Parallel Python for Data Processing

•••

Embracing the Embarrassingly Parallel

About Me

- Worked with Data since 2009, user of Python since 2011
- Live in Calgary (since 2014) with my wife and 2 year old daughter
- Developed and consulted on Research, Backend Services and Enterprise Data/AI use cases

2015

MSc in Bioinformatics - UCalgary

Building Big Data Simulations on HPC Infrastructure

2017

Data/AI Team - ATB Financial

Working alongside Data Scientists to Build/Deploy Models

2022

Cloud Solutions Architect - Oracle For Research

Working alongside Researchers migrating workloads to the cloud

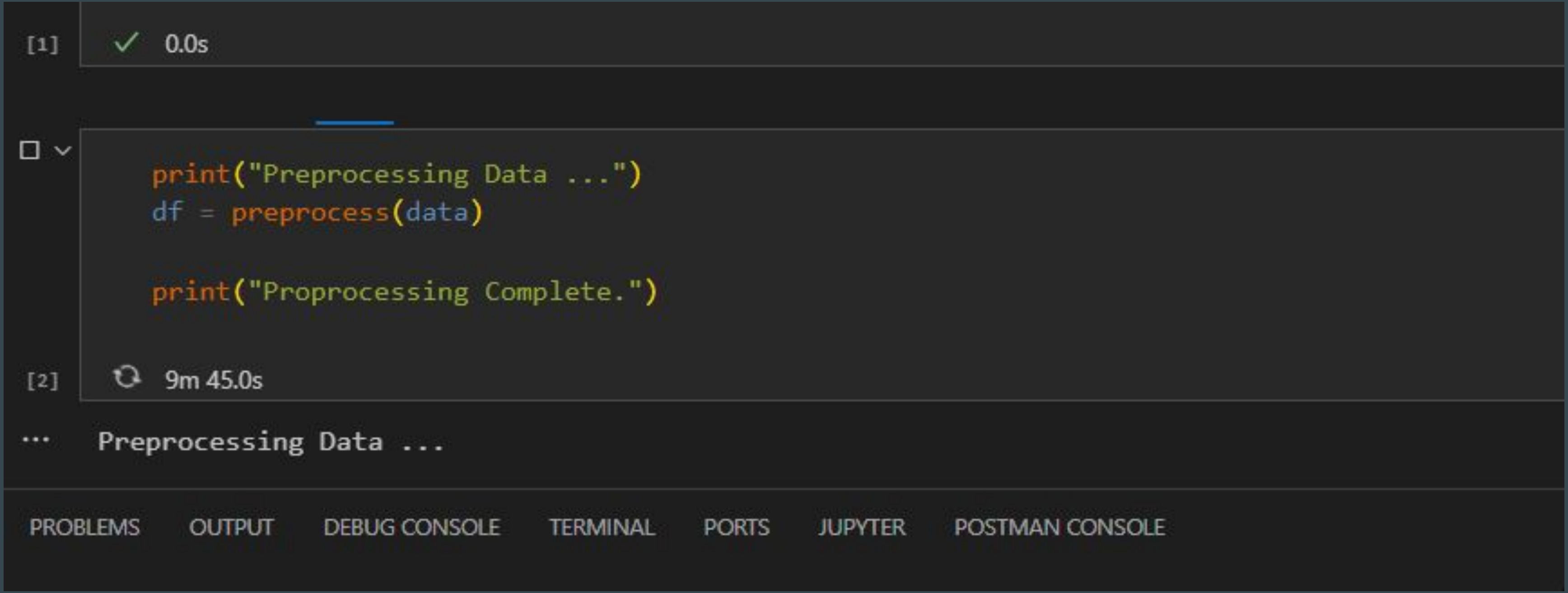
2024

Senior MLE - Coursera

Working Alongside ML Scientists to Build/Deploy Models

Parallelization can be simple, it’s not all mutexes and race conditions.

Why Parallelize?



```
[1]  ✓ 0.0s
```

```
print("Preprocessing Data ...")
df = preprocess(data)

print("Proprocessing Complete.")
```

```
[2]  9m 45.0s
```

```
Preprocessing Data ...
```

Types of Parallel Problems

Communication Intensive

Characterized by tasks that require **frequent data exchange or coordination**.

