Lesson 11: NumPy Basics

Data Science with Python – BSc Course

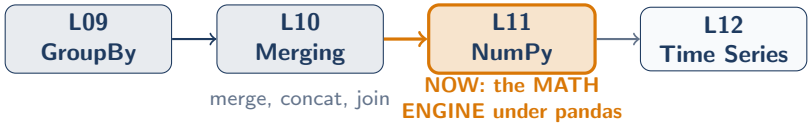
Data Science Program

BSc Course

45 Minutes

Where We Are

Last time: Merging connected data from multiple tables



You can load, clean, group, and merge data. But **pandas is built ON TOP of NumPy**. Understanding NumPy = understanding performance.

Every pandas operation ultimately runs NumPy array math under the hood

Learning Objectives

After this lesson, you will be able to:

- **Create** NumPy arrays and explain why they are faster than lists
- **Apply** vectorized operations instead of Python loops
- **Use** broadcasting for operations on arrays of different shapes
- **Perform** slicing, masking, reshaping, and aggregation
- **Calculate** portfolio returns and risk using NumPy linear algebra

Finance Application: Portfolio math – weights, returns, covariance, Monte Carlo.

NumPy is the foundation of the entire scientific Python ecosystem

Why Not Just Use Python Lists?

pandas is built ON TOP of NumPy. Understanding NumPy = understanding performance.

Python list:

- Each element is a Python object
- Stored at scattered memory addresses
- Loop required for math
- 1M additions: ∼500ms

NumPy array:

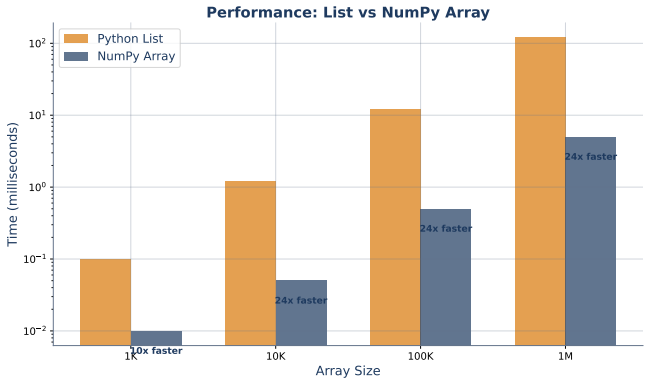
- Homogeneous type (all float64)
- Contiguous memory block
- Vectorized C operations
- 1M additions: ∼5ms

```
import numpy as np
prices = np.array([185.2, 350.1, 140.5])  # 100x faster than list
```

Same data, same result, 100x faster. The secret: contiguous memory + compiled C.

NumPy: same math as lists, but 10–100x faster due to compiled vectorization

NumPy Performance Advantage



NumPy arrays are 10–100x faster than Python lists for numerical operations

Array Features

NumPy Array vs Python List

Feature	Python List	NumPy Array
Element Types	Mixed types	Single type (homogeneous)
Memory	Scattered	Contiguous block
Math Operations	Loop required	Vectorized (fast)
Broadcasting	Not supported	Supported
Size After Creation	Dynamic (mutable)	Fixed

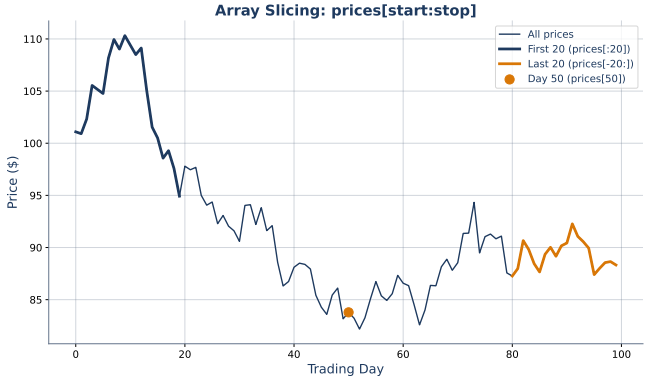
Example: Element-wise multiplication

```
# List (slow):
[x*2 for x in my_list]
```

```
# NumPy (fast):
my_array * 2
```

