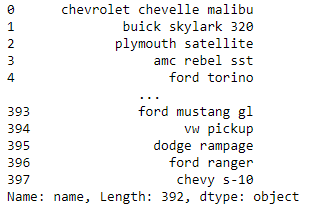
* 'inplace' makes changes to the original DataFrame.
* df.fillna(condition) can be used to fill all the missing values. The missing values are filled with mean, median, mode, or constant values.

**5. Selecting a subset of the data**

In addition to data access techniques, pandas also provides techniques for indexing and selection. Selecting a specific column in a DataFrame can be achieved in following ways:

* Passing the column name as shown below:

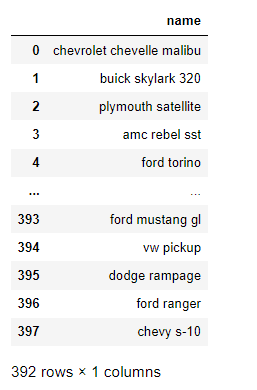
1. df['name']



Output is a Series containing car names.

* Passing the column name as a list as shown below:

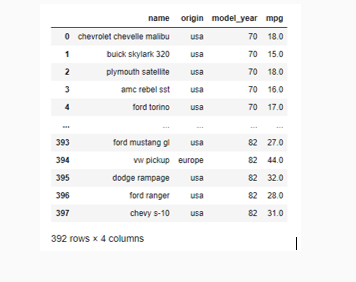
1. df[['name']]



Output is a DataFrame containing just one column.

* To extract the subset of the data, we can pass the column names in a list as shown below:

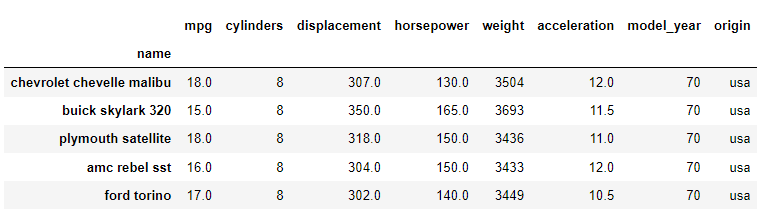
1. df[['name', 'origin', 'model\_year', 'mpg']]



## Setting custom index:

Custom index can be set to the DataFrame according to the requirements. The following example depicts the same:

1. *#creating a subset using head*
2. df\_head = df.head()
3. *#Setting name as custom index*
4. df\_head.set\_index('name', inplace = True)
5. df\_head



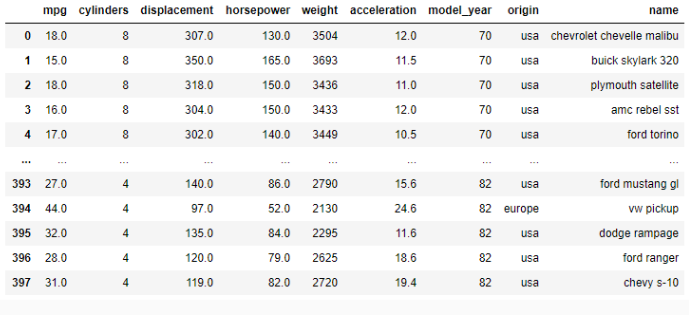
'iloc' and 'loc' are the two indexing techniques that help us in selecting specific rows and columns.

**1.      iloc- Access a group of rows and columns by integer index.**

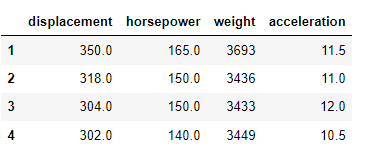
The ‘iloc’ indexer follows implicit index.

**Syntax - df.iloc[Rows, Columns]**

In the following demos, 'df' refers to XYZ Custom Cars DataFrame.



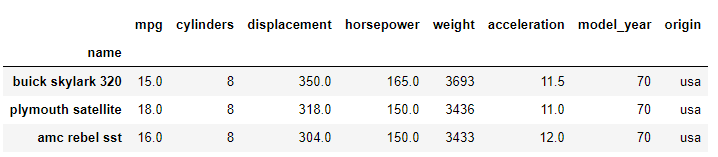
1. df.iloc[2,1]
2. df.iloc[2,-1]
3. df.iloc[1:5, 4:6]



**2.      loc- Access a group of rows and columns by custom index.**

The loc indexer follows explicit indexing.

