KAIROS PROJECT PRESENTATION

Data Analysis and Visualization

TEAM

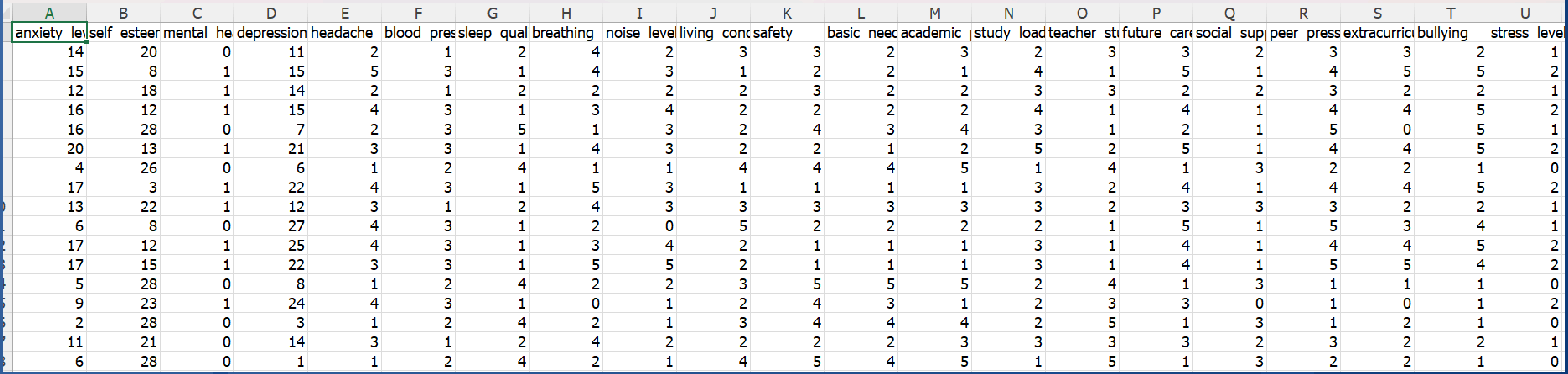
6688030	Sarach Islam
6688055	Takka Leeheng
6688059	Athip Madnoth
6688124	Nithit Teeraworawit
6688173	Passakorn Piboonmahachotikul

OUR OBJECTIVE

To optimize the process of analysis and visualization of data and create a better system for such task.

RECORD EXAMPLE OF OUR DATASET

This is raw data we will process and there are 20 column records to manage



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
anxiety_lev	self_esteer	mental_hea	depression	headache	blood_pres	sleep_qual	breathing_	noise_level	living_conc	safety	basic_neec	academic_	study_load	teacher_st	future_care	social_sup	peer_press	extracurric	bullying	stress_level
14	20	0	11	2	1	2	4	2	3	3	2	3	2	3	3	2	3	3	2	1
15	8	1	15	5	3	1	4	3	1	2	2	1	4	1	5	1	4	5	5	2
12	18	1	14	2	1	2	2	2	2	3	2	2	3	3	2	2	3	2	2	1
16	12	1	15	4	3	1	3	4	2	2	2	2	4	1	4	1	4	4	5	2
16	28	0	7	2	3	5	1	3	2	4	3	4	3	1	2	1	5	0	5	1
20	13	1	21	3	3	1	4	3	2	2	1	2	5	2	5	1	4	4	5	2
4	26	0	6	1	2	4	1	1	4	4	4	5	1	4	1	3	2	2	1	0
17	3	1	22	4	3	1	5	3	1	1	1	1	3	2	4	1	4	4	5	2
13	22	1	12	3	1	2	4	3	3	3	3	3	3	2	3	3	3	2	2	1
6	8	0	27	4	3	1	2	0	5	2	2	2	2	1	5	1	5	3	4	1
17	12	1	25	4	3	1	3	4	2	1	1	1	3	1	4	1	4	4	5	2
17	15	1	22	3	3	1	5	5	2	1	1	1	3	1	4	1	5	5	4	2
5	28	0	8	1	2	4	2	2	3	5	5	5	2	4	1	3	1	1	1	0
9	23	1	24	4	3	1	0	1	2	4	3	1	2	3	3	0	1	0	1	2
2	28	0	3	1	2	4	2	1	3	4	4	4	2	5	1	3	1	2	1	0
11	21	0	14	3	1	2	4	2	2	2	2	3	3	3	3	2	3	2	2	1
6	28	0	1	1	2	4	2	1	4	5	4	5	1	5	1	3	2	2	1	0

DATA READING

OS CONCEPT WE APPLIED

1. Simulate Process Scheduling

   Break tasks into separate multiprocessing jobs

2.Optimize File I/O

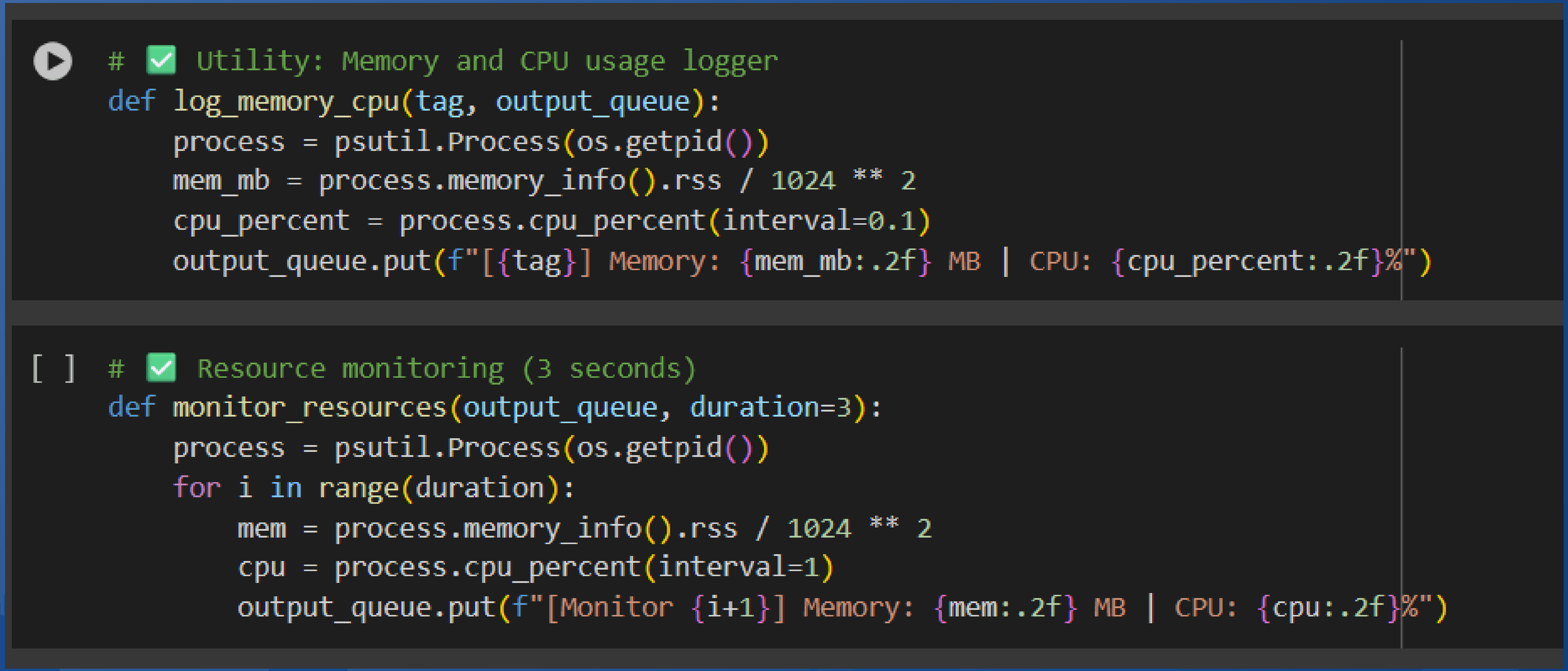
   Test and compare file reading speeds to simulate OS-level file handling.

3.Improve Memory Management

   Track how much memory each step uses — simulate what an OS does.

MEMORY MANAGEMENT

We use psutil to monitor the memory and CPU usage.