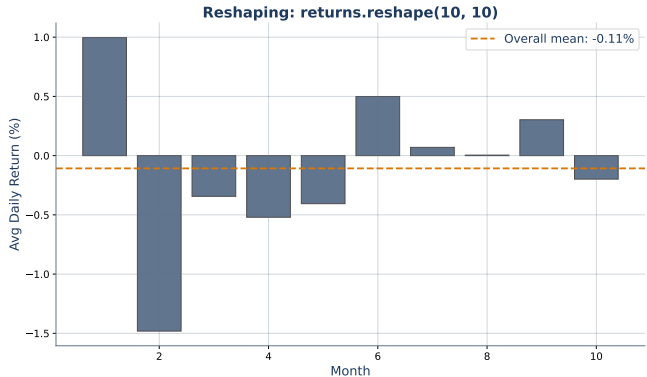
Homogeneous dtype, contiguous memory, vectorized operations

Indexing and Slicing



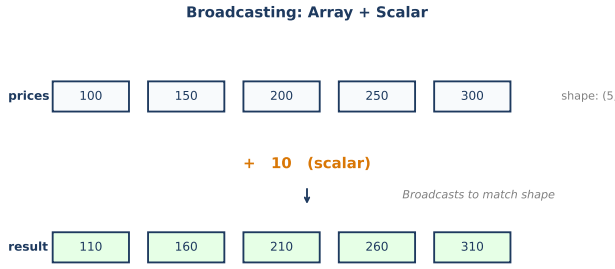
arr[start:stop:step] – same Python slicing syntax, but on arrays

Reshaping Arrays



reshape(), flatten(), transpose() change dimensions without copying data

Broadcasting: Scalar Operations

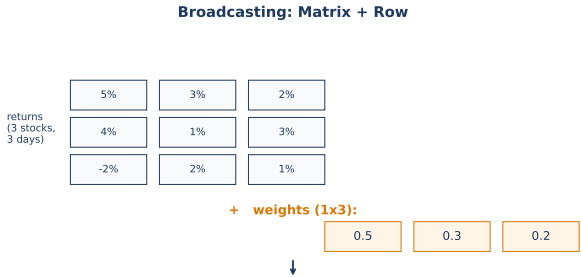


Finance: Add $10 fee to all prices

```
prices + 10  # No loop needed!
```

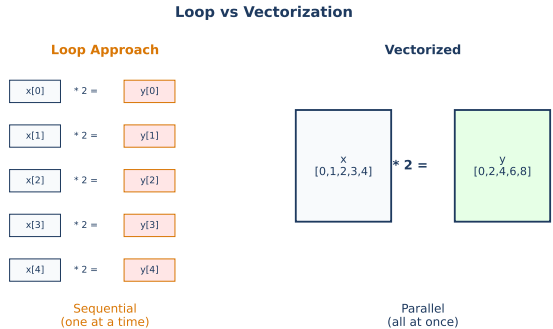
Scalar automatically broadcasts to match array shape – no loop needed

Broadcasting: Matrix Operations



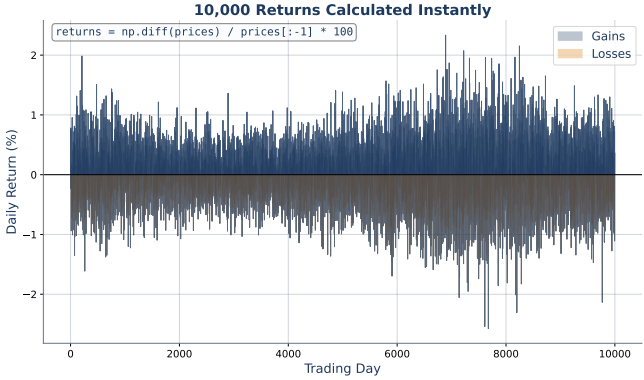
Arrays of different shapes broadcast according to NumPy rules

Vectorized Operations



Replace Python loops with NumPy vectorized operations – always

Vectorized Returns Calculation



Calculate returns for all 500 stocks simultaneously – no loop

Checkpoint: Test Your Understanding

Q1: Why is NumPy 100x faster than Python loops? (Two reasons.)

