



|| JAI SRI GURUDEV ||

Sri Adichunchanagiri Shikshana Trust®

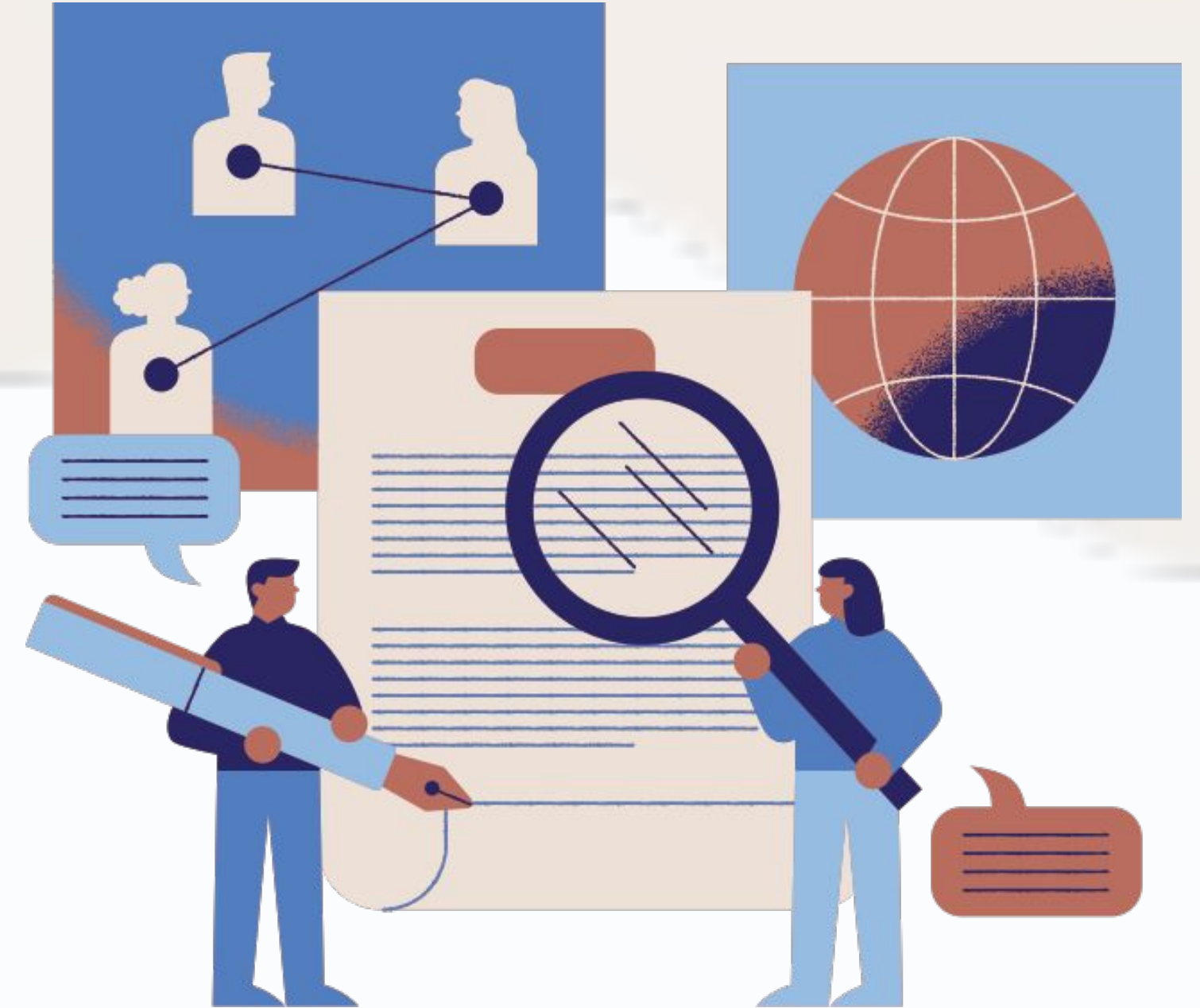
**SJB INSTITUTE OF TECHNOLOGY**

AN AUTONOMOUS INSTITUTE UNDER VISVESVARAYA TECHNOLOGICAL UNIVERSITY



# Course: Exploratory Data Analytics

## Course Code: 23CSE422



Mrs. Shilpashree S,  
Assistant Professor  
Department of CS & E

# Data Transformation - II

## Module 2



# Data Transformation Techniques

We will explore how to handle duplicate rows in a DataFrame, which is an essential part of data cleaning. We'll learn how to identify and remove duplicates to ensure the quality of our dataset.

Example:

Step 1: Create a Simple DataFrame:

```
# Create a simple DataFrame with duplicate entries
frame3 = pd.DataFrame({'column 1': ['Looping'] * 3 + ['Functions'] * 4,
                        'column 2': [10, 10, 22, 23, 23, 24, 24]})

# Display the DataFrame
frame3
```

	column 1	column 2
0	Looping	10
1	Looping	10
2	Looping	22
3	Functions	23
4	Functions	23
5	Functions	24
6	Functions	24



# Data Transformation Techniques

Example (Contd.):

Step 2: Identify Duplicates with duplicated()

To check for duplicate rows, we can use the duplicated() method, which returns a Boolean series indicating whether each row is a duplicate of a previous row.

Code:

```
# Identify duplicates  
frame3.duplicated()
```

**Note:** The rows that return True are duplicates (rows 1, 4, and 6).

	0
0	False
1	True
2	False
3	False
4	True
5	False
6	True

dtype: bool

# Data Transformation Techniques

Example (Contd.):

Step 3: Remove Duplicates with drop\_duplicates()

To remove duplicates, we use the drop\_duplicates() method:

Code:

```
# Remove duplicates
frame4 = frame3.drop_duplicates()

# Display the DataFrame without duplicates
frame4
```

	column 1	column 2
0	Looping	10
2	Looping	22
3	Functions	23
5	Functions	24

**Note: Rows 1, 4, and 6 are removed, and duplicates are eliminated by considering all columns.**

# Data Transformation Techniques

Example (Contd.):

Step 4: Remove Duplicates Based on Specific Columns

Instead of removing duplicates based on all columns, we can specify a subset of columns to check for duplicates:

Code:

```
# Add a new column for demonstration
frame3['column 3'] = range(7)

# Remove duplicates based on 'column 2'
frame5 = frame3.drop_duplicates(['column 2'])

# Display the result
frame5
```

	column 1	column 2	column 3
0	Looping	10	0
2	Looping	22	2
3	Functions	23	3
5	Functions	24	5

**Note: Rows 1, 4, and 6 are removed, and duplicates are eliminated by considering 2nd column.**

# Data Transformation Techniques

Example (Contd.):

Step 5: Keep the Last Occurrence with `take_last=True` (**Deprecated**)

By default, `drop_duplicates()` keeps the first occurrence of each duplicate. However, you can modify this behavior by setting the `take_last=True` argument, which keeps the last occurrence instead.

