Week 3

Python for Data Analytics

Mastering NumPy

- Performance Optimization
- Array Operations
- Data Filtering
- Data Aggregation
- Practical Lab

NumPy Arrays vs Python Lists

While Python lists are versatile, NumPy arrays are specifically designed for efficient numerical computation, providing significant performance advantages.

Homogeneous Data Types

Every element must be of the same type (e.g., int64 or float32), eliminating type-checking overhead.

Contiguous Memory

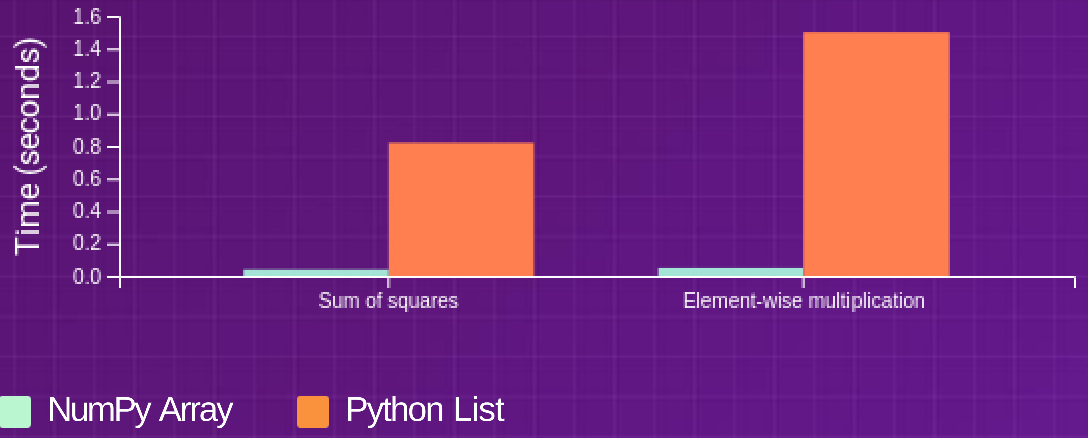
Stored as a single block, allowing processors to access data with maximum efficiency.

Vectorized Operations

Operations on entire arrays at once, delegating looping to optimized C code.

Performance Comparison

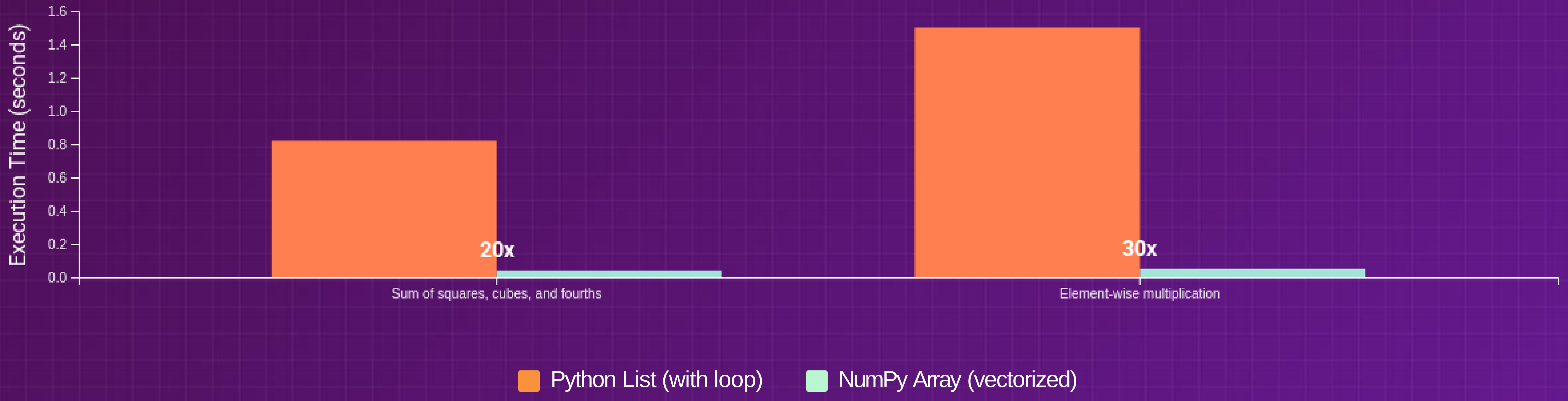
For an array of 1,000,000 elements:



NumPy can be **20-30x faster** for large datasets.

