

|| Jai Sri Gurudev |

Sri AdichunchanagiriShikshana Trust (R)

SJB INSTITUTE OF TECHNOLOGY



Reference Note Module 2

Subject Name: Exploratory Data Analytics

Subject Code: 23CSE422

By

Faculty Name: Mrs. Shilpashree S

Designation: Assistant Professor

Semester: IV



Department of Computer Science & Engineering

Aca. Year: Even Sem /2024-25

Reference and Textbook Information

This document is designed as a supplementary study aid for the “**Exploratory Data Analytics - 23CSE422**” module, utilizing the prescribed textbook, “**Hands-On Exploratory Data Analysis with Python**” by **Suresh Kumar Mukhiya** and **Usman Ahmed**. The content presented here is based on the concepts and information provided in the textbook, with additional explanations, examples, and elaborations intended to enhance student understanding. This document is provided for educational purposes, in alignment with the intended use of the prescribed course textbook. This document is a study aid, based upon the prescribed text book, and is intended for the students of this course.

Bibliographic information

| | |
|-----------|--|
| Title | Hands-On Exploratory Data Analysis with Python: Perform EDA Techniques to Understand, Summarize, and Investigate Your Data |
| Authors | Suresh Kumar Mukhiya , Usman Ahmed |
| Publisher | Packt Publishing, Limited, 2020 |
| ISBN | 1789537258, 9781789537253 |
| Length | 352 pages |

Module 2

Data Transformation: Merging database - style dataframes

Data Transformation

Data transformation involves converting data from one format or structure to another. This process helps improve data compatibility and ensures interoperability across systems by aligning data with common structures and formats.

Key Data Transformation Techniques

- **Data Deduplication:**
 - The process of identifying and removing duplicate data entries, ensuring only unique values remain. This improves data quality and efficiency.
- **Key Restructuring:**
 - Transforming keys with specific meanings into generic keys, allowing for standardization and better compatibility across systems.
- **Data Cleansing:**
 - Involves identifying and removing outdated, inaccurate, or incomplete data, enhancing the quality of the data while preserving its essential meaning.
- **Data Validation:**
 - Establishing rules or algorithms to check the validity of data against known issues. This ensures that data adheres to defined standards and is free of errors.
- **Format Revisioning:**
 - The process of converting data from one format to another, adapting it for use in different systems or applications.
- **Data Derivation:**
 - Creating new data from existing information through predefined rules, adding value by generating additional insights.
- **Data Aggregation:**
 - Summarizing and extracting meaningful information from large datasets, often used for reporting purposes.
- **Data Integration:**
 - Merging different data types into a unified structure or schema, ensuring consistency across data sources.
- **Data Filtering:**
 - Identifying and isolating relevant information for specific users or systems, streamlining data access.
- **Data Joining:**

- Combining two or more tables based on established relationships between the data, creating a more comprehensive dataset for analysis.

Data transformation aims to improve the representation of data, making it compatible with various data structures and systems. It plays a crucial role in ensuring data can be effectively used across different platforms and applications.

Merging Database-Style DataFrames in Pandas

1. Concatenation

Concatenating DataFrames (Axis=0)

When you want to stack DataFrames vertically (i.e., adding more rows), you can use the `concat()` method. Consider the case of merging two DataFrames containing student IDs and scores for the Software Engineering course.

- `dataframe = pd.concat([dataFrame1, dataFrame2], ignore_index=True)`
- **ignore_index=True**: This argument creates a new index for the resulting DataFrame. If it's not specified, the original indices are kept.
- **Axis=0**: The default axis for concatenation is 0, meaning the DataFrames are combined row-wise.

Concatenating DataFrames (Axis=1)

If you want to concatenate DataFrames side-by-side, you can specify `axis=1`.

- `pd.concat([dataFrame1, dataFrame2], axis=1)`

This operation adds columns to the existing DataFrame, not rows.

2. Merging DataFrames

In situations where you need to combine DataFrames based on common columns or indices, `merge()` comes into play. This is particularly useful for joining data from different tables.

Example Scenario

You are teaching two courses, Software Engineering (SE) and Machine Learning (ML), and you have received two DataFrames for each course, each containing student IDs and scores. You need to merge this information to get a comprehensive overview of all students.

