1st OCT 2024

Polars is a high-performance DataFrame library in Python that is designed for speed, parallelism, and low memory usage. It is an alternative to Pandas, and it’s optimized for larger datasets and high-performance computing environments. Polars is written in Rust, which gives it an advantage in terms of speed and performance.

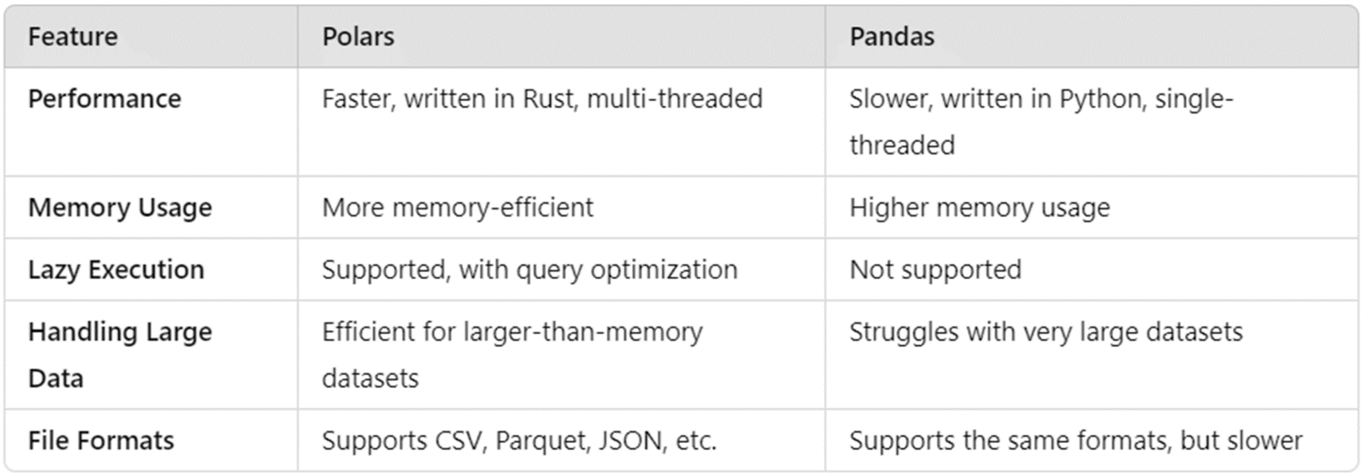
Key Features of Polars:

Fast: Polars uses a columnar memory layout and is written in Rust, making it significantly faster than Pandas for many operations.

Memory-efficient: Polars can handle larger datasets with lower memory usage due to its efficient data storage format.

Lazy execution: Polars supports lazy execution, allowing you to build query pipelines before actually executing the computations, optimizing them for speed.

Multi-threaded: Operations in Polars are parallelized to take advantage of modern CPUs with multiple cores.



Feature	Polars	Pandas
Performance	Faster, written in Rust, multi-threaded	Slower, written in Python, single-threaded
Memory Usage	More memory-efficient	Higher memory usage
Lazy Execution	Supported, with query optimization	Not supported
Handling Large Data	Efficient for larger-than-memory datasets	Struggles with very large datasets
File Formats	Supports CSV, Parquet, JSON, etc.	Supports the same formats, but slower

```
pip install openpyxl
```

```
Requirement already satisfied: openpyxl in /usr/local/lib/python3.10/dist-packages (3.1.5)
Requirement already satisfied: et-xmlfile in /usr/local/lib/python3.10/dist-packages (from openpyxl) (1.1.
```

```
!pip install --upgrade pandas

#conda update pandas

import pandas as pd
print(pd.__version__)

!pip show pandas
```

```
!pip install --upgrade pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.1.4)
Collecting pandas
  Downloading pandas-2.2.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl.metadata (89 kB)
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 89.9/89.9 kB 5.6 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.26
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from panc
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas) (202
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2
Downloading pandas-2.2.3-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (13.1 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.1/13.1 MB 41.3 MB/s eta 0:00:00
Installing collected packages: pandas
  Attempting uninstall: pandas
    Found existing installation: pandas 2.1.4
    Uninstalling pandas-2.1.4:
      Successfully uninstalled pandas-2.1.4
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed.
cudf-cu12 24.4.1 requires pandas<2.2.2dev0,>=2.0, but you have pandas 2.2.3 which is incompatible.
google-colab 1.0.0 requires pandas==2.1.4, but you have pandas 2.2.3 which is incompatible.
Successfully installed pandas-2.2.3
```

```
!pip show pandas
```

```
Name: pandas
Version: 2.2.3
Summary: Powerful data structures for data analysis, time series, and statistics
Home-page: https://pandas.pydata.org
Author:
Author-email: The Pandas Development Team <pandas-dev@python.org>
License: BSD 3-Clause License

        Copyright (c) 2008-2011, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData Developmer
        All rights reserved.

        Copyright (c) 2011-2023, Open source contributors.

        Redistribution and use in source and binary forms, with or without
        modification, are permitted provided that the following conditions are met:

        * Redistributions of source code must retain the above copyright notice, this
          list of conditions and the following disclaimer.

        * Redistributions in binary form must reproduce the above copyright notice,
          this list of conditions and the following disclaimer in the documentation
          and/or other materials provided with the distribution.

        * Neither the name of the copyright holder nor the names of its
          contributors may be used to endorse or promote products derived from
          this software without specific prior written permission.

        THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
        AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
        IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
        DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
        FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
        DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
        SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
        CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
        OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
        OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
        Copyright (c) 2010-2019 Keith Goodman
        Copyright (c) 2019 Bottleneck Developers
        All rights reserved.

        Redistribution and use in source and binary forms, with or without
        modification, are permitted provided that the following conditions are met:

            * Redistributions of source code must retain the above copyright notice,
import pandas as pd

# https://github.com/manishjainstorage/Advanced-Python/blob/main/StockMarketData-13krows.xlsx
# Define the correct raw URL of the .xlsx file
url = "https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krows.xlsx"

# Read the .xlsx file into a DataFrame, specifying the engine
df = pd.read_excel(url, engine='openpyxl')

# Display the first few rows of the DataFrame
df.head()
```

	date	open	high	low	close	volume	Name
0	2013-02-08	15.07	15.12	14.63	14.75	8407500	AAL
1	2013-02-11	14.89	15.01	14.26	14.46	8882000	AAL
2	2013-02-12	14.45	14.51	14.10	14.27	8126000	AAL
3	2013-02-13	14.30	14.94	14.25	14.66	10259500	AAL
4	2013-02-14	14.94	14.96	13.16	13.99	31879900	AAL

```
import pandas as pd
import polars as pl
import time

# Load large dataset using Pandas
start_time = time.time()
df_pandas = pd.read_excel("https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krow
print(f'Pandas load time: {time.time() - start_time} seconds')

# Load large dataset using Polars
start_time = time.time()
df_polars = pl.read_excel("https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krow
print(f'Polars load time: {time.time() - start_time} seconds')
```

```
Pandas load time: 2.8822035789489746 seconds
Polars load time: 0.482496976852417 seconds
```

```
!pip show polars
```

```
Name: polars
Version: 1.8.2
Summary: Blazingly fast DataFrame library
Home-page: https://www.pola.rs/
```

```
Author:
Author-email: Ritchie Vink <ritchie46@gmail.com>
License:
Location: /usr/local/lib/python3.10/dist-packages
Requires:
Required-by:
```

1. Lazy Evaluation

Polars: Polars supports lazy evaluation, which allows the computation graph to be optimized before execution. This can lead to significant performance improvements when dealing with large datasets.

Pandas: Does not support lazy evaluation. All operations in pandas are executed immediately

```
# A generator function for LE

def Lazy_no(n):
  print("start to generate nos")
  for i in range(n):
    print(f"producing {i}")
    yield i #  it will pause the function and send back to value

# create the generator ( but it doesn't start again yet)
my_generator = Lazy_no(5)

print("Gen is created however has't produce any no yet")

#generate a no one by one

for number in my_generator:
  print(f"Got number {number}")
```

```
Gen is created however has't produce any no yet
start to generate nos
producing 0
Got number 0
producing 1
Got number 1
producing 2
Got number 2
producing 3
Got number 3
producing 4
Got number 4
```

```
def a(n):
  i = 1
  while i<=n:
    yield i
    i+=1

gen = a(5)
gen = a(6)

print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
1
2
3
4
5
6
```

```python
import polars as pl

# Create a lazy frame
df = pl.scan_csv('https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krows_csv.csv

# Define transformations
df_lazy = df.filter(pl.col('open') > 15).select(pl.col('close') * 2)

# Check the logical plan (computation graph)
print(df_lazy.explain())

# Execute the computation graph
df_eager = df_lazy.collect()
df_eager
```

```
SELECT [[(col("close")) * (2.0)]] FROM
  Csv SCAN [https://github.com/manishjainstorage/Advanced-Python/raw/main/StockMarketData-13krows_csv.csv]
  PROJECT 2/7 COLUMNS
  SELECTION: [(col("open")) > (15.0)]
shape: (13_862, 1)
close
f64
29.5
31.0
31.82
32.5
31.96
…
82.06
82.784
80.76
81.2
81.84
```

2. Efficient Parallel and Multi-threaded Execution

**Polars: *Polars can automatically parallelize operations across multiple threads, making it more efficient for CPU-bound tasks and large datasets.

*Pandas: *Pandas operations are single-threaded, which can lead to slower performance for large-scale computations.

```
import pandas as pd
import polars as pl
import numpy as np
import time

# Define the size of the dataset
N = 10**7  # 10 million rows

# Generate a large dataset
data = {
    'a': np.random.rand(N),
    'b': np.random.rand(N)
}

# Function to perform a computation in Pandas
def pandas_computation(data):
    df = pd.DataFrame(data)
    # Perform a computation (e.g., summing the product of two columns)
    result = (df['a'] * df['b']).sum()
    return result

# Function to perform a computation in Polars
def polars_computation(data):
    df = pl.DataFrame(data)
    # Perform a computation (e.g., summing the product of two columns)
    result = (df['a'] * df['b']).sum()
    return result

# Measure execution time for Pandas
start_time = time.time()
pandas_result = pandas_computation(data)
pandas_time = time.time() - start_time
print(f"Pandas result: {pandas_result}, Time taken: {pandas_time:.2f} seconds")

# Measure execution time for Polars
start_time = time.time()
polars_result = polars_computation(data)
polars_time = time.time() - start_time
print(f"Polars result: {polars_result}, Time taken: {polars_time:.2f} seconds")
```

```
Pandas result: 2499938.0149599286, Time taken: 0.34 seconds
Polars result: 2499938.0149599267, Time taken: 0.14 seconds
```

Explanation

Data Generation:

We generate a large dataset with 10 million rows, containing two columns (a and b) filled with random numbers.

Pandas Computation:

The function pandas_computation(data) creates a Pandas DataFrame and computes the sum of the product of columns a and b. This operation runs in a single-threaded manner.

Polars Computation:

The function polars_computation(data) creates a Polars DataFrame and performs the same computation. Polars automatically parallelizes this operation, leveraging multiple threads for improved performance.

Execution Time Measurement:

We measure the execution time for both the Pandas and Polars computations using the time module.

Output:

The code prints the result of the computation along with the time taken for each library.

Expected Results

When you run this code, you should observe that:

Polars generally takes significantly less time compared to Pandas for large datasets due to its ability to utilize multiple threads for parallel execution.

The exact time may vary based on your machine's CPU capabilities and the current load, but Polars should consistently outperform Pandas in this example.

This demonstrates the efficiency of Polars in handling large-scale computations through parallel and multi-threaded execution compared to the single-threaded execution of Pandas.

3. Expressions API

Polars: Polars provides an advanced Expressions API that allows for more flexible and powerful transformations without intermediate DataFrame creation. It can chain multiple transformations efficiently.

Pandas: Operations in pandas are less optimized for chaining and often involve creating multiple intermediate DataFrames, which can lead to performance bottlenecks

```
df = pl.DataFrame({
    "a": [1, 2, 3],
    "b": [4, 5, 6],
    "c": [7, 8, 9]
})

# Apply a chain of expressions
df = df.with_columns([
    (pl.col("a") * pl.col("b")).alias("a_times_b"),
    (pl.col("c") + 5).alias("c_plus_5")
])

df
```

```
shape: (3, 5)
  a    b    c   a_times_b  c_plus_5
i64  i64  i64         i64       i64
  1    4    7           4        12
  2    5    8          10        13
  3    6    9          18        14
```

4. Dynamic Window Functions

Polars: Polars has dynamic window functions, which allow for powerful time-based aggregations and rolling window computations.

Pandas: Although pandas supports rolling windows, Polars offers more flexibility and performance, especially with dynamic windows.

```
df = pl.DataFrame({
    "time": [1, 2, 3, 4, 5],
    "value": [10, 20, 30, 40, 50]
})

# Dynamic window function with a window size of 2
df = df.with_columns([
    pl.col("value").rolling_sum(window_size=3).alias("rolling_sum")
])

df
```

```
shape: (5, 3)
 time  value  rolling_sum
  i64    i64          i64
    1     10         null
    2     20         null
    3     30           60
    4     40           90
    5     50          120
```

High-Performance Computing in Python

High-Performance Computing (HPC) in Python allows for leveraging the power of parallelism and distributed computing to solve complex and large-scale computational problems efficiently. Below are key concepts, tools, and best practices to implement HPC in Python:

Key Concepts

Parallel Computing: Dividing a task into smaller sub-tasks that can be executed simultaneously across multiple processors or machines.

Distributed Computing: Involves multiple computers working together to solve a problem, often over a network.

Concurrency: Managing multiple computations at the same time, which may involve parallel execution but does not necessarily mean they execute simultaneously.

Vectorization: Using libraries that leverage low-level optimizations to perform operations on entire arrays or data structures at once, which is often more efficient than looping.

Vectorization techniques and **Parallel processing with multiprocessing and concurrent.futures**

```
""" NumPy Array Operations

NumPy is the fundamental package for numerical computing in Python.
It provides powerful array objects and functions that support vectorization """
```

```
import numpy as np

# Create two large NumPy arrays
a = np.random.rand(1000000)
b = np.random.rand(1000000)

# Element-wise addition (vectorized operation)
c = a + b  # This operation is executed in C, making it faster than a Python loop
print(c)
```

```
[1.0881665  1.38354092 0.69107068 ... 0.72146779 1.61068822 1.43944195]
```

```
"""Broadcasting

Broadcasting allows NumPy to perform arithmetic operations on arrays of
different shapes without the need for explicit loops."""

import numpy as np

# Create a 1D array and a 2D array
a = np.array([1, 2, 3,4])
b = np.array([[10], [20], [30]])

# Broadcasting the 1D array to match the shape of the 2D array
result = a + b
print(result)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

```
import numpy as np

# Create a 1D array and a 2D array
a = np.array([[1, 2, 3],[4,5,6]])
b = np.array([10,20,20])

# Broadcasting the 1D array to match the shape of the 2D array
result = a + b
print(result)
```

```
[[11 22 23]
 [14 25 26]]
```

```
"""Using Numba

Numba is a just-in-time (JIT) compiler that translates a subset of Python and NumPy
code into fast machine code.

It can significantly speed up array operations."""

import numpy as np
from numba import jit

@jit(nopython=True)
def add_arrays(a, b):
    return a + b
```

```
# Create large arrays
a = np.random.rand(1000000)
b = np.random.rand(1000000)

# Call the JIT-compiled function
c = add_arrays(a, b)
print(c)
```

```
[1.81199835 0.39919907 1.25690862 ... 0.34071594 1.17085443 1.05272053]
```

```
"""SciPy for Scientific Computing

SciPy builds on NumPy and provides additional functionality for optimization,
 integration, interpolation,
and other scientific computations, often with vectorized implementations."""

from scipy import integrate
import numpy as np

# Define a function
def f(x):
    return x**2

# Vectorized integration
result, error = integrate.quad(f, 0, 1)  # Integrates f(x) from 0 to 1
print(result, error)
```

```
0.33333333333333337 3.700743415417189e-15
```

```
"""Dask for Out-of-Core Computing

Dask is a parallel computing library that integrates with NumPy and pandas to
handle larger-than-memory datasets efficiently,
 with support for lazy evaluations and parallelism."""

import dask.array as da

# Create a large Dask array
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Perform a vectorized operation
y = x + x.T  # Transpose and add
result = y.compute()  # Compute the result
print(result)
```

```
[[0.40631271 1.59814771 0.86065558 ... 0.81201027 1.09129456 1.29148079]
 [1.59814771 0.14809533 0.77080944 ... 1.58567307 0.90198611 1.13361025]
 [0.86065558 0.77080944 0.65898193 ... 0.89693504 1.07957005 1.92099078]
 ...
 [0.81201027 1.58567307 0.89693504 ... 0.74616648 1.16125423 0.77108196]
 [1.09129456 0.90198611 1.07957005 ... 1.16125423 1.86125274 0.59751993]
 [1.29148079 1.13361025 1.92099078 ... 0.77108196 0.59751993 0.47497537]]
```

```
import dask.dataframe as da

# Create a large Dask array
x = da.read_csv("/content/StockMarketData-13krows_csv.csv")

a = x['open'].mean()
```

```
result = a.compute()
print(result)
```

```
76.94264072900158
```

NumPy optimization strategies

Optimizing performance with NumPy is essential for efficient numerical computing in Python, especially when working with large datasets or complex calculations. Here are several strategies and best practices for optimizing NumPy code:

```
""" Utilize Vectorization

Description: Replace explicit loops with NumPy's built-in array operations,
which are optimized and run in compiled C code"""

import numpy as np

# Instead of this:
a = np.random.rand(1000000)
b = np.random.rand(1000000)
c = np.empty_like(a)
for i in range(len(a)):
    c[i] = a[i] + b[i]

# Use vectorized operations:
c = a + b  # Faster and more concise
```

```
"""Use Broadcasting

Description: Broadcasting allows NumPy to perform operations on arrays of different
 shapes without creating large temporary arrays."""

import numpy as np

a = np.array([1, 2, 3])
b = np.array([[10], [20], [30]])

# Broadcasting
result = a + b  # Automatically expands a to match the shape of b
```

```
"""Pre-allocate Arrays

Description: When creating arrays, pre-allocate them with the desired shape
and size to avoid resizing during operations. """

# Instead of dynamically growing an array:
result = []
for i in range(1000):
    result.append(i**2)
result = np.array(result)  # Conversion to NumPy array afterward
```

```
# Pre-allocate:
result = np.empty(1000)
for i in range(1000):
    result[i] = i**2



"""Avoid Unnecessary Copies

Description: Be mindful of operations that create copies of arrays
(e.g., slicing or reshaping). Use views when possible."""

# Slicing creates a view, while some operations create copies
a = np.array([1, 2, 3, 4, 5])
b = a[1:3]  # b is a view, changes in a reflect in b

# Use np.copy() if you need an independent copy
c = np.copy(a[1:3])  # Creates a new array



""" Use In-place Operations

Description: Modify existing arrays in place to save memory and improve performance."""

a = np.array([1, 2, 3])
a += 1  # In-place addition, modifies a directly




"""Choose the Right Data Type

Description: Use the most efficient data types for your data to save memory and improve performance.
For instance, use float32 instead of float64 when high precision is not needed."""

a = np.array([1, 2, 3], dtype=np.float32)  # Using float32 instead of float64




"""Leverage NumPy Functions

Description: Use NumPy's built-in functions for mathematical operations,
which are optimized for performance."""

a = np.random.rand(1000000)
mean_value = np.mean(a)  # Use NumPy's mean function



"""Profile Your Code

Description: Use profiling tools like cProfile or line_profiler to identify bottlenecks
 in your code and optimize those specific areas."""

import cProfile

def compute():
    a = np.random.rand(1000000)
    b = np.random.rand(1000000)
    return a + b
```

```
cProfile.run('compute()')

# percall =  tottime/ncalls
```

```
         6 function calls in 0.057 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.015    0.015    0.055    0.055 <ipython-input-37-a0af4d21ffa1>:8(compute)
        1    0.001    0.001    0.057    0.057 <string>:1(<module>)
        1    0.000    0.000    0.057    0.057 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    0.040    0.020    0.040    0.020 {method 'rand' of 'numpy.random.mtrand.RandomState' objects}
```

```
""" Avoid Global Variables

Description: Using global variables can slow down performance in NumPy.
Prefer passing arrays as function arguments."""

