Data Wrangling

Agenda

In this session, we will cover the following concepts with the help of a business use case:

- Data acquisition
- Different methods for data wrangling:
  - Merge datasets
  - Concatenate datasets
  - Identify unique values
  - Drop unnecessary columns
  - Check the dimension of the dataset
  - Check the datatype of the dataset
  - Check datatype summary
  - Treat missing values
  - Validate correctness of the data in primary level if applicable

What Is Data Wrangling?

Data wrangling is the process of converting and formating data from its raw form to usable format further down the data science pipeline.

What Is the Need for Data Wrangling?

Without feeding proper data into a model, one cannot expect a model that is dependable and gives higher accuracy.

Problem Statement

You are a junior data scientist and you are assigned a new task to perform data wrangling on a set of datasets. The datasets have many ambiguities. You have to identify those and apply different data wrangling techniques to get a dataset for further usage.

Dataset

- Download the `dataset_1 and dataset_2` from Course Resources and upload the datasets to the lab

Data Dictionary

Attribute Information:

- date = date of the ride
- season - 1 = spring, 2 = summer, 3 = fall, 4 = winter
- holiday - whether the day is considered a holiday
- workingday - whether the day is neither a weekend nor holiday
- weather - 1: Clear, Few clouds, Partly cloudy, Partly cloudy

```
        2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
        3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattere
d clouds
        4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
```

- temp - temperature in Celsius
- atemp - "feels like" temperature in Celsius
- humidity - relative humidity
- windspeed - wind speed
- casual - number of non-registered user rentals initiated

- registered - number of registered user rentals initiated
- count - number of total rentals

Import libraries

- Pandas is a high-level data manipulation tool
- NumPy is used for working with multidimensional arrays

In [2]:
```
1 import pandas as pd
2 import numpy as np
```

In [3]:
```
1 print(pd.show_versions())
```

```
C:\Users\ANANT KALEKAR\anaconda3\lib\site-packages\_distutils_hack\__init__.py:33: UserWarning: S
etuptools is replacing distutils.
  warnings.warn("Setuptools is replacing distutils.")


INSTALLED VERSIONS
------------------
commit           : ca60aab7340d9989d9428e11a51467658190bb6b
python           : 3.9.13.final.0
python-bits      : 64
OS               : Windows
OS-release       : 10
Version          : 10.0.22000
machine          : AMD64
processor        : Intel64 Family 6 Model 69 Stepping 1, GenuineIntel
byteorder        : little
LC_ALL           : None
LANG             : None
LOCALE           : English_India.1252
```

Load the first dataset

In [4]:
```
1 dataset_1 = pd.read_csv('dataset_1.csv')
```

Observations:

- We have to upload the dataset in the file explorer on the left panel of your lab
- We are reading the file through the dataset_1 variable
- The file is in CSV format
- We use the **`pd.read_csv()`** function to read a CSV file
- We provide the exact path of the file within the round bracket **`()`**

In [5]:
```
1 dataset_1
```

Out[5]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp
0	1	01-01-2011	1	0	1	0	False	6	1	0.24
1	2	01-01-2011	1	0	1	1	False	6	1	0.22
2	3	01-01-2011	1	0	1	2	False	6	1	0.22
3	4	01-01-2011	1	0	1	3	False	6	1	0.24
4	5	01-01-2011	1	0	1	4	False	6	1	0.24
...	...	...	...	...	...	...	...	...	...	...
605	606	28-01-2011	1	0	1	11	False	5	3	0.18
606	607	28-01-2011	1	0	1	12	False	5	3	0.18
607	608	28-01-2011	1	0	1	13	False	5	3	0.18
608	609	28-01-2011	1	0	1	14	False	5	3	0.22
609	610	28-01-2011	1	0	1	15	False	5	2	0.20

Check the type of dataset

- Execute the below command to understand type of data we are having

In [6]:
```
1 type(dataset_1)
```

Out[6]:
```
pandas.core.frame.DataFrame
```

Observations:

- The result shows that the dataset is DataFrame
- DataFrame is a tabular structure consisting of rows and columns

Shape of the dataset

In [7]:
```
1 dataset_1.shape
```

Out[7]:
```
(610, 10)
```

Observation:

- The `dataset_1` has 610 rows and 10 columns.

Print first 5 rows of the dataset

In [7]:
```
1 dataset_1.head()
```

Out[7]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp
0	1	01-01-2011	1	0	1	0	False	6	1	0.24
1	2	01-01-2011	1	0	1	1	False	6	1	0.22
2	3	01-01-2011	1	0	1	2	False	6	1	0.22
3	4	01-01-2011	1	0	1	3	False	6	1	0.24
4	5	01-01-2011	1	0	1	4	False	6	1	0.24

Observation:

- The **'dataset_1.head()'** function displays only the initial five rows of the dataset.

Load the second dataset

- Use the function carefully since it is an excel file

In [8]:
```
1 dataset_2 = pd.read_excel('dataset_2.xlsx')
```

Shape of the dataset

In [9]:
```
1 dataset_2.shape
```

Out[9]:
```
(610, 8)
```

Observation:

- The result shows that `dataset_2` has 610 rows and 8 columns.

Print first 5 rows of the dataset

In [10]:
```
1 dataset_2.head()
```

Out[10]:

	Unnamed: 0	instant	atemp	hum	windspeed	casual	registered	cnt
0	0	1	0.2879	0.81	0.0	3	13	16
1	1	2	0.2727	0.80	0.0	8	32	40
2	2	3	0.2727	0.80	0.0	5	27	32
3	3	4	0.2879	0.75	0.0	3	10	13
4	4	5	0.2879	0.75	0.0	0	1	1

Observation:

- We can see a column named `unnamed:0` , which is not in the data dictionary. Let's remove it.

Drop the column

In [9]:
```
1 dataset_2 = dataset_2.drop(['Unnamed: 0'], axis=1)
```

Lets check the shape of the dataset again after the drop

In [12]:
```
1 dataset_2.shape
```

Out[12]:
```
(610, 7)
```

Observation:

- We had 8 columns before the drop.
- When we check the shape of the file after the drop, we see that the column `Unnamed: 0` has been dropped

Top 5 rows of the dataset

- Let's check the dataset_2 again

In [13]:
```
1 dataset_2.head()
```

Out[13]:

	instant	atemp	hum	windspeed	casual	registered	cnt
0	1	0.2879	0.81	0.0	3	13	16
1	2	0.2727	0.80	0.0	8	32	40
2	3	0.2727	0.80	0.0	5	27	32
3	4	0.2879	0.75	0.0	3	10	13
4	5	0.2879	0.75	0.0	0	1	1

Observation:

- dataset_2 does not have `Unnamed: 0` column

Merge the datasets

- We have two datasets. They are dataset_1 and dataset_2
- As both datasets have one common column 'instant', let's merge the datasets on that column
- We are going to save the resultant data inside the **combined_data** as shown below