Concatenating DataFrames First

Before merging, you can concatenate the DataFrames for each course. For example, if you have two DataFrames for the SE course:

- `dfSE = pd.concat([df1SE, df2SE], ignore_index=True)`
- `dfML = pd.concat([df1ML, df2ML], ignore_index=True)`

Now that each subject's DataFrame has been concatenated, you can merge the DataFrames using different techniques:

Option 1: Concatenating with Axis=1

To combine both courses into a side-by-side DataFrame:

- `df = pd.concat([dfML, dfSE], axis=1)`
- This operation results in two columns from both the SE and ML DataFrames being placed side by side. However, you may notice a repeated StudentID column.

Option 2: Merging with Inner Join

- `df = dfSE.merge(dfML, how='inner')`

- **Inner Join:** This operation combines the rows that have matching StudentID values in both DataFrames. Only students who appear in both courses are included.

Option 3: Merging with Left Join

- `df = dfSE.merge(dfML, how='left')`
- **Left Join:** This operation includes all students from the SE DataFrame. If a student did not appear in the ML course, the corresponding columns will contain NaN values.

Option 4: Merging with Right Join

- `df = dfSE.merge(dfML, how='right')`
- **Right Join:** This operation includes all students from the ML DataFrame. If a student did not appear in the SE course, the corresponding columns will contain NaN values.

Option 5: Merging with Outer Join

- `df = dfSE.merge(dfML, how='outer')`
- **Outer Join:** This operation returns all students from both DataFrames. If a student appears in one course but not the other, the missing data will be represented as NaN.

3. Merging on Index

Sometimes the merge key is located in the DataFrame index. In such cases, the `left_index=True` or `right_index=True` argument can be used.

Example Scenario: Merging on Index

- `df = pd.merge(left1, right1, left_on='key', right_index=True)`

- In this example, the keys (e.g., 'apple', 'ball', 'cat') are in the left1 DataFrame, while the values are in the right1 DataFrame indexed by 'apple' and 'ball'. The merge operation finds matching keys and combines the DataFrames based on these values.

Outer Join Example on Index

- `df = pd.merge(left1, right1, left_on='key', right_index=True, how='outer')`
- This operation includes all keys from both DataFrames, filling in missing values with NaN.

Summary of Merge Types

1. **Inner Join:** Includes only the intersection of both DataFrames.
2. **Outer Join:** Includes all rows from both DataFrames, with missing values filled as NaN.
3. **Left Join:** Includes all rows from the left DataFrame, with NaN for missing values from the right DataFrame.
4. **Right Join:** Includes all rows from the right DataFrame, with NaN for missing values from the left DataFrame.

Reshaping and Pivoting

Stacking and Unstacking

Data transformation involves reshaping data into a consistent and analyzable format. Stacking and unstacking are important operations for reorganizing data.

1. **Stacking:**
 - The `stack()` function in pandas is used to **pivot** columns into rows, converting the DataFrame into a **Series**.

- This helps in rearranging the DataFrame when we need to perform hierarchical indexing.
- The rows of the DataFrame are **stacked vertically** into a multi-level index Series.

2. Unstacking:

- The `unstack()` function is the reverse operation of `stack()`, which rotates rows back into columns.
- It takes a **Series** and pivots it back into a DataFrame.
- Unstacking may lead to **missing data (NaN)** if the sub-groups in the multi-level index do not match perfectly between rows and columns.

Example:

1. Create a DataFrame for Rainfall, Humidity, and Wind Conditions in Various Norwegian Counties:

```
import numpy as np
import pandas as pd

data = np.arange(15).reshape((3, 5)) # Create 3x5 array of data
indexers = ['Rainfall', 'Humidity', 'Wind'] # Index labels
dframe1 = pd.DataFrame(data, index=indexers, columns=['Bergen',
'Oslo', 'Trondheim', 'Stavanger', 'Kristiansand'])

# Output of the DataFrame:
print(dframe1)
```

Output:

| | Bergen | Oslo | Trondheim | Stavanger | Kristiansand |
|----------|--------|------|-----------|-----------|--------------|
| Rainfall | 0 | 1 | 2 | 3 | 4 |
| Humidity | 5 | 6 | 7 | 8 | 9 |
| Wind | 10 | 11 | 12 | 13 | 14 |

2. Stacking the DataFrame:

```
stacked = dframe1.stack()  
print(stacked)
```

Output:

| | Berge n | Osl o | Trondhei m | Stavang er | Kristiansan d |
|--------------|--------------------|------------------|-----------------------|-----------------------|--------------------------|
| Rainfall | 0 | 1 | 2 | 3 | 4 |
| Humidit y | 5 | 6 | 7 | 8 | 9 |
| Wind | 10 | 11 | 12 | 13 | 14 |

This stack operation pivots the columns into rows, creating a multi-level index (first by the original rows and then by columns). The data now represents a series with a hierarchical index.