# Slow with global variable
global_array = np.array([1, 2, 3])

def process():
    return global_array * 2  # Slower access

# Better approach
def process(arr):
    return arr * 2

result = process(np.array([1, 2, 3]))
```

```
#!pip install pandas-profiling

import pandas as pd
from pandas_profiling import ProfileReport

df = pd.read_csv("/content/StockMarketData-13krows_csv.csv")

profile = ProfileReport(df, title="Pandas Profiling Report")
profile.to_file("output.html")
```

```
---------------------------------------------------------------------------
PydanticImportError                       Traceback (most recent call last)
<ipython-input-41-31a68a23779c> in <cell line: 4>()
      2 
      3 import pandas as pd
----> 4 from pandas_profiling import ProfileReport
      5 
      6 df = pd.read_csv("/content/StockMarketData-13krows_csv.csv")
```

4 frames

```
/usr/local/lib/python3.10/dist-packages/pydantic/_migration.py in wrapper(name)
    294             return import_string(REDIRECT_TO_V1[import_path])
    295         if import_path == 'pydantic:BaseSettings':
--> 296             raise PydanticImportError(
    297                 '`BaseSettings` has been moved to the `pydantic-settings` package. '
    298                 f'See https://docs.pydantic.dev/{version_short()}/migration/#basesettings-has-
moved-to-pydantic-settings '

PydanticImportError: `BaseSettings` has been moved to the `pydantic-settings` package. See
https://docs.pydantic.dev/2.9/migration/#basesettings-has-moved-to-pydantic-settings for more details.

For further information visit https://errors.pydantic.dev/2.9/u/import-error

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES

Profiling and Optimization

Profiling and optimization are crucial steps in improving the performance of your Python code, particularly when dealing with numerical computations or data processing tasks. Below is a guide on how to effectively profile your Python code to identify bottlenecks and subsequently optimize those areas for better performance.

```
"""1. Profiling Techniques

a. Using cProfile

cProfile is a built-in Python module for profiling your code.
 It provides a way to see where the time is being spent in your program."""

import cProfile
import pstats

def compute():
    # Simulate a heavy computation
    total = 0
    for i in range(1000000):
        total += i ** 2
    return total

# Profile the compute function
cProfile.run('compute()', 'output.stats')
```

```
# Load stats and print in a readable format
with open('output.txt', 'w') as f:
    p = pstats.Stats('output.stats', stream=f)
    p.sort_stats('cumulative').print_stats()



"""
b. Using Line Profiler

line_profiler is a third-party module that provides line-by-line profiling of functions.

"""

# Step 1: Install Line Profiler
!pip install line_profiler

# Step 2: Load the Line Profiler Extension
%load_ext line_profiler

# Step 3: Define the function to profile
def my_function():
    total = 0
    for i in range(10000):
        total += i ** 2
    return total

# Run the function first (optional)
my_function()

# Step 4: Use lprun to profile the function
%lprun -f my_function my_function()
```

```
Requirement already satisfied: line_profiler in /usr/local/lib/python3.10/dist-packages (4.1.3)
The line_profiler extension is already loaded. To reload it, use:
  %reload_ext line_profiler
```

```
# Step 1: Install required libraries
!pip install line_profiler

# Step 2: Load the Line Profiler Extension
%load_ext line_profiler

# Step 3: Define the function to profile
def my_function():
    total = 0
    for i in range(10000):
        total += i ** 2
    return total

# Step 4: Create a Line Profiler instance
from line_profiler import LineProfiler

# Create a profiler
profiler = LineProfiler()
```

```
profiler.add_function(my_function)

# Run the function under the profiler
profiler.run('my_function()')

# Step 5: Get profiling results as a string
profiling_results = io.StringIO()
profiler.print_stats(stream=profiling_results)
profiling_results = profiling_results.getvalue()

# Step 6: Save the profiling results to a text file
txt_file_path = "/content/profiling_results.txt"
with open(txt_file_path, "w") as txt_file:
    txt_file.write(profiling_results)

# Step 7: Download the text file
from google.colab import files
files.download(txt_file_path)
```

```
"""
c. Memory Profiling

To profile memory usage, you can use memory_profiler. This is useful to identify memory bottlenecks.

"""
```

```
# Step 1: Install the memory_profiler package
!pip install memory_profiler

# Step 2: Load the memory_profiler extension
%load_ext memory_profiler

# Step 3: Define a function to profile
def my_function():
    """Function that consumes memory by creating a large list."""
    large_list = [i for i in range(10**6)]  # Create a large list
    return sum(large_list)

# Step 4: Use memory_profiler's memory usage monitoring manually
from memory_profiler import memory_usage

# Function to monitor memory usage while executing my_function
def profile_memory():
    mem_usage = memory_usage((my_function,), interval=0.1)  # Capture memory usage
    return mem_usage

# Step 5: Call the profiling function and print memory usage
mem_usage = profile_memory()
print(f"Memory usage (in MiB): {mem_usage}")

print(len(mem_usage))
```

```
Requirement already satisfied: memory_profiler in /usr/local/lib/python3.10/dist-packages (0.61.0)
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages (from memory_profiler) (5.
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
Memory usage (in MiB): [545.16015625, 545.16015625, 545.3984375, 548.28515625, 550.96875, 553.65234375, 557
15
```

```
"""
2. Analyzing Profiling Results

After profiling, you'll get insights into:

Function Call Counts: How many times each function was called.
Cumulative Time: Total time spent in a function and all its sub-functions.
Per-Call Time: Average time spent in each call.
By analyzing these metrics, you can identify functions that take the most time or are
called frequently and may need optimization.


"""

import cProfile
import pstats
import io
import numpy as np

# Step 1: Define a function to profile
def my_function():
    """Function that performs some computations."""
    total = 0
    for i in range(1, 10000):
        total += np.sqrt(i)  # Some heavy computation
    return total

# Step 2: Profile the function using cProfile
def profile_my_function():
    pr = cProfile.Profile()
    pr.enable()  # Start profiling

    my_function()  # Call the function to be profiled

    pr.disable()  # Stop profiling

    # Step 3: Create a stream to capture the output
    s = io.StringIO()
    sortby = pstats.SortKey.CUMULATIVE
    ps = pstats.Stats(pr, stream=s).sort_stats(sortby)

    # Step 4: Print profiling results to the stream
    ps.print_stats()

    # Step 5: Get the profiling results as a string
    profiling_results = s.getvalue()

    # Return profiling results for further analysis
    return profiling_results

# Step 6: Analyze the profiling results
profiling_results = profile_my_function()
```

```
# Print the profiling results
print("Profiling Results:")
print(profiling_results)

# Optionally, you can save the results to a text file
with open("profiling_results.txt", "w") as file:
    file.write(profiling_results)



""" 3. Code Optimization techniques

Once you identify bottlenecks, consider the following optimization strategies:

a. Optimize Algorithms

Choose Efficient Algorithms: Evaluate whether a more efficient algorithm can be used.

Algorithm Complexity: Analyze the time complexity of your algorithms and look for improvements.
b. Utilize Vectorization

Replace explicit loops with vectorized operations using libraries like NumPy."""

import numpy as np
import time

# Original algorithm: O(n^2) complexity
def sum_of_squares_nested(n):
    total = 0
    for i in range(n):
        for j in range(n):
            total += i**2
    return total

# Optimized algorithm: O(n) complexity
def sum_of_squares_optimized(n):
    return n * (n - 1) * (2 * n - 1) // 6  # Formula for sum of squares

# Vectorized operation using NumPy
def sum_of_squares_vectorized(n):
    arr = np.arange(n)
    return np.sum(arr**2)

# Testing the algorithms
n = 10000

# Measure time for original algorithm
start_time = time.time()
result_nested = sum_of_squares_nested(n)
end_time = time.time()
print(f"Nested Sum of Squares Result: {result_nested}, Time: {end_time - start_time:.5f} seconds")

# Measure time for optimized algorithm
start_time = time.time()
result_optimized = sum_of_squares_optimized(n)
end_time = time.time()
print(f"Optimized Sum of Squares Result: {result_optimized}, Time: {end_time - start_time:.5f} seconds")

# Measure time for vectorized algorithm
start_time = time.time()
result_vectorized = sum_of_squares_vectorized(n)
```

```
end_time = time.time()
print(f"Vectorized Sum of Squares Result: {result_vectorized}, Time: {end_time - start_time:.5f} seconds")
```

```
Nested Sum of Squares Result: 3332833350000000, Time: 37.58366 seconds
Optimized Sum of Squares Result: 333283335000, Time: 0.00006 seconds
Vectorized Sum of Squares Result: 333283335000, Time: 0.00346 seconds
```

3rd OCT 2024

Caching and Memoization

Caching and memoization are optimization techniques used to enhance performance by storing the results of expensive function calls and reusing them when the same inputs occur again.

Caching refers to storing the results of expensive computations, typically in a way that is easily accessible. This is useful when the same computation might be needed multiple times. In Python, caching can be implemented using decorators like functools.lru_cache.

Key Differences

Scope:

Caching is generally used for any function that may need to store results, regardless of whether it’s recursive.

Memoization specifically targets recursive functions to avoid redundant calculations.

Implementation:

Caching is often implemented using decorators like lru_cache in Python.

Memoization typically involves a custom wrapper function to handle the cache.

Use Cases

Caching:

Web applications that fetch data from a database or API frequently. Image processing applications that perform repeated transformations.

Memoization:

Recursive algorithms like Fibonacci series, factorial calculations, or other dynamic programming problems where overlapping subproblems occur

```
import time
from functools import lru_cache

@lru_cache(maxsize=None)  # Cache results without a size limit
def expensive_function(n):
    """Simulates an expensive computation."""
    time.sleep(2)  # Simulate a delay
    return n * n

# Using the cached function
start_time = time.time()
print(expensive_function(4))  # First call, computes the value
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(4))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(4))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(5))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(5))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")
```

```
16
Time taken: 2.0024163722991943 seconds
16
Time taken: 0.00011372566223144531 seconds
16
Time taken: 0.00011348724365234375 seconds
25
Time taken: 2.002372980117798 seconds
25
Time taken: 0.0014226436614990234 seconds
```

```
import time
from functools import lru_cache

# Simulated API or database call
@lru_cache(maxsize=None)  # Caches up to 5 recent calls
def fetch_data_from_db(query):
    print(f"Fetching data for query: {query}")
    time.sleep(2)  # Simulate a time-consuming database query
    return f"Results for {query}"

# Testing the caching
if __name__ == "__main__":
    print(fetch_data_from_db("SELECT * FROM users WHERE id = 1"))
    print(fetch_data_from_db("SELECT * FROM users WHERE id = 2"))
    # This call will be cached, so no delay
    print(fetch_data_from_db("SELECT * FROM users WHERE id = 1"))
```

```
Fetching data for query: SELECT * FROM users WHERE id = 1
Results for SELECT * FROM users WHERE id = 1
Fetching data for query: SELECT * FROM users WHERE id = 2
Results for SELECT * FROM users WHERE id = 2
Results for SELECT * FROM users WHERE id = 1
```

Memoization

*Memoization is a specific form of caching that is used primarily to optimize recursive functions by storing the results of expensive function calls and returning the cached result when the same inputs occur again. This is especially useful in algorithms like Fibonacci series computation *

```
def memoize(func):
    cache = {}

    def wrapper(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]

    return wrapper

@memoize
def fibonacci(n):
    """Calculates the nth Fibonacci number."""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

# Using the memoized Fibonacci function
print(fibonacci(10))  # Computes the value
print(fibonacci(10))  # Retrieves from cache
```

```
55
55
```

joblib is a library in Python that provides tools for lightweight pipelining in Python and is particularly useful for parallel computing and caching. One of its key features is the ability to easily cache results of function calls using the Memory class. This is especially useful for functions that perform expensive computations or data processing, as it allows you to save and reuse the results without recalculating them.

Benefits of Using Joblib for Caching

Persistence: Cached results are stored on disk, which means you can reuse them across different sessions of your script.

Parallel Computing: joblib also supports parallel processing, allowing you to distribute computations across multiple CPU cores efficiently.

Ease of Use: Adding caching to functions is straightforward with decorators.

```
"""
Below is an example demonstrating how to cache results using joblib.Memory.
This will save the results of an expensive function call to disk,
 so it can be reused in subsequent calls with the same arguments.
"""

from joblib import Memory
import time

# Create a Memory object to store cache results
memory = Memory('./cachedir', verbose=0)
```

```
@memory.cache
def expensive_function(n):
    """Simulates an expensive computation."""
    time.sleep(2)  # Simulate a delay
    return n * n

# Using the cached function
start_time = time.time()
print(expensive_function(4))  # First call, computes the value
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(4))  # Second call, retrieves from cache
print(f"Time taken: {time.time() - start_time} seconds")

start_time = time.time()
print(expensive_function(5))  # Computes a different value
print(f"Time taken: {time.time() - start_time} seconds")
```

```
16
Time taken: 2.0036070346832275 seconds
16
Time taken: 0.002166748046875 seconds
25
Time taken: 2.0033326148986816 seconds
```

Distributed Computing

Distributed computing is a field of computer science that involves multiple computers working together to solve complex problems or process large datasets. In a distributed computing system, tasks are distributed across multiple machines, which collaborate to achieve a common goal. This approach can significantly enhance performance, fault tolerance, and scalability.

Key Concepts of Distributed Computing

Distributed Systems:

A collection of independent computers that appears to its users as a single coherent system. These computers can communicate and coordinate their actions through a network.

Parallel Processing:

This involves dividing a task into smaller sub-tasks that can be processed simultaneously across multiple processors or nodes, reducing the overall computation time.

Scalability:

Distributed systems can be scaled by adding more machines to the network, allowing them to handle increased workloads without a complete redesign of the system.

Fault Tolerance:

Distributed systems can continue to operate even if one or more nodes fail. This is achieved through redundancy and replication of data and services.

Load Balancing:

Distributing workloads evenly across nodes to prevent any single node from becoming a bottleneck.

Communication:

Nodes in a distributed system communicate with each other using various protocols (e.g., HTTP, TCP/IP) and mechanisms (e.g., message queues, RPC).

Common Distributed Computing Frameworks

Apache Hadoop:

A framework for processing and storing large datasets across clusters of computers using the MapReduce programming model.

Apache Spark:

A unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning, and graph processing.

Dask:

A flexible library for parallel computing in Python that scales from a single machine to large clusters. It integrates well with NumPy, Pandas, and other Python libraries.

Ray:

A distributed execution framework that makes it easy to build and scale applications in Python. It's particularly useful for machine learning and reinforcement learning tasks.

TensorFlow and PyTorch:

Both frameworks support distributed training of machine learning models, allowing computations to be performed across multiple GPUs or nodes.

```
!pip install dask[complete]
```

```
import dask.array as da
import dask

# Create a large Dask array
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Perform a computation (e.g., mean)
mean_result = x.mean()

# Trigger the computation
result = mean_result.compute()

print(f"The mean of the array is: {result}")

"""
Dask Array:

A Dask array is created, simulating a large 10,000 x 10,000 array with random values.
The array is divided into smaller chunks (1,000 x 1,000) for distributed processing.

Computation:
```

```
The mean of the Dask array is computed. The actual computation is not performed until the
compute() method is called, allowing Dask to optimize the execution plan and run the calculation
 in parallel.
"""
```

```
The mean of the array is: 0.5000173645829981
'\nDask Array:\n\nA Dask array is created, simulating a large 10,000 x 10,000 array with random values.\nT
he array is divided into smaller chunks (1,000 x 1,000) for distributed processing.\n\nComputation:\n\nThe
mean of the Dask array is computed. The actual computation is not performed until the\ncompute() method is
called, allowing Dask to optimize the execution plan and run the calculation\n in parallel.\n'
```

Scaling computations across multiple cores or machines

Scaling computations across multiple cores or machines is essential for maximizing the performance of computationally intensive tasks, especially in fields like data analysis, machine learning, and scientific computing. Here's an overview of how to achieve this scaling using various techniques and frameworks.

```
"""

1. Parallel Computing

Parallel computing involves breaking a task into smaller subtasks that can be executed simultaneously
on multiple processors or cores. This can be done either on a single machine with multiple cores or
across multiple machines in a cluster.

a. Multiprocessing in Python

The multiprocessing module in Python allows you to create processes that can run concurrently,
 enabling parallel execution of tasks. Here's a basic example:
"""

import multiprocessing

def square(n):
    return n * n

if __name__ == "__main__":
    # Define a list of numbers
    numbers = [1, 2, 3, 4, 5]

    # Create a pool of worker processes
    with multiprocessing.Pool(processes=4) as pool:
        # Map the function to the list of numbers
        results = pool.map(square, numbers)

    print(results)

"""
Process Pool: A pool of worker processes is created, allowing multiple tasks to run concurrently.

Mapping: The map function applies the square function to each element in the numbers list,
distributing the tasks across the worker processes.
"""
```

```
[1, 4, 9, 16, 25]
```

```
"""
2. Distributed Computing Frameworks

For larger-scale computations, especially when working with big data,
distributed computing frameworks like Apache Spark, Dask, and Ray can be utilized.

a. Dask
Dask is a flexible parallel computing library for analytics in Python that integrates
well with NumPy and Pandas.

"""

import dask.array as da

# Create a large Dask array
x = da.random.random((10000, 10000), chunks=(1000, 1000))

# Perform a computation (e.g., mean)
mean_result = x.mean()

# Trigger the computation
result = mean_result.compute()

print(f"The mean of the array is: {result}")
```

```
The mean of the array is: 0.4999540542630303
```

```
"""
b. Apache Spark

Apache Spark is a distributed computing framework designed for fast computation,
ideal for big data processing.
"""

!pip install pyspark

from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("example").getOrCreate()

# Create a DataFrame
data = [("Alice", 1), ("Bob", 2), ("Cathy", 3)]
df = spark.createDataFrame(data, ["Name", "Value"])

# Perform operations
result = df.groupBy("Name").sum("Value").show()
```

```
Collecting pyspark
  Downloading pyspark-3.5.3.tar.gz (317.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 317.3/317.3 MB 1.6 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pyspark) (0.
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.5.3-py2.py3-none-any.whl size=317840625 sha256=e54b445cce7
  Stored in directory: /root/.cache/pip/wheels/1b/3a/92/28b93e2fbfdbb07509ca4d6f50c5e407f48dce4ddbda69a4ab
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.5.3
```

```
+-----+----------+
| Name|sum(Value)|
+-----+----------+
|Alice|         1|
|  Bob|         2|
|Cathy|         3|
+-----+----------+
```

```
import time
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg

spark = SparkSession.builder.appName("example_spark").getOrCreate()

data = [(i,i*2) for i in range(10000000)]

columns = ["id" ,"value"]
df = spark.createDataFrame(data, columns)

start_time = time.time()
result = df.select(avg("value")).collect()[0][0]
end_time = time.time()

print(f"Spark Result: {result}, Time: {end_time - start_time:.5f} seconds")
```

```
Spark Result: 9999999.0, Time: 9.69326 seconds
```

```
import time
data = [(i,i*2) for i in range(10000000)]
start_time = time.time()
meanvaluenospark = sum([x[1] for x in data])/len(data)
end_time = time.time()
print(f"Spark Result: {meanvaluenospark}, Time: {end_time - start_time:.5f} seconds")
```

```
Spark Result: 9999999.0, Time: 1.08159 seconds
```

Optimization Strategies for Scaling

When scaling computations, consider the following strategies:

Chunking: Divide large datasets into smaller chunks that can be processed in parallel.

Load Balancing: Ensure that the workload is evenly distributed among available cores or machines.

Data Locality: Minimize data transfer times by processing data where it resides.

Optimized Algorithms: Use algorithms with lower time complexity and memory usage

```
import dask.array as da
import dask

# Step 1: Create a large Dask array with chunking
# Create an array of random numbers (e.g., 10 million elements) and chunk it into smaller arrays
large_array = da.random.random(size=(10000000,), chunks=(1000000,))

# Step 2: Define an optimized algorithm function
```

```
def optimized_computation(x):
    """A simple optimized computation: Calculate the square and then the mean."""
    return da.mean(x ** 2)

# Step 3: Perform the computation
# Use Dask to apply the optimized function to the large array
mean_result = optimized_computation(large_array)

# Step 4: Trigger computation (this uses load balancing internally)
result = mean_result.compute()  # This will execute the task in parallel, leveraging multiple cores

# Step 5: Print the result
print(f"The mean of the squares of the array is: {result}")
```

```
The mean of the squares of the array is: 0.33338164035171375
```

Asynchronous Programming

Asynchronous programming in advanced Python allows code to be executed concurrently, enabling tasks to run in the background without blocking the main thread.

