Name: DHWANI GUPTA

Enrollment Number: EBEON0223761985

Pandas Theory Questions

1. What is the purpose of the pandas library in Python? What are the available data structures in pandas?

The pandas library in Python is a powerful and widely used data manipulation and analysis tool. Its primary purpose is to provide data structures and functions to efficiently work with structured data, such as tabular and time series data.

The two main data structures provided by pandas are:

I. **DataFrame**: A DataFrame is a 2-dimensional labeled data structure with columns of potentially different data types. It can be thought of as a table or a spreadsheet, where each column represents a variable, and each row represents an observation or entry. DataFrames are versatile and offer functionalities similar to those found in spreadsheets or SQL tables.

II. **Series**: A Series is a one-dimensional labeled data structure. It can be seen as a single column of a DataFrame or a standalone object. Series are commonly used to represent sequences of data, such as time series, and they have built-in methods for easy manipulation and analysis.

These data structures provided by pandas are designed to handle large datasets efficiently, with various methods and functions for data cleaning, filtering, grouping, merging, reshaping, and more. pandas also integrate well with other libraries commonly used in the data science ecosystem, such as NumPy, Matplotlib, and scikit-learn, making it a popular choice for data analysis tasks in Python.

2. What are the notable features provided by the pandas library?

The pandas library offers several notable features that make it a powerful tool for data manipulation and analysis in Python. Here are some of its key features:

I. **Data Alignment**: pandas aligns data automatically based on labels, making it easy to perform operations on datasets with different indexes or column names. This feature simplifies data integration and handling missing or mismatched values.

II. **Data Input/Output**: pandas provide functions to read and write data in various formats, including CSV, Excel, SQL databases, JSON, and more. This makes it convenient to load data from different sources and save the results of your analysis.

III. **Data Cleaning**: pandas offer methods to handle missing data, duplicate data, and inconsistent data. You can easily filter out or fill in missing values, drop duplicate entries, and apply transformations to clean up your dataset.

IV. **Data Transformation**: pandas provide powerful tools for transforming data, such as filtering, sorting, grouping, and aggregating. You can reshape your data using pivot tables, melt/unmelt operations, and apply mathematical or statistical functions to subsets of your data.

V. **Time Series Analysis**: pandas have extensive support for working with time series data. It provides functionality for date/time indexing, resampling, frequency conversion, shifting, rolling window operations, and more. This makes it convenient for analyzing and manipulating time-based data.

VI. **Merge and Join**: pandas enable the merging and joining of datasets based on common columns or indexes. You can combine multiple DataFrames based on specific criteria, such as inner, outer, left, or right joins, and perform complex data merging operations.

VII. **Data Visualization**: pandas integrates well with other popular visualization libraries, such as Matplotlib and Seaborn, to create informative plots and charts. It provides a convenient interface to generate various types of plots, histograms, box plots, scatter plots, and more.

VIII. **Efficiency and Performance**: pandas is designed to handle large datasets efficiently. It uses optimized data structures and algorithms, allowing for fast and memory-efficient data processing. Additionally, pandas support parallelization, allowing you to speed up certain operations using multiple CPU cores.

These features, among others, make pandas a versatile library for data analysis tasks and have contributed to its popularity in the Python data science community.

3. **What are the differences between a series and a dataframe in pandas?**
    - A **DataFrame** is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column.
    - A **Series**, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list. And in fact you can create one with nothing more than a list. A **Series** is, in essence, a single column of a DataFrame. So you can assign row labels to the Series the same way as before, using an index parameter. However, a Series does not have a column name, it only has one overall name.

In pandas, a Series and a DataFrame are two distinct data structures with different characteristics. Here are the key differences between a Series and a DataFrame:

I. **Dimensionality**: A Series is a one-dimensional labeled array that can hold any data type (integer, string, float, etc.). It is similar to a column in a spreadsheet or a single variable in a dataset. In contrast, a DataFrame is a two-dimensional labeled data structure with rows and columns. It can be thought of as a table or a spreadsheet, where each column represents a variable, and each row represents an observation or entry.

II. **Structure**: A Series consists of an index and a corresponding array of values. The index provides labels for each element in the array, allowing for easy access and alignment of data. On the other hand, a DataFrame has both row and column indexes. It is essentially a collection of Series objects, where each column represents a Series.

III. **Data Storage**: While a Series stores a single column of data, a DataFrame stores multiple columns of data. The columns in a DataFrame can have different data types. This allows for the representation of heterogeneous data, where each column can be of a different type (e.g., integer, string, float, datetime).