Code:

```
# Remove duplicates and keep the last occurrence
```

```
frame6 = frame3.drop_duplicates(['column 2'], take_last=True)
```

```
# Display the result
```

```
frame6
```



# Data Transformation Techniques

Example (Contd.):

Step 5.1: Keep the Last Occurrence with keep="last"

To remove duplicates and keep the last occurrence, use the keep parameter in the drop\_duplicates() method:

Code:

```
# Remove duplicates and keep the last occurrence  
frame6 = frame3.drop_duplicates(['column 2'], keep='last')  
  
# Display the result  
frame6
```

	column 1	column 2	column 3
1	Looping	10	1
2	Looping	22	2
4	Functions	23	4
6	Functions	24	6



# Replacing Values

- During data analysis, you may need to replace specific values in a DataFrame for cleaning or standardizing data.
- Pandas provides the `replace()` method for this purpose.

## Single Value Replacement

Replace one specific value with another.

Code:

```
import pandas as pd
import numpy as np

replaceFrame = pd.DataFrame({
    'column 1': [200., 3000., -786., 3000., 234., 444., -786., 332., 3332.],
    'column 2': range(9)
})

# Replace -786 with NaN
replaceFrame.replace(to_replace=-786, value=np.nan)
```

	column 1	column 2
0	200.0	0
1	3000.0	1
2	NaN	2
3	3000.0	3
4	234.0	4
5	444.0	5
6	NaN	6
7	332.0	7
8	3332.0	8

# Replacing Values

- Multiple Value Replacement

Replace multiple values at once.

Code:

```
# Replace -786 with NaN and 0 with 2  
replaceFrame.replace(to_replace=[-786, 0], value=[np.nan, 2])
```

## Practical Use Cases

- Replacing missing values (NaN) with default values.
- Standardizing data by replacing inconsistent entries.
- Cleaning invalid data points.

	column 1	column 2
0	200.0	2
1	3000.0	1
2	NaN	2
3	3000.0	3
4	234.0	4
5	444.0	5
6	NaN	6
7	332.0	7
8	3332.0	8

# Handling Missing Data

## Introduction to Missing Data

- Missing values in a DataFrame are represented as NaN (Not a Number).

## Reasons for missing data:

- It can happen when data is retrieved from an external source and there are some incomplete values in the dataset.
- It can also happen when we join two different datasets and some values are not matched.
- Missing values due to data collection errors.
- When the shape of data changes, there are new additional rows or columns that are not determined.
- Reindexing of data can result in incomplete data.

## Example:

```
data = np.arange(15, 30).reshape(5, 3)
```

```
dfx = pd.DataFrame(data,  
                    index=['apple', 'banana', 'kiwi', 'grapes', 'mango'],  
                    columns=['store1', 'store2', 'store3'])
```

```
print(dfx)
```

## Note:

- Represents sales of different fruits across stores.
- No missing values initially.

	store1	store2	store3
apple	15	16	17
banana	18	19	20
kiwi	21	22	23
grapes	24	25	26
mango	27	28	29



# Handling Missing Data

Adding Missing Values:

Let's add some missing values to our dataframe:

Code:

```
dfx['store4'] = np.nan
dfx.loc['watermelon'] = np.arange(15, 19)
dfx.loc['oranges'] = np.nan
dfx['store5'] = np.nan
dfx['store4']['apple'] = 20.
print(dfx)
```

	store1	store2	store3	store4	store5
apple	15.0	16.0	17.0	20.0	NaN
banana	18.0	19.0	20.0	NaN	NaN
kiwi	21.0	22.0	23.0	NaN	NaN
grapes	24.0	25.0	26.0	NaN	NaN
mango	27.0	28.0	29.0	NaN	NaN
watermelon	15.0	16.0	17.0	18.0	NaN
oranges	NaN	NaN	NaN	NaN	NaN

Changes Made:

- Added store4 and store5 (new columns).
- Added watermelon and oranges (new rows).
- Sales for store4 (apple: 20 kg, watermelon: data available).
- store5 and oranges: No data available (all NaN).

## Characteristics of Missing Data

1. Entire row can contain NaN (e.g., oranges).
2. Entire column can contain NaN (e.g., store5).
3. Some values in both row and column can be NaN (e.g., store4, mango).



# NaN values in pandas objects

We can use the `isnull()` and `notnull()` function from the pandas library to identify NaN values.

Code:

```
dfx.isnull()
```

**Note:** True values indicate the values that are NaN.

	store1	store2	store3	store4	store5
apple	False	False	False	False	True
banana	False	False	False	True	True
kiwi	False	False	False	True	True
grapes	False	False	False	True	True
mango	False	False	False	True	True
watermelon	False	False	False	False	True
oranges	True	True	True	True	True

Code:

```
dfx.notnull()
```

**Note:** Complements `isnull()`, showing True for non-NaN values.

	store1	store2	store3	store4	store5
apple	True	True	True	True	False
banana	True	True	True	False	False
kiwi	True	True	True	False	False
grapes	True	True	True	False	False
mango	True	True	True	False	False
watermelon	True	True	True	True	False
oranges	False	False	False	False	False

# NaN values in pandas objects

Counting NaN Values:

We can use the `sum()` method to count the number of NaN values in each store.

1. **Count NaN values per column:**

Code:

```
dfx.isnull().sum()
```

```
0
store1  1
store2  1
store3  1
store4  5
store5  7
dtype: int64
```

2. **Total NaN values in the DataFrame:**

Code:

```
dfx.isnull().sum().sum()
```

Output: Total Number of NaN Values:15

3. **instead of counting the number of missing values, we can count the number of reported values.**

Code:

```
print("Number of Non-null values")
dfx.count()
```

```
Number of Non-null values
0
store1  6
store2  6
store3  6
store4  2
store5  0
dtype: int64
```

# Dropping Missing Values

One of the ways to handle missing values is to simply remove them from our dataset.

## 1. Identifying Non-NaN Values

- Use `notnull()` to filter out non-NaN values in a specific column:

Code:

```
dfx.store4[dfx.store4.notnull()]
```

store4	
apple	20.0
watermelon	18.0

**dtype:** float64



# Dropping Missing Values

## Dropping Rows with NaN Values

- we can use the `dropna()` method to remove the rows:

Code:

```
dfx.store4.dropna()
```

	store4
apple	20.0
watermelon	18.0

**dtype:** float64

Note: The `dropna()` method just returns a copy of the dataframe by dropping the rows with NaN. The original dataframe is not changed.

If `dropna()` is applied to the entire dataframe, then it will drop all the rows from the dataframe, because there is at least one NaN value in our dataframe:

Code:

```
dfx.dropna() #The output of this code is an empty dataframe.
```

store1	store2	store3	store4	store5
--------	--------	--------	--------	--------



# Dropping Missing Values

Dropping by rows:

- Use `how='all'` to drop only rows where all values are NaN

Note: Rows with NaN for all columns (e.g., oranges) are removed

	store1	store2	store3	store4	store5
apple	15.0	16.0	17.0	20.0	NaN
banana	18.0	19.0	20.0	NaN	NaN
kiwi	21.0	22.0	23.0	NaN	NaN
grapes	24.0	25.0	26.0	NaN	NaN
mango	27.0	28.0	29.0	NaN	NaN
watermelon	15.0	16.0	17.0	18.0	NaN

Output:

- Rows with NaN for all columns (e.g., oranges) are removed.