It’s ideal for I/O-bound tasks like network operations, file handling, or database access.

Python provides tools like asyncio, await, and async to manage asynchronous workflows efficiently.

Use Cases:

Web Scraping: Gathering data from multiple sources simultaneously without blocking.

API Calls: Handling multiple API requests concurrently, reducing latency.

File Operations: Reading or writing large files asynchronously to prevent application slowdowns.

```
"""
async and await:

Functions defined with async def become coroutines that can be paused and resumed.
await is used to call an asynchronous function or coroutine, allowing other tasks
to run during the wait time.

"""

import asyncio

async def fetch_data():
    print("Fetching data...")
    await asyncio.sleep(2)
    return "Data fetched"

async def main():
    result = await fetch_data()
    print(result)

# Directly await the `main` function
await main()
```

```
Fetching data...
Data fetched
```

```
"""
Event Loop:

The asyncio library uses an event loop to schedule and run asynchronous tasks.
The event loop manages when and how the coroutines are executed,
providing concurrency without requiring multiple threads.

Tasks:

Coroutines can be wrapped into Task objects that run asynchronously.
Tasks allow the program to execute multiple coroutines simultaneously (in an interleaved manner).
"""

"""
Python program that demonstrates the use of the asyncio library and an event loop to schedule
and run asynchronous tasks. This example
will show how multiple coroutines can be executed concurrently without blocking the main thread:

"""
import asyncio

from concurrent.futures import ProcessPoolExecutor

# Define an asynchronous function that simulates a task
async def task(name, delay):
    print(f"Task {name} started")  # Indicate the task has started
    await asyncio.sleep(delay)  # Simulate an I/O-bound operation (e.g., network request)
    print(f"Task {name} completed after {delay} seconds")  # Indicate the task has completed

# Define the main asynchronous function
async def main():
    # Create a list of tasks with varying delays
    tasks = [
        task("A", 2),  # Task A takes 2 seconds
        task("B", 1),  # Task B takes 1 second
        task("C", 3),  # Task C takes 3 seconds
    ]

    # Use asyncio.gather to run tasks concurrently
    await asyncio.gather(*tasks)

# Directly await the `main` function in Colab
await main()
```

```
Task A started
Task B started
Task C started
Task B completed after 1 seconds
Task A completed after 2 seconds
Task C completed after 3 seconds
```

Web Scraping Example

```python
#!pip install aiohttp beautifulsoup4 nest_asyncio

import asyncio
import aiohttp
from bs4 import BeautifulSoup
import nest_asyncio

# Apply nest_asyncio to allow nested event loops
nest_asyncio.apply()

# List of URLs to scrape
urls = [
    "https://example.com",
    "https://httpbin.org/get",
    "https://quotes.toscrape.com/",
    # Add more URLs as needed
]

async def fetch(session, url):
    async with session.get(url) as response:
        # Check for successful response
        if response.status == 200:
            html = await response.text()
            return html
        return None

async def scrape(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(session, url) for url in urls]
        return await asyncio.gather(*tasks)

def parse_html(html):
    soup = BeautifulSoup(html, 'html.parser')
    title_tag = soup.title
    # Check if the title tag exists and return its text; otherwise, return a default message
    return title_tag.string if title_tag else "No title found"

async def main():
    # Scrape the URLs
    html_responses = await scrape(urls)
    results = []

    for html in html_responses:
        if html:
            result = parse_html(html)
            results.append(result)

    # Save results to a text file
    with open('/content/scraped_data.txt', 'w') as f:
        for result in results:
            f.write(result + '\n')

# Run the main function using asyncio.run() in the already running event loop
await main()

# Download the scraped data file
from google.colab import files
files.download('/content/scraped_data.txt')
```

API Calls example

```
#!pip install aiohttp nest_asyncio

import asyncio
import aiohttp
import nest_asyncio

# Apply nest_asyncio to allow nested event loops
nest_asyncio.apply()

# List of API endpoints to call
api_urls = [
    "https://jsonplaceholder.typicode.com/posts/1",
    "https://jsonplaceholder.typicode.com/posts/2",
    "https://jsonplaceholder.typicode.com/posts/3",
    # Add more API endpoints as needed
]

async def fetch(session, url):
    async with session.get(url) as response:
        # Check for successful response
        if response.status == 200:
            data = await response.json()  # Parse response as JSON
            return data
        return None

async def main():
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(session, url) for url in api_urls]
        results = await asyncio.gather(*tasks)

        # Process and print the results
        for result in results:
            if result is not None:
                print(result)
            else:
                print("Failed to fetch data.")

# Run the main function
await main()
```

```
{'userId': 1, 'id': 1, 'title': 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit
{'userId': 1, 'id': 2, 'title': 'qui est esse', 'body': 'est rerum tempore vitae\nsequi sint nihil repreher
{'userId': 1, 'id': 3, 'title': 'ea molestias quasi exercitationem repellat qui ipsa sit aut', 'body': 'et
```

File Operations Example

```
#!pip install aiofiles

import asyncio
import aiofiles
```

```
# Function to write data to a file asynchronously
async def write_large_file(filename, data):
    async with aiofiles.open(filename, mode='w') as f:
        await f.write(data)

# Function to read data from a file asynchronously
async def read_large_file(filename):
    async with aiofiles.open(filename, mode='r') as f:
        content = await f.read()
        return content

async def main():
    filename = '/content/scraped_data.txt'

    # Simulate writing large data
    large_data = "This is a line in the file.\n" * 10000  # Example large data
    await write_large_file(filename, large_data)
    print(f"Written to {filename}.")

    # Read the data back from the file
    content = await read_large_file(filename)
    print(f"Read from {filename}: {content[:100]}...")  # Print the first 100 characters

# Run the main function
await main()
```

```
Written to /content/scraped_data.txt.
Read from /content/scraped_data.txt: This is a line in the file.
This is a line in the file.
This is a line in the file.
This is a line i...
```

```
!pip install aiofiles
```

```
Collecting aiofiles
  Downloading aiofiles-24.1.0-py3-none-any.whl.metadata (10 kB)
Downloading aiofiles-24.1.0-py3-none-any.whl (15 kB)
Installing collected packages: aiofiles
Successfully installed aiofiles-24.1.0
```

Database Integration and Optimization

Integrating and optimizing a database in Python typically involves using libraries that facilitate communication with various database systems (like SQLite, PostgreSQL, MySQL, etc.) and optimizing queries for performance.

```
!pip install sqlalchemy

import sqlite3
import pandas as pd
from sqlalchemy import create_engine

# Create a SQLite database connection
conn = sqlite3.connect('example.db')

# Create a cursor object
cursor = conn.cursor()
```

```
# Create a table
cursor.execute('''
CREATE TABLE IF NOT EXISTS employees (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER NOT NULL,
    department TEXT NOT NULL
)
''')

# Function to insert data into the database
def insert_data(data):
    cursor.executemany('''
    INSERT INTO employees (name, age, department) VALUES (?, ?, ?)
    ''', data)
    conn.commit()

# Generate sample data
sample_data = [
    ("Alice", 30, "HR"),
    ("Bob", 24, "Engineering"),
    ("Charlie", 28, "Marketing"),
    ("David", 35, "Engineering"),
    ("Eve", 29, "HR"),
    ("Frank", 32, "Marketing")
]

# Insert sample data into the database
insert_data(sample_data)

# Function to query data from the database
def query_data():
    return pd.read_sql_query('SELECT * FROM employees', conn)

# Query and display the data
dataframe = query_data()
print("Data before optimization:")
print(dataframe)

# Optimize by creating an index on the department column
cursor.execute('CREATE INDEX IF NOT EXISTS idx_department ON employees(department)')

# Query data again to show optimization effect
optimized_dataframe = pd.read_sql_query('SELECT * FROM employees WHERE department = "Engineering"', conn)
print("\nData after optimization (filtered by department):")
print(optimized_dataframe)

# Clean up and close the connection
conn.close()
```

```
Requirement already satisfied: sqlalchemy in /usr/local/lib/python3.10/dist-packages (2.0.35)
Requirement already satisfied: typing-extensions>=4.6.0 in /usr/local/lib/python3.10/dist-packages (from sq
Requirement already satisfied: greenlet!=0.4.17 in /usr/local/lib/python3.10/dist-packages (from sqlalchemy
Data before optimization:
   id     name  age   department
0   1    Alice   30           HR
1   2      Bob   24  Engineering
2   3  Charlie   28    Marketing
3   4    David   35  Engineering
4   5      Eve   29           HR
```

```
5   6    Frank   32    Marketing

Data after optimization (filtered by department):
   id   name  age   department
0   2    Bob   24  Engineering
1   4  David   35  Engineering
```

To integrate and optimize a free cloud-based database with Python, you can use Google Cloud Firestore or ElephantSQL (for PostgreSQL). I'll demonstrate using ElephantSQL as it provides a free-tier PostgreSQL instance and is commonly used for relational database operations.

Step-by-Step Guide Using PostgreSQL (ElephantSQL)

Set Up ElephantSQL

Create a Free PostgreSQL Instance on ElephantSQL:

Go to ElephantSQL.

Sign up and create a new PostgreSQL instance under the free plan (Tiny Turtle).

After creation, you'll receive the database URL which you'll need to connect

```
!pip install psycopg2-binary sqlalchemy pandas
```

```
import pandas as pd
from sqlalchemy import create_engine
import psycopg2

# Replace this with your own ElephantSQL connection string
DATABASE_URL = 'postgresql://username:password@hostname/dbname'

# Create a database engine using SQLAlchemy
engine = create_engine(DATABASE_URL)

# Function to create a table if it doesn't exist
def create_table():
    query = '''
    CREATE TABLE IF NOT EXISTS employees (
        id SERIAL PRIMARY KEY,
        name VARCHAR(50),
        age INTEGER,
        department VARCHAR(50)
    )
    '''
    with engine.connect() as conn:
        conn.execute(query)
    print("Table created successfully!")

# Function to insert data into the cloud database
def insert_data(data):
    query = '''
    INSERT INTO employees (name, age, department) VALUES (%s, %s, %s)
    '''
    with engine.connect() as conn:
        conn.execute(query, data)
    print("Data inserted successfully!")
```

```
# Function to bulk insert data into the cloud database
def bulk_insert_data(data):
    with engine.connect() as conn:
        conn.execute(
            "INSERT INTO employees (name, age, department) VALUES (%s, %s, %s)", data)
    print("Bulk data inserted successfully!")

# Function to query data from the cloud database
def query_data():
    df = pd.read_sql_query('SELECT * FROM employees', engine)
    return df

# Function to optimize by creating an index
def optimize_query():
    query = '''
    CREATE INDEX IF NOT EXISTS idx_department ON employees(department)
    '''
    with engine.connect() as conn:
        conn.execute(query)
    print("Index created on department column!")

# Main execution flow
if __name__ == "__main__":
    # Step 1: Create the table
    create_table()

    # Step 2: Insert sample data
    sample_data = [
        ("Alice", 30, "HR"),
        ("Bob", 24, "Engineering"),
        ("Charlie", 28, "Marketing"),
        ("David", 35, "Engineering"),
        ("Eve", 29, "HR"),
        ("Frank", 32, "Marketing")
    ]
    bulk_insert_data(sample_data)

    # Step 3: Query and print the data
    data_frame = query_data()
    print("Data before optimization:")
    print(data_frame)

    # Step 4: Optimize queries by creating an index
    optimize_query()

    # Step 5: Query optimized data
    optimized_df = pd.read_sql_query('SELECT * FROM employees WHERE department = %s', engine, params=('Engineer
    print("\nData after optimization (filtered by department):")
    print(optimized_df)



# prompt: rewrite above code to connect AWS RDS dataset

import pandas as pd
import psycopg2
from sqlalchemy import create_engine
!pip install psycopg2-binary sqlalchemy pandas


# Replace with your RDS credentials
```

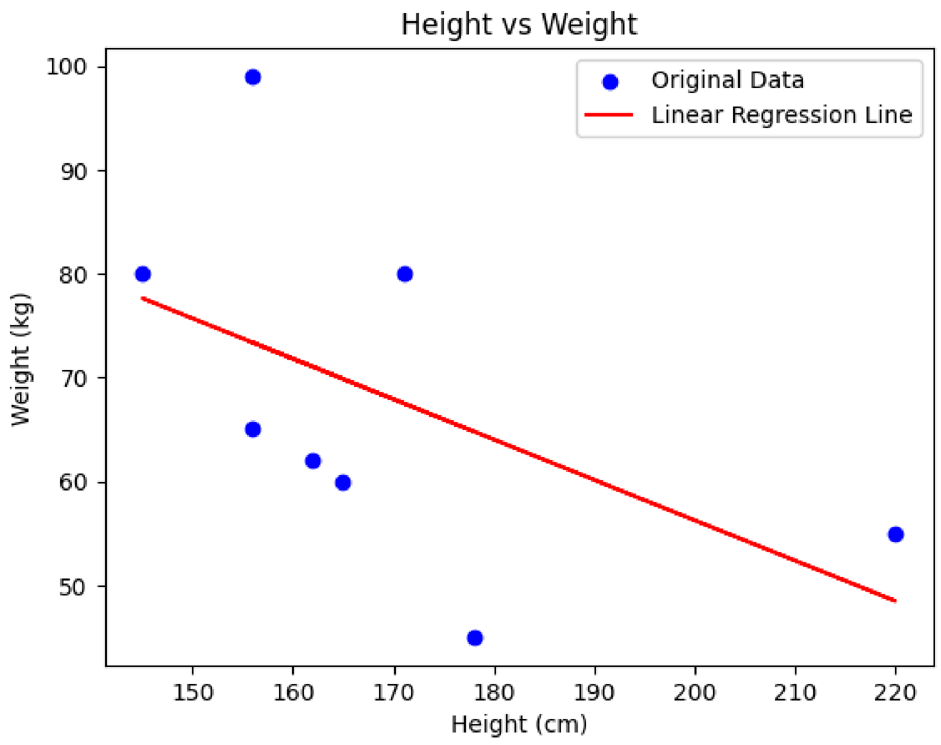
```
RDS_HOST = 'your_rds_host'
RDS_PORT = 'your_rds_port'  # Default is 5432
RDS_DATABASE = 'your_rds_database'
RDS_USER = 'your_rds_user'
RDS_PASSWORD = 'your_rds_password'

# Create a database engine using SQLAlchemy
engine = create_engine(f"postgresql://{RDS_USER}:{RDS_PASSWORD}@{RDS_HOST}:{RDS_PORT}/{RDS_DATABASE}")

# Function to create a table if it doesn't exist
def create_table():
    query = '''
    CREATE TABLE IF NOT EXISTS employees (
        id SERIAL PRIMARY KEY,
        name VARCHAR(50),
        age INTEGER,
        department VARCHAR(50)
    )
    '''
    with engine.connect() as conn:
        conn.execute(query)
    print("Table created successfully!")

# Function to insert data into the cloud database
def insert_data(data):
    query = '''
    INSERT INTO employees (name, age, department) VALUES (%s, %s, %s)
    '''
    with engine.connect() as conn:
        conn.execute(query, data)
    print("Data inserted successfully!")

# Function to bulk insert data into the cloud database
def bulk_insert_data(data):
    with engine.connect() as conn:
        conn.execute(
            "INSERT INTO employees (name, age, department) VALUES (%s, %s, %s)", data)
    print("Bulk data inserted successfully!")

# Function to query data from the cloud database
def query_data():
    df = pd.read_sql_query('SELECT * FROM employees', engine)
    return df

# Function to optimize by creating an index
def optimize_query():
    query = '''
    CREATE INDEX IF NOT EXISTS idx_department ON employees(department)
    '''
    with engine.connect() as conn:
        conn.execute(query)
    print("Index created on department column!")

# Main execution flow
if __name__ == "__main__":
    # Step 1: Create the table
    create_table()

    # Step 2: Insert sample data
    sample_data = [
        ("Alice", 30, "HR"),
        ("Bob", 24, "Engineering"),
        ("Charlie", 28, "Marketing"),
        ("David", 35, "Engineering"),
        ("Eve", 29, "HR"),
        ("Frank", 32, "Marketing")
    ]
    bulk_insert_data(sample_data)

    # Step 3: Query and print the data
    data_frame = query_data()
    print("Data before optimization:")
    print(data_frame)

    # Step 4: Optimize queries by creating an index
    optimize_query()

    # Step 5: Query optimized data
    optimized_df = pd.read_sql_query('SELECT * FROM employees WHERE department = %s', engine, params=('Engineer
    print("\nData after optimization (filtered by department):")
    print(optimized_df)
```

Machine Learning Optimization

Optimization in machine learning is a critical process that involves tuning model parameters to minimize (or maximize) a cost function to improve model performance. Optimization can be applied to both model training (e.g., tuning model weights) and hyperparameter tuning (e.g., selecting optimal learning rates)

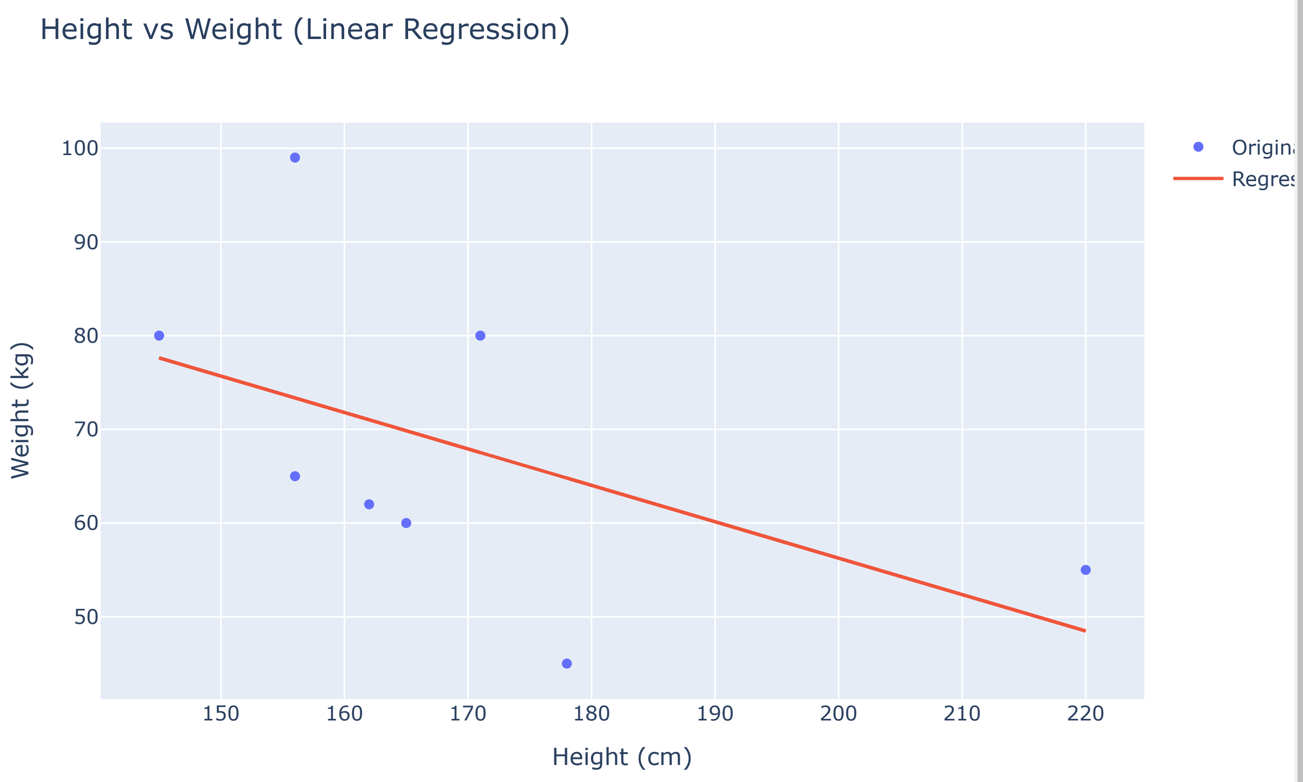
```
import numpy as np
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

# Step 1: Define height (X) and weight (y)
height = np.array([171, 165, 145, 162, 156, 220, 156, 178])
weight = np.array([80, 60, 80, 62, 65, 55, 99, 45])

# Fit linear regression model
clf = LinearRegression()
clf.fit(height.reshape(-1, 1), weight)

# Predict the weight for height = 136
predicted_weight = clf.predict([[136]])
print(f"Predicted weight for height 136 cm: {predicted_weight[0]:.2f} kg")

# Plot the data and the regression line
plt.scatter(height, weight, color='blue', label='Original Data')
plt.plot(height, clf.predict(height.reshape(-1, 1)), color='red', label='Linear Regression Line')
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')
plt.title('Height vs Weight')
plt.legend()
plt.show()
```

```
Predicted weight for height 136 cm: 81.12 kg
```