3. Unstacking the Stacked Series:

```
unstacked = stacked.unstack()  
print(unstacked)
```

Output:

| | Berge n | Osl o | Trondhei m | Stavang er | Kristiansan d |
|--------------|--------------------|------------------|-----------------------|-----------------------|--------------------------|
| Rainfall | 0 | 1 | 2 | 3 | 4 |
| Humidit y | 5 | 6 | 7 | 8 | 9 |
| Wind | 10 | 11 | 12 | 13 | 14 |

The unstacking function reverts the series back into the original DataFrame, with the hierarchical index structure turned back into the columns.

4. Handling Missing Data during Unstacking:

```
series1 = pd.Series([000, 111, 222, 333], index=['zeros',
'ones', 'twos', 'threes'])
series2 = pd.Series([444, 555, 666], index=['fours', 'fives',
'sixes'])
frame2 = pd.concat([series1, series2], keys=['Number1',
'Number2'])

unstacked_frame = frame2.unstack()
print(unstacked_frame)
```

Output:

| | Number 1 | Number 2 |
|------------|-------------|-------------|
| zero s | 000 | NaN |
| ones | 111 | NaN |
| twos | 222 | NaN |
| three s | 333 | NaN |
| fours | NaN | 444 |
| fives | NaN | 555 |
| sixes | NaN | 666 |

Explanation:

- Unstacking creates **NaN** values where the indexes do not match between the series, as seen in the rows for **fours**, **fives**, and **sixes** which were only present in **series2**.
- Similarly, **zeros**, **ones**, **twos**, and **threes** exist only in **series1**, leading to **NaN** values for the rows corresponding to **Number2**.
- _____

Data Deduplication:

The process of identifying and removing duplicate records in a dataset to enhance data quality. It helps in ensuring that analysis is performed on unique data points, preventing skewed results.

Methods for Data Deduplication:

1. **duplicate()** Method:

- This method is used to identify duplicate rows in a DataFrame.
- It returns a Boolean Series, where **True** indicates a duplicate row, and **False** means the row is unique.

Example:

```
frame3 = pd.DataFrame({'column 1': ['Looping'] * 3 +  
['Functions'] * 4, 'column 2': [10, 10, 22, 23, 23, 24, 24]})  
frame3.duplicated()
```

2. Output: A Boolean series showing which rows are duplicates.

3. **drop_duplicates()** Method:

- This method removes duplicate rows from a DataFrame based on all columns or a specified subset of columns.
- By default, it keeps the first occurrence and removes subsequent duplicates.
- **take_last=True**: Keeps the last occurrence instead of the first.

Example:

```
frame4 = frame3.drop_duplicates()  
frame4
```

4. Output: The DataFrame with duplicates removed.

5. **Detecting Duplicates in Specific Columns:**

- By passing a subset of columns to the **drop_duplicates()** method, we can check and remove duplicates based on specific columns, not the entire DataFrame.

Example:

```
frame5 = frame3.drop_duplicates(['column 2'])  
frame5
```

6. Output: The DataFrame with duplicates based on **column 2** removed.

Replacing Values

In data analysis, it is common to find and replace specific values in a DataFrame to clean or update the data. This helps in handling missing data or correcting erroneous values.

replace() Method: The `replace()` method in pandas is used to replace specific values with new ones in a DataFrame.

Steps to Replace Values:

1. Single Value Replacement:

- The `replace()` method allows replacing a single value in the DataFrame with another value.

Example:

```
import numpy as np
replaceFrame = pd.DataFrame({
    'column 1': [200., 3000., -786., 3000., 234., 444., -786.,
332., 3332.],
    'column 2': range(9)
})
replaceFrame.replace(to_replace=-786, value=np.nan)
```

- In this example, all instances of `-786` in the DataFrame are replaced with `NaN`.

2. Multiple Value Replacements:

- Multiple values can be replaced at once using a list for both `to_replace` and `value`.