# Dropping Missing Values

## Dropping Columns with All NaN Values

- Specify axis=1 to drop columns where all values are NaN

	<b>store1</b>	<b>store2</b>	<b>store3</b>	<b>store4</b>
<b>apple</b>	15.0	16.0	17.0	20.0
<b>banana</b>	18.0	19.0	20.0	NaN
<b>kiwi</b>	21.0	22.0	23.0	NaN
<b>grapes</b>	24.0	25.0	26.0	NaN
<b>mango</b>	27.0	28.0	29.0	NaN
<b>watermelon</b>	15.0	16.0	17.0	18.0
<b>oranges</b>	NaN	NaN	NaN	NaN

Output:

Columns like store5 are removed as they contain only NaN.

# Dropping Missing Values

## Dropping Columns Based on a Threshold

- Use thresh to define the minimum non-NaN values required to keep a column:

Code:

```
dfx.dropna(thresh=5, axis=1)
```

	<b>store1</b>	<b>store2</b>	<b>store3</b>
<b>apple</b>	15.0	16.0	17.0
<b>banana</b>	18.0	19.0	20.0
<b>kiwi</b>	21.0	22.0	23.0
<b>grapes</b>	24.0	25.0	26.0
<b>mango</b>	27.0	28.0	29.0
<b>watermelon</b>	15.0	16.0	17.0
<b>oranges</b>	NaN	NaN	NaN

# Mathematical Operations with NaN

## Handling NaN in Mathematical Operations

- NumPy: Returns NaN when encountering NaN values.
- Pandas: Ignores NaN values and processes available data.

## Comparing NumPy and Pandas Behavior:

Code:

```
ar1 = np.array([100, 200, np.nan, 300])
```

```
ser1 = pd.Series(ar1)
```

```
ar1.mean(), ser1.mean()
```

```
(nan, 200.0)
```



# Mathematical Operations with NaN

Summing Values in a Series with NaN

- Compute total sales from store4:

Code:

```
ser2 = dfx.store4  
ser2.sum()
```

Output: 38

Note: NaN values are treated as 0 during summing.

# Mathematical Operations with NaN

Calculating the Mean:

Compute average sales from store4

Code:

```
ser2.mean()
```

Output: 19

Cumulative Summing with NaN

- Example of cumulative sum:

Code:

```
ser2.cumsum()
```

	store4
apple	20.0
banana	NaN
kiwi	NaN
grapes	NaN
mango	NaN
watermelon	38.0
oranges	NaN
dtype: float64	

# Mathematical Operations with NaN

Key Points to Remember:



- NumPy: Operations result in NaN if any NaN values are present.
- Pandas: Ignores NaN in most operations, treating them as 0 (e.g., in sum and mean).
- Cumulative functions like `cumsum()` exclude NaN values but maintain their positions.

# Filling missing values

- Missing values (NaN) can distort analysis and lead to misleading results.
- Common methods to handle missing values:
  - Dropping rows or columns.
  - Filling with specific values or statistical measures.
- Using the fillna() Method:
  - Replace NaN values with specified values using fillna().

Code:

```
filledDf = dfx.fillna(0)  
filledDf
```



	store1	store2	store3	store4	store5
apple	15.0	16.0	17.0	20.0	0.0
banana	18.0	19.0	20.0	0.0	0.0
kiwi	21.0	22.0	23.0	0.0	0.0
grapes	24.0	25.0	26.0	0.0	0.0
mango	27.0	28.0	29.0	0.0	0.0
watermelon	15.0	16.0	17.0	18.0	0.0
oranges	0.0	0.0	0.0	0.0	0.0



# Filling missing values

- Impact of Filling NaN with 0:

- Replacing NaN with 0 can affect statistical calculations:

- Mean
- Sum
- Median

- Code: `dfx.mean()`

```
store1  20.0
store2  21.0
store3  22.0
store4  19.0
store5  NaN
dtype: float64
```

- Comparison After Filling NaN with 0:

- Recalculating mean after replacing NaN
- Code: `filledDf.mean()`

```
store1  17.142857
store2  18.000000
store3  18.857143
store4   5.428571
store5   0.000000
dtype: float64
```

Slightly different values indicate that filling NaN with 0 may not always be optimal.

# Backward and forward filling

- Forward Filling:
  - Replaces NaN values with the last known value.
- Backward Filling:
  - Replaces NaN values with the next known value.
- Used when missing values can reasonably be inferred from surrounding data.

- **Forward-Filling Technique:**

- **Code:**

```
dfx.store4.fillna(method='ffill')
```

- Last known value (20.0) is used to fill missing values sequentially.

apple	20.0
banana	20.0
kiwi	20.0
grapes	20.0
mango	20.0
watermelon	18.0
oranges	18.0

Name: store4, dtype: float64

## Backward-Filling Technique:

**Code:** `dfx.store4.fillna(method='bfill')`

- Next known value (18.0) is used to fill missing values sequentially.

apple	20.0
banana	18.0
kiwi	18.0
grapes	18.0
mango	18.0
watermelon	18.0
oranges	NaN

Name: store4, dtype: float64

# Backward and forward filling

## Key Differences Between ffill and bfill:

Aspect	Forward Fill ( <code>ffill</code> )	Backward Fill ( <code>bfill</code> )
Fill Direction	Uses the last known value	Uses the next known value
Example Fill	NaN → Last Known Value	NaN → Next Known Value

## Key Takeaways

- Forward-filling (`ffill`) propagates the last valid value forward.
- Backward-filling (`bfill`) propagates the next valid value backward.
- These techniques are simple but require careful application to avoid introducing bias.



# Interpolating missing values:

Interpolating missing values:

- Interpolation: A technique to estimate missing values based on other data points.
- pandas provides the `interpolate()` function for both Series and DataFrames.
- By default, `interpolate()` performs linear interpolation.

Example of Linear Interpolation

Code:

```
ser3 = pd.Series([100, np.nan, np.nan, np.nan, 292])
```

```
ser3.interpolate()
```

```
0    100.0
```

```
1    148.0
```

```
2    196.0
```

```
3    244.0
```

```
4    292.0
```

```
dtype: float64
```

# Interpolating missing values:

## Linear Interpolation:

- Interpolation uses the first and last known values around missing data.

- Formula:

$$\text{Step Size} = \frac{\text{Last Known Value} - \text{First Known Value}}{\text{Number of Steps}}$$

- In the example:

$$\text{Step Size} = \frac{292 - 100}{5 - 1} = 48$$

- Next value after 100:  $100 + 48 = 148$
- Subsequent values: Add 48 to each preceding value.

- pandas supports advanced interpolation methods, including:

- Polynomial interpolation
- Time series interpolation
- Spline interpolation

# Renaming Indexes Using map()

## Renaming Indexes Using map()

- What is map()?
  - map() is a method used to apply a function to each element of a pandas Series, including the index.
  - It is useful when you want to transform all index labels based on a function.
- Example:

```
dframe1.index = dframe1.index.map(str.upper)
```

## Explanation:

- The str.upper function is applied to every element of the index, converting all labels to uppercase.
- This is useful for standardizing the format of index labels.

## Output:

- After applying map(), the index labels are converted to uppercase.



	Bergen	Oslo	Trondheim	Stavanger	Kristiansand
RAINFALL	0	1	2	3	4
HUMIDITY	5	6	7	8	9
WIND	10	11	12	13	14

The index labels have been transformed to uppercase without modifying the data.