Performance Benchmark

Quantitative comparison between Python lists and NumPy arrays for operations on 1,000,000 elements:



Execution Time Comparison

NumPy operations can be **20-30x faster** than equivalent Python loops.

Performance Gap

The performance gap widens as dataset size and computational complexity increase.

Sum of squares: 20x faster

Element-wise multiplication: 30x faster

Creating NumPy Arrays

The most straightforward way to create a NumPy array is by converting existing Python data structures using `np.array()`.

1D Arrays

From a Python list

```python
list_a = [1, 2, 3, 4]
arr_1d = np.array(list_a)
```

[1, 2, 3, 4] → array([1, 2, 3, 4])

With specific data type

```python
arr_float = np.array([1, 2, 3], dtype=np.float32)
```

[1, 2, 3] → array([1., 2., 3.], dtype=float32)

Multi-dimensional Arrays

From nested lists

```python
list_b = [[1, 2, 3], [4, 5, 6]]
arr_2d = np.array(list_b)
```

[
[1, 2, 3],
[4, 5, 6]
]
→ array([[1, 2, 3],
[4, 5, 6]])

Key Benefits

- Preserves the structure of the input data
- Automatically determines data type
- Can be used with any nested sequence structure

Array Creation: arange() and linspace()

np.arange()

Similar to Python's `range()`

Random Array Generation

The numpy.random module is essential for simulations and statistical sampling, providing reproducible random data generation.

Best Practice: Use np.random.default_rng() to create a generator instance for reproducible results.

Creating Random Arrays

Uniform Distribution

```
# Create random float array
rand_uniform = rng.random((2, 3))
```

[0.77, 0.44, 0.86] [0.70, 0.09, 0.98]

Random Integers

```
# Create random integer array
rand_integers = rng.integers(low=1, high=10, size=(2,
3))
```

[8, 2, 7] [8, 2, 5]

Key Concepts

Reproducibility

Provide a seed to the generator for reproducible results:

```
# Create generator with seed
rng = np.random.default_rng(seed=42)
```

Distributions

The generator supports various statistical distributions:

- Uniform
- Normal
- Integer
- Choice

Array Indexing and Slicing

NumPy indexing works similarly to Python lists but extends to multiple dimensions with a more powerful and flexible syntax.

1D Arrays

Basic indexing:

```
import numpy as np
x = np.arange(10) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

0	1	2	3	4	5	6	7	8	9

Slicing:

```
x[1:7:2] # [1, 3, 5]
```

2D Arrays

Multi-dimensional indexing:

```
import numpy as np
y = np.array([[1, 2, 3], [4, 5, 6]])
```

1	2	3	4	5	6

Accessing elements:

```
y[1, 2] # 6
```

NumPy uses a single pair of brackets with comma-separated indices, more efficient than nested lists.

Key Points:

- Zero-based indexing
- Negative indices count from end
- Slices are views, not copies

Advanced Slicing and Boolean Filtering

Multi-dimensional Slicing

For 2D arrays, separate slice objects with commas to select specific rows and columns.

Example: Selecting submatrix

Original matrix:

1	2	3	4
5	6	7	8
9	10	11	12

Slicing operation:

```
matrix[:2, 1:3]
```

Result:

```
[[2, 3],
[6, 7]]
```

Note: Slices are views of the original data.

*Unlike basic slicing, boolean indexing always returns a **copy** of the data.*

Boolean Indexing

Select elements based on conditions. Creates a boolean array and uses it as an index.

Example: Filtering sales data

Sales data:

```
sales = np.array([150, 200, 120, 250, 300, 180, 90])
```

Boolean mask for sales > 190:

```
high_sales_mask = sales > 190
```

Filtered sales:

```
sales[high_sales_mask]
```

→ [200, 250, 300, 180]

Combining conditions:

```
sales[(sales > 100) & (sales < 200)]
```

→ [150, 120, 180]

Vectorized Operations

Vectorized operations enable execution of operations on entire arrays at once, delegating iteration to optimized C code for dramatic performance improvements.

Operation Comparison

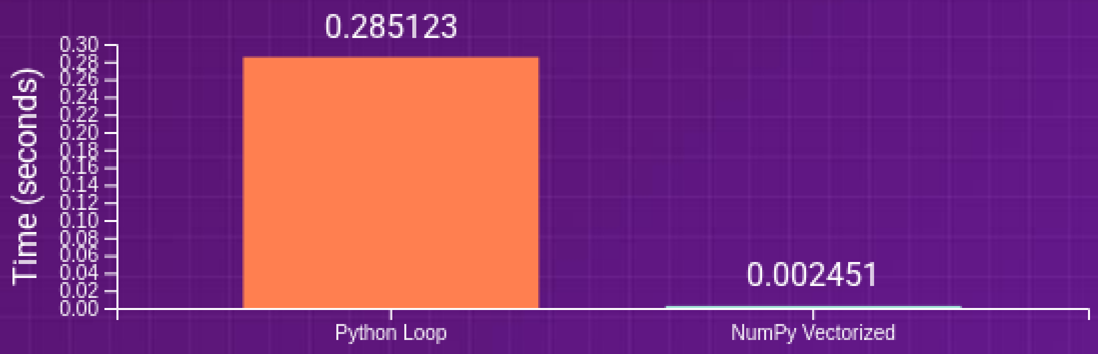
Python Loop

```
import time

large_list = list(range(1000000))
squared_list = []

for x in large_list:
```

NumPy Vectorized

```
import numpy as np
import time

large_array = np.arange(1000000)
squared_array = large_array**2
```

Performance Impact



Key Benefits

- Concise, readable code without explicit loops
- Delegates iteration to optimized C code
- Critical for data analytics with large datasets

Vectorized operation is **116x faster** for large datasets.

Broadcasting

Broadcasting allows NumPy to perform arithmetic operations between arrays of different shapes without copying data.

Broadcasting Rules

- Compares dimensions from right to left
- Dimensions are compatible if equal or one is 1
- Result shape is maximum of input shapes

Eliminates unnecessary memory copies

Examples

Example 1: Scalar + Array

```
import numpy as np

# Scalar broadcasting
a = np.array([1.0, 2.0, 3.0])
b = 5.0

# 'b' broadcasts to shape of 'a'
result = a + b
# Output: [6. 7. 8.]
```

Example 2: 1D + 2D Array

```
import numpy as np

# Array broadcasting
matrix = np.array([[ 0, 0, 0],
[10, 10, 10],
[20, 20, 20]])
vector = np.array([1, 2, 3])

# 'vector' broadcasts across rows
result = matrix + vector
# Output:
# [[ 1 2 3]
# [11 12 13]
# [21 22 23]]
```

Basic Aggregations

NumPy provides fast built-in functions to compute summary statistics on arrays.

np.sum()

Computes the total sum of array elements.

np.mean()

Calculates the average of array elements.

np.min()

Finds the minimum value in the array.

np.max()

Finds the maximum value in the array.

Examples

1D Array

```
daily_sales = np.array([150, 200, 180, 220, 250])
```

Total: 1000

Average: 200.0

Minimum: 150

Maximum: 250

2D Array

```
sales_data = np.array([[50, 60, 55, 65],
    [30, 35, 40, 33],
    [90, 85, 95, 88]])
```

axis=0

[170 180 190 186]

Sum down columns

axis=1

[230 138 358]

Sum across rows

These functions are implemented in compiled code, making them much faster than equivalent Python loops.

Cumulative Operations and Indices

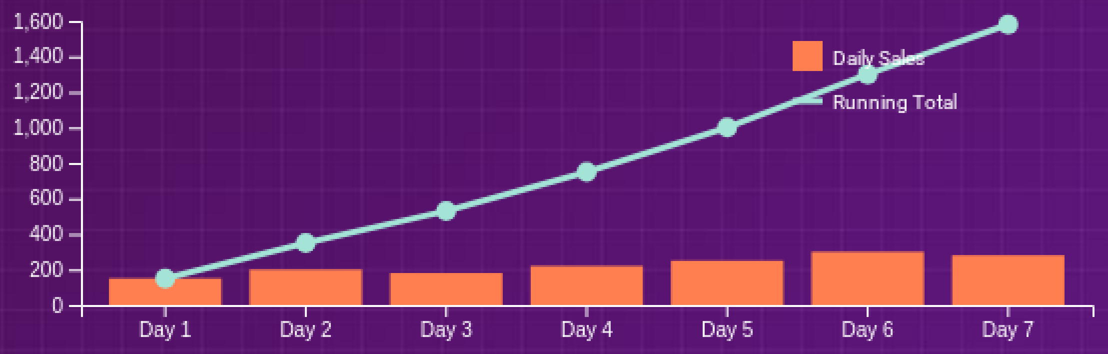
Cumulative Operations

Cumulative operations return an array of intermediate results.

np.cumsum()

Calculates the cumulative sum of elements along a given axis.

```
import numpy as np

daily_sales = np.array([150, 200, 180, 220, 250, 300, 280])
running_total = np.cumsum(daily_sales)
```



Useful for time series analysis and running totals.

argmin() and argmax()

These functions return the **index** of minimum and maximum values.

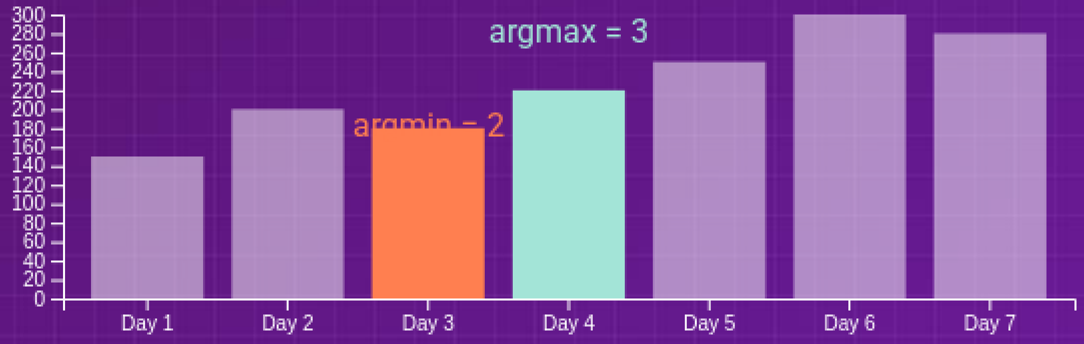
np.argmin()

Returns the index of the minimum element.

np.argmax()

Returns the index of the maximum element.

```
import numpy as np

sales = np.array([180, 220, 150, 250, 210])
best_day_index = np.argmax(sales)
worst_day_index = np.argmin(sales)
```



best_day_index = 3 → sales[3] = 250

worst_day_index = 2 → sales[2] = 150

Reshaping Arrays

The **reshape()** function changes an array's dimensions while preserving its data.

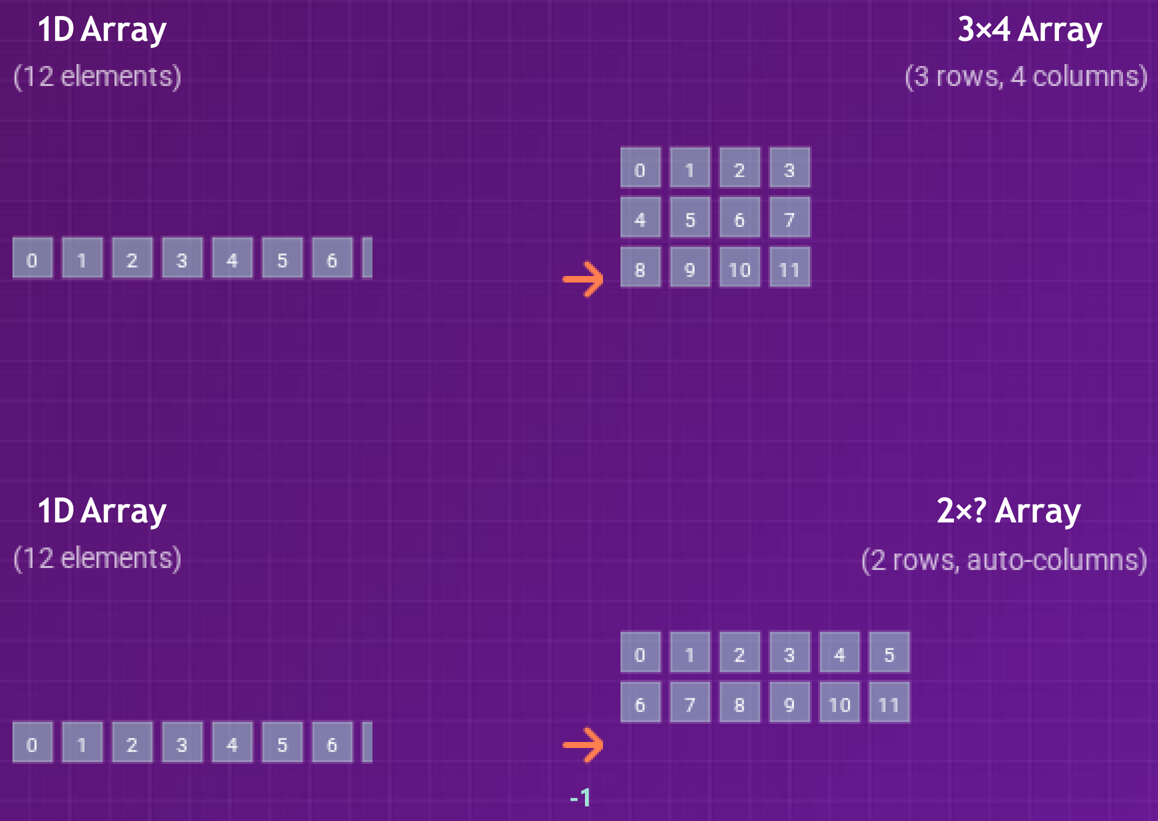
💡 Key Features

- Must maintain the same number of elements
- Use **-1** to let NumPy calculate a dimension

Code Examples:

```
import numpy as np

# 1D array with 12 elements
daily_sales = np.arange(12)

# Reshape to 3x4 array
reshaped = daily_sales.reshape(3, 4)

# Using -1 to auto-calculate
auto_reshaped = daily_sales.reshape(2, -1)
```

Reshape Visualization



NumPy infers the correct size for the dimension with **-1**.

Stacking Arrays

Stacking combines multiple arrays into a single, larger array. vstack() and hstack() are convenient functions for this purpose.

np.vstack()

Stacks arrays **on top of each other** (row-wise).

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} + \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} \rightarrow \begin{bmatrix}1 & 2\\3 & 4\\5 & 6\\7 & 8\end{bmatrix}$$

```
import numpy as np

array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])

v_stacked = np.vstack((array1, array2))
# Result: array([[1, 2], [3, 4], [5, 6], [7, 8]])
```

np.hstack()

Stacks arrays **side by side** (column-wise).

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} + \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} \rightarrow \begin{bmatrix}1 & 2 & 5 & 6\\3 & 4 & 7 & 8\end{bmatrix}$$

```
import numpy as np

array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])

h_stacked = np.hstack((array1, array2))
# Result: array([[1, 2, 5, 6], [3, 4, 7, 8]])
```

Note: vstack() and hstack() are wrappers for np.concatenate(). vstack() ≡ concatenate(..., axis=0) and hstack() ≡ concatenate(..., axis=1)

Lab: Simulating Sales Data

Let's simulate daily sales data for multiple products over a year to practice NumPy array operations.

```
import numpy as np

# Set a seed for reproducibility
np.random.seed(42)

# Parameters
num_days = 364
num_products = 4

# Simulate daily sales units
daily_sales = np.random.randint(
    low=10,
    high=101,
    size=(num_days, num_products)
)

# Display shape and first 5 days
print("Shape:", daily_sales.shape)
print("First 5 days:\\n", daily_sales[:5])
```

Lab: Computing Monthly Averages

Reshaping daily sales data into monthly structure to calculate average daily sales per product per month.

360 × 4 → **12 × 30 × 4** → **12 × 4**

Daily Sales
360 days, 4 products

Reshaped
12 months, 30 days, 4 products

Monthly Averages
12 months, 4 products

Vectorized Implementation

```
import numpy as np

# Parameters
num_days = 360  # 12 months of 30 days
num_products = 4

# Simulate daily sales units
daily_sales = np.random.randint(low=10, high=101, size=(num_days, num_products))

# Reshape into (months, days_per_month, products)
monthly_sales = daily_sales.reshape(12, 30, 4)

# Calculate average daily sales per product per month
monthly_avg_sales = monthly_sales.mean(axis=1)
```

Lab: Performance Comparison

Let's benchmark two approaches to calculate the total sales across all products and days:

Python Loop Approach

```
def sum_with_loop(data):
    total = 0
    for row in data:
        for item in row:
            total += item
    return total

# Timing
%timeit sum_with_loop(daily_sales)
```

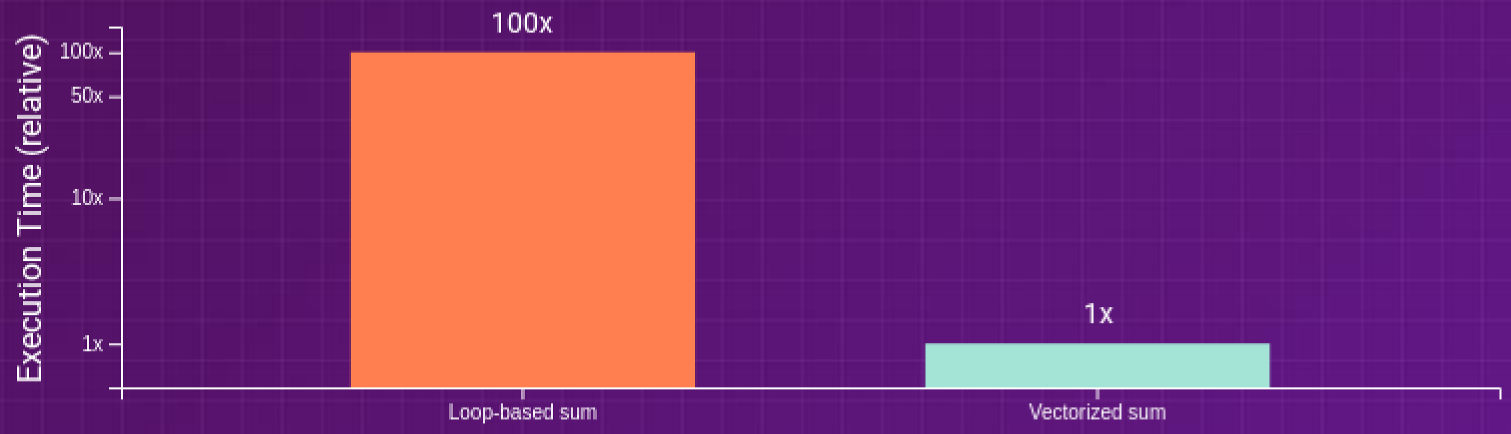
VS

Vectorized np.sum()

```
# Using NumPy's vectorized sum
%timeit np.sum(daily_sales)
```

Delegates iteration to optimized C code

Performance Results



Key Takeaway

Vectorized operations are **50-100x faster**.

Vectorization is essential for data analysis

Lab: Weekly Sales Aggregation

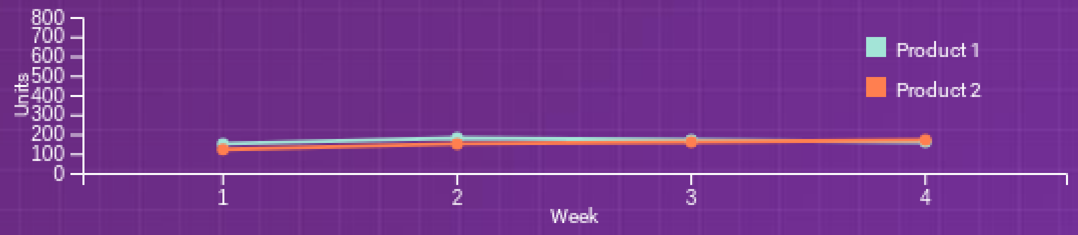
Transform daily sales data into weekly totals using NumPy reshaping and aggregation.

(364, 4) daily_sales → reshape → (52, 7, 4) weekly_sales_data → sum(axis=1) → (52, 4) total_weekly_sales

Code Example

```
# daily_sales shape is (364, 4)
# 364 days = 52 weeks * 7 days

import numpy as np

# Reshape into (weeks, days_per_week, products)
weekly_sales_data = daily_sales.reshape(52, 7, 4)

# Sum across 'days' axis to get weekly totals
total_weekly_sales = weekly_sales_data.sum(axis=1)

print("Shape of total_weekly_sales:", total_weekly_sales.shape)
print("Total sales for the first week:")
print(total_weekly_sales[0])
```

Result



Key Takeaways

- Combined reshaping and aggregation for time-series data
- Efficiently transform daily to weekly format