# NumPy

Numerical Python for Scientific Computing

logo-eps-converted-to.pdf

August 7, 2025

# Contents

# 1 Introduction to NumPy

## 1.1 What is NumPy?

NumPy (Numerical Python) is the fundamental library for numerical computing in Python. It provides:

- Powerful `ndarray` data structure for fast, vectorized computations
- Tools for linear algebra, Fourier transforms, and random number generation
- The core foundation of the scientific Python ecosystem (pandas, SciPy, scikit-learn, etc.)

## 1.2 Why NumPy?

| Feature | Python Lists | NumPy Arrays |
|---|---|---|
| Memory Usage | High | Low (contiguous memory) |
| Speed | Slow (interpreted) | Fast (C implementation) |
| Vectorization | Manual loops | Built-in operations |
| Mathematical Operations | Limited | Extensive |
| Broadcasting | No | Yes |

Table 1: Python Lists vs NumPy Arrays

## 1.3 Installation and Import

```
# Installation
pip install numpy

# Import (standard convention)
import numpy as np

# Check version
print(np.__version__)
```

# 2 Array Creation

## 2.1 Basic Array Creation

Array Creation Methods

```
# From Python lists
arr_from_list = np.array([1, 2, 3, 4, 5])
matrix = np.array([[1, 2, 3], [4, 5, 6]])

# Using built-in functions
zeros_arr = np.zeros((3, 4))         # 3x4 array of zeros
ones_arr = np.ones((2, 3))           # 2x3 array of ones
empty_arr = np.empty((2, 2))         # Uninitialized array
full_arr = np.full((3, 3), 7)        # Fill with specific value

# Range functions
range_arr = np.arange(0, 10, 2)      # [0, 2, 4, 6, 8]
linspace_arr = np.linspace(0, 1, 5) # [0, 0.25, 0.5, 0.75, 1.0]
```

```
logspace_arr = np.logspace(0, 2, 3) # [1, 10, 100]
```

### 2.2 Special Arrays

```
# Identity matrix
identity = np.eye(4)                # 4x4 identity matrix

# Diagonal arrays
diagonal = np.diag([1, 2, 3, 4])    # Diagonal matrix
off_diagonal = np.diag([1, 2, 3], k=1)  # Above main diagonal

# Random arrays
np.random.seed(42)  # For reproducibility
random_uniform = np.random.rand(3, 3)        # Uniform [0,1)
random_normal = np.random.randn(3, 3)        # Standard normal
random_int = np.random.randint(1, 10, (3, 3)) # Random integers
```

### 2.3 Data Types

| Type | NumPy dtype | Description |
|---|---|---|
| Integer | `int8, int16, int32, int64` | Signed integers |
| Unsigned | `uint8, uint16, uint32, uint64` | Unsigned integers |
| Float | `float16, float32, float64` | Floating point |
| Complex | `complex64, complex128` | Complex numbers |
| Boolean | `bool` | True/False |
| String | `U<n>` | Unicode strings |

Table 2: NumPy Data Types

```
# Specifying data types
int_array = np.array([1, 2, 3], dtype=np.int32)
float_array = np.array([1, 2, 3], dtype=np.float64)
bool_array = np.array([1, 0, 1], dtype=bool)

# Type conversion
float_arr = int_array.astype(np.float64)
```

## 3 Array Attributes and Properties

### 3.1 Essential Attributes

```
arr = np.random.rand(3, 4, 5)

# Shape and dimensions
print(f"Shape: {arr.shape}")          # (3, 4, 5)
print(f"Dimensions: {arr.ndim}")      # 3
print(f"Size: {arr.size}")            # 60 (total elements)

# Data type and memory
print(f"Data type: {arr.dtype}")      # float64
print(f"Item size: {arr.itemsize}") # 8 bytes per element
print(f"Total bytes: {arr.nbytes}") # 480 bytes
```

```python

# Memory layout
print(f"C-contiguous: {arr.flags.c_contiguous}")
print(f"Fortran-contiguous: {arr.flags.f_contiguous}")
print(f"Owns data: {arr.flags.owndata}")
```

