# Pandas

## GPT Link

<https://chatgpt.com/share/29ef99ca-66db-417e-a9c1-2ab03205b348>

## Part 1 - Questions (gpt 4)

### Question 1

Absolutely, let's make the debugging questions more advanced by incorporating concepts like data cleaning, handling missing values, and optimizing performance.

0 - Real-life Application of Pandas: Customer Data Analysis

1 - Scenario:

You are working on a project where you need to analyze customer data from an e-commerce platform. Your task involves cleaning the data, handling missing values, and performing advanced analysis to identify customer trends and behavior.

2 - Buggy Code:

```python

import pandas as pd

# Load customer data

df = pd.read\_csv('customer\_data.csv')

# Remove rows with missing values

df\_cleaned = df.dropna()

# Convert 'Registration Date' column to datetime

df\_cleaned['Registration Date'] = pd.to\_datetime(df\_cleaned['Registration Date'])

# Extract year from 'Registration Date' column

df\_cleaned['Registration Year'] = df\_cleaned['Registration Date'].dt.year

# Calculate the number of new registrations per year

new\_registrations\_per\_year = df\_cleaned.groupby('Registration Year').count()

# Save the result to a new CSV file

new\_registrations\_per\_year.to\_csv('new\_registrations\_per\_year.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to analyze new registrations per year but misses handling missing values properly and does not optimize the aggregation process for performance.

5 - Answer:

The corrected code should handle missing values, optimize the aggregation process, and use a more appropriate method for counting new registrations per year.

```python

import pandas as pd

# Load customer data

df = pd.read\_csv('customer\_data.csv')

# Drop rows with missing values in the 'Registration Date' column

df\_cleaned = df.dropna(subset=['Registration Date'])

# Convert 'Registration Date' column to datetime

df\_cleaned['Registration Date'] = pd.to\_datetime(df\_cleaned['Registration Date'])

# Extract year from 'Registration Date' column

df\_cleaned['Registration Year'] = df\_cleaned['Registration Date'].dt.year

# Count new registrations per year

new\_registrations\_per\_year = df\_cleaned.groupby('Registration Year').size().reset\_index(name='New Registrations')

# Save the result to a new CSV file

new\_registrations\_per\_year.to\_csv('new\_registrations\_per\_year.csv', index=False)

```

In the corrected code, we handle missing values by dropping rows with missing registration dates. We then use the `size` method to count the number of new registrations per year, which is a more efficient approach. Additionally, we reset the index and specify `index=False` while saving the result to a CSV file.

### Question 2

Certainly! Here's another scenario that tests advanced concepts in Pandas:

0 - Real-life Application of Pandas: Web Traffic Analysis

1 - Scenario:

You are working on a project where you need to analyze web traffic data from a popular website. Your task involves cleaning and preprocessing the data, identifying patterns in user behavior, and generating insights to optimize website performance.

2 - Buggy Code:

```python

import pandas as pd

# Load web traffic data

df = pd.read\_csv('web\_traffic\_data.csv')

# Convert 'Timestamp' column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Remove duplicate entries

df.drop\_duplicates()

# Calculate total page views per day

daily\_page\_views = df.groupby(df['Timestamp'].dt.date)['Page Views'].sum()

# Calculate average time spent on the website per day

df['Time Spent'] = pd.to\_timedelta(df['Time Spent'])

average\_time\_spent = df.groupby(df['Timestamp'].dt.date)['Time Spent'].mean()

# Merge the two DataFrames

daily\_stats = pd.concat([daily\_page\_views, average\_time\_spent], axis=1, keys=['Page Views', 'Average Time Spent'])

# Save the result to a new CSV file

daily\_stats.to\_csv('daily\_stats.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to analyze web traffic data but has issues with removing duplicates, calculating total page views, and computing the average time spent on the website per day.

5 - Answer:

The corrected code should properly handle duplicate entries, optimize the aggregation process for performance, and compute the average time spent on the website per day accurately.

```python

import pandas as pd

# Load web traffic data

df = pd.read\_csv('web\_traffic\_data.csv')

# Convert 'Timestamp' column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Remove duplicate entries

df.drop\_duplicates(inplace=True)

# Calculate total page views per day

daily\_page\_views = df.groupby(df['Timestamp'].dt.date)['Page Views'].sum()

# Calculate average time spent on the website per day

df['Time Spent'] = pd.to\_timedelta(df['Time Spent'])

average\_time\_spent = df.groupby(df['Timestamp'].dt.date)['Time Spent'].sum() / df.groupby(df['Timestamp'].dt.date)['Page Views'].count()

# Merge the two DataFrames

daily\_stats = pd.concat([daily\_page\_views, average\_time\_spent], axis=1, keys=['Page Views', 'Average Time Spent'])

# Save the result to a new CSV file

daily\_stats.to\_csv('daily\_stats.csv')

```

In the corrected code, we handle duplicate entries by using `inplace=True` in the `drop\_duplicates` method. We accurately compute the average time spent on the website per day by dividing the total time spent by the number of page views. Additionally, we use the `sum()` function to aggregate time spent on the website per day.

### Question 3

Certainly! Here's another scenario that tests advanced concepts in Pandas:

0 - Real-life Application of Pandas: Market Basket Analysis

1 - Scenario:

You are working on a project where you need to analyze transaction data from a retail store to perform market basket analysis. Your task involves preprocessing the data, identifying frequent itemsets, and generating association rules to understand purchasing patterns.

2 - Buggy Code:

```python

import pandas as pd

from mlxtend.frequent\_patterns import apriori

from mlxtend.frequent\_patterns import association\_rules

# Load transaction data

df = pd.read\_csv('transaction\_data.csv')

# Convert data to transaction format

transactions = df.groupby('TransactionID')['Item'].apply(list)

# One-hot encode the transaction data

one\_hot\_encoded = pd.get\_dummies(transactions.apply(pd.Series).stack()).sum(level=0)

# Find frequent itemsets using Apriori algorithm

frequent\_itemsets = apriori(one\_hot\_encoded, min\_support=0.1, use\_colnames=True)

# Generate association rules

rules = association\_rules(frequent\_itemsets, metric='lift', min\_threshold=1.0)

# Save the association rules to a new CSV file

rules.to\_csv('association\_rules.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to perform market basket analysis but has issues with preprocessing the transaction data, one-hot encoding, and generating association rules.

5 - Answer:

The corrected code should properly preprocess the transaction data, perform one-hot encoding, and generate association rules accurately.

```python

import pandas as pd

from mlxtend.frequent\_patterns import apriori

from mlxtend.frequent\_patterns import association\_rules

# Load transaction data

df = pd.read\_csv('transaction\_data.csv')

# Convert data to transaction format

transactions = df.groupby('TransactionID')['Item'].apply(list)

# Create a list of unique items

items = list(set(df['Item']))

# Create an empty DataFrame to store one-hot encoded transaction data

one\_hot\_encoded = pd.DataFrame(0, index=transactions.index, columns=items)

# Update the one-hot encoded DataFrame based on transactions

for index, transaction in transactions.iteritems():

one\_hot\_encoded.loc[index, transaction] = 1

# Find frequent itemsets using Apriori algorithm

frequent\_itemsets = apriori(one\_hot\_encoded, min\_support=0.1, use\_colnames=True)

# Generate association rules

rules = association\_rules(frequent\_itemsets, metric='lift', min\_threshold=1.0)

# Save the association rules to a new CSV file

rules.to\_csv('association\_rules.csv', index=False)

```

In the corrected code, we properly preprocess the transaction data and create an empty DataFrame to store one-hot encoded transaction data. We then update the one-hot encoded DataFrame based on transactions and generate association rules accurately. Additionally, we specify `index=False` while saving the association rules to a CSV file.

### Question 4

Certainly! Here's another scenario that tests advanced concepts in Pandas:

0 - Real-life Application of Pandas: Sensor Data Analysis

1 - Scenario:

You are working on a project where you need to analyze sensor data collected from various IoT devices deployed in a manufacturing plant. Your task involves preprocessing the sensor data, identifying anomalies, and generating insights to optimize manufacturing processes.

2 - Buggy Code:

```python

import pandas as pd

# Load sensor data

df = pd.read\_csv('sensor\_data.csv')

# Drop rows with missing values

df.dropna(inplace=True)

# Convert 'Timestamp' column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Calculate rolling average temperature for a 10-minute window

df['Rolling\_Avg\_Temperature'] = df['Temperature'].rolling(window='10T').mean()

# Identify anomalies in temperature

df['Anomaly'] = df['Temperature'] > df['Rolling\_Avg\_Temperature'] + 2 \* df['Temperature'].std()

# Save the result to a new CSV file

df.to\_csv('processed\_sensor\_data.csv', index=False)

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to preprocess sensor data and identify anomalies but has issues with handling missing values, calculating rolling average temperature, and identifying anomalies using statistical thresholds.

5 - Answer:

The corrected code should handle missing values properly, calculate the rolling average temperature accurately, and identify anomalies based on appropriate statistical thresholds.

```python

import pandas as pd

# Load sensor data

df = pd.read\_csv('sensor\_data.csv')

# Fill missing values with the mean temperature

df['Temperature'].fillna(df['Temperature'].mean(), inplace=True)

# Convert 'Timestamp' column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Sort DataFrame by timestamp

df.sort\_values(by='Timestamp', inplace=True)

# Calculate rolling average temperature for a 10-minute window

df['Rolling\_Avg\_Temperature'] = df['Temperature'].rolling(window='10T', min\_periods=1).mean()

# Identify anomalies in temperature

df['Anomaly'] = df['Temperature'] > (df['Rolling\_Avg\_Temperature'] + 2 \* df['Temperature'].std())

# Save the result to a new CSV file

df.to\_csv('processed\_sensor\_data.csv', index=False)

```

In the corrected code, we fill missing values with the mean temperature to handle missing data. We sort the DataFrame by timestamp to ensure the rolling average calculation is performed correctly. We use `min\_periods=1` to specify the minimum number of periods required to compute the rolling average. Additionally, we identify anomalies based on appropriate statistical thresholds by using the rolling average temperature and the standard deviation of temperature readings.

### Question 5

Certainly! Here's another scenario that tests advanced concepts in Pandas:

0 - Real-life Application of Pandas: Social Media Engagement Analysis

1 - Scenario:

You are working on a project where you need to analyze engagement metrics from social media posts to understand user interaction and optimize content strategy. Your task involves preprocessing the data, calculating engagement rates, and identifying trends in user engagement.

2 - Buggy Code:

```python

import pandas as pd

# Load social media data

df = pd.read\_csv('social\_media\_data.csv')

# Convert 'Date' column to datetime

df['Date'] = pd.to\_datetime(df['Date'])

# Calculate engagement rate

df['Engagement Rate'] = df['Likes'] / df['Impressions'] \* 100

# Group by month and calculate average engagement rate

monthly\_engagement = df.groupby(df['Date'].dt.month)['Engagement Rate'].mean()

# Save the result to a new CSV file

monthly\_engagement.to\_csv('monthly\_engagement.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to analyze social media engagement but has issues with aggregating data by month and calculating the average engagement rate.

5 - Answer:

The corrected code should properly aggregate the data by month and calculate the average engagement rate accurately.

```python

import pandas as pd

# Load social media data

df = pd.read\_csv('social\_media\_data.csv')

# Convert 'Date' column to datetime

df['Date'] = pd.to\_datetime(df['Date'])

# Calculate engagement rate

df['Engagement Rate'] = df['Likes'] / df['Impressions'] \* 100

# Group by month and calculate average engagement rate

monthly\_engagement = df.groupby(df['Date'].dt.month)['Engagement Rate'].mean()

# Convert the result to DataFrame

monthly\_engagement\_df = monthly\_engagement.reset\_index(name='Average Engagement Rate')

# Save the result to a new CSV file

monthly\_engagement\_df.to\_csv('monthly\_engagement.csv', index=False)

```

In the corrected code, we properly aggregate the data by month using `groupby` and calculate the average engagement rate. We reset the index and specify `index=False` while saving the result to a CSV file to avoid saving the index column.

### Question 6

Certainly! Here's another scenario that tests advanced concepts in Pandas:

0 - Real-life Application of Pandas: Customer Churn Prediction

1 - Scenario:

You are working on a project where you need to analyze customer data to predict churn for a subscription-based service. Your task involves preprocessing the data, feature engineering, building a predictive model, and evaluating model performance.

2 - Buggy Code:

```python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

# Load customer data

df = pd.read\_csv('customer\_data.csv')

# Drop rows with missing values

df.dropna(inplace=True)

# Convert categorical variables to dummy variables

df = pd.get\_dummies(df, columns=['Gender', 'Subscription Plan'], drop\_first=True)

# Split the data into features and target variable

X = df.drop('Churn', axis=1)

y = df['Churn']

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train a random forest classifier

clf = RandomForestClassifier()

clf.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = clf.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to predict customer churn using a random forest classifier but has issues with handling categorical variables, splitting the data, and evaluating model performance.

5 - Answer:

The corrected code should properly encode categorical variables, split the data into training and test sets, and evaluate model performance using appropriate metrics.

```python

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

# Load customer data

df = pd.read\_csv('customer\_data.csv')

# Drop rows with missing values

df.dropna(inplace=True)

# Convert categorical variables to dummy variables

df = pd.get\_dummies(df, columns=['Gender', 'Subscription Plan'], drop\_first=True)

# Split the data into features and target variable

X = df.drop('Churn', axis=1)

y = df['Churn']

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize and train a random forest classifier

clf = RandomForestClassifier()

clf.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = clf.predict(X\_test)

# Calculate accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

```

In the corrected code, we properly encode categorical variables using one-hot encoding to ensure compatibility with the random forest classifier. We split the data into training and test sets using `train\_test\_split` and evaluate model performance using accuracy as a metric.

### Question 7

Got it! Let's focus on bugs that are solely based on Pandas concepts:

0 - Real-life Application of Pandas: Retail Sales Analysis

1 - Scenario:

You are analyzing retail sales data to identify trends and patterns in customer purchasing behavior. Your task involves preprocessing the data, aggregating sales by different categories, and generating insights for business decisions.

2 - Buggy Code:

```python

import pandas as pd

# Load sales data

df = pd.read\_csv('sales\_data.csv')

# Convert 'Date' column to datetime

df['Date'] = pd.to\_datetime(df['Date'])

# Filter data for the current year

df\_current\_year = df[df['Date'].dt.year == 2023]

# Group by month and calculate total sales

monthly\_sales = df\_current\_year.groupby(df\_current\_year['Date'].dt.month)['Sales'].sum()

# Save the result to a new CSV file

monthly\_sales.to\_csv('monthly\_sales.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to aggregate monthly sales for the current year but does not handle missing months properly, leading to incomplete or incorrect results.

5 - Answer:

The corrected code should ensure that all months are included in the aggregation, even if there are no sales recorded for some months.

```python

import pandas as pd

# Load sales data

df = pd.read\_csv('sales\_data.csv')

# Convert 'Date' column to datetime

df['Date'] = pd.to\_datetime(df['Date'])

# Filter data for the current year

df\_current\_year = df[df['Date'].dt.year == 2023]

# Group by month and calculate total sales

monthly\_sales = df\_current\_year.groupby(df\_current\_year['Date'].dt.month)['Sales'].sum()

# Reindex to include all months

monthly\_sales = monthly\_sales.reindex(range(1, 13), fill\_value=0)

# Save the result to a new CSV file

monthly\_sales.to\_csv('monthly\_sales.csv')

```

In the corrected code, after aggregating monthly sales, we reindex the result to include all months from January to December using `reindex` and fill any missing months with 0 sales using `fill\_value=0`. This ensures that the output includes sales data for all months of the year.

### Question 8

Certainly! Here's another scenario that tests advanced Pandas concepts:

0 - Real-life Application of Pandas: Employee Performance Analysis

1 - Scenario:

You are working on a project where you need to analyze employee performance data to evaluate productivity and identify high performers. Your task involves data cleaning, aggregating performance metrics, and generating insights for management.

2 - Buggy Code:

```python

import pandas as pd

# Load employee performance data

df = pd.read\_csv('employee\_performance.csv')

# Fill missing values in the 'Performance Score' column with the mean

df['Performance Score'].fillna(df['Performance Score'].mean(), inplace=True)

# Convert 'Review Date' column to datetime

df['Review Date'] = pd.to\_datetime(df['Review Date'])

# Extract the year and month from 'Review Date'

df['Year'] = df['Review Date'].dt.year

df['Month'] = df['Review Date'].dt.month

# Group by employee and calculate the average performance score per month

monthly\_performance = df.groupby(['Employee ID', 'Year', 'Month'])['Performance Score'].mean()

# Pivot the table to have employees as rows and months as columns

pivot\_table = monthly\_performance.pivot(index='Employee ID', columns=['Year', 'Month'], values='Performance Score')

# Save the result to a new CSV file

pivot\_table.to\_csv('monthly\_performance.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to pivot the table to have employees as rows and months as columns, but it misses correctly resetting the index and properly handling the multi-index resulting from the `groupby` operation.

5 - Answer:

The corrected code should properly reset the index after the groupby operation and handle the multi-index appropriately before pivoting the table.

```python

import pandas as pd

# Load employee performance data

df = pd.read\_csv('employee\_performance.csv')

# Fill missing values in the 'Performance Score' column with the mean

df['Performance Score'].fillna(df['Performance Score'].mean(), inplace=True)

# Convert 'Review Date' column to datetime

df['Review Date'] = pd.to\_datetime(df['Review Date'])

# Extract the year and month from 'Review Date'

df['Year'] = df['Review Date'].dt.year

df['Month'] = df['Review Date'].dt.month

# Group by employee and calculate the average performance score per month

monthly\_performance = df.groupby(['Employee ID', 'Year', 'Month'])['Performance Score'].mean().reset\_index()

# Pivot the table to have employees as rows and months as columns

pivot\_table = monthly\_performance.pivot\_table(index='Employee ID', columns=['Year', 'Month'], values='Performance Score')

# Save the result to a new CSV file

pivot\_table.to\_csv('monthly\_performance.csv')

```

In the corrected code, after grouping by employee and calculating the average performance score per month, we use `reset\_index()` to flatten the multi-index. This ensures that the `pivot\_table` method works correctly by properly restructuring the DataFrame into the desired format with employees as rows and months as columns.

### Question 9

Certainly! Here's a scenario that tests advanced Pandas concepts, focusing on data merging, handling time series data, and complex groupby operations:

0 - Real-life Application of Pandas: Energy Consumption Analysis

1 - Scenario:

You are working on a project where you need to analyze energy consumption data from multiple sources to understand usage patterns and identify potential areas for energy savings. Your task involves merging datasets, handling time series data, and performing advanced aggregations.

2 - Buggy Code:

```python

import pandas as pd

# Load energy consumption data from two different sources

df1 = pd.read\_csv('energy\_data\_source1.csv')

df2 = pd.read\_csv('energy\_data\_source2.csv')

# Convert 'Timestamp' columns to datetime

df1['Timestamp'] = pd.to\_datetime(df1['Timestamp'])

df2['Timestamp'] = pd.to\_datetime(df2['Timestamp'])

# Merge the two datasets on 'Timestamp'

merged\_df = pd.merge(df1, df2, on='Timestamp')

# Fill missing values with the forward fill method

merged\_df.fillna(method='ffill', inplace=True)

# Calculate the total energy consumption by day

merged\_df['Date'] = merged\_df['Timestamp'].dt.date

daily\_consumption = merged\_df.groupby('Date')['Consumption'].sum()

# Identify peak consumption hours

merged\_df['Hour'] = merged\_df['Timestamp'].dt.hour

peak\_hours = merged\_df.groupby('Hour')['Consumption'].sum().sort\_values(ascending=False).head(5)

# Save the results to new CSV files

daily\_consumption.to\_csv('daily\_consumption.csv')

peak\_hours.to\_csv('peak\_hours.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to merge energy consumption data from two different sources and perform advanced aggregations. However, it has issues with handling overlapping columns, accurately aggregating consumption data, and properly dealing with time series data.

5 - Answer:

The corrected code should handle overlapping columns appropriately during the merge, ensure accurate aggregation of consumption data, and handle time series data correctly.

```python

import pandas as pd

# Load energy consumption data from two different sources

df1 = pd.read\_csv('energy\_data\_source1.csv')

df2 = pd.read\_csv('energy\_data\_source2.csv')

# Convert 'Timestamp' columns to datetime

df1['Timestamp'] = pd.to\_datetime(df1['Timestamp'])

df2['Timestamp'] = pd.to\_datetime(df2['Timestamp'])

# Rename columns to avoid overlapping during merge

df1.rename(columns={'Consumption': 'Consumption\_source1'}, inplace=True)

df2.rename(columns={'Consumption': 'Consumption\_source2'}, inplace=True)

# Merge the two datasets on 'Timestamp'

merged\_df = pd.merge(df1, df2, on='Timestamp', how='outer')

# Fill missing values with the forward fill method

merged\_df.fillna(method='ffill', inplace=True)

# Calculate the total energy consumption by adding both sources

merged\_df['Total\_Consumption'] = merged\_df['Consumption\_source1'] + merged\_df['Consumption\_source2']

# Calculate the total energy consumption by day

merged\_df['Date'] = merged\_df['Timestamp'].dt.date

daily\_consumption = merged\_df.groupby('Date')['Total\_Consumption'].sum().reset\_index()

# Identify peak consumption hours

merged\_df['Hour'] = merged\_df['Timestamp'].dt.hour

peak\_hours = merged\_df.groupby('Hour')['Total\_Consumption'].sum().sort\_values(ascending=False).head(5).reset\_index()

# Save the results to new CSV files

daily\_consumption.to\_csv('daily\_consumption.csv', index=False)

peak\_hours.to\_csv('peak\_hours.csv', index=False)

```

In the corrected code, we rename overlapping columns before merging to avoid conflicts. We ensure that missing values are handled using the forward fill method. We calculate the total energy consumption by summing the consumption from both sources. We also properly reset the index when saving the results to CSV files to ensure the data is accurately recorded.

### Question 10

Certainly! Here's a more complex scenario that tests advanced Pandas concepts such as multi-level indexing, handling missing data, and pivot operations:

0 - Real-life Application of Pandas: Sales Performance Analysis

1 - Scenario:

You are analyzing the sales performance of different products across multiple regions over several years. Your task involves cleaning the data, handling missing values, creating multi-level indexes, and generating pivot tables to summarize the sales data.

2 - Buggy Code:

```python

import pandas as pd

# Load sales data

df = pd.read\_csv('sales\_data.csv')

# Convert 'Date' column to datetime

df['Date'] = pd.to\_datetime(df['Date'])

# Extract year and month from 'Date'

df['Year'] = df['Date'].dt.year

df['Month'] = df['Date'].dt.month

# Fill missing values with zero

df.fillna(0, inplace=True)

# Set multi-level index using 'Region' and 'Product'

df.set\_index(['Region', 'Product'], inplace=True)

# Calculate total sales by year and month

total\_sales = df.groupby(['Year', 'Month'])['Sales'].sum().unstack()

# Create pivot table to show sales by region and product

pivot\_table = df.pivot(index='Year', columns='Region', values='Sales')

# Save the results to new CSV files

total\_sales.to\_csv('total\_sales.csv')

pivot\_table.to\_csv('pivot\_table.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to analyze sales performance by handling missing data, creating a multi-level index, and generating pivot tables. However, it has issues with the multi-level index, incorrect grouping for total sales, and the pivot table creation.

5 - Answer:

The corrected code should properly set the multi-level index, correctly group data for total sales, and accurately create the pivot table.

```python

import pandas as pd

# Load sales data

df = pd.read\_csv('sales\_data.csv')

# Convert 'Date' column to datetime

df['Date'] = pd.to\_datetime(df['Date'])

# Extract year and month from 'Date'

df['Year'] = df['Date'].dt.year

df['Month'] = df['Date'].dt.month

# Fill missing values with zero for 'Sales' column only

df['Sales'].fillna(0, inplace=True)

# Set multi-level index using 'Region' and 'Product'

df.set\_index(['Region', 'Product'], inplace=True)

# Calculate total sales by year and month

total\_sales = df.groupby(['Year', 'Month', 'Region', 'Product'])['Sales'].sum().unstack(level=[2, 3])

# Create pivot table to show sales by region and product

pivot\_table = df.reset\_index().pivot\_table(index='Year', columns=['Region', 'Product'], values='Sales', aggfunc='sum')

# Save the results to new CSV files

total\_sales.to\_csv('total\_sales.csv')

pivot\_table.to\_csv('pivot\_table.csv')

```

In the corrected code, we fill missing values specifically for the 'Sales' column. We use a multi-level index properly and ensure that the grouping for total sales includes all relevant levels. We also correctly create the pivot table by resetting the index and using the `pivot\_table` method with appropriate aggregation functions. This ensures accurate and comprehensive sales performance analysis.

### Question 11

Sure! Here’s another scenario focusing on advanced Pandas concepts such as merging, resampling time series data, and handling hierarchical indexing:

0 - Real-life Application of Pandas: IoT Sensor Data Analysis

1 - Scenario:

You are working on a project where you need to analyze IoT sensor data collected from multiple devices across various locations. Your task involves cleaning the data, merging datasets, resampling the time series data, and summarizing the data to identify trends and anomalies.

2 - Buggy Code:

```python

import pandas as pd

# Load sensor data from multiple files

df1 = pd.read\_csv('sensor\_data\_1.csv')

df2 = pd.read\_csv('sensor\_data\_2.csv')

# Convert 'Timestamp' column to datetime

df1['Timestamp'] = pd.to\_datetime(df1['Timestamp'])

df2['Timestamp'] = pd.to\_datetime(df2['Timestamp'])

# Merge the dataframes on 'Timestamp' and 'Device ID'

df\_merged = pd.merge(df1, df2, on=['Timestamp', 'Device ID'])

# Fill missing values with forward fill method

df\_merged.fillna(method='ffill', inplace=True)

# Set multi-level index with 'Device ID' and 'Location'

df\_merged.set\_index(['Device ID', 'Location'], inplace=True)

# Resample the data to hourly frequency and calculate mean values

df\_resampled = df\_merged.resample('H').mean()

# Calculate the rolling mean with a window of 24 hours

df\_resampled['Rolling\_Mean'] = df\_resampled['Sensor Value'].rolling(window=24).mean()

# Save the results to new CSV file

df\_resampled.to\_csv('resampled\_sensor\_data.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to merge IoT sensor data, resample it, and calculate rolling mean values. However, it has issues with correctly merging the dataframes, handling hierarchical indexing for resampling, and calculating the rolling mean.

5 - Answer:

The corrected code should properly merge the dataframes on the correct keys, handle hierarchical indexing, resample the data correctly, and calculate the rolling mean.

```python

import pandas as pd

# Load sensor data from multiple files

df1 = pd.read\_csv('sensor\_data\_1.csv')

df2 = pd.read\_csv('sensor\_data\_2.csv')

# Convert 'Timestamp' column to datetime

df1['Timestamp'] = pd.to\_datetime(df1['Timestamp'])

df2['Timestamp'] = pd.to\_datetime(df2['Timestamp'])

# Merge the dataframes on 'Timestamp' and 'Device ID'

df\_merged = pd.merge(df1, df2, on=['Timestamp', 'Device ID'], how='outer')

# Fill missing values with forward fill method

df\_merged.fillna(method='ffill', inplace=True)

# Set multi-level index with 'Device ID', 'Location', and 'Timestamp'

df\_merged.set\_index(['Device ID', 'Location', 'Timestamp'], inplace=True)

# Resample the data to hourly frequency and calculate mean values

df\_resampled = df\_merged.groupby(['Device ID', 'Location']).resample('H', level=2).mean()

# Calculate the rolling mean with a window of 24 hours

df\_resampled['Rolling\_Mean'] = df\_resampled['Sensor Value'].groupby(level=0).rolling(window=24).mean().reset\_index(level=0, drop=True)

# Save the results to new CSV file

df\_resampled.to\_csv('resampled\_sensor\_data.csv')

```

In the corrected code, we:

- Merge the dataframes using `how='outer'` to ensure no data is lost.

- Properly handle multi-level indexing by setting `Device ID`, `Location`, and `Timestamp` as the index.

- Resample the data to hourly frequency using the `resample` method with `level=2` to specify the `Timestamp` level.

- Calculate the rolling mean for each device separately by grouping the data and then applying the `rolling` method.

This ensures that the data is merged, resampled, and summarized correctly, providing accurate insights into IoT sensor data trends and anomalies.

### Question 12

Certainly! Here’s another scenario that tests advanced Pandas concepts such as handling categorical data, merging datasets with different shapes, and using advanced grouping operations:

0 - Real-life Application of Pandas: Marketing Campaign Analysis

1 - Scenario:

You are analyzing the effectiveness of a marketing campaign across different regions and customer segments. Your task involves cleaning the data, merging customer and campaign datasets, handling categorical data, and generating detailed performance reports.

2 - Buggy Code:

```python

import pandas as pd

# Load customer data

df\_customers = pd.read\_csv('customers.csv')

# Load campaign data

df\_campaign = pd.read\_csv('campaign.csv')

# Convert 'Date' column to datetime in both dataframes

df\_customers['Date'] = pd.to\_datetime(df\_customers['Date'])

df\_campaign['Date'] = pd.to\_datetime(df\_campaign['Date'])

# Merge the dataframes on 'Customer ID'

df\_merged = pd.merge(df\_customers, df\_campaign, on='Customer ID')

# Fill missing values with the mode for categorical columns

df\_merged['Region'].fillna(df\_merged['Region'].mode()[0], inplace=True)

df\_merged['Segment'].fillna(df\_merged['Segment'].mode()[0], inplace=True)

# Group by 'Region' and 'Segment' to calculate total sales and count of customers

region\_segment\_summary = df\_merged.groupby(['Region', 'Segment'])['Sales'].sum()

customer\_count = df\_merged.groupby(['Region', 'Segment'])['Customer ID'].count()

# Merge the summary dataframes

summary = pd.concat([region\_segment\_summary, customer\_count], axis=1, keys=['Total Sales', 'Customer Count'])

# Save the result to a new CSV file

summary.to\_csv('campaign\_summary.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to merge customer and campaign data, handle missing categorical data, and generate a summary report. However, it has issues with correctly merging the dataframes, handling missing values, and properly aligning the groupby results for concatenation.

5 - Answer:

The corrected code should handle merging the dataframes correctly, appropriately fill missing values, and align the groupby results for concatenation.

```python

import pandas as pd

# Load customer data

df\_customers = pd.read\_csv('customers.csv')

# Load campaign data

df\_campaign = pd.read\_csv('campaign.csv')

# Convert 'Date' column to datetime in both dataframes

df\_customers['Date'] = pd.to\_datetime(df\_customers['Date'])

df\_campaign['Date'] = pd.to\_datetime(df\_campaign['Date'])

# Merge the dataframes on 'Customer ID'

df\_merged = pd.merge(df\_customers, df\_campaign, on='Customer ID', how='left')

# Fill missing values with the mode for categorical columns

for col in ['Region', 'Segment']:

df\_merged[col].fillna(df\_merged[col].mode()[0], inplace=True)

# Group by 'Region' and 'Segment' to calculate total sales and count of customers

region\_segment\_summary = df\_merged.groupby(['Region', 'Segment'])['Sales'].sum().reset\_index()

customer\_count = df\_merged.groupby(['Region', 'Segment'])['Customer ID'].count().reset\_index()

# Merge the summary dataframes

summary = pd.merge(region\_segment\_summary, customer\_count, on=['Region', 'Segment'])

summary.rename(columns={'Customer ID': 'Customer Count'}, inplace=True)

# Save the result to a new CSV file

summary.to\_csv('campaign\_summary.csv', index=False)

```

In the corrected code, we:

- Merge the dataframes using `how='left'` to ensure that all customers are included, even if they didn't participate in the campaign.

- Use a loop to fill missing values for the categorical columns 'Region' and 'Segment' with the mode of each column.

- Reset the index after grouping by 'Region' and 'Segment' to ensure that the dataframes are properly aligned.

- Merge the summary dataframes on 'Region' and 'Segment', and rename the 'Customer ID' column to 'Customer Count'.

This ensures that the data is merged, cleaned, and summarized correctly, providing an accurate analysis of the marketing campaign's effectiveness across different regions and customer segments.

### Question 13

Sure! Let's focus on a different set of advanced Pandas concepts such as handling hierarchical indexing, reshaping data with `stack` and `unstack`, and performing advanced filtering operations.

0 - Real-life Application of Pandas: Employee Performance and Compensation Analysis

1 - Scenario:

You are working on a project where you need to analyze employee performance and compensation data across different departments and roles within a company. Your task involves cleaning the data, handling hierarchical indexing, reshaping the data, and performing complex filtering operations to generate detailed performance and compensation reports.

2 - Buggy Code:

```python

import pandas as pd

# Load performance data

df\_performance = pd.read\_csv('performance\_data.csv')

# Load compensation data

df\_compensation = pd.read\_csv('compensation\_data.csv')

# Convert 'Review Date' and 'Pay Date' columns to datetime

df\_performance['Review Date'] = pd.to\_datetime(df\_performance['Review Date'])

df\_compensation['Pay Date'] = pd.to\_datetime(df\_compensation['Pay Date'])

# Merge the dataframes on 'Employee ID'

df\_merged = pd.merge(df\_performance, df\_compensation, on='Employee ID')

# Set multi-level index with 'Department' and 'Role'

df\_merged.set\_index(['Department', 'Role'], inplace=True)

# Calculate the average performance score and total compensation by department and role

performance\_summary = df\_merged.groupby(['Department', 'Role'])['Performance Score'].mean()

compensation\_summary = df\_merged.groupby(['Department', 'Role'])['Compensation'].sum()

# Combine the summaries into a single DataFrame

summary = pd.concat([performance\_summary, compensation\_summary], axis=1, keys=['Avg Performance Score', 'Total Compensation'])

# Filter to get the top-performing roles in each department

top\_performers = summary[summary['Avg Performance Score'] > summary['Avg Performance Score'].quantile(0.75)]

# Save the result to a new CSV file

top\_performers.to\_csv('top\_performers.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to merge employee performance and compensation data, handle hierarchical indexing, and perform complex filtering operations. However, it has issues with properly merging the data, handling hierarchical indexing, and correctly filtering and reshaping the data.

5 - Answer:

The corrected code should properly merge the data, handle hierarchical indexing, and correctly filter and reshape the data.

```python

import pandas as pd

# Load performance data

df\_performance = pd.read\_csv('performance\_data.csv')

# Load compensation data

df\_compensation = pd.read\_csv('compensation\_data.csv')

# Convert 'Review Date' and 'Pay Date' columns to datetime

df\_performance['Review Date'] = pd.to\_datetime(df\_performance['Review Date'])

df\_compensation['Pay Date'] = pd.to\_datetime(df\_compensation['Pay Date'])

# Merge the dataframes on 'Employee ID'

df\_merged = pd.merge(df\_performance, df\_compensation, on='Employee ID', how='outer')

# Set multi-level index with 'Department', 'Role', and 'Employee ID'

df\_merged.set\_index(['Department', 'Role', 'Employee ID'], inplace=True)

# Calculate the average performance score and total compensation by department and role

performance\_summary = df\_merged.groupby(level=['Department', 'Role'])['Performance Score'].mean().unstack()

compensation\_summary = df\_merged.groupby(level=['Department', 'Role'])['Compensation'].sum().unstack()

# Combine the summaries into a single DataFrame

summary = pd.concat([performance\_summary, compensation\_summary], axis=1, keys=['Avg Performance Score', 'Total Compensation'])

# Filter to get the top-performing roles in each department

summary\_stack = summary.stack(level=0) # Reshape for easier filtering

top\_performers = summary\_stack[summary\_stack['Avg Performance Score'] > summary\_stack['Avg Performance Score'].groupby(level=0).transform('quantile', 0.75)]

# Save the result to a new CSV file

top\_performers.to\_csv('top\_performers.csv')

```

In the corrected code, we:

- Use `how='outer'` during merging to include all records.

- Set a multi-level index including 'Employee ID' for accurate grouping.

- Use `unstack` to reshape the data after grouping.

- Combine the summaries with `pd.concat`.

- Reshape the combined summary with `stack` for easier filtering.

- Apply the filter to get the top-performing roles based on the 75th percentile within each department.

This ensures accurate merging, proper hierarchical indexing, and correct filtering and reshaping for detailed performance and compensation analysis.

### Question 14

Sure! Let's focus on a scenario that tests advanced Pandas concepts such as time series resampling, handling missing data, and using advanced aggregation and filtering operations.

0 - Real-life Application of Pandas: Environmental Data Analysis

1 - Scenario:

You are analyzing environmental data collected from various sensors measuring temperature, humidity, and air quality over time. Your task involves cleaning the data, resampling the time series data to a daily frequency, handling missing values, and performing advanced aggregations to generate summary statistics.

2 - Buggy Code:

```python

import pandas as pd

# Load sensor data

df = pd.read\_csv('environmental\_data.csv')

# Convert 'Timestamp' column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Set 'Timestamp' as the index

df.set\_index('Timestamp', inplace=True)

# Resample data to daily frequency

df\_daily = df.resample('D').mean()

# Fill missing values with the mean of the column

df\_daily.fillna(df.mean(), inplace=True)

# Calculate daily summary statistics

summary = df\_daily.groupby(df\_daily.index)['Temperature', 'Humidity', 'Air Quality'].agg(['mean', 'max', 'min'])

# Filter to get days with extreme temperature variations

extreme\_days = summary[(summary['Temperature']['max'] - summary['Temperature']['min']) > 10]

# Save the result to a new CSV file

extreme\_days.to\_csv('extreme\_days.csv')

```

3 - Question:

Explain what issue you encounter in the code.

4 - Explanation:

The code attempts to resample environmental data to a daily frequency, handle missing values, and perform advanced aggregations. However, it has issues with handling missing values properly, correctly applying groupby operations, and properly filtering the results.

5 - Answer:

The corrected code should properly handle missing values, correctly apply aggregation operations, and filter the results accurately.

```python

import pandas as pd

# Load sensor data

df = pd.read\_csv('environmental\_data.csv')

# Convert 'Timestamp' column to datetime

df['Timestamp'] = pd.to\_datetime(df['Timestamp'])

# Set 'Timestamp' as the index

df.set\_index('Timestamp', inplace=True)

# Resample data to daily frequency

df\_daily = df.resample('D').mean()

# Fill missing values with the mean of the column

df\_daily.fillna(df\_daily.mean(), inplace=True)

# Calculate daily summary statistics

summary = df\_daily.agg({'Temperature': ['mean', 'max', 'min'],

'Humidity': ['mean', 'max', 'min'],

'Air Quality': ['mean', 'max', 'min']})

# Filter to get days with extreme temperature variations

extreme\_days = df\_daily[(df\_daily['Temperature'].max() - df\_daily['Temperature'].min()) > 10]

# Save the result to a new CSV file

extreme\_days.to\_csv('extreme\_days.csv')

```

In the corrected code, we:

- Fill missing values with the mean of each column after resampling (`df\_daily.mean()`).

- Use `agg` to calculate daily summary statistics for temperature, humidity, and air quality.

- Correctly filter the days with extreme temperature variations by applying the condition on the resampled daily data (`df\_daily`) instead of the summary statistics.

This ensures that the data is resampled, cleaned, and summarized accurately, providing useful insights into environmental conditions and identifying days with extreme variations.

# Django

## GPT Link

<https://chatgpt.com/share/9edfd86d-4b9d-4fc4-951c-dc56d3ed2013>

## Part 1 - Questions (gpt4)

### Question 1

0 - Topics

- Django ORM (Object-Relational Mapping)

- Query Optimization

- Transactions and Atomic Operations

- Custom Model Methods

1 - Scenario

A company uses a Django application to manage orders and their corresponding invoices. Each time an order is placed, an invoice should be created and linked to the order. The developer wrote a method to handle this process, but it seems that sometimes invoices are not being created, causing inconsistencies.

2 - Code

```python

from django.db import models, transaction

class Order(models.Model):

customer\_name = models.CharField(max\_length=100)

order\_date = models.DateTimeField(auto\_now\_add=True)

total\_amount = models.DecimalField(max\_digits=10, decimal\_places=2)

class Invoice(models.Model):

order = models.ForeignKey(Order, on\_delete=models.CASCADE)

invoice\_date = models.DateTimeField(auto\_now\_add=True)

amount\_due = models.DecimalField(max\_digits=10, decimal\_places=2)

class OrderService:

def create\_order\_with\_invoice(self, customer\_name, amount):

order = Order(customer\_name=customer\_name, total\_amount=amount)

order.save()

self.create\_invoice(order)

def create\_invoice(self, order):

invoice = Invoice(order=order, amount\_due=order.total\_amount)

invoice.save()

# Using the service

service = OrderService()

service.create\_order\_with\_invoice("John Doe", 150.00)

```

3 - Question

There seems to be an issue where invoices are not always created when an order is placed. Identify and explain the issue in the code.

4 - Explanation

The code does not handle potential errors that might occur when saving the order or creating the invoice. If an error occurs after the order is saved but before the invoice is saved, the order will exist without an associated invoice, causing data inconsistencies. This can happen if there is a database issue or some other unexpected exception.

5 - Answer

```python

from django.db import models, transaction

class Order(models.Model):

customer\_name = models.CharField(max\_length=100)

order\_date = models.DateTimeField(auto\_now\_add=True)

total\_amount = models.DecimalField(max\_digits=10, decimal\_places=2)

class Invoice(models.Model):

order = models.ForeignKey(Order, on\_delete=models.CASCADE)

invoice\_date = models.DateTimeField(auto\_now\_add=True)

amount\_due = models.DecimalField(max\_digits=10, decimal\_places=2)

class OrderService:

def create\_order\_with\_invoice(self, customer\_name, amount):

try:

with transaction.atomic():

order = Order(customer\_name=customer\_name, total\_amount=amount)

order.save()

self.create\_invoice(order)

except Exception as e:

# Handle exception or log error

print(f"Error occurred: {e}")

def create\_invoice(self, order):

invoice = Invoice(order=order, amount\_due=order.total\_amount)

invoice.save()

# Using the service

service = OrderService()

service.create\_order\_with\_invoice("John Doe", 150.00)

```

### Question 2

0 - Topics

- Custom Manager Methods

- QuerySets and Lazy Evaluation

- Aggregation

- Handling Large Querysets

1 - Scenario

A blog application needs to display the top 5 most commented posts. The developer has written a custom manager method to achieve this. However, the page displaying the top posts is very slow, indicating there might be inefficiencies in the code.

2 - Code

```python

from django.db import models

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

published\_date = models.DateTimeField(auto\_now\_add=True)

class Comment(models.Model):

post = models.ForeignKey(Post, related\_name='comments', on\_delete=models.CASCADE)

content = models.TextField()

created\_date = models.DateTimeField(auto\_now\_add=True)

class PostManager(models.Manager):

def top\_commented\_posts(self):

posts = self.get\_queryset()

posts\_with\_comment\_count = [(post, post.comments.count()) for post in posts]

sorted\_posts = sorted(posts\_with\_comment\_count, key=lambda x: x[1], reverse=True)

return [post for post, count in sorted\_posts[:5]]

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

published\_date = models.DateTimeField(auto\_now\_add=True)

objects = PostManager()

# Using the custom manager

top\_posts = Post.objects.top\_commented\_posts()

```

3 - Question

The page that displays the top 5 most commented posts is very slow. Identify and explain the inefficiency in the code.

4 - Explanation

The code retrieves all posts and then manually counts the comments for each post in Python, which results in multiple database queries (one for each post). This is highly inefficient, especially with a large number of posts, causing significant performance degradation.

5 - Answer

```python

from django.db import models

from django.db.models import Count

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

published\_date = models.DateTimeField(auto\_now\_add=True)

class Comment(models.Model):

post = models.ForeignKey(Post, related\_name='comments', on\_delete=models.CASCADE)

content = models.TextField()

created\_date = models.DateTimeField(auto\_now\_add=True)

class PostManager(models.Manager):

def top\_commented\_posts(self):

return self.get\_queryset().annotate(comment\_count=Count('comments')).order\_by('-comment\_count')[:5]

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

published\_date = models.DateTimeField(auto\_now\_add=True)

objects = PostManager()

# Using the custom manager

top\_posts = Post.objects.top\_commented\_posts()

```

### Question 3

0 - Topics

- Custom Model Methods

- Signals and Side Effects

- Data Integrity

- Concurrency Issues

1 - Scenario

An e-commerce application has a Product model with a method to decrease the stock quantity when an order is placed. A signal is used to call this method after an Order object is created. However, sometimes the stock quantity does not decrease correctly, especially under high load.

2 - Code

```python

from django.db import models

from django.db.models.signals import post\_save

from django.dispatch import receiver

class Product(models.Model):

name = models.CharField(max\_length=100)

stock = models.IntegerField()

def decrease\_stock(self, quantity):

self.stock -= quantity

self.save()

class Order(models.Model):

product = models.ForeignKey(Product, on\_delete=models.CASCADE)

quantity = models.PositiveIntegerField()

order\_date = models.DateTimeField(auto\_now\_add=True)

@receiver(post\_save, sender=Order)

def decrease\_product\_stock(sender, instance, \*\*kwargs):

instance.product.decrease\_stock(instance.quantity)

# Creating an order

product = Product.objects.create(name="Laptop", stock=10)

order = Order.objects.create(product=product, quantity=1)

```

3 - Question

Under high load, the stock quantity of products is not always updated correctly. Identify and explain the concurrency issue in the code.

4 - Explanation

The code does not handle concurrent updates to the product stock properly. When multiple orders are placed simultaneously, race conditions can occur, causing the stock quantity to be updated incorrectly. This happens because the `decrease\_stock` method reads and writes the stock without any locking mechanism to ensure atomicity.

5 - Answer

```python

from django.db import models

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.db import transaction

class Product(models.Model):

name = models.CharField(max\_length=100)

stock = models.IntegerField()

def decrease\_stock(self, quantity):

with transaction.atomic():

self.refresh\_from\_db()

self.stock -= quantity

self.save()

class Order(models.Model):

product = models.ForeignKey(Product, on\_delete=models.CASCADE)

quantity = models.PositiveIntegerField()

order\_date = models.DateTimeField(auto\_now\_add=True)

@receiver(post\_save, sender=Order)

def decrease\_product\_stock(sender, instance, \*\*kwargs):

instance.product.decrease\_stock(instance.quantity)

# Creating an order

product = Product.objects.create(name="Laptop", stock=10)

order = Order.objects.create(product=product, quantity=1)

```

### Question 4

0 - Topics

- Signal Handling

- Data Integrity

- Avoiding Circular Imports

- Best Practices in Signal Usage

1 - Scenario

A notification system in a Django application sends an email to the user each time their profile is updated. The developer implemented a signal to handle this. However, there are cases where users report receiving multiple emails for a single update.

2 - Code

```python

from django.db import models

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.core.mail import send\_mail

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

bio = models.TextField()

@receiver(post\_save, sender=UserProfile)

def send\_profile\_update\_email(sender, instance, \*\*kwargs):

send\_mail(

'Profile Updated',

'Your profile has been updated.',

'from@example.com',

[instance.user.email],

fail\_silently=False,

)

# Updating user profile

user\_profile = UserProfile.objects.get(user\_id=1)

user\_profile.bio = "New bio"

user\_profile.save()

```

3 - Question

Users are receiving multiple emails for a single profile update. Identify and explain the issue in the code.

4 - Explanation

The signal is connected in such a way that it triggers on every save

operation of the UserProfile model, including any internal or secondary saves that may occur within other signals or save operations, leading to multiple emails being sent for a single logical update.

5 - Answer

```python

from django.db import models

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.core.mail import send\_mail

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

bio = models.TextField()

@receiver(post\_save, sender=UserProfile)

def send\_profile\_update\_email(sender, instance, created, \*\*kwargs):

if not created:

send\_mail(

'Profile Updated',

'Your profile has been updated.',

'from@example.com',

[instance.user.email],

fail\_silently=False,

)

# Updating user profile

user\_profile = UserProfile.objects.get(user\_id=1)

user\_profile.bio = "New bio"

user\_profile.save()

```

In this corrected version, the signal handler checks if the instance is newly created or updated. The email is sent only if the instance is updated (not newly created), avoiding multiple emails for a single update.

### Question 5

0 - Topics

- Django Forms and Form Validation

- Custom Validators

- Handling File Uploads

1 - Scenario

An application allows users to upload their profile pictures. The developer has implemented a custom form to handle the upload and added a custom validator to ensure the uploaded file is an image. However, users are reporting that non-image files are being accepted.

2 - Code

```python

from django import forms

from django.core.exceptions import ValidationError

from django.core.files.images import get\_image\_dimensions

from django.core.files.uploadedfile import UploadedFile

from django.db import models

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

profile\_picture = models.ImageField(upload\_to='profile\_pics/')

class ProfilePictureForm(forms.ModelForm):

class Meta:

model = UserProfile

fields = ['profile\_picture']

def clean\_profile\_picture(self):

file = self.cleaned\_data.get('profile\_picture')

if file:

if not self.is\_image(file):

raise ValidationError("Uploaded file is not an image.")

return file

def is\_image(self, file: UploadedFile):

try:

get\_image\_dimensions(file)

return True

except Exception:

return False

# Handling the form in a view

def upload\_profile\_picture(request):

if request.method == 'POST':

form = ProfilePictureForm(request.POST, request.FILES)

if form.is\_valid():

form.save()

else:

form = ProfilePictureForm()

return render(request, 'upload.html', {'form': form})

```

3 - Question

Non-image files are being accepted as valid profile pictures. Identify and explain the issue in the code.

4 - Explanation

The `get\_image\_dimensions` function only checks the dimensions of the image but does not validate if the file is an image itself. If the file isn't an image, `get\_image\_dimensions` might still return dimensions without raising an exception, leading to incorrect validation results.

5 - Answer

```python

from django import forms

from django.core.exceptions import ValidationError

from django.core.files.images import get\_image\_dimensions

from django.core.files.uploadedfile import UploadedFile

from django.db import models

from PIL import Image

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

profile\_picture = models.ImageField(upload\_to='profile\_pics/')

class ProfilePictureForm(forms.ModelForm):

class Meta:

model = UserProfile

fields = ['profile\_picture']

def clean\_profile\_picture(self):

file = self.cleaned\_data.get('profile\_picture')

if file:

if not self.is\_image(file):

raise ValidationError("Uploaded file is not an image.")

return file

def is\_image(self, file: UploadedFile):

try:

# Open the file to verify it's an image

Image.open(file).verify()

return True

except Exception:

return False

# Handling the form in a view

def upload\_profile\_picture(request):

if request.method == 'POST':

form = ProfilePictureForm(request.POST, request.FILES)

if form.is\_valid():

form.save()

else:

form = ProfilePictureForm()

return render(request, 'upload.html', {'form': form})

```

### Question 6

0 - Topics

- Django Middleware

- Request/Response Processing

- Error Handling and Logging

1 - Scenario

A company uses custom middleware to log every request and response. They noticed that some requests and responses are not being logged, especially those that result in an error. The developer needs to ensure all requests and responses, including those leading to errors, are logged.

2 - Code

```python

import logging

from django.utils.deprecation import MiddlewareMixin

logger = logging.getLogger(\_\_name\_\_)

class LoggingMiddleware(MiddlewareMixin):

def process\_request(self, request):

logger.info(f"Request: {request.method} {request.get\_full\_path()}")

def process\_response(self, request, response):

logger.info(f"Response: {response.status\_code}")

return response

# Add middleware in settings

MIDDLEWARE = [

# Other middlewares

'myapp.middleware.LoggingMiddleware',

]

# settings.py for logger configuration

LOGGING = {

'version': 1,

'handlers': {

'console': {

'class': 'logging.StreamHandler',

},

},

'root': {

'handlers': ['console'],

'level': 'INFO',

},

}

```

3 - Question

Some requests and responses, particularly those resulting in errors, are not being logged. Identify and explain the issue in the code.

4 - Explanation

The current implementation only logs successful responses. If an error occurs during request processing, the `process\_response` method is not called, resulting in missed logs for those requests.

5 - Answer

```python

import logging

from django.utils.deprecation import MiddlewareMixin

logger = logging.getLogger(\_\_name\_\_)

class LoggingMiddleware(MiddlewareMixin):

def process\_request(self, request):

logger.info(f"Request: {request.method} {request.get\_full\_path()}")

def process\_response(self, request, response):

logger.info(f"Response: {response.status\_code}")

return response

def process\_exception(self, request, exception):

logger.error(f"Exception: {exception}")

return None # Allows other middleware to handle the exception

# Add middleware in settings

MIDDLEWARE = [

# Other middlewares

'myapp.middleware.LoggingMiddleware',

]

# settings.py for logger configuration

LOGGING = {

'version': 1,

'handlers': {

'console': {

'class': 'logging.StreamHandler',

},

},

'root': {

'handlers': ['console'],

'level': 'INFO',

},

}

```

In this corrected version, the `process\_exception` method is added to the middleware to log exceptions. This ensures that requests resulting in errors are also logged.

### Question 7

0 - Topics

- Django Caching

- Template Caching

- Cache Invalidation

1 - Scenario

A news website caches the homepage to improve performance. The homepage shows the latest articles. However, users are reporting that new articles are not appearing on the homepage immediately after being published.

2 - Code

```python

from django.core.cache import cache

from django.shortcuts import render

from myapp.models import Article

def homepage(request):

cached\_homepage = cache.get('homepage')

if cached\_homepage:

return cached\_homepage

articles = Article.objects.order\_by('-published\_date')[:10]

response = render(request, 'homepage.html', {'articles': articles})

cache.set('homepage', response, 300) # Cache for 5 minutes

return response

# Adding a new article

def add\_article(request):

if request.method == 'POST':

title = request.POST.get('title')

content = request.POST.get('content')

article = Article.objects.create(title=title, content=content)

cache.delete('homepage') # Invalidate the homepage cache

return redirect('homepage')

return render(request, 'add\_article.html')

```

3 - Question

New articles are not appearing on the homepage immediately after being published. Identify and explain the issue in the code.

4 - Explanation

The cache invalidation logic is correct, but there's a potential race condition. If the homepage is accessed frequently, there might be a small window where the cache is deleted but not yet regenerated, causing the old cache to still be served temporarily.

5 - Answer

```python

from django.core.cache import cache

from django.shortcuts import render

from myapp.models import Article

def homepage(request):

cached\_homepage = cache.get('homepage')

if cached\_homepage:

return cached\_homepage

articles = Article.objects.order\_by('-published\_date')[:10]

response = render(request, 'homepage.html', {'articles': articles})

cache.set('homepage', response, 300) # Cache for 5 minutes

return response

# Adding a new article

def add\_article(request):

if request.method == 'POST':

title = request.POST.get('title')

content = request.POST.get('content')

article = Article.objects.create(title=title, content=content)

# Double check if cache deletion is necessary before and after article creation

cache.delete('homepage') # Invalidate the homepage cache

return redirect('homepage')

return render(request, 'add\_article.html')

```

In the corrected version, we ensure the cache is deleted before redirecting to the homepage, thus minimizing the window where the old cache might still be served. Additional precautions can be taken if necessary, such as more aggressive invalidation or instant cache regeneration.

### Question 8

0 - Topics

- Django Signals

- Data Consistency

- Database Transactions

1 - Scenario

A web application tracks the number of views for each article. The developer implemented a signal to increment the view count every time an article is viewed. However, under heavy traffic, the view counts are inaccurate.

2 - Code

```python

from django.db import models

from django.db.models.signals import post\_save

from django.dispatch import receiver

class Article(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

view\_count = models.PositiveIntegerField(default=0)

class ArticleView(models.Model):

article = models.ForeignKey(Article, on\_delete=models.CASCADE)

view\_date = models.DateTimeField(auto\_now\_add=True)

@receiver(post\_save, sender=ArticleView)

def increment\_view\_count(sender, instance, \*\*kwargs):

article = instance.article

article.view\_count += 1

article

.save()

# View function to handle article viewing

def view\_article(request, article\_id):

article = Article.objects.get(id=article\_id)

ArticleView.objects.create(article=article)

return render(request, 'article.html', {'article': article})

```

3 - Question

The view counts are inaccurate under heavy traffic. Identify and explain the issue in the code.

4 - Explanation

The issue is due to a race condition where multiple increments to the view count can occur simultaneously, causing some increments to be lost. This happens because each view triggers a separate database save operation without any locking mechanism to ensure atomicity.

5 - Answer

```python

from django.db import models

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.db import transaction

class Article(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

view\_count = models.PositiveIntegerField(default=0)

class ArticleView(models.Model):

article = models.ForeignKey(Article, on\_delete=models.CASCADE)

view\_date = models.DateTimeField(auto\_now\_add=True)

@receiver(post\_save, sender=ArticleView)

def increment\_view\_count(sender, instance, \*\*kwargs):

with transaction.atomic():

article = instance.article

article.view\_count = Article.objects.select\_for\_update().get(id=article.id).view\_count + 1

article.save()

# View function to handle article viewing

def view\_article(request, article\_id):

article = Article.objects.get(id=article\_id)

ArticleView.objects.create(article=article)

return render(request, 'article.html', {'article': article})

```

In this corrected version, the `select\_for\_update` method is used within a transaction to lock the row and ensure atomicity, preventing race conditions and ensuring accurate view counts under heavy traffic.

These additional debugging questions cover more advanced Django topics, providing challenging scenarios for senior developers to solve.

### Question 9

0 - Topics

- Django ORM

- Migrations

- Data Consistency

1 - Scenario

A Django application manages events and attendees. The developer recently added a new field to track the date when an attendee registered for an event. However, after deploying the changes, users are reporting errors when registering for events.

2 - Code

```python

from django.db import models

class Event(models.Model):

name = models.CharField(max\_length=100)

event\_date = models.DateTimeField()

class Attendee(models.Model):

event = models.ForeignKey(Event, on\_delete=models.CASCADE)

name = models.CharField(max\_length=100)

email = models.EmailField()

registered\_date = models.DateTimeField(auto\_now\_add=True)

# View to handle attendee registration

def register\_attendee(request, event\_id):

event = Event.objects.get(id=event\_id)

name = request.POST.get('name')

email = request.POST.get('email')

attendee = Attendee(event=event, name=name, email=email)

attendee.save()

return render(request, 'registration\_success.html', {'attendee': attendee})

```

3 - Question

Users are reporting errors when registering for events after the new field was added. Identify and explain the issue in the code.

4 - Explanation

The issue likely arises from not running the database migrations after adding the new field to the `Attendee` model. This means the database schema is out of sync with the model definition, causing errors when attempting to save new attendees.

5 - Answer

```python

# Running the migrations

$ python manage.py makemigrations

$ python manage.py migrate

# No changes needed in the existing code since the migrations are now in sync

from django.db import models

class Event(models.Model):

name = models.CharField(max\_length=100)

event\_date = models.DateTimeField()

class Attendee(models.Model):

event = models.ForeignKey(Event, on\_delete=models.CASCADE)

name = models.CharField(max\_length=100)

email = models.EmailField()

registered\_date = models.DateTimeField(auto\_now\_add=True)

# View to handle attendee registration

def register\_attendee(request, event\_id):

event = Event.objects.get(id=event\_id)

name = request.POST.get('name')

email = request.POST.get('email')

attendee = Attendee(event=event, name=name, email=email)

attendee.save()

return render(request, 'registration\_success.html', {'attendee': attendee})

```

By ensuring that the database migrations are applied, the new field `registered\_date` will be properly added to the database schema, resolving the registration errors.

### Question 10

0 - Topics

- Django REST Framework

- Serializer Validation

- API Permissions

1 - Scenario

A Django REST API allows users to create and update profiles. The developer implemented custom validation to ensure that the username is unique. However, users are reporting that they can sometimes create profiles with duplicate usernames.

2 - Code

```python

from rest\_framework import serializers, viewsets

from rest\_framework.permissions import IsAuthenticated

from myapp.models import UserProfile

class UserProfileSerializer(serializers.ModelSerializer):

class Meta:

model = UserProfile

fields = ['username', 'email', 'bio']

def validate\_username(self, value):

if UserProfile.objects.filter(username=value).exists():

raise serializers.ValidationError("Username already exists")

return value

class UserProfileViewSet(viewsets.ModelViewSet):

queryset = UserProfile.objects.all()

serializer\_class = UserProfileSerializer

permission\_classes = [IsAuthenticated]

# Registering the viewset in urls.py

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from myapp.views import UserProfileViewSet

router = DefaultRouter()

router.register(r'profiles', UserProfileViewSet)

urlpatterns = [

path('', include(router.urls)),

]

```

3 - Question

Users are reporting that they can sometimes create profiles with duplicate usernames. Identify and explain the issue in the code.

4 - Explanation

The issue is due to a race condition where two requests can pass the `validate\_username` check simultaneously before either saves the new profile to the database. This allows both requests to create profiles with the same username.

5 - Answer

```python

from rest\_framework import serializers, viewsets

from rest\_framework.permissions import IsAuthenticated

from myapp.models import UserProfile

from django.db import IntegrityError

class UserProfileSerializer(serializers.ModelSerializer):

class Meta:

model = UserProfile

fields = ['username', 'email', 'bio']

def create(self, validated\_data):

try:

return UserProfile.objects.create(\*\*validated\_data)

except IntegrityError:

raise serializers.ValidationError("Username already exists")

class UserProfileViewSet(viewsets.ModelViewSet):

queryset = UserProfile.objects.all()

serializer\_class = UserProfileSerializer

permission\_classes = [IsAuthenticated]

# Registering the viewset in urls.py

from django.urls import path, include

from rest\_framework.routers import DefaultRouter

from myapp.views import UserProfileViewSet

router = DefaultRouter()

router.register(r'profiles', UserProfileViewSet)

urlpatterns = [

path('', include(router.urls)),

]

```

In this corrected version, the `create` method catches `IntegrityError` exceptions raised by the database if a duplicate username is attempted. This ensures that even concurrent requests cannot create duplicate usernames.

### Question 11

0 - Topics

- Asynchronous Views

- Django Channels

- WebSockets

1 - Scenario

A chat application uses Django Channels to handle WebSocket connections for real-time messaging. Users are reporting that messages are sometimes delayed or lost during heavy usage.

2 - Code

```python

# consumers.py

import json

from channels.generic.websocket import WebsocketConsumer

class ChatConsumer(WebsocketConsumer):

def connect(self):

self.accept()

def disconnect(self, close\_code):

pass

def receive(self, text\_data):

data = json.loads(text\_data)

message = data['message']

self.send(text\_data=json.dumps({

'message': message

}))

# routing.py

from django.urls import path

from myapp.consumers import ChatConsumer

websocket\_urlpatterns = [

path('ws/chat/', ChatConsumer.as\_asgi()),

]

# settings.py

INSTALLED\_APPS = [

# Other installed apps

'channels',

]

ASGI\_APPLICATION = 'myproject.asgi.application'

CHANNEL\_LAYERS = {

'default': {

'BACKEND': 'channels.layers.InMemoryChannelLayer',

},

}

```

3 - Question

Users are reporting that messages are sometimes delayed or lost during heavy usage. Identify and explain the issue in the code.

4 - Explanation

The issue is that the application uses an `InMemoryChannelLayer` which is not suitable for production. It does not handle scaling and persistence well, leading to message delays or losses under heavy usage.

5 - Answer

```python

# consumers.py

import json

from channels.generic.websocket import AsyncWebsocketConsumer

class ChatConsumer(AsyncWebsocketConsumer):

async def connect(self):

await self.accept()

async def disconnect(self, close\_code):

pass

async def receive(self, text\_data):

data = json.loads(text\_data)

message = data['message']

await self.send(text\_data=json.dumps({

'message': message

}))

# routing.py

from django.urls import path

from myapp.consumers import ChatConsumer

websocket\_urlpatterns = [

path('ws/chat/', ChatConsumer.as\_asgi()),

]

# settings.py

INSTALLED\_APPS = [

# Other installed apps

'channels',

]

ASGI\_APPLICATION = 'myproject.asgi.application'

CHANNEL\_LAYERS = {

'default': {

'BACKEND': 'channels\_redis.core.RedisChannelLayer',

'CONFIG': {

"hosts": [('127.0.0.1', 6379)],

},

},

}

```

In this corrected version, `AsyncWebsocketConsumer` is used for handling WebSocket connections asynchronously. The `CHANNEL\_LAYERS` setting is updated to use Redis as the backend, which is more suitable for production environments and handles message persistence and scaling much better.

### Question 12

0 - Topics

- Django Middleware

- Security

- Cross-Site Request Forgery (CSRF) Protection

1 - Scenario

A Django application uses custom middleware to log user actions. However, users are reporting that certain actions are being rejected with CSRF errors, even though CSRF protection is enabled.

2 - Code

```python

import logging

from django.utils.deprecation import MiddlewareMixin

from django.middleware.csrf import CsrfViewMiddleware

logger = logging.getLogger(\_\_name\_\_)

class ActionLoggingMiddleware(MiddlewareMixin):

def process\_view(self, request, view\_func, view\_args, view\_kwargs):

user = getattr(request, 'user', None)

if user and user.is\_authenticated:

logger.info(f"User {user.username} accessed {request.path}")

# settings.py

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'myapp.middleware.ActionLoggingMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

# Sample view

from django.shortcuts import render

from django.views.decorators.csrf import csrf\_protect

@csrf\_protect

def protected\_view(request):

if request.method == 'POST':

# Process the form

pass

return render(request, 'protected\_view.html')

```

3 - Question

Certain

actions are being rejected with CSRF errors. Identify and explain the issue in the code.

4 - Explanation

The custom middleware `ActionLoggingMiddleware` is placed after `CsrfViewMiddleware` in the middleware order. If `ActionLoggingMiddleware` modifies the request in any way that affects CSRF token validation, it can cause CSRF errors.

5 - Answer

```python

import logging

from django.utils.deprecation import MiddlewareMixin

from django.middleware.csrf import CsrfViewMiddleware

logger = logging.getLogger(\_\_name\_\_)

class ActionLoggingMiddleware(MiddlewareMixin):

def process\_view(self, request, view\_func, view\_args, view\_kwargs):

user = getattr(request, 'user', None)

if user and user.is\_authenticated:

logger.info(f"User {user.username} accessed {request.path}")

# settings.py

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'myapp.middleware.ActionLoggingMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

# Sample view

from django.shortcuts import render

from django.views.decorators.csrf import csrf\_protect

@csrf\_protect

def protected\_view(request):

if request.method == 'POST':

# Process the form

pass

return render(request, 'protected\_view.html')

```

In the corrected version, the `ActionLoggingMiddleware` is placed before `CsrfViewMiddleware` to ensure that any logging does not interfere with CSRF token validation.

### Question 13

0 - Topics

- Django Celery

- Asynchronous Task Queues

- Error Handling in Celery Tasks

1 - Scenario

A Django application uses Celery to handle background tasks such as sending emails. The developer noticed that some tasks fail silently without any indication of an error, causing important emails to be missed.

2 - Code

```python

# tasks.py

from celery import shared\_task

from django.core.mail import send\_mail

@shared\_task

def send\_email\_task(subject, message, from\_email, recipient\_list):

send\_mail(subject, message, from\_email, recipient\_list)

# views.py

from myapp.tasks import send\_email\_task

def send\_welcome\_email(request):

send\_email\_task.delay(

'Welcome!',

'Thank you for signing up.',

'from@example.com',

[request.user.email]

)

return render(request, 'email\_sent.html')

# celery.py

from \_\_future\_\_ import absolute\_import, unicode\_literals

import os

from celery import Celery

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'myproject.settings')

app = Celery('myproject')

app.config\_from\_object('django.conf:settings', namespace='CELERY')

app.autodiscover\_tasks()

```

3 - Question

Some Celery tasks fail silently without any indication of an error. Identify and explain the issue in the code.

4 - Explanation

The Celery task does not include any error handling or logging, so if an exception occurs (e.g., email sending fails), it fails silently. This makes it difficult to debug and understand why tasks are failing.

5 - Answer

```python

# tasks.py

from celery import shared\_task

from django.core.mail import send\_mail

import logging

logger = logging.getLogger(\_\_name\_\_)

@shared\_task(bind=True)

def send\_email\_task(self, subject, message, from\_email, recipient\_list):

try:

send\_mail(subject, message, from\_email, recipient\_list)

except Exception as exc:

logger.error(f"Error sending email: {exc}")

self.retry(exc=exc, countdown=60, max\_retries=3)

# views.py

from myapp.tasks import send\_email\_task

def send\_welcome\_email(request):

send\_email\_task.delay(

'Welcome!',

'Thank you for signing up.',

'from@example.com',

[request.user.email]

)

return render(request, 'email\_sent.html')

# celery.py

from \_\_future\_\_ import absolute\_import, unicode\_literals

import os

from celery import Celery

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'myproject.settings')

app = Celery('myproject')

app.config\_from\_object('django.conf:settings', namespace='CELERY')

app.autodiscover\_tasks()

```

In the corrected version, error handling and logging are added to the Celery task. If an exception occurs, it is logged, and the task retries up to three times with a delay, providing better error tracking and handling.

### Question 14

0 - Topics

- Django Template Tags and Filters

- Custom Template Tags

- Context Processing

1 - Scenario

A Django application uses a custom template tag to display a user's profile information in a sidebar. However, users are reporting that the sidebar sometimes displays incorrect or outdated information.

2 - Code

```python

# templatetags/profile\_tags.py

from django import template

from myapp.models import UserProfile

register = template.Library()

@register.simple\_tag

def user\_profile(user\_id):

try:

return UserProfile.objects.get(user\_id=user\_id)

except UserProfile.DoesNotExist:

return None

# sidebar.html

{% load profile\_tags %}

{% user\_profile request.user.id as profile %}

<div class="sidebar">

<h2>{{ profile.username }}</h2>

<p>{{ profile.bio }}</p>

</div>

# views.py

from django.shortcuts import render

def dashboard(request):

return render(request, 'dashboard.html')

```

3 - Question

The sidebar sometimes displays incorrect or outdated information. Identify and explain the issue in the code.

4 - Explanation

The custom template tag `user\_profile` fetches the profile using the user ID, but it does not account for any caching or data updates. If the profile data is updated, it might not reflect immediately due to caching or database lag.

5 - Answer

```python

# templatetags/profile\_tags.py

from django import template

from myapp.models import UserProfile

from django.core.cache import cache

register = template.Library()

@register.simple\_tag

def user\_profile(user\_id):

cache\_key = f'user\_profile\_{user\_id}'

profile = cache.get(cache\_key)

if not profile:

try:

profile = UserProfile.objects.get(user\_id=user\_id)

cache.set(cache\_key, profile, 300) # Cache for 5 minutes

except UserProfile.DoesNotExist:

return None

return profile

# sidebar.html

{% load profile\_tags %}

{% user\_profile request.user.id as profile %}

<div class="sidebar">

<h2>{{ profile.username }}</h2>

<p>{{ profile.bio }}</p>

</div>

# views.py

from django.shortcuts import render

def dashboard(request):

return render(request, 'dashboard.html')

```

In the corrected version, the custom template tag caches the user profile data to reduce database hits and ensure that changes are reflected after a cache expiry time, improving consistency and performance.

### Question 15

0 - Topics

- Django ORM

- Aggregation

- Prefetch Related

1 - Scenario

A blog application shows the number of comments for each post on the homepage. Users report that the homepage loads slowly when there are many posts and comments.

2 - Code

```python

from django.shortcuts import render

from myapp.models import Post, Comment

def homepage(request):

posts = Post.objects.all()

post\_comments = {post.id: post.comments.count() for post in posts}

return render(request, 'homepage.html', {'posts': posts, 'post\_comments': post\_comments})

# models.py

from django.db import models

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

class Comment(models.Model):

post = models.ForeignKey(Post, related\_name='comments', on\_delete=models.CASCADE)

content = models.TextField()

created\_date = models.DateTimeField(auto\_now\_add=True)

```

3 - Question

The homepage loads slowly when there are many posts and comments. Identify and explain the issue in the code.

4 - Explanation

The current implementation results in an N+1 query problem. For each post, a separate query is made to count the comments, which is highly inefficient and slows down the page load significantly when there are many posts and comments.

5 - Answer

```python

from django.shortcuts import render

from myapp.models import Post

from django.db.models import Count

def homepage(request):

posts = Post.objects.annotate(comment\_count=Count('comments')).all()

return render(request, 'homepage.html', {'posts': posts})

# models.py

from django.db import models

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

class Comment(models.Model):

post = models.ForeignKey(Post, related\_name='comments', on\_delete=models.CASCADE)

content = models.TextField()

created\_date = models.DateTimeField(auto\_now\_add=True)

```

In the corrected version, the `annotate` method is used to add a `comment\_count` attribute to each post in a single query, eliminating the N+1 query problem and significantly improving performance.

These questions test various advanced Django topics and combinations, providing a comprehensive challenge for senior developers.

# Flask

## GPT link

<https://chatgpt.com/share/729462e6-a50f-479f-aa2a-5d6226a19ed8>

## Part 1 - Questions (gpt4)

### Question 1

#### 0 - Topics

- Blueprints for Modular Applications

- Application Factories

#### 1 - Scenario

You are working on a large-scale web application and decide to use Flask Blueprints to modularize your application. You also want to implement the application factory pattern to create instances of your app with different configurations.

#### 2 - Code

```python

from flask import Flask, Blueprint, render\_template

# Define a blueprint

main = Blueprint('main', \_\_name\_\_)

@main.route('/')

def index():

return render\_template('index.html')

def create\_app():

app = Flask(\_\_name\_\_)

# Configuration settings

app.config['DEBUG'] = True

# Register blueprint

app.register\_blueprint(main, url\_prefix='/main')

return app

if \_\_name\_\_ == '\_\_main\_\_':

app = create\_app()

app.run()

```

#### 3 - Question

There is an issue with the way the blueprint and the application factory pattern are used in the above code. Identify and explain the issue.

#### 4 - Explanation

The issue is that the template folder is not specified for the Blueprint, and the URL prefix for the Blueprint is set incorrectly. Without specifying the template folder for the Blueprint, Flask won't be able to locate the templates associated with it. Additionally, the URL prefix should typically be more descriptive to avoid conflicts.

#### 5 - Answer

```python

from flask import Flask, Blueprint, render\_template

# Define a blueprint with a template folder

main = Blueprint('main', \_\_name\_\_, template\_folder='templates')

@main.route('/')

def index():

return render\_template('index.html')

def create\_app():

app = Flask(\_\_name\_\_)

# Configuration settings

app.config['DEBUG'] = True

# Register blueprint with a proper URL prefix

app.register\_blueprint(main, url\_prefix='/main')

return app

if \_\_name\_\_ == '\_\_main\_\_':

app = create\_app()

app.run()

```

---

### Question 2

#### 0 - Topics

- SQLAlchemy for Database Management

- Handling Database Migrations

#### 1 - Scenario

You are developing a web application using Flask with SQLAlchemy for database interactions. You want to ensure the database schema evolves smoothly using Flask-Migrate for migrations.

#### 2 - Code

```python

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)

migrate = Migrate(app, db)

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

The above code does not handle database migrations properly. Identify and explain the issue in the code.

#### 4 - Explanation

The problem is that the `Migrate` instance is not being properly integrated with Flask-Migrate's CLI commands. This setup will not allow you to run migration commands from the command line.

#### 5 - Answer

```python

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate, upgrade

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)

migrate = Migrate(app, db)

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

@app.cli.command()

def deploy():

"""Run deployment tasks."""

# Run migration scripts

upgrade()

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 3

#### 0 - Topics

- Celery for Asynchronous Tasks

- Redis Queues

#### 1 - Scenario

You are building a web application where some tasks need to be processed asynchronously to avoid blocking the main application thread. You decide to use Celery with Redis as the message broker.

#### 2 - Code

```python

from flask import Flask

from celery import Celery

app = Flask(\_\_name\_\_)

app.config['CELERY\_BROKER\_URL'] = 'redis://localhost:6379/0'

app.config['CELERY\_RESULT\_BACKEND'] = 'redis://localhost:6379/0'

def make\_celery(app):

celery = Celery(app.import\_name, broker=app.config['CELERY\_BROKER\_URL'])

celery.conf.update(app.config)

return celery

celery = make\_celery(app)

@celery.task

def add\_together(a, b):

return a + b

@app.route('/add/<int:a>/<int:b>')

def add(a, b):

result = add\_together.delay(a, b)

return f'Task result: {result.get()}'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

There is an issue with the asynchronous task processing in the above code. Identify and explain the issue.

#### 4 - Explanation

The problem is with the call to `result.get()` inside the route handler. This call will block the main thread until the task completes, defeating the purpose of using Celery for asynchronous task processing.

#### 5 - Answer

```python

from flask import Flask, jsonify

from celery import Celery

app = Flask(\_\_name\_\_)

app.config['CELERY\_BROKER\_URL'] = 'redis://localhost:6379/0'

app.config['CELERY\_RESULT\_BACKEND'] = 'redis://localhost:6379/0'

def make\_celery(app):

celery = Celery(app.import\_name, broker=app.config['CELERY\_BROKER\_URL'])

celery.conf.update(app.config)

return celery

celery = make\_celery(app)

@celery.task

def add\_together(a, b):

return a + b

@app.route('/add/<int:a>/<int:b>')

def add(a, b):

result = add\_together.delay(a, b)

return jsonify({'task\_id': result.id})

@app.route('/result/<task\_id>')

def result(task\_id):

task = add\_together.AsyncResult(task\_id)

if task.state == 'PENDING':

response = {'state': task.state, 'status': 'Pending...'}

elif task.state != 'FAILURE':

response = {'state': task.state, 'result': task.result}

else:

response = {'state': task.state, 'status': str(task.info)}

return jsonify(response)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

In this corrected version, the `add` route initiates the task and returns a task ID, while a new `result` route allows clients to check the status of the task asynchronously.

### Question 4

#### 0 - Topics

- Handling Large File Uploads

- Streaming Data

#### 1 - Scenario

You are developing a web application that allows users to upload large files. You need to ensure that the server handles the uploads efficiently and without running out of memory.

#### 2 - Code

```python

from flask import Flask, request, Response

app = Flask(\_\_name\_\_)

@app.route('/upload', methods=['POST'])

def upload\_file():

file = request.files['file']

data = file.read()

# Simulate processing the data

process\_data(data)

return 'File uploaded successfully'

def process\_data(data):

# Dummy function to simulate data processing

pass

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

There are issues with handling large file uploads in the above code. Identify and explain these issues.

#### 4 - Explanation

Reading the entire file into memory at once with `file.read()` can cause the server to run out of memory for large files. Instead, the file should be processed in chunks to handle large uploads efficiently.

#### 5 - Answer

```python

from flask import Flask, request, Response

app = Flask(\_\_name\_\_)

@app.route('/upload', methods=['POST'])

def upload\_file():

file = request.files['file']

def generate():

for chunk in iter(lambda: file.read(4096), b''):

process\_data(chunk)

yield chunk

return Response(generate(), content\_type='text/plain')

def process\_data(chunk):

# Dummy function to simulate data processing

pass

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 5

#### 0 - Topics

- Flask-Security for Role-Based Access Control

- Handling User Authentication and Authorization

#### 1 - Scenario

You are creating a web application that requires user authentication and role-based access control. You decide to use Flask-Security for managing users and roles.

#### 2 - Code

```python

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from flask\_security import Security, SQLAlchemyUserDatastore, UserMixin, RoleMixin, login\_required, roles\_required

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///security.db'

app.config['SECURITY\_REGISTERABLE'] = True

app.config['SECURITY\_PASSWORD\_SALT'] = 'salty'

db = SQLAlchemy(app)

roles\_users = db.Table('roles\_users',

db.Column('user\_id', db.Integer(), db.ForeignKey('user.id')),

db.Column('role\_id', db.Integer(), db.ForeignKey('role.id'))

)

class Role(db.Model, RoleMixin):

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(80), unique=True)

class User(db.Model, UserMixin):

id = db.Column(db.Integer(), primary\_key=True)

email = db.Column(db.String(255), unique=True)

password = db.Column(db.String(255))

active = db.Column(db.Boolean())

roles = db.relationship('Role', secondary=roles\_users, backref=db.backref('users', lazy='dynamic'))

user\_datastore = SQLAlchemyUserDatastore(db, User, Role)

security = Security(app, user\_datastore)

@app.route('/admin')

@roles\_required('admin')

def admin\_page():

return 'Welcome to the admin page!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

There is an issue with the role-based access control implementation in the above code. Identify and explain the issue.

#### 4 - Explanation

The code does not initialize the database with the necessary user and role data. Without creating roles and users, the access control checks will fail because no user will have the 'admin' role.

#### 5 - Answer

```python

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

from flask\_security import Security, SQLAlchemyUserDatastore, UserMixin, RoleMixin, login\_required, roles\_required

from flask\_security.utils import encrypt\_password

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///security.db'

app.config['SECURITY\_REGISTERABLE'] = True

app.config['SECURITY\_PASSWORD\_SALT'] = 'salty'

db = SQLAlchemy(app)

roles\_users = db.Table('roles\_users',

db.Column('user\_id', db.Integer(), db.ForeignKey('user.id')),

db.Column('role\_id', db.Integer(), db.ForeignKey('role.id'))

)

class Role(db.Model, RoleMixin):

id = db.Column(db.Integer(), primary\_key=True)

name = db.Column(db.String(80), unique=True)

class User(db.Model, UserMixin):

id = db.Column(db.Integer(), primary\_key=True)

email = db.Column(db.String(255), unique=True)

password = db.Column(db.String(255))

active = db.Column(db.Boolean())

roles = db.relationship('Role', secondary=roles\_users, backref=db.backref('users', lazy='dynamic'))

user\_datastore = SQLAlchemyUserDatastore(db, User, Role)

security = Security(app, user\_datastore)

@app.before\_first\_request

def create\_user():

db.create\_all()

if not user\_datastore.find\_role('admin'):

user\_datastore.create\_role(name='admin')

if not user\_datastore.get\_user('admin@example.com'):

user\_datastore.create\_user(email='admin@example.com', password=encrypt\_password('password'), roles=['admin'])

db.session.commit()

@app.route('/admin')

@roles\_required('admin')

def admin\_page():

return 'Welcome to the admin page!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 6

#### 0 - Topics

- Flask-RESTPlus for API Documentation

- Namespace Management

#### 1 - Scenario

You are developing a RESTful API and want to use Flask-RESTPlus to provide built-in API documentation and better manage your endpoints using namespaces.

#### 2 - Code

```python

from flask import Flask

from flask\_restplus import Api, Resource

app = Flask(\_\_name\_\_)

api = Api(app)

@api.route('/hello')

class HelloWorld(Resource):

def get(self):

return {'hello': 'world'}

@api.route('/goodbye')

class Goodbye(Resource):

def get(self):

return {'goodbye': 'world'}

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the Flask-RESTPlus implementation in the above code.

#### 4 - Explanation

The code does not use namespaces, which can lead to poorly organized APIs and potential route conflicts in larger applications. Additionally, proper documentation annotations are missing, which reduces the effectiveness of the automatic documentation feature.

#### 5 - Answer

```python

from flask import Flask

from flask\_restplus import Api, Resource, Namespace

app = Flask(\_\_name\_\_)

api = Api(app)

hello\_ns = Namespace('hello', description='Hello related operations')

goodbye\_ns = Namespace('goodbye', description='Goodbye related operations')

@hello\_ns.route('/')

class HelloWorld(Resource):

@hello\_ns.doc('get\_hello')

def get(self):

return {'hello': 'world'}

@goodbye\_ns.route('/')

class Goodbye(Resource):

@goodbye\_ns.doc('get\_goodbye')

def get(self):

return {'goodbye': 'world'}

api.add\_namespace(hello\_ns)

api.add\_namespace(goodbye\_ns)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 7

#### 0 - Topics

- Asynchronous Programming with Flask and AsyncIO

- Concurrent Requests Handling

#### 1 - Scenario

You want to enhance your Flask application to handle asynchronous tasks using AsyncIO to improve its performance under heavy load.

#### 2 - Code

```python

from flask import Flask, jsonify

import asyncio

app = Flask(\_\_name\_\_)

@app.route('/data')

async def get\_data():

await asyncio.sleep(2) # Simulate an asynchronous task

data = {'value': 'This is async data'}

return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the asynchronous handling implementation in the above code.

#### 4 - Explanation

Flask's default server does not support asynchronous endpoints out-of-the-box. Using async functions with Flask requires an ASGI server such as Uvicorn or Hypercorn to handle asynchronous requests.

#### 5 - Answer

```python

from flask import Flask, jsonify

import asyncio

import os

from hypercorn.asyncio import serve

from hypercorn.config import Config

app = Flask(\_\_name\_\_)

@app.route('/data')

async def get\_data():

await asyncio.sleep(2) # Simulate an asynchronous task

data = {'value': 'This is async data'}

return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':

config = Config()

config.bind = ["0.0.0.0:5000"]

asyncio.run(serve(app, config))

```

These questions focus on sophisticated topics like efficient large file handling, role-based access control, API documentation with Flask-RESTPlus, and asynchronous programming with Flask, ensuring that they are challenging for senior developers.

### Question 8

#### 0 - Topics

- OAuth2 Authentication with Flask-Dance

- External API Integration

#### 1 - Scenario

You are building a Flask application that requires OAuth2 authentication to allow users to log in with their Google accounts. You decide to use Flask-Dance for this purpose and need to integrate it properly.

#### 2 - Code

```python

from flask import Flask, redirect, url\_for

from flask\_dance.contrib.google import make\_google\_blueprint, google

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

google\_bp = make\_google\_blueprint(client\_id='my-client-id', client\_secret='my-client-secret', redirect\_to='google\_login')

app.register\_blueprint(google\_bp, url\_prefix='/login')

@app.route('/')

def index():

return 'Welcome to the Flask app!'

@app.route('/google')

def google\_login():

if not google.authorized:

return redirect(url\_for('google.login'))

resp = google.get('/plus/v1/people/me')

assert resp.ok, resp.text

return 'You are connected as {0}'.format(resp.json()['displayName'])

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the OAuth2 authentication implementation in the above code.

#### 4 - Explanation

The code does not handle storing the OAuth token securely or checking for errors during the OAuth flow. Additionally, the redirect URL configuration might be incorrect, and error handling for the Google API response is missing.

#### 5 - Answer

```python

from flask import Flask, redirect, url\_for, session

from flask\_dance.contrib.google import make\_google\_blueprint, google

from flask\_dance.consumer.storage.sqla import SQLAlchemyStorage

from flask\_sqlalchemy import SQLAlchemy

from flask\_dance.consumer import oauth\_authorized

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///oauth.db'

db = SQLAlchemy(app)

class OAuth(OAuthConsumerMixin, db.Model):

user\_id = db.Column(db.Integer, db.ForeignKey('user.id'))

user = db.relationship('User')

google\_bp = make\_google\_blueprint(client\_id='my-client-id', client\_secret='my-client-secret', storage=SQLAlchemyStorage(OAuth, db.session, user=current\_user))

app.register\_blueprint(google\_bp, url\_prefix='/login')

@app.route('/')

def index():

return 'Welcome to the Flask app!'

@app.route('/google')

def google\_login():

if not google.authorized:

return redirect(url\_for('google.login'))

resp = google.get('/plus/v1/people/me')

if not resp.ok:

return 'Failed to fetch user info', 400

return 'You are connected as {0}'.format(resp.json()['displayName'])

@oauth\_authorized.connect\_via(google\_bp)

def google\_logged\_in(blueprint, token):

# Store the token securely

session['google\_token'] = token

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 9

#### 0 - Topics

- Flask-RESTful Advanced Features

- Custom Error Handling

#### 1 - Scenario

You are developing a RESTful API with Flask-RESTful and want to implement custom error handling for your endpoints.

#### 2 - Code

```python

from flask import Flask, request

from flask\_restful import Resource, Api

app = Flask(\_\_name\_\_)

api = Api(app)

class CustomError(Exception):

pass

class HelloWorld(Resource):

def get(self):

if 'error' in request.args:

raise CustomError('This is a custom error')

return {'hello': 'world'}

api.add\_resource(HelloWorld, '/')

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the custom error handling implementation in the above code.

#### 4 - Explanation

The code does not include an error handler for the custom error, which means Flask will not know how to handle it, resulting in a generic 500 Internal Server Error response. Custom error handlers should be registered to handle specific exceptions.

#### 5 - Answer

```python

from flask import Flask, request, jsonify

from flask\_restful import Resource, Api

from werkzeug.exceptions import HTTPException

app = Flask(\_\_name\_\_)

api = Api(app)

class CustomError(Exception):

pass

@app.errorhandler(CustomError)

def handle\_custom\_error(error):

response = jsonify({'message': str(error)})

response.status\_code = 400

return response

class HelloWorld(Resource):

def get(self):

if 'error' in request.args:

raise CustomError('This is a custom error')

return {'hello': 'world'}

api.add\_resource(HelloWorld, '/')

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 10

#### 0 - Topics

- Flask-Mail for Sending Emails

- Asynchronous Email Sending

#### 1 - Scenario

You are building a Flask application that needs to send email notifications. You want to use Flask-Mail and send the emails asynchronously to avoid blocking the main application thread.

#### 2 - Code

```python

from flask import Flask, render\_template

from flask\_mail import Mail, Message

app = Flask(\_\_name\_\_)

app.config['MAIL\_SERVER'] = 'smtp.example.com'

app.config['MAIL\_PORT'] = 587

app.config['MAIL\_USE\_TLS'] = True

app.config['MAIL\_USERNAME'] = 'your-email@example.com'

app.config['MAIL\_PASSWORD'] = 'your-password'

mail = Mail(app)

@app.route('/send-email')

def send\_email():

msg = Message('Hello', sender='your-email@example.com', recipients=['recipient@example.com'])

msg.body = 'This is a test email'

mail.send(msg)

return 'Email sent!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the email sending implementation in the above code.

#### 4 - Explanation

The code sends the email synchronously, which can block the main thread if the email server is slow or unresponsive. It should send emails asynchronously to improve performance and user experience.

#### 5 - Answer

```python

from flask import Flask, render\_template

from flask\_mail import Mail, Message

from threading import Thread

app = Flask(\_\_name\_\_)

app.config['MAIL\_SERVER'] = 'smtp.example.com'

app.config['MAIL\_PORT'] = 587

app.config['MAIL\_USE\_TLS'] = True

app.config['MAIL\_USERNAME'] = 'your-email@example.com'

app.config['MAIL\_PASSWORD'] = 'your-password'

mail = Mail(app)

def send\_async\_email(app, msg):

with app.app\_context():

mail.send(msg)

@app.route('/send-email')

def send\_email():

msg = Message('Hello', sender='your-email@example.com', recipients=['recipient@example.com'])

msg.body = 'This is a test email'

thr = Thread(target=send\_async\_email, args=[app, msg])

thr.start()

return 'Email sent!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 11

#### 0 - Topics

- Flask-SQLAlchemy Advanced Querying

- SQL Injection Prevention

#### 1 - Scenario

You are developing an application with Flask-SQLAlchemy and want to ensure your query methods are safe from SQL injection attacks.

#### 2 - Code

```python

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///test.db'

db = SQLAlchemy(app)

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

@app.route('/user/<username>')

def get\_user(username):

user = db.engine.execute(f"SELECT \* FROM user WHERE username = '{username}'").fetchone()

if user:

return jsonify(username=user['username'], email=user['email'])

return 'User not found', 404

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the SQL query implementation in the above code.

#### 4 - Explanation

The code uses raw SQL with string interpolation, making it vulnerable to SQL injection attacks. The query should use parameterized queries to prevent SQL injection.

#### 5 - Answer

```python

from flask import Flask, request, jsonify

from flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///test.db'

db = SQLAlchemy(app)

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

@app.route('/user/<username>')

def get\_user(username):

user = User.query.filter\_by(username=username).first()

if user:

return jsonify(username=user.username, email=user.email)

return 'User not found', 404

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

These additional questions cover advanced concepts like OAuth2 authentication, custom error handling, asynchronous email sending, and SQL injection prevention. These should challenge even senior developers and test their deep understanding of Flask and associated libraries.

### Question 12

#### 0 - Topics

- Flask-Login for User Authentication

- Session Management

#### 1 - Scenario

You are building an application that requires user authentication using Flask-Login. You need to manage user sessions correctly and securely.

#### 2 - Code

```python

from flask import Flask, render\_template, redirect, url\_for, request

from flask\_login import LoginManager, UserMixin, login\_user, login\_required, logout\_user, current\_user

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

login\_manager = LoginManager()

login\_manager.init\_app(app)

class User(UserMixin):

def \_\_init\_\_(self, id, username):

self.id = id

self.username = username

users = {1: User(1, 'user1')}

@login\_manager.user\_loader

def load\_user(user\_id):

return users.get(int(user\_id))

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

username = request.form['username']

user = next((u for u in users.values() if u.username == username), None)

if user:

login\_user(user)

return redirect(url\_for('protected'))

return 'Invalid credentials'

return render\_template('login.html')

@app.route('/protected')

@login\_required

def protected():

return f'Logged in as: {current\_user.username}'

@app.route('/logout')

@login\_required

def logout():

logout\_user()

return redirect(url\_for('login'))

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the user authentication and session management in the above code.

#### 4 - Explanation

The code does not properly handle the case where the user is already logged in and tries to access the login page. Additionally, it lacks proper session protection and security measures such as `remember\_me` functionality.

#### 5 - Answer

```python

from flask import Flask, render\_template, redirect, url\_for, request

from flask\_login import LoginManager, UserMixin, login\_user, login\_required, logout\_user, current\_user

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

login\_manager = LoginManager()

login\_manager.init\_app(app)

login\_manager.login\_view = 'login'

class User(UserMixin):

def \_\_init\_\_(self, id, username):

self.id = id

self.username = username

users = {1: User(1, 'user1')}

@login\_manager.user\_loader

def load\_user(user\_id):

return users.get(int(user\_id))

@app.route('/login', methods=['GET', 'POST'])

def login():

if current\_user.is\_authenticated:

return redirect(url\_for('protected'))

if request.method == 'POST':

username = request.form['username']

user = next((u for u in users.values() if u.username == username), None)

if user:

login\_user(user, remember=True)

return redirect(url\_for('protected'))

return 'Invalid credentials'

return render\_template('login.html')

@app.route('/protected')

@login\_required

def protected():

return f'Logged in as: {current\_user.username}'

@app.route('/logout')

@login\_required

def logout():

logout\_user()

return redirect(url\_for('login'))

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 13

#### 0 - Topics

- Flask-CORS for Cross-Origin Resource Sharing

- Security

#### 1 - Scenario

You are building a Flask API that needs to be accessed by a web application hosted on a different domain. You decide to use Flask-CORS to handle cross-origin resource sharing.

#### 2 - Code

```python

from flask import Flask, jsonify

from flask\_cors import CORS

app = Flask(\_\_name\_\_)

CORS(app)

@app.route('/data')

def get\_data():

data = {'value': 'This is CORS-enabled data'}

return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the CORS implementation in the above code.

#### 4 - Explanation

The code enables CORS for all routes and all origins by default, which might pose security risks. It should restrict CORS to specific domains and methods as needed.

#### 5 - Answer

```python

from flask import Flask, jsonify

from flask\_cors import CORS

app = Flask(\_\_name\_\_)

CORS(app, resources={r"/data": {"origins": "http://example.com"}})

@app.route('/data')

def get\_data():

data = {'value': 'This is CORS-enabled data'}

return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 14

#### 0 - Topics

- Flask-Admin for Admin Interfaces

- Database Management

#### 1 - Scenario

You are building an admin interface for your Flask application using Flask-Admin. You need to manage user records stored in a database.

#### 2 - Code

```python

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_admin import Admin

from flask\_admin.contrib.sqla import ModelView

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///test.db'

db = SQLAlchemy(app)

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

admin = Admin(app)

admin.add\_view(ModelView(User, db.session))

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the admin interface implementation in the above code.

#### 4 - Explanation

The code does not provide any authentication or access control for the admin interface, allowing anyone to access it and potentially modify data. It should be secured to restrict access to authorized users only.

#### 5 - Answer

```python

from flask import Flask, redirect, url\_for, request

from flask\_sqlalchemy import SQLAlchemy

from flask\_admin import Admin, AdminIndexView, expose

from flask\_admin.contrib.sqla import ModelView

from flask\_login import LoginManager, UserMixin, login\_user, login\_required, current\_user

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'supersecretkey'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///test.db'

db = SQLAlchemy(app)

login\_manager = LoginManager(app)

class User(db.Model, UserMixin):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

email = db.Column(db.String(120), unique=True, nullable=False)

class MyModelView(ModelView):

def is\_accessible(self):

return current\_user.is\_authenticated

class MyAdminIndexView(AdminIndexView):

@expose('/')

def index(self):

if not current\_user.is\_authenticated:

return redirect(url\_for('login'))

return super(MyAdminIndexView, self).index()

admin = Admin(app, index\_view=MyAdminIndexView())

admin.add\_view(MyModelView(User, db.session))

@login\_manager.user\_loader

def load\_user(user\_id):

return User.query.get(int(user\_id))

@app.route('/login', methods=['GET', 'POST'])

def login():

if request.method == 'POST':

username = request.form['username']

user = User.query.filter\_by(username=username).first()

if user:

login\_user(user)

return redirect(url\_for('admin.index'))

return render\_template('login.html')

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---

### Question 15

#### 0 - Topics

- Flask-Migrate for Database Migrations

- Handling Multiple Database Schemas

#### 1 - Scenario

You are developing a Flask application that needs to manage multiple database schemas using Flask-Migrate.

#### 2 - Code

```python

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)

migrate = Migrate(app, db)

class User(db.Model):

id = db.Column(db.Integer, primary\_key=True)

username = db.Column(db.String(80), unique=True, nullable=False)

class Product(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String(80), unique=True, nullable=False)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the issues with the database migration implementation in the above code.

#### 4 - Explanation

The code does not provide a way to manage multiple database schemas effectively. Flask-Migrate can handle only one `db` instance, and the models should be structured to support multiple schemas.

#### 5 - Answer

```python

from flask import Flask

from flask\_sqlalchemy import SQLAlchemy

from flask\_migrate import Migrate

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///app.db'

db = SQLAlchemy(app)

class Base(db.Model):

\_\_abstract\_\_ = True

id = db.Column(db.Integer, primary\_key=True)

class User(Base):

username = db.Column(db.String(80), unique=True, nullable=False)

class Product(Base):

name = db.Column(db.String(80), unique=True, nullable=False)

migrate = Migrate(app,

db)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

In this corrected version, the `Base` class is used as an abstract base class to share the primary key column among multiple schemas, making it easier to manage migrations for multiple models.

---

These additional questions focus on more advanced Flask concepts like user authentication, cross-origin resource sharing, admin interface security, database migrations, and handling multiple database schemas. These should be challenging and educational for senior developers.

### Question 16

#### 0 - Topics

- Caching with Flask-Caching

- Performance Optimization

#### 1 - Scenario

You want to optimize your Flask application's performance by implementing caching for frequently accessed data using Flask-Caching.

#### 2 - Code

```python

from flask import Flask, jsonify

from flask\_caching import Cache

app = Flask(\_\_name\_\_)

app.config['CACHE\_TYPE'] = 'simple'

cache = Cache(app)

@app.route('/data')

@cache.cached(timeout=60)

def get\_data():

# Simulate a time-consuming operation

import time

time.sleep(5)

data = {'value': 'This is cached data'}

return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

#### 3 - Question

Identify and explain the potential issues with the caching implementation in the above code.

#### 4 - Explanation

The main issue is that the `cache.cached` decorator should be properly configured with a cache key. Additionally, the cache configuration should be checked to ensure it's set up correctly for the deployment environment.

#### 5 - Answer

```python

from flask import Flask, jsonify

from flask\_caching import Cache

app = Flask(\_\_name\_\_)

app.config['CACHE\_TYPE'] = 'simple'

app.config['CACHE\_DEFAULT\_TIMEOUT'] = 300 # Set a default timeout

cache = Cache(app)

@app.route('/data')

@cache.cached(timeout=60, key\_prefix='data\_view')

def get\_data():

# Simulate a time-consuming operation

import time

time.sleep(5)

data = {'value': 'This is cached data'}

return jsonify(data)

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

```

---