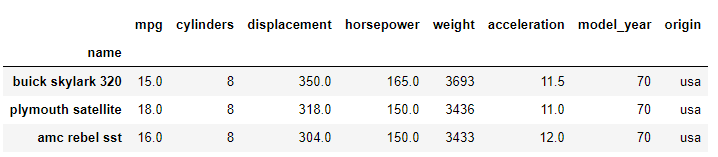
1. *#creating a subset using head. 'df' refers to XYZ Custom Cars DataFrame.*
2. df\_head = df.head()
3. *#Setting name as custom index*
4. df\_head.set\_index('name', inplace = True)
5. df\_head.loc['buick skylark 320': 'amc rebel sst']



To select a subset of columns, the column names can be passed as a list.

**Note:** While retrieving records using loc, the upper range of slice is inclusive.

1. *#Subsetting from the full dataset*
2. df.loc[0:5, ['cylinders', 'horsepower', 'name']]

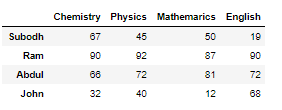


Consider the following table:

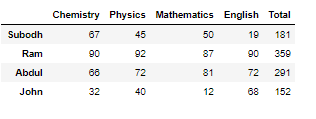
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Students | Chemistry | Physics | Mathematics | English |
| Subodh | 67 | 45 | 50 | 19 |
| Ram | 90 | 92 | 87 | 90 |
| Abdul | 66 | 72 | 81 | 72 |
| John | 32 | 40 | 12 | 68 |

The teacher wants to insert a ‘Total marks’ column which gives the sum of marks of all subjects.

1. marks = {'Chemistry': [67,90,66,32],
2. 'Physics': [45,92,72,40],
3. 'Mathematics': [50,87,81,12],
4. 'English': [19,90,72,68]}
5. marks\_df = pd.DataFrame(marks, index = ['Subodh', 'Ram', 'Abdul', 'John'])
6. marks\_df

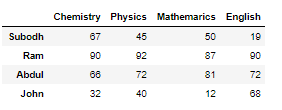


1. marks\_df['Total'] = marks\_df['Chemistry'] + marks\_df['Physics'] + marks\_df['Mathematics'] + marks\_df['English']
2. marks\_df



To drop a feature:

1. marks\_df.drop(columns = 'Total', inplace = True)

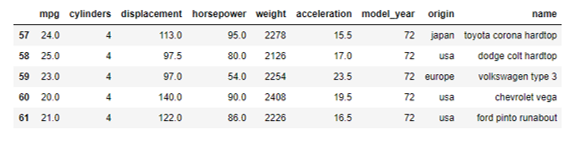


Operations in Pandas

### Problem statement : Retrieve details of all the cars built in year 72.

### Solution:

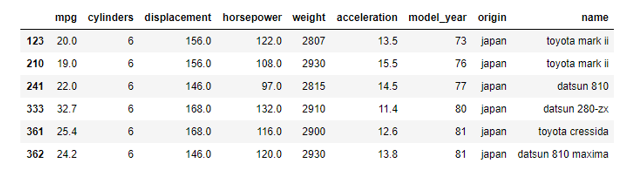
1. df.loc[df['model\_year'] == 72 ].head()



### Problem statement : Retrieve details of all the cars built in Japan having 6 cylinders

### Solution:

1. df.loc[(df['origin'] == 'japan') & (df['cylinders'] == 6)]



### Problem Statement:

XYZ Custom Cars want to categorize cars in different categories as follows:

|  |  |  |
| --- | --- | --- |
| Category | Description | Features coming in play |
| Fuel efficient | Cars designed with low power and high fuel efficiency | High MPG, Low Horsepower, Low weight |
| Muscle Cars | Intermediate sized cars designed for high performance | High displacement, High horsepower, Moderate weight |
| SUV | Big sized cars designed for high performance, long distance trips and family comfort | High horsepower, High weight |
| Racecar | Cars specifically designed for race tracks | Low weight, High acceleration |

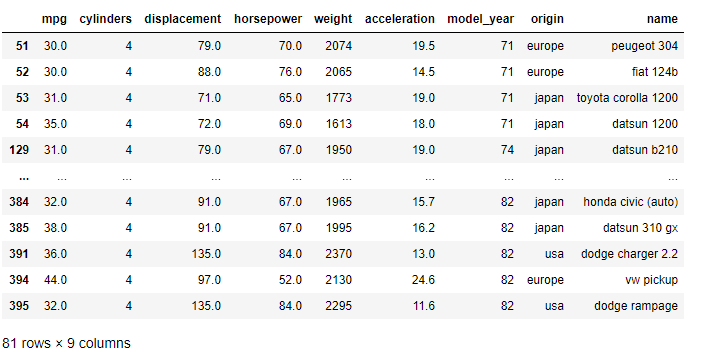
### Solution:

Their experienced engineers and mechanics have come up with the following parameters for these categories-

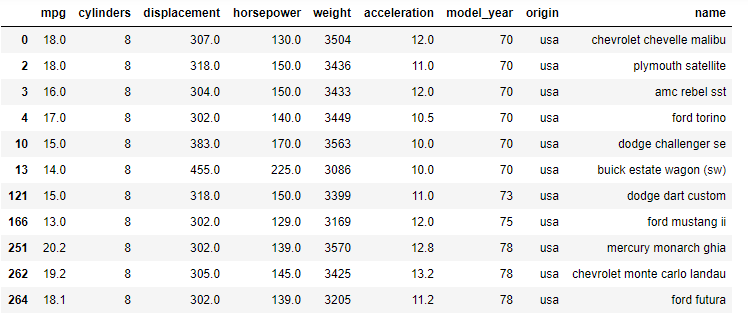
|  |  |  |
| --- | --- | --- |
| Category | Description | Features involved |
| Fuel efficient | Cars designed with low power and high fuel efficiency | MPG > 29, Horsepower < 93.5,  Weight < 2500 |
| Muscle Cars | Intermediate sized cars designed for high performance | Displacement >262, Horsepower > 126, Weight in range[2800, 3600] |
| SUV | Big sized cars designed for high performance, long distance trips and family comfort | Horsepower > 140 , Weight > 4500 |
| Racecar | Cars specifically designed for race tracks | Weight <2223, acceleration > 17 |

Let us see how we can find out the cars belonging to these categories based on the given parameters.

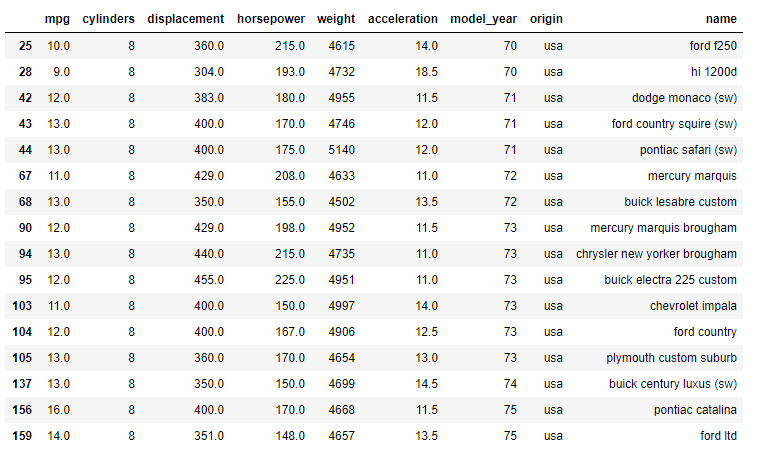
1. *# Fuel efficient*
2. *# MPG > 29, Horsepower < 93.5,*
3. *# Weight < 2500*
4. df.loc[(df['mpg'] > 29) & (df['horsepower'] < 93.5) & (df['weight'] < 2500)]



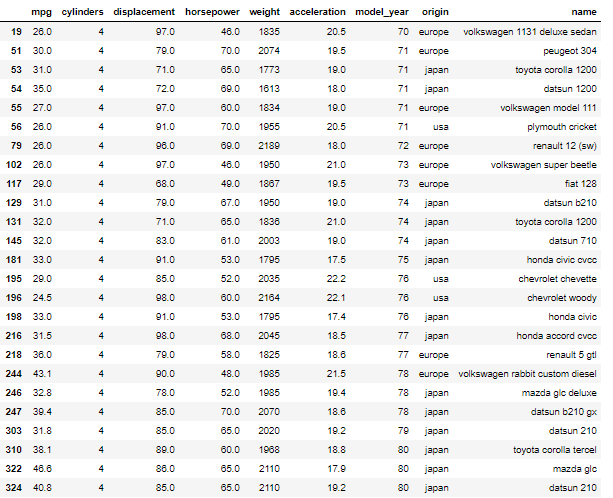
1. *# Muscle cars*
2. *# Displacement >262, Horsepower > 126, Weight in range[2800, 3600]*
3. df.loc[(df['displacement'] > 262) & (df['horsepower'] > 126) & (df['weight'] >=2800) & (df['weight'] <= 3600)]



1. *# SUV*
2. *# Horsepower > 140 , Weight > 4500*
3. df.loc[(df['horsepower'] > 140) & (df['weight'] >=4500)]



1. *# Racecar*
2. *# Weight <2223, acceleration > 17*
3. df.loc[(df['acceleration'] > 17) & (df['weight'] < 2223)]



The masking operation replaces values where the condition is True.

