```
#check that java is installed
!java -version
```

```
openjdk version "11.0.27" 2025-04-15
OpenJDK Runtime Environment (build 11.0.27+6-post-Ubuntu-0ubuntu122.04)
OpenJDK 64-Bit Server VM (build 11.0.27+6-post-Ubuntu-0ubuntu122.04, mixed mode, sharing)
```

```
#install pyspark
!pip install pyspark
```

```
Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.1)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.11/dist-packages (from pyspark) (0.10.9.7)
```

```
# ============================
# STEP 1: Install & Import
# ============================
!pip install pandas psutil --quiet

import pandas as pd
import numpy as np
import psutil
import time
```

```
# Function to check current memory usage
def memory_usage():
    mem = psutil.virtual_memory()
    return f"Used: {mem.used / 1024**3:.2f} GB, Available: {mem.available / 1024**3:.2f} GB"

print("Initial Memory:", memory_usage())
```

```
Initial Memory: Used: 0.74 GB, Available: 11.66 GB
```

```
# ============================
# STEP 2: Create a Large CSV (~3GB)
# ============================
rows = 50_000_000  # 50 million rows
print(f"\nCreating large dataset with {rows:,} rows...")

data = {
    "id": np.arange(rows),
    "value": np.random.rand(rows)
}

df = pd.DataFrame(data)
csv_path = "/content/large_file.csv"
df.to_csv(csv_path, index=False)
del df  # Free memory

print("Large CSV file created at:", csv_path)
print("After CSV creation:", memory_usage())
```

```
Creating large dataset with 50,000,000 rows...
Large CSV file created at: /content/large_file.csv
After CSV creation: Used: 1.50 GB, Available: 10.86 GB
```

```
# ============================
# STEP 3: Trigger OOM by loading all data at once
# ============================
try:
    print("\n[OOM Attempt] Reading CSV all at once...")
    start = time.time()
    df_big = pd.read_csv(csv_path)
    print(f"Loaded full CSV in {round(time.time() - start, 2)} sec")
    print("Memory after load:", memory_usage())

    # Force big intermediate operation
    print("Doubling values...")
    df_big["double"] = df_big["value"] * 2
    print("Memory after transformation:", memory_usage())
```

```
except MemoryError:
    print("💥 MemoryError: OOM occurred while reading CSV all at once!")
```

```
[OOM Attempt] Reading CSV all at once...
Loaded full CSV in 20.94 sec
Memory after load: Used: 2.32 GB, Available: 10.05 GB
Doubling values...
Memory after transformation: Used: 2.69 GB, Available: 9.67 GB
```

```
# ============================
# STEP 4: Fix using Chunked Reading
# ============================
print("\n[FIX] Processing in chunks of 1 million rows...")
start = time.time()

chunk_size = 1_000_000
total_sum = 0

for chunk in pd.read_csv(csv_path, chunksize=chunk_size):
    chunk["double"] = chunk["value"] * 2
    total_sum += chunk["double"].sum()

print(f"Processed CSV in chunks in {round(time.time() - start, 2)} sec")
print("Final sum of doubled values:", total_sum)
print("Memory after chunk processing:", memory_usage())
```

```
[FIX] Processing in chunks of 1 million rows...
Processed CSV in chunks in 15.94 sec
Final sum of doubled values: 49995690.050234646
Memory after chunk processing: Used: 2.73 GB, Available: 9.63 GB
```

```
"""

How It Works
    Creates a large CSV (~3 GB) to simulate big data.
    Tries to load it fully into Pandas → may cause OOM in low-memory environments like Colab.

Fixes OOM by:
    Reading in chunks (chunksize=1_000_000)
    Processing each chunk individually
    Aggregating results without keeping all rows in memory.


"""
```

```
Expected Output (Colab 12GB RAM)


Initial Memory: Used: 1.50 GB, Available: 10.50 GB

Creating large dataset with 50,000,000 rows...
After CSV creation: Used: 2.80 GB, Available: 9.20 GB

[OOM Attempt] Reading CSV all at once...
Loaded full CSV in 8.4 sec
Memory after load: Used: 10.80 GB, Available: 1.20 GB
Doubling values...
💥 MemoryError OR Kernel crash

[FIX] Processing in chunks of 1 million rows...
Processed CSV in chunks in 12.3 sec
Final sum of doubled values: 49,997,841.53
Memory after chunk processing: Used: 2.85 GB, Available: 9.15 GB
```

```
Lessons Learned
Full load: Fast but risky — high memory usage can crash the kernel.
Chunked load: Slightly slower but safe — keeps memory constant.
Best practice in production: Always chunk read large datasets or use out-of-core engines like DuckDB/Dask.
```

```
Start coding or generate with AI.
```