Use frameworks like MPI and hardware like HPC/Infiniband.

Examples:

- Simulations (weather forecasting, molecular dynamics)
- Large Language Model Training

Embarrassingly Parallel

Characterized by many **independent** tasks

Use frameworks like Mapreduce and pools of instances

Examples:

- Pre-processing data to build a training set for ML modelling
  - cleaning, transforming, aggregating
- Running multiple independent simulations to test the effect of different parameters

Ways we parallelize embarrassingly parallel Python processing

1. Shell Scripting
2. Python Multiprocessing (and Multithreading)
3. PySpark
4. Dask
5. Polars

Shell Scripting

Shell Scripting

Use shell commands to launch multiple Python scripts concurrently, each processing a portion of the data.

Python

```python
import pandas as pd
import sys

def calculate_mean_price(city_csv):
    df = pd.read_csv(city_csv)
    mean_price = df['price'].mean()
    print(f"{city_csv}: {mean_price}")

if __name__ == "__main__":
    if len(sys.argv) != 2:
       print("missing argument")
       sys.exit(1)
    city_csv = sys.argv[1]
    calculate_mean_price(city_csv)
```

Bash

```bash
for city in Calgary Edmonton Vancouver Toronto; do
    python mean_housing_price.py $city.csv &
done
```

Shell Scripting

Use shell commands to launch multiple Python scripts concurrently, each processing a portion of the data.

Advantages

- Easy to implement
- Ready-made to run on a an HPC cluster with schedulers like Slurm or PBS

Disadvantages

- Extra steps to aggregate and further process results

Multiprocessing

Multiprocessing vs Multithreading in Python

Threads	Processes
Lightweight execution units with a single process. They share the same memory space.	Independent execution environments with their own memory space and Python interpreter
Concurrent, but not parallel, execution (blame the GIL)	True parallel execution (on multi-core processors)
Good fit for I/O-bound tasks with lots of waiting (that said, you should look at asyncio for performance)	Good fit for CPU-bound tasks

Multiprocessing vs Multithreading in Python

```
import threading
import time

def task(name):
      print(f"Thread {name}: starting")
      time.sleep(2)
      print(f"Thread {name}: finishing")

if __name__ == "__main__":
      threads = []
      for i in range(3):
            thread = threading.Thread(target=task,
                                      args=(i,))
            threads.append(thread)
            thread.start()

      for thread in threads:
            thread.join()
```

```
from multiprocessing import Pool
import time

def task(name):
      print(f"Process {name}: starting")
      time.sleep(2)
      print(f"Process {name}: finishing")

if __name__ == "__main__":
      with Pool(processes=3) as pool:
            pool.map(task, range(3))
```

Why Multiprocessing over Shell Scripting?

