# The pandas Library

## Exercise 1: Using DataFrames to Manipulate Stored Student test score Data

In this exercise, you will create a **dictionary**, which is one of many ways to create a **`pandas`** DataFrame. You will then manipulate this data as required. In order to use pandas, you must import **`pandas`**, which is universally imported as **`pd`**. Pandas and NumPy are so omnipresent it's a good idea to import them first before performing any kind of data analysis. The following steps will enable you to complete the exercise:

1. You'll begin by importing **`pandas`** as **`pd`**:
   import pandas as pd
   Now that you have imported **`pandas`**, you will create a **`DataFrame`**.
2. First, you will create a dictionary of test scores as **`test_dict`**:
   # Create dictionary of test scores
   test_dict = {'Corey':[63,75,88], 'Kevin':[48,98,92], 'Akshay': [87, 86, 85]}
3. Next, you place the **`text_dict`** into the DataFrame using the **`DataFrame`** method:
   # Create DataFrame
   df = pd.DataFrame(test_dict)
4. Now, you can display the Dataframe:
   # Display DataFrame
   df
   You should get the following output:

Out[4]:

| | Corey | Kevin | Akshay |
|---|---|---|---|
| 0 | 63 | 48 | 87 |
| 1 | 75 | 98 | 86 |
| 2 | 88 | 92 | 85 |

Figure 1: Output with the values added in the DataFrame

You can inspect the **`DataFrame`**. First, each dictionary key is listed as a column. Second, the rows are labeled with indices starting with `0` by default. Third, the visual layout is clear and legible.

Each column and row of **`DataFrame`** is officially represented as a **Series**. A series is a one-dimensional **`ndarray`**.

Now, you will rotate the DataFrame, which is also known as a **transpose**, a standard **`pandas`** method. A transpose turns rows into columns and columns into rows.

5. Copy the code shown in the following code snippet to perform a transpose on the DataFrame:

```
# Transpose DataFrame
df = df.T
df
```

You should get the following output:

Out[5]:

| | 0 | 1 | 2 |
|---|---|---|---|
| Corey | 63 | 75 | 88 |
| Kevin | 48 | 98 | 92 |
| Akshay | 87 | 86 | 85 |

Figure 2: The output of the transpose on the DataFrame

In this exercise, you created a DataFrame that holds the values of **`testscores`**, and to finish, you transposed this DataFrame to get the output. In the next exercise, you will

look at renaming column names and selecting data from the DataFrame, which is an important part of working with pandas.

# Exercise 2: DataFrame Computations with the Student testscore Data

In this exercise, you will rename the columns of the DataFrame, and you will then select the data to display as output from the DataFrame:

1. Open a new Jupyter Notebook.
2. Import **`pandas`** as **`pd`** and enter the student value, as shown in *Exercise 1*, *Using DataFrames to Manipulate Stored Student testscore Data*. After this, convert it to a DataFrame and have it transposed:
   ```
   import pandas as pd
   # Create dictionary of test scores
   test_dict = {'Corey':[63,75,88], 'Kevin':[48,98,92],
   'Akshay': [87, 86, 85]}
   # Create DataFrame
   df = pd.DataFrame(test_dict)
   df = df.T
   ```
3. Now rename the columns to something more precise. You can use **`.columns`** on the DataFrame to rename the column names:
   ```
   # Rename Columns
   df.columns = ['Quiz_1', 'Quiz_2', 'Quiz_3']
   df
   ```
   You should get the following output:

| | Quiz_1 | Quiz_2 | Quiz_3 |
|---|---|---|---|
| Corey | 63 | 75 | 88 |
| Kevin | 48 | 98 | 92 |
| Akshay | 87 | 86 | 85 |

Figure 3: Output with changed column names

Now, select a range of values from specific rows and columns. You will be using **`.iloc`** with the index number, which is a function present in a pandas DataFrame for selection. This is shown in the following step:

4. Select a range of rows:
   # Access first row by index number
   df.**iloc**[0]
   You should get the following output:

```
Quiz_1    63
Quiz_2    75
Quiz_3    88
Name: Corey, dtype: int64
```

Figure 4: Output with the selected values

5. Now, select a column using its name, as shown in the following code snippet.
   You can access columns by putting the column name in quotes inside of brackets:
   # Access first column by name
   df['Quiz_1']
   You should get the following output:

```
Corey     63
Kevin     48
Akshay    87
Name: Quiz_1, dtype: int64
```

Figure 5: Output while accessing it with a change in the column name

6. Now, select a column using the dot (`.`) notation:
   # Access first column using dot notation
   df.Quiz_1

You should get the following output:

```
Corey     63
Kevin     48
Akshay    87
Name: Quiz_1, dtype: int64
```

Figure 5: Output of selecting a column using the dot notation

*Note*

*There are limitations to using dot notation, so bracket quotations are often preferable.*

In this exercise, you implemented and changed the column names of the DataFrame. Next, you used **`.iloc`** to select data as per our requirement from the DataFrame.

