Assignment 11 Solution

Submitted by: Yameen Ali

Go to Github repository

Question1:

- a. Create a new directory named "FileOps" on the desktop if it doesn't already exist.
- b. Navigate into the "FileOps" directory and create three new text files: "file1.txt", "file2.txt", and "file3.txt".
- c. Write some sample text data into each of the created text files.
- d. Display the list of files in the "FileOps" directory along with their respective sizes and creation timestamps.
- e. Rename "file1.txt" to "renamed_file.txt".
- f.Move "renamed_file.txt" to a new directory named "Archive" inside the "FileOps" directory.
- g. Display the list of files in the "FileOps" directory to verify the changes.
- h. Use the time module to get the current UTC and convert it to the local time in the "Asia/Kolkata" timezone using pytz.
- i. Print the current local time in a human-readable format.

Import Dependencies

In [32]:
```
import os
```

a. Create a new directory named "FileOps" on the desktop if it doesn't already exist.

In [30]:
```
# Get the path to the desktop
desktop_path = os.path.join(os.path.expanduser('~'), 'Desktop')
fileops_dir = os.path.join(desktop_path, 'FileOps')

# Create "FileOps" directory if it doesn't exist
if not os.path.exists(fileops_dir):
    os.makedirs(fileops_dir)
    print(f"Created directory: {fileops_path}")
```

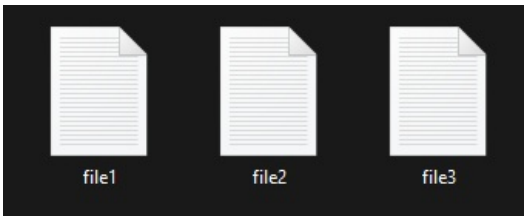
```
Created directory: C:\Users\PMYLS\Desktop\FileOps
```

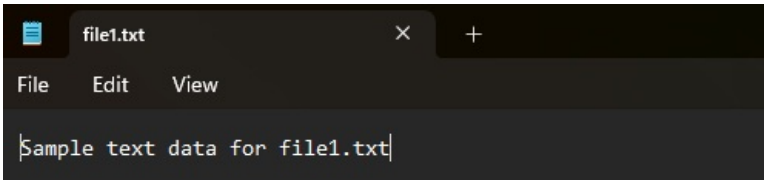
b. Navigate into the "FileOps" directory and create three new text files: "file1.txt", "file2.txt", and "file3.txt".

c. Write some sample text data into each of the created text files.

In [33]:
```
# Change the current working directory to "FileOps"
os.chdir(fileops_dir)

# Create three new text files
file1_path = os.path.join(fileops_dir, 'file1.txt')
file2_path = os.path.join(fileops_dir, 'file2.txt')
file3_path = os.path.join(fileops_dir, 'file3.txt')

with open(file1_path, 'w') as file1:
    file1.write("Sample text data for file1.txt")
```

```
with open(file2_path, 'w') as file2:
    file2.write("Sample text data for file2.txt")

with open(file3_path, 'w') as file3:
    file3.write("Sample text data for file3.txt")
```





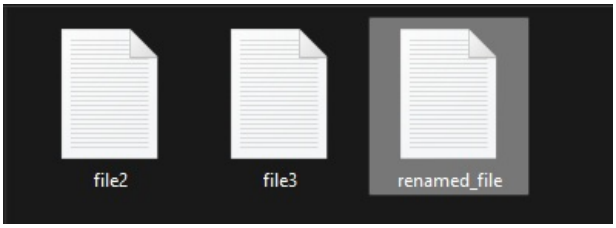
d. Display the list of files in the "FileOps" directory along with their respective sizes and creation timestamps.

In [34]:
```
from datetime import datetime

print("Files in FileOps directory:")
files_in_fileops = os.listdir(fileops_dir)
for file_name in files_in_fileops:
    file_path = os.path.join(fileops_dir, file_name)
    file_size = os.path.getsize(file_path)
    creation_time = datetime.fromtimestamp(os.path.getctime(file_path))
    print(f"{file_name}: Size - {file_size} bytes, Created - {creation_time}")
```

```
Files in FileOps directory:
file1.txt: Size - 30 bytes, Created - 2024-07-01 12:58:01.448514
file2.txt: Size - 30 bytes, Created - 2024-07-01 12:58:01.453512
file3.txt: Size - 30 bytes, Created - 2024-07-01 12:58:01.454512
```

e. Rename "file1.txt" to "renamed_file.txt".