Advantages

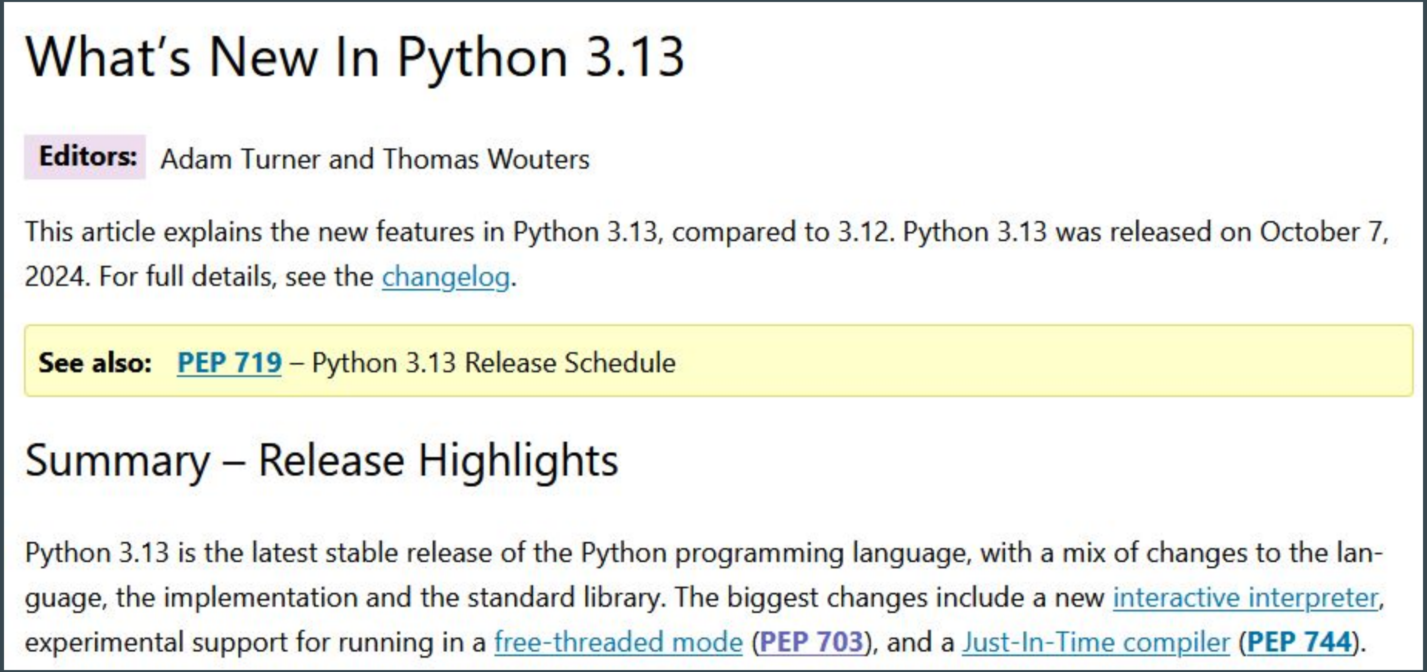
- Integrates seamlessly with the rest of your Python code -> can immediately process results within same python session
- Better error handling and debugging

Disadvantages

- Development overhead for small tasks

Threading and No-GIL

There are ongoing efforts to remove the GIL from CPython. This could significantly improve parallel performance for CPU-bound tasks using threads.



What's New In Python 3.13

Editors: Adam Turner and Thomas Wouters

This article explains the new features in Python 3.13, compared to 3.12. Python 3.13 was released on October 7, 2024. For full details, see the changelog.

See also: **PEP 719** – Python 3.13 Release Schedule

Summary – Release Highlights

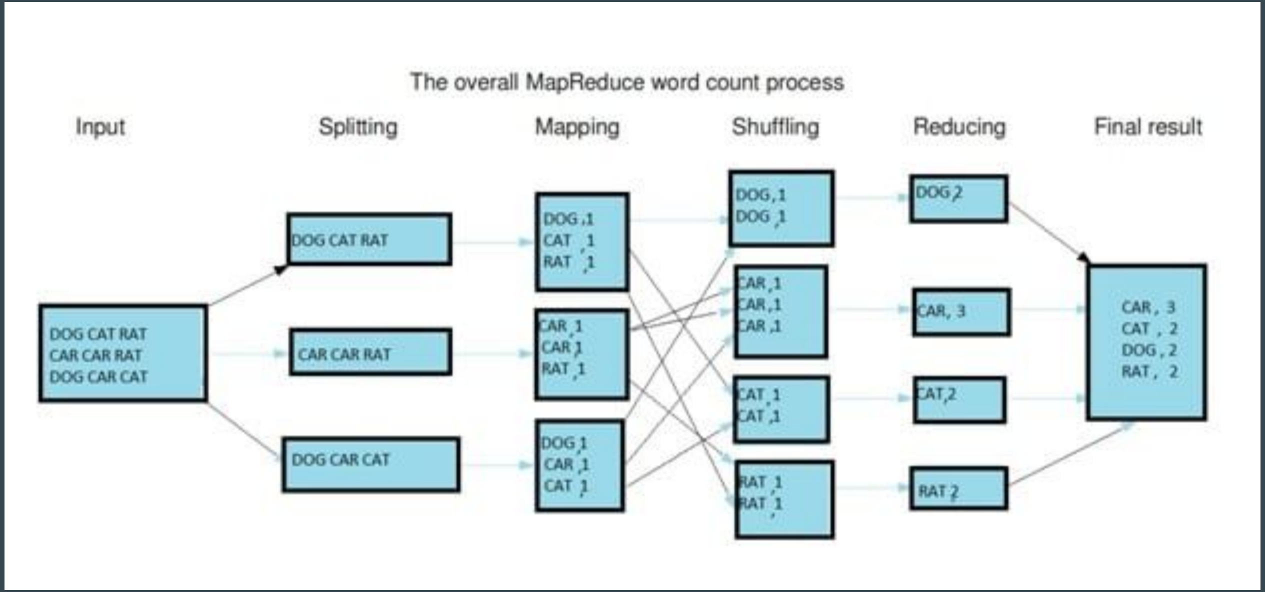
Python 3.13 is the latest stable release of the Python programming language, with a mix of changes to the language, the implementation and the standard library. The biggest changes include a new interactive interpreter, experimental support for running in a free-threaded mode (**PEP 703**), and a Just-In-Time compiler (**PEP 744**).