In [10]:
```
combined_data = pd.merge(dataset_1, dataset_2, on='instant')
```

Check the shape of combined dataset

In [15]:
```
combined_data.shape
```

Out[15]:
```
(610, 16)
```

Observation:

- The shape of the combined_data has 610 rows and 16 columns

Top 5 rows of the combined dataset

In [16]:
```
combined_data.head()
```

Out[16]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp	atemp	hum	windspeed	casual	registered
0	1	01-01-2011	1	0	1	0	False	6	1	0.24	0.2879	0.81	0.0	3	13
1	2	01-01-2011	1	0	1	1	False	6	1	0.22	0.2727	0.80	0.0	8	32
2	3	01-01-2011	1	0	1	2	False	6	1	0.22	0.2727	0.80	0.0	5	27
3	4	01-01-2011	1	0	1	3	False	6	1	0.24	0.2879	0.75	0.0	3	10
4	5	01-01-2011	1	0	1	4	False	6	1	0.24	0.2879	0.75	0.0	0	1

Now, load the 3rd dataset

- The dataset is saved in s3 bucket, we are going to download the dataset_3

In [11]:
```
import pip
!pip install wget
```

```
Requirement already satisfied: wget in c:\users\anant kalekar\anaconda3\lib\site-packages (3.2)
```

In [12]:
```
import wget

url = 'https://datasciencetrack.s3.us-east-2.amazonaws.com/dataset_3.csv'
wget.download(url)
```

```
100% [......................................................................] 25499 / 25499
```

Out[12]:
```
'dataset_3 (1).csv'
```

Observation:

- It will download the file directly from the main server after importing the wget module
- As shown above, we need to specify URL of the file only
- You will see the downloaded **`dataset_3.csv`** on your lab's file explorer

Import the dataset

In [13]:
```
dataset_3 = pd.read_csv('dataset_3.csv')
```

Check the shape of the dataset

In [20]:
```
1 dataset_3.shape
```

Out[20]: (390, 16)

Top 5 rows of the dataset

In [21]:
```
1 dataset_3.head()
```

Out[21]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp	atemp	hum	windspeed	casual	registered
0	620	29-01-2011	1	0	1	1	False	6	1	0.22	0.2273	0.64	0.1940	0	20
1	621	29-01-2011	1	0	1	2	False	6	1	0.22	0.2273	0.64	0.1642	0	15
2	622	29-01-2011	1	0	1	3	False	6	1	0.20	0.2121	0.64	0.1343	3	5
3	623	29-01-2011	1	0	1	4	False	6	1	0.16	0.1818	0.69	0.1045	1	2
4	624	29-01-2011	1	0	1	6	False	6	1	0.16	0.1818	0.64	0.1343	0	2

Bottom 15 rows of the dataset

- Just like the `head` function, the `tail` function is used to see the bottom rows of the dataset
- If you want to see the specific number of rows, then specify the number inside the **`bracket ()`** as shown below

In [22]:
```
1 dataset_3.tail(15)
```

Out[22]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp	atemp	hum	windspeed	casual	registe
375	995	14-02-2011	1	0	2	2	False	1	1	0.36	0.3333	0.40	0.2985	0	
376	996	14-02-2011	1	0	2	3	False	1	1	0.34	0.3182	0.46	0.2239	1	
377	997	14-02-2011	1	0	2	4	False	1	1	0.32	0.3030	0.53	0.2836	0	
378	998	14-02-2011	1	0	2	5	False	1	1	0.32	0.3030	0.53	0.2836	0	
379	999	14-02-2011	1	0	2	6	False	1	1	0.34	0.3030	0.46	0.2985	1	
380	1000	14-02-2011	1	0	2	7	False	1	1	0.34	0.3030	0.46	0.2985	2	
381	611	28-01-2011	1	0	1	16	False	5	1	0.22	0.2727	0.80	0.0000	10	

Observation:

- The bottom 15 rows of the dataset_3 is shown above, as we mention 15 inside the bracket ()
- Here, we can see that the rows are not sorted well according to the `instant` number. Let's resolve it.

Sort values of a column

- To sort the values per our will, we use the **`sort_values`** function and in the square brackets, we specify the name of the column by which we want to sort, as shown below

In [14]:
```
dataset_3 = dataset_3.sort_values(by=['instant'])
```

- Let's check head and tail to verify the sort operation

In [24]:
```
dataset_3.head()
```

Out[24]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp	atemp	hum	windspeed	casual	registered
381	611	28-01-2011	1	0	1	16	False	5	1	0.22	0.2727	0.80	0.0000	10	70
382	612	28-01-2011	1	0	1	17	False	5	1	0.24	0.2424	0.75	0.1343	2	147
383	613	28-01-2011	1	0	1	18	False	5	1	0.24	0.2273	0.75	0.1940	2	107
384	614	28-01-2011	1	0	1	19	False	5	2	0.24	0.2424	0.75	0.1343	5	84
385	615	28-01-2011	1	0	1	20	False	5	2	0.24	0.2273	0.70	0.1940	1	61

In [25]:
```
dataset_3.tail()
```

Out[25]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp	atemp	hum	windspeed	casual	registered
376	996	14-02-2011	1	0	2	3	False	1	1	0.34	0.3182	0.46	0.2239	1	1
377	997	14-02-2011	1	0	2	4	False	1	1	0.32	0.3030	0.53	0.2836	0	2
378	998	14-02-2011	1	0	2	5	False	1	1	0.32	0.3030	0.53	0.2836	0	3
379	999	14-02-2011	1	0	2	6	False	1	1	0.34	0.3030	0.46	0.2985	1	25
380	1000	14-02-2011	1	0	2	7	False	1	1	0.34	0.3030	0.46	0.2985	2	96

Concatenate the `combine_data` with `dataset_3`

- Let's concatenate both DataFrame combined_data and dataset_3 into a single DataFrame using the **`concat`** function, as shown below
- Store the final DataFrame inside the `final_data` variable

In [15]:
```
final_data = pd.concat([combined_data, dataset_3])
```

Check the shape of the new dataset

In [27]:
```
final_data.shape
```

Out[27]:
```
(1000, 16)
```

Observation:

- Now, the `final_data` has 1000 rows and 16 columns

In [28]:
```
1 final_data.head()
```

Out[28]:

	instant	dteday	season	yr	mnth	hr	holiday	weekday	weathersit	temp	atemp	hum	windspeed	casual	registered
0	1	01-01-2011	1	0	1	0	False	6	1	0.24	0.2879	0.81	0.0	3	13
1	2	01-01-2011	1	0	1	1	False	6	1	0.22	0.2727	0.80	0.0	8	32
2	3	01-01-2011	1	0	1	2	False	6	1	0.22	0.2727	0.80	0.0	5	27
3	4	01-01-2011	1	0	1	3	False	6	1	0.24	0.2879	0.75	0.0	3	10
4	5	01-01-2011	1	0	1	4	False	6	1	0.24	0.2879	0.75	0.0	0	1

