File handling is a critical skill for data engineers, as files are often used to store and process large datasets. Here are some **scenarios and problems** where file handling plays a key role, along with potential tasks you can practice:

---

1. Data Cleaning and Transformation

- **Scenario**: You are provided a large text or CSV file containing raw data that needs to be cleaned and transformed.
    - Read the file line-by-line and filter out rows that do not meet certain criteria (e.g., missing values).
    - Split a large file into smaller files based on conditions (e.g., split data by regions or dates).
    - Merge multiple text or CSV files into a single file for processing.
    - Convert data formats, such as CSV to JSON or XML.

Practice:

- Read a CSV file with missing values, fill the missing data, and save the cleaned file.
- Split a large file into smaller files of 1000 lines each.

---

2. Log Analysis

- **Scenario**: Analyze server logs stored in text files to extract useful insights.
    - Parse log files to extract timestamps, IP addresses, error codes, etc.
    - Count the frequency of different log levels (INFO, WARNING, ERROR).
    - Identify the most frequent users or IPs accessing the system.

Practice:

- Write a program to count the number of ERROR log entries in a file.
- Find the most common IP address from a web server log.

---

3. Data Aggregation

- **Scenario**: Perform calculations on numerical data stored in files.
    - Sum up values from multiple files (e.g., daily sales records).
    - Compute averages or other statistics from data files.
    - Generate summary reports from large datasets.

Practice:

- Write a program to read multiple sales CSV files and calculate the total revenue.
- Generate a report with average and total sales per product.

---

4. Configuration and Metadata Handling

- **Scenario**: Manage configuration files or metadata for pipelines.
    - Read and update configuration stored in text, JSON, or XML files.
    - Validate the format and content of configuration files.
    - Convert configuration files between different formats.

Practice:

- Create a program to read a JSON configuration file, modify a specific value, and save it back.

- Validate a JSON configuration file against a predefined schema.

---

5. Batch Processing

- **Scenario**: Automate the processing of large batches of files.
    - Process each file in a directory (e.g., compress, encrypt, or convert files).
    - Rename files based on their content or metadata.
    - Archive old files or move processed files to specific folders.

Practice:

- Write a script to read all files in a folder and compress them into a ZIP file.
- Automate renaming files based on their modification date.

---

6. Data Ingestion Pipelines

- **Scenario**: Create a pipeline to ingest and store data from files into a database.
    - Parse files and insert data into SQL or NoSQL databases.
    - Handle errors during ingestion and log problematic rows.
    - Implement data validation while reading files.

Practice:

- Write a script to read a CSV file and insert its rows into a PostgreSQL table.
- Validate the content of each row before storing it in a database.

---

7. Real-Time Data Streams

- **Scenario**: Process real-time logs or data files as they arrive.
    - Continuously monitor a directory for new files and process them.
    - Stream data from files to other systems (e.g., Kafka, databases).

Practice:

- Write a script to monitor a directory for new log files and process them immediately.
- Simulate streaming by reading lines from a file and sending them to a mock Kafka topic.

---

8. Report Generation

- **Scenario**: Generate and write reports based on data.
    - Read data from a file and create a PDF or Excel report.
    - Write summary statistics or visualizations into a text or CSV file.

Practice:

- Read a CSV file of sales data and generate a PDF report summarizing total sales per region.
- Write a program to create an Excel file with multiple sheets, each representing a different data category.

---

9. Web Scraping with File Handling

- **Scenario**: Store scraped data into files for further analysis.
    - Save raw HTML or JSON responses from web scraping.
    - Append scraped data to a file in batches.

Practice:

- Scrape data from a website and save it to a CSV file.
- Process the saved data to remove duplicates and format it for analysis.

---

10. Historical Data Archiving

- **Scenario**: Archive old data files for compliance or backup.
    - Compress and encrypt sensitive files before archiving.
    - Move files older than a certain date to an archive directory.

Practice:

- Write a program to find and archive files older than 30 days.
- Automate encryption of sensitive files before moving them to a backup folder.

---

11. Data Visualization

- **Scenario**: Visualize data stored in files.
    - Read CSV or JSON data and create visualizations (e.g., bar charts, pie charts).
    - Save visualizations to image files or PDFs.

Practice:

- Read sales data from a CSV file and create a bar chart saved as an image.
- Plot temperature data from a text file and save the plot as a PDF.

---

12. Error Handling and Logging

- **Scenario**: Log errors during data processing into a file.
    - Write errors to a log file for debugging.
    - Append success or failure status for each processed file.

Practice:

- Write a program to process files and log the results (success or error) to a log file.
- Create a system to retry processing failed files using the log file.

---

13. Data Comparison

- **Scenario**: Compare data from two files.
    - Identify differences between two files (e.g., mismatched records).
    - Check for duplicates or missing entries across files.

Practice:

- Write a program to compare two CSV files and output the differences to a new file.
- Identify and remove duplicate rows in a file.

---

14. Encoding and Decoding Files

- **Scenario**: Work with files in different encodings or formats.
    - Read and write files with specific encodings (e.g., UTF-8, ISO-8859-1).
    - Handle binary files (e.g., images, PDFs).

Practice:

- Write a script to read a file in one encoding and save it in another encoding.
- Process a binary file and extract specific information (e.g., metadata).

---

1. Data Cleaning and Transformation Tasks

Task 1: Read a CSV file with missing values, fill the missing data, and save the cleaned file

To address this, we'll use `pandas` for reading, cleaning, and writing the CSV file.

```
import pandas as pd

def clean_and_save_csv(input_file, output_file):
    # Read the CSV file
    df = pd.read_csv(input_file)

    # Fill missing values with a default value (e.g., 0 for numerical
columns, 'Unknown' for string columns)
    df.fillna({'column_name_1': 0, 'column_name_2': 'Unknown'},
inplace=True)  # Example fill strategy

    # Save the cleaned DataFrame to a new CSV file
    df.to_csv(output_file, index=False)

# Example usage
clean_and_save_csv('raw_data.csv', 'cleaned_data.csv')
```

Explanation:

- The `pandas.read_csv()` function reads the CSV file into a DataFrame.
- The `fillna()` method fills missing values, where specific columns can be filled with custom values.
- The cleaned data is saved to a new CSV file using `to_csv()`.

