PW SKILLS

Lesson Plan

Pandas

Introduction to Pandas

Python programming language's Pandas library is an open-source tool for handling and analyzing data. In order to make dealing with organized and tabular data simple and effective, it offers data structures and functions. With the addition of named axes (rows and columns) and more versatile data processing tools, Pandas, which is built on top of the NumPy library, expands its functionality. In data science and data analysis, it is a crucial tool.

Pandas Series

- A Pandas Series is a one-dimensional labeled array that can hold data of various types, such as integers, floats, strings, or even custom objects.

- It is akin to a column in a spreadsheet or a single variable in statistics. Each element in a Series is assigned an index, allowing for easy data retrieval and manipulation.

Creating Series:
To Create Pandas Series DataFrame we use “pd.Series()” by converting other data structures like List[] or NumPy arrays, Into Series.

Accessing Data or Information in Series:
- Once the Series DataFrame is available we access the Values of the df using “.values” or we can also access it with the help of the Indexing Method. Using “.index”.
- Pandas support Indexing and Slicing Methods. We can use integer-based positions or label-based indexing.

Data Alignment:
- One of the powerful features of Pandas Series is data alignment.
- When performing operations on multiple Series, Pandas aligns data based on index labels, making it easy to work with incomplete or differently indexed data.

What is a DataFrame?

- A DataFrame is akin to a spreadsheet or SQL table, with rows and columns that can hold a variety of data types.

- Each column in a DataFrame is essentially a Pandas Series, sharing a common index, making it easier to manage and manipulate data efficiently.

- There are different methods to generate a Dataframe. But commonly used approaches are:
  - Creating a Dict
  - Reading Custom External Data

Accessing Data:
There are different Operations we can perform in a Dataframe. To access and manipulate data some common operations are:

1. Accessing a specific column.
2. Accessing a specific row.
3. Slicing rows and columns

Data Manipulations:

- Pandas provides numerous functions to manipulate data within DataFrames, such as filtering, sorting, merging, and aggregating data.
- Some Common operations include:
  - Filtering
  - Sorting
  - Merging DataFrames

Data Structures in Pandas

Pandas primarily offers two fundamental data structures Series and DataFrame.

- **Series:** A Series is a one-dimensional array-like object that can hold various data types, including integers, floats, and strings. It is essentially a labeled array and has an associated index. You can think of a Series as a column in a spreadsheet.

- **DataFrame:** A DataFrame is a two-dimensional table, similar to a spreadsheet or a SQL table. It consists of rows and columns, and each column is a Series. DataFrames are the primary data structure for most Pandas operations.

Basic DataFrame Operation:

- Accessing Data: You can access specific columns and rows using column labels and row indices, respectively.
- Filtering Data: Filtering allows you to extract specific rows based on conditions.
- Adding and Modifying Data: You can add new columns and modify existing ones using pandas.

Summary Statistics:

Pandas provides functions to calculate summary statistics:
- Mean
- Median
- Mode

Handling Missing Data: Pandas also offers robust tools for handling missing data, which is a common issue in real-world datasets. You can use methods like “isna()”, “fillna()”, and “dropna()” to manage missing values effectively.

Reading and Writing Data:

- Pandas can read data from various file formats, such as CSV, Excel, and SQL databases, using functions like “read_csv()”, “read_excel()”, and “read_sql()”.
- It can also write DataFrames to these formats using functions like to_csv() and to_excel().

Reading data from various file system

Reading Data from CSV Files:

- CSV (Comma-Separated Values) files are one of the most common data formats.

- Pandas makes it easy to read data from CSV files using the read_csv() function.

- You can specify various options such as delimiter, encoding, and header rows to ensure the data is imported correctly.

Reading Data from Excel Files:

- Pandas can read data from Excel files, both in the older .xls format and the newer .xlsx format.
- You can use the read_excel() function to extract data from specific sheets and set custom data ranges.

Reading Data from JSON Files:

- JSON (JavaScript Object Notation) is a widely used format for data exchange.
- With Pandas, you can read JSON files using the read_json() function.
- This function can also handle nested JSON structures and convert them into DataFrames.

Reading Data from SQL Databases:

- Pandas provides built-in support for reading data from SQL databases.
- You can use the read_sql() function to retrieve data from a wide range of SQL databases, including PostgreSQL, MySQL, SQLite, and more.
- This enables you to work with structured data stored in databases directly.

Reading Data from Parquet Files:

- Parquet is a columnar storage format often used in big data environments.
- Pandas can read data from Parquet files using the read_parquet() function, which is particularly useful when dealing with large datasets efficiently.

Reading Data from HTML Tables:

- Sometimes, you may need to scrape data from web pages. Pandas allows you to read HTML tables from web pages with the read_html() function.
- This feature is especially handy for web scraping and data extraction.

Reading Data from HDF5 Files:

- HDF5 is a versatile data format used in scientific computing and data storage.

- Pandas supports reading data from HDF5 files through the read_hdf() function, making it accessible for scientific data analysis.

Re-Indexing in Pandas

- Reindexing is a fundamental operation in the Python Pandas library, allowing data scientists and analysts to reshape, realign, and modify data structures like Series and DataFrames.

- This process helps in ensuring data consistency and compatibility, which is crucial in various data manipulation and analysis tasks.

What is Re-indexing?

- Reindexing refers to the process of creating a new object with the data conformed to a new index.
- In Pandas, we can reindex Series and DataFrames, enabling you to manipulate data, add missing values, or rearrange data according to a different set of labels.

