Notebook 3.1 - NumPy for CEO Analytics

Management Science - Welcome to the CEO Suite!

Introduction

Welcome to your role as CEO of Bean Counter!

Congratulations, Chief Executive Officer!

After your outstanding performance as Operations Director, the board has unanimously appointed you as CEO of Bean Counter! You now lead a coffee empire with:

- 50+ locations across the country
- 1,000+ employees
- Millions of transactions per month
- Thousands of products and suppliers

The Challenge: As CEO, you're drowning in data. Your Python lists take minutes to process sales reports. Board meetings are tomorrow, and you need answers NOW!

Your new Tool: NumPy - a Python library that processes numerical data fast. What takes minutes with regular Python takes seconds with NumPy.

Section 1 - Understanding and Installing Packages

Before we dive into NumPy, let's understand what packages are and how to install them.

What are Packages?

As CEO, you wouldn't build every tool from scratch - you'd use the best tools available. The same applies to Python!

- Packages are collections of pre-written code that solve common problems
- They're created by experts and shared with the community
- Think of them as apps you can add to Python to extend its capabilities

> 💡 Why Use Packages?
>
> Instead of writing thousands of lines of code yourself, you can install a package in seconds and get professional-grade functionality. It's like the difference between building your own espresso machine vs. buying a professional one!

Installing NumPy with uv

To use NumPy (or any package), we first need to install it. We'll use `uv`, which we already used to install Python at the start. Do the following in the terminal where you have all

your notebooks as well as the files that `uv` added during its initialization. Not in the notebook!

```
# Install NumPy for numerical computing
uv add numpy
```

```
# While we're at it, let's also install pandas for next session
uv add pandas
```

> **i** How to Install
>
> 1. Open your terminal (or use the terminal in your IDE)
> 2. Type `uv add numpy pandas` and press Enter
> 3. Wait a few seconds for installation to complete
> 4. That's it! You can now use NumPy in your code

Using Installed Packages

Once installed, you can import and use packages in your Python code:

```
import numpy as np  # Import NumPy with alias 'np' (standard convention)
```

The `as np` part creates a shorthand. Instead of typing `numpy.array()` later, you can type `np.array()` and access numpy functions.

Now let's see why NumPy is essential!

> **⚠** Prerequisites
>
> Make sure you've installed NumPy using `uv add numpy` in your terminal before starting the next section!

Section 3 - Why NumPy? The Speed Advantage

Now let's see why NumPy is so important for large-scale data processing.

```
import numpy as np
import time

# Comparing Python lists vs NumPy for 1 million sales transactions
size = 1000000

# Python list approach (slow)
python_sales = list(range(1, size + 1)) # Creates a list
start = time.time() # Starts a timer
python_result = [x * 1.08 for x in python_sales]  # Add 8% tax
```

```
python_time = time.time() - start # Computes the time difference

# NumPy approach (fast!)
numpy_sales = np.arange(1, size + 1) # Creates a NumPy array
start = time.time() # Starts a timer
numpy_result = numpy_sales * 1.08  # Add 8% tax to ALL at once!
numpy_time = time.time() - start  # Computes the time difference

print(f"Processing {size:,} transactions:")
print(f"Python list: {python_time:.4f} seconds")
print(f"NumPy array: {numpy_time:.4f} seconds")
print(f"NumPy is {python_time/numpy_time:.1f}x faster!")
```

```
Processing 1,000,000 transactions:
Python list: 0.0173 seconds
NumPy array: 0.0010 seconds
NumPy is 17.2x faster!
```

NumPy Arrays vs Python Lists

- Python lists: Flexible but slow for math operations
- NumPy arrays: Specialized for numbers, blazing fast
- Rule of thumb: Use NumPy when doing math on many numbers at once

Section 3 - Working with NumPy Arrays

Let's learn how to create and use NumPy arrays, the foundation of everything in NumPy!

Creating and Using NumPy Arrays

A NumPy array is like a Python list, but optimized for numerical operations. Here's how to work with them:

```
import numpy as np

# Creating arrays from Python lists
prices = [4.50, 3.25, 5.00, 2.75, 4.00]
prices_array = np.array(prices)  # Convert list to NumPy array
print(f"Original list: {prices}")
print(f"NumPy array: {prices_array}")
print(f"Array type: {type(prices_array)}")

# Accessing elements (just like lists!)
print(f"\nFirst price: ${prices_array[0]}")
print(f"Last price: ${prices_array[-1]}")
print(f"Prices 2-4: {prices_array[1:4]}")

# The magic: operations on entire arrays at once!
```

```python
# Increase all prices by 10%
increased_prices = prices_array * 1.10
print(f"\nAfter 10% increase: {increased_prices}")

# Add $0.50 service charge to all
with_service = prices_array + 0.50
print(f"With service charge: {with_service}")

# Calculate total revenue if we sell 100 of each
quantities = np.array([100, 150, 80, 200, 120])
revenues = prices_array * quantities  # Element-wise multiplication!
print(f"\nRevenues: {revenues}")
print(f"Total revenue: ${np.sum(revenues)}")
```

```
Original list: [4.5, 3.25, 5.0, 2.75, 4.0]
NumPy array: [4.5  3.25 5.   2.75 4.  ]
Array type: <class 'numpy.ndarray'>

First price: $4.5
Last price: $4.0
Prices 2-4: [3.25 5.   2.75]

After 10% increase: [4.95  3.575 5.5   3.025 4.4  ]
With service charge: [5.   3.75 5.5  3.25 4.5 ]

Revenues: [450.  487.5 400.  550.  480. ]
Total revenue: $2367.5
```

> **Key NumPy Array Operations**
>
> - Create: `np.array([1, 2, 3])` - Convert list to array
> - Math: `array * 2`, `array + 5` - Operations apply to ALL elements
> - Access: `array[0]`, `array[1:3]` - Works like lists
> - Aggregate: `np.sum(array)`, `np.mean(array)` - Quick statistics

Exercise 3.1 - Your First Analysis

Create a NumPy array of this week's daily revenues and calculate the total.

```python
import numpy as np

# Daily revenues for the week (in thousands)
daily_revenues = [125.5, 132.8, 118.9, 145.2, 155.7, 189.3, 176.4]

# YOUR CODE BELOW
# Convert to NumPy array
revenues_array =
```

```
# Calculate total weekly revenue
total_revenue =
```

```
# Test your revenue calculation
assert isinstance(revenues_array, np.ndarray), "Should be a NumPy array"
assert int(total_revenue) == int(1043.8), f"Total should be 1043.8, got
{total_revenue}"
print(f"Weekly revenue: ${total_revenue:.1f}k")
print("Excellent! You've made your first CEO-level analysis with NumPy!")
```

Section 4 - Creating Arrays for Business Data

As CEO of Bean Counter, you need different ways to create and initialize data arrays.

```
import numpy as np

# Different ways to create arrays for business scenarios

# From a list - actual data
quarterly_sales = np.array([1250000, 1380000, 1420000, 1510000])

# Zeros - initialize budget tracking
budget_tracking = np.zeros(12)  # 12 months, all start at 0

# Ones - initialize satisfaction scores
initial_ratings = np.ones(10) * 4.5  # 10 stores, all start at 4.5

# Range - sequential IDs or time periods
store_ids = np.arange(1, 51)  # Store IDs from 1 to 50
months = np.arange(1, 13)  # Months 1-12

print(f"Q1 Sales: ${quarterly_sales[0]:,}")
print(f"Number of stores: {len(store_ids)}")
print(f"Initial ratings: {initial_ratings[:3]}...")  # First 3 stores
```

```
Q1 Sales: $1,250,000
Number of stores: 50
Initial ratings: [4.5 4.5 4.5]...
```

Exercise 4.1 - Initialize Company Metrics

As CEO, set up arrays for tracking various company metrics.

```
import numpy as np

# YOUR CODE BELOW
# 1. Create an array of 50 zeros for tracking store profits
store_profits =
```

```
# 2. Create an array with store IDs from 101 to 150 (50 stores)
store_codes =

# 3. Create an array of 12 months, each starting with budget of 100000
monthly_budgets =
```

```
# Test your arrays
assert store_profits.shape == (50,), "Should have 50 stores"
assert store_codes[0] == 101 and store_codes[-1] == 150, "Store codes
should be 101-150"
assert np.sum(monthly_budgets[:3]) == 300000, "Q1 budget should be 300000"
print(f"Profit tracking shape: {store_profits.shape}")
print(f"First 5 store codes: {store_codes[:5]}")
print(f"Q1 budget total: ${np.sum(monthly_budgets[:3]):,}")
print("Perfect! Your metric tracking system is initialized!")
```

Section 5 - Vectorized Operations for Mass Updates

As CEO, you need to update prices, apply taxes, and calculate profits across lots of products instantly.

```
import numpy as np

# Prices for 10 products
original_prices = np.random.uniform(2.50, 25.00, 10)
print(f"Sample original prices: {original_prices[:5].round(2)}")

# CEO Decision: 15% price increase across the board
new_prices = original_prices * 1.15

# Add 8% sales tax to all prices
prices_with_tax = new_prices * 1.08

# Calculate revenue if we sell 50 of each product
quantities = np.ones(10) * 50
revenues = prices_with_tax * quantities

print(f"\nAfter 15% increase + tax: {prices_with_tax[:5].round(2)}")
print(f"Total potential revenue: ${revenues.sum():,.2f}")
```

```
Sample original prices: [24.19 15.34  5.97 14.04 23.57]

After 15% increase + tax: [30.05 19.05  7.42 17.44 29.28]
Total potential revenue: $9,060.10
```

Random Number Generation

Numpy allows us, for example, to generate uniform random numbers between two values. The syntax is `np.random.uniform(a, b, c)`, where `a` and `b` are the lower and upper bounds of the range, respectively while `c` is the number of random numbers to generate.

No Loops Needed!

With NumPy, avoid writing loops for mathematical operations:

```
# Don't do this:
for i in range(len(prices)):
    prices[i] = prices[i] * 1.15

# Do this instead:
prices = prices * 1.15
```

Exercise 5.1 - Company-Wide Financial Calculations

Perform mass calculations across all Bean Counter stores.

```
import numpy as np

# Monthly data for 50 stores
revenues = np.array([125000, 98000, 145000, 87000, 156000, 134000, 92000,
167000,
                     118000, 143000, 99000, 175000, 132000, 89000, 154000,
121000,
                     138000, 95000, 162000, 108000, 147000, 131000, 88000,
159000,
                     126000, 141000, 93000, 168000, 115000, 152000, 128000,
86000,
                     144000, 119000, 137000, 96000, 171000, 113000, 149000,
124000,
                     135000, 91000, 164000, 107000, 146000, 129000, 85000,
158000,
                     122000, 140000])

# Cost is 65% of revenue for each store
costs = revenues * 0.65

# YOUR CODE BELOW
# 1. Calculate profit for each store (revenue - costs)
profits =

# 2. Calculate profit margin for each store (profit / revenue * 100)
profit_margins =
```

```
# 3. Apply 25% corporate tax to get after-tax profit
profits_after_tax =

print(f"Total monthly profit (before tax): ${profits.sum():,.2f}")
print(f"Total monthly profit (after tax): ${profits_after_tax.sum():,.2f}")
```

```
# Test your calculations
assert np.isclose(profits.sum(), 2240700), "Total profit before tax should
be 2,240,700"
assert np.isclose(profit_margins.mean(), 35.0), "Average margin should be
35%"
assert np.isclose(profits_after_tax.sum(), 1680525), "After-tax profit
should be 1,680,525"
print(f"Total monthly profit (before tax): ${profits.sum():,.2f}")
print(f"Total monthly profit (after tax): ${profits_after_tax.sum():,.2f}")
print("Fantastic! You've learned company-wide financial calculations!")
```

Section 6 - Statistical Analysis for CEO Insights

CEOs need quick statistical insights to make strategic decisions.

```
import numpy as np

# Customer satisfaction scores from 1000 surveys
np.random.seed(42)  # For reproducible results
satisfaction_scores = np.random.normal(4.2, 0.5, 1000)  # Mean 4.2, std
0.5, normal distribution
satisfaction_scores = np.clip(satisfaction_scores, 1, 5)  # Keep between
1-5

print(f"Survey Analysis (n=1000):")
print(f"Average satisfaction: {satisfaction_scores.mean():.2f}")
print(f"Standard deviation: {satisfaction_scores.std():.2f}")
print(f"Lowest score: {satisfaction_scores.min():.2f}")
print(f"Highest score: {satisfaction_scores.max():.2f}")
print(f"Median score: {np.median(satisfaction_scores):.2f}")

# How many customers gave 4+ stars?
highly_satisfied = np.sum(satisfaction_scores >= 4.0)
print(f"\nHighly satisfied (4+): {highly_satisfied}
({highly_satisfied/10:.1f}%)")
```

```
Survey Analysis (n=1000):
Average satisfaction: 4.20
Standard deviation: 0.46
Lowest score: 2.58
Highest score: 5.00
Median score: 4.21

Highly satisfied (4+): 649 (64.9%)
```

> **Note**
>
> Note, how we can use the methods provided by NumPy to calculate statistics on the satisfaction scores array. These methods are efficient and concise, making our code more readable and maintainable.

2D Arrays

So far, we've worked with 1D arrays (like a single row or column). As CEO, you often need 2D arrays. Think of them as tables with rows and columns!

```
import numpy as np

# Example: Sales data for 5 stores over 7 days
# Rows = stores, Columns = days
sales_table = np.array([
    [125, 132, 128, 145, 155, 189, 176],  # Store 1
    [98, 102, 95, 108, 115, 142, 138],    # Store 2
    [156, 162, 159, 171, 178, 198, 192],  # Store 3
    [87, 91, 88, 95, 102, 125, 118],      # Store 4
    [134, 139, 136, 148, 153, 178, 165]   # Store 5
])

print("Sales Table (5 stores × 7 days):")
print(sales_table)
print(f"\nShape: {sales_table.shape} (rows, columns)")

# Calculate statistics along different axes
total_per_store = np.sum(sales_table, axis=1)  # Sum across columns (days)
for each store
total_per_day = np.sum(sales_table, axis=0)    # Sum across rows (stores)
for each day

print(f"\nTotal sales per store: {total_per_store}")
print(f"Total sales per day: {total_per_day}")
```

```
Sales Table (5 stores × 7 days):
[[125 132 128 145 155 189 176]
 [ 98 102  95 108 115 142 138]
 [156 162 159 171 178 198 192]
 [ 87  91  88  95 102 125 118]
 [134 139 136 148 153 178 165]]

Shape: (5, 7) (rows, columns)

Total sales per store: [1050  798 1216  706 1053]
Total sales per day: [600 626 606 667 703 832 789]
```

Understanding axis Parameter

In 2D arrays:

- `axis=0` operates DOWN the rows (along columns)
- `axis=1` operates ACROSS the columns (along rows)

Think of it this way:

- `axis=1` gives you one value per row (e.g., average per store)
- `axis=0` gives you one value per column (e.g., average per day)

Boolean Filtering and Binary Vectors

An important concept: when you filter with a condition, NumPy creates a boolean (True/False) array, also called a binary vector!

```
import numpy as np

# Sample satisfaction scores
scores = np.array([4.8, 3.2, 4.5, 2.8, 4.9, 3.7, 4.2, 5.0])

# When we apply a condition, we get a boolean array (binary vector)
high_scores_mask = scores >= 4.0
print(f"Original scores: {scores}")
print(f"Boolean mask (>= 4.0): {high_scores_mask}")
print(f"Type: {type(high_scores_mask)}")

# We can use this binary vector in several ways:

# 1. Count True values (treating True=1, False=0)
count_high = np.sum(high_scores_mask)
print(f"\nNumber of high scores: {count_high}")

# 2. Filter to get only values that are True
filtered_scores = scores[high_scores_mask]
print(f"High scores only: {filtered_scores}")

# 3. Do it all in one line (common pattern)
count_directly = np.sum(scores >= 4.0)
print(f"Count directly: {count_directly}")
```

```
Original scores: [4.8 3.2 4.5 2.8 4.9 3.7 4.2 5. ]
Boolean mask (>= 4.0): [ True False  True False  True False  True  True]
Type: <class 'numpy.ndarray'>

Number of high scores: 5
High scores only: [4.8 4.5 4.9 4.2 5. ]
Count directly: 5
```

! Boolean Arrays (Binary Vectors)

When you write `array >= value`, NumPy creates a boolean array: - True (=1) where condition is met - False (=0) where condition is not met

This binary vector can be used to: - Count: `np.sum(condition)` - sums up the 1s and 0s - Filter: `array[condition]` - returns only True values - Analyze: Check what percentage meets criteria

Let's see another practical example:

```
import numpy as np

# Daily sales for 10 stores
daily_sales = np.array([125, 98, 156, 87, 134, 145, 92, 167, 118, 143])

# Find stores exceeding target of 120
target = 120
exceeds_target = daily_sales > target

print(f"Sales: {daily_sales}")
print(f"Exceeds {target}: {exceeds_target}")
print(f"\nStores meeting target: {np.sum(exceeds_target)}")
print(f"Percentage meeting target: {np.sum(exceeds_target) /
len(daily_sales) * 100:.1f}%")
print(f"Actual sales above target: {daily_sales[exceeds_target]}")
```

```
Sales: [125  98 156  87 134 145  92 167 118 143]
Exceeds 120: [ True False  True False  True  True False  True False  True]

Stores meeting target: 6
Percentage meeting target: 60.0%
Actual sales above target: [125 156 134 145 167 143]
```

Exercise 6.1 - Analyze Company Performance

Analyze performance metrics across all Bean Counter locations using 2D arrays.

```
import numpy as np

# Daily customer counts for 50 stores over 30 days
# This creates a 2D array: rows = stores, columns = days
np.random.seed(100)
daily_customers = np.random.randint(150, 500, size=(50, 30))  # 50 stores,
30 days

print(f"Data shape: {daily_customers.shape}")
print(f"First store's first 5 days: {daily_customers[0, :5]}")

# YOUR CODE BELOW
```

```
# 1. Calculate total customers served across all stores in the month
total_customers =

# 2. Calculate average daily customers per store (across all stores and
days)
avg_daily_per_store =

# 3. Find the busiest single day (max customers in one store on one day)
busiest_day =

# 4. Find stores that averaged over 350 customers per day
# Hint: Use np.mean(daily_customers, axis=1) to get average per store
# Then count how many stores have average > 350
store_averages =
high_traffic_stores =
```

```
# Test your analysis
assert total_customers == 492171, f"Total should be 492,171 , got
{total_customers}"
assert np.isclose(avg_daily_per_store, 328.1, atol=0.1), f"Average should
be ~328.1"
assert busiest_day == 499, f"Busiest day should be 499, got {busiest_day}"
print(f"Total customers served: {total_customers:,}")
print(f"Average daily customers per store: {avg_daily_per_store:.1f}")
print(f"Busiest single day: {busiest_day} customers")
print(f"High-traffic stores (>350/day): {high_traffic_stores}")
print("Excellent CEO-level analysis! You understand your company's traffic
patterns!")
```

Conclusion

Congratulations! You've learned NumPy for analytics!

You've learned:

- Array Creation - Initialize data structures for company-wide metrics
- Vectorized Operations - Update thousands of prices/costs instantly
- Statistical Analysis - Get insights from massive datasets in milliseconds
- Speed Advantage - Process millions of data points in seconds

Your Bean Counter CEO toolkit now includes:

- Lightning-fast analysis of millions of transactions
- Company-wide financial calculations in seconds
- Statistical insights for board presentations
- The ability to handle big data that would crash Excel

Remember:

- NumPy arrays are specialized for numerical operations
- Vectorized operations eliminate the need for loops

- Use `np.mean()`, `np.sum()`, `np.std()` for quick statistics
- Random functions help simulate business scenarios
- Always consider using NumPy when dealing with large numerical datasets

What's Next: In the next tutorial, you'll learn Pandas, the ultimate tool for working with structured business data. You'll import real sales data, filter it, group it, and uncover insights that will transform Bean Counter's strategy!

Solutions

You will likely find solutions to most exercises online. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Bibliography