intro-to-numpy

April 10, 2026

[1]:
```
# sources:
#    https://cs231n.github.io/python-numpy-tutorial/#numpy
#    Claude
```

[2]:
```
# Introduction to NumPy for Neural Network Computations
# ======================================================

# NumPy may need to be installed
# !python3.14 -m pip install numpy

import numpy as np
```

[3]:
```
# 1. NUMPY BASICS AND ARRAY CREATION
# ==================================

# NumPy's main object is the homogeneous (same data type) multidimensional array
# Key advantage: vectorized operations, much faster than Python lists

# Creating arrays from Python lists
vector = np.array([1, 2, 3, 4])
matrix = np.array([[1, 2, 3], [4, 5, 6]])
three_d_array = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]], dtype=np.float32)

print(f"Vector:\n{vector}")
print("Vector shape:", vector.shape)  # (4,) - shape is a tuple
print("Vector dimensions:", vector.ndim)  # 1
print("Vector size:", vector.size)  # 4 (total number of elements)
print("Vector data type:", vector.dtype)  # int64 by default for integers

print(f"\nMatrix:\n{matrix}")
print("Matrix shape:", matrix.shape)  # (2, 3)
print("Matrix dimensions:", matrix.ndim)  # 2
print("Matrix size:", matrix.size)


print(f"\n3D array:\n{three_d_array}")
print("3D array shape:", three_d_array.shape)  # (2, 2, 2)
print("3D array dimensions:", three_d_array.ndim)  # 3
```

```
print("3D array size:", three_d_array.size)  # 8
print("3D array data type:", three_d_array.dtype)  # int64
```

```
Vector:
[1 2 3 4]
Vector shape: (4,)
Vector dimensions: 1
Vector size: 4
Vector data type: int64

Matrix:
[[1 2 3]
 [4 5 6]]
Matrix shape: (2, 3)
Matrix dimensions: 2
Matrix size: 6

3D array:
[[[1. 2.]
  [3. 4.]]

 [[5. 6.]
  [7. 8.]]]
3D array shape: (2, 2, 2)
3D array dimensions: 3
3D array size: 8
3D array data type: float32
```

[4]:
```
# Array creation functions - convenient ways to create arrays
zeros = np.zeros((3, 4))                # 3x4 array of zeros
ones = np.ones((2, 3))                  # 2x3 array of ones
full = np.full((2, 2), 7)               # 2x2 array filled with 7
identity = np.eye(3)                    # 3x3 identity matrix
diagonal = np.diag([1, 2, 3, 4])        # Diagonal matrix
evenly_spaced = np.linspace(0, 10, 5)   # 5 evenly spaced values from 0 to 10
step_range = np.arange(0, 10, 2)        # Values from 0 to 10 with step 2

print("\nZeros:\n", zeros)
print("Ones:\n", ones)
print("Full:\n", full)
print("Identity:\n", identity)
print("Diagonal:\n", diagonal)
print("Evenly spaced:\n", evenly_spaced)
print("Step range:\n", step_range)
```

```
Zeros:
 [[0. 0. 0. 0.]
```

```
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
Ones:
 [[1. 1. 1.]
 [1. 1. 1.]]
Full:
 [[7 7]
 [7 7]]
Identity:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Diagonal:
 [[1 0 0 0]
 [0 2 0 0]
 [0 0 3 0]
 [0 0 0 4]]
Evenly spaced:
 [ 0.   2.5  5.   7.5 10. ]
Step range:
 [0 2 4 6 8]
```

[5]:
```python
# Random arrays
rand_uniform = np.random.rand(2, 3)      # Uniform random [0,1)
rand_normal = np.random.randn(2, 3)      # Normal distribution ( =0,  =1)

print("Random uniform array:\n", rand_uniform)
print("\nRandom normal array:\n", rand_normal)
```

```
Random uniform array:
 [[0.28469247 0.39705657 0.63075154]
 [0.27028269 0.06713568 0.29767807]]

Random normal array:
 [[ 1.15237864 -1.22678522 -2.16919444]
 [ 0.14332282 -1.4708949  -0.19480002]]
```