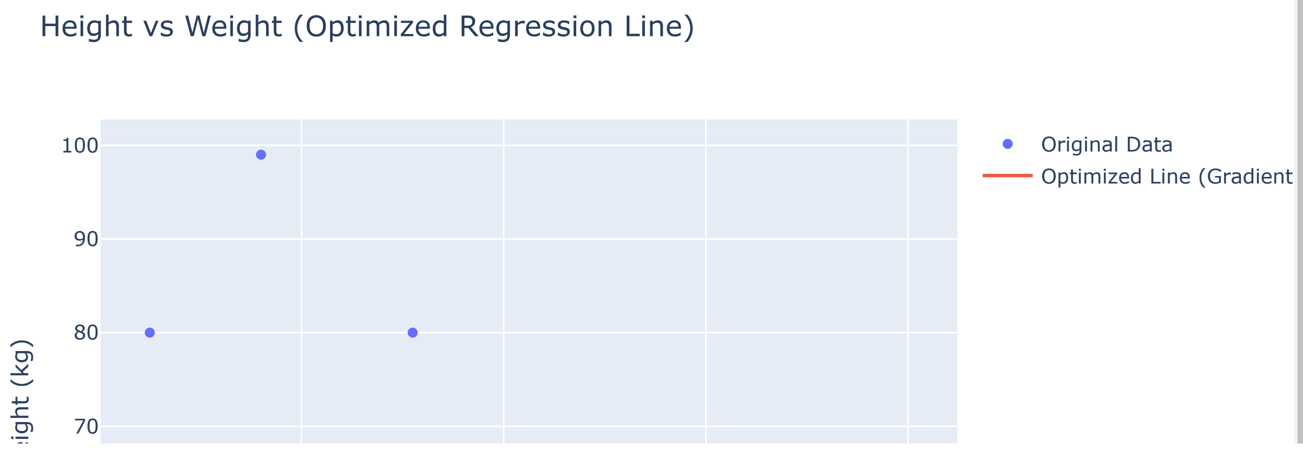


```
import numpy as np
import plotly.graph_objects as go
from sklearn.linear_model import LinearRegression

# Step 1: Data (Height and Weight)
height = np.array([171, 165, 145, 162, 156, 220, 156, 178])
weight = np.array([80, 60, 80, 62, 65, 55, 99, 45])

# Step 2: Linear Regression using sklearn
model = LinearRegression()
model.fit(height.reshape(-1, 1), weight)

# Predict weight for a given height (136 cm)
predicted_weight = model.predict([[136]])
print(f"Predicted weight for height 136 cm: {predicted_weight[0]:.2f} kg")

# Step 3: Plot Original Data and Regression Line using Plotly
# Create scatter plot for original data
scatter_data = go.Scatter(x=height, y=weight, mode='markers', name='Original Data')

# Create line plot for the regression line
line_data = go.Scatter(x=height, y=model.predict(height.reshape(-1, 1)), mode='lines', name='Regression Line')

# Display the plot
fig = go.Figure([scatter_data, line_data])
fig.update_layout(title='Height vs Weight (Linear Regression)',
                  xaxis_title='Height (cm)', yaxis_title='Weight (kg)')
fig.show()

# Step 4: Simplified Gradient Descent
def simple_gradient_descent(X, y, learning_rate=0.0001, iterations=1000):
    m = len(y)
    theta0 = 0  # Intercept (bias)
    theta1 = 0  # Slope
    for _ in range(iterations):
        y_pred = theta0 + theta1 * X  # Linear prediction
        d_theta0 = -(2/m) * np.sum(y - y_pred)  # Gradient for intercept
        d_theta1 = -(2/m) * np.sum((y - y_pred) * X)  # Gradient for slope
        theta0 -= learning_rate * d_theta0
        theta1 -= learning_rate * d_theta1

    return theta0, theta1

# Perform Gradient Descent
theta0, theta1 = simple_gradient_descent(height, weight)

# Step 5: Optimized Prediction using Gradient Descent
optimized_predicted_weight = theta0 + theta1 * 136
print(f"Optimized predicted weight for height 136 cm: {optimized_predicted_weight:.2f} kg")

# Step 6: Plot Optimized Line
optimized_line_data = go.Scatter(x=height, y=theta0 + theta1 * height, mode='lines', name='Optimized Line (Grad

# Display original data and optimized line
fig_optimized = go.Figure([scatter_data, optimized_line_data])
fig_optimized.update_layout(title='Height vs Weight (Optimized Regression Line)',
                            xaxis_title='Height (cm)', yaxis_title='Weight (kg)')
fig_optimized.show()
```

```
Predicted weight for height 136 cm: 81.12 kg
```



```
Optimized predicted weight for height 136 cm: nan kg
/usr/local/lib/python3.10/dist-packages/numpy/core/fromnumeric.py:88: RuntimeWarning:

overflow encountered in reduce

<ipython-input-20-b21aa9e55f87>:41: RuntimeWarning:

invalid value encountered in scalar subtract
```



Decision Tree

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
```

```
import pandas as pd
import numpy as np

df = pd.read_csv('/content/dtdb.csv')

lb = LabelEncoder()
df['outlook_'] = lb.fit_transform(df['outlook'])
df['temp_'] = lb.fit_transform(df['temp'] )
df['humidity_'] = lb.fit_transform(df['humidity'] )
df['windy_'] = lb.fit_transform(df['windy'] )
df['play_'] = lb.fit_transform(df['play'] )
X = df.iloc[:,5:9]
Y = df.iloc[:,9]

X_train, X_test , y_train,y_test = train_test_split(X, Y, test_size = 0.4, random_state = 100)

clf_entropy = DecisionTreeClassifier(criterion='entropy')
clf_entropy.fit(X_train.astype(int),y_train.astype(int))
y_pred_en = clf_entropy.predict(X_test)

print("Accuracy is :{0}".format(accuracy_score(y_test.astype(int),y_pred_en) * 100))
```

```
Accuracy is :66.66666666666666
```

To optimize the performance of your decision tree classifier and improve accuracy, you can explore various approaches, such as:

Hyperparameter Tuning: Adjusting parameters like max_depth, min_samples_split, min_samples_leaf, and criterion can significantly impact the model's accuracy.

Feature Engineering: Ensuring that your features are well-transformed, and adding new informative features might help.

Cross-Validation: Using cross-validation instead of a single train-test split can give a more robust measure of performance.

Advanced Models: Using more complex models such as Random Forest or Gradient Boosting may also improve accuracy.

Start coding or generate with AI.

```
"""
1. Hyperparameter Tuning using Grid Search:
Here, we'll try tuning the hyperparameters of the decision tree using GridSearchCV,
a popular method for finding the optimal set of parameters.
"""

from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {
    'criterion': ['gini', 'entropy'],               # Try both Gini and Entropy
    'max_depth': [3, 5, 10, None],                  # Try different tree depths
    'min_samples_split': [2, 5, 10],                # Minimum number of samples to split a node
    'min_samples_leaf': [1, 2, 4],                  # Minimum number of samples required in a leaf
    'max_features': [None, 'sqrt', 'log2'],         # Try limiting features at each split
}
```

```
# Create a base Decision Tree model
clf = DecisionTreeClassifier()

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5, n_jobs=-1, verbose=1)

# Fit grid search
grid_search.fit(X_train, y_train)

# Best hyperparameters found
print(f"Best hyperparameters: {grid_search.best_params_}")

# Predict with the best estimator
best_clf = grid_search.best_estimator_
y_pred_optimized = best_clf.predict(X_test)

# Check accuracy after optimization
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)
print(f"Optimized Accuracy: {accuracy_optimized * 100:.2f}%")
```

```
Fitting 5 folds for each of 216 candidates, totalling 1080 fits
/usr/local/lib/python3.10/dist-packages/sklearn/model_selection/_split.py:776: UserWarning:

The least populated class in y has only 3 members, which is less than n_splits=5.

Best hyperparameters: {'criterion': 'entropy', 'max_depth': 5, 'max_features': 'log2', 'min_samples_leaf':
Optimized Accuracy: 83.33%
```

```
"""
2. Cross-Validation for Robust Accuracy:
Instead of a single train-test split, cross-validation can give you a better understanding
of how well your model generalizes.
"""

from sklearn.model_selection import cross_val_score

# Perform 5-fold cross-validation
cv_scores = cross_val_score(best_clf, X, Y, cv=5)

# Average accuracy over 5 folds
mean_cv_accuracy = np.mean(cv_scores) * 100
print(f"Cross-validated accuracy: {mean_cv_accuracy:.2f}%")
```

```
Cross-validated accuracy: 73.33%
```

To create a stock recommendation system with sample stock market data and provide buy/sell recommendations, we can break this down into the following steps:

Generate Sample Stock Market Data: We'll create stock data with features like opening price, closing price, high, low, volume, and whether the stock price increased or decreased.

Train a Model: We'll use a simple classification model (e.g., Decision Tree) to recommend if the stock should be bought or not based on the data.

Optimize the Model: Use hyperparameter tuning or try more advanced models to increase the accuracy.

Evaluate Accuracy: We'll use a metric like accuracy to evaluate the performance of the recommendation system.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Create sample stock market data
np.random.seed(42)

# Generate synthetic stock data for 1000 days
data = {
    'open': np.random.uniform(100, 200, 1000),  # Opening price
    'high': np.random.uniform(100, 200, 1000),  # High price
    'low': np.random.uniform(100, 200, 1000),   # Low price
    'close': np.random.uniform(100, 200, 1000), # Closing price
    'volume': np.random.uniform(1000, 10000, 1000), # Volume traded
}

df = pd.DataFrame(data)

# Create a target variable: 1 if price increased, 0 if price decreased
df['price_increase'] = np.where(df['close'] > df['open'], 1, 0)

# Let's assume we want to recommend 'Buy' if the price increases and 'Not Buy' otherwise.
X = df[['open', 'high', 'low', 'volume']]
y = df['price_increase']

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)



# Train a Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)

# Make predictions
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Initial Accuracy: {accuracy * 100:.2f}%")
```

```
Initial Accuracy: 64.00%
```

```
"""
 Give Recommendations Based on Predictions
Now, based on the stock data, we can give recommendations.
If the prediction is 1, we recommend "Buy," otherwise "Not Buy."

"""

# Add recommendation to the test data
recommendations = np.where(y_pred == 1, 'Buy', 'Not Buy')
X_test['Recommendation'] = recommendations

# Display sample recommendations
print(X_test.head())
```

```
          open        high         low       volume Recommendation
521  138.089086  193.443603  113.443513  6194.053741            Buy
737  181.659944  119.462251  154.448233  4139.487167        Not Buy
740  146.267979  182.753790  136.568075  7468.463019        Not Buy
660  135.609673  160.059433  147.228180  5600.022746            Buy
411  195.006197  111.191962  114.055297  1037.686993        Not Buy
```

Optimization (Hyperparameter Tuning)

Let's optimize the model by tuning hyperparameters using GridSearchCV to improve accuracy.

```
from sklearn.model_selection import GridSearchCV

# Define the parameter grid for optimization
param_grid = {
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

# Grid Search
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5, n_jobs=-1, verbose=1)
grid_search.fit(X_train, y_train)

# Get the best estimator
best_clf = grid_search.best_estimator_


# Make predictions with optimized model
# Remove the 'Recommendation' column from X_test before prediction
y_pred_optimized = best_clf.predict(X_test.drop('Recommendation', axis=1))

# Calculate optimized accuracy
optimized_accuracy = accuracy_score(y_test, y_pred_optimized)
print(f"Optimized Accuracy: {optimized_accuracy * 100:.2f}%")

# Add optimized recommendation to test data
recommendations_optimized = np.where(y_pred_optimized == 1, 'Buy', 'Not Buy')
X_test['Optimized_Recommendation'] = recommendations_optimized

# Display sample recommendations with optimized model
print(X_test.head())
```

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
Optimized Accuracy: 79.00%
          open        high         low       volume Recommendation  \
521  138.089086  193.443603  113.443513  6194.053741            Buy
737  181.659944  119.462251  154.448233  4139.487167        Not Buy
740  146.267979  182.753790  136.568075  7468.463019        Not Buy
660  135.609673  160.059433  147.228180  5600.022746            Buy
411  195.006197  111.191962  114.055297  1037.686993        Not Buy

    Optimized_Recommendation
521                      Buy
737                  Not Buy
740                      Buy
660                      Buy
411                  Not Buy
```

Neural network using Keras

```
!pip install tensorflow
!pip install numpy pandas scikit-learn
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Create sample stock market data
np.random.seed(42)

# Generate synthetic stock data for 1000 days
data = {
    'open': np.random.uniform(100, 200, 1000),  # Opening price
    'high': np.random.uniform(100, 200, 1000),  # High price
    'low': np.random.uniform(100, 200, 1000),   # Low price
    'close': np.random.uniform(100, 200, 1000), # Closing price
    'volume': np.random.uniform(1000, 10000, 1000), # Volume traded
}

df = pd.DataFrame(data)

# Create a target variable: 1 if price increased, 0 if price decreased
df['price_increase'] = np.where(df['close'] > df['open'], 1, 0)

# Feature and target data
X = df[['open', 'high', 'low', 'volume']]
y = df['price_increase']

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.3, random_state=42)
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Build a simple neural network
model = Sequential([
    Dense(16, input_dim=4, activation='relu'),  # Input layer with 4 features
    Dense(8, activation='relu'),  # Hidden layer
    Dense(1, activation='sigmoid')  # Output layer (binary classification)
])

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=10, validation_data=(X_test, y_test))
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using

Epoch 1/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 9s 28ms/step - accuracy: 0.6306 - loss: 0.6623 - val_accuracy: 0.7367 - val_
Epoch 2/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.7108 - loss: 0.6087 - val_accuracy: 0.7567 - val_
Epoch 3/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7040 - loss: 0.5846 - val_accuracy: 0.7667 - val_
Epoch 4/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7208 - loss: 0.5601 - val_accuracy: 0.7833 - val_lo
Epoch 5/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.7402 - loss: 0.5340 - val_accuracy: 0.7867 - val_lo
Epoch 6/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - accuracy: 0.7253 - loss: 0.5434 - val_accuracy: 0.7933 - val_lo
Epoch 7/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.7525 - loss: 0.5139 - val_accuracy: 0.7933 - val_lo
Epoch 8/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7326 - loss: 0.5296 - val_accuracy: 0.7933 - val_lo
Epoch 9/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7944 - loss: 0.4805 - val_accuracy: 0.7867 - val_
Epoch 10/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - accuracy: 0.7504 - loss: 0.5161 - val_accuracy: 0.7867 - val_
Epoch 11/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7618 - loss: 0.5035 - val_accuracy: 0.7867 - val_lo
Epoch 12/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.7665 - loss: 0.4857 - val_accuracy: 0.7867 - val_
Epoch 13/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 2s 27ms/step - accuracy: 0.7794 - loss: 0.4957 - val_accuracy: 0.7833 - val_
Epoch 14/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.7254 - loss: 0.5423 - val_accuracy: 0.7833 - val_
Epoch 15/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 12ms/step - accuracy: 0.7688 - loss: 0.5133 - val_accuracy: 0.7833 - val_
Epoch 16/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7455 - loss: 0.5126 - val_accuracy: 0.7833 - val_lo
Epoch 17/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7739 - loss: 0.5023 - val_accuracy: 0.7867 - val_lo
Epoch 18/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7809 - loss: 0.4749 - val_accuracy: 0.7833 - val_lo
Epoch 19/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7747 - loss: 0.4865 - val_accuracy: 0.7867 - val_lo
Epoch 20/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7763 - loss: 0.4930 - val_accuracy: 0.7867 - val_lo
Epoch 21/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.7494 - loss: 0.5216 - val_accuracy: 0.7867 - val_lo
Epoch 22/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7478 - loss: 0.5193 - val_accuracy: 0.7867 - val_lo
Epoch 23/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7807 - loss: 0.4928 - val_accuracy: 0.7933 - val_lo
Epoch 24/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7798 - loss: 0.4778 - val_accuracy: 0.7900 - val_lo
Epoch 25/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7557 - loss: 0.5126 - val_accuracy: 0.7900 - val_lo
Epoch 26/50
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7508 - loss: 0.5126 - val_accuracy: 0.7900 - val_lo
Epoch 27/50
```

```
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Initial Model Accuracy: {accuracy * 100:.2f}%')
```

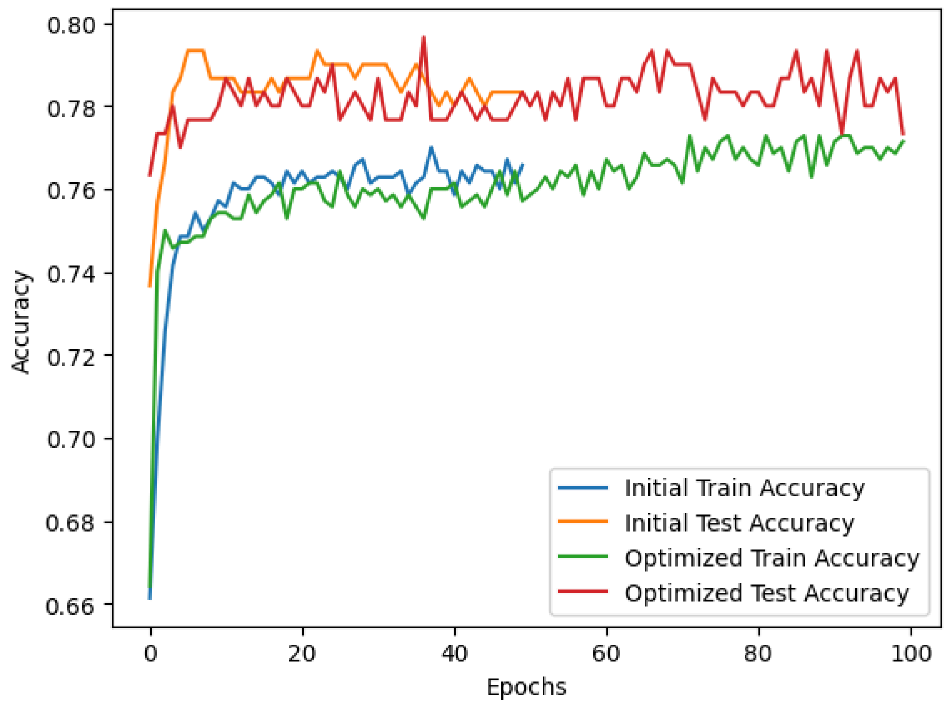
```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7998 - loss: 0.4382
Initial Model Accuracy: 78.33%
```

```
"""
Optimize the Neural Network
Let's optimize the model by increasing the number of neurons, changing the optimizer,
and using a different activation function.
"""
# Build an optimized neural network
model_optimized = Sequential([
    Dense(32, input_dim=4, activation='relu'),  # Increased neurons
    Dense(16, activation='relu'),  # More complex hidden layer
    Dense(1, activation='sigmoid')  # Output layer
])

# Compile with a different optimizer and more epochs
model_optimized.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the optimized model
history_optimized = model_optimized.fit(X_train, y_train, epochs=100, batch_size=10, validation_data=(X_test, y

# Evaluate the optimized model
loss_optimized, accuracy_optimized = model_optimized.evaluate(X_test, y_test)
print(f'Optimized Model Accuracy: {accuracy_optimized * 100:.2f}%')
```

