Usage of pandas in Data science

Pandas is a powerful and widely-used open-source data manipulation and analysis library for Python. It is built on top of NumPy and is particularly useful for working with structured data such as time series, tables, or datasets similar to those found in databases and spreadsheets.

Before we start using pandas ,firstly importing of pandas can be done as shown below

```
import pandas as pd
```

pandas have 2 Data structures .They are

1. series
2. Data frame

Series: A one-dimensional labeled array capable of holding any data type. It can be thought of as a single column in a table.

DataFrame: A two-dimensional labeled data structure with columns that can hold different types of data. It is similar to a table in a relational database or an Excel spreadsheet.

Series

1. Creation of series with list

```
data = [10, 20, 30, 40, 50]# Creating a Series from a list
series = pd.Series(data)
print(series)
```

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Series from a dictionary, where the dictionary keys become the index:

```
data = {"a": 10, "b": 20, "c": 30, "d": 40, "e": 50}# Creating a Series from a dictionary
series = pd.Series(data)
print(series)
```

```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

Creating a Series from a Scalar Value

If we want to create a Series with the same value repeated, we can use a scalar value:

```
series = pd.Series(100, index=["a", "b", "c", "d", "e"])# Creating a Series from a scalar value
print(series)
```

```
a    100
b    100
c    100
d    100
e    100
dtype: int64
```

2. Accessing Data in a Series

We can access elements in a Series using the index:

```
print(series["a"])  # single element
print(series["a", "c", "e"])14 multiple elements
```

```
10
a    10
c    30
e    50
dtype: int64
```

```
data = [10, 20, 30, 40, 50]
index = ["a", "b", "c", "d", "e"]
series = pd.Series(data, index=index)
print(series)
```

```
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

You can also access data by its integer position using the .iloc[]:

```
print(series.iloc[2])
print(series.iloc[0, 3, 4])
```

```
30
a    10
d    40
e    50
dtype: int64
```

Slicing Data

We can slice a Series to access a range of elements. Slicing works both with index labels and integer positions.

Slicing by Index Label

```
print(series["b":"d"])
```

```
b    20
c    30
d    40
dtype: int64
```

Slicing by Integer Position

```
print(series.iloc[1:4])
```

```
b    20
c    30
d    40
dtype: int64
```

Boolean Indexing

Boolean indexing allows you to filter data based on conditions.

```
print(series[series > 20])
```

```
c    30
d    40
e    50
dtype: int64
```

3. Vectorized Operations

Vectorized operations allow you to perform element-wise operations on the entire Series.

Arithmetic Operations

We can perform basic arithmetic operations directly on a Series:

```
print(series + 5)
print(series * 2)
print(series / 2)
```

```
a    15
b    25
c    35
d    45
e    55
dtype: int64
a    20
b    40
c    60
d    80
e    100
dtype: int64
a     5.0
b    10.0
c    15.0
d    20.0
e    25.0
dtype: float64
```

Applying Functions to a Series

We can apply any function to each element of a Series using the apply() method or directly using vectorized functions:

Using a Custom Function with apply()

```
def square(x):  # Defining a function
    return x * x
squared_series = series.apply(square)
print(squared_series)
```

```
a    100
b    400
c    900
d   1600
e   2500
dtype: int64
```

4. Handling Missing Data in Pandas Series

Missing data is a common issue in datasets. Pandas provides several methods to handle missing values efficiently.

Detecting Missing Data

detect missing data using the isnull() or notnull() methods:

```
data = [10, 20, None, 30, 50]
series = pd.Series(data)
print(series.isnull())  # For missing values
print(series.notnull()) # For non-missing values
```

```
0    False
1    False
2     True
3    False
4    False
dtype: bool
0    True
1    True
2    False
3    True
4    True
dtype: bool
```

Filling Missing Data

fill missing values with a specific value using fillna()

```
filled_series = series.fillna(0)
print(filled_series)
```

```
0    10.0
1    20.0
2     0.0
3    30.0
4    50.0
dtype: float64
```

Dropping Missing Data

drop any element with missing data using dropna()

```
clean_series = series.dropna()
print(clean_series)
```

```
0    10.0
1    20.0
3    30.0
4    50.0
dtype: float64
```

Data frame

```
        A Pandas DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is one of the most commonly used data
    structures in the pandas library .
```

1. Creating a DataFrame

A DataFrame can be created from various data structures, such as dictionaries, lists, NumPy arrays, or even other DataFrames.