# Renaming Indexes Using rename()

## Renaming Index and Columns Using rename()

- What is rename()?
  - The rename() method allows you to modify both row (index) and column labels.
  - Unlike map(), rename() is more flexible and can apply different functions to both the index and columns simultaneously.
- Example:

```
dframe1.rename(index=str.title, columns=str.upper)
```
- Explanation:
  - The index=str.title part of the code transforms the index labels into title case (capitalizes the first letter of each word).
  - The columns=str.upper part transforms the column labels into uppercase.
  - This is useful when you want to clean or format the labels for better readability.

## Output:

- After applying rename(), both the index and columns are modified: The index is transformed to title case, and the columns are transformed to uppercase.

	BERGEN	OSLO	TRONDHEIM	STAVANGER	KRISTIANSAND
Rainfall	0	1	2	3	4
Humidity	5	6	7	8	9
Wind	10	11	12	13	14

# Renaming Indexes Using map()

Aspect	map ( )	rename ( )
Purpose	Transforms index labels based on a function.	Renames both index and column labels, allowing transformations.
Effect	Modifies the index directly. Can only modify the index (row labels).	Modifies both index and columns, and allows more complex transformations. Does not modify original data unless reassigned.
Flexibility	Limited to simple transformations like changing case.	Highly flexible, can handle complex transformations, e.g., applying different functions to rows and columns.
Use Case	Simple transformations like case conversion or replacing elements.	Ideal for renaming and formatting both rows and columns simultaneously.

# Renaming Indexes Using map()

## Practical Applications of map() and rename()

### Use of map():

- Transforming index labels in simple ways (e.g., converting all labels to lowercase, title case, or replacing specific words).

### Example:

```
df.index = df.index.map(lambda x: x.replace('Store', 'Shop'))
```

### Use of rename():

- Renaming columns with more complex transformations, e.g., applying a function to each label separately.

### Example:

```
df.rename(columns=lambda x: x.lower(), index=str.title)
```

## When to Use Each?

- Use map() when you need to perform simple, element-wise transformations on the index.
- Use rename() when you need more flexibility, such as renaming both index and columns at the same time with more complex functions.



# Discretization and Binning

## Introduction:

- Discretization: The process of converting continuous data into discrete intervals.
- Binning: The intervals into which continuous data is divided. Each interval is called a bin.

## Example:

Given a dataset of heights:

height = [120, 122, 125, 127, 121, 123, 137, 131, 161, 145, 141, 132]

Goal: Convert this dataset into the following bins:

- (118 to 125)
- (126 to 135)
- (136 to 160)
- (160 and above)

# Discretization and Binning

Using Pandas cut() for Binning:

The cut() method is used to bin data based on specified intervals.

Code:

```
import pandas as pd
height = [120, 122, 125, 127, 121, 123, 137, 131, 161, 145, 141, 132]
bins = [118, 125, 135, 160, 200]
category = pd.cut(height, bins)
print(category)
```

Output:

```
(118, 125], (118, 125], (118, 125], (125, 135], (118, 125], ...
(125, 135], (160, 200], (135, 160], (135, 160], (125, 135]]
```

Length: 12

Categories (4, interval[int64]): [(118, 125] < (125, 135] < (135, 160] < (160, 200]]

- The categories are represented as intervals. For instance, (118, 125] means that 118 is excluded (open interval), but 125 is included (closed interval).

# Discretization and Binning

Changing Interval Type:

By using the `right=False` argument, we can change the form of the interval to left-closed, right-open.

Code:

```
category2 = pd.cut(height, [118, 126, 136, 161, 200], right=False)
print(category2)
```

Output:

```
[118, 126), [118, 126), [118, 126), [126, 136), [118, 126), ...
[126, 136), [161, 200), [136, 161), [136, 161), [126, 136)]
```

Left-closed, right-open intervals: `[118, 126)` means that 118 is included, but 126 is excluded.

# Discretization and Binning

Counting Values in Each Bin:

To count how many values fall into each bin, use `pd.value_counts()`:

Code:

```
pd.value_counts(category)
```

Output:

```
(118, 125]  5  
(135, 160]  3  
(125, 135]  3  
(160, 200]  1  
dtype: int64
```

This shows the number of values in each bin.



# Discretization and Binning

Adding Custom Bin Labels:

We can label the bins with custom names by passing a list of labels:

Code:

```
bin_names = ['Short Height', 'Average height', 'Good Height', 'Taller']  
pd.cut(height, bins, labels=bin_names)
```

Output:

```
[Short Height, Short Height, Short Height, Average height, Short Height, ...  
Average height, Taller, Good Height, Good Height, Average height]
```

```
Length: 12
```

```
Categories (4, object): [Short Height < Average height < Good Height < Taller]
```

The data is now labeled with the corresponding bin names.

# Discretization and Binning

## Equal-Length Bins:

If we pass an integer instead of a specific list of bins, `cut()` will automatically create equal-length bins based on the data's range:

## Code:

```
import numpy as np
pd.cut(np.random.rand(40), 5, precision=2)
```

## Output:

```
[(0.81, 0.99], (0.094, 0.27], (0.81, 0.99], (0.45, 0.63], (0.63, 0.81], ...
(0.81, 0.99], (0.45, 0.63], (0.45, 0.63], (0.81, 0.99], (0.81, 0.99]]
```

Length: 40

Categories (5, interval[float64]): [(0.094, 0.27] < (0.27, 0.45] < (0.45, 0.63] < (0.63, 0.81] < (0.81, 0.99]]

The data is divided into 5 equal-length bins.

# Discretization and Binning

## Quantile Binning with qcut()

The qcut() method divides data into equal-sized bins based on quantiles.

Code:

```
randomNumbers = np.random.rand(2000)
category3 = pd.qcut(randomNumbers, 4) # Divide into quartiles
print(category3)
```

Output:

```
[(0.77, 0.999], (0.261, 0.52], (0.261, 0.52], (-0.000565, 0.261], ...
```

Length: 2000

Categories (4, interval[float64]): [(-0.000565, 0.261] < (0.261, 0.52] < (0.52, 0.77] < (0.77, 0.999]]

The data is divided into 4 quartiles.

# Discretization and Binning

## Custom Quantile Binning:

You can also specify custom quantiles for binning:

Code:

```
pd.qcut(randomNumbers, [0, 0.3, 0.5, 0.7, 1.0])
```

Output:

```
[(0.722, 0.999], (-0.000565, 0.309], (0.309, 0.52], (-0.000565, 0.309], ...
```

Length: 2000

```
Categories (4, interval[float64]): [(-0.000565, 0.309] < (0.309, 0.52] < (0.52, 0.722] < (0.722, 0.999]]
```

- The data is now divided into 4 categories based on the custom quantiles provided.



# Outlier Detection and Filtering

## What are Outliers?

- Outliers are data points that deviate significantly from other observations.

## Examples:

- A customer spending \$1,000,000 in a small grocery store.
- A temperature reading of  $-100^{\circ}\text{C}$  in a dataset of summer temperatures.

## Why Do They Occur?

- Errors: Mistakes in data entry or measurement.
- Variability: Natural but extreme deviations in the data.
- Unusual Events: Rare, genuine anomalies.