[6]:
```python
# 2. IMPORTANT: FLOATING-POINT PRECISION
# ======================================

# NumPy uses floating-point arithmetic, not exact arithmetic!

# Example of floating-point precision issues
x = np.array([0.1, 0.2, 0.3])
print("Sum of 0.1 + 0.2:", x[0] + x[1])  # Not exactly 0.3!
print("Is 0.1 + 0.2 == 0.3?", (x[0] + x[1]) == x[2])  # Returns False!

# Machine epsilon (smallest number   such that 1 +    1)
```

```
print("Machine epsilon:", np.finfo(float).eps)

# For comparing floating-point values, use np.isclose or np.allclose
print("Is 0.1 + 0.2 close to 0.3?", np.isclose(x[0] + x[1], x[2]))
```

```
Sum of 0.1 + 0.2: 0.30000000000000004
Is 0.1 + 0.2 == 0.3? False
Machine epsilon: 2.220446049250313e-16
Is 0.1 + 0.2 close to 0.3? True
```

[7]:
```
# 3. DATA TYPES IN NUMPY
# ======================

# NumPy has a variety of data types
int_array = np.array([1, 2, 3])                    # int64 by default
float_array = np.array([1.0, 2.0, 3.0])            # float64 by default
complex_array = np.array([1+2j, 3+4j])             # complex128

print("Integer array:\n", int_array)
print("Float array:\n", float_array)
print("Complex array:\n", complex_array)
```

```
Integer array:
 [1 2 3]
Float array:
 [1. 2. 3.]
Complex array:
 [1.+2.j 3.+4.j]
```

[8]:
```
# Explicit type specification
int32_array = np.array([1, 2, 3], dtype=np.int32)
float32_array = np.array([1, 2, 3], dtype=np.float32)
print("Explicit int32 array:\n", int32_array)
print("Explicit float32 array:\n", float32_array)

# Neural networks often use float32 or even float16 for efficiency
# The choice of precision affects both memory usage and computation speed
```

```
Explicit int32 array:
 [1 2 3]
Explicit float32 array:
 [1. 2. 3.]
```

[9]:
```
# 4. ARRAY INDEXING AND SLICING
# =============================

# Create a sample 3x4 matrix
arr = np.array([
    [1, 2, 3, 4],
```

```
    [5, 6, 7, 8],
    [9, 10, 11, 12]
])

# Basic indexing (row, column)
print("Element at (0,0):", arr[0, 0])    # 1  arr[0][0]
print("Element at (2,3):", arr[2, 3])    # 12
```

```
Element at (0,0): 1
Element at (2,3): 12
```

[12]:
```
L = [1,2,3,4]
print(L[1:3])
print(L[:])
```

```
[2, 3]
[1, 2, 3, 4]
```

[11]:
```
arr = np.array([
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
])
# Slicing [start:stop:step]
print("\nSlicing examples:")
print("First row:\n", arr[0, :])           # [1 2 3 4]
print("First column:\n", arr[:, 0])        # [1 5 9]
print("2x2 sub-matrix:\n", arr[:2, :2])    # [[1 2],[5 6]]
print("Last column:\n", arr[:, -1])        # [4 8 12]
```

```
Slicing examples:
First row:
 [1 2 3 4]
First column:
 [1 5 9]
2x2 sub-matrix:
 [[1 2]
 [5 6]]
Last column:
 [ 4  8 12]
```

[14]:
```
L = [1,2,3,4]
print(L[2])
print(L[2:3])
```

```
3
[3]
```

[13]:
```
# Mixing integer indexing with slices
# This changes the dimensionality of the output!
print("\nMixing indexing with slices:")
print("Array:\n", arr)
print("Second row (1D):\n", arr[1, :])      # 1D array: [5 6 7 8]
print("Second row (2D):\n", arr[1:2, :])    # 2D array: [[5 6 7 8]]
print("Second row (2D):\n", arr[1:3, :])    # 2D array: [[5 6 7 8]]
print("Shape of 1D extract:\n", arr[1, :].shape)     # (4,)
print("Shape of 2D extract:\n", arr[1:2, :].shape)   # (1, 4)
```

```
Mixing indexing with slices:
Array:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
Second row (1D):
 [5 6 7 8]
Second row (2D):
 [[5 6 7 8]]
Second row (2D):
 [[ 5  6  7  8]
 [ 9 10 11 12]]
Shape of 1D extract:
 (4,)
Shape of 2D extract:
 (1, 4)
```