Task 2: Split a large file into smaller files of 1000 lines each

This can be done by reading the large file line by line, and splitting it into smaller files based on line count.

```
def split_large_file(input_file, lines_per_file=1000):
    with open(input_file, 'r') as file:
        # Read all lines
        lines = file.readlines()

        # Determine how many smaller files are needed
        total_lines = len(lines)
        num_files = total_lines // lines_per_file + (1 if total_lines
% lines_per_file else 0)
```

```
        for i in range(num_files):
            # Calculate the start and end indices for the lines to be
written to the current file
            start_idx = i * lines_per_file
            end_idx = min((i + 1) * lines_per_file, total_lines)
            output_file = f"part_{i + 1}.txt"  # Naming the output
file

            # Write the lines for the current file
            with open(output_file, 'w') as out_file:
                out_file.writelines(lines[start_idx:end_idx])

# Example usage
split_large_file('large_file.txt', 1000)
```

Explanation:

- The file is read into memory line by line using `readlines()`.
- The number of smaller files needed is determined based on the total number of lines and the `lines_per_file` value.
- Each chunk of 1000 lines is written to a new file named `part_1.txt`, `part_2.txt`, and so on.

Task 3: Merge multiple text or CSV files into a single file for processing

You can use `pandas` to merge CSV files or the built-in Python functionality for text files.

```
import pandas as pd
import glob

def merge_csv_files(input_folder, output_file):
    # Use glob to get all CSV files in the specified folder
    files = glob.glob(input_folder + "/*.csv")

    # List to hold DataFrames
    dfs = []

    # Loop over files and append them to the list
    for file in files:
        df = pd.read_csv(file)
```

```
            dfs.append(df)

    # Concatenate all DataFrames
    merged_df = pd.concat(dfs, ignore_index=True)

    # Save the merged DataFrame to a new CSV file
    merged_df.to_csv(output_file, index=False)

# Example usage
merge_csv_files('csv_folder', 'merged_data.csv')
```

Explanation:

- `glob.glob()` is used to get all files in the folder that match a pattern (e.g., all CSV files).
- Each file is read into a DataFrame and added to a list.
- `pandas.concat()` is used to combine all DataFrames into one, and the result is saved to a new CSV file.

Task 4: Convert CSV to JSON format
You can convert CSV data to JSON using `pandas`.

```
import pandas as pd

def csv_to_json(input_file, output_file):
    # Read the CSV file into a DataFrame
    df = pd.read_csv(input_file)

    # Convert the DataFrame to JSON and save it to a file
    df.to_json(output_file, orient='records', lines=True)

# Example usage
csv_to_json('data.csv', 'data.json')
```

Explanation:

- `pd.read_csv()` reads the CSV file into a DataFrame.
- `to_json()` converts the DataFrame to JSON format, where `orient='records'` outputs a list of dictionaries (one per row).
- The result is saved to a JSON file.

Practice Scenarios:

1. **Read a CSV file with missing values and fill them**: You can practice this by creating a sample CSV file with missing data and applying the code to clean it.
2. **Split a large file into smaller files**: You can test this by using a large text file or CSV and running the splitting function.

These tasks are common in data processing, and by practicing them, you'll become more proficient at handling real-world data cleaning and transformation scenarios.

Log Analysis Tasks

Task 1: Count the number of ERROR log entries in a file

You can use regular expressions to identify and count ERROR entries in a log file.

```
import re

def count_error_logs(file_path):
    error_count = 0
    with open(file_path, 'r') as file:
        for line in file:
            if "ERROR" in line:  # Checks if the log level "ERROR"
exists in the line
                error_count += 1
    print(f"Total number of ERROR logs: {error_count}")

# Example usage
count_error_logs('server_logs.txt')
```

Explanation:

- Opens the log file and reads it line by line.
- Checks if "ERROR" exists in each line, incrementing the count if it does.
- Prints the total count of ERROR log entries.

Task 2: Find the most common IP address from a web server log

To identify the most common IP address, you can parse the log file and count IP occurrences.

```
import re
from collections import Counter

def most_common_ip(file_path):
    ip_pattern = r'\b(?:[0-9]{1,3}\.){3}[0-9]{1,3}\b'  # Regular
expression to match IP addresses
    ip_addresses = []
```

```
    with open(file_path, 'r') as file:
        for line in file:
            match = re.search(ip_pattern, line)
            if match:
                ip_addresses.append(match.group(0))  # Extract the
matched IP address

    # Count occurrences of each IP and find the most common
    ip_counts = Counter(ip_addresses)
    most_common = ip_counts.most_common(1)[0]
    print(f"Most common IP: {most_common[0]} with {most_common[1]}
occurrences.")

# Example usage
most_common_ip('web_server_logs.txt')
```

Explanation:

- Uses a regular expression to match valid IPv4 addresses.
- Extracts and stores IP addresses in a list.
- Uses `collections.Counter` to count occurrences of each IP and identify the most frequent one.

Additional Insights:

Count the frequency of different log levels (INFO, WARNING, ERROR)

```
from collections import Counter

def log_level_frequency(file_path):
    log_levels = []

    with open(file_path, 'r') as file:
        for line in file:
            if "INFO" in line:
                log_levels.append("INFO")
            elif "WARNING" in line:
                log_levels.append("WARNING")
            elif "ERROR" in line:
                log_levels.append("ERROR")
```

```
    # Count occurrences of each log level
    level_counts = Counter(log_levels)
    for level, count in level_counts.items():
        print(f"{level}: {count}")

# Example usage
log_level_frequency('server_logs.txt')
```

Explanation:

- Checks each line for specific log levels (INFO, WARNING, ERROR).
- Appends the matched log level to a list.
- Uses `collections.Counter` to count the frequency of each log level.

Practice Scenarios:

1. **Count ERROR log entries**: Use a sample server log file with mixed log levels and run the program to count errors.
2. **Find the most common IP address**: Create or use a real web server log file with various IP addresses and test the IP extraction and counting logic.
3. **Count log level frequencies**: Modify the same server log file to include INFO, WARNING, and ERROR entries, and verify the output of the frequency count.

Data Aggregation Tasks

Task 1: Calculate total revenue from multiple sales CSV files

You can aggregate numerical data like revenue from multiple CSV files using `pandas`.

```
import pandas as pd
import glob

def calculate_total_revenue(folder_path):
    total_revenue = 0

    # Get all CSV files in the folder
    csv_files = glob.glob(folder_path + "/*.csv")

    # Iterate through each file
    for file in csv_files:
        # Read the CSV file
        df = pd.read_csv(file)

        # Assuming there is a 'revenue' column, sum its values
```

```
        if 'revenue' in df.columns:
            total_revenue += df['revenue'].sum()

    print(f"Total Revenue: ${total_revenue:.2f}")

# Example usage
calculate_total_revenue('sales_data')
```

Explanation:

- Uses `glob` to list all CSV files in the specified folder.
- Reads each CSV file into a `pandas` DataFrame and calculates the sum of the `revenue` column.
- Adds up revenue from all files to compute the total.