Re-indexing Syntax: To reindex a Pandas Series or DataFrame, you can use the reindex method. The syntax for reindexing is as follows:

```
new_data = data.reindex(new_index)
```

Re-indexing can be used in:

1. **Pandas Series Re-indexing:**
   - Reindexing for Series refers to the process of creating a new Series object with the data conformed to a new index.
   - This operation allows you to manipulate data, add missing values, or rearrange data according to a different set of labels while working exclusively with a Series object.
2. **Pandas DataFrame Re-indexing:**
   Reindexing can also be applied to Pandas DataFrames. You can reindex both rows (axis=0) and columns (axis=1) independently.
3. **Handling Missing Values using Re-Indexing:**
   - Reindexing also provides options for handling missing data.
   - You can use the method parameter to specify how missing values should be filled.
   - Some common methods include 'ffill' for forward filling, 'bfill' for backward filling, and more.

Pandas Iteration

Iterating through DataFrames:

- DataFrames are one of the primary data structures in Pandas.
- You can iterate through the rows using methods like “iterrows()” and “itertuples()”.
- “iterrows()” returns an iterator that yields index and row data as Pandas Series. However, it is not the most efficient method, especially for large DataFrames.
- “itertuples()” is faster than “iterrows()” and returns an iterator of named tuples, which can be more memory-efficient.

Iterating through Series:

- Series objects can be iterated using a simple ‘for’ loop, treating them like lists or arrays.
- You can use the ‘.iteritems()’ method to iterate through key-value pairs in a Series, making it useful for dictionary-like operations.

Vectorized Operations:

- Pandas is optimized for vectorized operations, meaning it performs operations on entire arrays rather than individual elements. This is much faster than traditional iteration.
- Minimize explicit iteration when working with Pandas to take full advantage of its speed and efficiency.

Using '.apply()' and '.applymap()':

- The '.apply()' method allows you to apply a function along the axis of a DataFrame or Series. It is particularly useful for custom operations.
- The '.applymap()' method is similar but works element-wise on DataFrames.

Conditional iteration:

You can use boolean indexing to iterate through data based on specific conditions. This allows you to select and work with subsets of your data.

Pandas Sorting:

Sorting data is a fundamental operation in data analysis and manipulation. Python, with its powerful data manipulation library, Pandas, offers various methods to sort data efficiently.
Understanding Data Sorting Data sorting involves arranging data in a specific order, making it easier to identify patterns, make comparisons, and gain insights. In Pandas, data can be sorted either by index or by values.

Different Types of approaches are:

1. **Sorting by Index:**

   a. Pandas DataFrames and Series can be sorted based on their index labels.

   b. This is particularly useful when the index represents meaningful labels or categories, and you want to reorder the data accordingly.

   c. To sort by index, you can use the 'sort_index()' method. You can specify the axis (0 for rows, 1 for columns) and the sorting order (ascending or descending).

2. **Sorting by Values:**
   a. Sorting by values is common when you want to arrange the data based on the content of one or more columns.

   b. The 'sort_values()' method is used for this purpose. You can specify the column(s) to sort by, the axis, and the sorting order.

3. **Multi-level Index Sorting:**

   a. Pandas also supports sorting with multi-level indexes. You can specify the levels to sort and the sorting order to suit your analysis requirements.

   b. The 'sort_index()' method can be used to achieve this.

Working With Text Data Options & Customization

Common Operations for Text Data in Pandas

- Pandas provide various string methods through the 'str' accessor for text data manipulation.
- You can access these methods using the '.str' attribute.

Code:

```
# Accessing text data and applying string methods
# Converting text to lowercase
df['Lowercase'] = df['Text'].str.lower()
# Finding the length of each text entry
df['Length'] = df['Text'].str.len()
# Splitting text into words
df['Words'] = df['Text'].str.split()
print(df)
```

Output:

```
                                  Text \
0                    Hello, how are you?
1                       Pandas is great!
2 Text data handling in Pandas is useful

                               Lowercase Length \
0                    hello, how are you? 19
1                       pandas is great! 16
2 text data handling in pandas is useful 38

                                       Words
0                       [Hello,, how, are, you?]
1                           [Pandas, is, great!]
2 [Text, data, handling, in, Pandas, is, useful]
```

Code:

```
#Analogy:- Counting the Number of Characters in a Text Column
import pandas as pd
# Sample DataFrame
data = {'Text': ['Hello, how are you?', 'Pandas is great!', 'Text data
handling in Pandas is useful']}
df = pd.DataFrame(data)
# Counting characters using len() and the str accessor
df['Char_Count'] = df['Text'].str.len()
print(df)
```

Output:

```
                                  Text Char_Count
0                    Hello, how are you?         19
1                       Pandas is great!         16
2 Text data handling in Pandas is useful         38
```

Code:

```
#Analogy-2:- Converting Text to Uppercase in a DataFrame Column
# Converting text to uppercase
df['Uppercase_Text'] = df['Text'].str.upper()
print(df)
```

Output:

```
                                  Text Char_Count \
0                    Hello, how are you? 19
1                       Pandas is great! 16
2 Text data handling in Pandas is useful 38

                           Uppercase_Text
0                    HELLO, HOW ARE YOU?
1                       PANDAS IS GREAT!
2 TEXT DATA HANDLING IN PANDAS IS USEFUL
```

Code:

```
#Analogy-3:- Extracting Words Starting with a Specific Letter
# Extracting words starting with 'P'
df['P_Words'] = df['Text'].str.split().apply(lambda x: [word for word in x
if word.startswith('P')])
print(df)
```

Output:

```
                                  Text Char_Count \
0                    Hello, how are you? 19
1                       Pandas is great! 16
2 Text data handling in Pandas is useful 38

                           Uppercase_Text P_Words
0                    HELLO, HOW ARE YOU? []
1                       PANDAS IS GREAT! [Pandas]
2 TEXT DATA HANDLING IN PANDAS IS USEFUL [Pandas]
```

Data Cleaning and Preprocessing Text Data:

Data cleaning and preprocessing are crucial steps in text data analysis. This involves handling missing values, removing duplicates, and normalizing text.