Consider the below table with student marks:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Students | Chemistry | Physics | Mathematics | English |
| Subodh | 67 | 45 | 50 | 19 |
| Ram | 90 | 92 | 87 | 90 |
| Abdul | 66 | 72 | 81 | 72 |
| John | 32 | 40 | 12 | 68 |

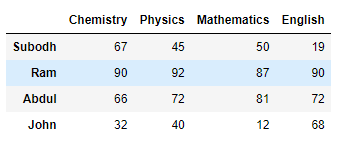
### Problem Statement:

The teacher does not want to reveal the marks of students who have failed. The condition is that if a student has scored marks >= 33, then they have passed, otherwise failed. The marks of failed students has to be replaced with ‘Fail’. So, how can the task be performed?.

### Solution:

First is to create the DataFrame as shown below:

1. marks = [{'Chemistry': 67, 'Physics': 45, 'Mathematics': 50, 'English' : 19},
2. {'Chemistry': 90, 'Physics': 92, 'Mathematics': 87, 'English' : 90},
3. {'Chemistry': 66, 'Physics': 72, 'Mathematics': 81, 'English' : 72},
4. {'Chemistry': 32, 'Physics': 40, 'Mathematics': 12, 'English' : 68}]
5. marks\_df = pd.DataFrame(marks, index = ['Subodh', 'Ram', 'Abdul', 'John'])
6. marks\_df



**Syntax:**

**DataFrame.mask(cond, other = nan, inplace = False, axis = None)**

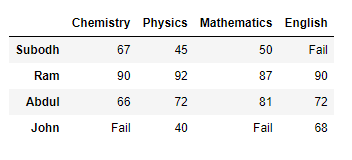
       cond – Where cond is False, keep the original value. Where True, replace with corresponding value from other

       other - Entries where cond is True are replaced with corresponding value from other.

       inplace - Whether to perform the operation in place on the data.

       axis – alignment axis

1. f = marks\_df < 33
2. marks\_df.mask(f, 'Fail')



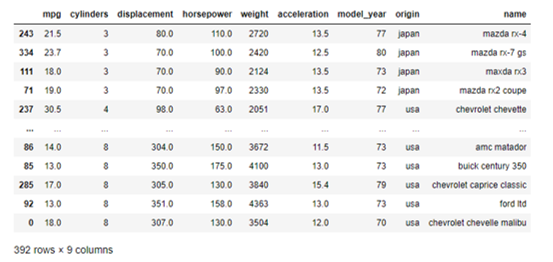
### Problem Statement:

XYZ Custom cars want the data sorted according to the number of cylinders.

### Solution:

The following method can be used to get the solution.

1. df.sort\_values(by = 'cylinders')



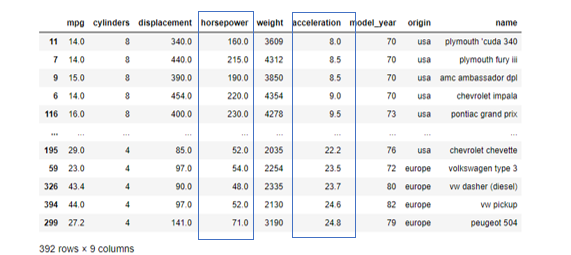
### Problem Statement:

There is a requirement in which the cars that have lowest acceleration must be assessed. It is also to be checked that which cars have higher horsepower despite having lower acceleration.