## Why Detect and Filter Outliers?

- Impact of Outliers:
  - Skew statistical results (e.g., mean or standard deviation).
  - Reduce the performance of machine learning models.
  - Mislead insights during decision-making.
- Benefits of Handling Outliers:
  - Improve data quality and analysis accuracy.
  - Focus on typical patterns in data.

# Outlier Detection and Filtering

## Dataset Example

- Source: The dataset can be downloaded from the GitHub repository [here](#).
- Loading the Dataset in Python:

Code:

```
import pandas as pd
```

```
df
```

```
=
```

```
pd.read_csv('https://raw.githubusercontent.com/PacktPublishing/hands-on-exploratory-data-analysis-with-python/  
master/Chapter%204/sales.csv')
```

```
df.head(10)
```

- Dataset Description:
  - Columns: Include UnitPrice, Quantity, and TotalPrice.
  - Purpose: Demonstrate detection of unusual transactions.
- Output Preview:

	Account	Company	Order	SKU	Country	Year	Quantity	UnitPrice	transactionComple
0	123456779	Kulas Inc	99985	s9-supercomputer	Aruba	1981	5148	545	F
1	123456784	GitHub	99986	s4-supercomputer	Brazil	2001	3262	383	F
2	123456782	Kulas Inc	99990	s10-supercomputer	Montserrat	1973	9119	407	
3	123456783	My SQ Man	99999	s1-supercomputer	El Salvador	2015	3097	615	F
4	123456787	ABC Dogma	99996	s6-supercomputer	Poland	1970	3356	91	
5	123456778	Super Sexy Dingo	99996	s9-supercomputer	Costa Rica	2004	2474	136	
6	123456783	ABC Dogma	99981	s11-supercomputer	Spain	2006	4081	195	F
7	123456785	ABC Dogma	99998	s9-supercomputer	Belarus	2015	6576	603	F
8	123456778	Loolo INC	99997	s8-supercomputer	Mauritius	1999	2460	36	F
9	123456775	Kulas Inc	99997	s7-supercomputer	French Guiana	2004	1831	664	

# Outlier Detection and Filtering

## Adding a New Column: TotalPrice

- Objective: Create a TotalPrice column as the product of UnitPrice and Quantity.

Code:

```
df['TotalPrice'] = df['UnitPrice'] * df['Quantity']  
df.head()
```

- A new column is added to the DataFrame.
- Result:

	Account	Company	Order	SKU	Country	Year	Quantity	UnitPrice	transactionComplete	TotalPrice
0	123456779	Kulas Inc	99985	s9-supercomputer	Aruba	1981	5148	545	False	2805660
1	123456784	GitHub	99986	s4-supercomputer	Brazil	2001	3262	383	False	1249346
2	123456782	Kulas Inc	99990	s10-supercomputer	Montserrat	1973	9119	407	True	3711433
3	123456783	My SQ Man	99999	s1-supercomputer	El Salvador	2015	3097	615	False	1904655
4	123456787	ABC Dogma	99996	s6-supercomputer	Poland	1970	3356	91	True	305396
5	123456778	Super Sexy Dingo	99996	s9-supercomputer	Costa Rica	2004	2474	136	True	336464
6	123456783	ABC Dogma	99981	s11-supercomputer	Spain	2006	4081	195	False	795795
7	123456785	ABC Dogma	99998	s9-supercomputer	Belarus	2015	6576	603	False	3965328
8	123456778	Loolo INC	99997	s8-supercomputer	Mauritius	1999	2460	36	False	88560
9	123456775	Kulas Inc	99997	s7-supercomputer	French Guiana	2004	1831	664	True	1215784



# Outlier Detection and Filtering

## Detecting Outliers Based on TotalPrice

- Defining Outliers:
  - Transactions with TotalPrice > 3,000,000 are considered outliers.

Code:

```
TotalTransaction = df["TotalPrice"]  
outliers = TotalTransaction[np.abs(TotalTransaction) > 3000000]  
print(outliers)
```

- np.abs() ensures the condition works for absolute values.
- The output highlights TotalPrice values exceeding the threshold.

```
2  3711433  
7  3965328  
13 4758900  
15 5189372  
17 3989325  
...  
9977 3475824  
9984 5251134  
9987 5670420  
9991 5735513  
9996 3018490  
Name: TotalPrice, Length: 2094, dtype: object
```



# Outlier Detection and Filtering

## Displaying Rows with Extreme Outliers

- Objective: Display rows where TotalPrice > 6,741,112.

Code:

```
extreme_outliers = df[np.abs(TotalTransaction) > 6741112]  
print(extreme_outliers)
```

	Account	Company	Order	SKU	Country	Year	Quantity	UnitPrice	transactionComplete	TotalPrice
818	123456781	Gen Power	99991	s1-supercomputer	Burkina Faso	1985	9693	696	False	6746328
1402	123456778	Will LLC	99985	s11-supercomputer	Austria	1990	9844	695	True	6841580
2242	123456770	Name IT	99997	s9-supercomputer	Myanmar	1979	9804	692	False	6784368
2876	123456772	Gen Power	99992	s10-supercomputer	Mali	2007	9935	679	False	6745865
3210	123456782	Loolo INC	99991	s8-supercomputer	Kuwait	2006	9886	692	False	6841112
3629	123456779	My SQ Man	99980	s3-supercomputer	Hong Kong	1994	9694	700	False	6785800
7674	123456781	Loolo INC	99989	s6-supercomputer	Sri Lanka	1994	9882	691	False	6828462
8645	123456789	Gen Power	99996	s11-supercomputer	Suriname	2005	9742	699	False	6809658
8684	123456785	Gen Power	99989	s2-supercomputer	Kenya	2013	9805	694	False	6804670

# Permutation and Random Sampling

## Introduction:

- Permutation: Randomly rearranging the order of elements in a series or dataframe.
- Random Sampling: Selecting a random subset of elements from a series or dataframe.

## Example of a DataFrame

We start by creating a DataFrame with NumPy and Pandas:

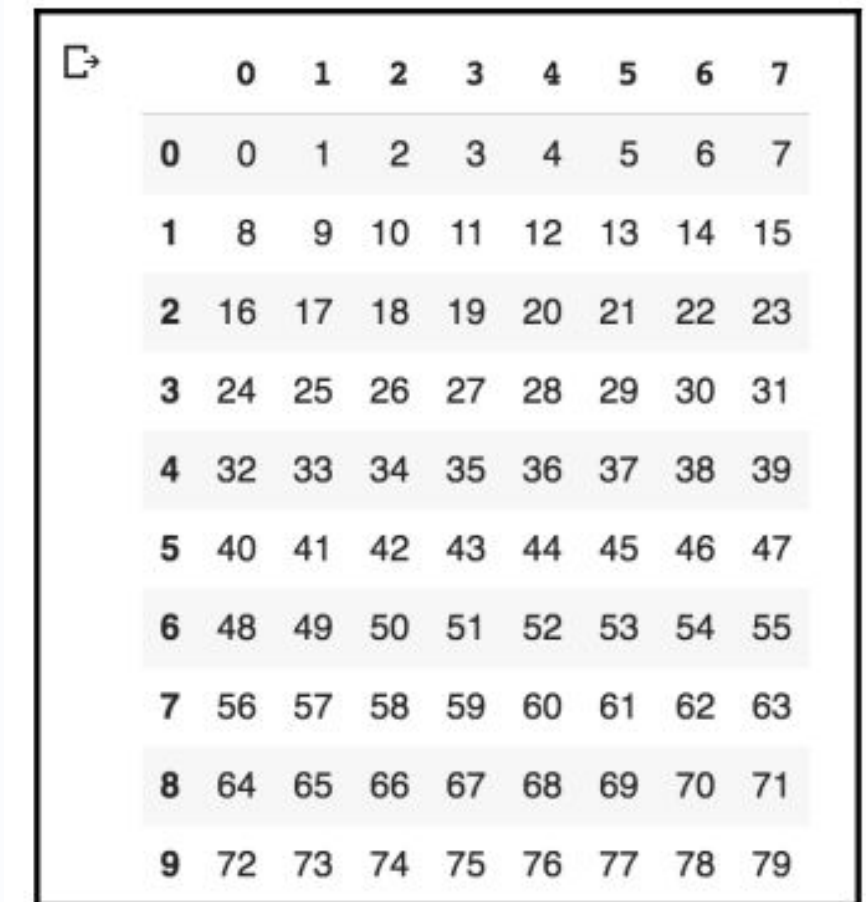
### Code:

```
import numpy as np
import pandas as pd
```

```
dat = np.arange(80).reshape(10, 8) # Creating an array with 10 rows and 8 columns
```

```
df = pd.DataFrame(dat) # Convert array to DataFrame
```

```
df
```



	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63
8	64	65	66	67	68	69	70	71
9	72	73	74	75	76	77	78	79

# Permutation and Random Sampling

## Random Permutation of DataFrame Rows

Next, we use `numpy.random.permutation()` to permute the rows of the DataFrame.

### Code:

```
sampler = np.random.permutation(10) # Permutes the indices of the DataFrame  
sampler
```

### Output:

```
array([1, 5, 3, 6, 2, 4, 9, 0, 7, 8])
```

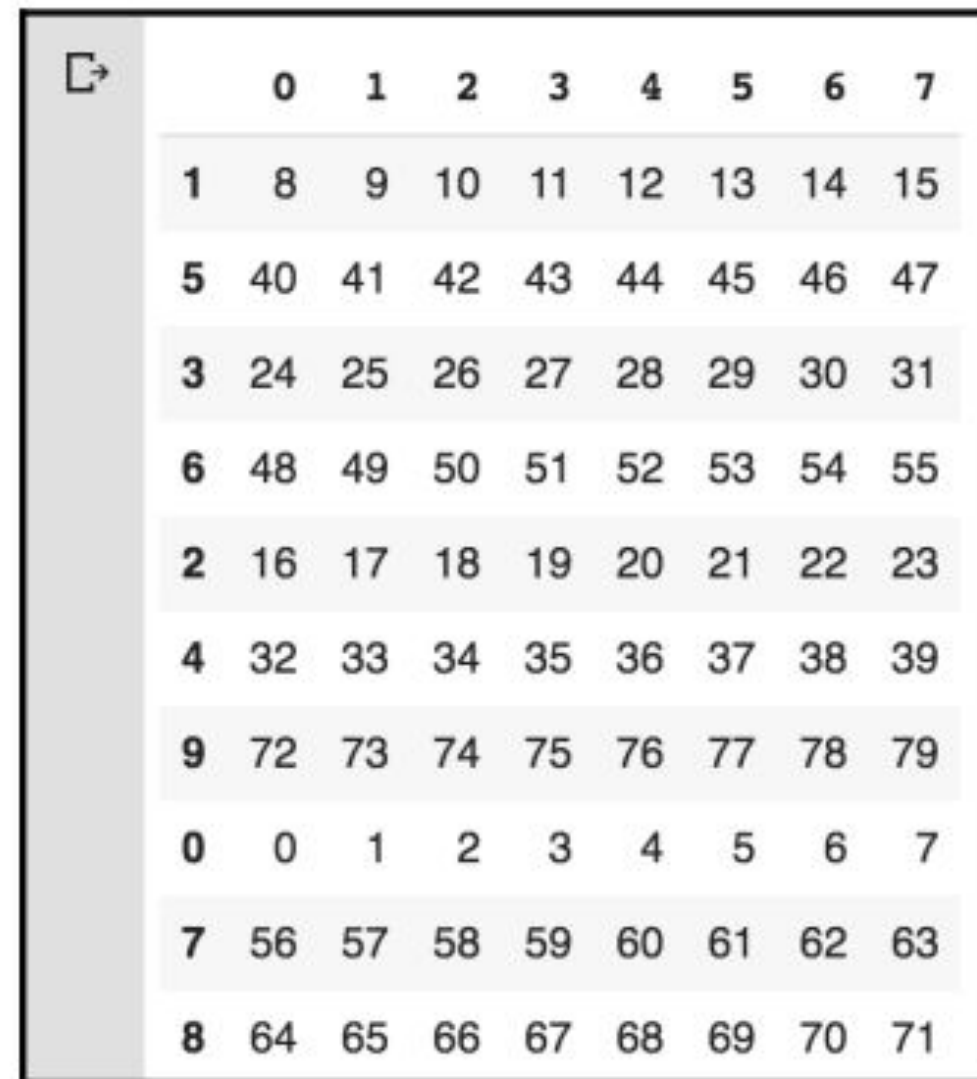
- This array indicates the new order of the rows.

# Permutation and Random Sampling

## Using Permutation for Row Reordering

Now, we use the `take()` function from Pandas along with the permuted indices to reorder the DataFrame rows:

Code: `df.take(sampler)`



	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
5	40	41	42	43	44	45	46	47
3	24	25	26	27	28	29	30	31
6	48	49	50	51	52	53	54	55
2	16	17	18	19	20	21	22	23
4	32	33	34	35	36	37	38	39
9	72	73	74	75	76	77	78	79
0	0	1	2	3	4	5	6	7
7	56	57	58	59	60	61	62	63
8	64	65	66	67	68	69	70	71

As seen, the rows are rearranged according to the `sampler` array:

- First row becomes the 2nd row (index 1).
- Fifth row becomes the 6th row (index 5), and so on.



# Permutation and Random Sampling

## Random Sampling without Replacement

### Introduction:

- Random Sampling without Replacement: A technique to sample elements randomly from a dataset without repeating any element in the sample.

### Create a Permutation Array

- We begin by generating a permutation array using `numpy.random.permutation()`:

### Code:

```
import numpy as np
```

```
import pandas as pd
```

```
dat = np.arange(80).reshape(10, 8)
```

```
df = pd.DataFrame(dat)
```

```
# Create a random permutation of the DataFrame indices
```

```
perm = np.random.permutation(len(df))
```

# Permutation and Random Sampling

Slice the Array to Select the Sample

Next, we slice off the first n elements from the permutation array where n is the desired sample size.