Handling Missing Values: Pandas provides methods like 'isnull()' and 'dropna()' to handle missing values in text data

```
# Handling missing values
df['Text'].fillna('No text available', inplace=True)
# Filling missing values with a default text
```

Removing Duplicates: To remove duplicate rows based on text data.

```
# Removing duplicate rows based on 'Text' column
df.drop_duplicates(subset='Text', keep='first', inplace=True)
```

Text Normalization: Text normalization involves converting text to a standard form, such as converting all text to lowercase or stemming (reducing words to their root form).

```
# Text normalization: converting text to lowercase
df['Normalized_Text'] = df['Text'].str.lower()
df['Normalized_Text']
```

```
0                       hello, how are you?
1                          pandas is great!
2 text data handling in pandas is useful
Name: Normalized_Text, dtype: object
```

Text Data Manipulation and Customization: Text data manipulation involves various operations such as extracting specific information, transforming text, and customizing data according to requirements.

```
#Analogy-1 : Text Data as a Bookshelf
import pandas as pd
# Sample DataFrame with book titles (text data)
data = {'book_title': ['The Great Gatsby', 'To Kill a Mockingbird',
'1984', 'Pride and Prejudice']}
df = pd.DataFrame(data)
# Sorting book titles alphabetically
df_sorted = df.sort_values('book_title')
print(df_sorted)
```

Output:

```
book_title


 2                  1984
 3   Pride and Prejudice
 0      The Great Gatsby
 1 To Kill a Mockingbird
```

Code:

```
#Analogy-2 : Text Data as a Letter Editing
import pandas as pd

# Sample DataFrame with text data
data = {'letter_content': ['hello', 'world', 'python', 'pandas']}
df = pd.DataFrame(data)
# Capitalizing first letter of each word
df['letter_content_capitalized'] = df['letter_content'].str.capitalize()
print(df)
```

Output:

```
  letter_content letter_content_capitalized
0          hello                      Hello
1          world                      World
2         python                     Python
3         pandas                     Pandas
```

Code:

```
#Analogy-3 : Text Data as Newspaper Articles
import pandas as pd
# Sample DataFrame with news headlines (text data)
data = {'headline': ['Breaking: New discovery in science', 'Politics:
Election results announced', 'Technology: AI revolutionizing industries']}
df = pd.DataFrame(data)
# Extracting category from headlines
df['category'] = df['headline'].str.split(':').str[0]
print(df)
```

Output:

```
                                    headline category
0                                           headline category
1       0        Breaking: New discovery in science Breaking
2 Techı 1      Politics: Election results announced Politics
        2 Technology: AI revolutionizing industries Technology
```

Advanced Text Data Handling in Pandas

- Advanced text data handling involves tokenization, vectorization, and dealing with multi-indexing to manage and process textual information more efficiently.

```
#Analogy-1 : Newspaper Article Word Count
import pandas as pd
# Sample text data (analogous to a newspaper article)
article = "Pandas is a powerful library for data manipulation. Pandas
provides easy-to-use data structures."
# Tokenizing words and counting their frequency
words = article.split()
word_freq = pd.Series(words).value_counts()
print(word_freq)
```

Output:

```
Pandas         2
data           2
is             1
a              1
powerful       1
library        1
for            1
manipulation.  1
provides       1
easy-to-use    1
structures.    1
dtype: int64
```

```
#Analogy-2 : Product Reviews Sentiment Analysis
from textblob import TextBlob
import pandas as pd
# Sample product review data (analogous to customer reviews)
reviews = [
"The product is fantastic and worth the price.",
"Okay product, but could be better.",
"Terrible product, not recommended."
]
# Performing sentiment analysis on reviews
sentiments = [TextBlob(review).sentiment.polarity for review in reviews]

df_sentiments = pd.DataFrame({'Review': reviews, 'Sentiment': sentiments})
print(df_sentiments)
```

Output:

```
Review  Sentiment
```

```
0 The product is fantastic and worth the price.   0.35
1 Okay product, but could be better.              0.50
2 Terrible product, not recommended.             -1.00
```

```
# Analogy-3 : Extracting Important Entities from Articles
import spacy
import pandas as pd
# Sample text data (analogous to a news article)
text = "Elon Musk, the CEO of SpaceX, announced plans to colonize Mars."
# Loading spaCy's English model
nlp = spacy.load("en_core_web_sm")
# Performing Named Entity Recognition (NER)
doc = nlp(text)
entities = [(ent.text, ent.label_) for ent in doc.ents]
df_entities = pd.DataFrame(entities, columns=['Entity', 'Label'])
print(df_entities)
```

Output:

```
  Entity      Label
0 Elon Musk   PERSON
1 Mars        LOC
```

Understanding Indexing and Selecting

- **Indexing with square brackets [ ]**: Pandas allows selecting specific elements, rows, or columns using square brackets.
- **Selection using .loc[] and .iloc[]:**
  - loc[] is used for label-based indexing, accessing rows and columns by their labels.
  - iloc[] is used for position-based indexing, accessing rows and columns by their integer position.

Code:

```
import pandas as pd

# Creating a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, 30, 35, 40],
'City': ['New York', 'San Francisco', 'Los Angeles', 'Chicago']}

df = pd.DataFrame(data)
# Selecting specific columns using square brackets
ages = df['Age']
```

```
print("Selecting column 'Age' using square brackets:")
print(ages)
print()
# Using .loc[] for label-based indexing
row = df.loc[1] # Selecting the row with label/index 1
print("Row with label/index 1 using .loc[]:")
```

Output:

```
Selecting column 'Age' using square brackets:
0 25
1 30
2 35
3 40
Name: Age, dtype: int64
Row with label/index 1 using .loc[]:
```

Indexing Methods in Pandas

- **Label-based Indexing (using .loc[]):** Using .loc[], you can access rows and columns based on their labels.
- **Position-based Indexing (using .iloc[]):** With .iloc[], you can access rows and columns based on their integer positions.
- **Boolean Indexing**: Boolean indexing allows filtering data based on a certain condition.