A note on Jupyter and Multiprocessing

- The `if __name__ == ‘__main__’` Guard ensures code creating new processes is only run when a script is executed (not imported as a module).
  - Jupyter notebooks execute code in a way that confused the multiprocessing libraries, leading to issues if this guard is missing.
- Jupyter can also introduce issues with ‘pickling’ and global variables.
- When using jupyter, try `ipyparallel`

PySpark

MapReduce



https://medium.com/@traininghub.io/hadoop-mapreduce-architecture-7e167e264595

Spark

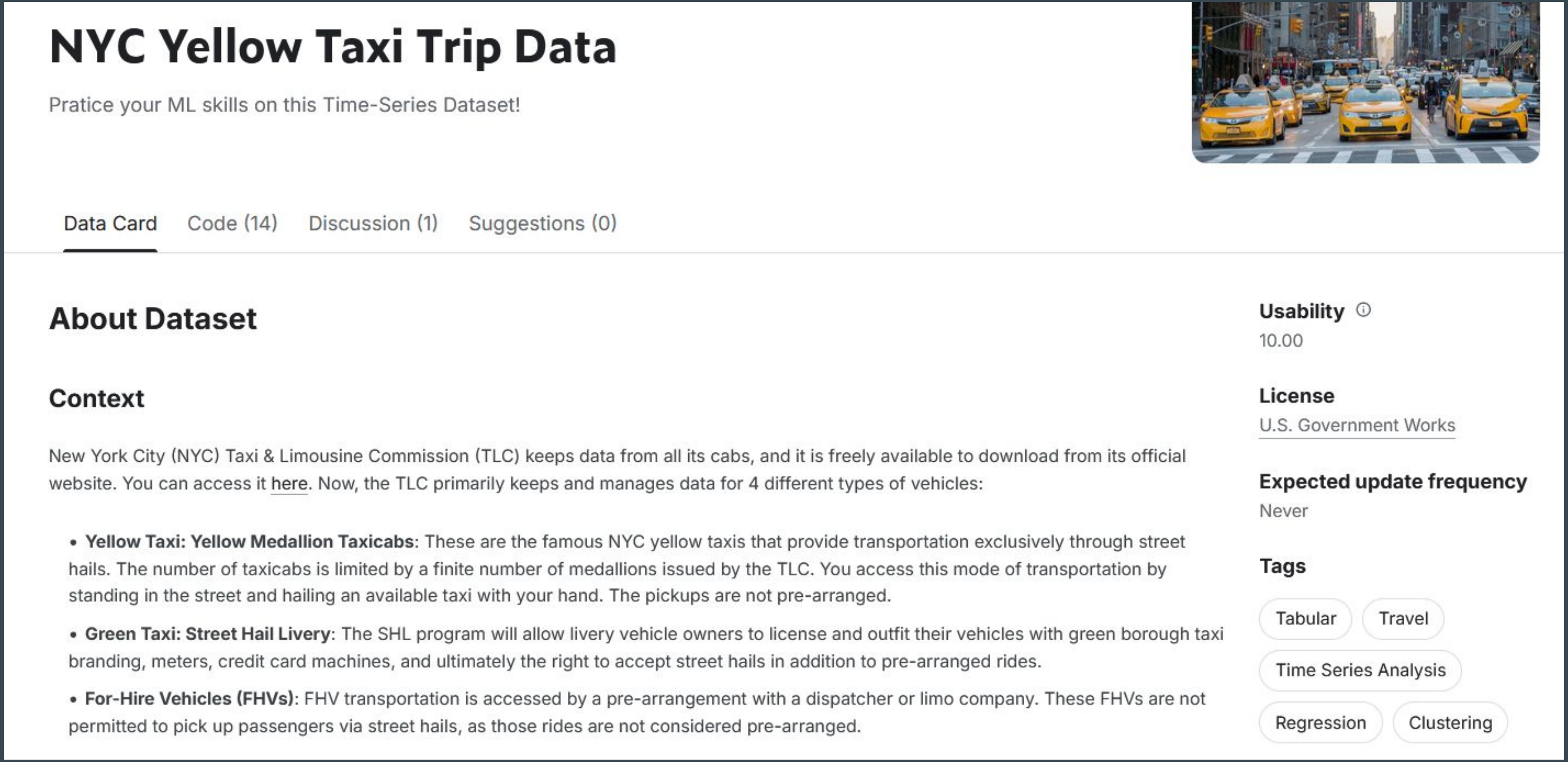
- Developed as a faster and more versatile alternative to Hadoop's MapReduce (but often still runs with Hadoop)
- Supports Batch Processing, Stream Processing, ML and Graph Processing
- Distributes data and computations across a cluster of machines
  - Process really big datasets