Code:

```
# Take the first 3 elements of the permutation array  
sample = perm[:3]
```

Use df.take() to Obtain the Sample

Now, we use the take() method to retrieve the rows corresponding to the sampled indices:

Code:

```
df.take(sample)
```

	0	1	2	3	4	5	6	7
9	72	73	74	75	76	77	78	79
2	16	17	18	19	20	21	22	23
0	0	1	2	3	4	5	6	7

In this example, we randomly selected 3 rows from the DataFrame, without repetition, based on the permutation of indices.

# Permutation and Random Sampling

## Random Sampling with Replacement

### Introduction:

- Random Sampling with Replacement: A method of sampling where the same element can be selected multiple times.
- Generate Random Integers for Sampling
  - We use `numpy.random.randint()` to generate random indices for sampling with replacement. These indices can repeat, allowing the same element to be selected multiple times.

### Code:

```
sack = np.array([4, 8, -2, 7, 5])  
sampler = np.random.randint(0, len(sack), size=10)  
sampler
```

- Output of Random Integers (sampler): `array([3, 3, 0, 4, 0, 0, 1, 2, 1, 4])`
- The indices in the sampler array represent the positions from the sack array that are randomly selected, with some elements being selected multiple times.

# Permutation and Random Sampling

Draw the Samples Using take()

Next, we use the take() method to draw the random samples based on the generated indices:

Code:

```
draw = sack.take(sampler)
draw
```

Output of Random Sampling with Replacement: array([ 7, 7, 4, 5, 4, 4, 8, -2, 8, 5])

- The output shows that some elements of the sack array were repeated in the drawn sample, demonstrating the concept of sampling with replacement.



# Computing Indicators/Dummy Variables

## Introduction:

- **Dummy Variables:** Often used in statistical modeling and machine learning, dummy variables are binary columns representing categorical data.
- These variables help convert categorical data into numerical format, which can be used in statistical and machine learning models.

- **Creating a DataFrame with Categorical Data**

- Let's start by creating a DataFrame containing categorical data

```
import pandas as pd
```

```
# Create a DataFrame with gender and votes columns.
```

```
df = pd.DataFrame({
```

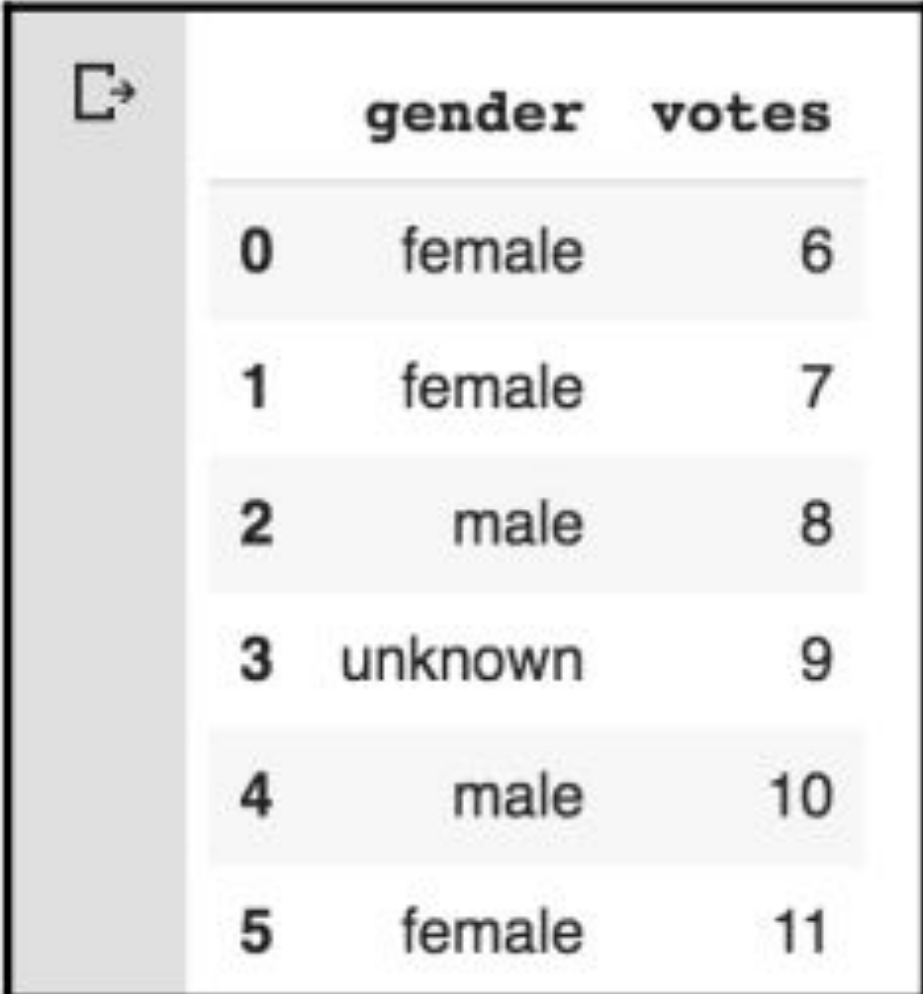
```
'gender': ['female', 'female', 'male', 'unknown', 'male', 'female'],
```

```
'votes': range(6, 12, 1)
```

```
})
```

```
df
```

In the DataFrame, gender is a categorical variable,  
and votes is numerical.