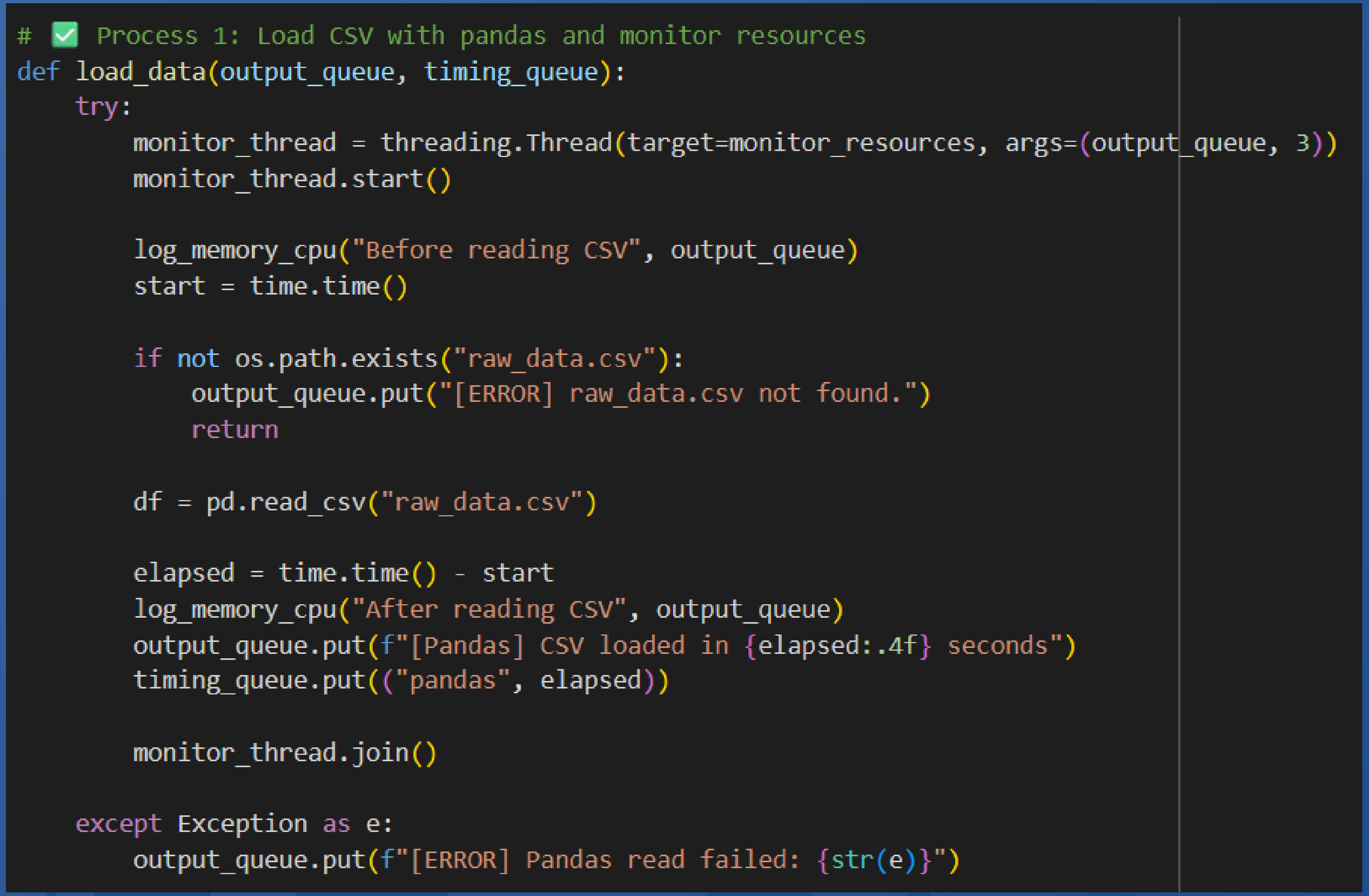


```
# ✅ Utility: Memory and CPU usage logger
def log_memory_cpu(tag, output_queue):
    process = psutil.Process(os.getpid())
    mem_mb = process.memory_info().rss / 1024 ** 2
    cpu_percent = process.cpu_percent(interval=0.1)
    output_queue.put(f"[{tag}] Memory: {mem_mb:.2f} MB | CPU: {cpu_percent:.2f}%")
```

```
# ✅ Resource monitoring (3 seconds)
def monitor_resources(output_queue, duration=3):
    process = psutil.Process(os.getpid())
    for i in range(duration):
        mem = process.memory_info().rss / 1024 ** 2
        cpu = process.cpu_percent(interval=1)
        output_queue.put(f"[Monitor {i+1}] Memory: {mem:.2f} MB | CPU: {cpu:.2f}%")
```

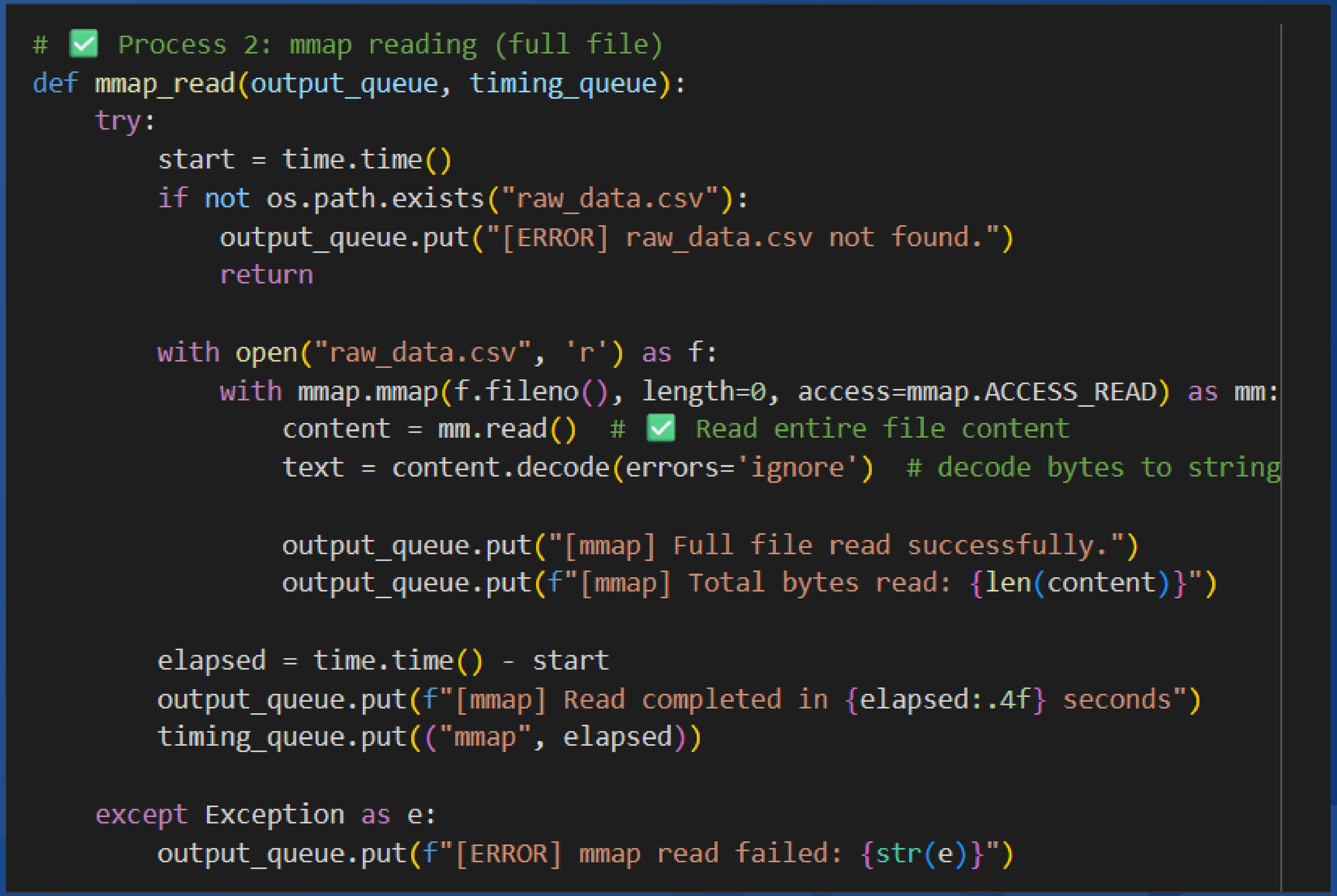
FILES I/O

Instead of downloading our file with panda, we use mmap instead. We also track the time it takes.



```python
# ✅ Process 1: Load CSV with pandas and monitor resources
def load_data(output_queue, timing_queue):
    try:
        monitor_thread = threading.Thread(target=monitor_resources, args=(output_queue, 3))
        monitor_thread.start()

        log_memory_cpu("Before reading CSV", output_queue)
        start = time.time()

        if not os.path.exists("raw_data.csv"):
            output_queue.put("[ERROR] raw_data.csv not found.")
            return

        df = pd.read_csv("raw_data.csv")

        elapsed = time.time() - start
        log_memory_cpu("After reading CSV", output_queue)
        output_queue.put(f"[Pandas] CSV loaded in {elapsed:.4f} seconds")
        timing_queue.put(("pandas", elapsed))

        monitor_thread.join()

    except Exception as e:
        output_queue.put(f"[ERROR] Pandas read failed: {str(e)}")
```

Load CSV file with panda

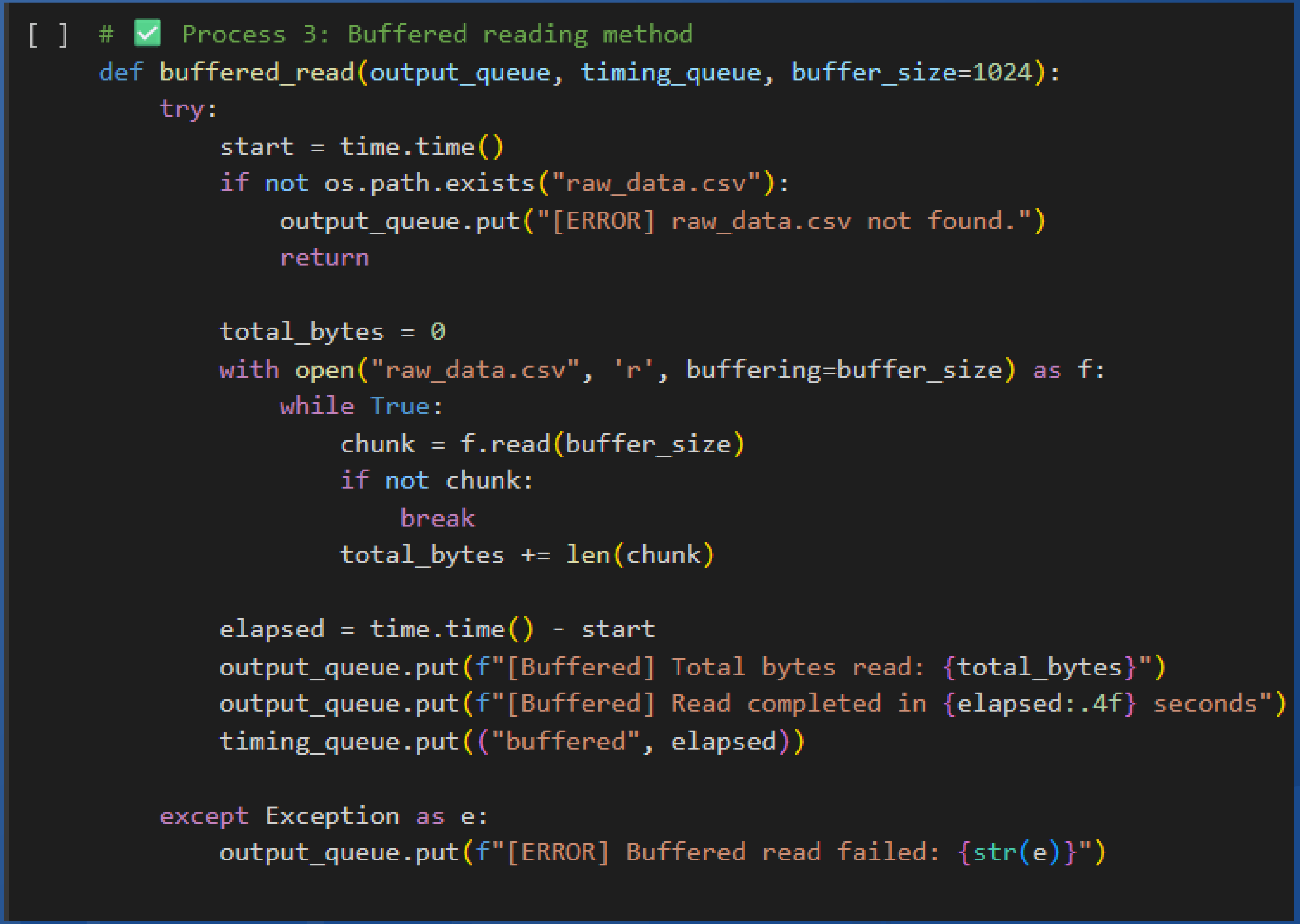


```python
# ✅ Process 2: mmap reading (full file)
def mmap_read(output_queue, timing_queue):
    try:
        start = time.time()
        if not os.path.exists("raw_data.csv"):
            output_queue.put("[ERROR] raw_data.csv not found.")
            return

        with open("raw_data.csv", 'r') as f:
            with mmap.mmap(f.fileno(), length=0, access=mmap.ACCESS_READ) as mm:
                content = mm.read()  # ✅ Read entire file content
                text = content.decode(errors='ignore')  # decode bytes to string

                output_queue.put("[mmap] Full file read successfully.")
                output_queue.put(f"[mmap] Total bytes read: {len(content)}")

        elapsed = time.time() - start
        output_queue.put(f"[mmap] Read completed in {elapsed:.4f} seconds")
        timing_queue.put(("mmap", elapsed))

    except Exception as e:
        output_queue.put(f"[ERROR] mmap read failed: {str(e)}")
```