# 4 Array Indexing and Slicing

## 4.1 Basic Indexing

1D Array Indexing

```python
arr = np.arange(10)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Basic indexing
print(arr[0])         # 0 (first element)
print(arr[-1])        # 9 (last element)
print(arr[2:5])       # [2, 3, 4] (slice)
print(arr[::2])       # [0, 2, 4, 6, 8] (every 2nd element)
print(arr[::-1])      # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (reverse)
```

2D Array Indexing

```python
matrix = np.arange(12).reshape(3, 4)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Element access
print(matrix[1, 2])         # 6 (row 1, column 2)
print(matrix[1][2])         # 6 (alternative syntax)

# Row and column access
print(matrix[1, :])         # [4 5 6 7] (entire row 1)
print(matrix[:, 2])         # [2 6 10] (entire column 2)

# Submatrix
print(matrix[1:3, 1:3])     # 2x2 submatrix
print(matrix[::2, ::2])     # Every other row and column
```

## 4.2 Advanced Indexing

```python
arr = np.arange(10)

# Boolean indexing
mask = arr > 5
print(arr[mask])            # [6 7 8 9]

# Fancy indexing with arrays
indices = [1, 3, 5, 7]
print(arr[indices])         # [1 3 5 7]

# Combining conditions
mask = (arr > 2) & (arr < 8)
print(arr[mask])            # [3 4 5 6 7]

```

```python
# 2D boolean indexing
matrix = np.random.randint(0, 10, (3, 4))
print(matrix[matrix > 5])  # All elements > 5 (flattened)
```

# 5 Array Operations and Broadcasting

## 5.1 Element-wise Operations

```python
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

# Arithmetic operations
print(a + b)            # [6  8 10 12]
print(a * b)            # [5 12 21 32]
print(a ** 2)           # [1  4  9 16]
print(np.sqrt(a))       # [1.    1.414 1.732 2.   ]

# Comparison operations
print(a > 2)            # [False False  True  True]
print(a == b)           # [False False False False]

# Logical operations
print(np.logical_and(a > 1, a < 4))  # [False  True  True False]
```

## 5.2 Broadcasting

Broadcasting allows operations between arrays of different shapes following specific rules:

**Broadcasting Rules**

1. Arrays are aligned from the trailing dimension
2. Dimensions of size 1 can be ”stretched” to match
3. Missing dimensions are assumed to be size 1

**Broadcasting Examples**

```python
# Scalar and array
arr = np.array([[1, 2, 3], [4, 5, 6]])
result = arr + 10          # Adds 10 to every element

# 1D array with 2D array
row_vector = np.array([10, 20, 30])      # Shape: (3,)
result = arr + row_vector                # Shape: (2, 3)

# Column vector with row vector
col_vector = np.array([[1], [2]])        # Shape: (2, 1)
result = col_vector + row_vector         # Shape: (2, 3)

# Mathematical example
x = np.linspace(0, 5, 6).reshape(6, 1)   # Column vector
y = np.linspace(0, 4, 5).reshape(1, 5)   # Row vector
distance = np.sqrt(x**2 + y**2)          # 6x5 distance matrix
```

# 6 Array Manipulation

## 6.1 Reshaping and Resizing

```
arr = np.arange(12)

# Reshaping (must preserve total size)
reshaped = arr.reshape(3, 4)       # 1D to 2D
reshaped = arr.reshape(2, 2, 3)    # 1D to 3D
reshaped = arr.reshape(-1, 4)      # Auto-calculate rows

# Flattening
flattened = reshaped.flatten()     # Always returns copy
raveled = reshaped.ravel()         # Returns view if possible

# Transpose
transposed = reshaped.T            # Transpose
transposed = np.transpose(reshaped)

# Adding/removing dimensions
expanded = np.expand_dims(arr, axis=1)  # Add dimension
squeezed = np.squeeze(expanded)         # Remove size-1 dimensions
```