In the next exercise, you will implement different computations on the DataFrame.

# Exercise 3: Computing DataFrames within DataFrames

In this exercise, you will use the same **`testscore`** data and perform more computations on the DataFrame. The following steps will enable you to complete the exercise:

1. Open a new Jupyter Notebook.
2. Import **`pandas`** as **`pd`** and enter the student value as shown in *Exercise 2*, *DataFrame Computations with the Student testscore Data*. After this, convert it to a DataFrame:
   import pandas as pd
   # Create dictionary of test scores

test_dict = {'Corey':[63,75,88], 'Kevin':[48,98,92], 'Akshay': [87, 86, 85]}
# Create DataFrame
df = pd.DataFrame(test_dict)

3. Now, begin by arranging the rows of the DataFrame, as shown in the following code snippet.
   You can use the same bracket notation, **`[]`**, for rows as for lists and strings:
   # Limit DataFrame to first 2 rows
   df[0:2]
   You should get the following output:

Out[3]:

| | Corey | Kevin | Akshay |
|---|---|---|---|
| 0 | 63 | 48 | 87 |
| 1 | 75 | 98 | 86 |

Figure 6: Output of arranging rows

4. Transpose the DataFrame:
   df = df.T
   df
5. Now, rename the columns to **`Quiz_1`**, **`Quiz_2`**, and **`Quiz_3`**, which was covered in *Exercise 135*, *DataFrame Computations with the Student testscore Data*:
   # Rename Columns
   df.columns = ['Quiz_1', 'Quiz_2', 'Quiz_3']
   df
   You should get the following output:

Out[6]:

| | Quiz_1 | Quiz_2 | Quiz_3 |
|---|---|---|---|
| Corey | 63 | 75 | 88 |
| Kevin | 48 | 98 | 92 |
| Akshay | 87 | 86 | 85 |

Figure 7: Output with the renamed column names

6. Now, define a new **`DataFrame`** from the first two rows and the last two columns.

You can choose the rows and columns by name first, as shown in the following code snippet:

```
# Defining a new DataFrame from first 2 rows and last 2 columns
rows = ['Corey', 'Kevin']
cols = ['Quiz_2', 'Quiz_3']
df_spring = df.loc[rows, cols]
df_spring
```

You should get the following output:

Out[8]:

| | Quiz_1 | Quiz_2 |
|---|---|---|
| Corey | 63 | 75 |
| Kevin | 48 | 98 |

Figure 8: Output of the newly defined DataFrame

7. Now, select the first two rows and the last two columns using index numbers.

   You can use **`.iloc`** to select rows and columns by index, as shown in the following code snippet:

   ```
   # Select first 2 rows and last 2 columns using index numbers
   df.iloc[[0,1], [1,2]]
   ```

   You should get the following output:

Out[9]:

| | Quiz_2 | Quiz_3 |
|---|---|---|
| Corey | 75 | 88 |
| Kevin | 98 | 92 |

Figure 9: Output of selecting the first two rows and last two columns using index numbers

Now, add a new column to find the quiz average of our students.

You can generate new columns in a variety of ways. One way is to use available methods such as the mean. In pandas, it's important to specify the axis. An axis of 0 represents the index, and an axis of 1 represents the rows.