### Solution:

In this case, the data must be sorted in ascending order of acceleration and descending order of horsepower as follows:

1. df.sort\_values(['acceleration', 'horsepower'], ascending = (1,0))



Pandas preserves the index and column labels in the output. For binary operations such as addition and multiplication, Pandas will automatically align indices when passing the objects to the functions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Students | Chemistry | Physics | Mathematics | English |
| Subodh | 67 | 45 | 50 | 19 |
| Ram | 90 | 92 | 87 | 90 |
| Abdul | 66 | 72 | 81 | 72 |
| John | 32 | 40 | 12 | 68 |

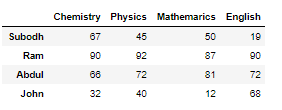
### Problem Statement:

The teacher wants to encrypt the marks for confidential reasons. Therefore, the teacher decides to save the marks as sine of the original marks. For example, if Subodh has scored 67 in chemistry, then his encrypted marks will be sin(67) = -0.855520

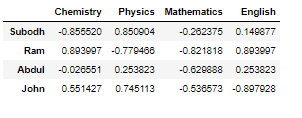
### Solution:

To get the solution, following code must be used:

1. marks = {'Chemistry': [67,90,66,32],
2. 'Physics': [45,92,72,40],
3. 'Mathematics': [50,87,81,12],
4. 'English': [19,90,72,68]}
5. marks\_df = pd.DataFrame(marks, index = ['Subodh', 'Ram', 'Abdul', 'John'])
6. marks\_df



1. #encrypting marks as sine of marks
2. encrypted\_marks = np.sin(marks\_df)
3. encrypted\_marks

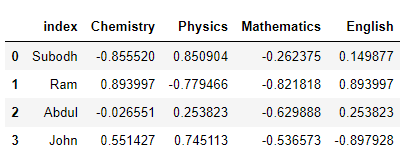


The  encrypted marks are with same indices as the original marks. This is called as index preservation.

### Resetting Index:

In case of a requirement where the index has to be restored to the default index, reset\_index() function must be used. It adds the existing index as a new column in the DataFrame. This can be done as follows:

1. encrypted\_marks.reset\_index(inplace = True)
2. encrypted\_marks



Broadcasting refers to a set of rules to operate between data of different sizes and shapes.

Consider the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Students | Chemistry | Physics | Mathematics | English |
| Subodh | 67 | 45 | 50 | 19 |
| Ram | 90 | 92 | 87 | 90 |
| Abdul | 66 | 72 | 81 | 72 |
| John | 32 | 40 | 12 | 68 |

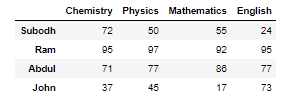
### Problem Statement:

The teacher wants to award five bonus marks to all the students.

### Solution:

This can be done by using broadcasting methods available in Pandas.

1. new\_marks = marks\_df + 5
2. new\_marks



### Problem Statement:

The teacher wants to increase the marks of all the students as follows-

Chemistry: + 5

Physics: + 10

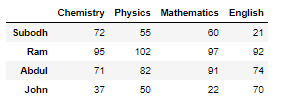
Mathematics: +10

English: + 2

### Solution:

This can be done as follows:

1. new\_marks = marks\_df + [5,10,10,2]
2. new\_marks



# Apply

This method is used to apply a function along an axis of the DataFrame.

**Syntax:**

**DataFrame.apply(func, axis = 0, result\_type = None)**

       func : Function to apply to each column or row.

       axis: Axis along which the function is applied.

       result\_type: one out of 'expand', 'reduce' or 'broadcast'. In the demo, 'broadcast' is used.

* ‘broadcast’ : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Students | Chemistry | Physics | Mathematics | English |
| James | 67 | 45 | 50 | 19 |
| Lee | 90 | 92 | 87 | 90 |
| Anderson | 66 | 72 | 81 | 72 |
| John | 32 | 40 | 12 | 68 |

1. *#Creating the DataFrame*
2. marks = {'Chemistry': [67,90,66,32],
3. 'Physics': [45,92,72,40],
4. 'Mathematics': [50,87,81,12],
5. 'English': [19,90,72,68]}
6. marks\_df = pd.DataFrame(marks, index = ['James', 'Lee', 'Anderson', 'John'])

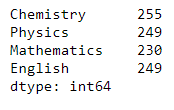
### Problem statement :

The teacher wants to get the total marks scored in each subject

### Solution :

This can be done as follows:

1. marks\_df.apply(np.sum, axis = 0)



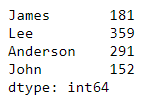
### Problem statement:

The teacher wants to get the total marks scored by each student.

### Solution:

This can be done as follows:

1. marks\_df.apply(np.sum, axis = 1)



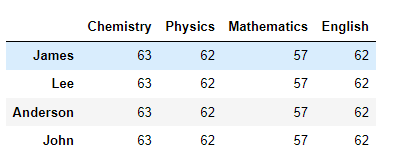
### Problem Statement:

The students were unable to attend the next set of exams due to the pandemic. Hence, the teacher decides to award them average marks based on their previous performance.