Let's diplay the columns of the `final_data` DataFrame

In [16]:
```
1 final_data = final_data.rename(columns={'dteday': 'date', 'yr': 'year', 'mnth':'month','hr':'hou
2                                         'weathersit':'weather', 'hum':'humidity', 'cnt':'count'}
```

In [17]:
```
1 final_data.head()
```

Out[17]:

	instant	date	season	year	month	hour	holiday	weekday	weather	temp	atemp	humidity	windspeed	casual	registe
0	1	01-01-2011	1	0	1	0	False	6	1	0.24	0.2879	0.81	0.0	3	
1	2	01-01-2011	1	0	1	1	False	6	1	0.22	0.2727	0.80	0.0	8	
2	3	01-01-2011	1	0	1	2	False	6	1	0.22	0.2727	0.80	0.0	5	
3	4	01-01-2011	1	0	1	3	False	6	1	0.24	0.2879	0.75	0.0	3	
4	5	01-01-2011	1	0	1	4	False	6	1	0.24	0.2879	0.75	0.0	0	

Data types of different column values

In [31]:
```
1 final_data.dtypes
```

Out[31]:
```
instant         int64
date           object
season          int64
year            int64
month           int64
hour            int64
holiday          bool
weekday         int64
weather         int64
temp          float64
atemp         float64
humidity      float64
windspeed     float64
casual          int64
registered      int64
count           int64
dtype: object
```

Observations:

- We can see that the majority of our data columns are of type int64. They are therefore 64-bit integers. Some of the columns are of the type float64, which implies that they have decimals in them. However, only the date column has an

object type, indicating that it contains strings.

Check for null values

- Execute the given command to check the unknown values in the DataFrame

In [32]:
```
1 final_data.isna()
```

Out[32]:

	instant	date	season	year	month	hour	holiday	weekday	weather	temp	atemp	humidity	windspeed	casual	re
0	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
1	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
2	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
3	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
4	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
376	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
377	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
378	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
379	False	False	False	False	False	False	False	False	False	False	False	False	False	False	
380	False	False	False	False	False	False	False	False	False	False	False	False	False	False	

1000 rows × 16 columns

Observation:

- The **`isna()`** function returns DataFrame of Boolean values that are True for null values
- In a huge dataset, the code given above is not going to help
- We do not get enough idea of the null values by looking at the given tabular dataset
- The next line of code is more convenient in this case.

In [33]:
```
1 final_data.isna().sum(axis=0)
```

Out[33]:
```
instant        0
date           0
season         0
year           0
month          0
hour           0
holiday        0
weekday        0
weather        0
temp           0
atemp         11
humidity       0
windspeed      0
casual         0
registered     0
count          0
dtype: int64
```

Observations:

- The `isna().sum(axis=0)` function provides a clear picture of the number of null values in a DataFrame
- In the given result, we can see that the `atemp` column has 11 null values

Let's check the percentage of the rows with missing value

- We are performing this operation to determine whether the NA value rows can be dropped off or not so that we cannot deviate from our desired model

In [18]:
```
percentage_of_missing_values = (final_data['atemp'].isna().sum(axis=0)/final_data.shape[0])*100
percentage_of_missing_values
```

Out[18]:
```
1.0999999999999999
```

Observations:

- We divide the number of null values by the shape of the DataFrame to get the percentage of missing values.
- Since the percentage is 1, it is very less. Usually, the industry practice allows us to drop rows up to 30%. So, we can drop the rows with missing values.

Drop the rows with missing values

- We will use the `dropna` function to drop the null value rows

In [19]:
```
final_data = final_data.dropna(axis=0)
final_data.shape
```

Out[19]:
```
(989, 16)
```

Observations:

- We can see that the shape of the DataFrame reduced to 989 from 1000. It shows that the missing value rows have been wiped off.
- In further lessons of this course, we'll see different methods to treat missing values.

Now, let's again check the missing value count after the drop

In [20]:
```
final_data.isna().sum(axis=0)
```

Out[20]:
```
instant       0
date          0
season        0
year          0
month         0
hour          0
holiday       0
weekday       0
weather       0
temp          0
atemp         0
humidity      0
windspeed     0
casual        0
registered    0
count         0
dtype: int64
```

Perform sanity checks on the dataset

- It verifies the logical correctness of the data points

Check if `casual + registered` is always equal to `count`

In [21]:
```
1 np.sum(final_data['casual'] + final_data['registered'] - final_data['count'])
```

Out[21]:
```
0
```

Month values should be in the range of 1-12

- We will use the `unique()` function to find the elements of an array

In [22]:
```
1 np.unique(final_data.month)
```

Out[22]:
```
array([1, 2], dtype=int64)
```

Hour should be in the range of 1-24

In [23]:
```
1 np.unique(final_data.hour)
```

Out[23]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23], dtype=int64)
```

Print the statistical summary of the data

- We will use the `describe()` function to see the stastical summary of the dataset

In [24]:
```
1 print(final_data.describe())
```

```
           instant  season   year       month        hour     weekday  \
count   989.000000   989.0  989.0  989.000000  989.000000  989.000000
mean    505.622851     1.0    0.0    1.315470   11.753286    2.991911
std     286.274765     0.0    0.0    0.464938    6.891129    2.084727
min       1.000000     1.0    0.0    1.000000    0.000000    0.000000
25%     259.000000     1.0    0.0    1.000000    6.000000    1.000000
50%     506.000000     1.0    0.0    1.000000   12.000000    3.000000
75%     753.000000     1.0    0.0    2.000000   18.000000    5.000000
max    1000.000000     1.0    0.0    2.000000   23.000000    6.000000

          weather        temp       atemp    humidity   windspeed      casual  \
count  989.000000  989.000000  989.000000  989.000000  989.000000  989.000000
mean     1.479272    0.204712    0.211958    0.581769    0.194609    4.921132
std      0.651085    0.077789    0.076703    0.187706    0.129225    7.666231
min      1.000000    0.020000    0.000000    0.210000    0.000000    0.000000
25%      1.000000    0.160000    0.166700    0.440000    0.104500    0.000000
50%      1.000000    0.200000    0.212100    0.550000    0.164200    3.000000
75%      2.000000    0.240000    0.257600    0.700000    0.283600    6.000000
max      4.000000    0.460000    0.454500    1.000000    0.582100   62.000000

       registered       count
count  989.000000  989.000000
mean    53.689585   58.610718
std     48.019224   51.120572
min      0.000000    1.000000
25%     15.000000   16.000000
50%     46.000000   50.000000
75%     75.000000   84.000000
max    247.000000  249.000000
```

Note: We have seen almost all the methods of data wrangling, now let's see explicitly outlier detection and removing

Import the Libraries

- Apart from the Pandas and NumPy, this time we are also calling Scikit-learn
- Scikit-learn (sklearn) is an open-source module that has some inbuilt datasets, like boston and iris
- Each dataset has a corresponding function used to load the dataset

- These functions follow the same format: “load_DATASET()”, where DATASET refers to the name of the dataset
- We are importing two datasets from `sklearn.datasets` in the cell below

In [26]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import load_boston, load_iris
```