8. Now, create a new column as the mean, as shown in the following code snippet:
   # Define new column as mean of other columns
   df['Quiz_Avg'] = df.mean(axis=1)
   df
   You should get the following output:

Out[10]:

| | Quiz_1 | Quiz_2 | Quiz_3 | Quiz_Avg |
|---|---|---|---|---|
| Corey | 63 | 75 | 88 | 75.333333 |
| Kevin | 48 | 98 | 92 | 79.333333 |
| Akshay | 87 | 86 | 85 | 86.000000 |

Figure 10: Adding a new quiz_avg column to the output

   A new column can also be added as a list by choosing the rows and columns by name first.
9. Create a new column as a list, as shown in the following code snippet:
   df['Quiz_4'] = [92, 95, 88]
   df
   You should get the following output:

Out[11]:

| | Quiz_1 | Quiz_2 | Quiz_3 | Quiz_Avg | Quiz_4 |
|---|---|---|---|---|---|
| Corey | 63 | 75 | 88 | 75.333333 | 92 |
| Kevin | 48 | 98 | 92 | 79.333333 | 95 |
| Akshay | 87 | 86 | 85 | 86.000000 | 88 |

Figure 11: Output with a newly added column using lists

   What if you need to delete the column you created? You can do so by using the **`del`** function. It's easy to delete columns in pandas using **`del`**.
10. Now, delete the **`Quiz_Avg`** column as it is not needed anymore:
    del df['Quiz_Avg']
    df
    You should get the following output:

Out[12]:

| | Quiz_1 | Quiz_2 | Quiz_3 | Quiz_4 |
|---|---|---|---|---|
| Corey | 63 | 75 | 88 | 92 |
| Kevin | 48 | 98 | 92 | 95 |
| Akshay | 87 | 86 | 85 | 88 |

Figure 12: Output with a deleted column

In this exercise, you implemented different ways to add and remove columns as per our requirements. In the next section, you will be looking at new rows and **`NaN`**, which is an official **`numpy`** term.

# New Rows and NaN

It's not easy to add new rows to a pandas DataFrame. A common strategy is to generate a new DataFrame and then to concatenate the values.

Say you have a new student who joins the class for the fourth quiz. What values should you put for the other three quizzes? The answer is NaN. It stands for **Not a Number**

**`NaN`** is an official NumPy term. It can be accessed using **`np.NaN`**. It is case-sensitive. In later exercises, you will look at how NaN can be used. In the next exercise, you will look at concatenating and working with null values.

# Exercise 4: Concatenating and Finding the Mean with Null Values for Our testscore Data

In this exercise, you will be concatenating and finding the mean with null values for the student **`testscore`** data you

created in *Exercise 3, Computing DataFrames within DataFrames* with four quiz scores. The following steps will enable you to complete the exercise:

1. Open a new Jupyter Notebook.
2. Import **`pandas`** and **`numpy`** and create a dictionary with the **`testscore`** data to be transformed into a DataFrame, as mentioned in *Exercise 134, Using DataFrames to Manipulate Stored Student testscore Data:*

   ```
   import pandas as pd
   # Create dictionary of test scores
   test_dict = {'Corey':[63,75,88], 'Kevin':[48,98,92],
   'Akshay': [87, 86, 85]}
   # Create DataFrame
   df = pd.DataFrame(test_dict)
   ```
3. Transpose the DataFrame and rename the columns:

   ```
   df = df.T
   df
   # Rename Columns
   df.columns = ['Quiz_1', 'Quiz_2', 'Quiz_3']
   df
   ```

   You should get the following output:

   Out[5]:

   | | Quiz_1 | Quiz_2 | Quiz_3 |
   |---|---|---|---|
   | Corey | 63 | 75 | 88 |
   | Kevin | 48 | 98 | 92 |
   | Akshay | 87 | 86 | 85 |

   Figure 13: The transposed and renamed DataFrame
4. Now, add a new column, as shown in *Exercise 136, Computing DataFrames within DataFrames*:

   ```
   df['Quiz_4'] = [92, 95, 88]
   df
   ```

   You should get the following output:

Out[7]:

| | Quiz_1 | Quiz_2 | Quiz_3 | Quiz_4 |
|---|---|---|---|---|
| Corey | 63 | 75 | 88 | 92 |
| Kevin | 48 | 98 | 92 | 95 |
| Akshay | 87 | 86 | 85 | 88 |

Figure 14: Output of the added Quiz_4 column

5. Now, add a new row with the value set as **`Adrian`**, as shown in the following code snippet:
   ```
   import numpy as np
   # Create new DataFrame of one row
   df_new = pd.DataFrame({'Quiz_1':[np.NaN],
   'Quiz_2':[np.NaN], 'Quiz_3': [np.NaN], 'Quiz_4':[71]},
   index=['Adrian'])
   ```
6. Now, concatenate **`Dataframe`** with the added new row, **`Adrian`**, and display the new **`Dataframe`** value using **`df`**:
   ```
   # Concatenate DataFrames
   df = pd.concat([df, df_new])
   # Display new DataFrame
   df
   ```
   You should get the following output:

Out[7]:

| | Quiz_1 | Quiz_2 | Quiz_3 | Quiz_4 |
|---|---|---|---|---|
| Corey | 63.0 | 75.0 | 88.0 | 92 |
| Kevin | 48.0 | 98.0 | 92.0 | 95 |
| Akshay | 87.0 | 86.0 | 85.0 | 88 |
| Adrian | NaN | NaN | NaN | 71 |

Figure 15: Output with the row added to the DataFrame

You can now compute the new mean, but you must skip the **`NaN`** values; otherwise, there will be no mean score for **`Adrian`**. This will be fixed in step 7.

7. Find the **`mean`** value ignoring NaN and use these values to create a new column named **`Quiz-Avg`**, as shown in the following code snippet:
df['Quiz_Avg'] = df.mean(axis=1, skipna=True)
df
You should get the following output:

Out[8]:

| | Quiz_1 | Quiz_2 | Quiz_3 | Quiz_4 | Quiz_Avg |
|---|---|---|---|---|---|
| **Corey** | 63.0 | 75.0 | 88.0 | 92 | 79.50 |
| **Kevin** | 48.0 | 98.0 | 92.0 | 95 | 83.25 |
| **Akshay** | 87.0 | 86.0 | 85.0 | 88 | 86.50 |
| **Adrian** | NaN | NaN | NaN | 71 | 71.00 |

Figure 16: Output with quiz_4 values added to Adrian

Notice that all values are floats except for **`Quiz_4`**. There will be occasions when you need to cast all values in a particular column as another type.

# Cast Column Types

For the sake of consistency, you will do that here. Cast all the ints in **`Quiz_4`** that you used in *Exercise 137*, *Concatenating and Finding the Mean with Null Values for Our testscore Data* as floats using the following code snippet:

df.Quiz_4.astype(float)

You should get the following output:

```
In [9]: df.Quiz_4.astype(float)

Out[9]: Corey     92.0
        Kevin     95.0
        Akshay    88.0
        Adrian    71.0
        Name: Quiz_4, dtype: float64
```

Figure 17: Output with the cast column types

You can observe the change in values by examining the DataFrame yourself. Now, move on to the next topic: which is data.

# Data

Now that you have been introduced to NumPy and pandas, you will use them to analyze some real data.

Data scientists analyze data that exists in the cloud or online. One strategy is to download data directly to your computer.

## Downloading Data

Data comes in many formats, and pandas is equipped to handle most of them. In general, when looking for data to analyze, it's worth searching the keyword "dataset." A dataset is a collection of data. Online, "data" is everywhere, whereas datasets contain data in its raw format.

You will start by examining the famous Boston Housing dataset from 1980, which is available on our GitHub repository.

This dataset can be found here “HousingData.csv”

You can begin by first downloading the dataset onto our system.

## Reading Data

Now that the data is downloaded, and the Jupyter Notebook is open, you are ready to read the file. The most important part of reading a file is the extension. Our file is a **`.csv`** file. You need a method for reading **`.csv`** files.

CSV stands for Comma-Separated Values. CSV files are a popular way of storing and retrieving data, and pandas handles them very well.

Here is a list of standard data files that pandas will read, along with the code for reading data:

| type of file | code |
|---|---|
| csv files: | pd.read_csv('file_name') |
| excel files: | pd.read_excel('file_name') |
| feather files: | pd.read_feather('file_name') |
| html files: | pd.read_html('file_name') |
| json files: | pd.read_json('file_name') |
| sql database: | pd.read_sql('file_name') |

Figure 18: Standard data files that pandas read

If the files are clean, pandas will read them properly. Sometimes, files are not clean, and changing function parameters may be required. It's advisable to copy any errors and search for solutions online.

A further point of consideration is that the data should be read into a **`DataFrame`**. Pandas will convert the data into a DataFrame upon reading it, but you need to save **`DataFrame`** as a variable.

*Note*

**`df`** *is often used to store DataFrames, but it's not universal since you may be dealing with many DataFrames.*

In *Exercise 1, Reading and Viewing the Boston Housing Dataset*, you will be using the Boston Housing dataset and performing basic actions on the data.