Example:

```
replaceFrame = pd.DataFrame({
    'column 1': [200., 3000., -786., 3000., 234., 444., -786.,
332., 3332.],
    'column 2': range(9)
})
replaceFrame.replace(to_replace=[-786, 0], value=[np.nan, 2])
```

- In this case:
 - All occurrences of `-786` are replaced by `NaN`.
 - All occurrences of `0` are replaced by `2`.
-

Handling Missing Data

- **NaN (Not a Number):**
 - Indicates a missing or unspecified value in a dataset.
 - Reasons for NaN values:
 - Data retrieval from an external source with incomplete data.
 - Mismatched data when joining datasets.
 - Errors during data collection.
 - Shape changes in data (e.g., adding rows or columns).
 - Reindexing resulting in incomplete data.
-

Example: Initial DataFrame Creation

```
data = np.arange(15, 30).reshape(5, 3)
dfx = pd.DataFrame(data, index=['apple', 'banana', 'kiwi',
'grapes', 'mango'], columns=['store1', 'store2', 'store3'])
dfx
```

- This DataFrame does not have missing values initially.
- Represents sales data for different fruits across different stores.

Adding Missing Data to DataFrame

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

- New stores (`store4` and `store5`) and fruits (`watermelon` and `oranges`) are added.
- `store4` has some data (e.g., apples and watermelon sales) while `store5` has only NaN values, and `oranges` has NaN values in all columns.

Characteristics of Missing Values

- NaN values can appear in:
 - Entire rows.
 - Entire columns.
 - Some values in specific rows or columns.

Identifying Missing Data

isnull() Method: Checks for NaN values.

```
dfx.isnull()
```

notnull() Method: Checks for non-NaN values.

```
dfx.notnull()
```

- **sum() Method:** Counts the number of NaN values per column.

```
dfx.isnull().sum()
```
- Shows how many NaN values exist in each store.

count() Method: Counts non-NaN values per column.

```
dfx.count()
```

Dropping Missing Data

- **dropna() Method:** Removes rows with NaN values.

Drops rows with NaN values:

```
dfx.dropna()
```

- If applied to the entire DataFrame, it removes any rows with at least one NaN value.

dropna() with how='all': Drops rows where all values are NaN.

```
dfx.dropna(how='all')
```

- **Dropping Columns with NaN:**
 - Use **axis=1** to drop columns with NaN values.

```
dfx.dropna(how='all', axis=1)
```

- **Using thresh Parameter:** Drops columns if they contain more than a specified number of NaN values.

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

Mathematical operations with NaN

- **NaN in NumPy:** When performing mathematical operations, NumPy functions return **NaN** if the data contains **NaN** values.

Example:

```
ar1 = np.array([100, 200, np.nan, 300])  
ar1.mean() # Result: nan
```

Since **NaN** values are present, the mean is computed as **NaN**.

Handling NaN in Pandas

- **NaN in Pandas:** Pandas functions handle **NaN** values differently. It ignores **NaN** values during computations like sum, mean, and cumulative operations, treating them as 0.

Example:

```
ser1 = pd.Series([100, 200, np.nan, 300])  
ser1.mean() # Result: 200.0
```

Pandas computes the mean of the non-NaN values, resulting in 200.0.

Mathematical Operations with NaN Values in Pandas

1. Sum Operation:

NaN values are treated as 0 during summing in Pandas.

```
ser2 = pd.Series([np.nan, 200, np.nan, 100])
```

```
ser2.sum() # Result: 300.0
```

If all values are NaN, the result is NaN.

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

```
ser3.sum() # Result: nan
```

2. Mean Calculation:

Pandas ignores NaN values and computes the mean of non-NaN values.

```
ser2.mean() # Result: 19.0 (NaNs are treated as 0s)
```

3. Cumulative Sum:

In cumulative summing, only the non-NaN values are used in the calculation. NaN values are treated as 0s in this context.

```
ser2.cumsum()
```

```
# Output:
```

```
# apple      20.0
```

```
# banana     NaN
```

```
# kiwi       NaN
```

```
# grapes     NaN
```

```
# mango      NaN
```

```
# watermelon 38.0
```

```
# oranges    NaN
```

```
# Name: store4, dtype: float64
```

The cumulative sum includes only the actual values (NaNs are treated as 0s).