Task 2: Generate a report with average and total sales per product
To summarize sales data, you can group by product names and calculate metrics like total and average sales.

```
def generate_sales_report(input_file, output_file):
    # Read the sales data
    df = pd.read_csv(input_file)

    # Group by 'product' and calculate total and average sales
    report = df.groupby('product').agg(
        total_sales=('sales', 'sum'),
        average_sales=('sales', 'mean')
    ).reset_index()

    # Save the report to a CSV file
    report.to_csv(output_file, index=False)
    print("Sales report generated successfully!")

# Example usage
generate_sales_report('sales_data.csv', 'sales_report.csv')
```

Explanation:

- Groups data by the `product` column using `groupby()` and calculates the sum and mean of the `sales` column.
- The aggregated DataFrame is saved to a new CSV file for the summary report.

Additional Scenarios for Practice:

Sum up values from multiple files (e.g., daily sales records)

If each file contains the same structure but represents different time periods, aggregate data across all files:

```
def aggregate_sales_data(folder_path, output_file):
    # List all CSV files in the folder
    csv_files = glob.glob(folder_path + "/*.csv")

    # Concatenate all files into a single DataFrame
    all_data = pd.concat([pd.read_csv(file) for file in csv_files],
ignore_index=True)

    # Save the combined data to a new CSV file
    all_data.to_csv(output_file, index=False)
    print(f"Aggregated data saved to {output_file}")

# Example usage
aggregate_sales_data('daily_sales', 'aggregated_sales.csv')
```

Explanation:

- Combines multiple CSV files into one DataFrame.
- Allows further analysis on the unified dataset.

Practice Scenarios:

1. **Calculate total revenue from sales files**: Use multiple CSV files, each containing sales data with a `revenue` column, and calculate the total revenue.
2. **Generate product-level sales report**: Use a sample CSV file with `product` and `sales` columns, and create a report summarizing total and average sales per product.
3. **Aggregate daily sales records**: Combine multiple daily sales CSV files into a single file for more comprehensive analysis.

These exercises help you practice data aggregation techniques commonly used in real-world scenarios for reporting and analysis.

Configuration and Metadata Handling Tasks

Task 1: Read and Modify a JSON Configuration File

You can use Python's `json` module to read, modify, and save JSON configuration files.

```
import json

def modify_config(file_path, key, new_value):
```

```
    # Read the JSON configuration file
    with open(file_path, 'r') as file:
        config = json.load(file)

    # Modify the specified key
    if key in config:
        config[key] = new_value
        print(f"Updated '{key}' to '{new_value}'")
    else:
        print(f"Key '{key}' not found in the configuration.")

    # Save the updated configuration back to the file
    with open(file_path, 'w') as file:
        json.dump(config, file, indent=4)
    print("Configuration updated successfully!")

# Example usage
modify_config('config.json', 'api_key', 'new_api_key_value')
```

Explanation:

- Reads the JSON file and loads it into a Python dictionary.
- Checks if the specified key exists; if so, updates its value.
- Saves the modified dictionary back to the JSON file in a formatted way.

Task 2: Validate a JSON Configuration File Against a Schema

You can use the `jsonschema` library to validate a JSON file against a predefined schema.

```
import json
from jsonschema import validate, ValidationError

def validate_config(file_path, schema):
    # Read the JSON configuration file
    with open(file_path, 'r') as file:
        config = json.load(file)

    # Validate the JSON configuration against the schema
    try:
        validate(instance=config, schema=schema)
        print("Configuration is valid.")
```

```
    except ValidationError as e:
        print(f"Configuration is invalid: {e.message}")

# Example usage
schema = {
    "type": "object",
    "properties": {
        "api_key": {"type": "string"},
        "timeout": {"type": "number"},
        "enabled": {"type": "boolean"}
    },
    "required": ["api_key", "timeout", "enabled"]
}

validate_config('config.json', schema)
```

Explanation:

- Defines a JSON schema with rules for expected keys, types, and required fields.
- Loads the JSON file and checks its structure and content against the schema using `jsonschema.validate`.
- Handles validation errors and reports issues.

Additional Scenarios for Practice

Convert Configuration Files Between Formats

Convert a JSON configuration file into an XML format:

```
import json
import dicttoxml  # Install this library using `pip install dicttoxml`

def json_to_xml(json_file, xml_file):
    # Read the JSON file
    with open(json_file, 'r') as file:
        data = json.load(file)

    # Convert the dictionary to XML
    xml_data = dicttoxml.dicttoxml(data, custom_root='configuration',
attr_type=False)

    # Save the XML to a file
    with open(xml_file, 'wb') as file:
```

```
        file.write(xml_data)
    print(f"Configuration saved as XML: {xml_file}")

# Example usage
json_to_xml('config.json', 'config.xml')
```

Explanation:

- Reads a JSON file and converts it into a dictionary.
- Uses the `dicttoxml` library to transform the dictionary into XML format.
- Saves the XML content to a file.

Practice Scenarios:

1. **Modify a configuration file**: Create a sample `config.json` file and update specific fields using the provided program.
2. **Validate JSON configuration**: Write a JSON schema to validate a sample `config.json` file and test with valid/invalid configurations.
3. **Convert JSON to XML**: Practice converting a JSON configuration file into XML format for compatibility with different systems.

Batch Processing Tasks

Task 1: Compress All Files in a Folder into a ZIP File

You can use Python's `zipfile` module to compress multiple files into a ZIP archive.

```
import os
import zipfile

def compress_files(folder_path, zip_file_name):
    # Create a ZIP file
    with zipfile.ZipFile(zip_file_name, 'w') as zipf:
        # Walk through the directory and add files to the ZIP
        for root, _, files in os.walk(folder_path):
            for file in files:
                file_path = os.path.join(root, file)
                zipf.write(file_path, arcname=file)
    print(f"All files compressed into {zip_file_name}")

# Example usage
compress_files('data_folder', 'archive.zip')
```

Explanation:

- Uses `os.walk` to iterate through all files in the specified folder.
- Adds each file to the ZIP archive while maintaining the relative path using `arcname`.