Load CSV file with mmap

FILES I/O

Additionally, we try another method called ‘Buffer’

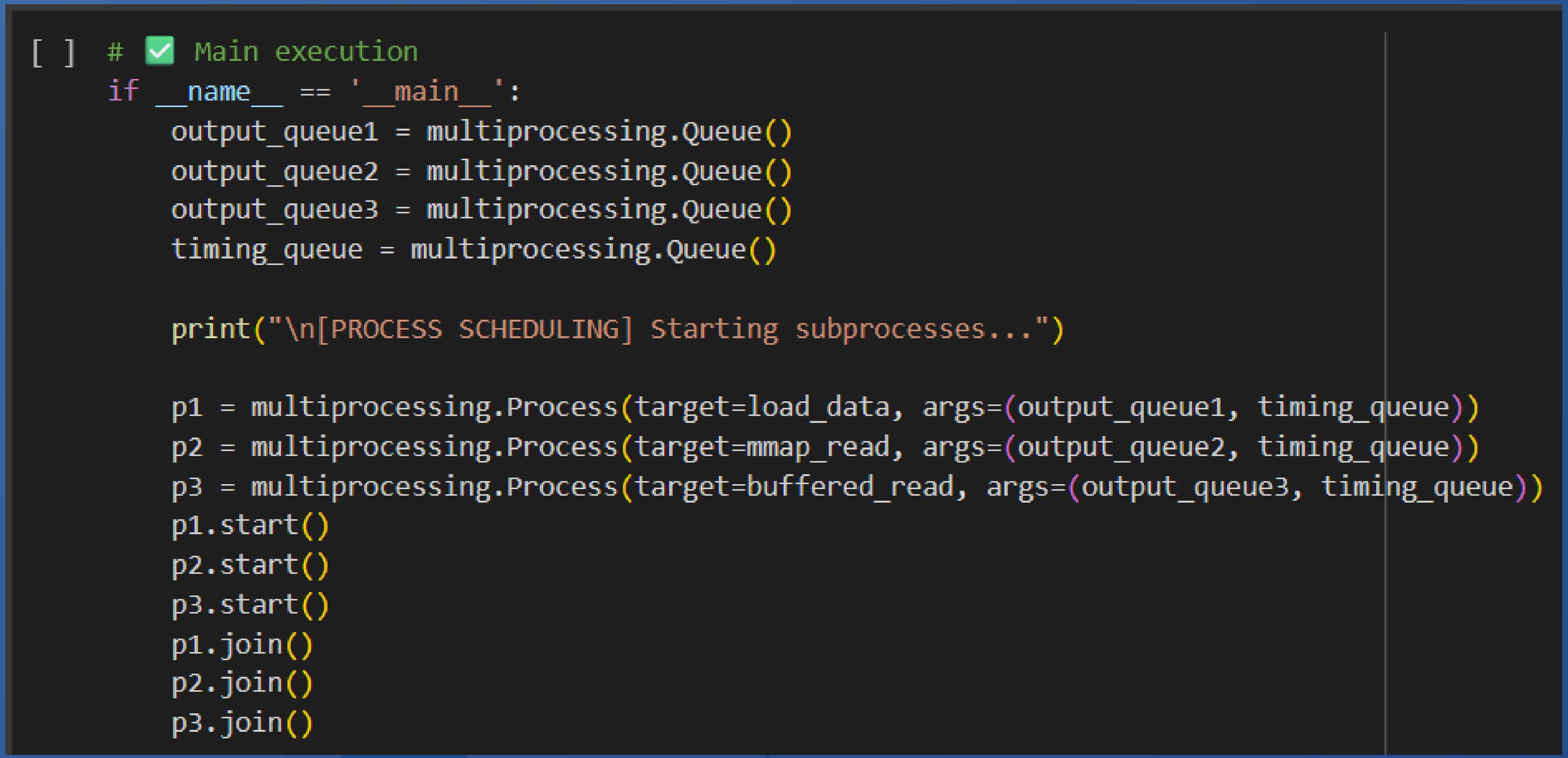


```python
# ✅ Process 3: Buffered reading method
def buffered_read(output_queue, timing_queue, buffer_size=1024):
    try:
        start = time.time()
        if not os.path.exists("raw_data.csv"):
            output_queue.put("[ERROR] raw_data.csv not found.")
            return

        total_bytes = 0
        with open("raw_data.csv", 'r', buffering=buffer_size) as f:
            while True:
                chunk = f.read(buffer_size)
                if not chunk:
                    break
                total_bytes += len(chunk)

        elapsed = time.time() - start
        output_queue.put(f"[Buffered] Total bytes read: {total_bytes}")
        output_queue.put(f"[Buffered] Read completed in {elapsed:.4f} seconds")
        timing_queue.put(("buffered", elapsed))

    except Exception as e:
        output_queue.put(f"[ERROR] Buffered read failed: {str(e)}")
```