### Solution:

This can be done as follows:

1. marks\_df.apply(func = np.mean, axis = 0, result\_type = 'broadcast')



### Problem Statement:

Consider the scenario where the board of XYZ custom cars wants to know about minimum and maximum of all the numerical columns.

### Solution:

Aggregation operation is used to aggregate using one or more operations over the specified axis.

**Syntax:**

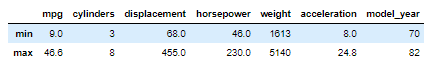
**DataFrame.agg(func, axis = 0)**

func - Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when  passed to DataFrame.apply.

axis: If 0 or ‘index’: apply function to each column. If 1 or ‘columns’: apply function to each row.

The below code is used to find minimum and maximum values of the numerical attributes:

1. *#Using list comprehension to get the numerical columns*
2. list1 = [col for col in df.columns if df[col].dtype in ['float', 'int64']]
3. df[list1].agg(['min', 'max'])



XYZ custom cars want to know the number of cars manufactured in each year.

This would require a grouping operation. Pandas supports a group by feature to group our data for aggregate operations.

**Syntax:**

**DataFrame.groupby(by = column\_name, axis, sort)**

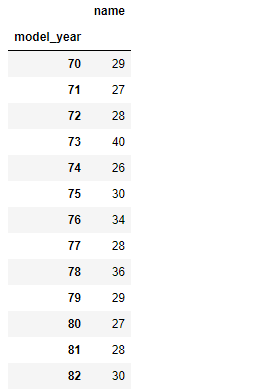
### Problem statement:

How many cars belong to each year?

### Solution:

In addition to the groupby function, the count function can be used as shown below. Since, cars are counted by names in each model year, the ‘name’ column in a list is used to get the output as a DataFrame.

1. df.groupby(['model\_year']).count()[['name']]



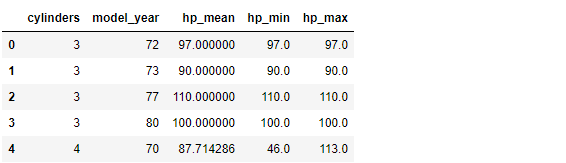
### Problem Statement:

Some senior engineers in XYZ custom cars want to understand about the effect of model year and number of cylinders on horsepower.

### Solution:

One of the engineers suggests about checking the mean, minimum and maximum horsepower based on number of cylinders and model year. For such requirement, the ‘agg’ function can be combined with groupby function as shown below:

1. *#Creating a DataFrame grouped on cylinders and model\_year and finding mean, min and max of horsepower*
2. grouped\_multiple = df.groupby(['cylinders', 'model\_year']).agg({'horsepower': ['mean', 'min', 'max']})
3. *#Naming columns in grouped DataFrame*
4. grouped\_multiple.columns = ['hp\_mean', 'hp\_min', 'hp\_max']
5. *#Resetting index*
6. grouped\_multiple = grouped\_multiple.reset\_index()
7. *#Viewing head of resulting DataFrame*
8. grouped\_multiple.head()



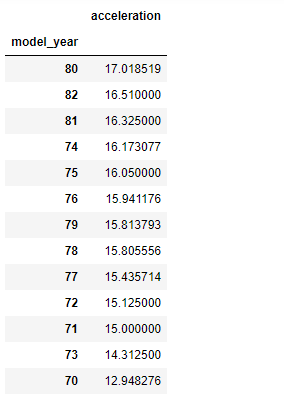
### Problem statement:

The engineers at XYZ Custom Cars want to know about the relationship between model year and acceleration of cars.

### Solution:

For better understanding, the grouped results can be sorted based on average acceleration of cars built in each model year.

1. df.groupby(['model\_year']).mean().sort\_values('acceleration', ascending = False)[['acceleration']]



Consider the following tables of student marks belonging to different sections.

#### Section A:

|  |  |  |
| --- | --- | --- |
| Students | Chemistry | Physics |
| Subodh | 67 | 45 |
| Ram | 90 | 92 |
| Abdul | 66 | 72 |
| John | 32 | 40 |

#### Section B:

|  |  |  |
| --- | --- | --- |
| Students | Chemistry | Physics |
| Nandini | 72 | 78 |
| Zoya | 45 | 34 |
| Shivam | 60 | 72 |
| James | 98 | 95 |

1. marks\_A = {'Chemistry': [67,90,66,32],
2. 'Physics': [45,92,72,40],
3. }
4. marks\_A\_df = pd.DataFrame(marks\_A, index = ['Subodh', 'Ram', 'Abdul', 'John'])
5. marks\_B = {'Chemistry': [72,45,60,98],
6. 'Physics': [78,34,72,95],
7. }
8. marks\_B\_df = pd.DataFrame(marks\_B, index = ['Nandini', 'Zoya', 'Shivam', 'James'])

### Problem Statement:

The teacher wants to combine the marks of these students.

