Pandas For Data Science - Part 2

Type	Data science masterclass

VI. Data Manipulation and Transformation

Data manipulation and transformation are essential skills for analyzing and preparing data for further processing. Pandas provides powerful tools for indexing, sorting, merging, and reshaping datasets. This section covers:

- **6.1 Indexing and Selecting Data**
- **6.2 Sorting and Ranking**
- **6.3 Merging, Joining, and Concatenating**
- **6.4 Reshaping Data**

6.1 Indexing and Selecting Data

Indexing and selecting data are fundamental operations in Pandas that allow efficient access to subsets of your dataset.

Label-based and Positional Indexing (`.loc` and `.iloc`)

- `loc[]` **– Label-based indexing:**
    - Selects rows and columns by explicit index labels.
    - **Example:**

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'Salary': [50000, 60000, 70000, 80000]}
```

```
df = pd.DataFrame(data, index=['A', 'B', 'C', 'D'])

# Selecting a single row by label
print(df.loc['A'])

# Selecting multiple rows and specific columns
print(df.loc[['A', 'C'], ['Name', 'Salary']])
```

- `iloc[]` **– Positional indexing:**
  - Selects rows and columns based on numerical position (zero-based).
  - **Example:**

```
# Selecting by row index position
print(df.iloc[0])  # First row

# Selecting a range of rows
print(df.iloc[1:3])  # Second and third rows

# Selecting specific rows and columns
print(df.iloc[[0, 2], [1, 2]])  # Rows 0 & 2, Columns 1 & 2
```

Boolean Indexing and Conditional Selection

- **Boolean Indexing:** Select data that meets a condition.
  - **Example:**

```
high_salary = df[df['Salary'] > 60000]
print(high_salary)
```

- **Multiple Conditions with `&` (AND) and `|` (OR):**
  - **Example:**

```
df_filtered = df[(df['Age'] > 25) & (df['Salary'] < 80000)]
print(df_filtered)
```

6.2 Sorting and Ranking

Sorting and ranking allow reordering data based on values or index labels.

Sorting by Index or Values

- **Sorting by Values (`sort_values`)**
    - **Example: Sorting by Salary (ascending and descending)**

    ```
    df_sorted = df.sort_values(by='Salary', ascending=False)
    print(df_sorted)
    ```

- **Sorting by Multiple Columns**
    - **Example: Sort by Age (ascending) and Salary (descending)**

    ```
    df_sorted = df.sort_values(by=['Age', 'Salary'], ascending=[True, False])
    print(df_sorted)
    ```

- **Sorting by Index (`sort_index`)**
    - **Example: Sorting alphabetically by index labels**

    ```
    df_sorted_idx = df.sort_index()
    print(df_sorted_idx)
    ```

Ranking Data Within a DataFrame

- **Ranking assigns a rank to values within a column.**
    - **Example:**

```
df['Salary_rank'] = df['Salary'].rank(ascending=False)
print(df)
```

- **Handling Ties in Ranking (`method` parameter):**
    - `average` : Default, assigns the mean rank for ties.
    - `min` : Assigns the minimum rank for ties.
    - `max` : Assigns the maximum rank for ties.
    - **Example:**

    ```
    df['Salary_rank_min'] = df['Salary'].rank(method='min')
    print(df)
    ```

6.3 Merging, Joining, and Concatenating

Combining datasets is a fundamental task in data manipulation. Pandas provides multiple methods for merging and joining data.

Concatenation and Appending DataFrames

- **Concatenating along Rows (`axis=0`)**
    - **Example:**

    ```
    df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
    df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

    df_concat = pd.concat([df1, df2], axis=0)
    print(df_concat)
    ```

- **Concatenating along Columns (`axis=1`)**
    - **Example:**

```
df_concat = pd.concat([df1, df2], axis=1)
print(df_concat)
```

- **Appending a Single Row (`append()`)**
    - **Example:**