Key Differences Between NumPy and Pandas Handling of NaN

- NumPy:
 - When NaN values are present, most mathematical functions (e.g., `mean()`, `sum()`) return NaN.
 - Pandas:
 - Ignores NaN values during operations and treats them as 0 for summing and other aggregate functions like `mean()`, `cumsum()`.
-

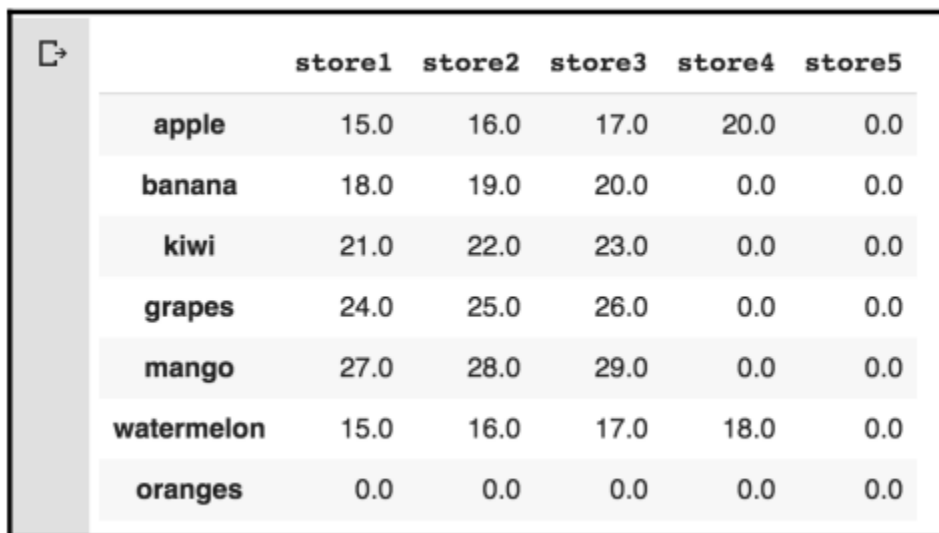
Filling Missing Values

`fillna()` Method: Used to replace NaN values in a DataFrame with specific values.

Example:

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

Output:



| | 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 |

Impact of Replacing NaN with 0:

- Alters statistical metrics such as mean, sum, and median.

Example:

Original DataFrame mean calculation:

```
dfx.mean()
```

Output:

```
store1    20.0
store2    21.0
store3    22.0
store4    19.0
```

```
store5    NaN  
dtype: float64
```

Filled DataFrame mean calculation:

```
filledDf.mean()
```

Output:

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

Replacing NaN with 0 is not always the optimal solution as it can distort statistical analysis.

Backward and Forward Filling

- **Handling NaN Values with Filling Techniques:**
 - NaN values in a DataFrame can be replaced using the last known value or next known value.
- **Forward Filling (`method='ffill'`):**
 - This technique fills NaN values with the last known value from the previous row.

Example:

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

Output:

```
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
```

The value 20.0 is propagated forward until a new value (18.0) is encountered.

- **Backward Filling (`method='bfill'`):**
 - This technique fills NaN values with the next known value from the following row.

Example:

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

Output:

```
apple      20.0  
banana     18.0  
kiwi       18.0  
grapes     18.0  
mango      18.0  
watermelon 18.0  
oranges    NaN  
Name: store4, dtype: float64
```

The value 18.0 is propagated backward until a NaN value (oranges) remains at the end.

Observations:

- The choice between forward and backward filling depends on the dataset's context and the nature of the analysis.
 - Forward filling propagates previous values, while backward filling substitutes upcoming ones.
-

Interpolating Missing Values

Interpolation in Pandas:

- Pandas provides the `interpolate()` function for both `Series` and `DataFrame` objects.

- By default, `interpolate()` performs linear interpolation to fill missing values.

Linear Interpolation:

- Linear interpolation estimates missing values by finding the difference between the first known value before and the first known value after the NaN sequence, dividing this difference evenly.

Example:

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

Output:

```
0    100.0
1    148.0
2    196.0
3    244.0
4    292.0
```

dtype: float64

Calculation Explanation:

- Known values: 100 (index 0) and 292 (index 4).
- Interpolation step:
 $(292 - 100) / (5 - 1) = 48$
 $(292 - 100) / (5 - 1) = 48$
- Missing values are filled as:
 - Index 1: $100 + 48 = 148$
 - Index 2: $148 + 48 = 196$
 - Index 3: $196 + 48 = 244$

Advanced Interpolation:

- The `interpolate()` method supports complex interpolation techniques, especially useful in time series data.

Renaming Axis Indexes in Pandas

Renaming Indexes:

- The `index.map()` function allows transformation of index terms, applying a specific function to each element of the index.

Example:

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



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

- Converts all index terms in `dframe1` to uppercase.
- Modifies the original DataFrame.

Using the `rename()` Method:

- The `rename()` method can transform index or column names without altering the original DataFrame.