[15]:
```
# 5. VIEWS VS COPIES - CRITICAL!!
# ========================================

# Slicing creates VIEWS, not copies - modifying a slice modifies the original!
original = np.array([[1, 2, 3], [4, 5, 6]])
print("Original array:\n", original)

# Create a view through slicing
view = original[:, :2]  # First two columns
print("\nView of first two columns:\n", view)

# Modify the view - this changes the original array!
view[0, 0] = 99
print("\nAfter modifying the view:")
print("View:\n", view)
print("Original array (also changed):\n", original)
```

```
Original array:
 [[1 2 3]
 [4 5 6]]
```

```
View of first two columns:
 [[1 2]
 [4 5]]

After modifying the view:
View:
 [[99  2]
 [ 4  5]]
Original array (also changed):
 [[99  2  3]
 [ 4  5  6]]
```

[17]:
```
# To create an independent copy, use .copy()
original = np.array([[1, 2, 3], [4, 5, 6]])
copy = original[:, :2].copy()
copy[0, 0] = 99
print("\nAfter modifying the copy:")
print("Copy:\n", copy)
print("Original array (unchanged):\n", original)

# - Views are memory-efficient but can cause unexpected side effects
# - Copies are safer but consume more memory
```

```
After modifying the copy:
Copy:
 [[99  2]
 [ 4  5]]
Original array (unchanged):
 [[1 2 3]
 [4 5 6]]
```

[16]:
```
# 6. BOOLEAN INDEXING
# ====================

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

# Create a boolean mask
mask = data > 5
print("Boolean mask:\n", mask)
```

```
Boolean mask:
 [[False False False False]
 [False  True  True  True]
```

```
 [ True  True  True  True]]
```

[17]:
```
# Use the mask to extract elements
filtered = data[mask]  # Returns a 1D array of values where mask is True
print("\nValues > 5:", filtered)
```

```
Values > 5: [ 6  7  8  9 10 11 12]
```

[19]:
```
data[data % 3 == 0]
```

[19]:
```
array([ 3,  6,  9, 12])
```

[20]:
```
print("Values divisible by 3:", data[data % 3 == 0])
```

```
Values divisible by 3: [ 3  6  9 12]
```

[21]:
```
combined_mask = (data > 3) & (data < 10)
print("Values between 3 and 10:", data[combined_mask])
```

```
Values between 3 and 10: [4 5 6 7 8 9]
```

[22]:
```
data_copy = data.copy()
data_copy[data_copy % 3 == 0] = -1
print("Replace multiples of 3 with -1:\n", data_copy)
```

```
Replace multiples of 3 with -1:
 [[ 1  2 -1  4]
 [ 5 -1  7  8]
 [-1 10 11 -1]]
```

[23]:
```
# 7. RESHAPING AND ARRAY MANIPULATION
# ==================================

# Create a sample array
arr = np.arange(12)  # [0, 1, 2, ..., 11]
print("Original array:", arr)
print("Shape:", arr.shape)  # (12,)

# Reshape to a 3x4 matrix
matrix = arr.reshape(3, 4)
print("Reshaped to 3x4 matrix:\n", matrix)
```

```
Original array: [ 0  1  2  3  4  5  6  7  8  9 10 11]
Shape: (12,)
Reshaped to 3x4 matrix:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

[26]:
```
# Use -1 to automatically determine the size of one dimension
auto_reshaped = arr.reshape(2, -1)  # 2 rows, columns determined automatically
print("Reshaped to 2 rows:\n", auto_reshaped)  # 2x6 matrix
other_way = arr.reshape(-1, 2) # 2 columns, numpy figures out the number of rows
print(other_way)
```

```
Reshaped to 2 rows:
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
[[ 0  1]
 [ 2  3]
 [ 4  5]
 [ 6  7]
 [ 8  9]
 [10 11]]
```

[27]:
```
# Flattening back to 1D
flattened = matrix.flatten()  # Returns a copy
print("Flattened array:", flattened)
```

```
Flattened array: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

[31]:
```
arr # 1D vector
print(arr.reshape(-1,1))
```

```
[[ 0]
 [ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]
 [11]]
```

[32]:
```
# Transpose (swap rows and columns)
transposed = matrix.T
print("Matrix:\n",matrix)
print("\nTransposed matrix:\n", transposed)  # Now 4x3
```

```
Matrix:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

Transposed matrix:
```

```
 [[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

[33]:
```
# Adding new axes (useful for broadcasting)
column_vector = arr.reshape(-1, 1)  # Convert to column vector
print("Arr:\n",arr)
print("Column vector:\n", column_vector)
print("Column vector shape:", column_vector.shape)  # (12, 1)
```