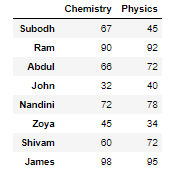
### Solution:

Using concatenation to combine the marks-

**Syntax:**

**pd.concat(data1, data2, sort)**

1. pd.concat([marks\_A\_df,marks\_B\_df], sort = False)

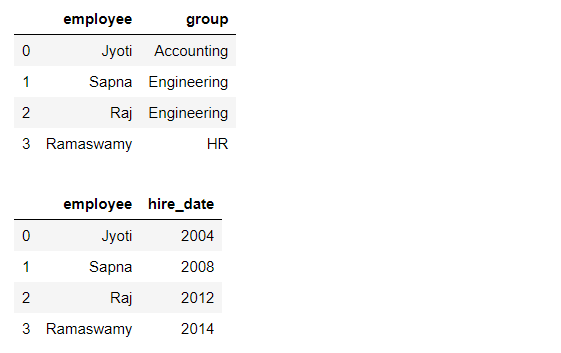


Now consider the following two tables.

|  |  |
| --- | --- |
| Employee | Group |
| Jyoti | Accounting |
| Sapna | Engineering |
| Raj | Engineering |
| Ramaswamy | HR |

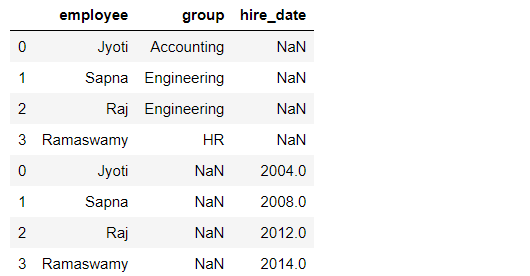
|  |  |
| --- | --- |
| Employee | Hire\_Date |
| Jyoti | 2004 |
| Sapna | 2008 |
| Raj | 2012 |
| Ramaswamy | 2014 |

1. df1 = pd.DataFrame({'employee': ['Jyoti', 'Sapna', 'Raj', 'Ramaswamy'],
2. 'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
3. df2 = pd.DataFrame({'employee': ['Jyoti', 'Sapna', 'Raj', 'Ramaswamy'],
4. 'hire\_date': [2004, 2008, 2012, 2014]})
5. display(df1,df2)



In this case, trying to concatenate the two tables will result in some null values because of column mismatch.

1. pd.concat([df1,df2], sort = False)



As one can observe, the NaN values are not giving any concrete information. Hence in this case, the concat function does not work effectively.

### Using Merge in case of column mismatch

To resolve the above condition, the merge function can be used which joins two tables based on a key.

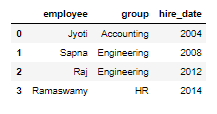
**Syntax:**

**pd.merge(data1, data2, how = 'inner')**

|  |  |
| --- | --- |
| Employee | Group |
| Jyoti | Accounting |
| Sapna | Engineering |
| Raj | Engineering |
| Ramaswamy | HR |

|  |  |
| --- | --- |
| Employee | Hire\_Date |
| Jyoti | 2004 |
| Sapna | 2008 |
| Raj | 2012 |
| Ramaswamy | 2014 |

1. df3 = pd.merge(df1,df2)
2. df3



In Pandas, the merge keyword automatically performs the inner join. For other types of joins, the 'how' parameter must be specified.

Note: Refer to the Pandas documentation for exploring more functionalities of merge() and read about functions like append().

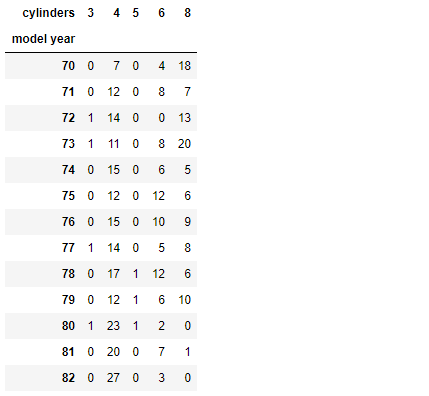
### Problem statement:

The engineers at XYZ Custom Cars want to know the frequency distribution of different number of cylinders across different years.