Task 2: Rename Files Based on Their Modification Date
You can rename files in a folder using their last modification timestamp.

```
import os
import time

def rename_files_by_date(folder_path):
    for file_name in os.listdir(folder_path):
        file_path = os.path.join(folder_path, file_name)
        if os.path.isfile(file_path):
            # Get the modification timestamp
            mod_time = os.path.getmtime(file_path)
            formatted_time = time.strftime("%Y%m%d_%H%M%S",
time.localtime(mod_time))
            # Generate the new file name
            new_name = f"{formatted_time}_{file_name}"
            new_path = os.path.join(folder_path, new_name)
            os.rename(file_path, new_path)
            print(f"Renamed {file_name} to {new_name}")

# Example usage
rename_files_by_date('data_folder')
```

Explanation:
- Fetches the last modification timestamp of each file using `os.path.getmtime`.
- Formats the timestamp to a readable string using `time.strftime`.
- Renames the file by prepending the timestamp to the original file name.

Additional Scenarios for Practice

Move Processed Files to Specific Folders
Organize files into folders based on their type (e.g., `.txt`, `.csv`).

```
def organize_files_by_type(folder_path):
    for file_name in os.listdir(folder_path):
        file_path = os.path.join(folder_path, file_name)
        if os.path.isfile(file_path):
            # Extract file extension
```

```
            file_extension = os.path.splitext(file_name)[1][1:]
            # Create a folder for the file type if it doesn't exist
            type_folder = os.path.join(folder_path, file_extension)
            os.makedirs(type_folder, exist_ok=True)
            # Move the file to the folder
            new_path = os.path.join(type_folder, file_name)
            os.rename(file_path, new_path)
            print(f"Moved {file_name} to {type_folder}")

# Example usage
organize_files_by_type('data_folder')
```

Explanation:

- Extracts the file extension using `os.path.splitext`.
- Creates a folder named after the file type if it doesn't exist.
- Moves each file to its corresponding type-specific folder.

Practice Scenarios:

1. **Compress all files**: Create a sample folder with various files and compress them into a single ZIP file.
2. **Rename files by date**: Use a folder of test files and rename them based on their modification timestamps.
3. **Organize files by type**: Categorize a set of mixed files into folders by their extensions (e.g., `txt`, `csv`, `png`).
4. **Archive old files**: Write a script to move files older than a certain date to an archive folder.

Data Ingestion Pipelines Tasks

Task 1: Ingest Data from a CSV File into a PostgreSQL Table

You can use the `psycopg2` library to connect to PostgreSQL and insert data.

```
import csv
import psycopg2

def ingest_csv_to_postgres(csv_file, db_config, table_name):
    try:
        # Connect to the PostgreSQL database
        connection = psycopg2.connect(**db_config)
        cursor = connection.cursor()
```

```
        # Open and read the CSV file
        with open(csv_file, 'r') as file:
            reader = csv.reader(file)
            headers = next(reader)  # Get the header row

            # Insert rows into the table
            for row in reader:
                cursor.execute(
                    f"INSERT INTO {table_name} ({', '.join(headers)})
VALUES ({', '.join(['%s'] * len(row))})",
                    row
                )

        # Commit changes
        connection.commit()
        print(f"Data from {csv_file} successfully inserted into
{table_name}.")

    except Exception as e:
        print(f"Error: {e}")
    finally:
        if connection:
            cursor.close()
            connection.close()

# Example usage
db_config = {
    "dbname": "testdb",
    "user": "postgres",
    "password": "password",
    "host": "localhost",
    "port": 5432
}
ingest_csv_to_postgres('data.csv', db_config, 'products')
```

Explanation:

- Connects to a PostgreSQL database using `psycopg2`.
- Reads a CSV file and maps its headers to table columns.
- Uses parameterized queries to insert rows securely.

Task 2: Validate and Log Problematic Rows
Add validation to check the content of each row before insertion.

```
def validate_and_ingest(csv_file, db_config, table_name):
    try:
        connection = psycopg2.connect(**db_config)
        cursor = connection.cursor()
        with open(csv_file, 'r') as file:
            reader = csv.reader(file)
            headers = next(reader)
            for row in reader:
                if validate_row(row):
                    cursor.execute(
                        f"INSERT INTO {table_name} ({',
'.join(headers)}) VALUES ({', '.join(['%s'] * len(row))})",
                        row
                    )
                else:
                    log_problematic_row(row)

        connection.commit()
        print(f"Data from {csv_file} validated and inserted into
{table_name}.")

    except Exception as e:
        print(f"Error: {e}")
    finally:
        if connection:
            cursor.close()
            connection.close()

def validate_row(row):
    # Example: Check for missing or negative values
    return all(row) and all(float(value) >= 0 for value in row[1:])

def log_problematic_row(row):
    with open("error_log.txt", "a") as log_file:
        log_file.write(",".join(row) + "\n")
```

```
# Example usage
validate_and_ingest('data.csv', db_config, 'products')
```

Explanation:

- Checks for missing or invalid data in each row before insertion.
- Logs problematic rows into a file (`error_log.txt`) for further analysis.

Additional Scenarios for Practice

Handle Errors Gracefully During Ingestion

```
def ingest_with_error_handling(csv_file, db_config, table_name):
    try:
        connection = psycopg2.connect(**db_config)
        cursor = connection.cursor()
        with open(csv_file, 'r') as file:
            reader = csv.reader(file)
            headers = next(reader)
            for row in reader:
                try:
                    cursor.execute(
                        f"INSERT INTO {table_name} ({',
'.join(headers)}) VALUES ({', '.join(['%s'] * len(row))})",
                        row
                    )
                except Exception as row_error:
                    log_problematic_row(row)
                    print(f"Row error: {row_error}")
        connection.commit()
    except Exception as e:
        print(f"Error: {e}")
    finally:
        if connection:
            cursor.close()
            connection.close()

# Example usage remains the same.
```

Explanation:

- Catches and logs errors for specific rows without halting the ingestion process.

Practice Scenarios:

1. **Basic Ingestion**: Create a table in PostgreSQL and ingest rows from a simple CSV file.
2. **Validation**: Add data validation checks for missing or invalid values.
3. **Error Logging**: Write invalid rows to a log file for debugging purposes.
4. **Bulk Insert**: Optimize performance by batching multiple rows into a single `INSERT` query.
5. **Extend to NoSQL**: Modify the script to store data in a NoSQL database like MongoDB.