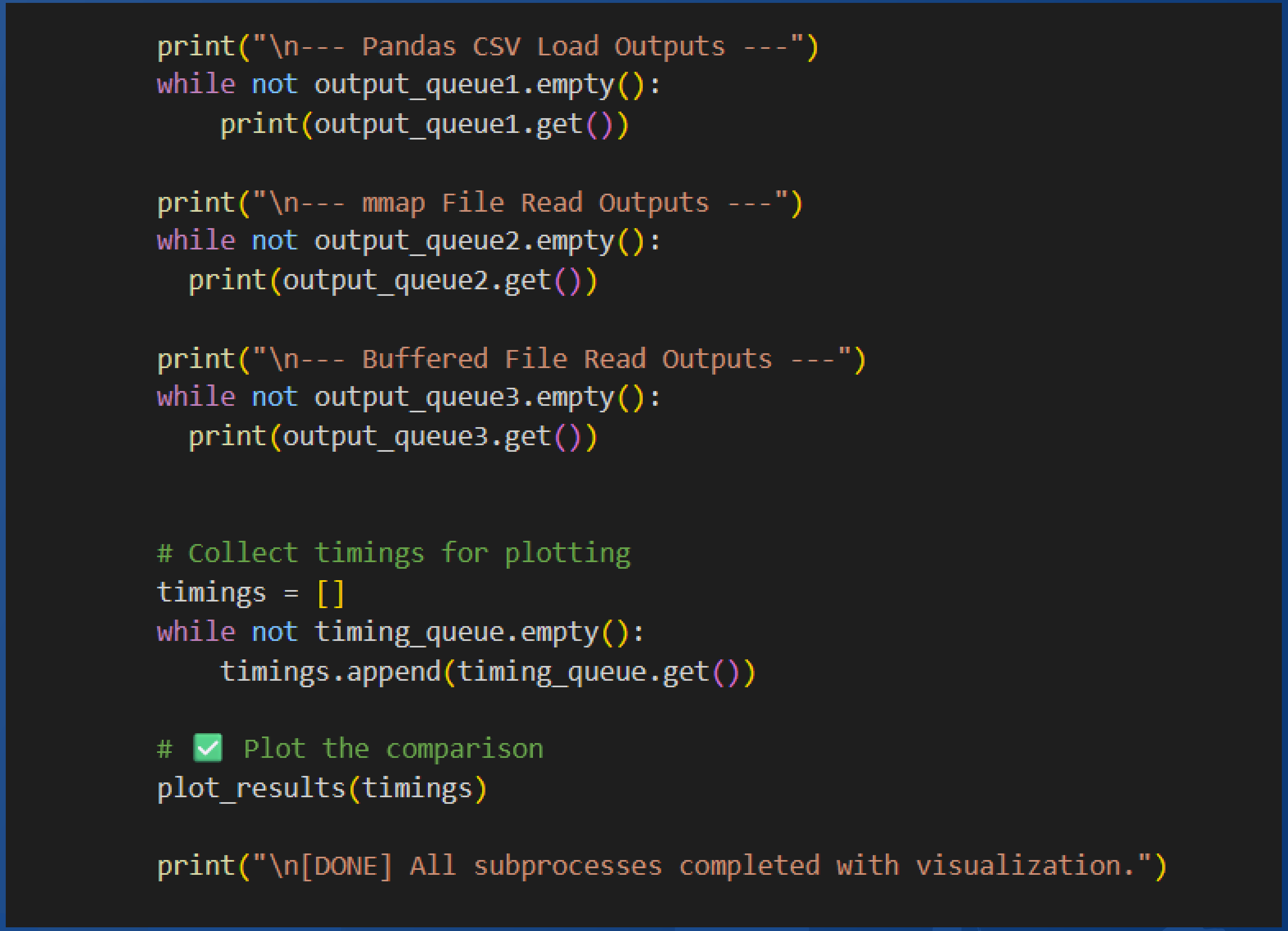
Load CSV with Buffer

PROCESS SCHEDULING

We also updated our scheduling to handle the additional process



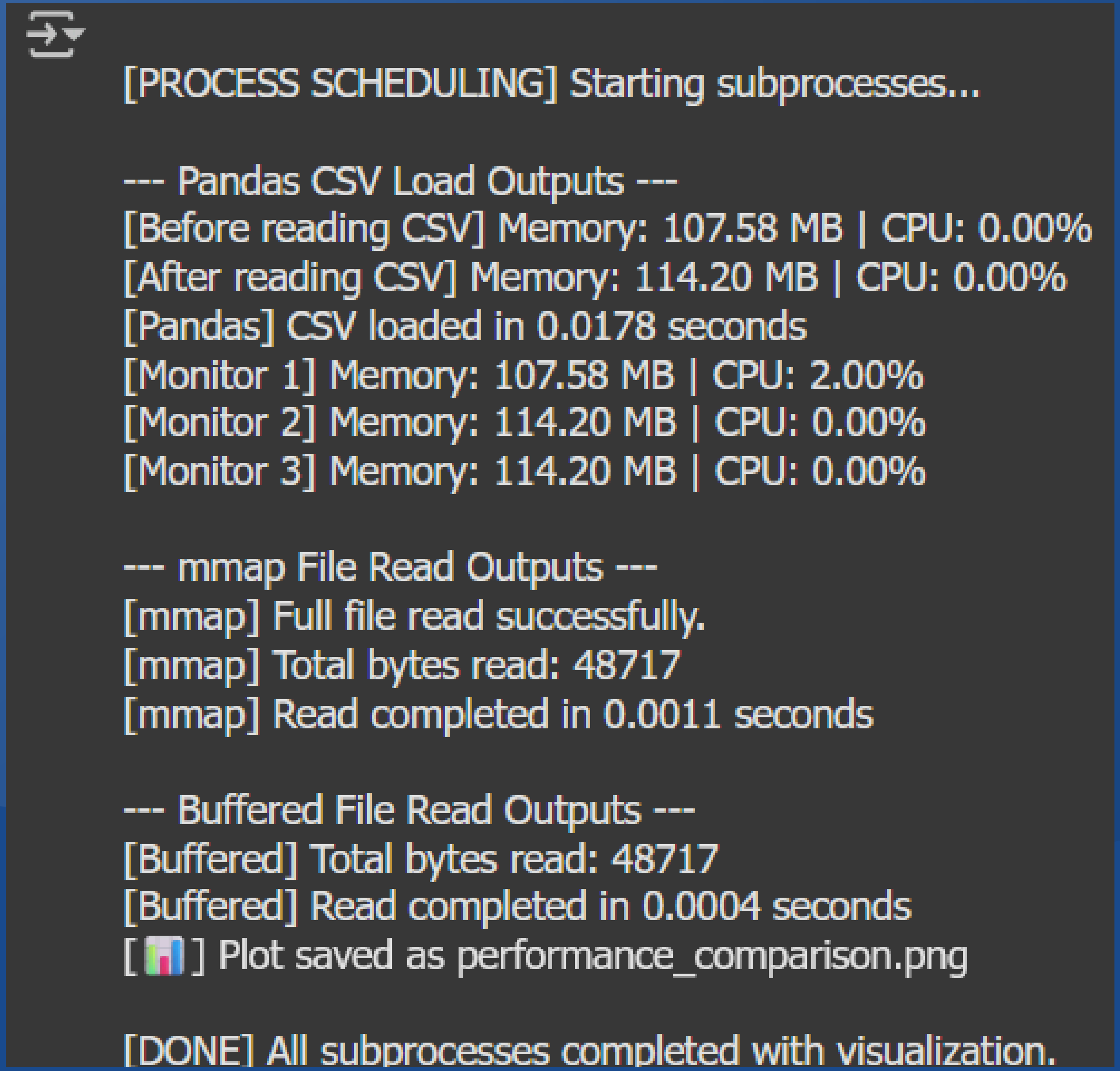
```
[ ]  # ✅ Main execution
     if __name__ == '__main__':
         output_queue1 = multiprocessing.Queue()
         output_queue2 = multiprocessing.Queue()
         output_queue3 = multiprocessing.Queue()
         timing_queue = multiprocessing.Queue()

         print("\n[PROCESS SCHEDULING] Starting subprocesses...")

         p1 = multiprocessing.Process(target=load_data, args=(output_queue1, timing_queue))
         p2 = multiprocessing.Process(target=mmap_read, args=(output_queue2, timing_queue))
         p3 = multiprocessing.Process(target=buffered_read, args=(output_queue3, timing_queue))
         p1.start()
         p2.start()
         p3.start()
         p1.join()
         p2.join()
         p3.join()
```



```
print("\n--- Pandas CSV Load Outputs ---")
while not output_queue1.empty():
    print(output_queue1.get())

print("\n--- mmap File Read Outputs ---")
while not output_queue2.empty():
  print(output_queue2.get())

print("\n--- Buffered File Read Outputs ---")
while not output_queue3.empty():
  print(output_queue3.get())


# Collect timings for plotting
timings = []
while not timing_queue.empty():
    timings.append(timing_queue.get())

# ✅ Plot the comparison
plot_results(timings)

print("\n[DONE] All subprocesses completed with visualization.")
```

PROCESS SCHEDULING

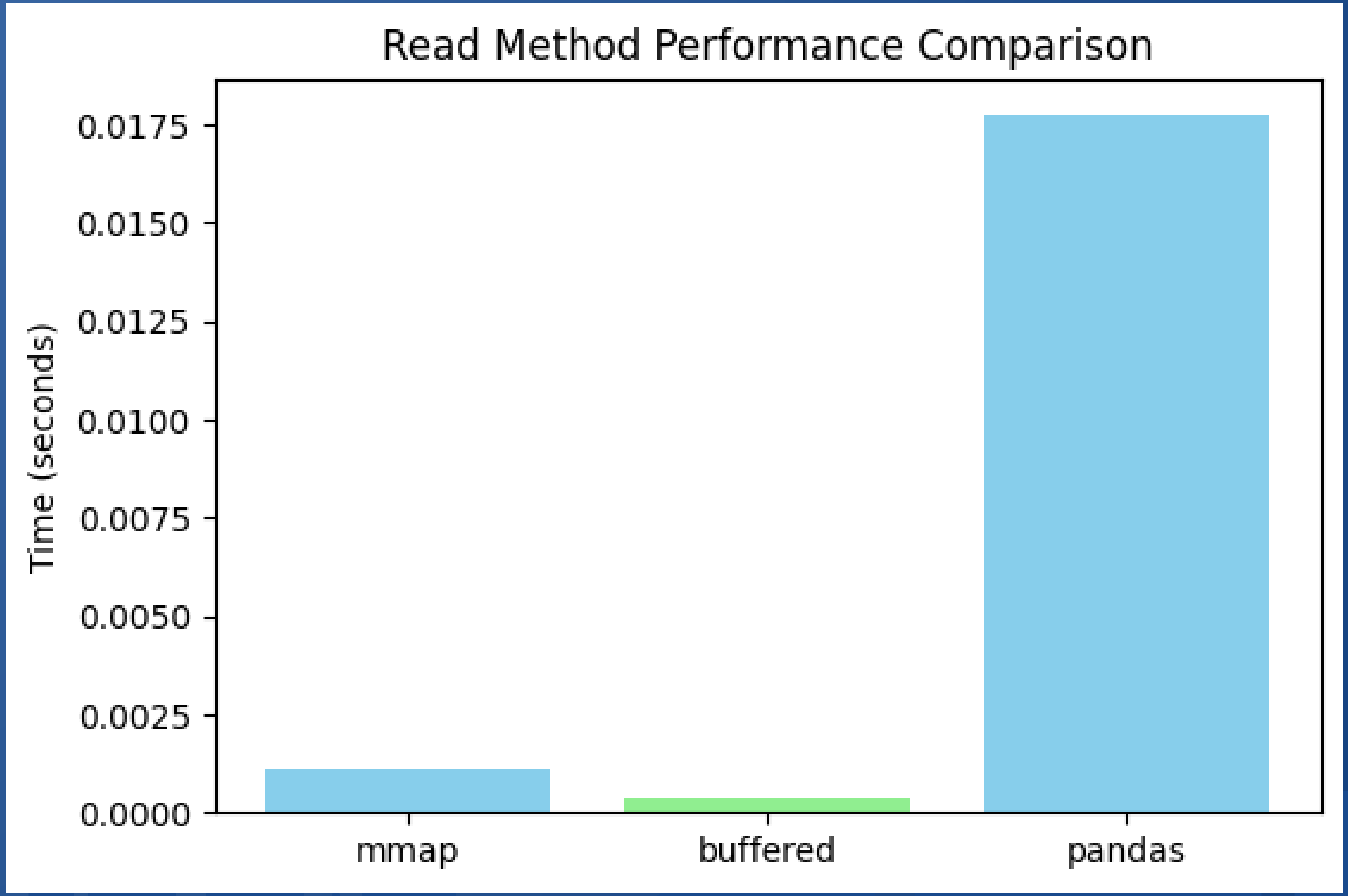
The result of the run, including the memory usage and the time used for each process



```
[PROCESS SCHEDULING] Starting subprocesses...

--- Pandas CSV Load Outputs ---
[Before reading CSV] Memory: 107.58 MB | CPU: 0.00%
[After reading CSV] Memory: 114.20 MB | CPU: 0.00%
[Pandas] CSV loaded in 0.0178 seconds
[Monitor 1] Memory: 107.58 MB | CPU: 2.00%
[Monitor 2] Memory: 114.20 MB | CPU: 0.00%
[Monitor 3] Memory: 114.20 MB | CPU: 0.00%

--- mmap File Read Outputs ---
[mmap] Full file read successfully.
[mmap] Total bytes read: 48717
[mmap] Read completed in 0.0011 seconds

--- Buffered File Read Outputs ---
[Buffered] Total bytes read: 48717
[Buffered] Read completed in 0.0004 seconds
[📊] Plot saved as performance_comparison.png

[DONE] All subprocesses completed with visualization.
```