### Solution:

For such given condition, cross tab is used. It gives us a tabular representation of the frequency distribution.

1. pd.crosstab(df['model\_year'], df['cylinders'])



A Pivot Table is used to summarise, sort, reorganise, group, count, total or average data stored in a table. If we want to create spreadsheet-style pivot table as a data frame, pandas provides us with an option.

### Problem Statement:

The engineers at XYZ custom cars want to know the mean of all the numerical attributes of cars for each year

### Solution:

We can use a pivot table for this as follows:

**Syntax :**

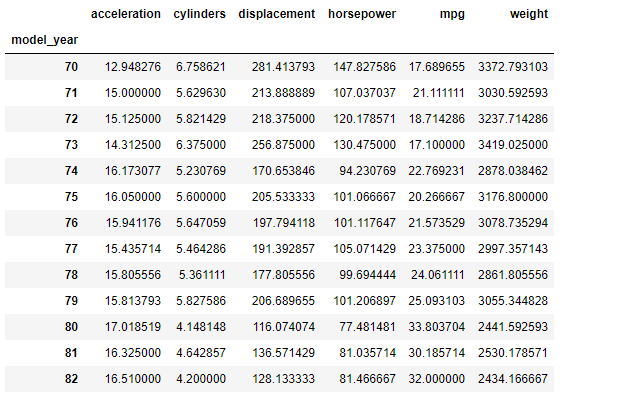
**pd.pivot\_table(data, index, aggfunc)**

      data: DataFrame

       index: column to be set as index

       aggfunc: function/list of functions, default = numpy.mean

1. pivot1 = pd.pivot\_table(df, index = 'model\_year', aggfunc=np.mean)
2. pivot1



Pandas Plots

Pandas also provides us options to visualize the data. Here are some of the examples:

**Syntax:**

**df.plot(X, y, marker, kind)**

       X = value on X axis

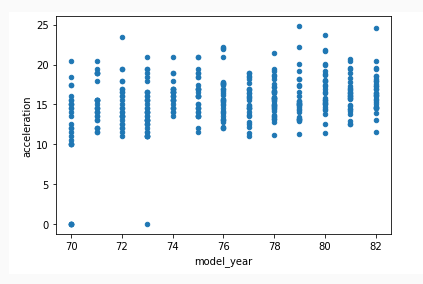
       y = value on y axis

       marker = shape in case of specific plots like a scatter plot

       kind = type of plot

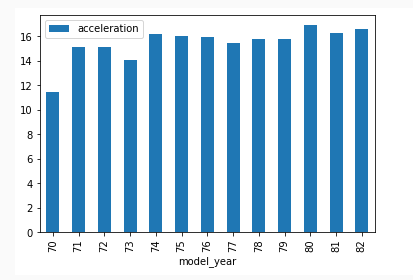
A scatter plot to visualize the trend of acceleration in different years.

1. df.plot(x = 'model\_year', y = 'acceleration', marker = 'o', kind = 'scatter');



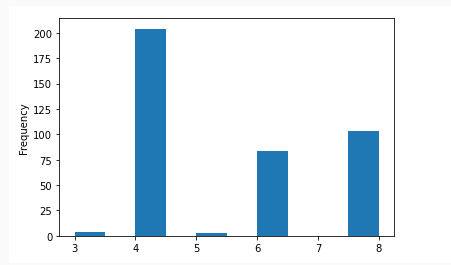
**A bar plot to visualize mean acceleration in different years.**

1. df.groupby('model\_year').mean()[['acceleration']].plot(kind = 'bar');



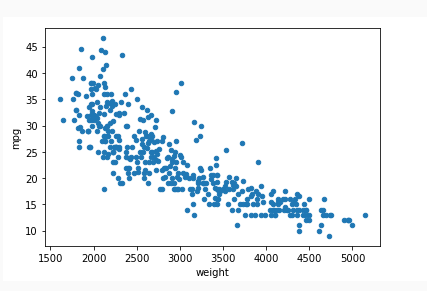
**A histogram to visualize the frequency distribution of cylinders**

1. df['cylinders'].plot(kind = 'hist')



**A scatter plot to visualize the relationship between weight and mpg.**

1. df.plot(x = 'weight', y = 'mpg', kind = 'scatter')



**A bar plot to visualize the sorted mean values of acceleration with respect to number of cylinders.**

1. df.groupby('cylinders').mean().sort\_values('acceleration')[['acceleration']].plot(kind = 'bar')