Load the Data

- Since these datasets are directly importing from Scikit Learn, the load functions (such as load_boston()) do not return data in the tabular format
- The data is stored in the form of keys (words) and values (definition) like the dictionary structure
- Let’s load the dataset and store it in a variable called boston
- Now, we are going to call the keys for boston dataset `print(boston.keys())`

In [27]:
```
boston = load_boston()

#Find the dic keys
print(boston.keys())
```

```
C:\Users\ANANT KALEKAR\anaconda3\lib\site-packages\sklearn\utils\deprecation.py:87: FutureWarning: F
unction load_boston is deprecated; `load_boston` is deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use of this
    dataset unless the purpose of the code is to study and educate about
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np


        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.

  warnings.warn(msg, category=FutureWarning)
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename', 'data_module'])
```

Observations:

- We get the keys such as data, target, feature_names, DESCR, and filename
- The first two keys 'data' and 'target' has the only actual data, rest serve a descriptive purpose
- `data` has all the input features of the dataset in a NumPy array and `target` has the output feature based on which we do the prediction. `target` is in the NumPy array
- `feature_names` has all the column names of the dataset in a NumPy array and `DESCR` is the description of the dataset `filename` that has the file path in CSV format

Find features name

- Let's see the columns in the dataset

In [28]:
```
#Find features and target
x = boston.data
y = boston.target

columns = boston.feature_names
columns
```

Out[28]:
```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [44]:
```
x
```

Out[44]:
```
array([[6.3200e-03, 1.8000e+01, 2.3100e+00, ..., 1.5300e+01, 3.9690e+02,
        4.9800e+00],
       [2.7310e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9690e+02,
        9.1400e+00],
       [2.7290e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9283e+02,
        4.0300e+00],
       ...,
       [6.0760e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
        5.6400e+00],
       [1.0959e-01, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9345e+02,
        6.4800e+00],
       [4.7410e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
        7.8800e+00]])
```

In [45]:
```
y
```

Out[45]:
```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15. ,
       18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
       15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
       13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
       21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
       35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
       19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
       20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
       23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
       33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
       21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
       20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
       23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
       15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
       17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
       25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
       23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
       32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
       34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
```

Description of dataset

In [ ]:
```
```

In [29]:
```
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually
the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of black people by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/ (https://archive.ics.uci.edu/ml/machine-learning-databases/housing/)


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Col
linearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Ten
th International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morg
an Kaufmann.
```

Let's convert the array to a DataFrame i.e into tabular structure

- Since both data and target are in a NumPy array, we need to convert it to a DataFrame

In [30]:
```
boston_df = pd.DataFrame(boston.data)
boston_df.columns = columns
boston_df["MEDV"] = y
boston_df_o = boston_df
print(boston_df.shape)
print(boston_df_o.shape)
```

```
(506, 14)
(506, 14)
```

In [48]:
```
1 boston_df
```

Out[48]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88	11.9

506 rows × 14 columns

Observation:

- boston_df has 506 rows and 14 columns

In [49]:
```
1 print(boston_df.describe())
```

```
              CRIM          ZN       INDUS        CHAS         NOX          RM  \
count   506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean      3.613524   11.363636   11.136779    0.069170    0.554695    6.284634
std       8.601545   23.322453    6.860353    0.253994    0.115878    0.702617
min       0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
25%       0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
50%       0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
75%       3.677083   12.500000   18.100000    0.000000    0.624000    6.623500
max      88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

              AGE         DIS         RAD         TAX     PTRATIO           B  \
count   506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean     68.574901    3.795043    9.549407  408.237154   18.455534  356.674032
std      28.148861    2.105710    8.707259  168.537116    2.164946   91.294864
min       2.900000    1.129600    1.000000  187.000000   12.600000    0.320000
25%      45.025000    2.100175    4.000000  279.000000   17.400000  375.377500
50%      77.500000    3.207450    5.000000  330.000000   19.050000  391.440000
75%      94.075000    5.188425   24.000000  666.000000   20.200000  396.225000
max     100.000000   12.126500   24.000000  711.000000   22.000000  396.900000

            LSTAT        MEDV
count  506.000000  506.000000
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

In [45]:
```
1 boston_df.head()
```

Out[45]:

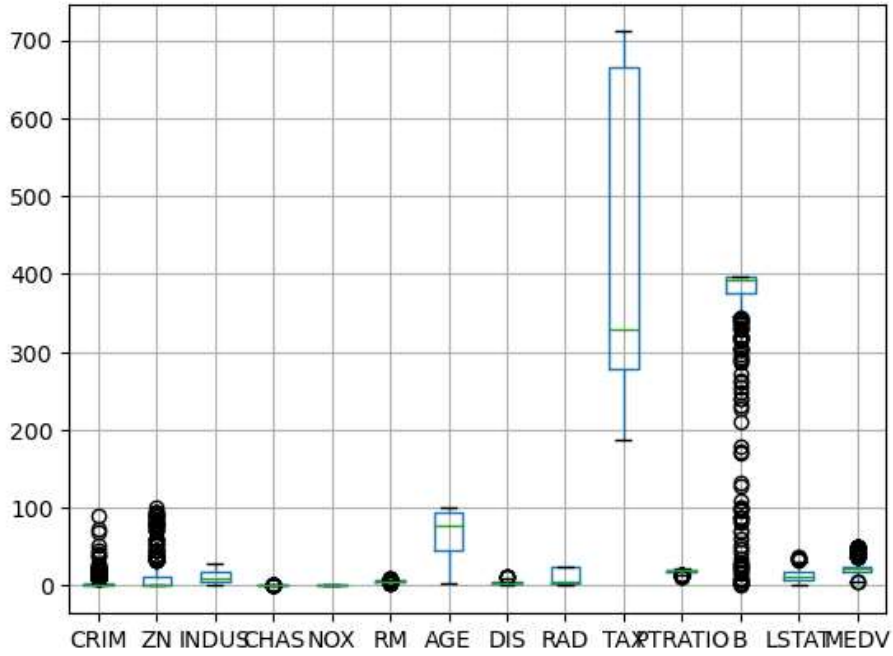
	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

Outlier Detection - Univarite - Boxplot

- Boxplot is a graphical method for representing the values in the data
- It shows how the values are spread and skewed

In [31]:
```
1 boston_df.boxplot()
```

Out[31]: <AxesSubplot:>



Observations:

- Since the values are different from each other, it is not right to put every column in the same scale of measurement