Real-Time Data Streams Tasks

Task 1: Monitor a Directory for New Log Files

You can use the `watchdog` library to monitor a directory for new files and process them as they arrive.

```
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
import time

class NewFileHandler(FileSystemEventHandler):
    def on_created(self, event):
        if event.is_directory:
            return
        print(f"New file detected: {event.src_path}")
        process_file(event.src_path)

def process_file(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            print(f"Processing line: {line.strip()}")

def monitor_directory(path):
    event_handler = NewFileHandler()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=False)
    observer.start()
    print(f"Monitoring directory: {path}")
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
```

```
        observer.stop()
    observer.join()

# Example usage
monitor_directory('logs')
```

Explanation:

- Uses `watchdog` to detect new files in a specified directory.
- Processes each new file by reading its content line-by-line.

Task 2: Simulate Streaming to a Mock Kafka Topic
Simulate streaming data by reading lines from a file and sending them to a Kafka-like queue.

```
import time
from queue import Queue

def simulate_streaming(file_path, queue):
    with open(file_path, 'r') as file:
        for line in file:
            queue.put(line.strip())
            print(f"Streamed: {line.strip()}")
            time.sleep(1)  # Simulate delay for real-time streaming

def consume_stream(queue):
    while True:
        if not queue.empty():
            data = queue.get()
            print(f"Consumed: {data}")
        time.sleep(0.5)

# Example usage
log_queue = Queue()
simulate_streaming('log.txt', log_queue)
consume_stream(log_queue)
```

Explanation:

- Simulates real-time data streaming by reading a file and pushing lines into a queue.
- A consumer continuously fetches data from the queue for processing.

Additional Scenarios for Practice

Task 3: Monitor Directory and Stream to a Mock Kafka
Combine directory monitoring with simulated streaming.

```
def process_file_stream(file_path, queue):
    with open(file_path, 'r') as file:
        for line in file:
            queue.put(line.strip())
            print(f"Streamed: {line.strip()}")
            time.sleep(0.5)

class FileStreamHandler(FileSystemEventHandler):
    def __init__(self, queue):
        self.queue = queue

    def on_created(self, event):
        if not event.is_directory:
            print(f"New file detected: {event.src_path}")
            process_file_stream(event.src_path, self.queue)

# Usage
log_queue = Queue()
event_handler = FileStreamHandler(log_queue)
observer = Observer()
observer.schedule(event_handler, 'logs', recursive=False)
observer.start()

try:
    consume_stream(log_queue)
except KeyboardInterrupt:
    observer.stop()
observer.join()
```

Practice Scenarios:

1. **Basic Monitoring**: Set up a script to detect new files in a directory.
2. **Simulate Streaming**: Read data from a file line-by-line with delays to simulate real-time streaming.
3. **Stream Aggregation**: Continuously aggregate log statistics (e.g., count log levels) from a stream.
4. **Kafka Integration**: Use a real Kafka broker to produce and consume data streams.

5. **Alerting**: Add logic to trigger alerts based on specific patterns in the log stream.

Report Generation Tasks

Task 1: Generate a PDF Report from Sales Data

You can use the `pandas` library for data processing and `fpdf` or `reportlab` for PDF generation.

```
import pandas as pd
from fpdf import FPDF

def generate_pdf_report(csv_file, output_pdf):
    data = pd.read_csv(csv_file)
    summary = data.groupby('Region')['Sales'].sum().reset_index()

    pdf = FPDF()
    pdf.set_font("Arial", size=12)
    pdf.add_page()

    pdf.cell(200, 10, txt="Sales Report by Region", ln=True,
align='C')
    pdf.cell(200, 10, txt="", ln=True)  # Add space

    for index, row in summary.iterrows():
        pdf.cell(200, 10, txt=f"Region: {row['Region']} - Total Sales:
${row['Sales']:.2f}", ln=True)

    pdf.output(output_pdf)
    print(f"PDF report generated: {output_pdf}")

# Example usage
generate_pdf_report('sales_data.csv', 'sales_report.pdf')
```

Explanation:

- Reads sales data from a CSV file and calculates total sales per region.
- Generates a PDF summarizing the sales data with `fpdf`.

Task 2: Create an Excel File with Multiple Sheets

You can use the `pandas` and `openpyxl` libraries to create Excel files.

```
import pandas as pd
```

```
def generate_excel_report(data_dict, output_excel):
    with pd.ExcelWriter(output_excel, engine='openpyxl') as writer:
        for sheet_name, data in data_dict.items():
            df = pd.DataFrame(data)
            df.to_excel(writer, sheet_name=sheet_name, index=False)
    print(f"Excel report generated: {output_excel}")

# Example usage
data = {
    "Sales": {"Product": ["A", "B", "C"], "Sales": [100, 200, 300]},
    "Regions": {"Region": ["North", "South", "East"], "Sales": [500,
600, 700]},
}

generate_excel_report(data, 'report.xlsx')
```

Explanation:

- Uses a dictionary to represent multiple data categories.
- Writes each category to a separate sheet in an Excel file.

Additional Scenarios for Practice

Task 3: Generate a Combined Report with Visualizations

Add visualizations using `matplotlib` and embed them into a PDF report.

```
import matplotlib.pyplot as plt

def generate_sales_chart(data, output_image):
    regions = data['Region']
    sales = data['Sales']

    plt.bar(regions, sales, color='skyblue')
    plt.xlabel('Region')
    plt.ylabel('Total Sales')
    plt.title('Sales by Region')
    plt.savefig(output_image)
    plt.close()

# Integrate chart into PDF
```

```
def generate_visual_pdf_report(csv_file, output_pdf):
    data = pd.read_csv(csv_file)
    summary = data.groupby('Region')['Sales'].sum().reset_index()

    chart_file = 'sales_chart.png'
    generate_sales_chart(summary, chart_file)

    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Arial", size=12)
    pdf.cell(200, 10, txt="Sales Report with Chart", ln=True,
align='C')
    pdf.cell(200, 10, txt="", ln=True)  # Add space

    for index, row in summary.iterrows():
        pdf.cell(200, 10, txt=f"Region: {row['Region']} - Total Sales:
${row['Sales']:.2f}", ln=True)

    pdf.image(chart_file, x=10, y=50, w=150)
    pdf.output(output_pdf)
    print(f"PDF with visualization generated: {output_pdf}")

# Example usage
generate_visual_pdf_report('sales_data.csv',
'visual_sales_report.pdf')
```

Explanation:

- Generates a bar chart of sales by region using `matplotlib`.
- Embeds the chart in a PDF report along with textual summaries.