Code:

```
# Label-based Indexing with .loc[]
subset = df.loc[1:2, ['Name', 'City']] # Selecting rows 1 to 2 and
columns 'Name' and 'City'
print("Label-based Indexing with .loc[]:")
print(subset)
print()
# Position-based Indexing with .iloc[]
subset2 = df.iloc[0:2, 1:] # Selecting rows 0 to 1 and columns from the
second column onwards
print("Position-based Indexing with .iloc[]:")
print(subset2)
print()
# Boolean Indexing
filtered_data = df[df['Age'] > 30] # Selecting rows where Age is greater
than 30
print("Boolean Indexing:")
print(filtered_data)
```

Output:

```
Label-based Indexing with .loc[]:
      Name           City
```

```
1      Bob  San Francisco
2  Charlie    Los Angeles

Position-based Indexing with .iloc[]:
     Age           City
0     25       New York
1     30  San Francisco

Boolean Indexing:
       Name  Age  City
2   Charlie  35   Los Angeles
3     David  40   Chicago
```

Basic Indexing and Selection in Pandas

In Pandas, basic indexing and selection involve accessing specific rows, columns, or elements from a DataFrame using various methods:

- Analogy Code:

Code:

```
#Analogy-1

#In a bookstore's inventory, consider books organized by different
attributes. Let's represent these attributes as columns in a DataFrame.
import pandas as pd
# Bookstore inventory
data = {
'Title': ['Python for Data Science', 'Introduction to Machine
Learning', 'Data Analysis with Pandas'],
'Author': ['John Smith', 'Emma Johnson', 'Chris Brown'],
'Genre': ['Programming', 'Machine Learning', 'Data Science']
}
df = pd.DataFrame(data)
# Selecting 'Title' column
titles = df['Title']
print(titles)
```

Output:

```
0          Python for Data Science
1 Introduction to Machine Learning
2        Data Analysis with Pandas
Name: Title, dtype: object
```

Code:

```
# Analogy-2
# Imagine a company's sales report by month. We can represent each month's
# sales as rows in a DataFrame.

import pandas as pd
# Sales report by month
data = {
'Month': ['January', 'February', 'March', 'April', 'May'],
'Sales': [35000, 42000, 38000, 41000, 37000]
}
df = pd.DataFrame(data)
# Slicing to get sales for first three months
first_three_months = df[:3]
print(first_three_months)
```

Output:

```
     Month Sales
0  January 35000
1 February 42000
2    March 38000
```

Code:

```
# Analogy-3
# Think of a music playlist where each track represents an index. Resetting
# the index reorders the playlist.

import pandas as pd
# Music playlist
data = {
'Track': ['Song A', 'Song B', 'Song C', 'Song D'],
'Artist': ['Artist 1', 'Artist 2', 'Artist 1', 'Artist 3']
}
df = pd.DataFrame(data)
# Resetting index
df_reset = df.reset_index(drop=True)
print(df_reset)
```

Output:

```
   Track   Artist
0 Song A  Artist 1
```

```
1 Song B  Artist 2
2 Song C  Artist 1
3 Song D  Artist 3
```

Conditional Selection in Pandas: Conditional selection involves filtering data based on specific conditions

Code:

```
# Analogy-1
# Consider a company's employee dataset containing names and salaries.
# Here, we filter employees earning above a certain salary threshold
import pandas as pd

# Creating a DataFrame with employee data
data = {
'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
'Salary': [60000, 75000, 50000, 90000, 65000]
}
df = pd.DataFrame(data)

# Selecting employees earning above $70,000
high_earners = df[df['Salary'] > 70000]
print(high_earners)
```

Output:

```
   Name Salary
1   Bob 75000
3 David 90000
```

Code:

```
# Analogy-2
# Suppose you have a student dataset with their names and exam scores. We
# filter students who scored above a specific grade
import pandas as pd
# Creating a DataFrame with student names and scores
data = {
'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
'Score': [85, 70, 92, 68, 78]
}
df = pd.DataFrame(data)
# Selecting high-performing students with a score above 80
high_achievers = df[df['Score'] > 80]

print(high_achievers)
```

Output:

```
     Name Score
0   Alice 85
2 Charlie 92
```

Code:

```
#Analogy-3

#Imagine having a product catalog dataset containing product names and
prices. Here, we filter products within a specific price range
import pandas as pd
# Creating a DataFrame with product names and prices
data = {
'Product': ['Laptop', 'Phone', 'Tablet', 'Headphones', 'Camera'],
'Price': [1200, 800, 500, 200, 1500]
}
df = pd.DataFrame(data)
# Selecting products within the price range of $500 to $1000
affordable_products = df[(df['Price'] >= 500) & (df['Price'] <= 1000)]
print(affordable_products)
```

Output:

```
  Product Price
1   Phone 800
2  Tablet 500
```

Statistics with pandas

Loading Data:

```
import seaborn as sns
import pandas as pd

# Load the 'iris' dataset from seaborn
iris_data = sns.load_dataset('iris')
# Display first few rows of the dataset
print(iris_data.head())
```

Output:

sepal_length sepal_width petal_length petal_width species

```
0 5.1 3.5 1.4 0.2 setosa
1 4.9 3.0 1.4 0.2 setosa
2 4.7 3.2 1.3 0.2 setosa
3 4.6 3.1 1.5 0.2 setosa
4 5.0 3.6 1.4 0.2 setosa
```

 # Basic information about the DataFrame
print(iris_data.info())

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
# Column Non-Null Count Dtype
--- ------ -------------- -----
0 sepal_length 150 non-null float64
1 sepal_width 150 non-null float64
2 petal_length 150 non-null float64
3 petal_width 150 non-null float64
4 species 150 non-null object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
```

 # Summary statistics of numerical columns
print(iris_data.describe())

Output:

```
sepal_length sepal_width petal_length petal_width
count 150.000000 150.000000 150.000000 150.000000
mean 5.843333 3.057333 3.758000 1.199333
std 0.828066 0.435866 1.765298 0.762238
min 4.300000 2.000000 1.000000 0.100000
25% 5.100000 2.800000 1.600000 0.300000

50% 5.800000 3.000000 4.350000 1.300000
75% 6.400000 3.300000 5.100000 1.800000
max 7.900000 4.400000 6.900000 2.500000
```

Descriptive Statistics with pandas:

Pandas offers a range of functions to compute descriptive statistics on DataFrame columns, providing insights into the central tendency, dispersion, and shape of the dataset.

1. **Central Tendency:** Central tendency refers to the tendency of data to cluster around a central value or a typical value within a dataset. It helps to identify a single representative value that best summarizes the entire dataset. Like mean, median & mode.

2. **Dispersion (Variability):** Dispersion or variability measures the spread or extent of how the data points differ from the central value (mean, median, or mode). It indicates how much the data is scattered or spread out. Like Variance, Standard Deviation, Range & IQR

Code:

```
# Calculating mean, median, and standard deviation
print(iris_data.mean())
print(iris_data.median())
print(iris_data.std())
```

Output:

sepal_length 5.843333
sepal_width 3.057333
petal_length 3.758000
petal_width 1.199333
dtype: float64
sepal_length 5.80
sepal_width 3.00
petal_length 4.35
petal_width 1.30
dtype: float64
sepal_length 0.828066
sepal_width 0.435866

petal_length 1.765298
petal_width 0.762238
dtype: float64

```
# Describing the DataFrame
print(iris_data.describe())
```

Output:

sepal_length sepal_width petal_length petal_width
count 150.000000 150.000000 150.000000 150.000000
mean 5.843333 3.057333 3.758000 1.199333
std 0.828066 0.435866 1.765298 0.762238
min 4.300000 2.000000 1.000000 0.100000
25% 5.100000 2.800000 1.600000 0.300000
50% 5.800000 3.000000 4.350000 1.300000
75% 6.400000 3.300000 5.100000 1.800000
max 7.900000 4.400000 6.900000 2.500000

Aggregation Functions

Aggregation functions in pandas help summarize data based on certain conditions or groupings, allowing computation of statistics like mean, sum, count, etc., on grouped data.

```
import pandas as pd
from sklearn.datasets import load_iris
# Load the Iris dataset
iris = load_iris()
iris_data = pd.DataFrame(data=iris.data, columns=iris.feature_names)
# Aggregating with multiple functions
result = iris_data.agg({
'sepal length (cm)': ['mean', 'median', 'std'],
'sepal width (cm)': ['min', 'max'],
'petal length (cm)': ['sum', 'count']
})
print(result)
```

Output:

	sepal length (cm)	sepal width (cm)	petal length (cm)
mean	5.843333	NaN	NaN
median	5.800000	NaN	NaN
std	0.828066	NaN	NaN
min	NaN	2.0	NaN
max	NaN	4.4	NaN
sum	NaN	NaN	563.7
count	NaN	NaN	150.0

Data Cleaning and Handling Missing Values:

- Data cleaning involves preparing and tidying up the data for analysis.
- Handling missing values is a crucial step in this process.
- Missing values in datasets can affect analysis and modeling. Methods such as dropping missing values, filling them with a specific value, or imputing values based on statistical measures are common strategies.

```
import pandas as pd
from sklearn.datasets import load_iris
# Load Iris dataset
iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
# Introduce missing values
```

```
iris_df.loc[::5, 'sepal length (cm)'] = None
# Check for missing values
print("Missing values before handling:")
print(iris_df.isnull().sum())
# Handling missing values: Fill missing values with mean
iris_df.fillna(iris_df.mean(), inplace=True)
# Check missing values after handling
print("\nMissing values after handling:")
print(iris_df.isnull().sum())
```

Output:

Missing values before handling:
sepal length (cm) 30
sepal width (cm) 0
petal length (cm) 0
petal width (cm) 0
dtype: int64
Missing values after handling:
sepal length (cm) 0
sepal width (cm) 0
petal length (cm) 0
petal width (cm) 0
dtype: int64

Correlation and Covariance:

- Correlation measures the relationship between two variables, while covariance measures their joint variability.
- Correlation values range between -1 to 1, indicating the strength and direction of the relationship between variables.
- Covariance measures how two variables change together.

```
# Calculate correlation matrix
correlation_matrix = iris_df.corr()
# Calculate covariance matrix
covariance_matrix = iris_df.cov()
print("Correlation Matrix:")
print(correlation_matrix)
print("\nCovariance Matrix:")
print(covariance_matrix)
```

Outlier Detection and Treatment:

- Outliers are data points significantly different from other observations in a dataset.
- Outliers can affect statistical measures and model performance.

Techniques involve detecting and handling outliers, like removing or transforming them.

```
from scipy import stats
# Detect outliers using z-score
z_scores = stats.zscore(iris_df)
outliers = iris_df[(z_scores > 3).any(axis=1)]
# Remove outliers
iris_no_outliers = iris_df[(z_scores < 3).all(axis=1)]
print("Outliers:")
print(outliers)

print("\nDataFrame without outliers:")
print(iris_no_outliers)
```

Applying Functions to DataFrames:

- Pandas allows applying functions across rows/columns of DataFrames using various methods.
- Functions can be applied using '.apply()', '.map()', or '.applymap()' methods for specific column-wise or element-wise operations.

```
# Applying function to DataFrame columns
iris_df['sepal length squared'] = iris_df['sepal length
(cm)'].apply(lambda x: x ** 2)
# Applying function element-wise
iris_df_squared = iris_df.applymap(lambda x: x ** 2)
print("DataFrame with 'sepal length squared' column:")
print(iris_df[['sepal length (cm)', 'sepal length squared']].head())
print("\nDataFrame with all values squared:")
print(iris_df_squared.head())
```

Introduction to Window Functions

- Window functions in Pandas allow for performing calculations on a specific subset of data within a defined window or range.
- These functions operate on a sliding or rolling window over a series or DataFrame, enabling various analytical operations.
-

Rolling Windows

- Rolling windows are a fundamental aspect of window functions.
- They involve calculating statistics or applying functions to a specified window size as it moves through the data.