Syntax:

```
dframe1.rename(index=str.title, columns=str.upper)
```



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

- `index=str.title`: Converts index terms to title case.
- `columns=str.upper`: Converts column names to uppercase.
- **Key Note:** The `rename()` method does not create a copy of the DataFrame but returns a transformed version.

Discretization and Binning

Introduction

Discretization is the process of converting continuous data into discrete intervals

or categories. This is commonly done in data analysis to simplify complex datasets, making them easier to interpret or analyze. The technique is often referred to as binning, where each interval is called a bin.

Key Concepts

1. Discretization

This process involves dividing continuous values into intervals. Each interval represents a bin, where data points fall into one of the bins based on their value.

2. Binning

Binning is the main method for discretization. The binning process splits data into intervals, and each interval can either be closed or open on the boundaries.

3. Interval Representation

- An open interval means the boundary value is not included in the interval.
- A closed interval includes both boundary values. For example:
- (118, 125] means 118 is not included, but 125 is included.
- [118, 125) means 118 is included, but 125 is not.

4. Using `cut()` Method in Pandas

Pandas provides the `cut()` method to bin continuous data. The function divides data into specified intervals and assigns them to respective bins. Example:

```
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)
```

The output will categorize the height data into the bins as defined.

5. Customizing Interval Boundaries with `right=False`

The argument `right=False` can be passed to change the default behavior of the interval boundaries. It makes the right side of the interval open, meaning it does not include the upper limit.

Example:

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

6. Counting Values in Bins

The `value_counts()` method can be used to count how many values fall into each bin.

Example:

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

This will display how many values fall into each interval.

7. Labeling Bins

Bins can be labeled for easier interpretation. A list of labels can be passed to the `cut()` method to assign names to the bins.

Example:

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

8. Equal-Length Bins

If only the number of bins is specified, Pandas will automatically compute equal-length bins. For example:

python

CopyEdit

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

9. Quantiles and `qcut()` Method

Quantiles divide the data into intervals where each interval has approximately the same number of data points. The `qcut()` method in Pandas is used to create quantile-based bins.

Example:

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

10. Custom Quantiles in `qcut()`

Custom quantiles can be defined by passing a list of quantiles.

Example:

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

Outlier Detection and Filtering

1. Definition of Outliers:

Outliers are data points that significantly deviate from the other observations in a dataset. These deviations may arise due to errors, rare events, or variability in the data.

2. Importance of Outlier Detection:

- Outliers can skew statistical results and model performance.
- Removing or addressing outliers improves the reliability of analysis and insights.

3. Steps in Outlier Detection and Filtering:

Step 1: Load the Dataset:

Use a dataset to analyze and detect outliers. Example code snippet:

```
df =  
pd.read_csv('https://raw.githubusercontent.com/PacktPublishing/h  
ands-on-exploratory-data-analysis-with-python/master/Chapter%204  
/sales.csv')  
df.head(10)
```

- Dataset: Contains sales data with columns such as `Quantity`, `UnitPrice`, and others.
- Preview of data (`df.head(10)`): Displays the first 10 rows for an overview.

Step 2: Calculate a Derived Metric (TotalPrice):

Create a new column by multiplying `Quantity` and `UnitPrice`.

```
df['TotalPrice'] = df['UnitPrice'] * df['Quantity']  
d
```


- Adds a new column **TotalPrice** to the dataset.

Step 3: Detect Outliers Using Conditions:

Example: Identify transactions with **TotalPrice** exceeding a threshold, such as 3,000,000.

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

- Result: A subset of rows with **TotalPrice** greater than 3,000,000.

Step 4: Filter Outliers Based on Custom Thresholds:

Example: Display rows where **TotalPrice** exceeds 6,741,112.

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

- Outputs only rows that meet the condition.

4. Techniques for Outlier Detection:

- Statistical thresholds (e.g., values beyond a fixed range).
- Visual methods (e.g., box plots or scatter plots).
- Domain-specific knowledge for setting thresholds.