	gender	votes
0	female	6
1	female	7
2	male	8
3	unknown	9
4	male	10
5	female	11

# Computing Indicators/Dummy Variables

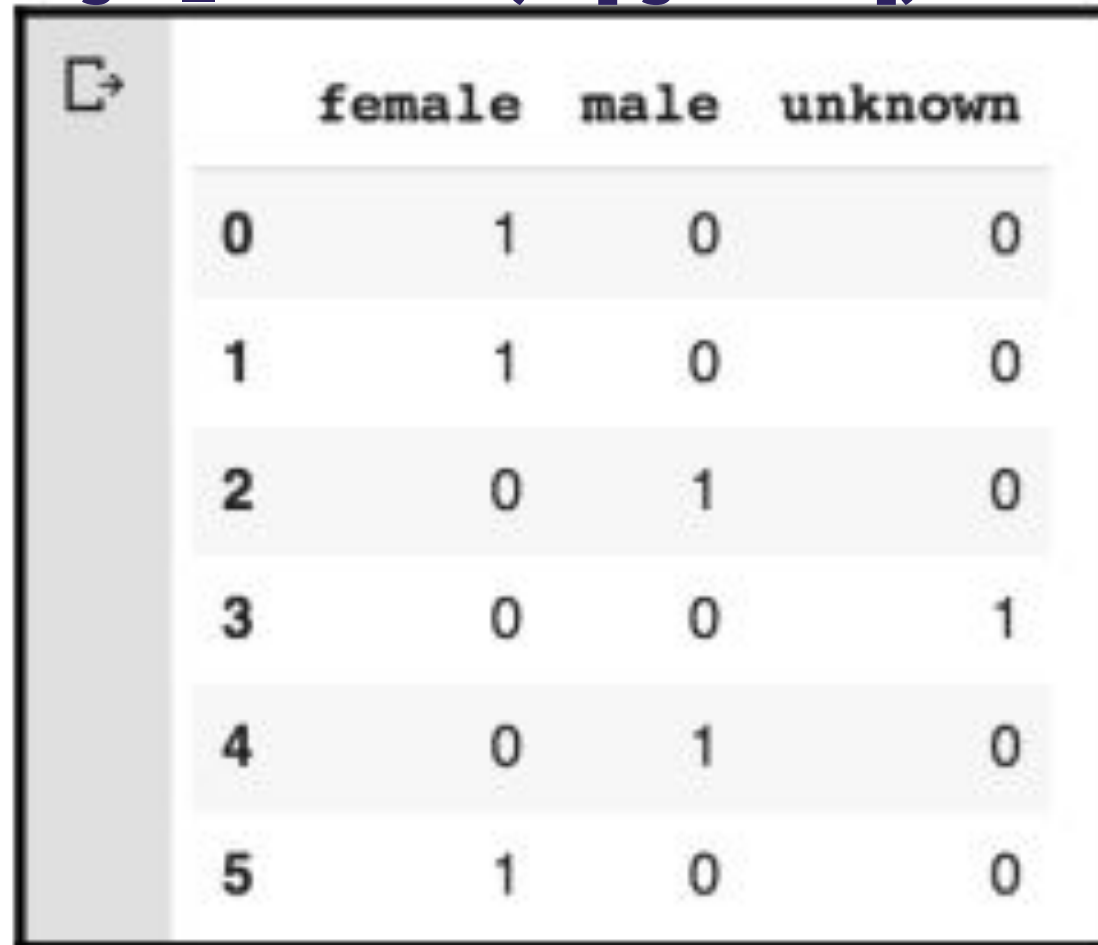
## Creating Dummy Variables

To convert the gender column into dummy variables (binary columns representing the categories), we can use the `pd.get_dummies()` function.

Code:

```
# Create dummy variables for the 'gender' column
```

```
pd.get_dummies(df['gender'])
```



	female	male	unknown
0	1	0	0
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0
5	1	0	0

The output shows a dummy matrix with 1s and 0s:

Each unique value in the gender column becomes a new column.

Each row corresponds to the original value of gender, with 1 indicating the category of the row and 0 for all others.

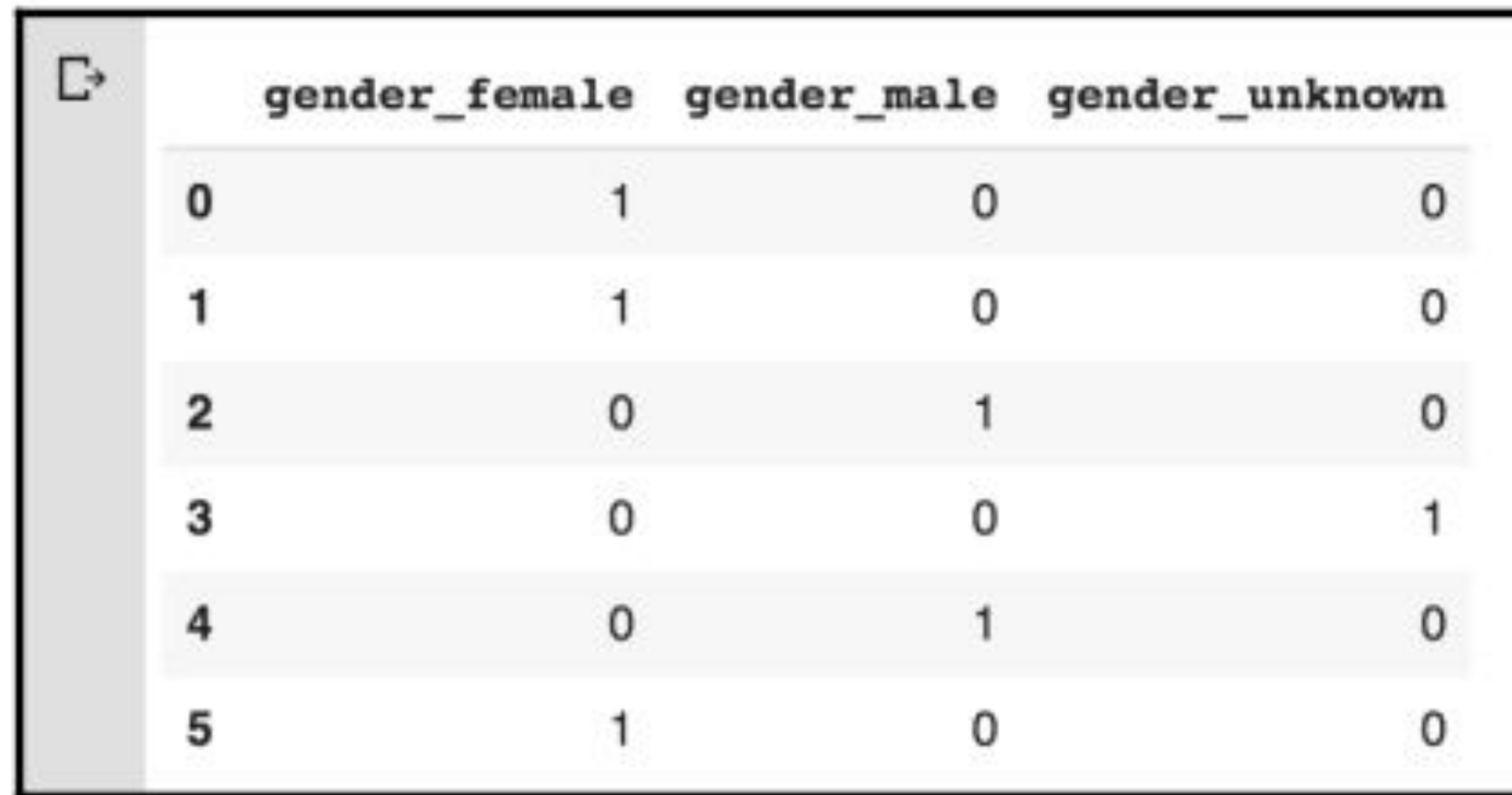
# Computing Indicators/Dummy Variables

## Adding Prefix to Column Names

Sometimes, we may want to add a prefix to the new column names for clarity. This can be done using the prefix argument in `pd.get_dummies()`.

Code:

```
# Create dummy variables with a prefix
dummies = pd.get_dummies(df['gender'], prefix='gender')
dummies
```



The image shows a Jupyter Notebook cell output. On the left is a grey sidebar with a copy icon. The main area displays a DataFrame with 6 rows (indexed 0 to 5) and 3 columns: 'gender\_female', 'gender\_male', and 'gender\_unknown'. The data is as follows:

	gender_female	gender_male	gender_unknown
0	1	0	0
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0
5	1	0	0

Each column name now includes the prefix `gender_`, making it easier to identify which categorical variable the dummy variables represent.

# String Manipulation

## Self Study Component

For a deeper dive into string manipulation, please consider this document as a valuable self-study component. You can reinforce your understanding by exploring the accompanying Jupyter Notebook, which provides examples and exercises. Additionally, refer to the prescribed textbook - Appendix for further theoretical and comprehensive coverage of the topic.

[https://github.com/ShilpaVasista/Exploratory-Data-Analytics/blob/main/String\\_Manipulation.ipynb](https://github.com/ShilpaVasista/Exploratory-Data-Analytics/blob/main/String_Manipulation.ipynb)