## 6.2 Concatenation and Splitting

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Concatenation
vertical = np.concatenate([a, b], axis=0)    # Stack vertically
horizontal = np.concatenate([a, b], axis=1)  # Stack horizontally

# Convenient functions
vstacked = np.vstack([a, b])       # Vertical stack
hstacked = np.hstack([a, b])       # Horizontal stack
dstacked = np.dstack([a, b])       # Depth stack (3D)

# Splitting
large_arr = np.arange(12).reshape(4, 3)
split_arrays = np.split(large_arr, 2, axis=0)     # Split into 2 parts
hsplit_arrays = np.hsplit(large_arr, 3)           # Split horizontally
vsplit_arrays = np.vsplit(large_arr, 2)           # Split vertically
```

# 7 Views vs Copies

Understanding when NumPy creates views (shared memory) vs copies (independent memory) is crucial for performance and correctness.

**Important: Views vs Copies**

Views share memory with the original array, while copies create independent arrays. Modifying a view affects the original array!

## 7.1 Operations that Create Views

```python
original = np.arange(12).reshape(3, 4)

# These create VIEWS (share memory)
slice_view = original[1:3]              # Slicing
transpose_view = original.T             # Transpose
reshape_view = original.reshape(4, 3)   # Reshape (when possible)
ravel_view = original.ravel()           # Ravel (when possible)

# Test if it's a view
print(slice_view.base is original)      # True for views
print(original.flags.owndata)          # False for views
```

### 7.2 Operations that Create Copies

```python
# These create COPIES (independent memory)
explicit_copy = original.copy()         # Explicit copy
flatten_copy = original.flatten()       # Flatten always copies
fancy_copy = original[[0, 2]]           # Fancy indexing
boolean_copy = original[original > 5]   # Boolean indexing

# Test memory sharing
print(np.shares_memory(original, explicit_copy))  # False for copies
```

### 7.3 Practical Example

```python
# Demonstrate view behavior
matrix = np.arange(20).reshape(4, 5)

# Working with views - efficient but affects original
first_row = matrix[0]         # This is a view!
first_row *= 10               # Modifies original matrix!

# Working with copies - safe but uses more memory
safe_first_row = matrix[0].copy()
safe_first_row *= 10          # Original matrix unchanged
```

## 8 Mathematical and Statistical Operations

### 8.1 Basic Mathematical Functions

```python
arr = np.array([1, 4, 9, 16, 25])

# Trigonometric functions
angles = np.array([0, np.pi/4, np.pi/2, np.pi])
print(np.sin(angles))
print(np.cos(angles))
print(np.tan(angles))

# Exponential and logarithmic
print(np.exp(arr))            # e^x
print(np.log(arr))            # Natural log
print(np.log10(arr))          # Base-10 log
print(np.sqrt(arr))           # Square root

# Rounding
float_arr = np.array([1.2, 2.7, 3.1, 4.9])
print(np.round(float_arr))    # [1. 3. 3. 5.]
```

```python
print(np.floor(float_arr))   # [1. 2. 3. 4.]
print(np.ceil(float_arr))    # [2. 3. 4. 5.]
```

### 8.2 Statistical Functions

```python
data = np.random.randn(1000)  # Normal distribution

# Central tendency
print(f"Mean: {np.mean(data):.3f}")
print(f"Median: {np.median(data):.3f}")
print(f"Mode: {stats.mode(data)[0][0]:.3f}")  # Requires scipy

# Spread
print(f"Standard deviation: {np.std(data):.3f}")
print(f"Variance: {np.var(data):.3f}")
print(f"Range: {np.ptp(data):.3f}")  # Peak-to-peak

# Quantiles
print(f"25th percentile: {np.percentile(data, 25):.3f}")
print(f"75th percentile: {np.percentile(data, 75):.3f}")
print(f"IQR: {np.percentile(data, 75) - np.percentile(data, 25):.3f}")

# Extremes
print(f"Min: {np.min(data):.3f}")
print(f"Max: {np.max(data):.3f}")
print(f"Argmin: {np.argmin(data)}")  # Index of minimum
print(f"Argmax: {np.argmax(data)}")  # Index of maximum
```

### 8.3 Axis Parameter

The `axis` parameter specifies along which dimension to perform operations:

**Understanding Axis Parameter**

```python
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# axis=0: operate down rows (collapse rows)
col_sums = np.sum(matrix, axis=0)    # [15, 18, 21, 24]
col_means = np.mean(matrix, axis=0)  # [5., 6., 7., 8.]