By using dictionaries

```
data = {
    "Name": ["Alice", "Bob", "Charlie", "David"],
    "Age": [24, 27, 22, 32],
    "City": ["New York", "Los Angeles", "Chicago", "Houston"]
}
df = pd.DataFrame(data)
print(df)
```

```
      Name  Age         City
0    Alice   24     New York
1      Bob   27  Los Angeles
2  Charlie   22      Chicago
3    David   32      Houston
```

We can also create a DataFrame from a list of dictionaries, where each dictionary represents a row.

```
data = [
    {"Name": "Alice", "Age": 24, "City": "New York"},
    {"Name": "Bob", "Age": 27, "City": "Los Angeles"},
    {"Name": "Charlie", "Age": 22, "City": "Chicago"},
    {"Name": "David", "Age": 32, "City": "Houston"}
]
df = pd.DataFrame(data)
print(df)
```

```
      Name  Age         City
0    Alice   24     New York
1      Bob   27  Los Angeles
2  Charlie   22      Chicago
3    David   32      Houston
```

Accessing Data in a DataFrame

Data in a DataFrame can be accessed using labels, similar to how you would access data in a dictionary or a Series.

Accessing Columns can be done by using the column name:

```
print(df["Name"])
print(df[["Name", "City"]])
```

```
0      Alice
1        Bob
2    Charlie
3      David
Name: Name, dtype: object
      Name         City
0    Alice     New York
1      Bob  Los Angeles
2  Charlie      Chicago
3    David      Houston
```

Accessing Rows can be done by using labels via the loc method, or by integer position using the iloc method.

```
print(df.loc[1])
print(df.iloc[1])
```

```
Name          Bob
Age            27
City    Los Angeles
Name: 1, dtype: object
Name          Bob
Age            27
City    Los Angeles
Name: 1, dtype: object
```

Basic Operations on DataFrames

Pandas DataFrames support various operations that make data manipulation easy.

Arithmetic Operations on DataFrames are performed element-wise:

```
df = {'A': [1, 2, 3],  # numeric data
      'B': [4, 5, 6],
      'C': [7, 8, 9]}
df = pd.DataFrame(data)
print(df * 2)
print(df + 5)
print(df / 2)
```

```
   A   B   C
0  2   8  14
1  4  10  16
2  6  12  18
   A  B  C
0  6  9  12
1  7  10 13
2  8  11 14
     A    B    C
0  0.5  2.0  3.5
1  1.0  2.5  4.0
2  1.5  3.0  4.5
```

Data Aggregation

We can perform aggregation functions on DataFrame columns, such as sum, mean, max, etc.

```
print(df.sum())
print(df.mean())
```

```
A    6
B   15
C   24
dtype: int64
A    2.0
B    5.0
C    8.0
dtype: float64
```

Handling Missing Data

DataFrames provide easy methods to detect, fill, or drop missing data

```
data = {
    "Name": ["Alice", "Bob", None, "David"],
    "Age": [24, None, 22, 32],
    "City": ["New York", "Los Angeles", "Chicago", None]
}
df = pd.DataFrame(data)
print(df.isnull())  # Detect missing values
print(df.fillna("Unknown"))
print(df.dropna())
```

```
    Name    Age   City
0  False  False  False
1  False   True  False
2   True  False  False
3  False  False   True
      Name  Age         City
0    Alice   24     New York
1      Bob  Unknown  Los Angeles
2  Unknown   22.0     Chicago
3    David   32.0     Unknown
    Name Age      City
0  Alice  24  New York
```

Using csv file

```
import csv

data = [
['Name', 'Roll_no', 'section'],
['Alice', 25, 'cse'],
['Robin', 26, 'cse'],
['Charlie',28,'cse']
]

with open('student.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)

df = pd.read_csv('student.csv')
print("\nDataFrame from CSV:\n",df.head())
```

```
DataFrame from CSV:
      Name  roll_no section
0    Alice       25     cse
1    Robin       26     cse
2  Charlie       28     cse
```

```
print("\n Missing values:\n", df.isnull().sum())
```

```
 Missing values:
Name       0
roll_no    0
section    0
dtype: int64
```

Advantages of pandas

1. Ease of Use

Intuitive Data Structures: Pandas provides easy-to-use data structures like Series and DataFrame that allow for straightforward manipulation and analysis of data.