Practice Scenarios

1. **Basic Report**: Generate a CSV summary report of total and average sales per product.
2. **Detailed PDF Report**: Include text, tables, and charts in a PDF summarizing data trends.
3. **Multi-Sheet Excel**: Create an Excel file with sheets for each category of data (e.g., sales, customers, regions).
4. **Automated Reporting**: Automate the generation of weekly sales reports from raw data files.
5. **Interactive Reports**: Create interactive Excel reports with filters and pivot tables using `xlsxwriter`.

Web Scraping with File Handling Tasks

Task 1: Scrape Data and Save to a CSV File

You can use the `requests` library for fetching data and `BeautifulSoup` from `bs4` for parsing HTML, saving the results in a CSV.

```
import requests
from bs4 import BeautifulSoup
import csv

def scrape_to_csv(url, output_file):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')

    # Example: Scraping a table with class 'data-table'
    table = soup.find('table', class_='data-table')
    rows = table.find_all('tr')

    with open(output_file, mode='w', newline='', encoding='utf-8') as
file:
        writer = csv.writer(file)

        for row in rows:
            columns = row.find_all(['th', 'td'])
            writer.writerow([col.get_text(strip=True) for col in
columns])

    print(f"Data scraped and saved to {output_file}")

# Example usage
scrape_to_csv('https://example.com/data', 'scraped_data.csv')
```

Explanation:

- Fetches a webpage and parses it with `BeautifulSoup`.
- Extracts table rows and writes them to a CSV file.

Task 2: Process Saved Data to Remove Duplicates and Format

Use `pandas` to clean and format the scraped data for analysis.

```
import pandas as pd
```

```
def process_scraped_data(input_file, output_file):
    df = pd.read_csv(input_file)

    # Remove duplicates
    df = df.drop_duplicates()

    # Format columns (e.g., trimming strings or converting types)
    df['Column1'] = df['Column1'].str.strip()
    df['NumericColumn'] = pd.to_numeric(df['NumericColumn'],
errors='coerce')

    # Save cleaned data
    df.to_csv(output_file, index=False)
    print(f"Cleaned data saved to {output_file}")

# Example usage
process_scraped_data('scraped_data.csv', 'cleaned_data.csv')
```

Explanation:

- Removes duplicates and performs column-wise formatting.
- Saves the cleaned data to a new file for further analysis.

Additional Scenarios for Practice

Task 3: Append Scraped Data in Batches

If scraping is performed in chunks, append new data to an existing file.

```
def append_data_to_csv(url, output_file):
    response = requests.get(url)
    soup = BeautifulSoup(response.content, 'html.parser')

    # Extract data as before
    table = soup.find('table', class_='data-table')
    rows = table.find_all('tr')

    with open(output_file, mode='a', newline='', encoding='utf-8') as
file:
        writer = csv.writer(file)
```

```
        for row in rows:
            columns = row.find_all('td')
            writer.writerow([col.get_text(strip=True) for col in
columns])

    print(f"Data appended to {output_file}")

# Example usage
append_data_to_csv('https://example.com/data', 'scraped_data.csv')
```

Explanation:

- Appends data to an existing file without overwriting.
- Useful for scraping sites in chunks.

Task 4: Save Raw HTML for Offline Processing
Save the raw HTML response to a file for later analysis.

```
def save_raw_html(url, output_file):
    response = requests.get(url)
    with open(output_file, 'w', encoding='utf-8') as file:
        file.write(response.text)
    print(f"Raw HTML saved to {output_file}")

# Example usage
save_raw_html('https://example.com', 'raw_data.html')
```

Explanation:

- Saves the raw HTML response from the website to a file for offline inspection or parsing.

Practice Scenarios

1. **Basic Web Scraping**: Scrape a list of items from an e-commerce website and save the details to a CSV file.
2. **Data Cleaning**: Process a CSV file of scraped product data to remove duplicates and format prices.
3. **Batch Processing**: Scrape data in batches from paginated web pages, appending results to a file.
4. **Error Handling**: Implement error handling to log failed requests and retry mechanisms.
5. **Structured Storage**: Store scraped data in a database (e.g., SQLite) for complex queries and analysis.

Data Visualization Tasks

Task 1: Create a Bar Chart from Sales Data and Save as an Image
You can use the `pandas` library for reading the CSV file and `matplotlib` for plotting the chart.

```
import pandas as pd
import matplotlib.pyplot as plt

def create_sales_bar_chart(csv_file, output_image):
    # Read the CSV data
    data = pd.read_csv(csv_file)

    # Assuming the CSV has 'Product' and 'Sales' columns
    plt.bar(data['Product'], data['Sales'], color='skyblue')

    plt.xlabel('Product')
    plt.ylabel('Sales')
    plt.title('Sales per Product')

    # Save the plot as an image
    plt.savefig(output_image)
    plt.close()
    print(f"Bar chart saved as {output_image}")

# Example usage
create_sales_bar_chart('sales_data.csv', 'sales_bar_chart.png')
```

Explanation:

- Reads sales data from a CSV file (with columns like `Product` and `Sales`).
- Creates a bar chart visualizing sales per product.
- Saves the chart as an image (PNG).

Task 2: Plot Temperature Data from a Text File and Save the Plot as a PDF
You can use `matplotlib` to generate the plot and `pandas` to process the data.

```
import pandas as pd
import matplotlib.pyplot as plt

def plot_temperature_data(input_file, output_pdf):
    # Read the data from a text file (assumes tab-separated data)
    data = pd.read_csv(input_file, sep='\t')  # Assuming tab-separated
```

```
    # Assuming the data has 'Date' and 'Temperature' columns
    plt.plot(data['Date'], data['Temperature'], marker='o',
color='orange')

    plt.xlabel('Date')
    plt.ylabel('Temperature (°C)')
    plt.title('Temperature Over Time')

    # Save the plot as a PDF
    plt.savefig(output_pdf)
    plt.close()
    print(f"Temperature plot saved as {output_pdf}")

# Example usage
plot_temperature_data('temperature_data.txt', 'temperature_plot.pdf')
```

Explanation:

- Reads temperature data from a text file (assuming tab-separated values).
- Plots temperature over time using a line chart.
- Saves the chart as a PDF.