# axis=1: operate across columns (collapse columns)
row_sums = np.sum(matrix, axis=1)    # [10, 26, 42]
row_means = np.mean(matrix, axis=1)  # [2.5, 6.5, 10.5]

# No axis: operate on entire array
total_sum = np.sum(matrix)           # 78
overall_mean = np.mean(matrix)       # 6.5
```

## 9 Linear Algebra

### 9.1 Basic Operations

```python
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
```

```python
v = np.array([1, 2])

# Matrix multiplication
C = np.dot(A, B)        # Matrix multiplication
C = A @ B               # Alternative syntax (Python 3.5+)

# Vector operations
dot_product = np.dot(v, v)      # Scalar result
outer_product = np.outer(v, v)  # Matrix result

# Matrix properties
det_A = np.linalg.det(A)        # Determinant
trace_A = np.trace(A)           # Trace (sum of diagonal)
```

### 9.2 Advanced Linear Algebra

```python
# Create a symmetric positive definite matrix
A = np.array([[4, 2, 1],
              [2, 5, 3],
              [1, 3, 6]])

# Eigenvalues and eigenvectors
eigenvals, eigenvecs = np.linalg.eig(A)
print(f"Eigenvalues: {eigenvals}")

# Matrix decompositions
U, s, Vt = np.linalg.svd(A)     # Singular Value Decomposition
L = np.linalg.cholesky(A)       # Cholesky decomposition
Q, R = np.linalg.qr(A)          # QR decomposition

# Matrix inverse and pseudo-inverse
A_inv = np.linalg.inv(A)        # Inverse (for square matrices)
A_pinv = np.linalg.pinv(A)      # Pseudo-inverse (Moore-Penrose)

# Solving linear systems: Ax = b
b = np.array([1, 2, 3])
x = np.linalg.solve(A, b)       # Solve Ax = b
print(f"Solution: {x}")
print(f"Verification: {np.allclose(A @ x, b)}")
```

## 10 Working with Missing Data

### 10.1 NaN Handling

```python
# Creating arrays with NaN
data = np.array([1.0, 2.0, np.nan, 4.0, 5.0])

# Detecting NaN
nan_mask = np.isnan(data)       # Boolean array
has_nan = np.any(np.isnan(data))  # True if any NaN

# NaN-aware functions
print(f"Mean (ignoring NaN): {np.nanmean(data)}")
print(f"Sum (ignoring NaN): {np.nansum(data)}")
print(f"Std (ignoring NaN): {np.nanstd(data)}")

# Removing NaN values
clean_data = data[~np.isnan(data)]

```

```python
# Replacing NaN values
filled_data = np.where(np.isnan(data), 0, data)  # Replace with 0
filled_data = np.nan_to_num(data, nan=0.0)       # Replace with 0
```

### 10.2 Masked Arrays

```python
# Masked arrays for handling missing/invalid data
temperatures = np.array([20.5, -999, 22.1, -999, 19.8])

# Create masked array (mask invalid values)
masked_temp = np.ma.masked_where(temperatures == -999, temperatures)

# Operations automatically ignore masked values
print(f"Mean temperature: {masked_temp.mean():.1f}")
print(f"Valid measurements: {masked_temp.count()}")

# Fill masked values
filled = masked_temp.filled(fill_value=20.0)
```

## 11 File I/O Operations

### 11.1 NumPy Native Formats

```python
# Save and load single arrays
data = np.random.randn(100, 50)