IV. **Functionality**: Series and DataFrames have different functionalities. Series provides methods and operations specific to one-dimensional data, such as indexing, slicing, mathematical operations, and statistical computations. DataFrame, being a two-dimensional structure, provides additional operations that work across columns and rows. These include data alignment, grouping, merging, reshaping, and more.

V. **Use Cases**: Series are often used to represent a single variable or sequence of data, such as time series, sensor readings, or a single column from a larger dataset. DataFrames, with their tabular structure, are well-suited for working with structured datasets containing multiple variables.

They are commonly used for data analysis, data manipulation, and performing operations across multiple columns or variables.

In **summary**, a Series is a one-dimensional labeled data structure, while a DataFrame is a two-dimensional labeled data structure with multiple columns. Series represent a single variable or sequence of data, while DataFrames represent structured datasets with multiple variables.

4. **What are the various methods for creating a series in pandas?**

In pandas, there are multiple methods available for creating a Series. Here are some common methods for creating a Series in pandas:

a. **From a List or Array**: You can create a Series from a Python list or NumPy array by passing it as an argument to the pd.Series() function. For example:
import pandas as pd
my_list = [10, 20, 30, 40, 50]
my_series = pd.Series(my_list)

b. **From a Dictionary**: You can create a Series from a Python dictionary where keys will be used as the index labels and values as the data. For example:
import pandas as pd
my_dict = {'A': 10, 'B': 20, 'C': 30}
my_series = pd.Series(my_dict)

c. **From a Scalar Value**: You can create a Series with a fixed length and fill it with a scalar value using the pd.Series() function. Specify the length using the index parameter or by passing the range() function. For example:
import pandas as pd
my_series = pd.Series(5, index=['A', 'B', 'C', 'D'])

d. **From a Range of Values**: You can create a Series with a range of values using the pd.Series() function in combination with the range() function or NumPy's np.arange() function. For example:
import pandas as pd
my_series = pd.Series(range(5))

e. **From a CSV or Text File**: You can create a Series from a CSV or text file using the pd.Series() function along with the pd.read_csv() or pd.read_table() functions. For example:
import pandas as pd
my_series = pd.Series(pd.read_csv('data.csv', header=None, squeeze=True))

These are some of the common methods for creating a Series in pandas. Depending on your specific use case and data source, you may choose the most appropriate method.

5. **How can a dataframe be created in different ways?**

In pandas, there are several ways to create a DataFrame. Here are some common methods for creating a DataFrame in pandas:

a. **From a Dictionary of Lists or Arrays**: You can create a DataFrame from a Python dictionary where each key represents a column name, and the corresponding value is a list or array containing the data for that column. For example:

```
import pandas as pd
data = {'Name': ['John', 'Alice', 'Bob'],
        'Age': [25, 28, 30],
        'City': ['New York', 'Paris', 'London']}
df = pd.DataFrame(data)
```

b. **From a List of Dictionaries**: You can create a DataFrame from a list of dictionaries where each dictionary represents a row of data, and the keys correspond to column names. For example:

```
import pandas as pd
data = [{'Name': 'John', 'Age': 25, 'City': 'New York'},
        {'Name': 'Alice', 'Age': 28, 'City': 'Paris'},
        {'Name': 'Bob', 'Age': 30, 'City': 'London'}]
df = pd.DataFrame(data)
```

c. **From a NumPy Array**: You can create a DataFrame from a NumPy array by passing the array as an argument to the pd.DataFrame() function. You can also specify column names and index labels if needed. For example:

```
import pandas as pd
import numpy as np
data = np.array([[10, 20, 30],
                 [40, 50, 60],
                 [70, 80, 90]])
df = pd.DataFrame(data, columns=['A', 'B', 'C'], index=['X', 'Y', 'Z'])
```

d. **From a CSV or Text File**: You can create a DataFrame from a CSV or text file using the pd.read_csv() or pd.read_table() functions. These functions read the file and return a DataFrame with the data. For example:

```
import pandas as pd
df = pd.read_csv('data.csv')
```

e. **From a SQL Database**: You can create a DataFrame by executing a SQL query on a SQL database and fetching the results into a DataFrame using the pd.read_sql_query() function. You need to establish a database connection before executing the query. For example:

```
import pandas as pd
import sqlite3

conn = sqlite3.connect('database.db')
query = "SELECT * FROM table"
df = pd.read_sql_query(query, conn)
conn.close()
```

These are some of the common methods for creating a DataFrame in pandas. Depending on your data source and requirements, you can choose the most appropriate method to create your DataFrame.