```
Epoch 1/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6156 - loss: 0.6467 - val_accuracy: 0.7633 - val_lo
Epoch 2/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7277 - loss: 0.5579 - val_accuracy: 0.7733 - val_lo
Epoch 3/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7491 - loss: 0.5315 - val_accuracy: 0.7733 - val_lo
Epoch 4/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7503 - loss: 0.5231 - val_accuracy: 0.7800 - val_lo
Epoch 5/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7470 - loss: 0.5113 - val_accuracy: 0.7700 - val_lo
Epoch 6/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7729 - loss: 0.4907 - val_accuracy: 0.7767 - val_lo
Epoch 7/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7173 - loss: 0.5451 - val_accuracy: 0.7767 - val_lo
Epoch 8/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7881 - loss: 0.4715 - val_accuracy: 0.7767 - val_lo
Epoch 9/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7621 - loss: 0.4868 - val_accuracy: 0.7767 - val_lo
Epoch 10/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7883 - loss: 0.4654 - val_accuracy: 0.7800 - val_lo
Epoch 11/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7373 - loss: 0.5210 - val_accuracy: 0.7867 - val_lo
Epoch 12/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7212 - loss: 0.5313 - val_accuracy: 0.7833 - val_lo
Epoch 13/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7580 - loss: 0.5004 - val_accuracy: 0.7800 - val_lo
Epoch 14/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7760 - loss: 0.4859 - val_accuracy: 0.7867 - val_lo
Epoch 15/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7590 - loss: 0.5120 - val_accuracy: 0.7800 - val_lo
Epoch 16/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7435 - loss: 0.5276 - val_accuracy: 0.7833 - val_lo
Epoch 17/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7580 - loss: 0.5027 - val_accuracy: 0.7800 - val_lo
Epoch 18/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7467 - loss: 0.5222 - val_accuracy: 0.7800 - val_lo
Epoch 19/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7552 - loss: 0.5042 - val_accuracy: 0.7867 - val_lo
```

```
Epoch 20/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7514 - loss: 0.5138 - val_accuracy: 0.7833 - val_lo
Epoch 21/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7778 - loss: 0.4990 - val_accuracy: 0.7800 - val_lo
Epoch 22/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7403 - loss: 0.5059 - val_accuracy: 0.7800 - val_lo
Epoch 23/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7690 - loss: 0.4999 - val_accuracy: 0.7867 - val_lo
Epoch 24/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7787 - loss: 0.4778 - val_accuracy: 0.7833 - val_lo
Epoch 25/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7285 - loss: 0.5286 - val_accuracy: 0.7900 - val_lo
Epoch 26/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7495 - loss: 0.5061 - val_accuracy: 0.7767 - val_lo
Epoch 27/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7488 - loss: 0.4967 - val_accuracy: 0.7800 - val_lo
Epoch 28/100
70/70 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7779 - loss: 0.4830 - val_accuracy: 0.7833 - val_lo
```

```
import matplotlib.pyplot as plt

# Plot accuracy over epochs for both initial and optimized models
plt.plot(history.history['accuracy'], label='Initial Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Initial Test Accuracy')
plt.plot(history_optimized.history['accuracy'], label='Optimized Train Accuracy')
plt.plot(history_optimized.history['val_accuracy'], label='Optimized Test Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



Optimization techniques

Shallow/Deep Copy: Choose wisely to avoid unnecessary memory usage.

Avoid Copies: Use references, views, or memory-efficient data structures.

List Comprehension & Generators: Speed up loops and reduce memory usage.

Built-in Functions: Leverage faster built-ins like sum(), min(), etc.

Efficient Data Structures: Use sets, dictionaries, and NumPy arrays.

Parallel Processing: Use threading and multiprocessing for I/O and CPU-bound tasks.

NumPy: Optimize array and matrix operations.

Memoization: Cache expensive function results.

with Statements: Ensure resource efficiency with files or database connections.

Profiling Tools: Use cProfile to identify and optimize slow code segments.

Shallow Copy vs. Deep Copy

```
"""
Shallow Copy: Creates a new object but inserts references into it instead of
copying the original objects.
"""

import copy

original = [[1, 2, 3], [4, 5, 6]]
shallow_copy = copy.copy(original)  # Creates a new container, but elements inside are references to the origin
shallow_copy[0][0] = 99
print(original)  # Original data is modified as well
```

```
[[99, 2, 3], [4, 5, 6]]
```

```
"""
Deep Copy: Recursively copies all objects and creates a
new reference for every object, which is more memory-intensive.
"""
deep_copy = copy.deepcopy(original)
deep_copy[0][0] = 1
print(original)  # Original remains unchanged
```

```
[[99, 2, 3], [4, 5, 6]]
```

```
"""
Avoiding Unnecessary Copies
Avoid making unnecessary copies of large data structures, especially when working with lists or arrays.
 Use views or iterators instead to reduce memory overhead.
"""

original = [1, 2, 3, 4, 5]
sliced = original[:]  # Creates a new list (copy)

#Instead of copying, you can pass references or views:

array_view = memoryview(bytes(original))  # memoryview creates a view without copying data
```

```
"""
List Comprehension: A concise and optimized way to create lists
"""

# Instead of this
result = []
for i in range(1000):
    result.append(i * 2)

# Use this
result = [i * 2 for i in range(1000)]  # Faster and more readable
```

```
"""
Generator Expressions: Instead of creating a full list in memory,
 use generator expressions to lazily evaluate the elements.
"""

gen = (i * 2 for i in range(1000))  # More memory-efficient than list comprehension
```

```
"""
Use of with Statements for File Operations
The with statement ensures proper resource management
and can make file operations more efficient and readable.

"""

with open('large_file.txt', 'r') as f:
    data = f.read()  # Automatically closes the file after operation
```

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

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

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

Efficient Data Visualization

Fast plotting with Matplotlib and Seaborn

Reduce Data Size: When plotting large datasets, consider sampling or aggregating your data to reduce the number of points being plotted.

Use Faster Backends: Matplotlib supports various backends for rendering. You can switch to a faster backend for interactive plotting, like Agg or Qt5Agg

import matplotlib

matplotlib.use('Agg') # Use a non-interactive backend for speed

Batch Plotting: If you need to create multiple plots, consider creating them in a loop and showing them at once to reduce rendering time.

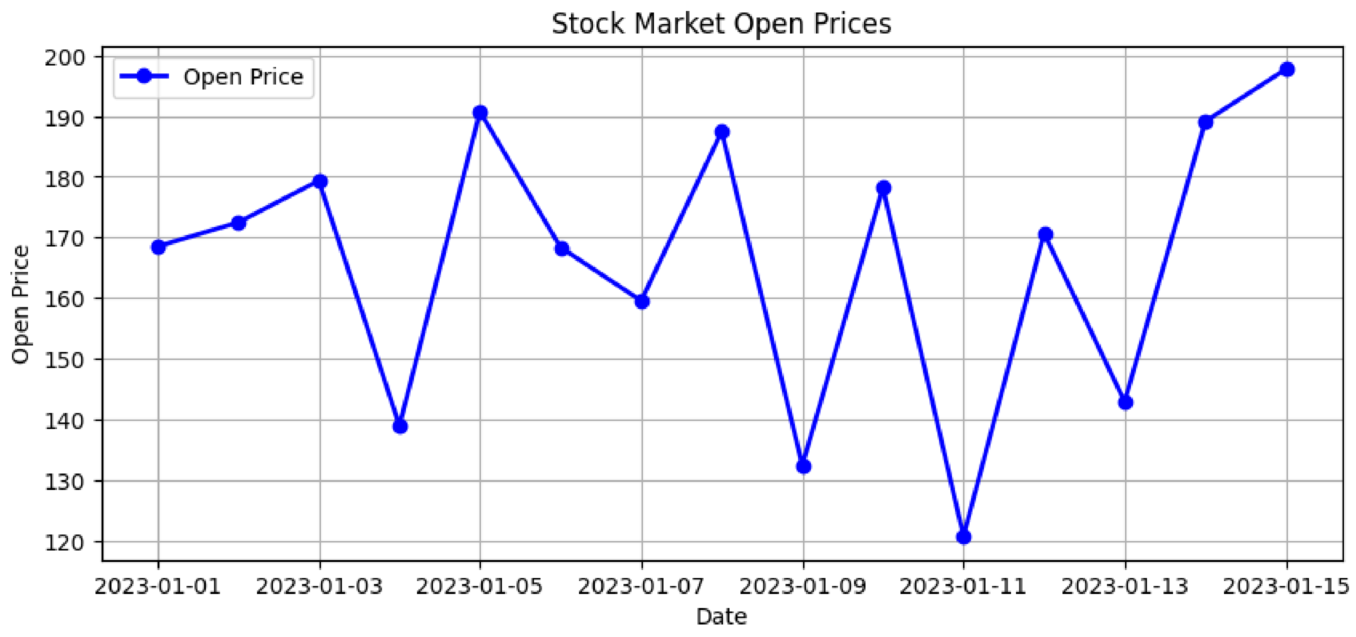
Avoid Redundant Settings: Set properties like labels and titles only once for multiple plots if they share common attributes.

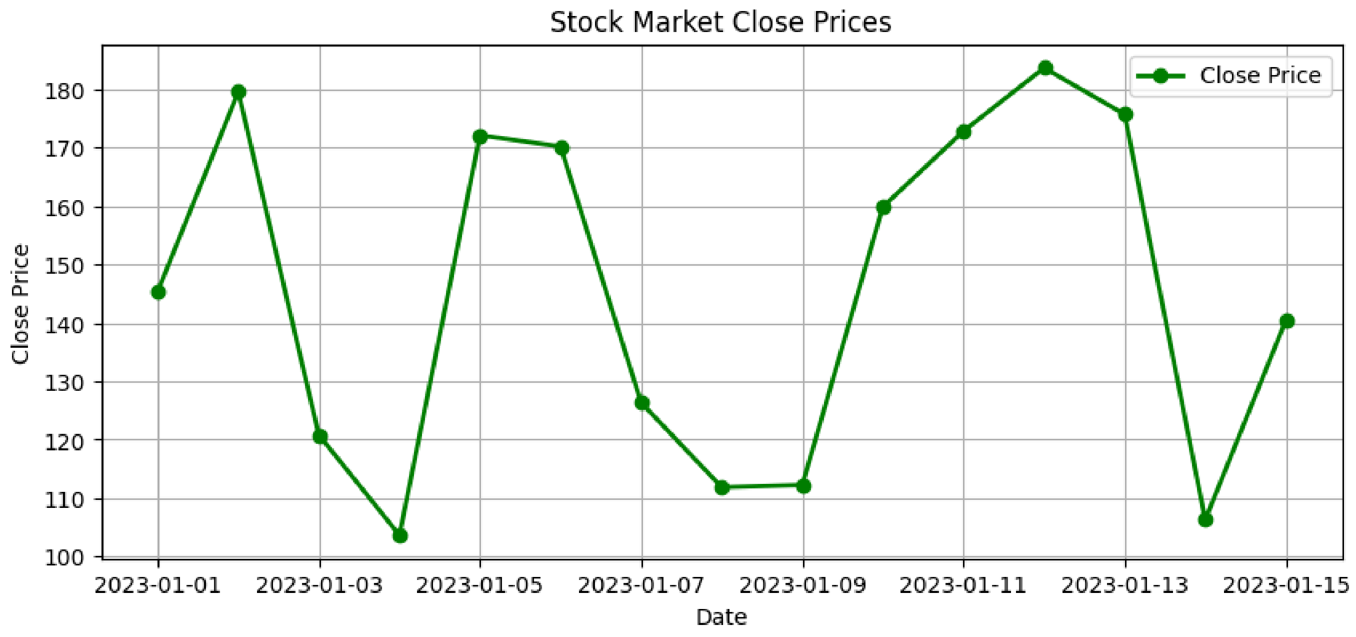
Caching: If generating plots takes time, consider saving them to disk and loading them when needed instead of regenerating them every time.

```
!pip install matplotlib seaborn pandas pillow
```

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

# Step 1: Use a faster backend for rendering
matplotlib.use('Agg')  # Use a non-interactive backend for speed

# Step 2: Generate sample stock market data
dates = pd.date_range(start='2023-01-01', periods=15)
data = {
    'Date': dates,
    'Open': np.random.uniform(100, 200, size=15),
    'Close': np.random.uniform(100, 200, size=15),
    'Volume': np.random.randint(1000, 5000, size=15)
}

# Create a DataFrame
stock_data = pd.DataFrame(data)

# Step 3: Sample or aggregate data if necessary (here we take the first 15 rows)
# Since we already have only 15 rows, we'll plot them directly.

# Step 4: Batch plotting
# Create multiple plots in a loop
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10, 10))
fig.subplots_adjust(hspace=0.4)  # Adjust space between plots

# Plot Open vs Date
axes[0].plot(stock_data['Date'], stock_data['Open'], marker='o', label='Open Price', color='blue', linewidth=2)
axes[0].set_title('Stock Market Open Prices')
axes[0].set_xlabel('Date')
axes[0].set_ylabel('Open Price')
axes[0].legend()
axes[0].grid(True)

# Plot Close vs Date
axes[1].plot(stock_data['Date'], stock_data['Close'], marker='o', label='Close Price', color='green', linewidth=
axes[1].set_title('Stock Market Close Prices')
axes[1].set_xlabel('Date')
axes[1].set_ylabel('Close Price')
axes[1].legend()
axes[1].grid(True)

# Step 5: Save the plots to disk instead of displaying them
plt.savefig('stock_market_plots.png')
plt.close()  # Close the figure to free up memory
```

```
# Step 6: Loading saved plots (as an example of caching)
from PIL import Image
saved_plot = Image.open('stock_market_plots.png')
saved_plot.show()  # This will display the saved plot in an image viewer



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

# Step 1: Set up inline plotting for Google Colab
%matplotlib inline

# Step 2: Generate sample stock market data
dates = pd.date_range(start='2023-01-01', periods=15)
data = {
    'Date': dates,
    'Open': np.random.uniform(100, 200, size=15),
    'Close': np.random.uniform(100, 200, size=15),
    'Volume': np.random.randint(1000, 5000, size=15)
}

# Create a DataFrame
stock_data = pd.DataFrame(data)

# Step 3: Sample or aggregate data if necessary (here we take the first 15 rows)
# Since we already have only 15 rows, we'll plot them directly.

# Step 4: Batch plotting
# Create multiple plots in a loop
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(10, 10))
fig.subplots_adjust(hspace=0.4)  # Adjust space between plots

# Plot Open vs Date
axes[0].plot(stock_data['Date'], stock_data['Open'], marker='o', label='Open Price', color='blue', linewidth=2)
axes[0].set_title('Stock Market Open Prices')
axes[0].set_xlabel('Date')
axes[0].set_ylabel('Open Price')
axes[0].legend()
axes[0].grid(True)

# Plot Close vs Date
axes[1].plot(stock_data['Date'], stock_data['Close'], marker='o', label='Close Price', color='green', linewidth
axes[1].set_title('Stock Market Close Prices')
axes[1].set_xlabel('Date')
axes[1].set_ylabel('Close Price')
axes[1].legend()
axes[1].grid(True)

# Step 5: Show the plots
plt.show()  # Display the plots in the notebook
```





Working with Time Series Data

Efficient time series analysis using Pandas

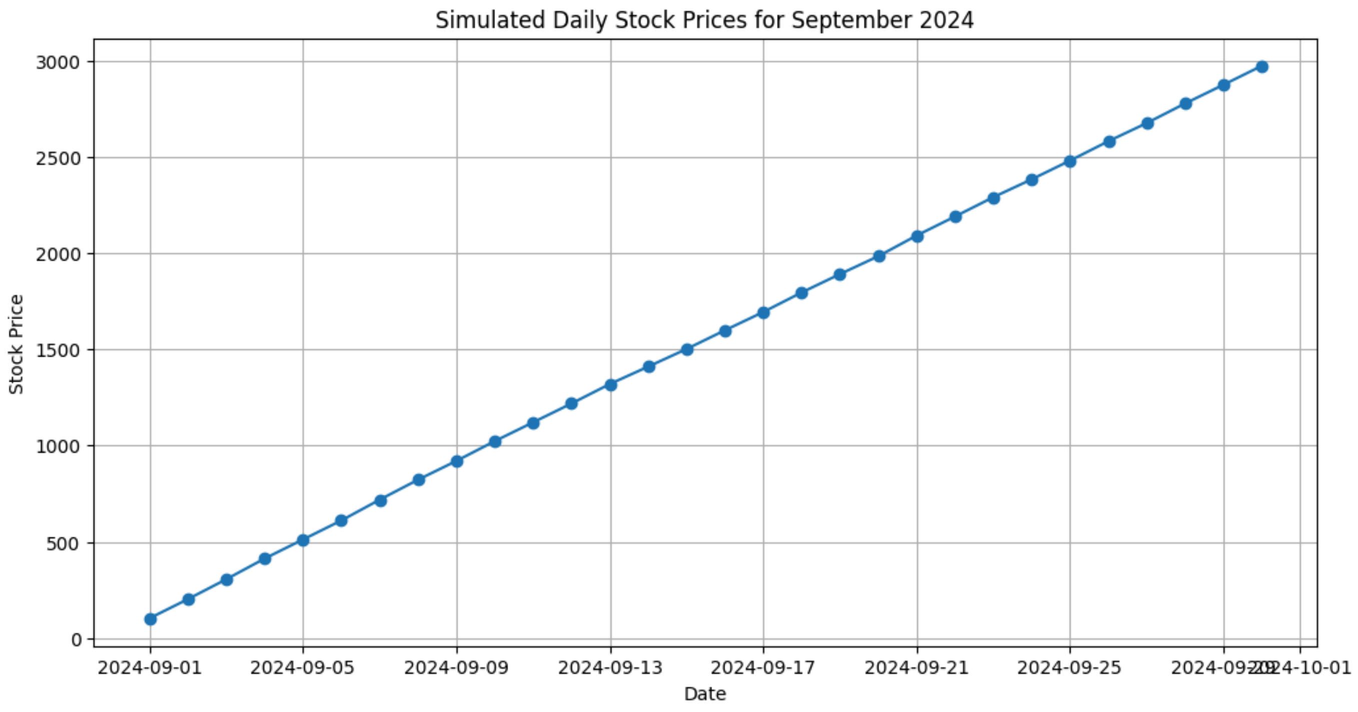
```
pip install pandas matplotlib
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.2.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.26
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from panc
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas) (202
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotli
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotli
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# Generate date range
dates = pd.date_range(start='2024-09-01', end='2024-09-30', freq='D')

# Create demo data: simulating stock prices
np.random.seed(42)  # For reproducibility
prices = np.random.normal(loc=100, scale=5, size=len(dates)).cumsum()