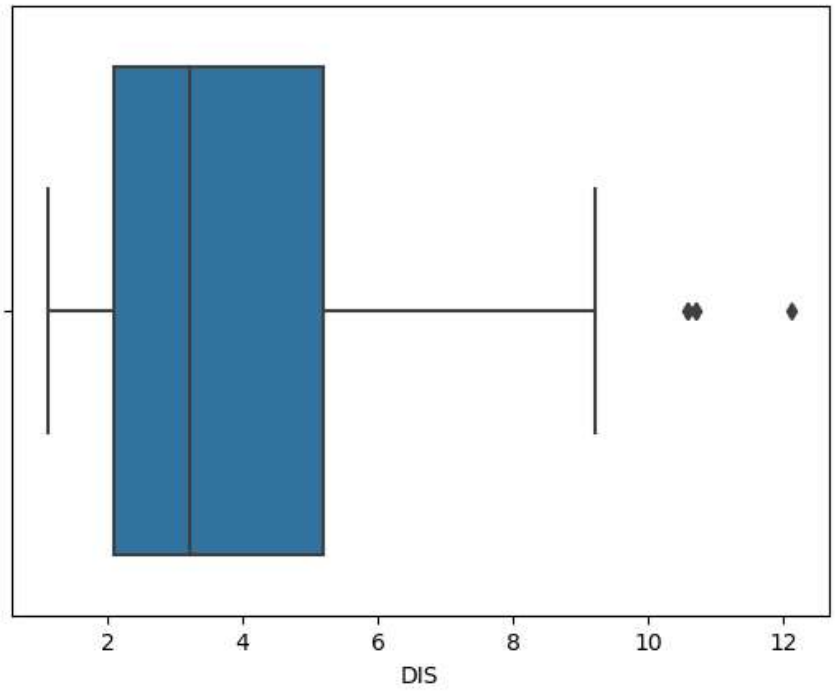
Let's import the libraries like matplotlib and seaborn

- Matplotlib is a data visualizing and graphical plotting library for Python
- Seaborn is also a data visualization library built on top of Matplotlib, which is used for making statistical graphics

In [32]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
sns.boxplot(x=boston_df['DIS'])
```

Out[32]: <AxesSubplot:xlabel='DIS'>



IQR (Interquartile range) technique for outlier treatment

In [33]:
```
def outlier_treatment(col):
    sorted(col)
    Q1,Q3 = np.percentile(col , [25,75])
    IQR = Q3 - Q1
    lower_range = Q1 - (1.5 * IQR)
    upper_range = Q3 + (1.5 * IQR)
    return lower_range,upper_range
```

Observations:

- In this technique, we divide our dataset under percentiles like 25th and 75th of a sample
- After that, we find the IQR between these two percentiles
- Now, to remove the outliers, we calculate the lower and upper range by using the given formula
- All the values which are beyond these ranges are considered outliers and must be removed

In [34]:
```
lower_range,upper_range = outlier_treatment(boston_df['DIS'])
print("Lower Range:",lower_range)
print("Upper Range:",upper_range)
```

```
Lower Range: -2.5322000000000005
Upper Range: 9.820800000000002
```

Observation:

- We have calculated the lower and upper ranges for `DIS` feature of our `boston_df`

In [35]:
```
lower_boston_df = boston_df[boston_df["DIS"].values < lower_range]
lower_boston_df
```

Out[35]:

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV

Let us show the values which are beyond upper and lower range in our dataset

In [36]:
```
upper_boston_df = boston_df[boston_df["DIS"].values > upper_range]
upper_boston_df

```

Out[36]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
351	0.07950	60.0	1.69	0.0	0.411	6.579	35.9	10.7103	4.0	411.0	18.3	370.78	5.49	24.1
352	0.07244	60.0	1.69	0.0	0.411	5.884	18.5	10.7103	4.0	411.0	18.3	392.33	7.79	18.6
353	0.01709	90.0	2.02	0.0	0.410	6.728	36.1	12.1265	5.0	187.0	17.0	384.46	4.50	30.1
354	0.04301	80.0	1.91	0.0	0.413	5.663	21.9	10.5857	4.0	334.0	22.0	382.80	8.05	18.2
355	0.10659	80.0	1.91	0.0	0.413	5.936	19.5	10.5857	4.0	334.0	22.0	376.04	5.57	20.6

Observations:

- There are no rows that are lesser than the lower range, but there are five rows that are greater than our upper range

In [37]:
```

lower_outliers = lower_boston_df.value_counts().sum(axis=0)
upper_outliers = upper_boston_df.value_counts().sum(axis=0)
total_outliers = lower_outliers + upper_outliers

print("Total Number of Outliers:",total_outliers)
```

```
Total Number of Outliers: 5
```

Observation:

- With the given code, we are summing up the total number of outlier rows

Let us list down the row numbers that contain outliers:

In [38]:
```

lower_index = list(boston_df[ boston_df['DIS'] < lower_range ].index)

upper_index = list(boston_df[ boston_df['DIS'] > upper_range ].index)

total_index = list(lower_index + upper_index)

print(total_index)

```

```
[351, 352, 353, 354, 355]
```

Drop the outlier rows

In [39]:
```
print("Shape Before Dropping Outlier Rows:", boston_df.shape)

boston_df.drop(total_index, inplace = True)

print("Shape After Dropping Outlier Rows:", boston_df.shape)
```

```
Shape Before Dropping Outlier Rows: (506, 14)
Shape After Dropping Outlier Rows: (501, 14)
```

Observation:

- In the given code, we checked the shape of the dataset before and after dropping outliers rows
- You can see that the rows before dropping the outliers were 506 and after dropping it became 501. Thus, we have successfully dropped the unwanted rows

In [63]:
```
1 print(boston_df.mean())
```