RESULT BENCHMARK

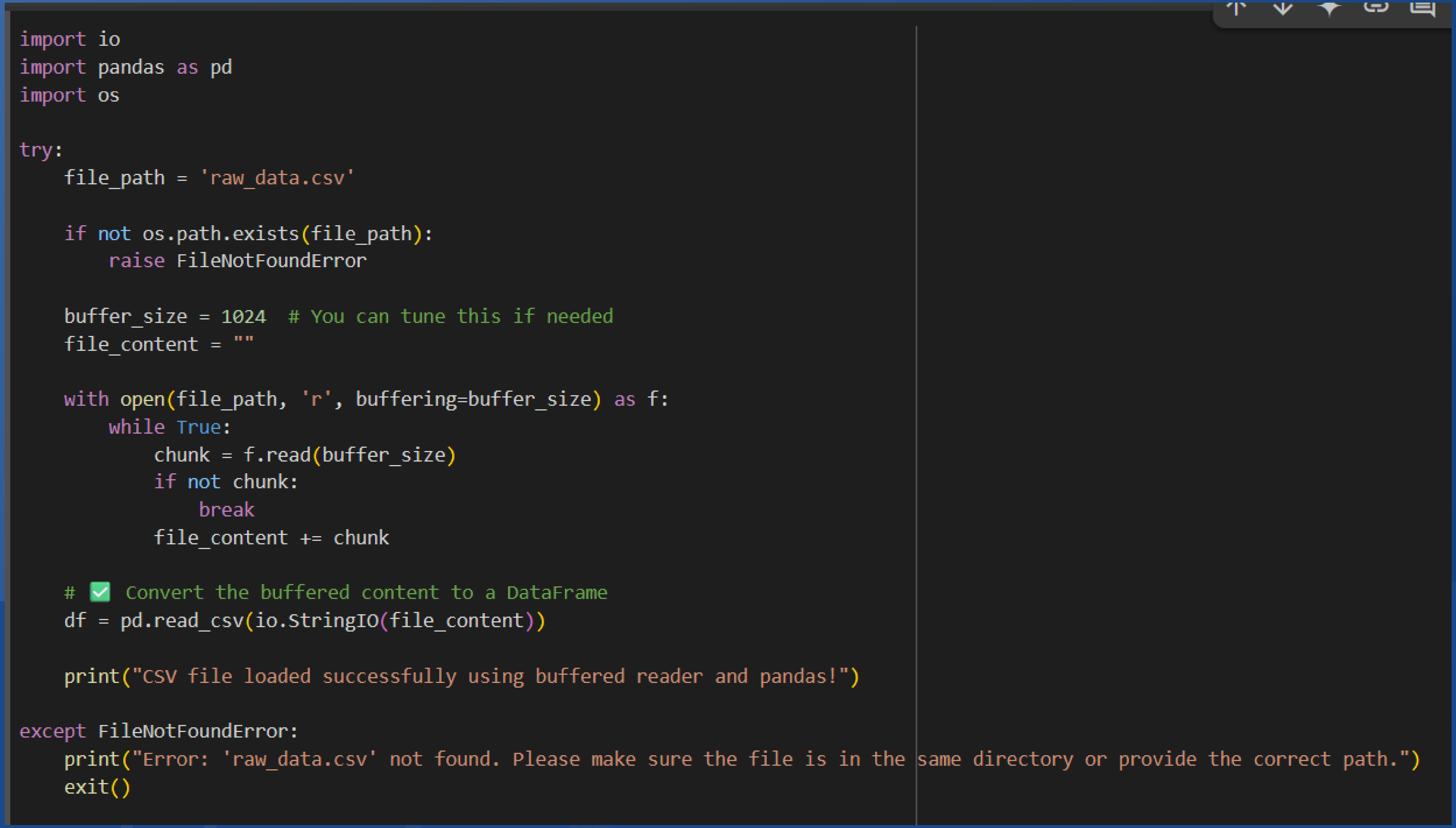
Performance different between these three methods.



DATA PROCESSING

READ THE DATA

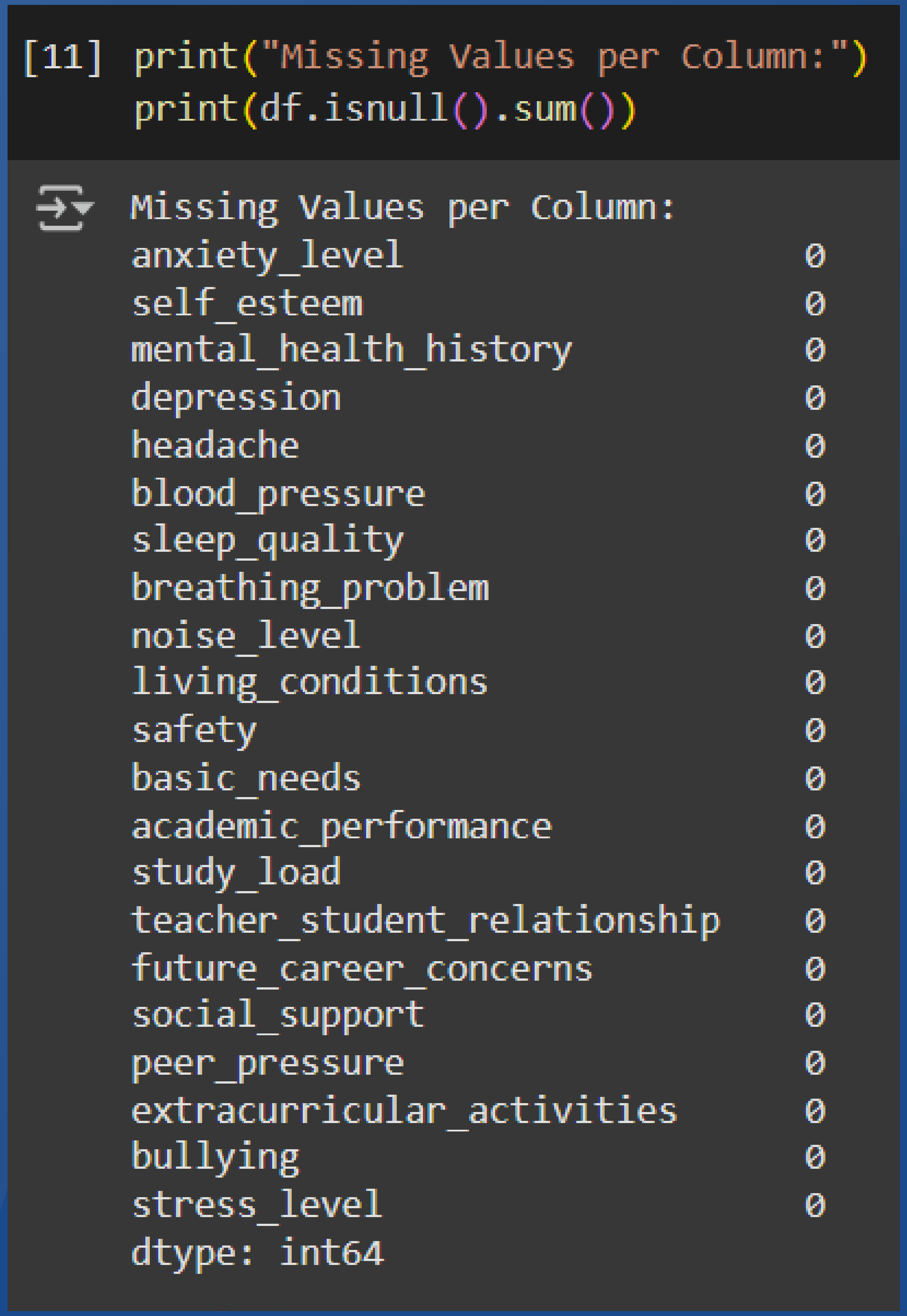
Using the best methods we discovered (Panda and Buffer)



```python
import io
import pandas as pd
import os

try:
    file_path = 'raw_data.csv'

    if not os.path.exists(file_path):
        raise FileNotFoundError

    buffer_size = 1024  # You can tune this if needed
    file_content = ""

    with open(file_path, 'r', buffering=buffer_size) as f:
        while True:
            chunk = f.read(buffer_size)
            if not chunk:
                break
            file_content += chunk

    # ✅ Convert the buffered content to a DataFrame
    df = pd.read_csv(io.StringIO(file_content))

    print("CSV file loaded successfully using buffered reader and pandas!")

except FileNotFoundError:
    print("Error: 'raw_data.csv' not found. Please make sure the file is in the same directory or provide the correct path.")
    exit()
```

IDENTIFY NULL VALUE

There is none.

