Python

Numpy

Alish Bista - 11/11/2025

NumPy

NumPy

Numerical Python

- Numerical Python library
- Created in 2005
- Foundation of data science ecosystem
- Powers Pandas underneath
- Essential for analytics & ML
- Industry standard for numerical computing

```
import numpy as np
import pandas as pd


# Pandas DataFrames use
NumPy arrays internally
df =
pd.DataFrame({'sales':
[100, 200, 300]})
numpy_array = df.values  #
This is NumPy!
```

Lists Limitations

How Lists Struggles With Numeric Data

Problem with Lists:

- Slow for mathematical operations & requires loops for calculations
- Not optimized for numerical work & mixed types can cause inefficiency

How NumPy Helps

- Vectorized operations (no loops!), 10-100x faster
- Optimized for numbers
- Clean, readable code

```
# Python lists
prices = [10, 20, 30, 40]
new_prices = [price + 5 for price in prices]
doubled = [price * 2 for price in prices]

# NumPy arrays
import numpy as np
prices = np.array([10, 20, 30, 40])
new_prices = prices + 5  # That's it!
doubled = prices * 2     # Magic!
print(new_prices)  # [15, 25, 35, 45]
```

Array

Building the Numerical Foundation

From Existing Data:

- Convert lists to arrays
- Create 1D or 2D structures
- Perfect for real datasets

Quick Generators:

- Zeros and ones
- Number ranges
- Evenly spaced values
- Great for testing & simulation

```
import numpy as np

# From lists
sales = np.array([100, 150, 200, 250])
table = np.array([[1, 2, 3],
                  [4, 5, 6]])

# Quick creation
zeros = np.zeros(5)            # [0. 0. 0. 0. 0.]
ones = np.ones(3)              # [1. 1. 1.]
sequence = np.arange(0, 10, 2)  # [0 2 4 6 8]
spaced = np.linspace(0, 1, 5)   # 5 numbers from 0 to
1

# Random data for testing
random_sales = np.random.randint(50, 200, size=10)
```

Array Anatomy

Dissecting The Array

Key Properties:

- **Shape**: Dimensions of your array
- **ndim**: Number of dimensions
- **size**: Total element count
- **dtype**: Data type stored

```
import numpy as np

# Create a sales table (3 stores, 4 weeks)
sales = np.array([[100, 150, 200, 180],
                  [120, 180, 210, 190],
                  [110, 160, 190, 175]])

# Inspect the array
print(sales.shape)    # (3, 4) - 3 rows, 4 columns
print(sales.ndim)     # 2 - two dimensions
print(sales.size)     # 12 - total elements
print(sales.dtype)    # int64 - integer type

# Access specific info
num_stores = sales.shape[0]    # 3
num_weeks = sales.shape[1]     # 4
```

Element Access

Finding Things in Arrays

1D Array Indexing:

- Like Python lists
- Zero-based counting
- Negative indices work backwards
- Slicing creates subarrays

2D Array Indexing:

- [row, column] notation
- Get entire rows or columns
- Slice both dimensions
- Essential for data extraction

```
# 1D arrays
sales = np.array([100, 150, 200, 250, 300])
print(sales[0])      # 100 - first
print(sales[-1])     # 300 - last
print(sales[1:4])    # [150, 200, 250] - slice

# 2D arrays (rows, columns)
data = np.array([[100, 150, 200],
                 [120, 180, 210],
                 [110, 160, 190]])

print(data[0, 1])     # 150 - row 0, col 1
print(data[1])        # [120, 180, 210] - entire row
print(data[:, 0])     # [100, 120, 110] - entire column
print(data[0:2, 1:3]) # Slice both dimensions
```

Math Operations

Quick Math

Vectorized Operations:

- Apply to entire array at once: No loops required
- Extremely fast. Intuitive syntax

Array Arithmetic:

- Add, subtract, multiply, divide
- Works element-by-element
- Combine arrays together
- Perfect for data transformation

```
# Single array operations
prices = np.array([100, 200, 300, 400])

increased = prices * 1.1       # 10% increase
discounted = prices - 50       # $50 off each
doubled = prices * 2           # Double everything
halved = prices / 2            # Half price

# Array-to-array operations
cost = np.array([80, 150, 200, 300])
revenue = np.array([100, 200, 300, 400])

profit = revenue - cost        # [20, 50, 100, 100]
margin = profit / revenue      # [0.2, 0.25, 0.33, 0.25]

# Compare with lists (requires loops!)
# profit = [r - c for r, c in zip(revenue, cost)]
```

Statistical Functions

Instant Insights

Aggregate Functions:

- Calculate across entire array
- Common statistics built-in
- Essential for data analysis
- Fast on large datasets

Axis Operations:

- axis=0: Down columns
- axis=1: Across rows
- Control calculation direction
- Powerful for multi-dimensional data

```
# 1D array statistics
daily_sales = np.array([120, 150, 180, 145, 200, 220, 190])

total = daily_sales.sum()          # 1205
average = daily_sales.mean()       # 172.14
std_dev = daily_sales.std()        # Standard deviation
minimum = daily_sales.min()        # 120
maximum = daily_sales.max()        # 220

# 2D array statistics (by row/column)
store_data = np.array([[100, 150, 200],
                       [120, 180, 210],
                       [110, 160, 190]])

col_totals = store_data.sum(axis=0)  # [330, 490, 600]
row_totals = store_data.sum(axis=1)  # [450, 510, 460]
col_avg = store_data.mean(axis=0)    # Per-column average
```

Practice

Getting Started With NumPy

1. Create visitor array for the week
2. Calculate total weekly visitors
3. Find average daily visitors
4. Identify best & worst days
5. Project 15% growth scenario

```
# Starter code
import numpy as np

# Step 1: Create your array
visitors = np.array([120, 150, 180, 145, 200, 220, 190])

# Your code here...
# total = ?
# average = ?
# best_day = ?
# worst_day = ?
# growth_scenario = ?
```

Array Reshaping

Transforming Data Dimensions

Why Reshape?

- Prepare for analysis
- Match required formats
- Group data logically

Common Patterns:

- 1D to 2D (create table)
- 2D to 1D (flatten)
- Reorganize dimensions

```
# Transform monthly data to quarterly
monthly_sales = np.array([100, 120, 110,  # Q1
                          150, 160, 155,  # Q2
                          180, 190, 175,  # Q3
                          200, 210, 205]) # Q4

# Reshape: 12 elements → 4 rows × 3 columns
quarterly = monthly_sales.reshape(4, 3)
print(quarterly)

# Calculate quarterly totals
q_totals = quarterly.sum(axis=1)  # [330, 465, 545, 615]

# Flatten back to 1D
flat = quarterly.flatten()  # Back to 12 elements
```

Combining Arrays

Stacking and Merging Data

Vertical Stacking:

- Add rows (stack downward)
- Combine similar datasets
- Build larger tables
- vstack() function

Horizontal Stacking:

- Add columns (stack sideways)
- Merge related data
- Expand features
- hstack() function

```
# Add quarters together (vertical)
q1 = np.array([100, 150, 200])
q2 = np.array([120, 160, 210])
q3 = np.array([110, 170, 220])

yearly = np.vstack([q1, q2, q3])

# Add metrics side by side (horizontal)
revenue = np.array([100, 120, 110])
costs = np.array([60, 70, 65])

combined = np.hstack([revenue, costs])
# [100, 120, 110, 60, 70, 65]

# For 2D: use axis parameter
data = np.concatenate([arr1, arr2], axis=0)  # Rows
data = np.concatenate([arr1, arr2], axis=1)  # Cols
```

Boolean Filtering

Selecting Data With Conditions

How It Works:

- Create true/false mask
- Apply mask to filter
- Just like Pandas filtering!
- Multiple conditions possible

Use Cases:

- Find outliers
- Identify trends
- Filter by thresholds
- Segment data

```
sales = np.array([100, 250, 180, 320, 150, 280, 90, 400])

# Create boolean mask
high_performers = sales > 200
print(high_performers)
# [False, True, False, True, False, True, False, True]

# Filter the array
result = sales[high_performers]
print(result)  # [250, 320, 280, 400]

# One-liner version
result = sales[sales > 200]

# Multiple conditions
mid_range = sales[(sales >= 150) & (sales <= 300)]
# [250, 180, 150, 280]

# OR condition
extremes = sales[(sales < 100) | (sales > 350)]
# [90, 400]
```

Random Generation

Creating Data For Testing & Simulation

Why Generate Data?

- Test your analysis code
- Simulate scenarios
- Model uncertainty
- Practice before real data

Random Functions:

- Integers in range
- Floating point numbers
- Normal distributions
- Random selections

```
import numpy as np

# Random integers (simulate daily visitors)
visitors = np.random.randint(50, 200, size=30)

# Random floats between 0 and 1
probabilities = np.random.rand(10)

# Normal distribution (realistic data!)
# mean=100, std=15, 1000 samples
test_scores = np.random.normal(100, 15, size=1000)

# Random choice from options
products = np.random.choice(['A', 'B', 'C'], size=100)

# Set seed for reproducibility
np.random.seed(42)
data = np.random.randint(1, 100, 10)
# Same results every time you run it!

# Practical: simulate 100 days of sales
daily_sales = np.random.normal(5000, 500, size=100)
```

Q&A