```
CRIM         3.648951
ZN          10.738523
INDUS       11.229521
CHAS         0.069860
NOX          0.556123
RM           6.285898
AGE         68.996008
DIS          3.723699
RAD          9.602794
TAX        408.964072
PTRATIO     18.444910
B          356.428443
LSTAT       12.716667
MEDV        22.534930
dtype: float64
```

Let's check the skewness of the data

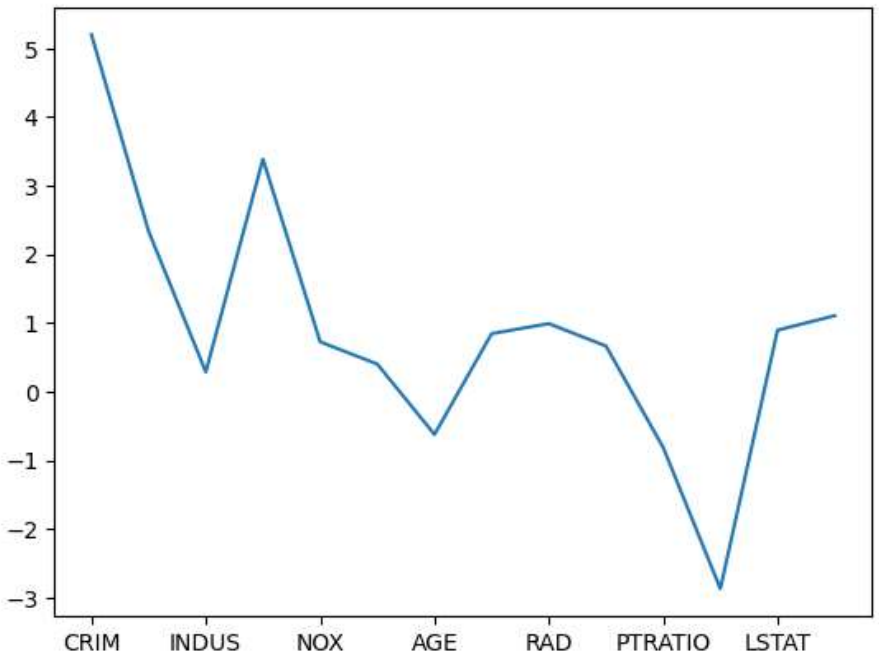
- Skewness is a way of checking the outliers' position in the dataset. If the data is right-skewed, it means that most of the outliers are at the right side of the distribution and vice-versa.

In [64]:
```
1 boston_df.skew(axis = 0, skipna = True)
```

Out[64]:
```
CRIM        5.199083
ZN          2.333143
INDUS       0.286236
CHAS        3.384961
NOX         0.724482
RM          0.399880
AGE        -0.624705
DIS         0.843752
RAD         0.989519
TAX         0.662769
PTRATIO    -0.814678
B          -2.871741
LSTAT       0.894723
MEDV        1.105009
dtype: float64
```

In [65]:
```
1 boston_df.skew(axis = 0, skipna = True).plot()
```

Out[65]:
```
<AxesSubplot:>
```



Correlation

- Correlation is a statistical term that refers to how close two variables are to form a linear relationship with one another.
- High correlation features are more linearly dependent and therefore have almost the same effect on the dependent variable. When two features have a high correlation, we can drop one of them.

In [66]:
```
pearsoncorr = boston_df.corr(method='pearson')
pearsoncorr
```

Out[66]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATI
CRIM	1.000000	-0.197045	0.405070	-0.057085	0.419441	-0.220593	0.350953	-0.389244	0.624689	0.582689	0.29391
ZN	-0.197045	1.000000	-0.522068	-0.036798	-0.506471	0.329664	-0.556961	0.632905	-0.307898	-0.313254	-0.42434
INDUS	0.405070	-0.522068	1.000000	0.059823	0.759689	-0.398559	0.637736	-0.710742	0.593388	0.723164	0.39579
CHAS	-0.057085	-0.036798	0.059823	1.000000	0.088550	0.090968	0.083442	-0.095687	-0.009062	-0.036855	-0.12092
NOX	0.419441	-0.506471	0.759689	0.088550	1.000000	-0.307300	0.726982	-0.779518	0.609673	0.669190	0.19755
RM	-0.220593	0.329664	-0.398559	0.090968	-0.307300	1.000000	-0.247881	0.223962	-0.211945	-0.292789	-0.35291
AGE	0.350953	-0.556961	0.637736	0.083442	0.726982	-0.247881	1.000000	-0.750649	0.452882	0.507512	0.27546
DIS	-0.389244	0.632905	-0.710742	-0.095687	-0.779518	0.223962	-0.750649	1.000000	-0.504962	-0.552740	-0.26461
RAD	0.624689	-0.307898	0.593388	-0.009062	0.609673	-0.211945	0.452882	-0.504962	1.000000	0.911417	0.47165
TAX	0.582689	-0.313254	0.723164	-0.036855	0.669190	-0.292789	0.507512	-0.552740	0.911417	1.000000	0.46565
PTRATIO	0.293911	-0.424341	0.395794	-0.120921	0.197554	-0.352913	0.275461	-0.264611	0.471650	0.465659	1.00000
B	-0.384430	0.174996	-0.356755	0.049561	-0.379776	0.129013	-0.272692	0.300387	-0.443783	-0.441691	-0.17964
LSTAT	0.454222	-0.405733	0.599738	-0.056617	0.586774	-0.618199	0.598830	-0.498097	0.486202	0.543021	0.38142
MEDV	-0.389168	0.374678	-0.489142	0.175463	-0.431362	0.694797	-0.383383	0.265689	-0.383126	-0.468504	-0.50804

Observations:

- The correlation coefficient value lies between -1 to +1.
- The attributes closer to 1 are more positively correlated.
- Values close to -1 are more negatively correlated.
- We can see more detail about correlation in Heatmap with a better visualization.

Pearson Coefficient

- The Pearson coefficient is a measure of the intensity and direction of a linear relationship between two variables that allows no assumptions about causality. It represents correlation rather than causation and has a scale of +1 to -1, with +1 representing a positive correlation, -1 representing a negative correlation, and 0 representing no relationship.

Pearson Correlation Calculation

We use **method = pearson** for the **boston_df.corr** for calculating the pearson coefficient of correlation.

In [1]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

plt.figure(figsize= (10,10), dpi=100)
#sns.heatmap(boston_df.corr(), annot=True)
sns.heatmap(pearsoncorr,
xticklabels=pearsoncorr.columns,
yticklabels=pearsoncorr.columns,
annot=True,
linewidth=0.5)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_4688\1842402310.py in <module>
      5 plt.figure(figsize= (10,10), dpi=100)
      6 #sns.heatmap(boston_df.corr(), annot=True)
----> 7 sns.heatmap(pearsoncorr,
      8 xticklabels=pearsoncorr.columns,
      9 yticklabels=pearsoncorr.columns,

NameError: name 'pearsoncorr' is not defined


<Figure size 1000x1000 with 0 Axes>
```

Heatmap Observations:

- Each square represents the correlation between the variables on each axis, ranging from -1 to +1.
- There is no linear relationship between the two variables when the values are close to zero.
- The attributes closer to 1 are more positively correlated. If one increases, so does the other, and the closer they are to 1, the stronger their relationship.
- Values close to -1 are more negatively correlated. When one increases, the other decreases, and the closer they are to -1, the weaker their relationship.
- The diagonals are all 1 (light peach in color), as the squares are correlating each variable to itself (so it's a perfect correlation).
- For the others, the larger the number and brighter the color, the stronger is the correlation between the two variables.
- Since the same two variables are paired together in squares of the diagonal, the plot is therefore symmetrical around it.

In [68]:
```
#Correlation with output variable
cor_target = abs(pearsoncorr["MEDV"]) #Selecting highly correlated features
relevant_features = cor_target[cor_target>0.5]
relevant_features
```

Out[68]:
```
RM         0.694797
PTRATIO    0.508040
LSTAT      0.740998
MEDV       1.000000
Name: MEDV, dtype: float64
```

Relative feature observations

- The features RM, PTRATIO, and LSTAT are strongly correlated with the output variable MEDV (target variable) per above analysis.
- As a result, we will remove all other features except these.
- However, one of the assumptions of linear regression is that the independent variables are not correlated to one another.
- If these variables are correlated with one another, we can hold just one and discard the others.