# Exercise 5: Reading and Viewing the Boston Housing Dataset

In this exercise, your goal is to read and view the Boston Housing dataset in our Jupyter Notebook. The following steps will enable you to complete the exercise:

1. Open a new Jupyter Notebook.
2. Import **`pandas`** as **`pd`**:
   import pandas as pd
3. Now, choose a variable to store **`DataFrame`** and place the **`HousingData.csv`** file. Then, run the following command:
   housing_df = pd.read_csv('HousingData.csv')
   If no errors arise, the file has been properly read. Now you can examine and view the file.
4. Now, you need to view the file by entering the following command:
   housing_df.head()
   The **`pandas .head`** method does just that. By default, it selects the first five rows. You may enter more if you choose by adding a number in parentheses.
   You should get the following output:

| | 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 | 296 | 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 | 242 | 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 | 242 | 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 | 222 | 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 | 222 | 18.7 | 396.90 | NaN | 36.2 |

Figure 19: Output letting us view the first five rows of the dataset

Before you explore operations performed on this dataset, you may wonder what values such as **`CRIM`** and **`ZN`** mean in the dataset.

It's always nice to view data to get a general feel for things. In this particular case, you might want to know what the columns actually mean:

```
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 centers
RAD      index of accessibility to radial highways
TAX      full-value property-tax rate per $10,000
PTRATIO  pupil-teacher ratio by town
LSTAT    % lower status of the population
MEDV     Median value of owner-occupied homes in $1000's
```

Figure 20: Representation of the column values of the dataset

Now that you know what the values in the dataset mean you will start performing some advanced operations on the dataset. You will cover this in the following exercise.

# Exercise 6: Gaining Data Insights on the Boston Housing Dataset

In this exercise, you will be performing some more advanced operations and using pandas methods to understand the dataset and get desired insights. The following steps will enable you to complete the exercise:

1. Open a new Jupyter Notebook and copy the dataset file into a separate folder where you will perform this exercise.
2. Import pandas and choose a variable to store **`DataFrame`** and place the **`HousingData.csv`** file:
   import pandas as pd
   housing_df = pd.read_csv('HousingData.csv')
3. Now, use the **`describe()`** method to display key statistical measures of each column, including the mean, median, and quartiles, as shown in the following code snippet:
   housing_df.**describe**()
   You should get the following output:

| | CRIM | ZN | INDUS | CHAS | NOX | RM | AGE | DIS | RAD | TAX | PTRATIO | B | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| count | 486.000000 | 486.000000 | 486.000000 | 486.000000 | 506.000000 | 506.000000 | 486.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 506.000000 | 486.0 |
| mean | 3.611874 | 11.211934 | 11.083992 | 0.069959 | 0.554695 | 6.284634 | 68.518519 | 3.795043 | 9.549407 | 408.237154 | 18.455534 | 356.674032 | 12.71 |
| std | 8.720192 | 23.388876 | 6.835896 | 0.255340 | 0.115878 | 0.702617 | 27.999513 | 2.105710 | 8.707259 | 168.537116 | 2.164946 | 91.294864 | 7.15 |
| min | 0.006320 | 0.000000 | 0.460000 | 0.000000 | 0.385000 | 3.561000 | 2.900000 | 1.129600 | 1.000000 | 187.000000 | 12.600000 | 0.320000 | 1.73 |
| 25% | 0.081900 | 0.000000 | 5.190000 | 0.000000 | 0.449000 | 5.885500 | 45.175000 | 2.100175 | 4.000000 | 279.000000 | 17.400000 | 375.377500 | 7.12 |
| 50% | 0.253715 | 0.000000 | 9.690000 | 0.000000 | 0.538000 | 6.208500 | 76.800000 | 3.207450 | 5.000000 | 330.000000 | 19.050000 | 391.440000 | 11.43 |
| 75% | 3.560262 | 12.500000 | 18.100000 | 0.000000 | 0.624000 | 6.623500 | 93.975000 | 5.188425 | 24.000000 | 666.000000 | 20.200000 | 396.225000 | 16.95 |
| max | 88.976200 | 100.000000 | 27.740000 | 1.000000 | 0.871000 | 8.780000 | 100.000000 | 12.126500 | 24.000000 | 711.000000 | 22.000000 | 396.900000 | 37.97 |

Figure 21: Output with the desired () method

In this output, you will review the meaning of each row.