PySpark

- PySpark is the Python API for Spark
- Write spark code with Python syntax (using dataframes similar to pandas, but distributed)

Demo 1

Our Sample Problem



NYC Yellow Taxi Trip Data

Pratice your ML skills on this Time-Series Dataset!

Data Card | Code (14) | Discussion (1) | Suggestions (0)

About Dataset

Context

New York City (NYC) Taxi & Limousine Commission (TLC) keeps data from all its cabs, and it is freely available to download from its official website. You can access it here. Now, the TLC primarily keeps and manages data for 4 different types of vehicles:

- **Yellow Taxi: Yellow Medallion Taxicabs**: These are the famous NYC yellow taxis that provide transportation exclusively through street hails. The number of taxicabs is limited by a finite number of medallions issued by the TLC. You access this mode of transportation by standing in the street and hailing an available taxi with your hand. The pickups are not pre-arranged.
- **Green Taxi: Street Hail Livery**: The SHL program will allow livery vehicle owners to license and outfit their vehicles with green borough taxi branding, meters, credit card machines, and ultimately the right to accept street hails in addition to pre-arranged rides.
- **For-Hire Vehicles (FHVs)**: FHV transportation is accessed by a pre-arrangement with a dispatcher or limo company. These FHVs are not permitted to pick up passengers via street hails, as those rides are not considered pre-arranged.

Usability
10.00

License
U.S. Government Works

Expected update frequency
Never

Tags
Tabular, Travel, Time Series Analysis, Regression, Clustering

Dask

Dask

An easy to use, flexible library for parallel computing in Python.

Scales from single machines to clusters.

Provides familiar APIs, mirroring NumPy and Pandas.

Breaks large computations into smaller tasks and builds a task graph.

Dynamically schedules and executes tasks based on dependencies and available resources.

Optimizes data locality and minimizes communication overhead.