# Create DataFrame
data = pd.DataFrame(data={'Date': dates, 'Stock Price': prices})

# Set Date as the index
data.set_index('Date', inplace=True)

print(data.head())
```

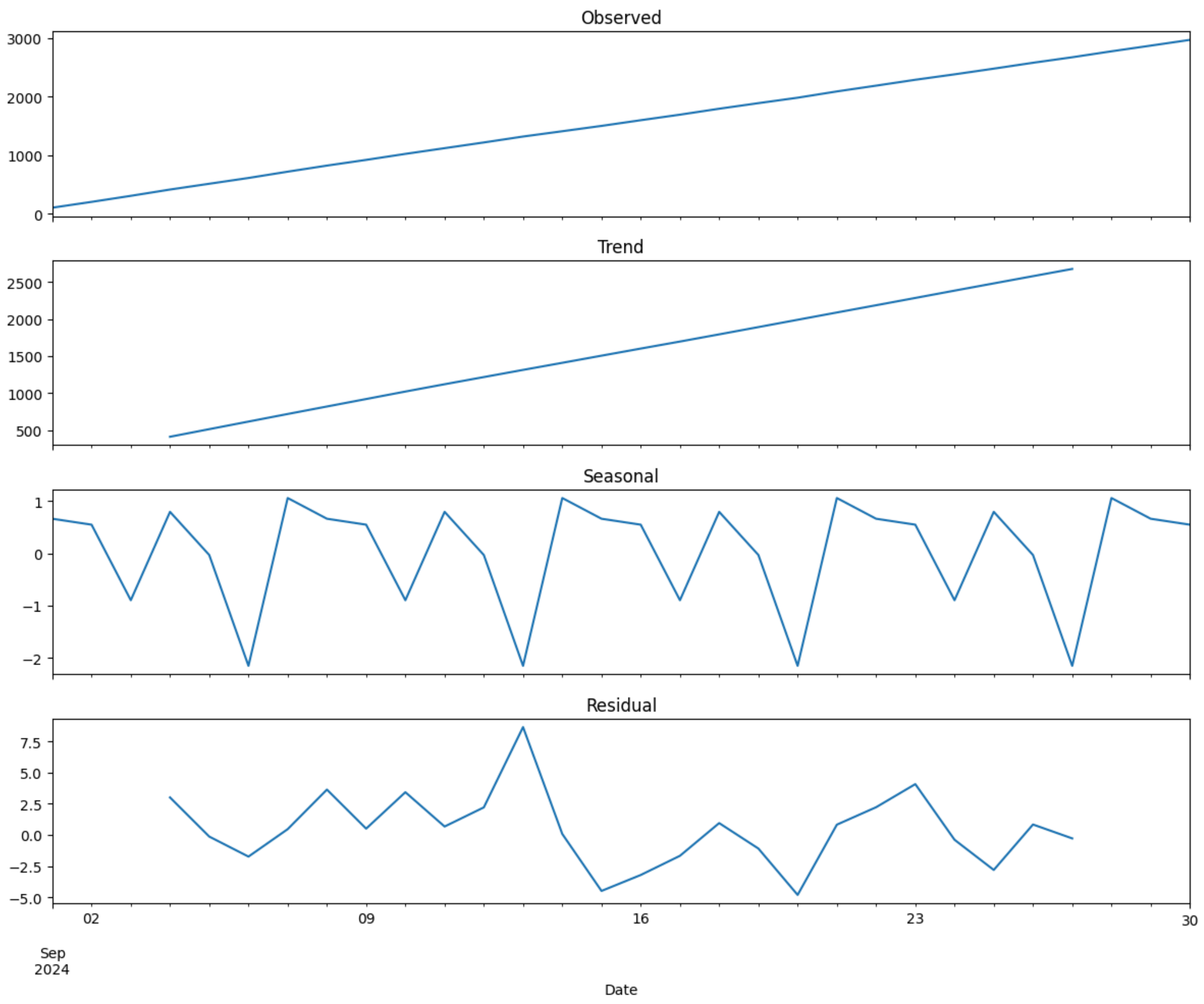
```
            Stock Price
     Date
     2024-09-01   102.483571
     2024-09-02   201.792249
     2024-09-03   305.030692
     2024-09-04   412.645841
     2024-09-05   511.475074
```

```
# Plot the time series data
plt.figure(figsize=(12, 6))
plt.plot(data.index, data['Stock Price'], marker='o', linestyle='-')
plt.title('Simulated Daily Stock Prices for September 2024')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.grid()
plt.show()


# Resample the data to weekly frequency and calculate the mean
weekly_data = data.resample('W').mean()

print(weekly_data)
```

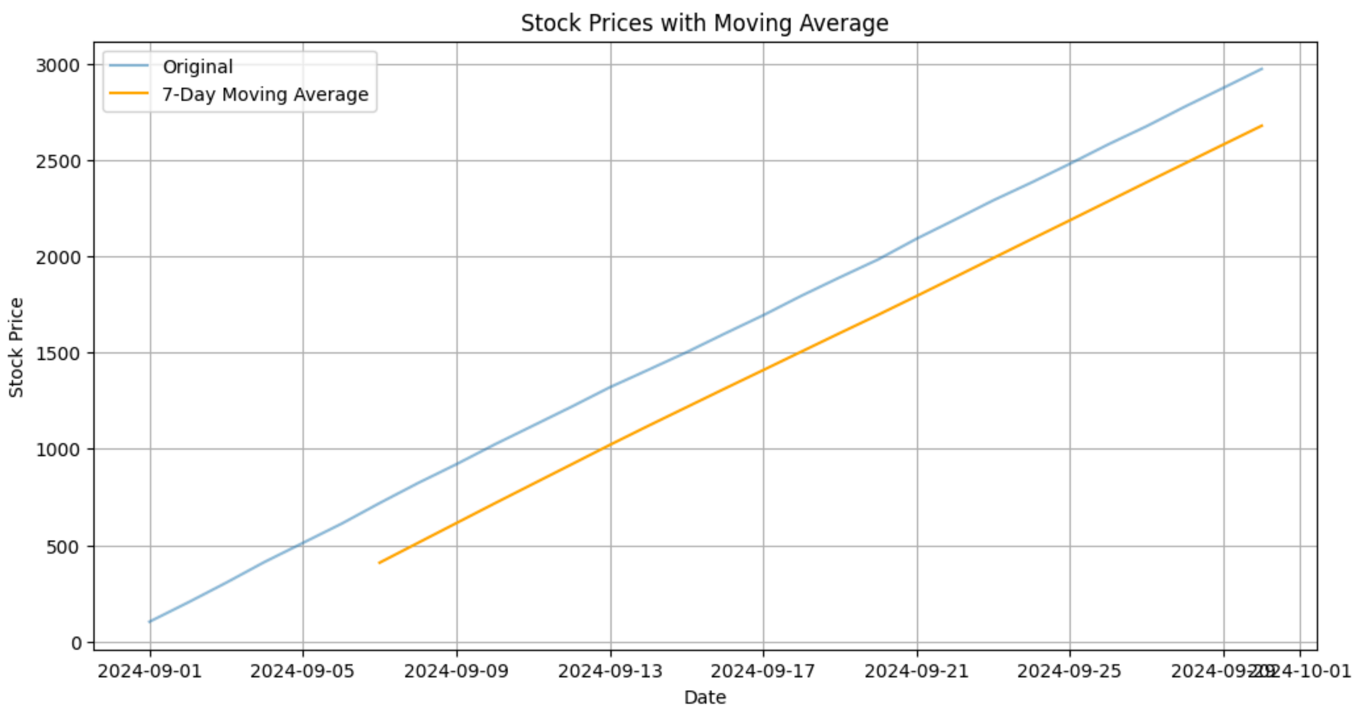


```
            Stock Price
Date
2024-09-01   102.483571
2024-09-08   511.640904
2024-09-15  1215.582942
2024-09-22  1891.058104
2024-09-29  2579.315050
2024-10-06  2971.777966
```

```
#Time Series Decomposition
#You can decompose the time series into trend, seasonality, and residuals using statsmodels

from statsmodels.tsa.seasonal import seasonal_decompose

# Decompose the time series
decomposition = seasonal_decompose(data['Stock Price'], model='additive')

# Plot the decomposition
fig, ax = plt.subplots(4, 1, figsize=(12, 10), sharex=True)
decomposition.observed.plot(ax=ax[0], title='Observed', legend=False)
decomposition.trend.plot(ax=ax[1], title='Trend', legend=False)
decomposition.seasonal.plot(ax=ax[2], title='Seasonal', legend=False)
decomposition.resid.plot(ax=ax[3], title='Residual', legend=False)

plt.xlabel('Date')
plt.tight_layout()
plt.show()
```



```
"""Moving Average
Calculate and visualize a moving average to smooth out the fluctuations."""

# Calculate moving average
data['Moving Average'] = data['Stock Price'].rolling(window=7).mean()

# Plot original and moving average
plt.figure(figsize=(12, 6))
plt.plot(data['Stock Price'], label='Original', alpha=0.5)
plt.plot(data['Moving Average'], label='7-Day Moving Average', color='orange')
plt.title('Stock Prices with Moving Average')
plt.xlabel('Date')
plt.ylabel('Stock Price')
plt.legend()
plt.grid()
plt.show()
```

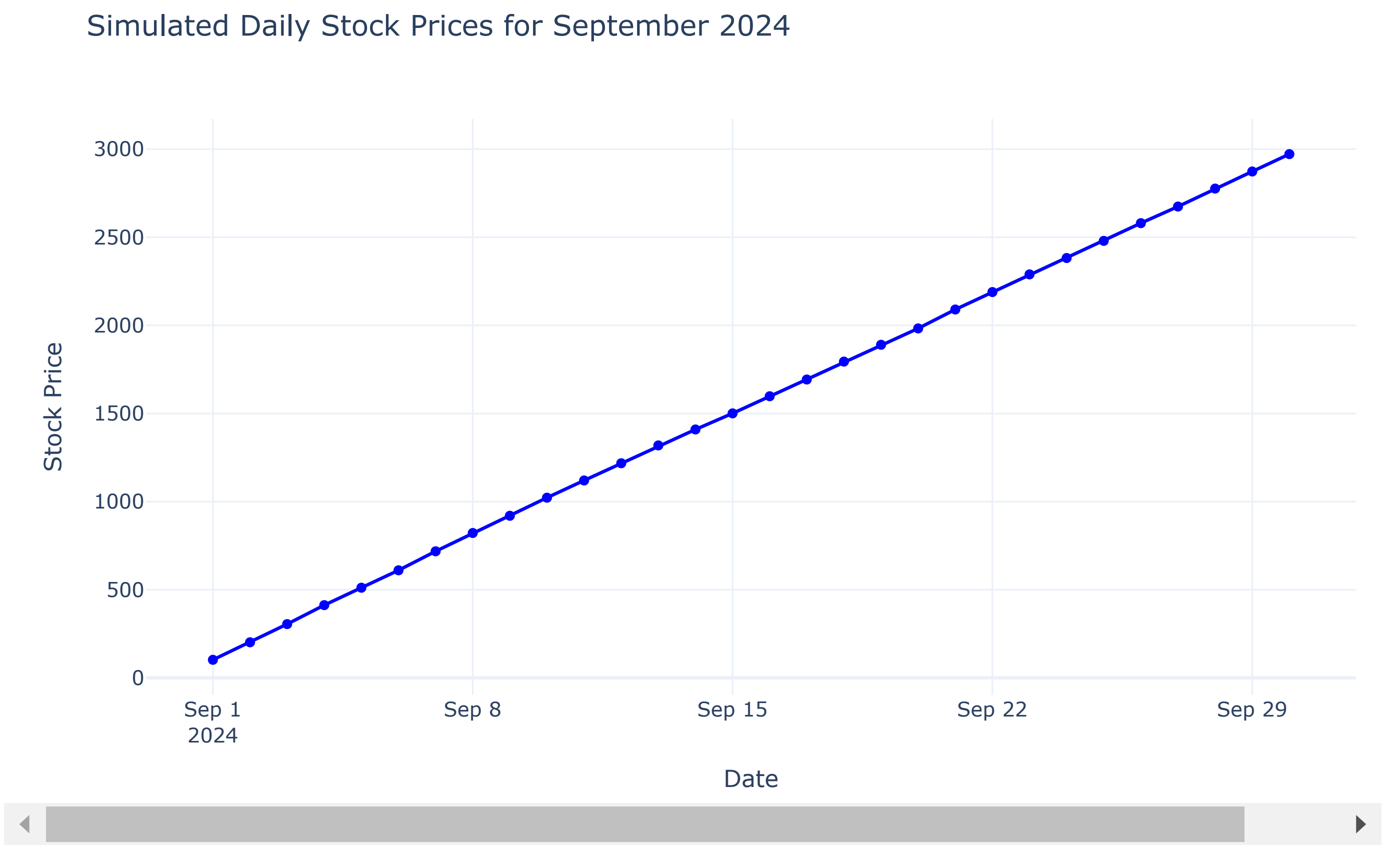


Using Plotly

```
!pip install plotly

import plotly.graph_objects as go
import plotly.express as px
import pandas as pd
import numpy as np

# Generate date range
dates = pd.date_range(start='2024-09-01', end='2024-09-30', freq='D')

# Create demo data: simulating stock prices
np.random.seed(42)  # For reproducibility
prices = np.random.normal(loc=100, scale=5, size=len(dates)).cumsum()

# Create DataFrame
data = pd.DataFrame(data={'Date': dates, 'Stock Price': prices})

# Set Date as the index
data.set_index('Date', inplace=True)
```

```
Requirement already satisfied: plotly in /usr/local/lib/python3.10/dist-packages (5.24.1)
Requirement already satisfied: tenacity>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from plotly) (9.
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from plotly) (24.1)
```

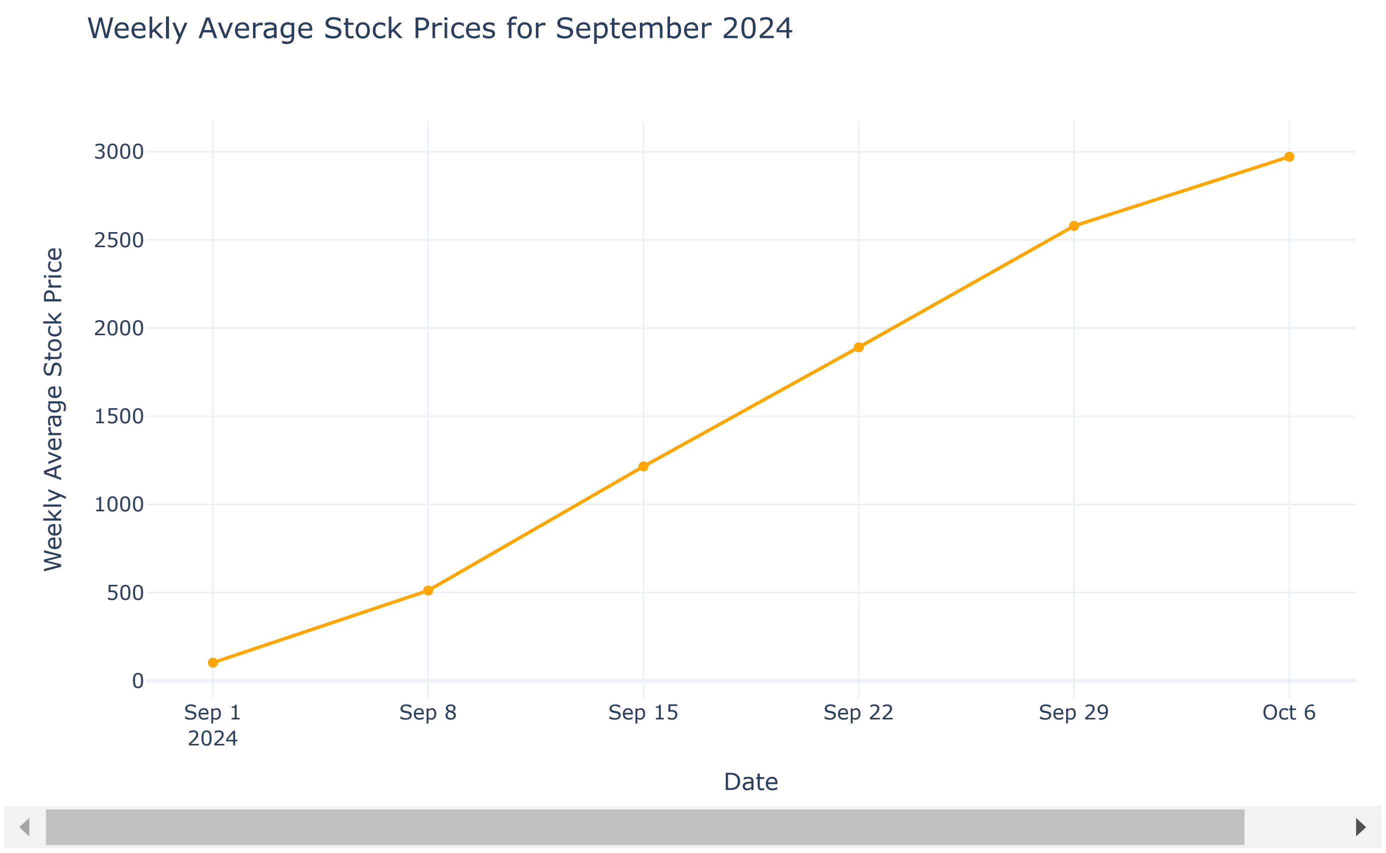
```
# Plotting the time series data with Plotly
fig1 = go.Figure()

fig1.add_trace(go.Scatter(
    x=data.index,
    y=data['Stock Price'],
    mode='lines+markers',
    name='Stock Price',
    marker=dict(color='blue')
))

fig1.update_layout(
    title='Simulated Daily Stock Prices for September 2024',
    xaxis_title='Date',
    yaxis_title='Stock Price',
    template='plotly_white'
)

fig1.show()
```



Step 4: Resample the Data

Plot 2: Weekly Average Stock Prices

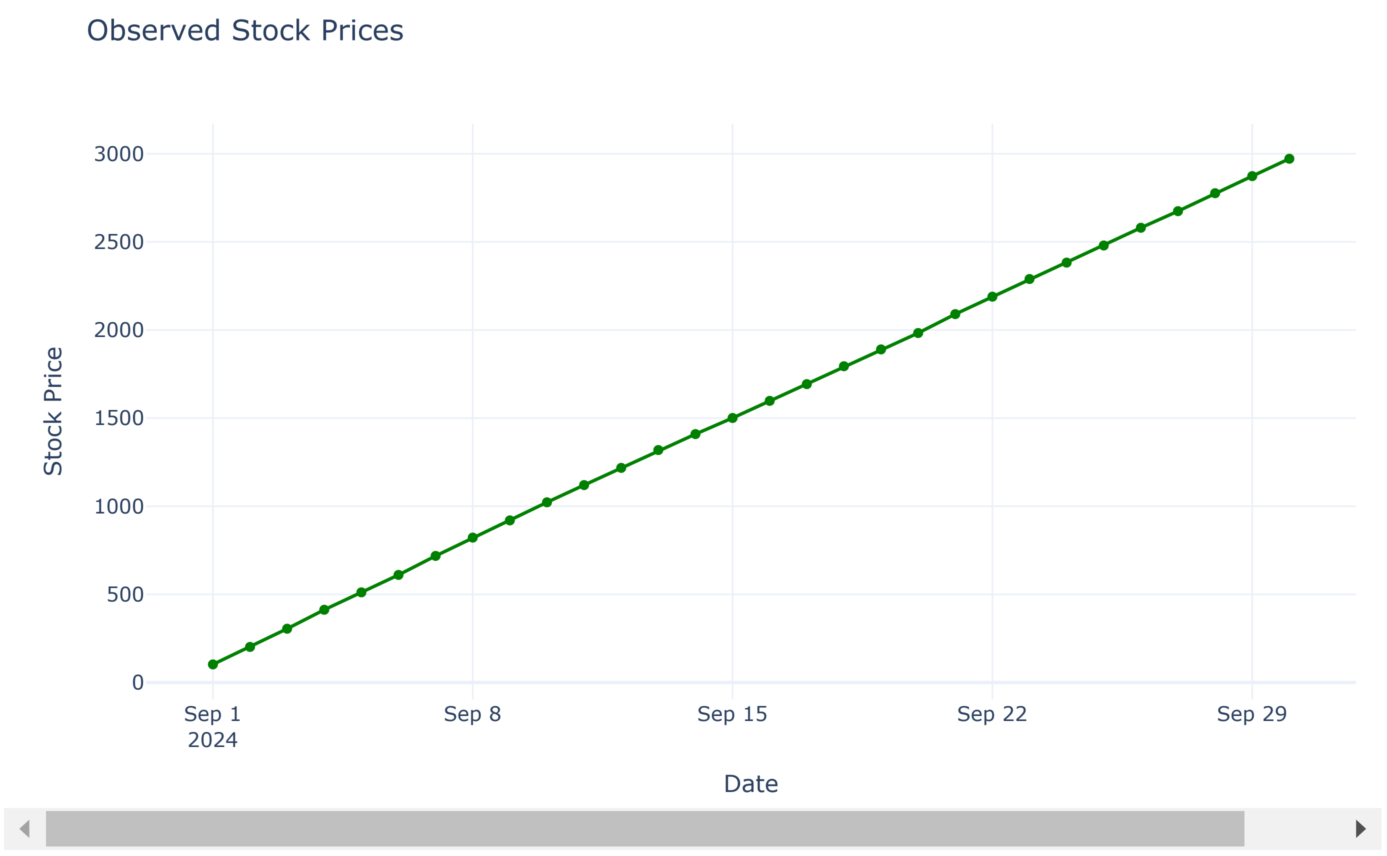
```
# Resample the data to weekly frequency and calculate the mean
weekly_data = data.resample('W').mean()