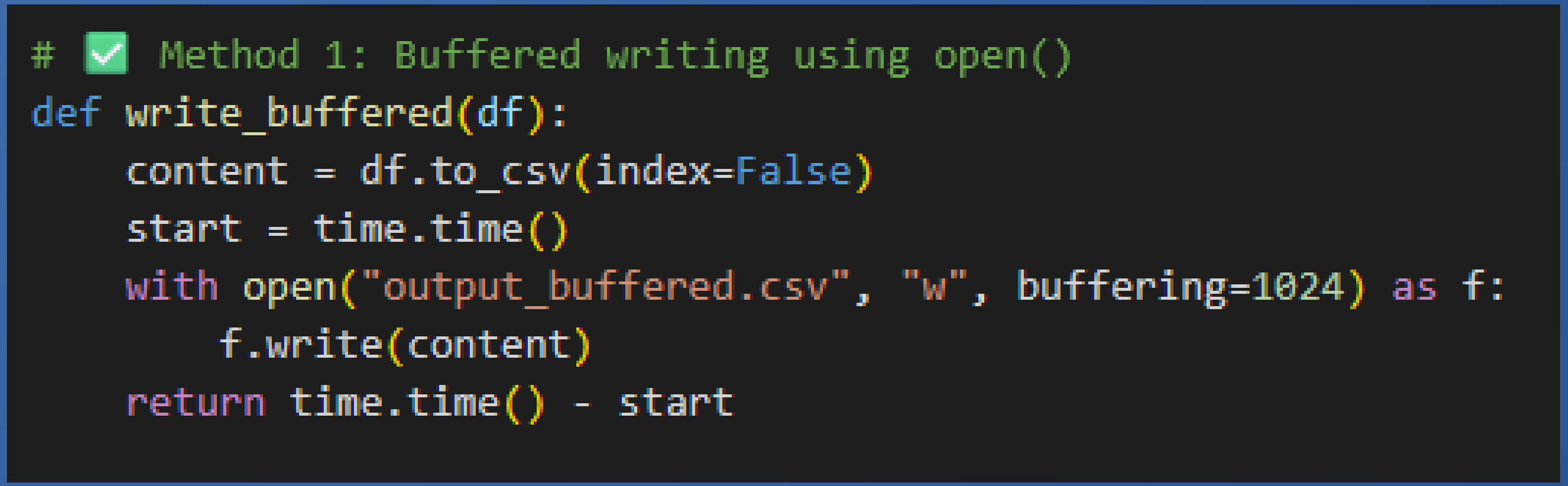


```
[11] print("Missing Values per Column:")
     print(df.isnull().sum())
```

```
Missing Values per Column:
anxiety_level                   0
self_esteem                     0
mental_health_history           0
depression                      0
headache                        0
blood_pressure                  0
sleep_quality                   0
breathing_problem               0
noise_level                     0
living_conditions               0
safety                          0
basic_needs                     0
academic_performance            0
study_load                      0
teacher_student_relationship    0
future_career_concerns          0
social_support                  0
peer_pressure                   0
extracurricular_activities      0
bullying                        0
stress_level                    0
dtype: int64
```

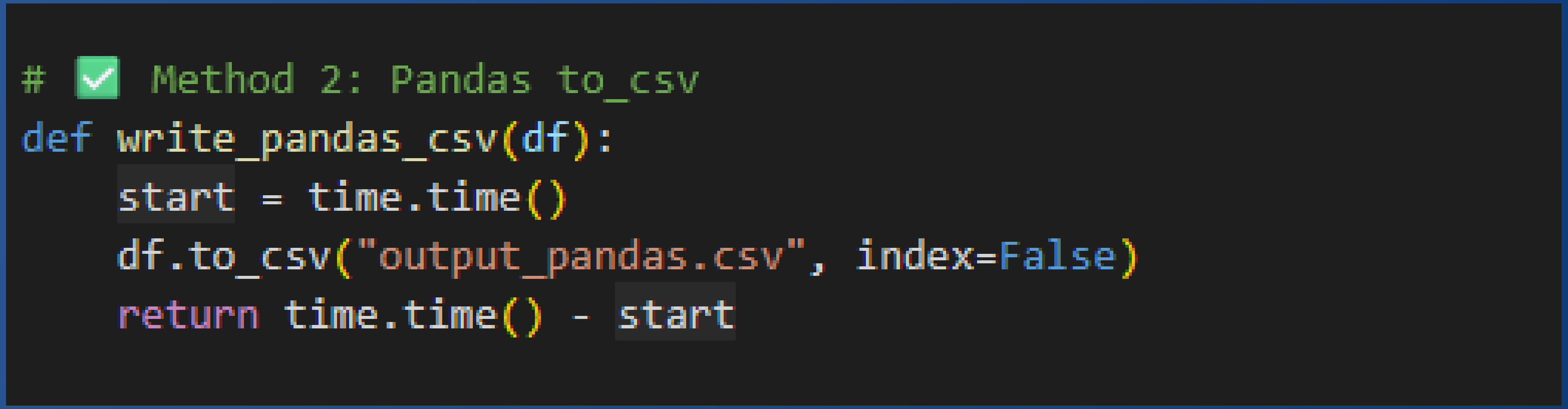
WRITE THE DATA

Save the data into processed file. We used and compared three methods for this task.



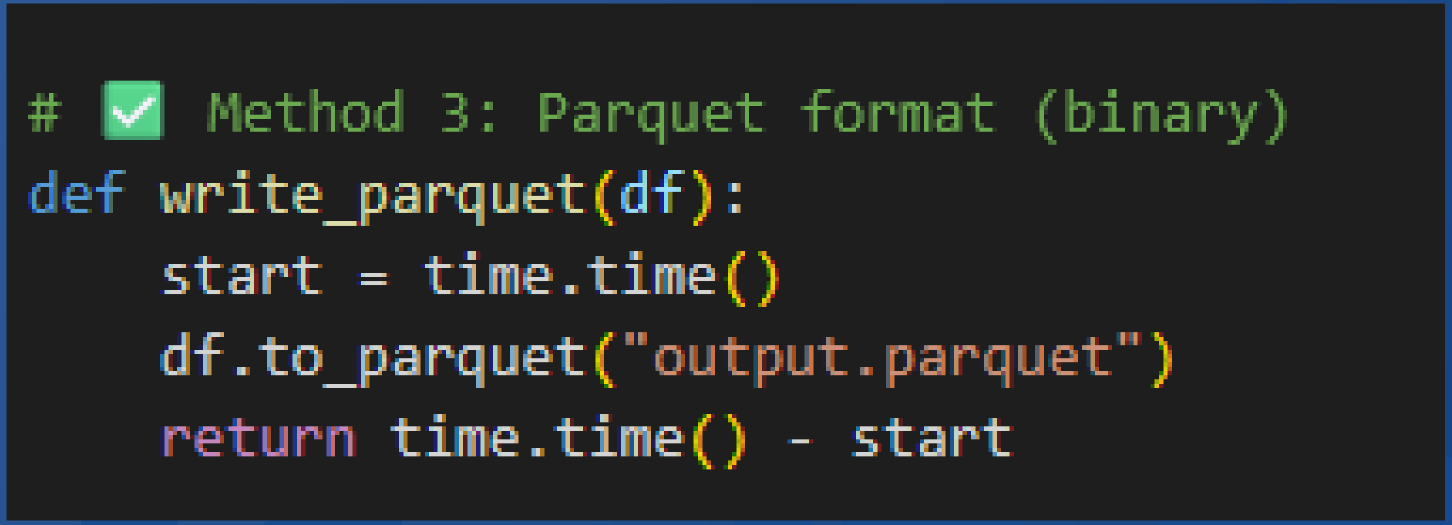
```python
# ✅ Method 1: Buffered writing using open()
def write_buffered(df):
    content = df.to_csv(index=False)
    start = time.time()
    with open("output_buffered.csv", "w", buffering=1024) as f:
        f.write(content)
    return time.time() - start
```

Buffer



```python
# ✅ Method 2: Pandas to_csv
def write_pandas_csv(df):
    start = time.time()
    df.to_csv("output_pandas.csv", index=False)
    return time.time() - start
```

Pandas

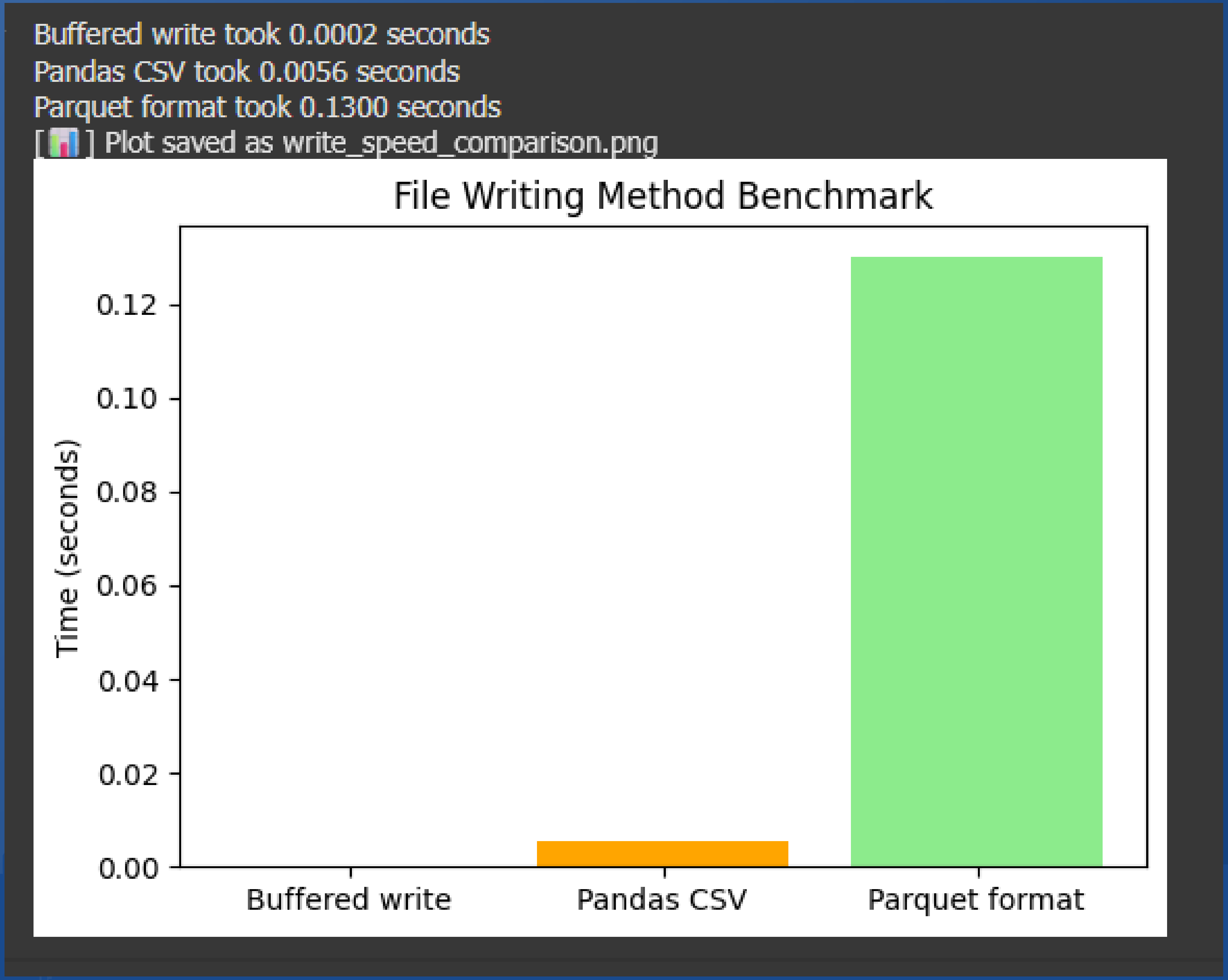


```python
# ✅ Method 3: Parquet format (binary)
def write_parquet(df):
    start = time.time()
    df.to_parquet("output.parquet")
    return time.time() - start
```