In [35]:
```
renamed_file_path = os.path.join(fileops_dir, 'renamed_file.txt')
os.rename(file1_path, renamed_file_path)
```

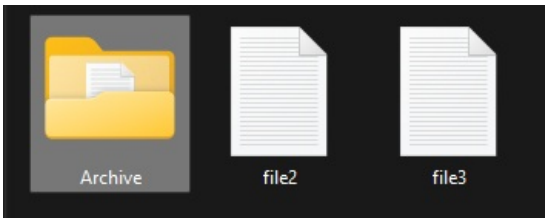


f. Move "renamed_file.txt" to a new directory named "Archive" inside the "FileOps" directory.

- Create the "Archive" directory if it doesn't exist.
- Use shutil.move() to move the file

In [36]:
```
import shutil

archive_dir = os.path.join(fileops_dir, 'Archive')
os.makedirs(archive_dir, exist_ok=True)
shutil.move(renamed_file_path, archive_dir)
```

Out[36]:
```
'C:\\Users\\PMYLS\\Desktop\\FileOps\\Archive\\renamed_file.txt'
```



g. Display the list of files in the "FileOps" directory to verify the changes.

- Similar to step 4, but after the file operations.

In [37]:
```
print("\nFiles in FileOps directory after changes:")
files_in_fileops_updated = os.listdir(fileops_dir)
for file_name in files_in_fileops_updated:
    file_path = os.path.join(fileops_dir, file_name)
    file_size = os.path.getsize(file_path)
    creation_time = datetime.fromtimestamp(os.path.getctime(file_path))
    print(f"{file_name}: Size - {file_size} bytes, Created - {creation_time}")
```

```
Files in FileOps directory after changes:
Archive: Size - 0 bytes, Created - 2024-07-01 13:08:56.684246
file2.txt: Size - 30 bytes, Created - 2024-07-01 12:58:01.453512
file3.txt: Size - 30 bytes, Created - 2024-07-01 12:58:01.454512
```

h. Use the time module to get the current UTC and convert it to the local time in the "Asia/Kolkata" timezone using pytz.

- Get the current UTC time.
- Convert the UTC time to the "Asia/Kolkata" timezone using pytz.

In [38]:
```
from datetime import datetime
import pytz

# Get the current UTC time
utc_now = datetime.utcnow().replace(tzinfo=pytz.utc)
kolkata_tz = pytz.timezone('Asia/Kolkata')
local_now = utc_now.astimezone(kolkata_tz)
```

i. Print the current local time in a human-readable format.

Format the local time in a human-readable string using strftime().

In [39]:
```
print(f"\nCurrent local time in Asia/Kolkata: {local_now.strftime('%Y-%m-%d %H:%M:%S %Z')}")
```

```
Current local time in Asia/Kolkata: 2024-07-01 13:42:00 IST
```

Question2: (Maintain log file)

Ensure that the log entries in the "error_log.txt" file are formatted with specific columns:

Import Required Modules

In [47]:
```
import os
from datetime import datetime
```

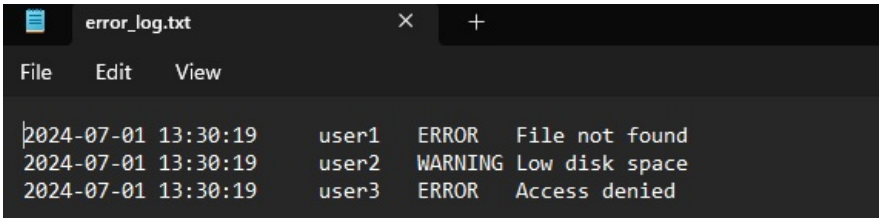
a) The first column should display the timestamp of each log entry in the format "YYYY-MM-DD HH:MM:SS".

b) The second column should display the username or user identifier associated with the logged event.

c) The third column should indicate the type of error or warning (e.g., "ERROR" or "WARNING").

d) The fourth column should contain the detailed error or warning message.

In [54]:
```
def write_log_entry(username, error_type, message):
    # Get the current timestamp in the format "YYYY-MM-DD HH:MM:SS"
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    # Define the log entry format
    log_entry = f"{timestamp}\t{username}\t{error_type}\t{message}\n"

    # Define the log file path
    log_file_path = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'error_log.txt')

    # Ensure the FileOps directory exists
    os.makedirs(os.path.dirname(log_file_path), exist_ok=True)

    # Append the log entry to "error_log.txt"
    with open(log_file_path, 'a') as log_file:
        log_file.write(log_entry)

# Example usage of the function
write_log_entry("user1", "ERROR", "File not found")
write_log_entry("user2", "WARNING", "Low disk space")
write_log_entry("user3", "ERROR", "Access denied")
```



Question 3:

You are working on a web application that generates log files containing timestamps, user IDs, and request details. Write a Python script that analyzes a specific log file and provides the following insights:

- a) Total number of requests processed on a particular date (provided as user input).
- b) Top 10 most frequent user IDs making requests.
- c) Identify any requests taking longer than a certain threshold (e.g., 1 second) and log them to a separate error file

Generate a Sample Log File

In [62]:
```
def generate_sample_log_file():
    log_file_path = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'app_log.txt')
    os.makedirs(os.path.dirname(log_file_path), exist_ok=True)

    sample_logs = [
        "2024-06-30 12:00:00 user1 0.5",
        "2024-06-30 12:05:00 user2 1.5",
        "2024-06-30 12:10:00 user1 0.3",
        "2024-06-30 12:15:00 user3 2.0",
        "2024-07-01 13:00:00 user2 0.6",
        "2024-07-01 13:05:00 user1 1.2",
        "2024-07-01 13:10:00 user3 0.4",
        "2024-07-01 13:15:00 user2 3.0",
        "2024-07-02 14:00:00 user3 0.8",
        "2024-07-02 14:05:00 user1 1.1"
    ]

    with open(log_file_path, 'w') as file:
        for log in sample_logs:
            file.write(log + "\n")

    print(f"Sample log file created at: {log_file_path}")

generate_sample_log_file()
```

```
Sample log file created at: C:\Users\PMYLS\Desktop\FileOps\app_log.txt
```

Step 1: Import Required Modules

- We need the collections module to count user IDs and the datetime module to handle dates and times.

In [55]:
```
import os
from collections import Counter
from datetime import datetime
```

Step 2: Define Function to Read Log File

- Read the log file and return its contents.

In [56]:
```
def read_log_file(log_file_path):
    with open(log_file_path, 'r') as file:
        return file.readlines()
```

Step 3: Define Function to Parse Log Entries

- Extract timestamp, user ID, and request details from each log entry.

In [57]:
```
def parse_log_entry(log_entry):
    parts = log_entry.strip().split()
    timestamp_str = parts[0] + ' ' + parts[1]
    user_id = parts[2]
    request_time = float(parts[3])
    return timestamp_str, user_id, request_time
```

Step 4: Count Requests on a Specific Date

- Prompt the user for a date and count the total number of requests on that date.

In [58]:
```
def count_requests_on_date(log_entries, date_str):
    date_obj = datetime.strptime(date_str, "%Y-%m-%d")
    count = 0
    for entry in log_entries:
        timestamp_str, _, _ = parse_log_entry(entry)
        entry_date = datetime.strptime(timestamp_str, "%Y-%m-%d %H:%M:%S").date()
        if entry_date == date_obj.date():
            count += 1
    return count
```

Step 5: Find Top 10 Most Frequent User IDs

- Use a Counter to count the number of requests made by each user ID.

In [59]:
```
def top_10_user_ids(log_entries):
    user_counter = Counter()
    for entry in log_entries:
        _, user_id, _ = parse_log_entry(entry)
        user_counter[user_id] += 1
    return user_counter.most_common(10)
```

Step 6: Identify Requests Taking Longer Than a Threshold

- Check if the request time exceeds a specified threshold and log these requests to a separate error file.

In [60]:
```
def log_long_requests(log_entries, threshold, error_log_file_path):
    with open(error_log_file_path, 'a') as error_log_file:
        for entry in log_entries:
            timestamp_str, user_id, request_time = parse_log_entry(entry)
            if request_time > threshold:
                error_log_file.write(entry)
```

Step 7: Usage of the Functions

- Combine all the steps to analyze the log file and provide the required insights.

In [63]:
```
def main():
    log_file_path = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'app_log.txt')
    error_log_file_path = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'error_log.txt')

    # Read log file
    log_entries = read_log_file(log_file_path)

    # Count requests on a specific date
    date_str = input("Enter the date (YYYY-MM-DD): ")
    total_requests = count_requests_on_date(log_entries, date_str)
    print(f"Total number of requests on {date_str}: {total_requests}")

    # Find top 10 most frequent user IDs
    top_users = top_10_user_ids(log_entries)
    print("Top 10 most frequent user IDs:")
    for user_id, count in top_users:
        print(f"{user_id}: {count} requests")

    # Log requests taking longer than the threshold
    threshold = float(input("Enter the request time threshold in seconds: "))
    log_long_requests(log_entries, threshold, error_log_file_path)
    print(f"Requests taking longer than {threshold} seconds have been logged to {error_log_file_path}")

if __name__ == "__main__":
    main()
```

```
Enter the date (YYYY-MM-DD): 2024-01-01
Total number of requests on 2024-01-01: 0
Top 10 most frequent user IDs:
user1: 4 requests
user2: 3 requests
user3: 3 requests
Enter the request time threshold in seconds: 1
Requests taking longer than 1.0 seconds have been logged to C:\Users\PMYLS\Desktop\FileOps\error_log.txt
```

Question4:

You are managing a system that generates large data files daily. Design a Python script to

automate the following tasks:

- a) Compress old data files (e.g., older than 30 days) into a ZIP archive using a naming convention based on the date.
- b) Move the compressed archives to a separate directory for backup.
- c) Delete any original data files older than a specific threshold (e.g., 60 days) to free up disk space.