6. Can you provide some examples of statistical functions available in the pandas library for Python?

Here are some examples of statistical functions available in the pandas library for Python:

a. Mean: Calculate the arithmetic mean of a Series or DataFrame using the mean() function. It returns the average value of the data.

```
import pandas as pd
data = pd.Series([10, 20, 30, 40, 50])
mean_value = data.mean()
print(mean_value)
```

b. **Median:** Compute the median value of a Series or DataFrame using the median() function. It returns the middle value of the data when sorted in ascending order.

```
import pandas as pd
data = pd.Series([10, 20, 30, 40, 50])
median_value = data.median()
print(median_value)
```

c. **Mode:** Find the mode(s) of a Series or DataFrame using the mode() function. It returns the most frequently occurring value(s) in the data.

```
import pandas as pd
data = pd.Series([10, 20, 30, 30, 40, 40, 50])
mode_values = data.mode()
print(mode_values)
```

d. **Standard Deviation:** Calculate the standard deviation of a Series or DataFrame using the std() function. It measures the dispersion or variability of the data.

```
import pandas as pd
data = pd.Series([10, 20, 30, 40, 50])
std_value = data.std()
print(std_value)
```

e. **Variance:** Compute the variance of a Series or DataFrame using the var() function. It measures the average squared deviation from the mean.

```
import pandas as pd
data = pd.Series([10, 20, 30, 40, 50])
var_value = data.var()
print(var_value)
```

f. **Correlation:** Calculate the correlation between two Series or multiple columns of a DataFrame using the corr() function. It measures the strength and direction of the linear relationship between variables.

```
import pandas as pd
data1 = pd.Series([10, 20, 30, 40, 50])
data2 = pd.Series([5, 15, 25, 35, 45])
correlation = data1.corr(data2)
print(correlation)
```

These are just a few examples of the statistical functions available in the pandas library. pandas offers a wide range of statistical functions to analyze and summarize data, including min, max, sum, count, quantile, covariance, and more. These functions can be applied to Series or along specific axes (rows or columns) of a DataFrame.

7. How can you sort a dataframe in pandas?

You can sort a datarame in pandas in ways ascending and descending**. Ascending order is set as default sort.**

I. **Ascending order: sorted_df = df.sort_values(by='col name')**
II. **Descending order: sorted_df = df.sort_values(by='col name'', ascending=False)**

8. What is the process of setting an index for a pandas dataframe?

- DataFrame is the tabular structure in the Python pandas library. It represents each row and column by the label. Row label is called an index, whereas column label is called column index/header.
- By default, while creating DataFrame, Python pandas assign a range of numbers (starting at 0) as a row index. Row indexes are used to identify each row. We can set a new row index or replace the existing ones using DataFrame.set_index() function.

9. How do you add a new row to an existing pandas dataframe?

- We can use Dataframe.loc or Dataframe.append method to add a row at the end of Dataframe. If you want to insert a row at any specific position, then you can use Dataframe.insert() method.

10. What is the procedure for adding a new column to a pandas dataframe?

- Sometimes it is required to add a new column in the DataFrame. DataFrame.insert() function is used to insert a new column in DataFrame at the specified position.

11. How can you convert a numpy array into a dataframe using pandas?

- To convert a NumPy array into a DataFrame using pandas, you can use the pandas.DataFrame() constructor.

```
import pandas as pd
import numpy as np
# Creating a NumPy array
numpy_array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Converting the NumPy array to a DataFrame
df = pd.DataFrame(numpy_array)
# Printing the DataFrame
print(df)
```

- In this example, we import the necessary libraries, NumPy and pandas. Then, we create a NumPy array called numpy_array. Finally, we pass the NumPy array to the pd.DataFrame() constructor, which converts it into a DataFrame named df. The resulting DataFrame is then printed.
- You can also customize the column names and index labels by passing additional parameters to the pd.DataFrame() constructor. For example, you can specify column names using the columns parameter and index labels using the index parameter.

12. What is the method to delete a row from a pandas dataframe?

To delete a row from a pandas DataFrame, you can use the drop() method. The drop() method allows you to remove a specific row based on its index label or position. You can drop a feature permanently and temporary also.

1. **Delete temporary - df.drop(['Feature name'],axis = 1)**
2. **Delete permanently - df.drop(['Feature name '],axis = 1,inplace = True)**