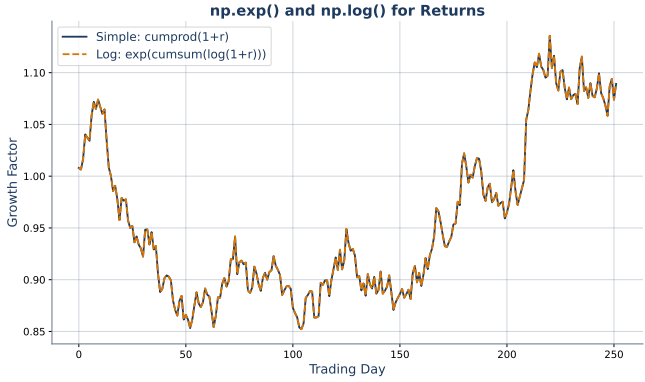
Q2: What does "broadcasting" mean? Give a one-sentence explanation.

Q3: `arr = np.array([10, 20, 30])`. What does `arr * 2 + 1` return?

Think for 30 seconds. Answers: Q1: Contiguous memory + compiled C code. Q2: Automatically stretching a smaller array to match a larger one. Q3: [21, 41, 61].

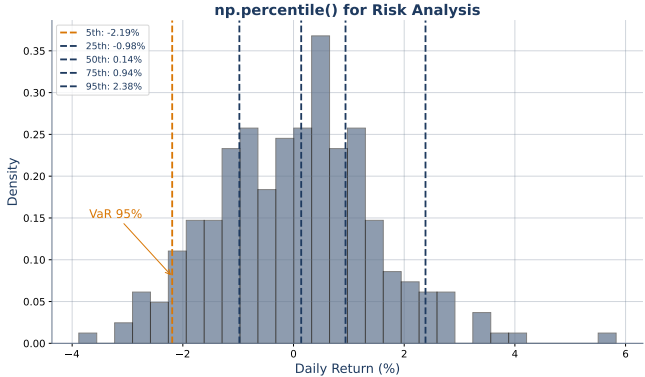
If you understand vectorization and broadcasting, you understand NumPy

Math Functions: exp() and log()



np.exp() for compounding, np.log() for log returns – core finance math

Statistical Functions: Percentiles



np.percentile() for VaR and quantile calculations

Random Number Generation

```
# Uniform [0, 1)
np.random.rand(1000)
# Normal (custom params)
np.random.normal(
    loc=0.0008, scale=0.02,
    size=252)
# Reproducible results
np.random.seed(42)
```

Finance uses:

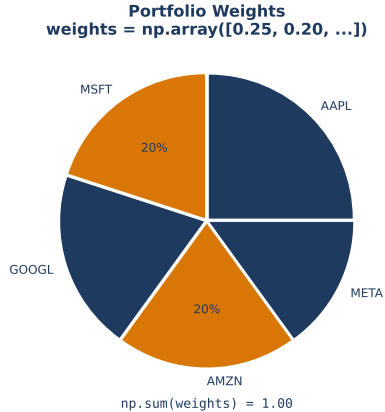
- Simulate daily returns
- Monte Carlo portfolio paths
- Bootstrap confidence intervals

Always set seed for reproducibility.

---

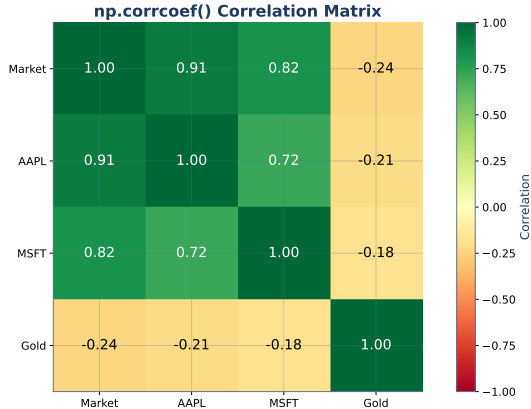
np.random.seed(42) ensures identical results every run

Finance: Portfolio Weights



Portfolio weights must sum to 1.0 – np.sum(weights) == 1.0

Finance: Covariance and Correlation



np.corrcoef() and np.cov() – the building blocks of portfolio theory

Finance: Monte Carlo with NumPy

Simulate 10,000 portfolio outcomes in one line

```
np.random.seed(42)
daily_returns = np.random.normal(
    loc=0.0008, scale=0.02, size=(252, 10000))
# Cumulative wealth paths
paths = np.cumprod(1 + daily_returns, axis=0)
# Final values
final = paths[-1, :]
print(f"Mean final:  {final.mean():.2f}")
print(f"5th pctl:   {np.percentile(final, 5):.2f}")
```

Result: 10,000 simulated year-end portfolio values.
The 5th percentile estimates Value-at-Risk (VaR).

No loops. Pure NumPy. 252 days $\times$ 10,000 paths in milliseconds.

Monte Carlo simulation: generate random scenarios, analyze the distribution

Finance: Portfolio Math with NumPy

Core formulas – all one-liners in NumPy:

Metric	NumPy Code
Portfolio return	`np.dot(weights, returns)`
Portfolio volatility	`np.sqrt(w @ cov @ w)`
Sharpe ratio	`(ret - rf) / vol`
Correlation matrix	`np.corrcoef(returns_matrix)`
Covariance matrix	`np.cov(returns_matrix)`
VaR (5%)	`np.percentile(returns, 5)`
Cumulative returns	`np.cumprod(1 + r) - 1`

Key operators:

- `@` or `np.dot()` – matrix multiplication
- `*` – element-wise multiplication
- `.T` – transpose

NumPy replaces spreadsheet formulas with fast, reproducible array math

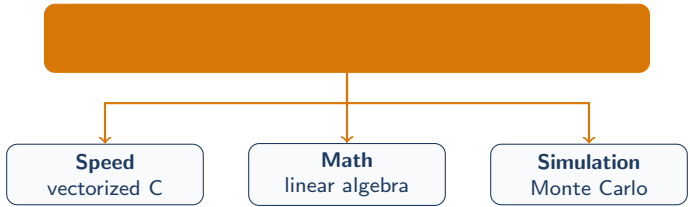
Hands-on Exercise (25 min)

Build a portfolio analyzer with NumPy:

1. Create an array of 5 stock returns: `np.random.normal(0.001, 0.02, (252, 5))`
2. Calculate mean return per stock: `returns.mean(axis=0)`
3. Calculate the covariance matrix: `np.cov(returns.T)`
4. Define equal weights: `w = np.ones(5) / 5`
5. Compute portfolio return: `np.dot(w, mean_returns)`
6. Compute portfolio volatility: `np.sqrt(w @ cov @ w)`
7. Run a 1,000-path Monte Carlo simulation

This is the foundation of Modern Portfolio Theory (Markowitz, 1952)

The Big Idea



pandas handles your tables. NumPy handles the math inside them.

Think of NumPy as the engine and pandas as the dashboard

Key Takeaways

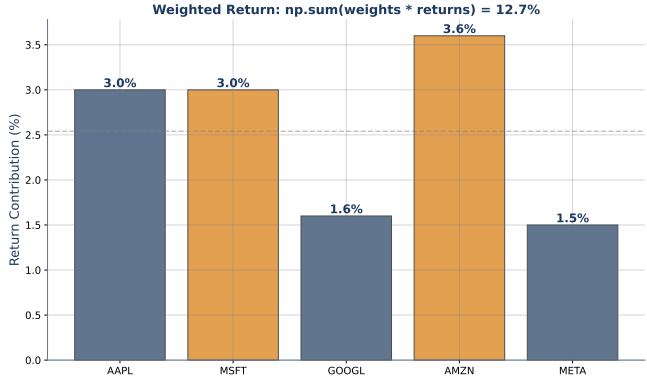
What you learned today:

- NumPy arrays are 10–100x faster than Python lists (contiguous memory + C)
- Vectorized operations replace loops: `arr * 2` not `for x in arr`
- Broadcasting automatically stretches arrays to compatible shapes
- `np.dot()`, `np.cov()`, `np.corrcoef()` power portfolio math
- Monte Carlo simulation generates thousands of scenarios in milliseconds

Next Lesson: L12 – Time Series
Financial data has a **timestamp**. This changes everything – resampling, rolling windows, lags, and seasonality.

NumPy + pandas = the complete toolkit for financial data analysis

Finance: Weighted Portfolio Returns



np.dot(weights, returns) computes portfolio return in one operation