In [40]:
```
print(boston_df[["LSTAT","PTRATIO"]].corr())
print()
print(boston_df[["RM","PTRATIO"]].corr())
print()
print(boston_df[["RM","LSTAT"]].corr())
```

```
             LSTAT   PTRATIO
LSTAT    1.000000  0.381429
PTRATIO  0.381429  1.000000

                RM   PTRATIO
RM       1.000000 -0.352913
PTRATIO -0.352913  1.000000

               RM     LSTAT
RM       1.000000 -0.618199
LSTAT   -0.618199  1.000000
```

Correlation observations:

- From the above analysis, it is visible that the variables LSTAT and RM are highly correlated with each other (-0.618199). Hence, we will keep only one variable and drop the other. We will keep LSTAT since its correlation with MEDV is higher than that of RM.
- After **dropping RM**, we are left with two features, **LSTAT** and **PTRATIO.** These are the final features given by Pearson correlation.

In [41]:
```
boston_df.drop(columns = ["RM"], inplace = True)
```

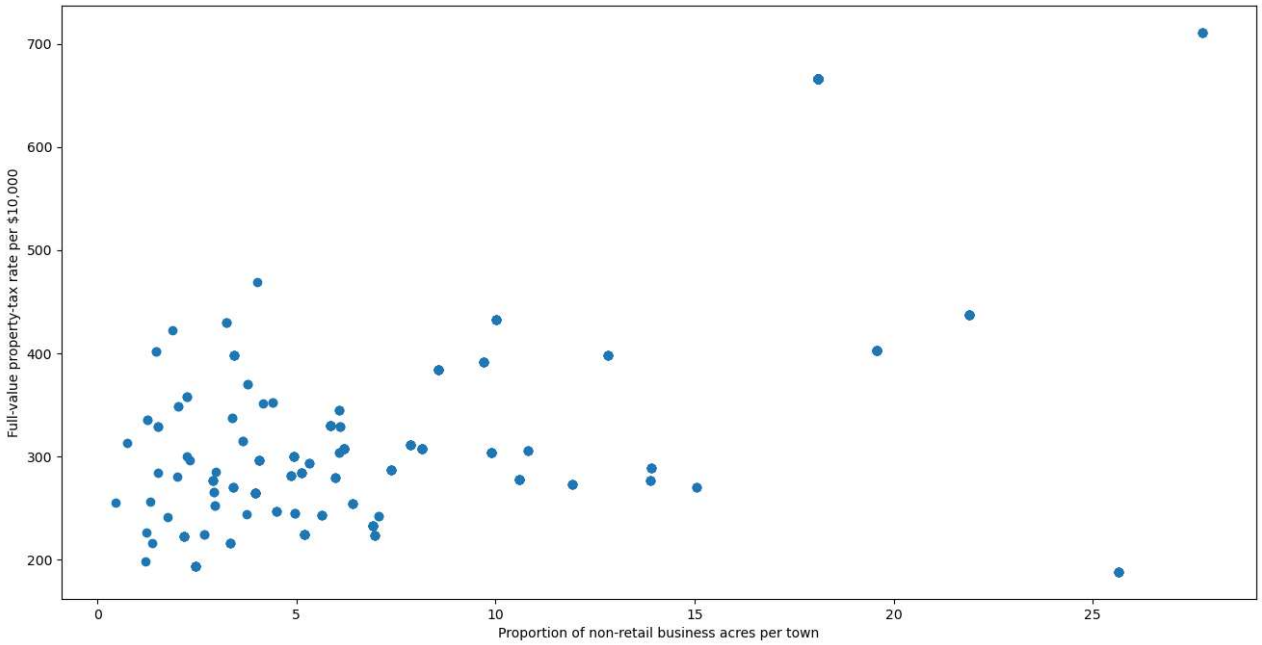
In [76]:
```
boston_df.columns
```

Out[76]:
```
Index(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'AGE', 'DIS', 'RAD', 'TAX',
       'PTRATIO', 'B', 'LSTAT', 'MEDV'],
      dtype='object')
```

Multivariate Outlier Analysis

- We will draw a scatter plot between `INDUS` and `TAX` columns to check how the values are placed
- We will see the dependency of one cloumn over other
- We will also see that some of the results do not match the patterns that others do, which we refer to as outliers

In [77]:
```
fig, ax = plt.subplots(figsize=(16,8))
ax.scatter(boston_df['INDUS'], boston_df['TAX'])
ax.set_xlabel('Proportion of non-retail business acres per town')
ax.set_ylabel('Full-value property-tax rate per $10,000')
plt.show()
```



Outliers detection using Z-Score

- Z-score indicates how many standard deviations away a data point is
- Standard deviation measures the amount of variation of a set of values from its mean
- We can drop the rows which lie beyond our threshold, i.e., z=3

In [79]:
```
from scipy import stats
import numpy as np
z = np.abs(stats.zscore(boston_df))
print(z)
```

```
        CRIM        ZN     INDUS      CHAS       NOX       AGE       DIS  \
0   0.422165  0.322399  1.307044  0.274057  0.156980  0.135909  0.184288
1   0.419732  0.476775  0.609526  0.274057  0.754647  0.354595  0.625560
2   0.419734  0.476775  0.609526  0.274057  0.754647  0.282702  0.625560
3   0.419146  0.476775  1.326094  0.274057  0.849928  0.830491  1.176510
4   0.414895  0.476775  1.326094  0.274057  0.849928  0.529744  1.176510
..       ...       ...       ...       ...       ...       ...       ...
501 0.415639  0.476775  0.102646  0.274057  0.146185  0.003723  0.626414
502 0.417651  0.476775  0.102646  0.274057  0.146185  0.275828  0.722558
503 0.415855  0.476775  0.102646  0.274057  0.146185  0.787813  0.782930
504 0.410196  0.476775  0.102646  0.274057  0.146185  0.726948  0.671543
505 0.417403  0.476775  0.102646  0.274057  0.146185  0.422621  0.613132

          RAD       TAX   PTRATIO         B     LSTAT      MEDV
0    0.985955  0.669009  1.455319  0.441722  1.083636  0.158846
1    0.871346  0.988814  0.298435  0.441722  0.500966  0.101367
2    0.871346  0.988814  0.298435  0.397300  1.216697  1.318967
3    0.756737  1.107260  0.118044  0.416946  1.369368  1.178018
4    0.756737  1.107260  0.118044  0.441722  1.034613  1.481601
..       ...       ...       ...       ...       ...       ...
501  0.985955  0.805222  1.182377  0.388132  0.426731  0.014629
502  0.985955  0.805222  1.182377  0.441722  0.509369  0.209790
503  0.985955  0.805222  1.182377  0.441722  0.991193  0.148004
504  0.985955  0.805222  1.182377  0.404067  0.873538  0.057998
505  0.985955  0.805222  1.182377  0.441722  0.677447  1.153065

[501 rows x 13 columns]
```