# Plotting the weekly average stock prices
fig2 = go.Figure()

fig2.add_trace(go.Scatter(
    x=weekly_data.index,
    y=weekly_data['Stock Price'],
    mode='lines+markers',
    name='Weekly Average Stock Price',
    marker=dict(color='orange')
))

fig2.update_layout(
    title='Weekly Average Stock Prices for September 2024',
    xaxis_title='Date',
    yaxis_title='Weekly Average Stock Price',
    template='plotly_white'
)

fig2.show()
```



Step 5: Time Series Decomposition

For decomposition, we will keep it simple. The following example uses a plot for the observed data only (you can decompose using statsmodels if you want).

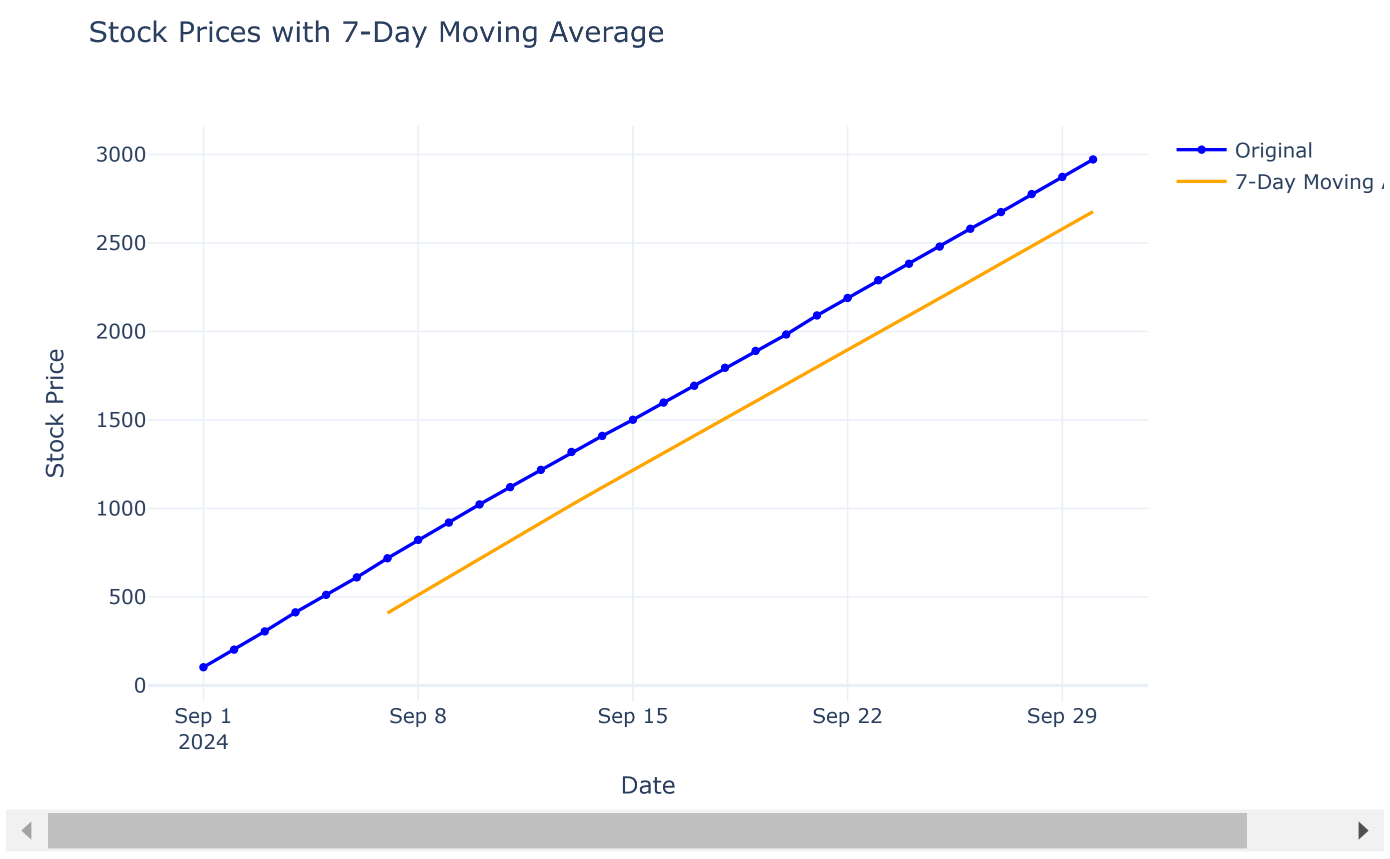
```
# Plotting the observed stock prices for decomposition
fig3 = go.Figure()

fig3.add_trace(go.Scatter(
    x=data.index,
    y=data['Stock Price'],
    mode='lines+markers',
    name='Observed',
    marker=dict(color='green')
))

fig3.update_layout(
    title='Observed Stock Prices',
    xaxis_title='Date',
    yaxis_title='Stock Price',
    template='plotly_white'
)

fig3.show()
```



Step 6: Moving Average

Plot 4: Stock Prices with Moving Average

```
# Calculate moving average
data['Moving Average'] = data['Stock Price'].rolling(window=7).mean()

# Plotting original and moving average
fig4 = go.Figure()

fig4.add_trace(go.Scatter(
    x=data.index,
    y=data['Stock Price'],
    mode='lines+markers',
    name='Original',
    line=dict(color='blue', width=2),
    marker=dict(size=5)
))

fig4.add_trace(go.Scatter(
    x=data.index,
    y=data['Moving Average'],
    mode='lines',
    name='7-Day Moving Average',
    line=dict(color='orange', width=2)
))

fig4.update_layout(
    title='Stock Prices with 7-Day Moving Average',
    xaxis_title='Date',
    yaxis_title='Stock Price',
    template='plotly_white'
)

fig4.show()
```



Resampling and rolling window calculations

Resampling and rolling window calculations are essential techniques in time series analysis used to manipulate and analyze data over different time intervals. Below is a detailed explanation of both concepts:

1. Resampling

Resampling involves changing the frequency of your time series data, either by increasing (upsampling) or decreasing (downsampling) the frequency of the observations. This is particularly useful when you want to analyze data at different time intervals, such as aggregating daily data to monthly data or interpolating to fill in gaps for higher frequency data.

Key Operations:

Downsampling: Reducing the frequency of the data, such as converting daily data into weekly or monthly data. You typically use aggregation functions like mean, sum, or median.

Example of downsampling: Daily to monthly average

monthly_data = daily_data.resample('M').mean()

Upsampling: Increasing the frequency of the data, such as converting monthly data into daily data. When upsampling, interpolation methods like forward fill or backward fill are often applied to estimate missing values.

Example of upsampling: Monthly to daily

daily_data = monthly_data.resample('D').ffill() # Forward fill

```
import pandas as pd
import numpy as np

# Generate sample daily data
dates = pd.date_range('2024-01-01', periods=10)
data = pd.Series(np.random.randint(1, 10, size=10), index=dates)

# Downsample to a weekly frequency
weekly_data = data.resample('W').sum()  # Sum for each week
```

Rolling Window Calculations

Rolling window calculations involve applying a function to a fixed-size window of data points over a time series. This technique allows you to compute statistics over a moving period, making it useful for smoothing data or analyzing trends over time.

Key Operations:

Moving Average: A common rolling window operation where you compute the average over the last n observations. This helps in smoothing the data to identify trends.

Example of calculating a moving average

moving_avg = data.rolling(window=3).mean() # 3-day moving average

Other Functions: You can apply other functions like sum, min, max, or standard deviation over a rolling window.

rolling_sum = data.rolling(window=3).sum() # 3-day rolling sum

```
# Using the same sample daily data
# Calculate a rolling mean with a window size of 3
rolling_mean = data.rolling(window=3).mean()
```

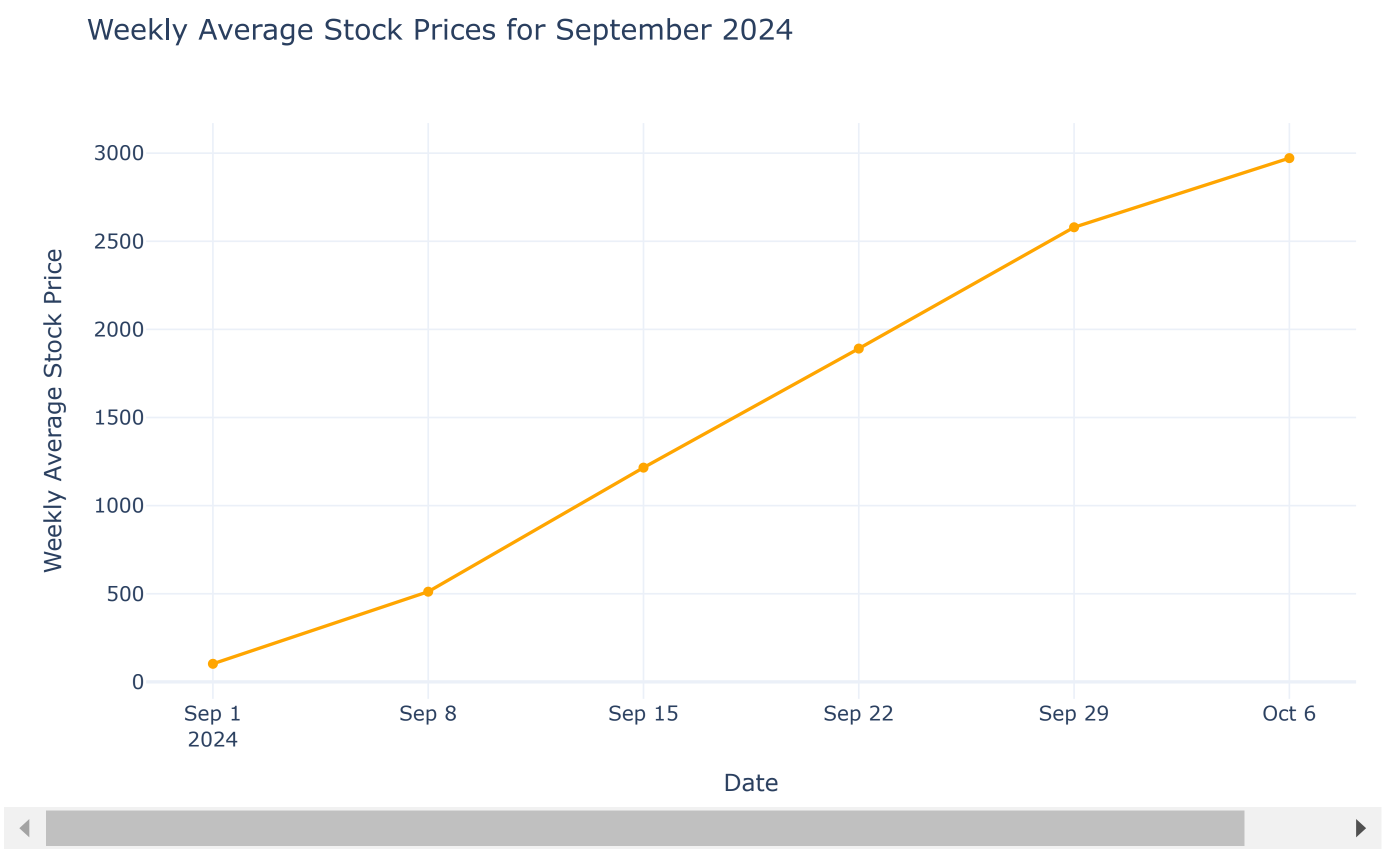
Resampling is useful for changing the frequency of your data, allowing for analysis at different time scales. It can involve aggregating or interpolating values.

Rolling window calculations allow you to compute statistics over a specified number of observations, helping to identify trends and smooth out fluctuations in time series data.

```
import pandas as pd
import numpy as np
import plotly.graph_objects as go



# Generate date range
dates = pd.date_range(start='2024-09-01', end='2024-09-30', freq='D')

# Create demo data: simulating stock prices
np.random.seed(42)  # For reproducibility
prices = np.random.normal(loc=100, scale=5, size=len(dates)).cumsum()

# Create DataFrame
data = pd.DataFrame(data={'Date': dates, 'Stock Price': prices})

# Set Date as the index
data.set_index('Date', inplace=True)



# Resample to weekly frequency and calculate the mean
weekly_data = data.resample('W').mean()

print("Weekly Average Stock Prices:")
print(weekly_data)
```

```
Weekly Average Stock Prices:
            Stock Price
Date
2024-09-01   102.483571
2024-09-08   511.640904
2024-09-15  1215.582942
2024-09-22  1891.058104
2024-09-29  2579.315050
2024-10-06  2971.777966
```

```
# Plotting the weekly average stock prices
fig_weekly = go.Figure()

fig_weekly.add_trace(go.Scatter(
    x=weekly_data.index,
    y=weekly_data['Stock Price'],
    mode='lines+markers',
    name='Weekly Average Stock Price',
    marker=dict(color='orange')
))

fig_weekly.update_layout(
    title='Weekly Average Stock Prices for September 2024',
    xaxis_title='Date',
    yaxis_title='Weekly Average Stock Price',
    template='plotly_white'
)

fig_weekly.show()
```



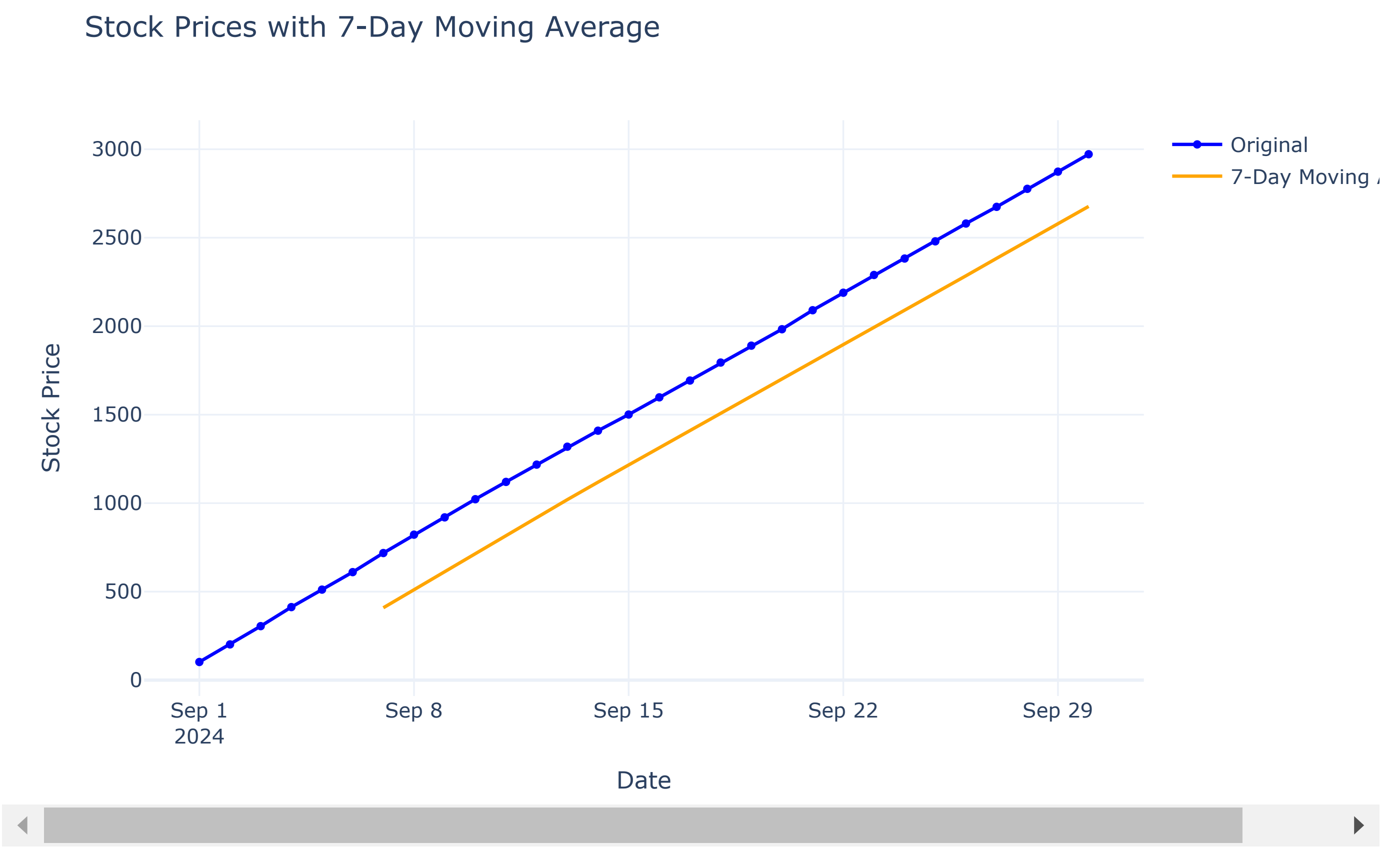
Rolling Window Calculations

Rolling window calculations allow you to compute statistics (e.g., mean, sum) over a specified window of data points.

```
# Calculate 7-day moving average
data['7-Day Moving Average'] = data['Stock Price'].rolling(window=7).mean()

print("Data with 7-Day Moving Average:")
print(data.tail(10))  # Show last 10 entries
```

```
Data with 7-Day Moving Average:
            Stock Price  7-Day Moving Average
Date
2024-09-21  2090.198388           1792.730481
2024-09-22  2189.069506           1891.058104
2024-09-23  2289.407147           1989.835597
2024-09-24  2382.283406           2088.318862
2024-09-25  2479.561493           2186.188821
2024-09-26  2580.116106           2284.786599
2024-09-27  2674.361138           2383.571026
2024-09-28  2776.239628           2481.576918
2024-09-29  2873.236434           2579.315050
2024-09-30  2971.777966           2676.796596
```

```
# Plotting original and moving average
fig_moving_avg = go.Figure()

fig_moving_avg.add_trace(go.Scatter(
    x=data.index,
    y=data['Stock Price'],
    mode='lines+markers',
    name='Original',
    line=dict(color='blue', width=2),
    marker=dict(size=5)
))

fig_moving_avg.add_trace(go.Scatter(
    x=data.index,
    y=data['7-Day Moving Average'],
    mode='lines',
    name='7-Day Moving Average',
    line=dict(color='orange', width=2)
))

fig_moving_avg.update_layout(
    title='Stock Prices with 7-Day Moving Average',
    xaxis_title='Date',
    yaxis_title='Stock Price',
    template='plotly_white'
)

fig_moving_avg.show()
```



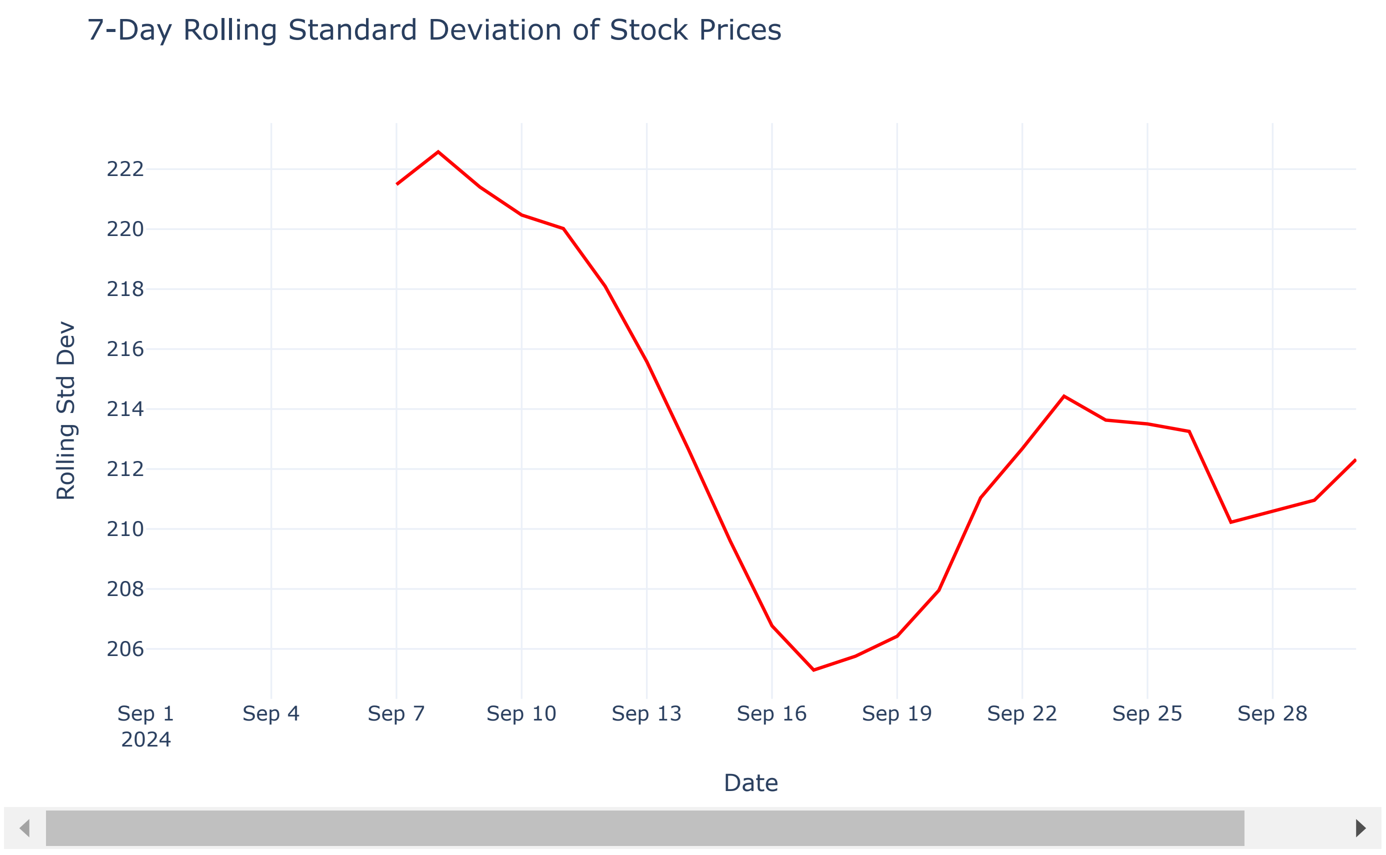
Other Rolling Calculations

You can also calculate other statistics like the rolling standard deviation, sum, or custom functions.