# Binary format (fast, preserves exact data)
np.save('data.npy', data)
loaded_data = np.load('data.npy')

# Compressed format
np.savez_compressed('data.npz', data=data)

# Multiple arrays
sales = np.random.randint(100, 1000, (12, 4))
costs = sales * 0.6
np.savez('company_data.npz', sales=sales, costs=costs)

# Load multiple arrays
loaded = np.load('company_data.npz')
sales_loaded = loaded['sales']
costs_loaded = loaded['costs']
```

### 11.2 Text Formats

```python
# Save to text (human-readable)
data = np.random.rand(5, 3)
np.savetxt('data.csv', data, delimiter=',', fmt='%.3f')

# Load from text
loaded = np.loadtxt('data.csv', delimiter=',')

# Advanced loading with headers and specific columns
# File format: "Name,Age,Score1,Score2,Score3"
student_scores = np.loadtxt('students.csv',
                           delimiter=',',
                           skiprows=1,          # Skip header
```

```
                                  usecols=(2, 3, 4),  # Load only scores
                                  dtype=float)

# Handle missing data
data_with_missing = np.genfromtxt('messy_data.csv',
                                  delimiter=',',
                                  missing_values='N/A',
                                  filling_values=0)
```

# 12 Performance Optimization

## 12.1 Vectorization vs Loops

**Performance Tip**

Always prefer vectorized operations over Python loops for numerical computations. NumPy operations are implemented in C and are typically 10-100x faster.

```
import time

# Large dataset
large_array = np.random.rand(1_000_000)

# Python loop (slow)
def python_sum(arr):
    total = 0.0
    for value in arr:
        total += value
    return total

# Time comparison
start = time.time()
py_result = python_sum(large_array)
py_time = time.time() - start

start = time.time()
np_result = np.sum(large_array)
np_time = time.time() - start

print(f"Python loop: {py_time:.4f} seconds")
print(f"NumPy vectorized: {np_time:.4f} seconds")
print(f"Speedup: {py_time/np_time:.1f}x")
```

## 12.2 Memory Optimization

```
# Use appropriate data types
small_ints = np.arange(1000, dtype=np.int8)    # 1 byte per element
large_ints = np.arange(1000, dtype=np.int64)   # 8 bytes per element

print(f"int8 memory: {small_ints.nbytes} bytes")
print(f"int64 memory: {large_ints.nbytes} bytes")

# In-place operations to save memory
arr = np.random.rand(1000, 1000)
arr += 5        # In-place addition (saves memory)
arr *= 2        # In-place multiplication

# Instead of:
```

```python
# arr = arr + 5   # Creates new array
# arr = arr * 2   # Creates new array

# Pre-allocate arrays when possible
result = np.empty(1000)  # Faster than appending to lists
for i in range(1000):
    result[i] = expensive_computation(i)
```

# 13 Common Pitfalls and Debugging

## 13.1 Frequent Mistakes

**Common Pitfall: Unexpected Broadcasting**

```python
# This can create unexpected results
a = np.array([[1, 2, 3]])      # Shape: (1, 3)
b = np.array([[1], [2], [3]])  # Shape: (3, 1)
result = a + b                  # Shape: (3, 3) - often unexpected!

# Be explicit about your intentions
a_explicit = np.array([1, 2, 3])            # Shape: (3,)
b_explicit = np.array([1, 2, 3])            # Shape: (3,)
result_explicit = a_explicit + b_explicit  # Shape: (3,)
```

**Common Pitfall: View Modifications**

```python
# Views can cause unexpected side effects
matrix = np.arange(12).reshape(3, 4)
row = matrix[0]     # This is a view!
row[0] = 999        # Modifies the original matrix!

# Solution: Use copy() when you need independence
safe_row = matrix[0].copy()
safe_row[0] = 999  # Original matrix unchanged
```

## 13.2 Debugging Techniques

```python
def debug_array(arr, name="Array"):
    """Comprehensive array debugging information"""
    print(f"\n=== {name} Debug Info ===")
    print(f"Shape: {arr.shape}")
    print(f"Dtype: {arr.dtype}")
    print(f"Size: {arr.size}")
    print(f"Dimensions: {arr.ndim}")
    print(f"Memory (bytes): {arr.nbytes}")
    print(f"Strides: {arr.strides}")
    print(f"C-contiguous: {arr.flags.c_contiguous}")
    print(f"F-contiguous: {arr.flags.f_contiguous}")
    print(f"Owns data: {arr.flags.owndata}")

    if arr.size > 0:
        print(f"Min: {arr.min()}")
        print(f"Max: {arr.max()}")
        print(f"Mean: {arr.mean():.3f}")

    if arr.size <= 20:
```

```python
        print(f"Values:\n{arr}")
    else:
        print(f"First 5 values: {arr.flat[:5]}")

# Usage
problematic_array = np.random.rand(3, 4)
debug_array(problematic_array, "My Array")
```

# 14 Advanced Topics

## 14.1 Custom Universal Functions (ufuncs)

```python
# Create custom vectorized functions
def sigmoid(x):
    """Sigmoid activation function"""
    return 1 / (1 + np.exp(-x))

# Vectorize for NumPy arrays
vectorized_sigmoid = np.vectorize(sigmoid)

# Usage
x = np.linspace(-10, 10, 100)
y = vectorized_sigmoid(x)

# Alternative: Use existing NumPy functions
def fast_sigmoid(x):
    """Faster implementation using NumPy"""
    return 1 / (1 + np.exp(-np.clip(x, -500, 500)))  # Prevent overflow
```

## 14.2 Structured Arrays

```python
# Define custom data types
dtype = np.dtype([
    ('name', 'U20'),         # Unicode string, max 20 chars
    ('age', 'i4'),           # 32-bit integer
    ('salary', 'f8'),        # 64-bit float
    ('department', 'U10')    # Unicode string, max 10 chars
])

# Create structured array
employees = np.array([
    ('Alice', 30, 75000.0, 'Engineering'),
    ('Bob', 25, 50000.0, 'Marketing'),
    ('Charlie', 35, 85000.0, 'Engineering')
], dtype=dtype)

# Access fields
print(employees['name'])         # All names
print(employees['salary'])       # All salaries
print(employees[employees['age'] > 30])  # Employees over 30
```

# 15 Integration with Scientific Python Ecosystem

## 15.1 With Pandas

```python
import pandas as pd

```

```python
# NumPy to Pandas
data = np.random.randn(100, 4)
df = pd.DataFrame(data, columns=['A', 'B', 'C', 'D'])

# Pandas to NumPy
numpy_data = df.values           # All data
numpy_subset = df[['A', 'B']].values  # Specific columns
```

### 15.2 With Matplotlib

```python
import matplotlib.pyplot as plt

# Generate data
x = np.linspace(0, 2*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Plot
plt.figure(figsize=(10, 6))
plt.plot(x, y1, label='sin(x)', linewidth=2)
plt.plot(x, y2, label='cos(x)', linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Trigonometric Functions')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

### 15.3 With SciPy

```python
from scipy import stats, optimize, integrate

# Statistical analysis
data = np.random.normal(100, 15, 1000)
mean_est, std_est = stats.norm.fit(data)

# Optimization
def objective(x):
    return (x - 3)**2 + 1

result = optimize.minimize(objective, x0=0)
print(f"Minimum at x = {result.x[0]:.3f}")

# Numerical integration
def integrand(x):
    return np.exp(-x**2)

integral, error = integrate.quad(integrand, 0, np.inf)
print(f"Integral: {integral:.6f}    {error:.2e}")
```

## 16 Real-World Applications

### 16.1 Image Processing

```python
# Create synthetic image
def create_image(size=256):
    x = np.linspace(-2, 2, size)
```

```python
    y = np.linspace(-2, 2, size)
    X, Y = np.meshgrid(x, y)

    # Create pattern
    image = np.sin(2*np.pi*X) * np.cos(2*np.pi*Y)
    image = (image + 1) / 2 * 255  # Normalize to 0-255
    return image.astype(np.uint8)

image = create_image()

# Image operations
def image_operations(img):
    # Horizontal flip
    flipped = np.fliplr(img)

    # Rotation (90 degrees)
    rotated = np.rot90(img)

    # Brightness adjustment
    brighter = np.clip(img + 50, 0, 255)

    # Contrast enhancement
    enhanced = np.clip(img * 1.5, 0, 255)

    return flipped, rotated, brighter, enhanced

processed_images = image_operations(image)
```

## 16.2 Data Analysis Example

```python
# Sales data analysis
np.random.seed(42)

# Generate synthetic sales data
months = 12
products = 5
sales_data = np.random.normal(1000, 200, (months, products))
sales_data = np.maximum(sales_data, 0)  # Ensure non-negative

product_names = ['Product A', 'Product B', 'Product C', 'Product D', 'Product E
    ']
month_names = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
               'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

# Analysis
total_sales = np.sum(sales_data)
monthly_totals = np.sum(sales_data, axis=1)
product_totals = np.sum(sales_data, axis=0)

# Find best and worst performers
best_month_idx = np.argmax(monthly_totals)
worst_month_idx = np.argmin(monthly_totals)
best_product_idx = np.argmax(product_totals)

print(f"Best month: {month_names[best_month_idx]} (${monthly_totals[
    best_month_idx]:.2f})")
print(f"Worst month: {month_names[worst_month_idx]} (${monthly_totals[
    worst_month_idx]:.2f})")
print(f"Best product: {product_names[best_product_idx]} (${product_totals[
    best_product_idx]:.2f})")

# Growth analysis
```

```python
monthly_growth = np.diff(monthly_totals) / monthly_totals[:-1] * 100
average_growth = np.mean(monthly_growth)
print(f"Average monthly growth: {average_growth:.2f}%")
```

# 17 Practice Exercises

## 17.1 Beginner Exercises

1. Create a 5×5 matrix with values ranging from 0 to 24
2. Extract the border elements of the matrix
3. Replace all odd numbers with -1
4. Calculate the sum of each row and column

## 17.2 Intermediate Exercises

1. Implement a function to normalize a dataset (zero mean, unit variance)
2. Create a function to find the k-nearest neighbors in a 2D dataset
3. Implement moving average smoothing for a 1D signal
4. Calculate correlation matrix for a multi-variate dataset

## 17.3 Advanced Exercises

1. Implement Principal Component Analysis (PCA) from scratch
2. Create a function for convolution of two signals
3. Implement k-means clustering algorithm
4. Design a neural network layer using only NumPy

# 18 Best Practices and Guidelines

NumPy Best Practices

1. **Always use vectorized operations** instead of Python loops
2. **Choose appropriate data types** to minimize memory usage
3. **Be aware of views vs copies** to avoid unexpected behavior
4. **Use axis parameter wisely** for multi-dimensional operations
5. **Preallocate arrays** when size is known in advance
6. **Use in-place operations** when possible to save memory
7. **Handle NaN and infinity values** appropriately
8. **Profile your code** to identify bottlenecks

## 19 Conclusion

NumPy is the foundation of scientific computing in Python, providing efficient array operations, mathematical functions, and seamless integration with the broader scientific Python ecosystem. Mastering NumPy concepts like broadcasting, views vs copies, and vectorization will significantly improve your data analysis and scientific computing capabilities.

Key takeaways:

- NumPy arrays are faster and more memory-efficient than Python lists
- Understanding broadcasting enables elegant solutions to complex problems
- Views and copies behavior affects both performance and correctness
- Vectorized operations should always be preferred over loops
- NumPy integrates seamlessly with pandas, matplotlib, SciPy, and scikit-learn

## 20 Further Reading

- Official NumPy Documentation: `https://numpy.org/doc/`
- NumPy User Guide: `https://numpy.org/doc/stable/user/`
- SciPy Lecture Notes: `https://scipy-lectures.org/`
- Python Data Science Handbook: `https://jakevdp.github.io/PythonDataScienceHandbook/`