Practical Python – Error Handling, Logging, and Data Manipulation Assignment -

Question 1 : What is the difference between multithreading and multiprocessing?

Answer :

Multithreading involves running multiple threads within a single process. Each thread shares the same memory space and resources.
Multiprocessing involves running multiple processes, each with its own memory space and resources.
Key Difference
- Concurrency: Both multithreading and multiprocessing enable concurrent execution, but multiprocessing achieves true parallelism for CPU-bound tasks.
- Resource Usage: Multithreading is more memory-efficient, while multiprocessing requires more resources due to separate

memory spaces.

Question 2: What are the challenges associated with memory management in Python?

Answer:

Python uses automatic memory management through a garbage collector, which frees developers from worrying about memory allocation and deallocation. However, there are still challenges associated with memory management in Python.

Challenges
1. Memory Leaks: Circular references can cause memory leaks if not properly handled.
2. Global Interpreter Lock (GIL): GIL can limit the effectiveness of multithreading and lead to memory-related issues.
3. Memory Fragmentation: Frequent allocation and deallocation of memory can lead to fragmentation, reducing performance.
4. Reference Cycles: Objects referencing each other can prevent garbage collection.
5. Large Objects: Handling large objects, such as data structures or files, can consume

significant memory.
6. Memory Profiling: Identifying memory bottlenecks and leaks can be challenging.

Common Issues
1. MemoryError: Insufficient memory for large data structures or objects.
2. Performance Degradation: Excessive memory usage can slow down the system.

Question 3: Write a Python program that logs an error message to a log file when a division by zero exception occurs.

Answer:

```
import logging

# Configure logging
logging.basicConfig(filename='division_error.log', level=logging.ERROR)

def divide(a, b):
    try:
        result = a / b
        return result
```

```
    except ZeroDivisionError:
        logging.error("Division by zero occurred")

# Test the function
print(divide(10, 2))  # Should return 5.0
print(divide(10, 0))  # Should log an error
```

Question 4: Write a Python program that reads from one file and writes its content to another file.

Answer:

```
def copy_file(source_file, destination_file):
    try:
        with open(source_file, 'r') as source:
            content = source.read()
        with open(destination_file, 'w') as
destination:
            destination.write(content)
        print(f"File copied successfully from
{source_file} to {destination_file}")
    except FileNotFoundError:
        print(f"File {source_file} not found")
    except Exception as e:
        print(f"An error occurred: {e}")
```

```
# Test the function
copy_file('source.txt', 'destination.txt')
```

Question 5: Write a program that handles both IndexError and KeyError using a try-except block.

Answer:

```
def handle_errors():
    try:
        # Test IndexError
        numbers = [1, 2, 3]
        print(numbers[5])  # This will raise
IndexError

    except IndexError:
        print("IndexError: Index out of range")

    try:
        # Test KeyError
        person = {"name": "John", "age": 30}
        print(person["city"])  # This will raise
KeyError

    except KeyError:
        print("KeyError: Key not found")
```

```
# Alternatively, you can use a single try-except block
def handle_errors_alternative():
    try:
        numbers = [1, 2, 3]
        print(numbers[5])  # This will raise IndexError

        person = {"name": "John", "age": 30}
        print(person["city"])  # This will not be executed if IndexError occurs

    except IndexError:
        print("IndexError: Index out of range")
    except KeyError:
        print("KeyError: Key not found")

# Test the functions
handle_errors()
handle_errors_alternative()
```

Question 6: What are the differences between NumPy arrays and Python lists?

Answer:

1. Multi-dimensional data structure: Supports vectors, matrices, and higher-dimensional arrays.
2. Homogeneous data type: All elements must have the same data type.
3. Memory-efficient: Stores data in a contiguous block of memory, reducing memory usage.
4. Faster operations: Optimized for numerical computations, providing faster performance.
5. Vectorized operations: Supports element-wise operations, reducing the need for loops.

Python Lists
1. Dynamic data structure: Can grow or shrink dynamically.
2. Heterogeneous data type: Can store elements of different data types.
3. More memory usage: Stores pointers to each element, resulting in higher memory usage.
4. Slower operations: Not optimized for numerical computations, leading to slower performance.
5. More flexible: Supports various operations, such as insertion, deletion, and append.

Question 7:Explain the difference between apply() and map() in Pandas.

Answer:

apply()
1. Apply a function: Applies a function to each row or column of a DataFrame.
2. Flexible: Can be used with any function, including lambda functions.
3. Row/Column-wise operation: Can operate on rows or columns.
4. Returns: Returns a Series or DataFrame.

map()
1. Map values: Maps values of a Series to a dictionary or a function.
2. Element-wise operation: Operates on individual elements.
3. Returns: Returns a Series.

Key Differences:
Operation : Row/Column-wise , Element-wise.
Functionality: More flexible, can use any function, Limited to mapping values.
Usage : Data Frames and Series .

Answer:

Question 8: Create a histogram using Seaborn to visualize a distribution.

Answer:

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Generate sample data
np.random.seed(0)
data = np.random.randn(1000)

# Create a histogram
plt.figure(figsize=(8, 6))
sns.histplot(data, kde=True, bins=30)

# Set title and labels
plt.title('Histogram of Normal Distribution')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Show the plot
plt.show()
```

Question 9: Use Pandas to load a CSV file and display its first 5 rows.

Answer:

```
import pandas as pd

# Load the CSV file
def load_csv(file_path):
    try:
        df = pd.read_csv(file_path)
        return df
    except FileNotFoundError:
        print("File not found. Please check the file path.")
        return None
    except pd.errors.EmptyDataError:
        print("File is empty. Please check the file contents.")
        return None

# Display the first 5 rows
def display_rows(df):
    if df is not None:
        print(df.head(5))
```

```
# Usage
file_path = 'data.csv'  # Replace with your CSV
file path
df = load_csv(file_path)
display_rows(df)
```

Question 10: Calculate the correlation matrix using Seaborn and visualize it with a heatmap.

Answer:

```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Generate sample data
np.random.seed(0)
data = np.random.randn(100, 5)
df = pd.DataFrame(data, columns=['A', 'B', 'C',
'D', 'E'])

# Calculate the correlation matrix
corr_matrix = df.corr()

# Visualize the correlation matrix with a
heatmap
plt.figure(figsize=(8, 6))
```

```
sns.heatmap(corr_matrix, annot=True,
cmap='coolwarm', square=True, fmt='.2f')
plt.title('Correlation Matrix')
plt.show()
```