```
# Rolling Windows Example
window_size = 5
rolling_window = data['y'].rolling(window=window_size)
```

```
# Calculating Rolling Mean within the window
data['Rolling_Mean'] = rolling_window.mean()
# Displaying the DataFrame with Rolling Mean
print(data)
```

Window Statistics and Aggregations:

Window statistics and aggregations involve computing summary statistics within a window, which may include mean, sum, min, max, standard deviation, and custom aggregations.

```
# Window Statistics and Aggregations Example
window_stats = data['y'].rolling(window=window_size)
# Calculating Rolling Sum within the window
data['Rolling_Sum'] = window_stats.sum()
# Calculating Rolling Minimum within the window
data['Rolling_Min'] = window_stats.min()
# Displaying the DataFrame with Rolling Sum and Min
print(data)
```

Window Grouping and Operations:

Window grouping involves applying operations within specific groups of data rather than the entire dataset. This is achieved using Pandas' groupby() function along with window operations.

```
print(data)
```

Custom Window Functions:

Custom window functions involve defining and applying user-defined functions to windows in Pandas.

```
# Define a custom function to calculate the mean ignoring NaN values
def custom_mean(window):
return window.mean(skipna=True)
# Applying the custom function to a rolling window
window_size = 5
custom_window = data['y'].rolling(window=window_size)
data['Custom_Rolling_Mean'] = custom_window.apply(custom_mean)
# Displaying the DataFrame with Custom Rolling Mean
print(data)
```

Handling Missing Values in Windows:

Handling missing values within windows involves strategies like filling or interpolating missing data to ensure the integrity of window-based calculations.

```
# Introducing missing values into the 'y' column
data.loc[5:10, 'y'] = np.nan
# Filling missing values with forward fill within a rolling window
filled_window =
data['y'].fillna(method='ffill').rolling(window=window_size)
# Calculating rolling mean with filled missing values
data['Filled_Rolling_Mean'] = filled_window.mean()
# Displaying the DataFrame with Filled Rolling Mean
print(data)
```

Introduction to Date Functionality in Pandas

Date and Time Data Types in Pandas

Pandas primarily utilize two main data types for handling date and time information

- **'Timestamp' Data Type**
  - Represents a single timestamp and is the fundamental type for Pandas to work with dates and times.
  - Each Timestamp object can store nanosecond precision and is based on the datetime64 data type in NumPy.

- **DatetimeIndex and PeriodIndex**
  - DatetimeIndex is used to index Pandas data structures like Series and DataFrame based on timestamps
  - PeriodIndex represents a period of time, such as a specific day, month, quarter, etc..

Date Range Creation:

- Creating date ranges is a common task when working with time series data.
- Pandas provides the 'pd.date_range()' function for generating sequences of dates.

```
import pandas as pd
# Generating a date range from 2023-01-01 to 2023-01-10 with daily
frequency
date_range = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
print(date_range)
```

Output:

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
               '2023-01-05', '2023-01-06', '2023-01-07', '2023-01-08',
               '2023-01-09', '2023-01-10'],
              dtype='datetime64[ns]', freq='D')
```

```
#Analogy-1
#Suppose you need to generate a date range for a daily sales report over a
specific period.
import pandas as pd
# Generating a date range for daily sales report for the last week
start_date = pd.Timestamp.now() - pd.DateOffset(weeks=1)
end_date = pd.Timestamp.now()

date_range = pd.date_range(start=start_date, end=end_date, freq='D')
print(date_range)
```

Output:

```
DatetimeIndex(['2023-12-01 06:37:15.547683', '2023-12-02 06:37:15.547683',
               '2023-12-03 06:37:15.547683', '2023-12-04 06:37:15.547683',
               '2023-12-05 06:37:15.547683', '2023-12-06 06:37:15.547683',
               '2023-12-07 06:37:15.547683', '2023-12-08 06:37:15.547683'],
              dtype='datetime64[ns]', freq='D')
```

Date Parsing and Formatting:

- Pandas facilitate parsing strings into Timestamp objects and formatting Timestamp objects into different string representations using 'strftime' and 'strptime'.

```
#Analogy-1
#Let's consider a scenario where a user inputs a date in string format,
and the code converts it into a 'datetime' object.
from datetime import datetime
# User-entered date in string format
user_input = "2023-11-16"
# Parsing string to datetime object
parsed_date = datetime.strptime(user_input, '%Y-%m-%d')
print(parsed_date)
```

Output:

```
Friday, December 08, 20230:00
```

```
#Analogy-2
#Imagine you have a 'datetime' object, and you want to display it in a
different format.
from datetime import datetime
# Current datetime object
current_datetime = datetime.now()

# Formatting datetime object to a custom string format
formatted_date = current_datetime.strftime('%A, %B %d, %Y')
print(formatted_date)
```

Output:

```
Friday, December 08, 2023
```

```
#Analogy-3
#Suppose you have a 'datetime' object that includes time information, and
you want to display it without seconds.
from datetime import datetime
# Current datetime object
current_datetime = datetime.now()
# Formatting datetime object to exclude seconds
formatted_date = current_datetime.strftime('%Y-%m-%d %H:%M')
print(formatted_date)
```

Output:

```
2023-12-08 06:53
```

Working with Date and Time Components:

- Pandas provides the 'dt' accessor, allowing easy extraction of different components (like year, month, day, etc.) from date and time objects.
- This accessor enables accessing various attributes and methods for datetime objects in a Pandas Series or DataFrame.

```
#Analogy-1
#Analogous to a calendar where you extract specific details like the year,
month, and day from a date.
import pandas as pd
# Creating a DataFrame with date-time data
data = {'date': ['2023-01-01', '2023-02-01', '2023-03-01']}
df = pd.DataFrame(data)
df['date'] = pd.to_datetime(df['date']) # Convert 'date' column to
datetime
# Extracting components
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
print(df)
```