Parquet

WRITE THE DATA

Time used by each method

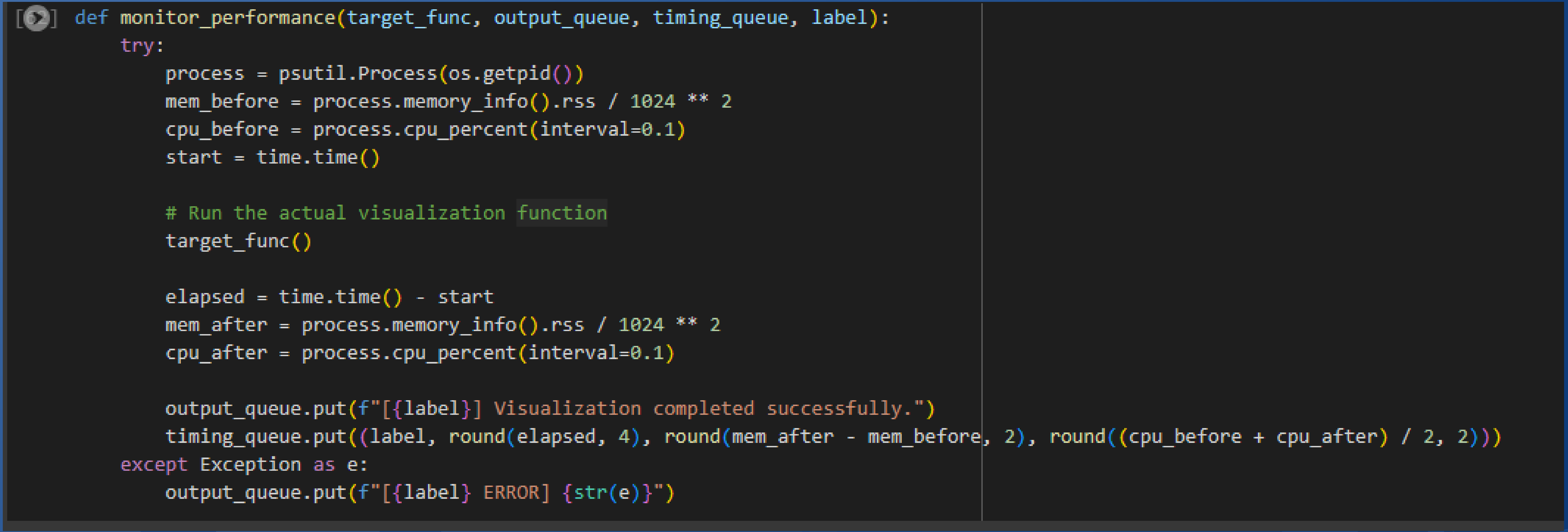


```
Buffered write took 0.0002 seconds
Pandas CSV took 0.0056 seconds
Parquet format took 0.1300 seconds
[📊] Plot saved as write_speed_comparison.png
```

VISUALIZATION

RESOURCE MANAGEMENT

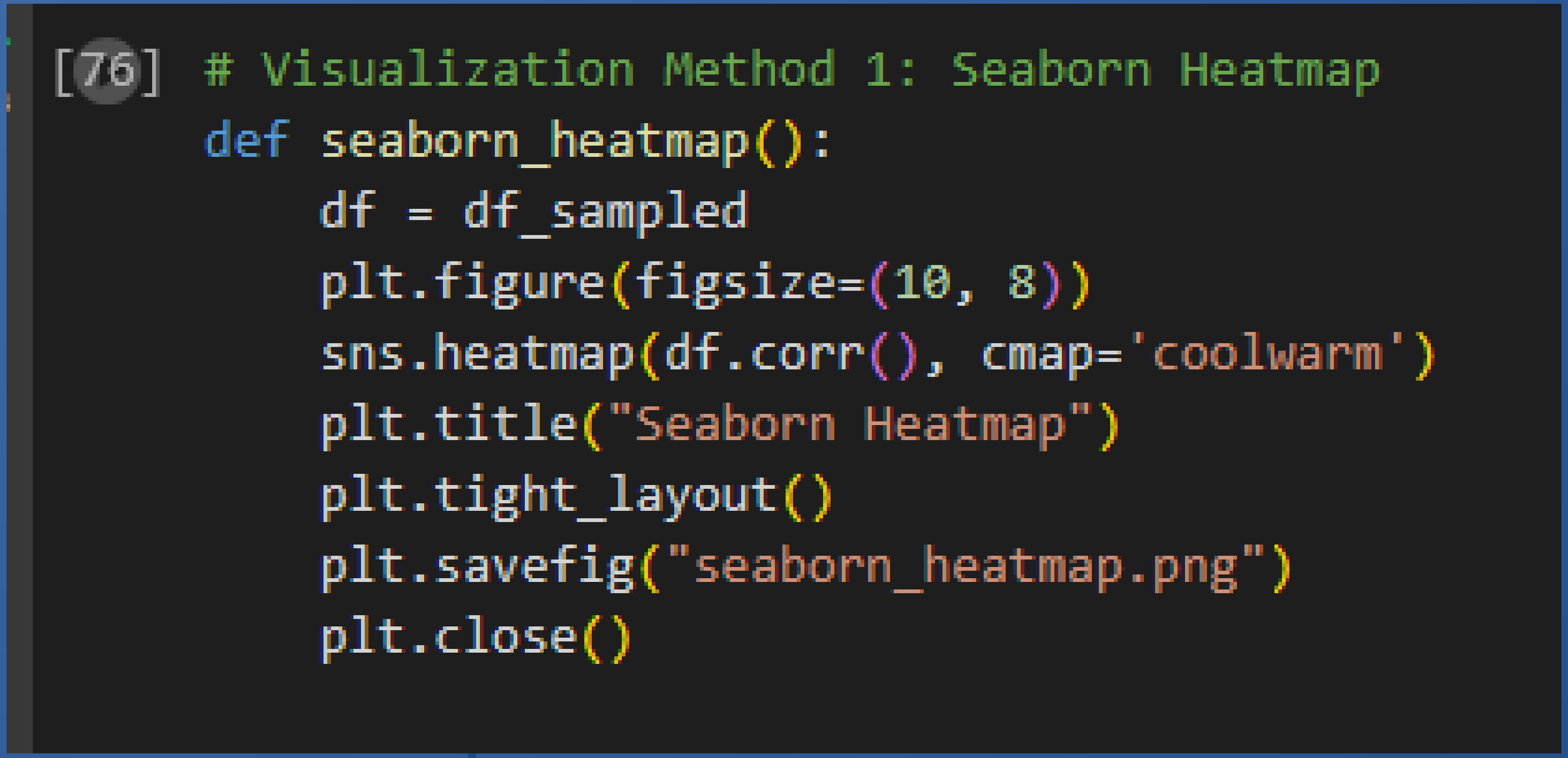
We used psuilt and process to monitor each method performance



```python
def monitor_performance(target_func, output_queue, timing_queue, label):
    try:
        process = psutil.Process(os.getpid())
        mem_before = process.memory_info().rss / 1024 ** 2
        cpu_before = process.cpu_percent(interval=0.1)
        start = time.time()

        # Run the actual visualization function
        target_func()

        elapsed = time.time() - start
        mem_after = process.memory_info().rss / 1024 ** 2
        cpu_after = process.cpu_percent(interval=0.1)

        output_queue.put(f"[{label}] Visualization completed successfully.")
        timing_queue.put((label, round(elapsed, 4), round(mem_after - mem_before, 2), round((cpu_before + cpu_after) / 2, 2)))
    except Exception as e:
        output_queue.put(f"[{label} ERROR] {str(e)}")
```