Additional Scenarios for Practice

Task 3: Create a Pie Chart of Sales Data and Save as an Image

```
def create_sales_pie_chart(csv_file, output_image):
    data = pd.read_csv(csv_file)

    # Assuming 'Product' and 'Sales' columns
    plt.pie(data['Sales'], labels=data['Product'], autopct='%1.1f%%',
colors=['lightblue', 'lightgreen', 'lightcoral'])

    plt.title('Sales Distribution by Product')

    # Save the pie chart as an image
    plt.savefig(output_image)
    plt.close()
    print(f"Pie chart saved as {output_image}")

# Example usage
```

```
create_sales_pie_chart('sales_data.csv', 'sales_pie_chart.png')
```

Explanation:

- Reads sales data and creates a pie chart to show the percentage distribution of sales across products.
- Saves the chart as a PNG image.

Task 4: Generate a Scatter Plot from JSON Data and Save as an Image

```
import json
import matplotlib.pyplot as plt

def create_scatter_plot(json_file, output_image):
    with open(json_file, 'r') as file:
        data = json.load(file)

    # Assuming the JSON has 'x' and 'y' values for the scatter plot
    plt.scatter(data['x'], data['y'], color='blue')

    plt.xlabel('X-Axis')
    plt.ylabel('Y-Axis')
    plt.title('Scatter Plot of Data')

    # Save the scatter plot as an image
    plt.savefig(output_image)
    plt.close()
    print(f"Scatter plot saved as {output_image}")

# Example usage
create_scatter_plot('data.json', 'scatter_plot.png')
```

Explanation:

- Reads JSON data and creates a scatter plot using the 'x' and 'y' values.
- Saves the plot as a PNG image.

Practice Scenarios

1. **Sales Data Visualization**: Generate a bar chart for sales data by product, save it as a PNG file.
2. **Temperature Plot**: Read temperature data from a CSV and create a line plot, saving it as a PDF.
3. **Pie Chart**: Create a pie chart showing the distribution of sales per product.

4. **Scatter Plot**: Plot data points from a JSON file and save the plot as an image.
5. **Multiple Plots**: Create a multi-page PDF report with different types of visualizations (e.g., bar chart, line plot, pie chart).

These tasks can help you practice visualizing data in various formats and saving those visualizations in appropriate file formats.

Error Handling and Logging Tasks

Task 1: Process Files and Log Results (Success or Error) to a Log File

You can use the `logging` module to log errors and process results while handling errors during file processing.

```
import os
import logging

# Set up the logging configuration
logging.basicConfig(filename='file_processing.log',
level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

def process_file(file_path):
    try:
        if not os.path.exists(file_path):
            raise FileNotFoundError(f"File '{file_path}' does not
exist.")

        with open(file_path, 'r') as file:
            data = file.read()
            # Simulate processing the file (e.g., parsing or
transforming data)
            logging.info(f"Successfully processed file: {file_path}")

    except FileNotFoundError as e:
        logging.error(f"Error processing file {file_path}: {e}")

    except Exception as e:
        logging.error(f"Unexpected error processing file {file_path}:
{e}")

# Example usage: Process files in a list and log results
```

```
files = ['file1.txt', 'file2.txt', 'file3.txt']

for file in files:
    process_file(file)
```

Explanation:

- Configures logging to write to `file_processing.log` with timestamp, log level, and message format.
- Tries to read and process files and logs success or failure.
- If the file does not exist, it logs a `FileNotFoundError`. Other unexpected errors are also logged.

Task 2: Retry Processing Failed Files Using the Log File
This scenario adds retry logic by reading the log file and retrying failed files.

```
import time

def retry_failed_files(log_file='file_processing.log', retry_limit=3):
    with open(log_file, 'r') as file:
        logs = file.readlines()

    failed_files = []
    for log in logs:
        if 'Error' in log:  # Check for lines indicating errors
            failed_files.append(log.split(" ")[-1].strip())  # Extract
file name from the log

    # Retry processing failed files
    retries = 0
    for file in failed_files:
        while retries < retry_limit:
            try:
                process_file(file)  # Retry processing
                logging.info(f"Successfully retried processing file:
{file}")
                break
            except Exception as e:
                retries += 1
                logging.error(f"Retry {retries} failed for file
{file}: {e}")
```

```
                time.sleep(2)  # Wait before retrying (simulating
backoff)

# Example usage: Retry failed files after initial processing attempt
retry_failed_files()
```

Explanation:

- Reads the log file and identifies failed file entries (lines containing "Error").
- Retries processing each failed file up to a specified limit (`retry_limit`).
- Logs each retry attempt and the result.

Key Concepts and Practices

1. **Logging**:
   - Use `logging` to capture success and error messages during file processing.
   - Log messages should include the file name and error details for easy debugging.
2. **Error Handling**:
   - Handle common errors (e.g., `FileNotFoundError`, permission issues) and unexpected ones (e.g., `IOError`) in the `try-except` blocks.
   - Log detailed error messages for troubleshooting.
3. **File Retry Mechanism**:
   - Read from the log file to identify failed files and retry processing them.
   - Use a retry limit to prevent infinite loops and log each retry attempt.

Additional Practice

1. **Task 3: Implement a Retry with Backoff Strategy**: Retry file processing with increasing time delays after each failed attempt.
2. **Task 4: Log Detailed Execution Status**: Extend logging to include the time taken for each file's processing, whether it was successful or failed, and other relevant metrics.
3. **Task 5: Process Multiple Directories**: Extend the logic to process files across multiple directories and log the result for each.

These tasks help you practice handling errors during file processing and create a robust system for logging and retrying operations when needed.

Data Comparison Tasks

Task 1: Compare Two CSV Files and Output the Differences to a New File

This task involves reading two CSV files, comparing their contents, and writing the differences (mismatched records) to a new file.

```
import csv

def compare_csv(file1, file2, output_file):
    # Read both CSV files into sets for comparison
```

```
    with open(file1, 'r') as f1, open(file2, 'r') as f2:
        reader1 = csv.reader(f1)
        reader2 = csv.reader(f2)

        data1 = set(tuple(row) for row in reader1)
        data2 = set(tuple(row) for row in reader2)

    # Find differences between the two files
    diff1 = data1 - data2  # Records in file1 but not in file2
    diff2 = data2 - data1  # Records in file2 but not in file1

    # Write differences to an output file
    with open(output_file, 'w', newline='') as outfile:
        writer = csv.writer(outfile)
        writer.writerow(["Differences in File 1"])  # Header for
file1-only differences
        writer.writerows(diff1)

        writer.writerow(["Differences in File 2"])  # Header for
file2-only differences
        writer.writerows(diff2)

    print(f"Differences have been written to {output_file}")

# Example usage
compare_csv('file1.csv', 'file2.csv', 'differences.csv')
```

Explanation:

- This code reads two CSV files and compares their content row by row.
- It uses `set` to store the rows as tuples for easy comparison and identifies records that are unique to each file.
- The differences are written to a new CSV file, separating differences in each file.