Output:

```
        date year month day
0 2023-01-01 2023 1     1
1 2023-02-01 2023 2     1
2 2023-03-01 2023 3     1
```

Output:

```
#Analogy-2
#Similar to determining a person's age by subtracting their birth year
from the current year.
import pandas as pd
# Current date
current_date = pd.to_datetime('2023-11-16')
# Birthdates
birthdates = ['1990-05-15', '1985-10-20', '2000-03-25']
# Calculating ages
ages = current_date.year - pd.to_datetime(birthdates).year
print(ages)
```

Output:

```
Int64Index([33, 38, 23], dtype='int64')
```

```
#Analogy-3
#Like counting the days left until a specific event (e.g., New Year's
```

```
Day).
import pandas as pd
# Current date

current_date = pd.to_datetime('2023-11-16')
# Future event date
event_date = pd.to_datetime('2024-01-01')
# Countdown calculation
days_until_event = (event_date - current_date).days
print(f"Days until New Year's Day: {days_until_event}")
```

Output:

```
Days until New Year's Day: 46
```

Resampling and Shifting Time Series Data:

- Resampling in Pandas refers to changing the frequency of a time series data. This is achieved using the 'resample()' method.
- It enables the aggregation of data based on a specific time-frequency (e.g., daily data aggregated into monthly data).

```
#Analogy-1
#Real-time scenario: Aggregating daily sales data into monthly summaries
for reporting purposes.
import pandas as pd
# Simulated daily sales data
date_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')
sales_data = pd.DataFrame({'Date': date_rng, 'Sales':
range(len(date_rng))})
# Resampling to monthly sales
sales_data.set_index('Date', inplace=True)
monthly_sales = sales_data.resample('M').sum()
print(monthly_sales)
```

Output:

```
Date       Sales
2023-01-31 465
2023-02-28 1246
2023-03-31 2294
2023-04-30 3135
2023-05-31 4185
2023-06-30 4965
2023-07-31 6076
2023-08-31 7037
```

```
2023-09-30 7725
2023-10-31 8928
2023-11-30 9555
2023-12-31 10819
```

```
#Analogy-2
#Real-time scenario: Averaging hourly weather data to daily values for
analysis or visualization.
import pandas as pd
import numpy as np
# Simulated hourly weather data
np.random.seed(0)
date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='H')
weather_data = pd.DataFrame({'Date': date_rng, 'Temperature':
np.random.randint(0, 30, len(date_rng))})
# Resampling to daily average temperature
weather_data.set_index('Date', inplace=True)
daily_weather = weather_data.resample('D').mean()
print(daily_weather)
```

Output:

```
Date       Temperature
2023-01-01 13.541667
2023-01-02 15.500000
2023-01-03 10.875000
2023-01-04 12.833333
2023-01-05 13.500000
2023-01-06 9.750000
2023-01-07 14.666667
2023-01-08 12.541667
2023-01-09 13.916667
2023-01-10 11.000000
```

```
#Analogy-3
#Real-time scenario: Downsampling high-frequency (e.g., minute-level)
stock prices to hourly frequency for analysis.
import pandas as pd
import numpy as np
# Simulated high-frequency stock price data
np.random.seed(1)
```

```
date_rng = pd.date_range(start='2023-01-01', end='2023-01-02', freq='T')
stock_prices = pd.DataFrame({'Timestamp': date_rng, 'Price':
np.random.randint(50, 150, len(date_rng))})
# Resampling to hourly stock prices
stock_prices.set_index('Timestamp', inplace=True)
hourly_prices = stock_prices.resample('H').ohlc()
print(hourly_prices)
```

Output:

```
                     Price
Timestamp           open high low close
2023-01-01 00:00:00 87   146  50  130
2023-01-01 01:00:00 91   148  50  86
2023-01-01 02:00:00 57   148  52  134
2023-01-01 03:00:00 93   149  50  129
2023-01-01 04:00:00 91   149  55  104
2023-01-01 05:00:00 50   148  50  93
2023-01-01 06:00:00 51   149  51  147
2023-01-01 07:00:00 88   148  50  83
2023-01-01 08:00:00 60   149  52  117
2023-01-01 09:00:00 84   149  53  110
2023-01-01 10:00:00 62   148  55  94
2023-01-01 11:00:00 66   147  50  51
2023-01-01 12:00:00 81   149  50  99
2023-01-01 13:00:00 96   147  50  93
2023-01-01 14:00:00 111  149  51  80
2023-01-01 15:00:00 57   147  51  71
2023-01-01 16:00:00 101  148  50  120
2023-01-01 17:00:00 126  149  55  87
2023-01-01 18:00:00 139  145  54  127
2023-01-01 19:00:00 92   149  54  88
2023-01-01 20:00:00 133  149  50  124
2023-01-01 21:00:00 139  149  52  59
2023-01-01 22:00:00 138  148  50  96
2023-01-01 23:00:00 70   148  50  119
2023-01-02 00:00:00 118  118  118 118
```

Time Zone Handling

- Pandas enables easy handling of time zones for datetime objects using 'tz_localize()' and 'tz_convert()' methods.
- This functionality is crucial when working with datasets from different time zones or requiring conversion between time zones.

```
#Analogy-1
#This analogy represents travelers moving from one city to another and
adjusting their watches accordingly.
import pandas as pd
# Creating a datetime series in UTC
dates = pd.date_range(start='2023-01-01 12:00', periods=3, freq='H',
tz='UTC')
# Converting to another time zone (e.g., New York)
dates_ny = dates.tz_convert('America/New_York')
print(dates_ny)
```