Label-Based Indexing: The ability to use labels for rows and columns (like in DataFrames) makes data access and manipulation more intuitive, especially for users transitioning from Excel or SQL.

2. Handling of Missing Data

Flexible Handling: Pandas provides various options for dealing with missing data, such as filling them with specific values, forward/backward filling, or dropping them entirely.

Integration with Other Libraries: Pandas integrates well with NumPy, allowing for seamless handling of missing data using both libraries' functions.

3. Powerful Data Wrangling Capabilities

Data Cleaning: Pandas offers a wide range of functions for cleaning messy data, including handling duplicates, merging, concatenating, and reshaping data.

Data Transformation: You can easily transform data, perform group-by operations, and apply custom functions to datasets, making it a powerful tool for data preprocessing.

4. High Performance

Optimized for Speed: Pandas is built on top of NumPy, which is written in C, making it highly efficient for handling large datasets.

Vectorized Operations: Operations in pandas are vectorized, which means they are applied element-wise across arrays, allowing for faster computation compared to loops in Python.

5. Flexibility

Supports Diverse Data Formats: Pandas can read and write data fromto various formats, including CSV, Excel, SQL databases, JSON, and more.

Heterogeneous Data Handling: Unlike NumPy, which requires homogeneous data types, pandas DataFrames can handle heterogeneous data types, making it suitable for real-world datasets.

6. Rich Functionality

Statistical and Mathematical Operations: Pandas offers built-in functions for statistical analysis, such as mean, median, mode, standard deviation, correlation, and more.

Time Series Data: Pandas has specialized functionality for time series data, including date range generation, frequency conversion, moving window statistics, and more.

Applications of pandas

1. Financial Analysis and Modeling

Stock Market Analysis: Pandas is used to analyze and visualize stock price data, calculate moving averages, and perform backtesting of trading strategies.

Portfolio Management: Financial analysts use pandas to manage and optimize investment portfolios, calculate risk metrics, and simulate different investment scenarios.

Risk Management: Banks and financial institutions use pandas to process and analyze large datasets for assessing credit risk, market risk, and operational risk.

2. Healthcare Data Analysis

Electronic Health Records (EHR) Processing: Pandas is used to clean and analyze patient data from EHR systems, helping healthcare providers make data-driven decisions about patient care.

Genomic Data Analysis: Researchers use pandas to manage and analyze large genomic datasets, facilitating studies in genetics and personalized medicine. Clinical Trial Data: Pandas is used to organize, analyze, and visualize data from clinical trials, enabling researchers to monitor results and draw conclusions about drug efficacy and safety.

3. Social Media and Web Analytics

Social Media Sentiment Analysis: Pandas is used to process and analyze data from social media platforms, helping companies gauge public sentiment about their brand or products.

Website Traffic Analysis: Web analysts use pandas to analyze website traffic data, track user behavior, and identify trends that can inform website optimization efforts.

Content Recommendation: E-commerce and media companies use pandas to analyze user data and build recommendation systems that suggest products or content to users based on their preferences.

4. Sports Analytics

Player Performance Analysis: Sports teams and analysts use pandas to track and analyze player performance data, helping to make decisions on player selection, training, and game strategy.

Game Statistics: Pandas is used to analyze game statistics, identify trends, and generate insights that can be used by coaches and analysts to improve team performance.

Fantasy Sports: Pandas is used to analyze player data and predict performance, helping fantasy sports enthusiasts make informed decisions when selecting their teams.

5. Retail and E-Commerce

Customer Behavior Analysis: Retailers use pandas to analyze customer purchase data, understand shopping patterns, and optimize product placement and promotions.

Pricing Strategy: Pandas is used to analyze competitors' pricing data, track market trends, and develop dynamic pricing strategies.

In [ ]: