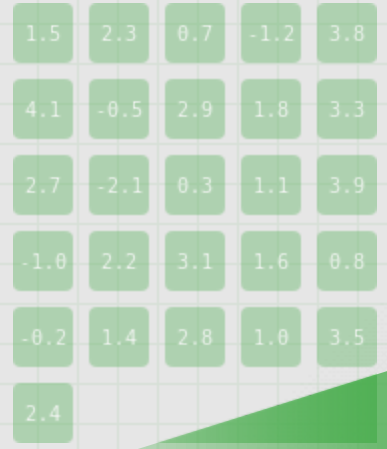
Introduction to NumPy

for Economics & Finance

A comprehensive guide to numerical computing with Python for quantitative analysis in economics and finance



🏛 Economics & Finance Department

Report Date: 2025-09-12

Why NumPy for Econ & Finance?

Foundational Package

NumPy is the cornerstone of numerical computing in Python, providing essential array objects and tools for numerical operations.

Rich Functionality

Built-in mathematical functions (mean, std, dot) simplify complex financial calculations like portfolio returns and correlation matrices.

Processing Efficiency

Optimized C/Fortran backend provides vectorized operations that significantly enhance performance for large financial datasets.

Library Foundation

Serves as the backbone for essential Python libraries like Pandas (data manipulation) and Matplotlib (data visualization).

Memory Optimization

Contiguous memory allocation and homogeneous data types make NumPy arrays more memory-efficient than Python lists.

Key Applications

Critical for financial modeling, econometrics, and data analysis in quantitative finance and applied economics.

NumPy Arrays vs. Python Lists

	NumPy Arrays	Python Lists
Speed	**Highly optimized** for numerical operations via vectorization (C/Fortran backend)	**Slower** for numerical operations due to overhead and lack of vectorization
Memory Usage	**More memory-efficient** due to homogeneous data types and contiguous memory allocation	**Less memory-efficient**; can store heterogeneous data types, leading to scattered memory
Functionality	**Rich set** of built-in mathematical functions (e.g., `np.mean()`	

Creating NumPy Arrays

np.array()

Converts Python lists or tuples into NumPy arrays.

```
# Stock prices
import numpy as np
prices = [150.25, 151.50, 149.80]
np.array(prices)
```

Output: [150.25 151.5 149.8]

np.zeros()

Creates arrays filled with zeros as placeholders.

```
# Daily returns
import numpy as np
np.zeros(5)
```

Output: [0. 0. 0. 0. 0.]

np.ones()

Creates arrays filled with ones for weighting.

```
# Weight matrix
import numpy as np
np.ones((2, 3))
```

Output: [[1. 1. 1.][1. 1. 1.]]

np.arange()

Generates arrays with regularly spaced values.

```
# Time steps
import numpy as np
np.arange(10)
```

Output: [0 1 2 3 4 5 6 7 8 9]

np.linspace()

Creates arrays with evenly spaced values over an interval.

```
# Probability distribution
import numpy as np
np.linspace(0, 1, 5)
```

Output: [0. 0.25 0.5 0.75 1.]

Applications

- Time series analysis
- Portfolio optimization
- Monte Carlo simulations
- Statistical modeling

Array Attributes and Structure

.ndim

The number of dimensions (axes) of the array.

```
# 1D array example
prices_1d = np.array([100, 101, 102])
print(prices_1d.ndim) # Output: 1

# 2D array example
prices_2d = np.array([[100, 101], [200, 201]])
print(prices_2d.ndim) # Output: 2
```

.shape

A tuple indicating the size of the array in each dimension.

```
# Financial data example
financial_data = np.zeros((5, 3))
# 5 time periods for 3 assets
print(financial_data.shape) # Output: (5, 3)
```

.size

The total number of elements in the array.

```
# Continuing from previous example
print(financial_data.size) # Output: 15
# Calculated as 5 * 3 = 15 elements
```

.dtype

The data type of the elements in the array.

```
# Price array example
price_array = np.array([150.25, 151.50, 149.80])
print(price_array.dtype) # Output: float64

# Importance for financial calculations
# - float64 for prices and returns
# - int64 for counts or indices
```

Vectorized Operations

What are Vectorized Operations?

Mathematical operations applied element-wise to entire arrays without explicit Python loops. Vectorization leverages NumPy's C/Fortran backend for enhanced performance.

Benefits for Financial Calculations

- Enhanced performance for large financial datasets
- Simplified complex calculations (e.g., portfolio returns)
- Reduced memory overhead compared to loops

Key Applications

- Calculating daily returns for stock prices
- Portfolio optimization and allocation
- Time series analysis and forecasting

Example: Daily Returns Calculation

```
import numpy as np

# Stock prices for two consecutive days
prices_t0 = np.array([100, 102, 105, 103, 106]) # Prices a
prices_t1 = np.array([102, 103, 104, 105, 107]) # Prices a

# Calculate daily returns using vectorized arithmetic
daily_returns = (prices_t1 - prices_t0) / prices_t0
print("Daily Returns:", daily_returns)
```

Output: [0.02 0.01 0.009524 0.019608 0.009434]

Vectorized vs. Looped Approach

Vectorized (Element-wise operation)

```
result = operation(array1, array2)
```

Single function call handles all elements

Looped (Explicit iteration)

```
for i in range(n):
  result[i] = operation(array1[i], array2[i])
```

Python loop with multiple function calls

Indexing and Slicing

1D Array Indexing

Access specific elements in 1D arrays (time series).

100	101	**102**	100	99	103	105	104	106	107

```
stock_prices[2]  # Access third element (102)
stock_prices[2:6]  # Subset from index 2 to 5
```

Practical Example

Extracting specific time periods:

```
import numpy as np

stock_prices = np.array([100, 101, 102, 100, 99, 103, 105,
104, 106, 107])
# Days 3-5 (index 2-5)
period = stock_prices[2:6]
print(period)  # Output: [102 100 99 103]
```

2D Array Indexing

Access elements in 2D arrays (multi-asset data).

150	2500	300	152	**2510**	302	151	2490	301	153	2520	305	155
2530	307											

```
multi_stock[:, 1]  # All rows, second column (Google)
multi_stock[2, :]  # Third row, all columns
```

Financial Example

Extracting specific asset prices:

```
import numpy as np

multi_stock = np.array([
    [150, 2500, 300],
    [152, 2510, 302],
    [151, 2490, 301],
    [153, 2520, 305],
    [155, 2530, 307]
])
# Extract Google stock prices
google_prices = multi_stock[:, 1]
print(google_prices)
# Output: [2500 2510 2490 2520 2530]
```

Boolean Indexing for Data Filtering

Boolean Indexing Concept

Boolean indexing allows you to select elements from an array based on a condition, creating a powerful tool for data filtering in finance.

- Creates a boolean array (True/False) based on the condition
- Uses the boolean array to index the original array

Example 1: Prices above Threshold

Select stock prices above a certain threshold (e.g., 150):

```
# Example stock prices
prices = np.array([145, 152, 148, 155, 160, 150, 140, 158])

# Select prices above 150
prices_above_150 = prices[prices > 150]
```

Result: `[152 155 160 158]`

Example 2: Negative Returns

Identify days with negative returns:

```
# Example daily returns
daily_returns = np.array([0.01, -0.005, 0.02, -0.015, 0.03, -0.002])

# Identify negative returns
negative_returns_days = daily_returns[daily_returns < 0]
```

Result: `[-0.005 -0.015 -0.002]`

Practical Applications

- Filtering time periods based on market conditions
- Identifying outliers in financial data
- Selecting assets meeting specific criteria
- Analyzing market volatility periods

Statistical Methods for Finance

Key Statistical Functions

np.mean()
Average return calculation

```
np.mean(returns)
```

np.std()
Volatility measurement

```
np.std(prices)
```

np.var()
Data dispersion

```
np.var(returns)
```

np.corrcoef()
Correlation matrix

```
np.corrcoef(assets)
```

Log Returns Calculation

Using np.diff() and np.log()

```
log_returns = np.diff(np.log(prices))
```

Example:

prices: [100, 102, 101, 105]
log_returns: [0.0198, -0.0099, 0.0392]

Correlation Visualization

Application Example

Calculate correlation matrix for portfolio diversification:

```
import numpy as np
assets = np.array([[100, 102, 101], [150, 148, 152]])
correlation_matrix = np.corrcoef(assets)
```

Next Steps in Python for Finance

Summary: NumPy's Importance

NumPy serves as the foundational library for numerical computing in Python, providing powerful array objects and mathematical tools essential for quantitative analysis in economics and finance.

Its efficiency in handling large datasets and performing vectorized operations makes it indispensable for tasks ranging from financial modeling to econometric analysis.

Practice with Real Financial Data

Apply NumPy concepts to analyze real-world financial data, such as historical stock prices from sources like Yahoo Finance.

Explore Pandas

Delve into the Pandas library, which provides high-performance data structures and analysis tools built on top of NumPy. Essential for handling tabular data in finance and economics.

Learn Matplotlib

Familiarize yourself with Matplotlib, a powerful plotting library that allows for effective visualization of financial and economic data when combined with NumPy and Pandas.

Continue your Python journey for finance!

Practice Challenge: NumPy for Finance

Portfolio Analysis Challenge

You are a financial analyst tasked with analyzing a portfolio of three stocks over a 5-day period. Using NumPy, you need to calculate daily returns, identify days with negative returns, and compute key statistics to assess portfolio performance.

Initial Data

```
import numpy as np

# Stock prices for 3 stocks over 5 days (rows: days, columns: stocks)
prices = np.array([
    [150, 2500, 300],  # Day 1
    [152, 2510, 302],  # Day 2
    [151, 2490, 301],  # Day 3
    [153, 2520, 305],  # Day 4
    [155, 2530, 307]   # Day 5
])

# Portfolio weights for the 3 stocks
weights = np.array([0.5, 0.3, 0.2])

# Your code below:
# 1. Calculate daily returns for each stock
# daily_returns =

# 2. Calculate weighted portfolio returns for each day
# portfolio_returns =
```