```
# Calculate 7-day rolling standard deviation
data['7-Day Rolling Std'] = data['Stock Price'].rolling(window=7).std()

print("Data with 7-Day Rolling Standard Deviation:")
print(data.tail(10))  # Show last 10 entries
```

```
Data with 7-Day Rolling Standard Deviation:
            Stock Price  7-Day Moving Average  7-Day Rolling Std
Date
2024-09-21  2090.198388           1792.730481         211.040443
2024-09-22  2189.069506           1891.058104         212.680294
2024-09-23  2289.407147           1989.835597         214.428198
2024-09-24  2382.283406           2088.318862         213.630872
2024-09-25  2479.561493           2186.188821         213.504053
2024-09-26  2580.116106           2284.786599         213.254009
2024-09-27  2674.361138           2383.571026         210.226016
2024-09-28  2776.239628           2481.576918         210.576554
2024-09-29  2873.236434           2579.315050         210.959041
2024-09-30  2971.777966           2676.796596         212.322196
```

```
# Plotting the rolling standard deviation
fig_rolling_std = go.Figure()

fig_rolling_std.add_trace(go.Scatter(
    x=data.index,
    y=data['7-Day Rolling Std'],
    mode='lines',
    name='7-Day Rolling Std Dev',
    line=dict(color='red', width=2)
))

fig_rolling_std.update_layout(
    title='7-Day Rolling Standard Deviation of Stock Prices',
    xaxis_title='Date',
    yaxis_title='Rolling Std Dev',
    template='plotly_white'
)

fig_rolling_std.show()
```



Time series forecasting techniques

Time series forecasting is a crucial aspect of data analysis that involves predicting future values based on previously observed values. Various techniques can be employed, ranging from simple statistical methods to advanced machine learning algorithms. Below is an overview of some commonly used time series forecasting techniques:

1. Naïve Forecasting

This is the simplest method, where the forecast for the next time period is simply the last observed value.

Use Case: Suitable for short-term forecasting when trends and seasonality are not pronounced.

2. Moving Average

This technique smooths out short-term fluctuations and highlights longer-term trends or cycles by averaging a fixed number of past observations.

Use Case: Effective for time series with no trend or seasonal patterns.

3. Exponential Smoothing

Exponential smoothing methods apply decreasing weights to past observations, with more recent observations receiving more weight. Variants include:

Simple Exponential Smoothing: For time series without trend or seasonality.

Holt’s Linear Trend Model: For time series with a trend.

Holt-Winters Seasonal Model: For time series with both trend and seasonality.

Use Case: Suitable for data with trends or seasonality.

4. Machine Learning Approaches

Various machine learning algorithms can be employed for time series forecasting, such as:

Decision Trees and Random Forests: Good for capturing non-linear relationships. Gradient Boosting Machines (GBM): Effective for complex patterns in the data. Support Vector Machines (SVM): Useful for high-dimensional data.

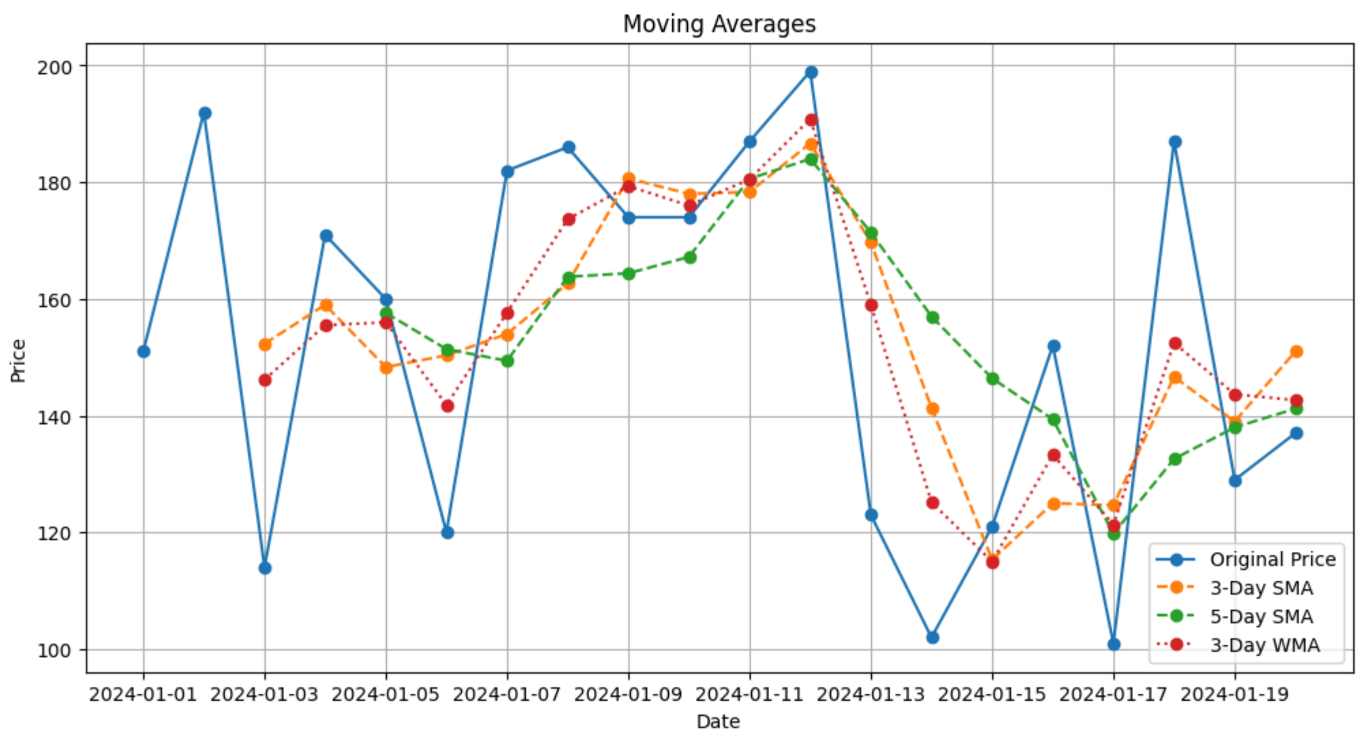
5. Deep Learning Approaches

Advanced neural networks can be applied to time series forecasting, including:

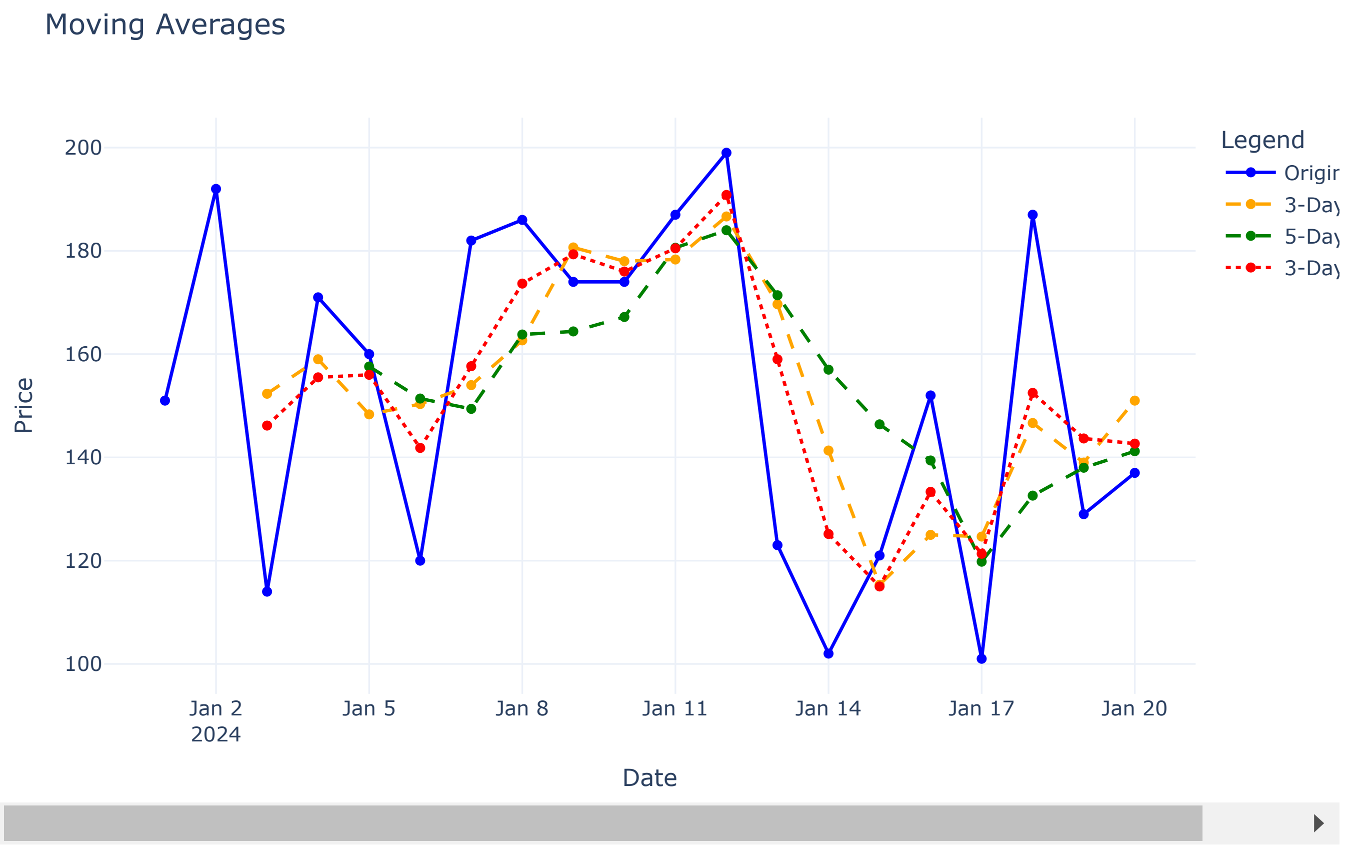
Recurrent Neural Networks (RNN): Designed for sequential data and can remember previous inputs. Long Short-Term Memory (LSTM): A type of RNN that can capture long-term dependencies in time series data. Convolutional Neural Networks (CNN): Can be adapted for time series forecasting.

Moving Average

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

# Step 1: Create Sample Data
dates = pd.date_range(start='2024-01-01', periods=20)
np.random.seed(42)
prices = np.random.randint(100, 200, size=len(dates))

data = pd.DataFrame(data={'Date': dates, 'Price': prices})
data.set_index('Date', inplace=True)

# Step 2: Calculate Moving Averages
data['SMA_3'] = data['Price'].rolling(window=3).mean()  # 3-day SMA
data['SMA_5'] = data['Price'].rolling(window=5).mean()  # 5-day SMA
weights = np.arange(1, 4)  # For a 3-day window
data['WMA_3'] = data['Price'].rolling(window=3).apply(lambda x: np.dot(x, weights)/weights.sum(), raw=True)

# Step 3: Visualize Results
plt.figure(figsize=(12, 6))
plt.plot(data['Price'], label='Original Price', marker='o')
plt.plot(data['SMA_3'], label='3-Day SMA', linestyle='--', marker='o')
plt.plot(data['SMA_5'], label='5-Day SMA', linestyle='--', marker='o')
plt.plot(data['WMA_3'], label='3-Day WMA', linestyle=':', marker='o')

plt.title('Moving Averages')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid()
plt.show()
```



```
import pandas as pd
import numpy as np
import plotly.graph_objects as go

# Step 1: Create Sample Data
dates = pd.date_range(start='2024-01-01', periods=20)
np.random.seed(42)
prices = np.random.randint(100, 200, size=len(dates))

data = pd.DataFrame(data={'Date': dates, 'Price': prices})
data.set_index('Date', inplace=True)

# Step 2: Calculate Moving Averages
data['SMA_3'] = data['Price'].rolling(window=3).mean()  # 3-day SMA
data['SMA_5'] = data['Price'].rolling(window=5).mean()  # 5-day SMA
weights = np.arange(1, 4)  # For a 3-day window
data['WMA_3'] = data['Price'].rolling(window=3).apply(lambda x: np.dot(x, weights)/weights.sum(), raw=True)

# Step 3: Plot with Plotly
fig = go.Figure()

# Add traces for original prices and moving averages
fig.add_trace(go.Scatter(x=data.index, y=data['Price'], mode='lines+markers', name='Original Price', line=dict(c
fig.add_trace(go.Scatter(x=data.index, y=data['SMA_3'], mode='lines+markers', name='3-Day SMA', line=dict(dash='
fig.add_trace(go.Scatter(x=data.index, y=data['SMA_5'], mode='lines+markers', name='5-Day SMA', line=dict(dash='
fig.add_trace(go.Scatter(x=data.index, y=data['WMA_3'], mode='lines+markers', name='3-Day WMA', line=dict(dash='

# Update layout
fig.update_layout(
    title='Moving Averages',
    xaxis_title='Date',
    yaxis_title='Price',
    legend_title='Legend',
    template='plotly_white'
`
)

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



Pyarrow

PyArrow is a Python library that provides an interface for working with the Apache Arrow framework, which is designed for high-performance data processing and analytics. Apache Arrow provides a standardized columnar memory format that allows for efficient data interchange between different systems and languages. PyArrow enables users to leverage these capabilities in Python.

Here's an overview of the key features and functionalities of PyArrow, along with some code examples:

Key Features

**Columnar Memory Format:* *PyArrow allows you to work with data in a columnar format, which can lead to significant performance improvements for analytical workloads.

Interoperability: It supports interoperability with various data formats like Parquet, Feather, and Arrow IPC (Inter-Process Communication), making it easier to read from and write to these formats.

Integration with Pandas: PyArrow can be seamlessly integrated with Pandas, allowing users to convert between Pandas DataFrames and Arrow Tables easily.

Zero-Copy Reads: PyArrow enables zero-copy reads for fast data access without the overhead of serialization and deserialization.

File Formats: It supports reading and writing different file formats like CSV, Parquet, and Feather.

Compute Functions: PyArrow includes various compute functions for performing operations on Arrow Arrays and Tables

```
pip install pyarrow
```

```
Requirement already satisfied: pyarrow in /usr/local/lib/python3.10/dist-packages (16.1.0)
Requirement already satisfied: numpy>=1.16.6 in /usr/local/lib/python3.10/dist-packages (from pyarrow) (1.2
```

```
import pyarrow as pa

# Create an Arrow array
data = pa.array([1, 2, 3, 4, 5])
print(data)

# Create an Arrow table
table = pa.table({'column1': data, 'column2': pa.array(['A', 'B', 'C', 'D', 'E'])})
print(table)
```

```
[
  1,
  2,
  3,
  4,
  5
]
pyarrow.Table
column1: int64
column2: string
----
column1: [[1,2,3,4,5]]
column2: [["A","B","C","D","E"]]
```

```
import pyarrow.parquet as pq

# Create a sample Arrow table
data = {
    'column1': pa.array([1, 2, 3]),
    'column2': pa.array(['A', 'B', 'C'])
}
table = pa.table(data)

# Write the table to a Parquet file
pq.write_table(table, 'example.parquet')

# Read the table back from the Parquet file
read_table = pq.read_table('example.parquet')
print(read_table)
```

```
pyarrow.Table
column1: int64
column2: string
----
column1: [[1,2,3]]
column2: [["A","B","C"]]
```

```
import pandas as pd
```

```
# Create a sample Pandas DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': ['X', 'Y', 'Z']
})

# Convert Pandas DataFrame to Arrow Table
arrow_table = pa.Table.from_pandas(df)
print(arrow_table)

# Convert Arrow Table back to Pandas DataFrame
df_from_arrow = arrow_table.to_pandas()
print(df_from_arrow)
```

```
pyarrow.Table
A: int64
B: string
----
A: [[1,2,3]]
B: [["X","Y","Z"]]
   A  B
0  1  X
1  2  Y
2  3  Z
```

```
# Create an Arrow array
data = pa.array([1, 2, 3, 4, 5])

# Convert to NumPy array without copying data
numpy_array = data.to_numpy(zero_copy_only=True)
print(numpy_array)
```

```
[1 2 3 4 5]
```

Best Practices for Maintainable and Efficient Code

Writing clean, optimized code

1. Follow Naming Conventions

Use meaningful names for variables, functions, and classes. Follow naming conventions such as snake_case for variables and functions, and CamelCase for classes.

```
def calculate_area(radius):
    return 3.14 * radius ** 2
```

2.Keep Functions Small and Focused

Each function should perform a single task and be as short as possible. Use descriptive names to indicate what the function does.

```
def fetch_data(source):
    # Fetch data from the specified source
    pass

def process_data(data):
    # Process the fetched data
    pass

def save_data(data, destination):
    # Save the processed data to the destination
    pass
```

3. Avoid Code Duplication

Use functions or classes to encapsulate repeated logic. Refactor duplicated code into reusable components

```
# Bad: Duplicated code
def calculate_total_price(item_price, tax):
    total = item_price + (item_price * tax)
    return total

def calculate_total_discounted_price(item_price, tax, discount):
    total = (item_price - discount) + ((item_price - discount) * tax)
    return total

# Good: Refactored
def calculate_total(price, tax, discount=0):
    total = (price - discount) + ((price - discount) * tax)
    return total
```

4. Use Comments and Docstrings

Use comments to explain complex logic and decisions.

Use docstrings to document functions and classes.

```
def calculate_fibonacci(n):
    """Calculate the nth Fibonacci number."""
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return calculate_fibonacci(n-1) + calculate_fibonacci(n-2)
```

5. Use List Comprehensions and Generator Expressions

Prefer list comprehensions for concise and readable code when generating lists. Use generator expressions for memory-efficient iteration.

```
# List comprehension
squares = [x**2 for x in range(10)]

# Generator expression
squares_gen = (x**2 for x in range(10))
```

6. Optimize for Performance

Use built-in functions and libraries that are optimized for performance. Avoid premature optimization but be aware of performance implications.

```
# Bad: Using a loop to calculate sum
total = 0
for num in range(1, 1000000):
    total += num

# Good: Using built-in sum function
total = sum(range(1, 1000000))
```

7.Handle Exceptions Gracefully

Use try-except blocks to handle exceptions and provide meaningful error messages. Avoid catching general exceptions without handling them.

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("Error: Division by zero is not allowed.", e)
```

```
import tkinter as tk
```