**`Count`**: The number of rows with actual values.

**`Mean`**: The sum of each entry divided by the number of entries. It is often a good estimate of the average.
**`Std`**: The number of unit entries that are expected to deviate from the mean. It is a good measure of spread.
**`Min`**: The smallest entry in each column.
**`25%`**: The first quartile. 25% of the data has a value less than this number.
**`50%`**: The median. The halfway marker of the data. It is another good estimate of the average.
**`75%`**: The third quartile. 75% of the data has a value less than this number.
**`Max`**: The largest entry in each column.

4. Now use the **`info()`** method to deliver a full list of columns.
   **`info()`** is especially valuable when you have hundreds of columns, and it takes a long time to horizontally scroll through each one:
   housing_df.**info**()
   You should get the following output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       486 non-null float64
ZN         486 non-null float64
INDUS      486 non-null float64
CHAS       486 non-null float64
NOX        506 non-null float64
RM         506 non-null float64
AGE        486 non-null float64
DIS        506 non-null float64
RAD        506 non-null int64
TAX        506 non-null int64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      486 non-null float64
MEDV       506 non-null float64
dtypes: float64(12), int64(2)
memory usage: 55.4 KB
```

Figure 22: Output with the info () method

As you can see, **`.info()`** reveals the count of non-null values in each column along with the column type. Since some columns have less than 506 non-null values, it's safe to assume that the other values are null values.

In this dataset, there are a of total of 506 rows and 14 columns. You can use the **`.shape`** attribute to obtain this information.

Now confirm the number of rows and columns in the dataset:

housing_df.**shape**

You should get the following output:

(506, 14)

This confirms that you have 506 rows and 14 columns. Notice that **`shape`** does not have any parentheses after it. This is because it's technically an attribute and pre-computed.

In this exercise, you performed basic operations on the dataset, such as describing the dataset and finding the number of rows and columns in the dataset.

In the next section, you will cover null values.

# Null Values

You need to do something about the null values. There are several popular choices when dealing with null values:

1. Eliminate the rows: A great approach if null values are a very small percentage, such as 1% of the total dataset.
2. Replace with a significant value, such as the median or the mean: A great approach if the rows are valuable, and the column is reasonably balanced.
3. Replace with the most likely value, perhaps a 0 or 1: It's preferable to option 2 when the median might be useless. The median can often work here.
   *Note*
   **mode** *is the official term for the value that occurs the greatest number of times.*

As you can see, which option you choose depends on the data.

# Exercise 7: Null Value Operations on the Dataset

In this exercise, you will perform a null value operation. You can only select the columns that have null values in our dataset:

1. Open a new Jupyter Notebook and copy the dataset file within a separate folder where you will perform this exercise.
2. Import **`pandas`** and choose a variable to store the **`DataFrame`** and place the **`HousingData.csv`** file:
   import pandas as pd
   housing_df = pd.read_csv('HousingData.csv')
3. Now, find values and columns in the dataset with **`null`** values, as shown in the following code snippet:
   housing_df.**isnull()**.any()

You should get the following output:

```
CRIM        True
ZN          True
INDUS       True
CHAS        True
NOX        False
RM         False
AGE         True
DIS        False
RAD        False
TAX        False
PTRATIO    False
B          False
LSTAT       True
MEDV       False
dtype: bool
```

Figure 23: Output of the columns with null values

The **`.isnull()`** method will display an entire **`DataFrame`** of **`True/False`** values depending on the **`Null`** value. Give it a try.

The **`.any()`** method returns the individual columns. Take it a step further and choose the **`DataFrame`** with those columns.

4. Now, using the **`DataFrame`**, find the null columns. You can use **`.loc`** to find the location of particular rows. You will select the first five rows and all of the columns that have null values, as shown in the following code snippet:

```
housing_df.loc[:5, housing_df.isnull().any()]
```

You should get the following output:

| | CRIM | ZN | INDUS | CHAS | AGE | LSTAT |
|---|---|---|---|---|---|---|
| **0** | 0.00632 | 18.0 | 2.31 | 0.0 | 65.2 | 4.98 |
| **1** | 0.02731 | 0.0 | 7.07 | 0.0 | 78.9 | 9.14 |
| **2** | 0.02729 | 0.0 | 7.07 | 0.0 | 61.1 | 4.03 |
| **3** | 0.03237 | 0.0 | 2.18 | 0.0 | 45.8 | 2.94 |
| **4** | 0.06905 | 0.0 | 2.18 | 0.0 | 54.2 | NaN |
| **5** | 0.02985 | 0.0 | 2.18 | 0.0 | 58.7 | 5.21 |

Figure 24: Output of the dataset with the null columns

Now for the final step. Use the **`.describe`** method on these particular columns but select all of the rows.

5. Use the **`.describe()`** method on the null columns of the dataset.
   The code mentioned ahead is a long piece of the code snippet. **`housing_df`** is the DataFrame. **`.loc`** allows you to specify rows and columns. **`:`** selects all rows. **`housing_df.isnull().any()`** selects only columns with null values. **`.describe()`** pulls up the statistics:
   housing_df.loc[:, housing_df.isnull().any()].describe()
   You should get the following output:

| | CRIM | ZN | INDUS | CHAS | AGE | LSTAT |
|---|---|---|---|---|---|---|
| count | 486.000000 | 486.000000 | 486.000000 | 486.000000 | 486.000000 | 486.000000 |
| mean | 3.611874 | 11.211934 | 11.083992 | 0.069959 | 68.518519 | 12.715432 |
| std | 8.720192 | 23.388876 | 6.835896 | 0.255340 | 27.999513 | 7.155871 |
| min | 0.006320 | 0.000000 | 0.460000 | 0.000000 | 2.900000 | 1.730000 |
| 25% | 0.081900 | 0.000000 | 5.190000 | 0.000000 | 45.175000 | 7.125000 |
| 50% | 0.253715 | 0.000000 | 9.690000 | 0.000000 | 76.800000 | 11.430000 |
| 75% | 3.560262 | 12.500000 | 18.100000 | 0.000000 | 93.975000 | 16.955000 |
| max | 88.976200 | 100.000000 | 27.740000 | 1.000000 | 100.000000 | 37.970000 |

Figure 25: Output with the .describe() method on particular columns but instead selects all of the rows

Consider the first column, **`CRIM`**. The mean is way more than the median (**`50%`**). This indicates that the data is very right-skewed with some outliers. Indeed, you can see that the maximum of **`88.97`** is much larger than the **`3.56`** value of the **`75th`** percentile. This makes the mean a poor replacement candidate for this column.

It turns out, after examining every column, that the median is a good candidate. Although the median is clearly not better than the mean in some cases, there are a few cases where the mean is clearly worse (**`CRIM`**, **`ZN`**, and **`CHAS`**).

The choice depends on what you ultimately want to do with the data. If the goal is a straightforward data analysis, eliminating the rows with null values is worth consideration. However, if the goal is to use machine learning to predict data, then perhaps more is to be gained by changing the null values to a suitable replacement. It's impossible to know in advance.

A more thorough examination could be warranted depending on the data. For instance, if analyzing new medical drugs, it would be worth putting more time and energy into appropriately dealing with null values. You may want to perform more analysis to determine whether a value is 0 or 1, depending upon other factors.

In this particular case, replacing all the null values with the median is warranted. You can have a look at this in the following example.

# Replacing Null Values

Pandas include a nice method, **`fillna`**, which can be used to replace null values. It works for individual columns and entire DataFrames. You will use three approaches, replacing the null values of a column with the mean, replacing the null values of a column with another value, and replacing all the null values in the entire dataset with the median. You will use the same Boston Housing dataset.

Replace null column values with **`mean`**:

housing_df['AGE'] =
housing_df['AGE'].**fillna**(housing_df.mean())

Then, replace null column values with a value:

housing_df['CHAS'] = housing_df['CHAS'].**fillna**(0)

Replace null DataFrame values with **`median`**:

housing_df = housing_df.**fillna**(housing_df.median())

Finally, check that all null values have been replaced:

housing_df.info()

You should get the following output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM       506 non-null float64
ZN         506 non-null float64
INDUS      506 non-null float64
CHAS       506 non-null float64
NOX        506 non-null float64
RM         506 non-null float64
AGE        506 non-null float64
DIS        506 non-null float64
RAD        506 non-null int64
TAX        506 non-null int64
PTRATIO    506 non-null float64
B          506 non-null float64
LSTAT      506 non-null float64
MEDV       506 non-null float64
dtypes: float64(12), int64(2)
memory usage: 55.4 KB
```

Figure 26: Output with the null values eliminated

After eliminating all null values, the dataset is much cleaner. There may also be unrealistic outliers or extreme outliers that will lead to poor prediction. These can often be detected through visual analysis, which you will be covering in the next section.