```
df_new = pd.DataFrame({'A': [7], 'B': [9]})
df_updated = df1.append(df_new, ignore_index=True)
print(df_updated)
```

Merge and Join Operations (Inner, Outer, Left, Right)

- **Merging on a Key Column (`merge()`)**
    - **Example:**

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [1, 2, 4], 'Salary': [50000, 60000, 70000]})

df_merged = pd.merge(df1, df2, on='ID', how='inner')
print(df_merged)
```

- **Different Types of Joins:**
    - `inner` : Only matching rows (default).
    - `left` : All rows from the left DataFrame and matching rows from the right.
    - `right` : All rows from the right DataFrame and matching rows from the left.
    - `outer` : All rows from both DataFrames (fills unmatched with NaN).
    - **Example:**

```
df_outer = pd.merge(df1, df2, on='ID', how='outer')
print(df_outer)
```

6.4 Reshaping Data

Reshaping allows transforming data between wide and long formats for better analysis.

Pivot Tables and Cross-tabulations

- **Creating a Pivot Table (`pivot_table()`)**
    - **Example:**

```
df = pd.DataFrame({'Department': ['HR', 'IT', 'HR', 'IT'],
                   'Gender': ['Male', 'Female', 'Female', 'Male'],
                   'Salary': [50000, 70000, 55000, 80000]})

pivot_table = df.pivot_table(values='Salary', index='Department', colu
mns='Gender', aggfunc='mean')
print(pivot_table)
```

- **Cross-tabulation (`pd.crosstab()`)**
    - **Example:**

```
crosstab = pd.crosstab(df['Department'], df['Gender'])
print(crosstab)
```

Melting and Stacking DataFrames

- **Melting (Wide to Long Format)**
    - **Example:**

```
df_melted = pd.melt(df, id_vars=['Department'], value_vars=['Salary'])
print(df_melted)
```

- **Stacking (Convert Columns to Rows)**
    - **Example:**

```
df_stacked = df.set_index(['Department', 'Gender']).stack()
print(df_stacked)
```

VII. Aggregation and Group Operations

Aggregation and grouping are essential techniques for summarizing and analyzing large datasets efficiently. Pandas provides powerful tools to group data, apply aggregation functions, and perform advanced computations.

This section covers:

- **7.1 The GroupBy Mechanism**
- **7.2 Advanced Aggregations**

7.1 The GroupBy Mechanism

The `groupby()` function in Pandas is used to split data into groups based on a column's values and then apply aggregation, transformation, or filtration functions.

Grouping Data and Applying Aggregation Functions

- **Basic Grouping**
  - `groupby()` creates a GroupBy object that can be used for aggregation.
  - **Example:**

```
import pandas as pd

# Sample DataFrame
data = {'Department': ['HR', 'IT', 'HR', 'IT', 'Finance', 'Finance'],
        'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
        'Salary': [50000, 70000, 52000, 80000, 60000, 75000]}
```

```
df = pd.DataFrame(data)

# Group by 'Department' and calculate the average salary
avg_salary = df.groupby('Department')['Salary'].mean()
print(avg_salary)
```

- **Common Aggregation Functions**
    - `mean()`, `sum()`, `count()`, `min()`, `max()`, `std()`, `var()`.
    - **Example:**

    ```
    df_grouped = df.groupby('Department')['Salary'].agg(['mean', 'sum',
    'count'])
    print(df_grouped)
    ```

- **Grouping by Multiple Columns**
    - **Example:**

    ```
    df_grouped = df.groupby(['Department', 'Employee'])['Salary'].sum()
    print(df_grouped)
    ```

Transformation vs. Aggregation vs. Filtration

Pandas `groupby()` supports three main operations:

1. Aggregation (`agg()`)

- Reduces each group to a single value.
- **Example:**

```
df_agg = df.groupby('Department')['Salary'].agg('sum')
print(df_agg)
```

2. Transformation (`transform()`)

- Returns a result of the same shape as the original data.

- Useful for normalization and filling missing values within groups.
- **Example:**

  ```
  df['Salary_mean'] = df.groupby('Department')['Salary'].transform('mean')
  print(df)
  ```

3. Filtration (`filter()`)

- Removes groups based on a condition.
- **Example:**

  ```
  df_filtered = df.groupby('Department').filter(lambda x: x['Salary'].mean() >
  60000)
  print(df_filtered)
  ```

7.2 Advanced Aggregations

Pandas allows for custom aggregation functions and multiple operations on grouped data.