In [66]:
```
1 z.shape
```

Out[66]:
```
(501, 13)
```

We will now print the rows whose z value is greater than the threshold

In [80]:
```
1 threshold = 3
2 print(np.where(z > 3))
```

```
(array([ 55,  56,  57,  65,  66, 102, 141, 142, 152, 154, 155, 160, 162,
       163, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 208, 209,
       210, 211, 212, 216, 218, 219, 220, 221, 222, 234, 236, 254, 255,
       256, 269, 273, 274, 276, 277, 282, 283, 283, 284, 286, 290, 291,
       292, 347, 348, 351, 352, 353, 358, 359, 364, 365, 367, 368, 369,
       375, 393, 399, 400, 405, 405, 406, 407, 407, 409, 409, 410, 411,
       413, 413, 414, 418, 419, 420, 421, 422, 422, 424, 426, 431, 432,
       433, 440, 445, 449, 450, 451, 452, 461], dtype=int64), array([ 1,  1,  1,  1,  1, 10, 11,  3,
3,  3,  3,  3,  3,  3,  1,  1,  1,
        1,  1,  1,  1,  1,  1,  1,  3,  3,  3,  3,  3,  3,  3,  3,  3,  3,
        3,  3,  3,  1,  1,  1,  3,  3,  3,  3,  3,  3,  1,  3,  1,  1,  1,
        1,  1,  1,  1,  3,  3,  3,  3,  3,  3,  3,  3, 11, 11,  0,  0,  0,
        0,  0, 10, 10, 10, 11,  0, 11, 10, 10,  0, 10, 10, 10, 10, 10, 10,
        0, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
      dtype=int64))
```

Removing Outliers

- Taking only those rows which lies under our threshold

In [71]:
```
1 boston_df_o = boston_df_o[(z < 3).all(axis=1)]
```

```
C:\Users\TEMP\AppData\Local\Temp\ipykernel_17052\3283073001.py:1: UserWarning: Boolean Series key wi
ll be reindexed to match DataFrame index.
  boston_df_o = boston_df_o[(z < 3).all(axis=1)]
```

In [72]:
```
1 boston_df.shape
```

Out[72]:
```
(501, 13)
```

In [73]:
```
1 boston_df_o.shape
```

Out[73]:
```
(408, 13)
```

In [74]:
```
1 boston_df_o1 = boston_df
```

Now, let us use IQR (Interquartile range) technique once again to remove outliers

In [75]:
```
1 Q1 = boston_df_o1.quantile(0.25)
2 Q3 = boston_df_o1.quantile(0.75)
3 IQR = Q3 - Q1
4 print(IQR)
```

```
CRIM         3.61046
ZN          12.50000
INDUS       12.91000
CHAS         0.00000
NOX          0.17100
AGE         48.40000
DIS          3.02850
RAD         20.00000
TAX        387.00000
PTRATIO      2.80000
B           20.91000
LSTAT        9.97000
MEDV         8.20000
dtype: float64
```

In [76]:
```
IQR["CRIM"]
```

Out[76]: 3.61046

In [77]:
```
boston_df_o.shape
```

Out[77]: (408, 13)

In [78]:
```
# Custom function to find and remove outliers based on each column
def multi_outlier_treatment(data):
    columns = data.drop(columns=["MEDV"]).columns  # Getting the names of each row except the ta
    Q1 = boston_df_o1.quantile(0.25)
    Q3 = boston_df_o1.quantile(0.75)
    IQR = Q3 - Q1
    row_index_list = list()
    print()

    for col in columns:
      lower_range = Q1[col] - (1.5 * IQR[col])
      upper_range = Q3[col] + (1.5 * IQR[col])

      lower_data_df = data[data[col].values < lower_range]
      upper_data_df = data[data[col].values > upper_range]

      lower_outliers = lower_data_df.value_counts().sum(axis=0)
      upper_outliers = upper_data_df.value_counts().sum(axis=0)
      total_outliers = lower_outliers + upper_outliers
      print("Total Number of Outliers in "+str(col)+" column:",total_outliers)

      lower_index = list(data[ data[col] < lower_range ].index)
      upper_index = list(data[ data[col] > upper_range ].index)
      total_index = list(lower_index + upper_index)

      row_index_list = list(row_index_list + total_index)

    row_index_list = list(set(row_index_list))
    row_index_list.sort()
    total_rows_count = len(row_index_list)
    print()
    print("Total number of unique rows to delete:",total_rows_count)
    print()
    print("Unique rows to delete:\n", row_index_list)
    print()
    return data.drop(row_index_list)

```

Let's drop all the rows which do not lie between the upper and lower range

- We are going to drop 120 rows from our dataset that contains outliers

In [79]:
```
print("Shape before dropping outlier rows:", boston_df_o.shape)
boston_df_o = multi_outlier_treatment(boston_df_o)
print("Shape after dropping outlier rows:", boston_df_o.shape)
```

```
Shape before dropping outlier rows: (408, 13)

Total Number of Outliers in CRIM column: 42
Total Number of Outliers in ZN column: 34
Total Number of Outliers in INDUS column: 0
Total Number of Outliers in CHAS column: 0
Total Number of Outliers in NOX column: 0
Total Number of Outliers in AGE column: 0
Total Number of Outliers in DIS column: 0
Total Number of Outliers in RAD column: 0
Total Number of Outliers in TAX column: 0
Total Number of Outliers in PTRATIO column: 12
Total Number of Outliers in B column: 44
Total Number of Outliers in LSTAT column: 0

Total number of unique rows to delete: 120

Unique rows to delete:
 [18, 25, 27, 32, 34, 39, 40, 54, 118, 134, 145, 146, 151, 153, 156, 165, 167, 168, 169, 170, 187, 1
88, 189, 190, 191, 192, 193, 194, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 275, 2
78, 285, 287, 288, 289, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 331, 332, 341, 343, 344, 3
49, 350, 366, 367, 371, 375, 376, 377, 378, 379, 381, 382, 384, 385, 386, 387, 388, 392, 394, 399, 4
00, 401, 402, 403, 406, 407, 408, 409, 413, 417, 420, 421, 422, 428, 430, 432, 433, 434, 435, 439, 4
40, 441, 443, 444, 446, 447, 448, 449, 458, 460, 465, 467, 468, 469, 475, 477, 478, 479, 490]

Shape after dropping outlier rows: (288, 13)
```

Conclusion

- The completed dataset, `boston_df_o` , can now be utilised to construct a model. As the dataset is so well-defined, the model will be more accurate.

In [80]:
```
print(boston_df_o.isna().sum(axis=0))
```

```
CRIM       0
ZN         0
INDUS      0
CHAS       0
NOX        0
AGE        0
DIS        0
RAD        0
TAX        0
PTRATIO    0
B          0
LSTAT      0
MEDV       0
dtype: int64
```

Note: In this notebook, we saw the use of the data wrangling methods, but in the second notbook of this lesson we are going to use one of these methods as a component for analyzing the data acquired with "Web Scraping".