Task 2: Identify and Remove Duplicate Rows in a File

This task involves reading a CSV file, identifying duplicate rows, and creating a new file without duplicates.

```
import csv

def remove_duplicates(input_file, output_file):
```

```
    # Read input file and store unique rows
    with open(input_file, 'r') as infile:
        reader = csv.reader(infile)
        unique_rows = set(tuple(row) for row in reader)

    # Write unique rows to output file
    with open(output_file, 'w', newline='') as outfile:
        writer = csv.writer(outfile)
        writer.writerows(unique_rows)

    print(f"Duplicate rows removed. Cleaned file saved to
{output_file}")

# Example usage
remove_duplicates('data.csv', 'cleaned_data.csv')
```

Explanation:

- This code reads a CSV file, converts each row to a tuple (so it can be added to a set, which removes duplicates), and writes only the unique rows to a new file.
- The `set` ensures that only one occurrence of each row remains, effectively removing duplicates.

Key Concepts and Practices

1. **Set Operations**:
    - Use sets to find differences and duplicates easily, as sets inherently eliminate duplicate values.
    - The `set - set` operation allows us to find records that are only in one file or the other.
2. **CSV Handling**:
    - Use the `csv` module to read and write CSV files in Python.
    - Handle row data as tuples when comparing files or eliminating duplicates, as they are hashable and can be added to sets.
3. **File Writing**:
    - After processing and comparison, write the results to new CSV files.
    - Be mindful of file handling (e.g., using `newline=''` when writing CSVs to prevent extra blank lines).

Additional Practice

1. **Task 3: Compare JSON Files**: Write a program to compare two JSON files and output the differences.

2. **Task 4: Handle Large Files Efficiently**: Implement a solution that processes large files in chunks to handle memory efficiently.
3. **Task 5: Advanced Data Comparison**: Implement fuzzy matching to compare records that may not be exact but are similar (e.g., using the `fuzzywuzzy` package).

These tasks will help you practice comparing data across different files, handling duplicates, and saving the results for further analysis.

Encoding and Decoding Files

Task 1: Convert File Encoding (e.g., from ISO-8859-1 to UTF-8)

This task involves reading a text file in one encoding and saving it in another encoding. For example, you may need to convert a file from ISO-8859-1 encoding to UTF-8.

```
def convert_encoding(input_file, output_file,
input_encoding='ISO-8859-1', output_encoding='UTF-8'):
    # Read the file with the input encoding
    with open(input_file, 'r', encoding=input_encoding) as infile:
        content = infile.read()

    # Write the content to the output file with the new encoding
    with open(output_file, 'w', encoding=output_encoding) as outfile:
        outfile.write(content)

    print(f"File has been converted from {input_encoding} to
{output_encoding} and saved as {output_file}")

# Example usage
convert_encoding('input_file.txt', 'output_file.txt', 'ISO-8859-1',
'UTF-8')
```

Explanation:

- The script opens the `input_file.txt` in ISO-8859-1 encoding, reads its content, and then writes it into a new file (`output_file.txt`) with UTF-8 encoding.
- This allows for seamless conversion between different character encodings.

Task 2: Process a Binary File (e.g., Extract Metadata from an Image or PDF)

This task involves reading a binary file (e.g., an image or PDF) and extracting specific metadata (e.g., image dimensions or PDF properties). Below is an example using the `Pillow` library for image files to extract metadata:

```
from PIL import Image
```

```
def extract_image_metadata(input_file):
    # Open the image file
    with Image.open(input_file) as img:
        # Extract image dimensions
        width, height = img.size
        format = img.format
        mode = img.mode

    # Print extracted metadata
    print(f"Image format: {format}")
    print(f"Image dimensions: {width}x{height}")
    print(f"Image mode: {mode}")

# Example usage
extract_image_metadata('image.jpg')
```

Explanation:

- This script uses the `PIL` (Python Imaging Library, now part of `Pillow`) to open an image file and extract basic metadata like its format (e.g., JPEG, PNG), dimensions (width and height), and color mode.
- This method works for various image formats and helps in extracting useful information without altering the file itself.

For PDF files, you can use `PyPDF2` or `pdfminer.six` to extract metadata:

```
import PyPDF2

def extract_pdf_metadata(input_file):
    with open(input_file, 'rb') as f:
        reader = PyPDF2.PdfFileReader(f)
        metadata = reader.getDocumentInfo()
        num_pages = reader.getNumPages()

    print(f"PDF Title: {metadata.title}")
    print(f"Author: {metadata.author}")
    print(f"Number of Pages: {num_pages}")

# Example usage
extract_pdf_metadata('document.pdf')
```

Explanation:

- The `PyPDF2` library is used to open and read the PDF file in binary mode (`'rb'`), extract metadata (such as the title and author), and also get the number of pages in the document.

Key Concepts and Practices

1. **Encoding Handling**:
    - When reading and writing files, ensure that the correct encoding is used to avoid issues with special characters or non-ASCII content.
    - Python's `open()` function allows specifying the encoding, which is essential when working with international characters or legacy files.
2. **Binary File Processing**:
    - When dealing with binary files (e.g., images, PDFs), ensure that the file is opened in binary mode (`'rb'` for reading and `'wb'` for writing).
    - Libraries like `Pillow` (for images) or `PyPDF2` (for PDFs) provide specialized methods for extracting metadata without needing to load the entire file into memory.
3. **Metadata Extraction**:
    - Extracting metadata can be useful for understanding file properties (e.g., image dimensions or document details) without having to open or manually inspect the content.
    - This is particularly useful for batch processing of files, such as organizing images or indexing PDF documents based on their metadata.

Additional Practice

1. **Task 3: Convert Image Formats**: Write a script to read an image file in one format (e.g., PNG) and save it as another format (e.g., JPEG).
2. **Task 4: Metadata Extraction from Audio/Video Files**: Use libraries like `mutagen` or `pydub` to extract metadata from audio or video files (e.g., bit rate, codec, duration).
3. **Task 5: Binary Data Manipulation**: Write a program that processes binary data (e.g., modifying a file's header or extracting sections of the file).