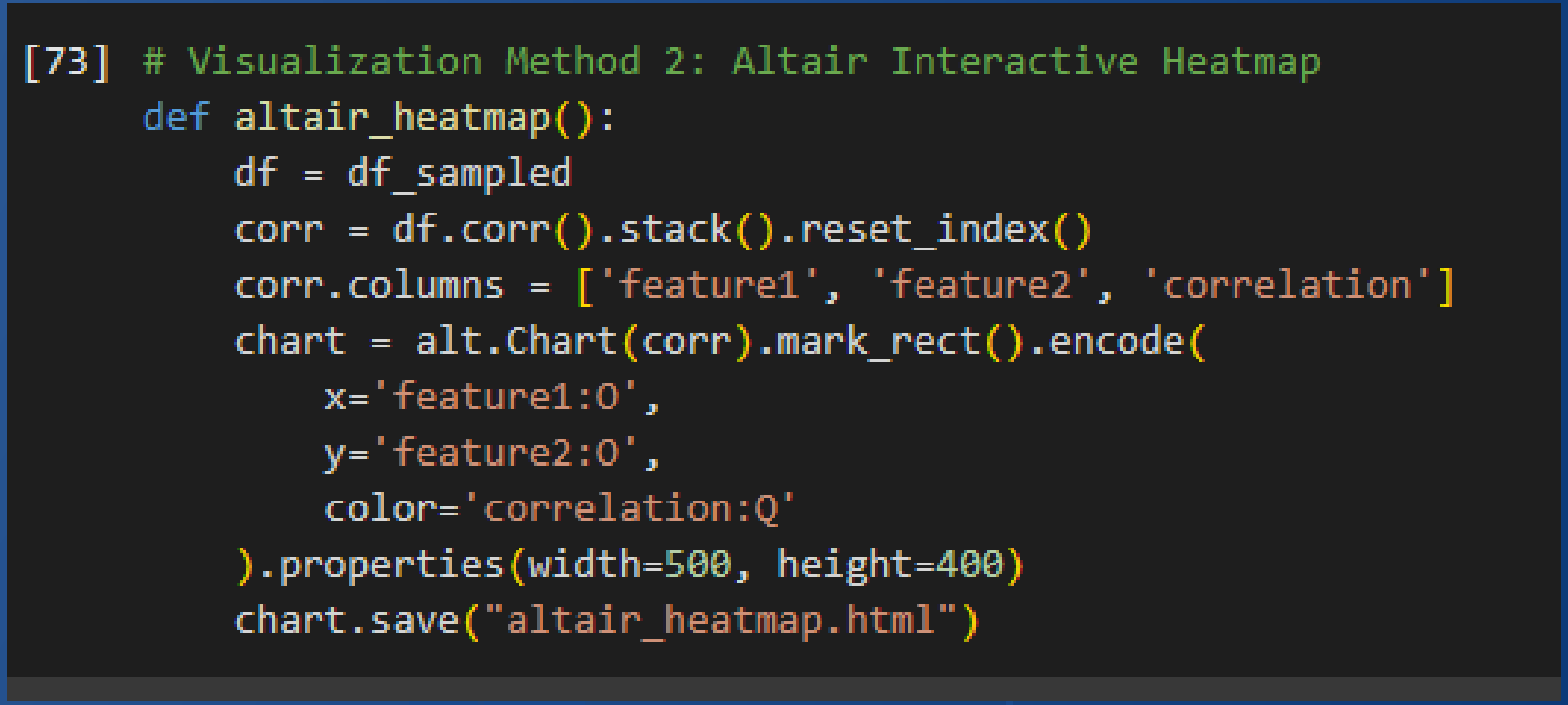
VISUALIZATION

We selected 2 methods; Seaborn and Altair



```
[76] # Visualization Method 1: Seaborn Heatmap
     def seaborn_heatmap():
         df = df_sampled
         plt.figure(figsize=(10, 8))
         sns.heatmap(df.corr(), cmap='coolwarm')
         plt.title("Seaborn Heatmap")
         plt.tight_layout()
         plt.savefig("seaborn_heatmap.png")
         plt.close()
```

Seaborn

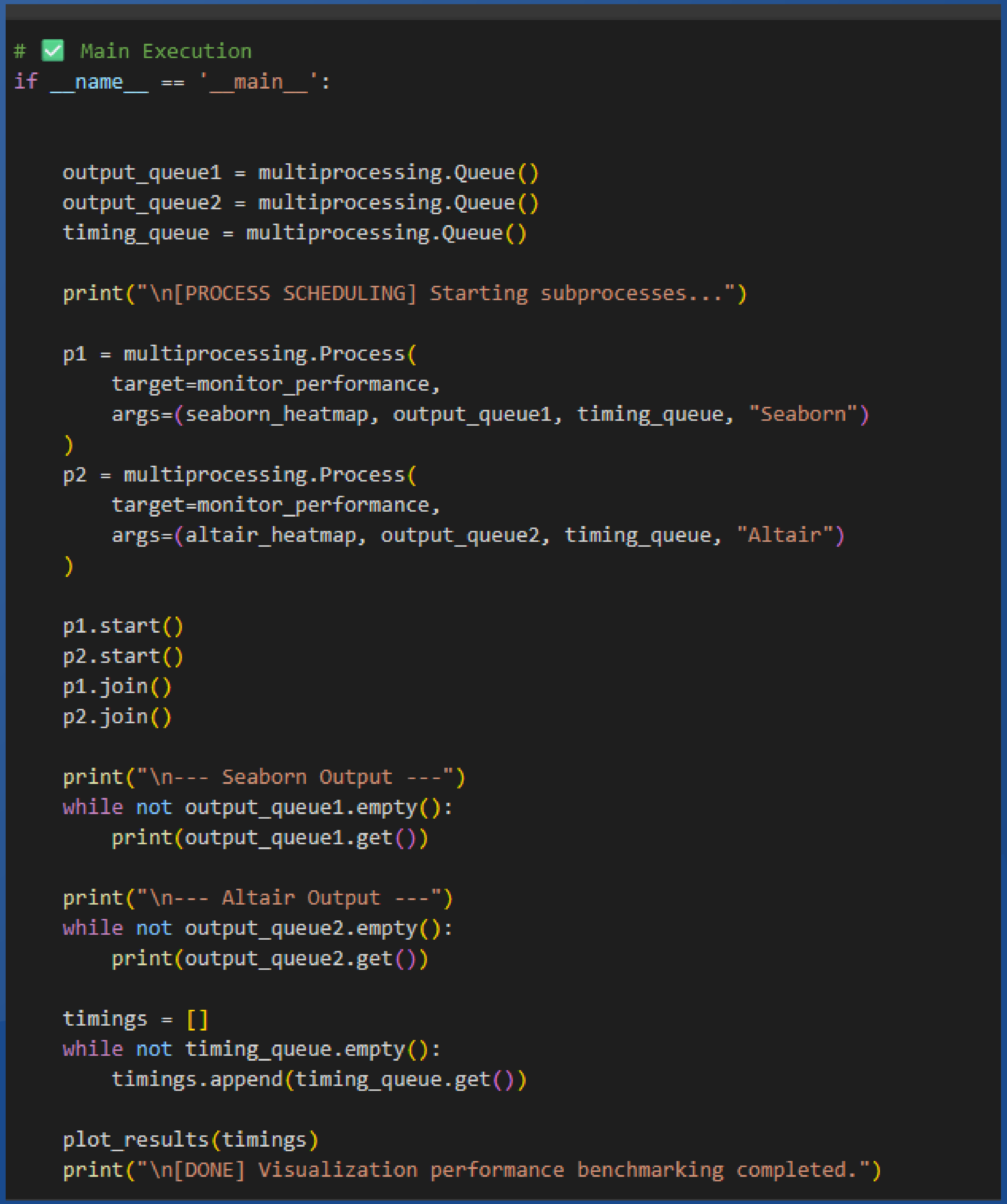


```
[73] # Visualization Method 2: Altair Interactive Heatmap
     def altair_heatmap():
         df = df_sampled
         corr = df.corr().stack().reset_index()
         corr.columns = ['feature1', 'feature2', 'correlation']
         chart = alt.Chart(corr).mark_rect().encode(
             x='feature1:O',
             y='feature2:O',
             color='correlation:Q'
         ).properties(width=500, height=400)
         chart.save("altair_heatmap.html")
```

Altair

MAIN EXECUTION

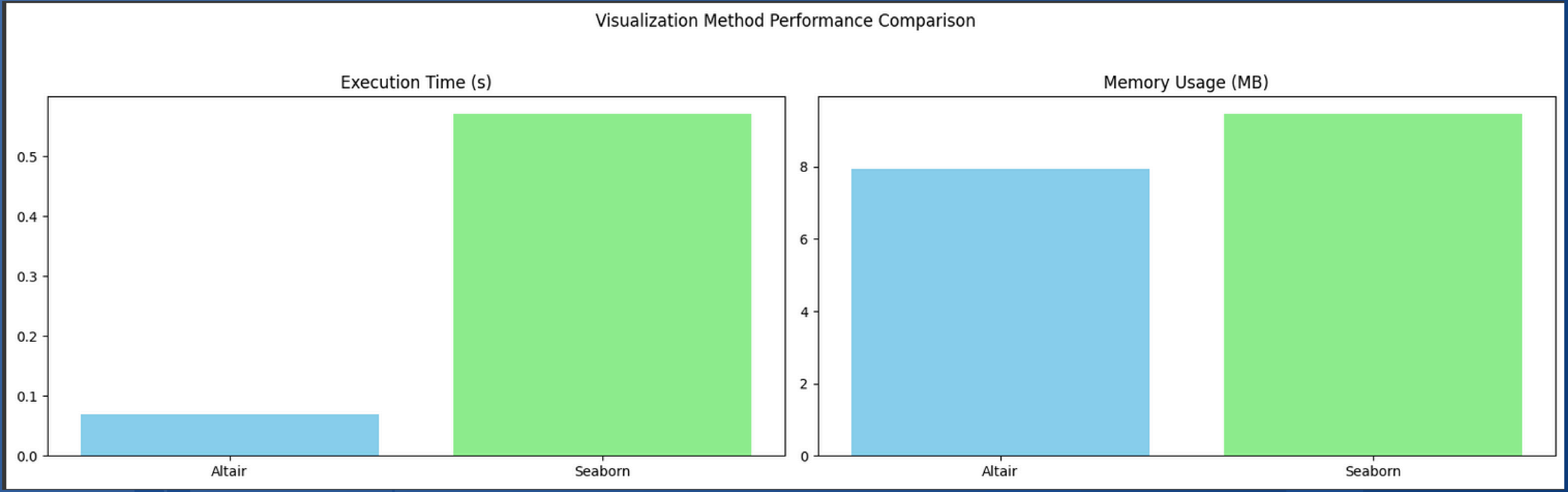
Also include the process scheduling from before



```
# ✅ Main Execution
if __name__ == '__main__':


    output_queue1 = multiprocessing.Queue()
    output_queue2 = multiprocessing.Queue()
    timing_queue = multiprocessing.Queue()

    print("\n[PROCESS SCHEDULING] Starting subprocesses...")

    p1 = multiprocessing.Process(
        target=monitor_performance,
        args=(seaborn_heatmap, output_queue1, timing_queue, "Seaborn")
    )
    p2 = multiprocessing.Process(
        target=monitor_performance,
        args=(altair_heatmap, output_queue2, timing_queue, "Altair")
    )

    p1.start()
    p2.start()
    p1.join()
    p2.join()

    print("\n--- Seaborn Output ---")
    while not output_queue1.empty():
        print(output_queue1.get())

    print("\n--- Altair Output ---")
    while not output_queue2.empty():
        print(output_queue2.get())

    timings = []
    while not timing_queue.empty():
        timings.append(timing_queue.get())

    plot_results(timings)
    print("\n[DONE] Visualization performance benchmarking completed.")
```

RESULT BENCHMARK

The execution time and memory usage of each method



THANK YOU