Custom Aggregation Functions

- **Using a Custom Function in** `agg()`
  - Define a custom function and apply it to each group.
  - **Example:**

    ```
    def range_diff(x):
        return x.max() - x.min()

    df_grouped = df.groupby('Department')['Salary'].agg(range_diff)
    print(df_grouped)
    ```

- **Lambda Functions Inside** `agg()`
  - **Example:**

```
df_grouped = df.groupby('Department')['Salary'].agg(lambda x: x.max
() - x.min())
print(df_grouped)
```

Working with Multiple Aggregation Operations

- **Applying Multiple Functions to a Column**
    - Use a list of functions inside `agg()`.
    - **Example:**

    ```
    df_grouped = df.groupby('Department')['Salary'].agg(['mean', 'sum',
    'min', 'max'])
    print(df_grouped)
    ```

- **Applying Different Functions to Different Columns**
    - Use a dictionary inside `agg()`.
    - **Example:**

    ```
    df_grouped = df.groupby('Department').agg({'Salary': ['mean', 'sum'],
    'Employee': 'count'})
    print(df_grouped)
    ```

Grouping and Sorting

- **Sorting After Grouping**
    - **Example:**

    ```
    df_grouped = df.groupby('Department')['Salary'].mean().sort_values(a
    scending=False)
    print(df_grouped)
    ```

- **Resetting Index After Grouping**
    - **Example:**

```
df_grouped = df.groupby('Department')['Salary'].mean().reset_index()
print(df_grouped)
```

VIII. Time Series Analysis

Time series data is a sequence of data points recorded at specific time intervals. Pandas provides robust tools for working with time series data, including time-based indexing, resampling, and visualization.

This section covers:

- **8.1 Handling Time Series Data**
- **8.2 Resampling and Frequency Conversion**
- **8.3 Time Series Visualization and Analysis**

8.1 Handling Time Series Data

Date and Time Indexing

Pandas allows setting timestamps as index values, enabling easy selection, filtering, and analysis.

- **Creating a DateTime Index from a Column**

```
import pandas as pd

data = {'Date': ['2023-01-01', '2023-02-01', '2023-03-01'],
        'Sales': [200, 250, 300]}

df = pd.DataFrame(data)

# Convert 'Date' column to DateTime and set as index
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
```

```
print(df)
```

- **Generating a Range of Dates (`date_range`)**

```
date_range = pd.date_range(start='2023-01-01', periods=6, freq='M')
print(date_range)
```

- **Selecting Data by Date Index**

```
# Select data for a specific date
print(df.loc['2023-02-01'])

# Select data for a date range
print(df.loc['2023-01-01':'2023-03-01'])
```

Parsing Dates and Time Zones

- **Parsing Date Formats Automatically**

```
df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')
```

- **Handling Different Time Zones**

```
df.index = df.index.tz_localize('UTC')  # Set to UTC
df.index = df.index.tz_convert('Asia/Kolkata')  # Convert to IST
print(df)
```

8.2 Resampling and Frequency Conversion

Upsampling and Downsampling Techniques

- **Resampling Changes the Frequency of Time Series Data**
    - **Downsampling (Reducing Frequency)**

```
df_resampled = df.resample('M').sum()  # Monthly total sales
print(df_resampled)
```

- **Upsampling (Increasing Frequency with Interpolation)**

```
df_upsampled = df.resample('D').interpolate()  # Daily frequency with i
nterpolation
print(df_upsampled)
```

Rolling Statistics and Window Functions

- **Rolling Mean (Moving Average)**

```
df['Rolling_Mean'] = df['Sales'].rolling(window=2).mean()
print(df)
```

- **Expanding Window (Cumulative Sum)**

```
df['Cumulative_Sum'] = df['Sales'].expanding().sum()
print(df)
```

8.3 Time Series Visualization and Analysis

Plotting Trends Over Time

- **Simple Line Plot of Time Series Data**

```
import matplotlib.pyplot as plt

df.plot(y='Sales', title='Sales Trend Over Time')
plt.show()
```

- **Rolling Mean Visualization**

```
df['Rolling_Mean'].plot(label='Rolling Mean', legend=True)
df['Sales'].plot(label='Original Sales', legend=True)
plt.show()
```

Seasonal Decomposition

- **Decomposing Time Series into Trend, Seasonal, and Residual Components**

```
from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(df['Sales'], model='additive', period=2)
result.plot()
plt.show()
```

IX. Performance Optimization

Pandas is a powerful library, but working with large datasets can lead to slow performance and high memory usage. Optimizing computations and memory usage is crucial for handling big data efficiently.

This section covers:

- **9.1 Vectorization and Efficient Computation**
- **9.2 Memory Management**
- **9.3 Working with Large Datasets**

9.1 Vectorization and Efficient Computation

Vectorization speeds up computations by applying operations to entire arrays instead of iterating through elements with loops. Pandas leverages NumPy's optimized array computations for efficiency.

Avoiding Python Loops: Leveraging Pandas and NumPy Operations

- **Inefficient Python Loop:**

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

df = pd.DataFrame({'A': np.random.randint(1, 100, 10)})

# Using a loop (slow)
start = time.time()
df['B'] = [x * 2 for x in df['A']]
print("Loop Time:", time.time() - start)
```

- **Efficient Vectorized Operation:**

```
# Using Pandas vectorized operation (fast)
start = time.time()
df['B'] = df['A'] * 2
print("Vectorized Time:", time.time() - start)
```

Using `.apply()` , `.map()` , and `.applymap()` Wisely

Using `.apply()` for Column-wise Operations

- **Slow Loop-Based Approach:**

```
def square(x):
    return x ** 2

df['Square_Loop'] = [square(x) for x in df['A']]
```

- **Faster `apply()` Approach:**

```
df['Square_Apply'] = df['A'].apply(square)
```

Using `.map()` for Element-wise Operations in Series

- **Example:**

```
df['Mapped'] = df['A'].map(lambda x: x ** 2)
```

Using `.applymap()` for Element-wise Operations in DataFrames

- **Example:**

```
df = pd.DataFrame(np.random.randint(1, 10, (3, 3)), columns=['A', 'B', 'C'])
df_squared = df.applymap(lambda x: x ** 2)
```

✅ **Best Practice:** Use vectorized operations whenever possible instead of `.apply()` or `.map()` for performance gains.

9.2 Memory Management

Large datasets consume a lot of memory, which can slow down operations. Optimizing data types reduces memory usage significantly.

Optimizing Data Types for Large Datasets

- **Checking Data Types and Memory Usage:**

```
print(df.info(memory_usage='deep'))
```

- **Optimizing Integer and Float Data Types:**

```
df['A'] = df['A'].astype('int16')  # Use 'int16' instead of 'int64'
df['C'] = df['C'].astype('float32')  # Use 'float32' instead of 'float64'
```

- **Converting Strings to Categorical Type (Huge Memory Savings):**

```
df['Category'] = df['Category'].astype('category')
```

Best Practices for Handling Big Data with Pandas

- **Use `astype()` to downcast data types** to smaller types (`int8`, `float32`).
- **Drop unnecessary columns** to reduce memory footprint.
- **Convert string columns to categorical** to save memory.
- **Use chunking when reading large files** (e.g., `pd.read_csv(..., chunksize=10000)`).

9.3 Working with Large Datasets

When datasets become too large to fit into memory, alternative techniques are required.

Techniques for Out-of-Core Computation

- **Using `chunksize` for Processing Large CSV Files**

```
chunk_iter = pd.read_csv('large_data.csv', chunksize=10000)
for chunk in chunk_iter:
    print(chunk.shape)
```

- **Processing Data in Chunks and Aggregating Results**

```
total_sum = 0
for chunk in chunk_iter:
    total_sum += chunk['column_name'].sum()
print("Total Sum:", total_sum)
```

Introduction to Dask or Other Scalable Libraries

Pandas loads data into memory, but **Dask** is an alternative that allows working with datasets larger than RAM.

- **Installing Dask:**

```
pip install dask
```

- **Using Dask Instead of Pandas for Large DataFrames**

```
import dask.dataframe as dd

df_large = dd.read_csv('large_data.csv')
print(df_large.head())  # Works like Pandas but loads data lazily
```

✅ **Best Practice:** Use **Dask** when handling large datasets that do not fit into memory.

X. Advanced DataFrame Operations

Pandas provides advanced functionality for handling complex data structures like multi-index DataFrames and performance tuning. These techniques allow for better data organization, efficient lookups, and optimized computations.

This section covers:

- **Working with Multi-Index DataFrames**
- **Customizing DataFrame Display and Performance Tuning**

Working with Multi-Index DataFrames

Multi-indexing allows for more complex and hierarchical data structures by using multiple levels of row and/or column indices.

Creating a Multi-Index DataFrame

1. Using `set_index()` with Multiple Columns

- Convert multiple columns into a hierarchical index.

```
import pandas as pd

data = {
    'Country': ['USA', 'USA', 'Canada', 'Canada'],
    'City': ['New York', 'Los Angeles', 'Toronto', 'Vancouver'],
    'Population': [8419600, 3980400, 2930000, 631490]
}

df = pd.DataFrame(data)
df.set_index(['Country', 'City'], inplace=True)  # Set multi-index
print(df)
```

2. Creating Multi-Index Using `pd.MultiIndex.from_tuples()`

- Explicitly define a hierarchical index.

```
index = pd.MultiIndex.from_tuples([
    ('USA', 'New York'),
    ('USA', 'Los Angeles'),
    ('Canada', 'Toronto'),
    ('Canada', 'Vancouver')
], names=['Country', 'City'])

df = pd.DataFrame({'Population': [8419600, 3980400, 2930000, 631490]}, index=index)
print(df)
```

Selecting and Filtering Data in Multi-Index DataFrames

- **Accessing Data for a Specific Index Level**

```
print(df.loc['USA'])  # Select all cities in the USA
print(df.loc[('USA', 'New York')])  # Select New York data
```

- **Filtering by Specific Index Levels**

```
print(df.xs('USA', level='Country'))  # Get all cities in USA
```

- **Selecting Data Using `slice`**

```
from pandas import IndexSlice
print(df.loc[IndexSlice[:, 'New York'], :])  # Get all data for New York
```

Resetting and Reordering Multi-Index

- **Reset Multi-Index to Columns (`reset_index()`)**

```
df_reset = df.reset_index()
print(df_reset)
```

- **Swapping Index Levels (`swaplevel()`)**

```
df_swapped = df.swaplevel()
print(df_swapped)
```

- **Sorting Multi-Index (`sort_index()`)**

```
df_sorted = df.sort_index(level=['Country', 'City'])
print(df_sorted)
```

Customizing DataFrame Display and Performance Tuning

Customizing DataFrame Display

- **Changing Display Options in Pandas**

```
pd.set_option('display.max_rows', 50)  # Set max rows displayed
pd.set_option('display.max_columns', 20)  # Set max columns displayed
pd.set_option('display.float_format', '{:.2f}'.format)  # Set float precision
```

- **Displaying Large DataFrames Using `to_string()`**

```
print(df.to_string())  # Display entire DataFrame
```

- **Styling DataFrames for Better Readability (`style`)**

```
df.style.set_properties(**{'background-color': 'lightgray', 'color': 'black'})
```

Performance Tuning for Large DataFrames

- **Convert Data Types to Reduce Memory Usage**

```
df['Population'] = df['Population'].astype('int32')  # Convert to smaller dat
a type
print(df.info(memory_usage='deep'))
```

- **Using `query()` for Fast Filtering**

```
df_filtered = df.query("Population > 3000000")
print(df_filtered)
```

- **Optimizing `apply()` with Vectorized Operations**

```
df['Population_Squared'] = df['Population'] ** 2  # Vectorized operation
```