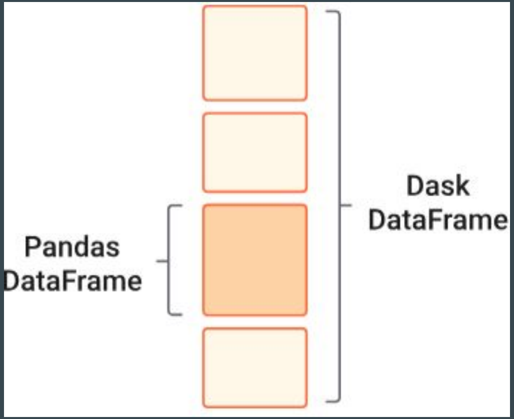
https://docs.dask.org/en/stable/

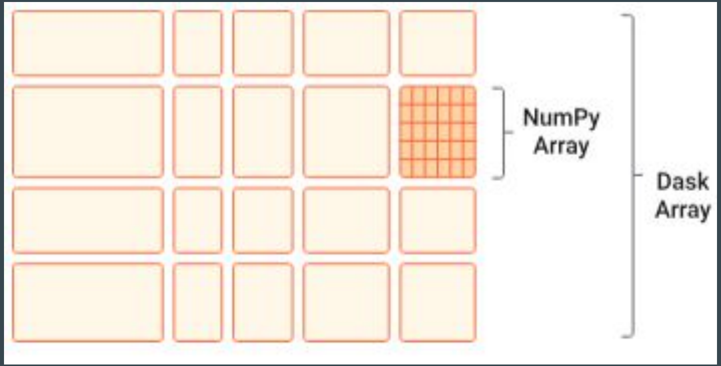
Dask Collections

Dataframes - Parallelize the pandas library

Arrays - Parallelize the NumPy library

- Larger-than-memory execution for single machines
- Parallel execution, Distributed when available





Why Dask?

Advantages

- Easy to use or add to your numpy/pandas workflow
- Flexible to various parallel computing needs
- Scales from 1 machine to many

Disadvantages

- Scheduling/communication overhead, noticeable on smaller tasks

Dask vs Spark

- Generally Dask is smaller and lighter weight than Spark.
  - Fewer features and, instead, is used in conjunction with other libraries

- Many enterprise data lakes are built on top of spark (e.g. Databricks, AWS Elastic MapReduce, GCP Dataproc, Cloudera, etc.) making it a default choice.

- https://www.reddit.com/r/Python/comments/198f5vc/one_billion_row_challenge_dask_vs_spark/

https://docs.dask.org/en/latest/spark.html

Polars

Polars vs Dask

When your data fits in memory -> Try Polars first

Polars also *may* be better for text processing

Demo 2

ML Package Parallelism

TensorFlow - CPU/GPU/TPU Parallelism (but not always easy)

Scikit-learn - Some operations support CPU parallelism

XGBoost - Supports CPU/Multi-machine parallelism

LightGBM - CPU/GPU Parallelism

PyTorch - CPU/GPU/Multi-machine Parallelism

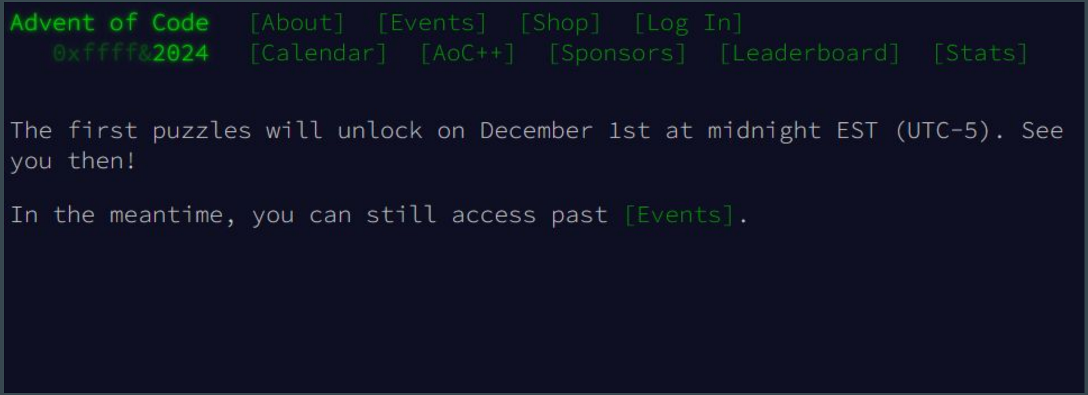
How to choose

- Have you verified you need parallelism
  - No -> KISS
  - Yes: Does your package of choice support it natively?
    - Yes ->Use that package
    - No: Do you have access to a public cloud or a cluster?
      - No -> Dask/Polars
      - Yes:
        - You’re in academia with access to ARC -> Shell Scripting
        - You’re in industry with access to Spark/Databricks -> PySpark
        - Other -> Dask

Thanks!

https://www.linkedin.com/in/nathanbryans

https://github.com/nbryans/parallel_python_examples


Advent of Code
0xffff&2024
[About] [Events] [Shop] [Log In]
[Calendar] [AoC++] [Sponsors] [Leaderboard] [Stats]

The first puzzles will unlock on December 1st at midnight EST (UTC-5). See you then!

In the meantime, you can still access past [Events].