Step 1: Import Required Modules

- We'll import the necessary modules for file operations, date calculations, and zip operations.

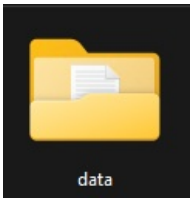
In [89]:
```
import os
import shutil
from datetime import datetime, timedelta
import zipfile
```

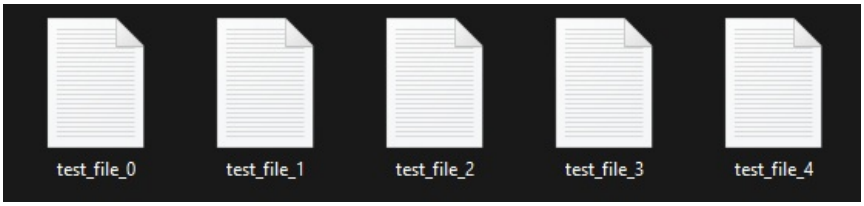
Create Test Data and Adjust File Times

In [93]:
```
def create_sample_files(directory, num_files=5):
    ensure_directory_exists(directory)
    current_time = time.time()

    for i in range(num_files):
        file_path = os.path.join(directory, f'test_file_{i}.txt')
        with open(file_path, 'w') as f:
            f.write(f"This is test file {i}")

        # Modify the file's modification time to simulate an old file
        old_time = current_time - (i + 1) * 86400 * 31  # 31 days old
        os.utime(file_path, (old_time, old_time))

def ensure_directory_exists(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)
```

In [94]:
```
def main():
    data_directory = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'data')

    # Create sample files and modify their modification times
    create_sample_files(data_directory)

    print("Sample files created and modification times set.")

if __name__ == "__main__":
    main()
```

```
Sample files created and modification times set.
```





Step 2: Define a Function to Compress Old Data Files

- Compress data files older than a specified number of days into a ZIP archive.

In [95]:
```
def compress_old_files(directory, days_old, archive_dir):
    cutoff_date = datetime.now() - timedelta(days=days_old)

    for root, dirs, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            file_mod_time = datetime.fromtimestamp(os.path.getmtime(file_path))

            if file_mod_time < cutoff_date:
                archive_name = os.path.join(archive_dir, file_mod_time.strftime('%Y-%m-%d') + '.zip')

                with zipfile.ZipFile(archive_name, 'a') as zipf:
```

```
                    zipf.write(file_path, arcname=file)

                os.remove(file_path)
```

Step 3: Define a Function to Move Compressed Archives

- Move the compressed ZIP archives to a separate backup directory.

In [96]:
```
def move_archives(archive_dir, backup_dir):
    if not os.path.exists(backup_dir):
        os.makedirs(backup_dir)

    for file in os.listdir(archive_dir):
        if file.endswith('.zip'):
            shutil.move(os.path.join(archive_dir, file), os.path.join(backup_dir, file))
```

Step 4: Define a Function to Delete Old Data Files

- Delete any data files older than a specified number of days to free up disk space.

In [97]:
```
def delete_old_files(directory, days_old):
    cutoff_date = datetime.now() - timedelta(days=days_old)

    for root, dirs, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            file_mod_time = datetime.fromtimestamp(os.path.getmtime(file_path))

            if file_mod_time < cutoff_date:
                os.remove(file_path)
```

Step 5: Main Function to Combine All Steps

- Combine all the steps into a main function to automate the entire process.

In [98]:
```
def main():
    data_directory = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'data')
    archive_directory = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'archives')
    backup_directory = os.path.join(os.path.expanduser('~'), 'Desktop', 'FileOps', 'backup')

    # Ensure directories exist
    ensure_directory_exists(data_directory)
    ensure_directory_exists(archive_directory)
    ensure_directory_exists(backup_directory)

    # Compress files older than 30 days
    compress_old_files(data_directory, 30, archive_directory)

    # Move compressed archives to backup directory
    move_archives(archive_directory, backup_directory)

    # Delete original data files older than 60 days
    delete_old_files(data_directory, 60)

    print("Data file management tasks completed successfully.")

if __name__ == "__main__":
    main()
```

```
Data file management tasks completed successfully.
```