Output:

```
DatetimeIndex(['2023-01-01 07:00:00-05:00',
               '2023-01-01 08:00:00-05:00',
               '2023-01-01 09:00:00-05:00'],
              dtype='datetime64[ns, America/New_York]',
              freq='H')
```

```
#Analogy-2
#Imagine a location that observes daylight saving time and adjusts clocks
accordingly.
import pandas as pd
# Create a datetime series before and after DST change

dates_dst = pd.date_range(start='2023-03-10 00:00', end='2023-03-13
00:00', freq='H', tz='US/Eastern')
print(dates_dst)
```

```
#Analogy-3

#Think of global operations where times need to be compared irrespective
of their time zones.
import pandas as pd
# Creating datetime series in different time zones
dates_utc = pd.date_range(start='2023-01-01 12:00', periods=3, freq='H',
tz='UTC')
dates_ny = pd.date_range(start='2023-01-01 12:00', periods=3, freq='H',
tz='America/New_York')
# Comparing times
print(dates_utc > dates_ny)
```

Output:

```
[False False False]
```

Handling Missing Dates and Time Series Gaps:

- When working with time series data, missing dates or gaps can occur.
- Pandas provides methods like 'reindex()' or 'asfreq()' to handle missing dates by either adding missing dates and filling them with default values or by specifying a frequency to fill in the gaps.

```
#Analogy-1
#Suppose you have daily sales data, but some dates are missing. You want
to fill in the missing dates and set sales as 0 for those dates.
import pandas as pd
# Sample daily sales data with missing dates
data = {'date': ['2023-01-01', '2023-01-02', '2023-01-04'],

'sales': [100, 120, 150]}

df = pd.DataFrame(data)
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)
# Reindexing to fill missing dates and setting sales as 0 for missing
dates
date_range = pd.date_range(start=df.index.min(), end=df.index.max())
df = df.reindex(date_range, fill_value=0)
print(df)
```

Output:

```
            sales
2023-01-01  100
2023-01-02  120
2023-01-03  0
2023-01-04  150
```

```
#Analogy-2
#Suppose you have monthly temperature readings, but the data intervals are
irregular. You want to resample it to monthly data with average
temperature values.
# Sample monthly temperature data with irregular intervals
data = {'date': ['2023-01-01', '2023-02-05', '2023-04-15'],

'temperature': [25, 28, 22]}
```

```
df = pd.DataFrame(data)
df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)
# Resampling to monthly frequency with average temperature
df_resampled = df.resample('M').mean()
print(df_resampled)
```

Output:

```
date       temperature
2023-01-31 25.0
2023-02-28 28.0
2023-03-31 NaN
2023-04-30 22.0
```

```
#Analogy-3
#Suppose you have financial data that doesn't include weekends. You want
to fill in the missing weekends with the last available price.
# Sample financial data without weekends
data = {'date': ['2023-11-01', '2023-11-02', '2023-11-03'],

'price': [100, 105, 110]}

df = pd.DataFrame(data)

df['date'] = pd.to_datetime(df['date'])
df.set_index('date', inplace=True)
# Reindexing to include weekends and filling missing values with last
available price
idx = pd.date_range(start=df.index.min(), end=df.index.max(), freq='D')
df = df.reindex(idx, method='ffill')
print(df)
```

Output:

```
           price
2023-11-01 100
2023-11-02 105
2023-11-03 110
```

Pandas Timedelta:

Pandas' Timedelta represents a duration or difference in time. It's useful for performing arithmetic operations on dates or time-related data, such as calculating time differences or adding/subtracting time intervals.

import pandas as pd

```
# Creating a Timedelta of 5 days
delta = pd.Timedelta(days=5)
# Using Timedelta with dates
date1 = pd.to_datetime('2023-01-01')
date2 = date1 + delta # Adding 5 days to date1
print(date2)
```

Output:

```
2023-01-06 00:00:00
```

Handling Time Differences:

Timedelta can also be used to calculate the difference between two dates or time-related data.

```
# Calculating time difference
start_time = pd.to_datetime('2023-01-01 12:00:00')
end_time = pd.to_datetime('2023-01-02 12:00:00')
time_diff = end_time - start_time
print(time_diff)
```

Output:

```
1 days 00:00:00
```

Application in Time Series Operations:

Timedelta is often used in time series operations, such as shifting or creating date offsets.

```
#Analogy-1
#Stock Market Analysis
import pandas as pd
import yfinance as yf
# Fetching stock data from Yahoo Finance
stock_data = yf.download('AAPL', start='2022-01-01', end='2023-01-01')
# Resampling daily stock data to monthly frequency
monthly_stock_data = stock_data['Close'].resample('M').mean()
print(monthly_stock_data)
```

```
#Analogy-2
#Website Traffic Analysis
import pandas as pd
import numpy as np
# Creating a sample DataFrame with website traffic data
date_rng = pd.date_range(start='2023-01-01', end='2023-01-31', freq='H')
website_traffic = pd.DataFrame(date_rng, columns=['date'])
```

```
traffic_values = [100, 120, 130, 110, 90] * (len(date_rng) // 5)
website_traffic['visitors'] = np.resize(traffic_values, len(date_rng))
# Resampling hourly data to daily traffic
daily_website_traffic = website_traffic.resample('D', on='date').sum()
print(daily_website_traffic)
```

```
#Analogy-3
#Temprature Analysis
import pandas as pd
import numpy as np
# Creating a sample DataFrame with temperature data
date_rng = pd.date_range(start='2023-01-01', end='2023-01-31', freq='H')
temperature_data = pd.DataFrame(date_rng, columns=['date'])
temperature_values = [20, 21, 22, 23, 24] * (len(date_rng) // 5)
temperature_data['temperature'] = np.resize(temperature_values,
len(date_rng))
# Resampling hourly temperature to daily average
daily_avg_temperature = temperature_data.resample('D', on='date').mean()
print(daily_avg_temperature)
```