Examples from the Dataset:

Rows where **TotalPrice > 3,000,000**:

```
2 3711433  
7 3965328  
13 4758900  
...
```

- Rows where **TotalPrice > 6,741,112**:
Only rows meeting this condition are displayed.
-

Permutation and Random Sampling

1. Introduction

- **Permutation:** Refers to rearranging the elements of a series or DataFrame in a random order.
 - **Random Sampling:** Refers to randomly selecting rows from a DataFrame for analysis or experimentation.
-

2. Performing Permutation and Random Sampling using Pandas

Step 1: Create a DataFrame

Using NumPy, create a dataset and convert it to a Pandas DataFrame:

```
import numpy as np
import pandas as pd

dat = np.arange(80).reshape(10, 8)
df = pd.DataFrame(dat)
print(df)
```

Output:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 8 | 9 | 1 | 11 | 1 | 1 | 1 | 1 |
| | | | 0 | | 2 | 3 | 4 | 5 |
| 2 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 |
| .. | ... | ... | ... | ... | ... | ... | ... | ... |
| . | | | | | | | | |

Step 2: Permute Rows Using `np.random.permutation()`

Generate a random order for the rows:

```
sampler = np.random.permutation(10)
print(sampler)
```

Output:

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

Step 3: Apply the Permutation to the DataFrame

Use the `take()` function in Pandas with the generated sampler:

```
df_permuted = df.take(sampler)
print(df_permuted)
```

This rearranges the rows based on the order specified in the `sampler`.

Random Sampling Without Replacement

- **Definition:** Random sampling without replacement means selecting a subset of data randomly, ensuring no data point is chosen more than once.

Steps to Perform Random Sampling Without Replacement

1. **Create a Permutation Array:**
Use the `numpy.random.permutation()` method to shuffle the dataset.
2. **Slice the Array:**
After shuffling, slice the first `n` elements where `n` is the size of the sample you wish to draw.
3. **Use `df.take()` Method:**
Apply the sliced array to select the desired random sample from the DataFrame.

Example:

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

```
# Sample DataFrame
dat = np.arange(80).reshape(10, 8)
df = pd.DataFrame(dat)

# Random Sampling without Replacement
sample = df.take(np.random.permutation(len(df))[:3])
print(sample)
```

Output: A random selection of 3 rows from the DataFrame.

2. Random Sampling With Replacement

- **Definition:** Random sampling with replacement means selecting a subset of data randomly, where the same data point can be selected more than once.

Steps to Perform Random Sampling With Replacement

1. **Generate Random Integers:**
Use `numpy.random.randint()` to generate random integers to serve as indices for sampling.
2. **Draw the Required Samples:**
Use the generated integers to sample from the dataset, allowing repeated selections of the same index.

Example:

```
import numpy as np

# Sample Data
sack = np.array([4, 8, -2, 7, 5])

# Generate Random Sample with Replacement
sampler = np.random.randint(0, len(sack), size=10)
print(sampler)

# Draw Samples from sack
draw = sack.take(sampler)
```

```
print(draw)
```

Output:

```
# Sampled indices:  
array([3, 3, 0, 4, 0, 0, 1, 2, 1, 4])  
  
# Drawn samples:  
array([ 7, 7, 4, 5, 4, 4, 8, -2, 8, 5])
```

Explanation:

- The indices in `sampler` are selected randomly, and the `take()` method is used to sample values from `sack`. Repeated values indicate that some elements were chosen more than once due to the sampling with replacement.
-

Computing Indicators/Dummy Variables

Definition:

- Dummy variables (or indicator variables) are binary (0 or 1) variables used to represent categories or groups in statistical modeling or machine learning.
- They are useful when you need to convert a categorical variable into a form that can be used in statistical models, which typically require numerical input.

Process of Creating Dummy Variables:

1. Creating a DataFrame:
 - Consider a DataFrame with categorical data (e.g., gender) and numeric data (e.g., votes).

Example:

```
import pandas as pd
```

```
# Sample DataFrame
df = pd.DataFrame({
    'gender': ['female', 'female', 'male', 'unknown', 'male',
'female'],
    'votes': range(6, 12)
})
print(df)
```

Output:

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

2.

3. Converting Categorical Data to Dummy Variables:

- To encode the categorical column ('gender' in this case) into dummy variables, use `pd.get_dummies()`.

Example:

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

Output:

| | 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 |

4.

- Each unique value (female, male, unknown) is turned into a column.
- The rows are filled with 1s and 0s indicating whether the original value matches the column header.

5. Adding a Prefix to Column Names:

- To add a prefix to the columns, use the `prefix` argument in `pd.get_dummies()`.

Example:

```
dummies = pd.get_dummies(df['gender'], prefix='gender')  
print(dummies)
```

Output:

| | 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 |

6.

- This method ensures that the column names have the prefix (`gender_`) followed by the category name.

String Manipulation

Creating a String

Strings in Python can be created using single quotes (`'`), double quotes (`"`), or even triple quotes (`' '` or `"""`).

```
# Creating a string with single, double, and triple quotes  
String1 = 'Creating a String with single Quotes.'  
String2 = "Creating a String with double Quotes."  
String3 = '''Creating a String with triple Quotes.'''  
  
print(String1)  
print(String2)  
print(String3)
```

2. Accessing Characters in a String

You can access individual characters using indexing. Positive indexing starts from **0**, and negative indexing starts from **-1** for the last character.

```
String = "Exploratory Data Analysis"

# First character
print("\nFirst character of String is:", String[0])

# Last character
print("\nLast character of String is:", String[-1])
```

3. String Slicing

Slicing allows you to extract a portion of the string.

```
String = "Exploratory Data Analysis"

# Slicing from 3rd to 12th character
print("\nSlicing characters from 3-12:", String[3:12])

# Slicing from 3rd to 2nd last character
print("\nSlicing characters from 3rd to 2nd last character:",
String[3:-2])
```

4. Deleting/Updating a String

Strings in Python are immutable, meaning individual characters cannot be updated or deleted. You need to update the entire string.

```
# Updating an entire string
String1 = "Exploratory Data Analysis"
print("\nInitial String:", String1)

String2 = "Welcome to the world of Data Science."
print("\nUpdated String:", String2)

# Deleting an entire string (removes the reference)
```



```
String = "Exploratory Data Analysis"
del String
# This will raise a NameError since the string is deleted
```

5. Escape Sequencing

Escape sequences allow special characters to be used in strings, such as newlines, quotes, or backslashes.

```
String = '''I'm a "Data Scientist"'''
print("Initial String with Triple Quotes:", String)
```

```
# Escape single and double quotes
String = 'I\'m a "Data Scientist"'
print("\nEscaping Single Quote:", String)
```

```
String = "I'm a \"Data Scientist\""
print("\nEscaping Double Quotes:", String)
```

```
# Escape backslashes
String = "C:\\Python\\DataScience\\"
print("\nEscaping Backslashes:", String)
```

6. Formatting of Strings

Python provides several ways to format strings, such as positional formatting, keyword formatting, and `format()`.

```
String1 = "{} {} {}".format('Exploratory ', 'Data ', 'Analysis')
print("Print String in default order:", String1)
```

```
String1 = "{1} {0} {2}".format('Exploratory', 'Data',
'Analysis')
print("\nPrint String in Positional order:", String1)
```

```
String1 = "{l} {f} {g}".format(g='Exploratory', f='Data',  
l='Analysis')  
print("\nPrint String in order of Keywords:", String1)
```

7. Working with a Text Dataset

You can load a dataset using pandas and apply string operations like converting to lowercase, slicing, or splitting.

```
import pandas as pd  
  
# Load dataset  
text = pd.read_csv("comments.csv")  
text = text["body"] # Convert DataFrame to Series  
  
# Convert to lowercase  
text.str.lower().head(8)  
  
# Get the length of all comments  
text.str.len().head(8)  
  
# Split comments on spaces  
text.str.split(" ").head(8)  
  
# Strip leading and trailing brackets  
text.str.strip("[]").head(8)  
  
# Combine strings from Series into one  
text.str.cat()[0:500]  
  
# Slice first 10 characters  
text.str.slice(0, 10).head(8)
```

8. Regular Expressions

Regular expressions (regex) are powerful for string pattern matching and searching.

```
# Match any substring ending in 'abb'
my_words = pd.Series(["abaa", "cabb", "Abaa", "sabb", "dcbb"])
my_words.str.contains(".abb")

# Match 'T' or 't' followed by 'ill'
my_words.str.contains("[Aa]bb")

# Match 'He' or 'he' at the beginning of a string
Sentence_series = pd.Series(["Where did he go", "He went to the
shop", "he is good"])
Sentence_series.str.contains("^(He|he)")

# Match 'go' at the end of a string
Sentence_series.str.contains("(go)$")

# Count occurrences of 'Wolves'
text.str.count(r"[Ww]olves").head(8)

# Find all occurrences of 'Wolves'
text.str.findall(r"[Ww]olves").head(8)
```

9. Getting Posts with Web Links

You can extract posts containing links using regex.

```
# Extract posts with web links
web_links = text.str.contains(r"https?:")
posts_with_links = text[web_links]
print(len(posts_with_links))
posts_with_links.head(5)

# Extract the URLs from the posts
```

```
only_links = posts_with_links.str.findall(r"https?:[^\n\\]")  
only_links.head(10)
```

These string manipulation techniques are essential for data processing, particularly when working with textual data.