```
Arr:
 [ 0  1  2  3  4  5  6  7  8  9 10 11]
Column vector:
 [[ 0]
 [ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]
 [11]]
Column vector shape: (12, 1)
```

[34]:
```
row_vector = arr.reshape(1, -1)
print("Row vector shape:", row_vector.shape)  # (1, 12)
print("Row vector:\n", row_vector)
```

```
Row vector shape: (1, 12)
Row vector:
 [[ 0  1  2  3  4  5  6  7  8  9 10 11]]
```

[ ]:
```
# Reshaping makes a VIEW (most of the time), not a copy!
# This means the reshaped array shares the same underlying memory as the␣
 ↪original.
# Modifying one will modify the other. If you need an independent copy, use .
 ↪copy().
# NumPy is forced to make a copy when the data isn't contiguous in memory
# (e.g., after a transpose or non-contiguous slice).
```

[35]:
```
arr
```

[35]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

[36]:
```
column_vector[4, 0] = 99
column_vector
```

[36]:
```
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [99],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10],
       [11]])
```

[37]:
```
arr
```

[37]:
```
array([ 0,  1,  2,  3, 99,  5,  6,  7,  8,  9, 10, 11])
```

[38]:
```
# 8. ELEMENT-WISE OPERATIONS
# ==========================

# Create sample arrays
a = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
b = np.array([[9, 10], [11, 12], [13, 14], [15, 16]])

print("Array a:\n", a)
print("Array b:\n", b)
```

```
Array a:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
Array b:
 [[ 9 10]
 [11 12]
 [13 14]
 [15 16]]
```

[39]:
```
# Basic arithmetic operations (element-wise)
print("a + b =\n", a + b)
print("a - b =\n", a - b)
print("a * b =\n", a * b)
print("b / a =\n", b / a)
print("b // a =\n", b // a)
print("a ** 2 =\n", a ** 2)
```

```
print("a % 2 =\n", a % 2)
```

```
a + b =
 [[10 12]
 [14 16]
 [18 20]
 [22 24]]
a - b =
 [[-8 -8]
 [-8 -8]
 [-8 -8]
 [-8 -8]]
a * b =
 [[  9  20]
 [ 33  48]
 [ 65  84]
 [105 128]]
b / a =
 [[9.         5.        ]
 [3.66666667 3.        ]
 [2.6        2.33333333]
 [2.14285714 2.        ]]
b // a =
 [[9 5]
 [3 3]
 [2 2]
 [2 2]]
a ** 2 =
 [[ 1  4]
 [ 9 16]
 [25 36]
 [49 64]]
a % 2 =
 [[1 0]
 [1 0]
 [1 0]
 [1 0]]
```

[40]:
```
print("a squared:\n", np.square(a)) # versus a ** 2
print("Square root of a:\n", np.sqrt(a))
print("Exponential of a:\n", np.exp(a))
print("Natural log of a:\n", np.log(a))
print("Sine of a:\n", np.sin(a))
print("Absolute values:\n", np.abs([-1, -2, 3]))

# note the "np." in each of these! This uses a numpy version of these functions␣
 ↪to apply element-wise, and FAST
```

```
a squared:
 [[ 1  4]
 [ 9 16]
 [25 36]
 [49 64]]
Square root of a:
 [[1.         1.41421356]
 [1.73205081 2.        ]
 [2.23606798 2.44948974]
 [2.64575131 2.82842712]]
Exponential of a:
 [[2.71828183e+00 7.38905610e+00]
 [2.00855369e+01 5.45981500e+01]
 [1.48413159e+02 4.03428793e+02]
 [1.09663316e+03 2.98095799e+03]]
Natural log of a:
 [[0.         0.69314718]
 [1.09861229 1.38629436]
 [1.60943791 1.79175947]
 [1.94591015 2.07944154]]
Sine of a:
 [[ 0.84147098  0.90929743]
 [ 0.14112001 -0.7568025 ]
 [-0.95892427 -0.2794155 ]
 [ 0.6569866   0.98935825]]
Absolute values:
 [1 2 3]
```

[42]:
```
print(2*a)
```

```
[[ 2  4]
 [ 6  8]
 [10 12]
 [14 16]]
```

[43]:
```
print("Element-wise comparisons:")
print("a > 2:\n", a > 2)
print("a == b:\n", a == b)
print("a < b:\n", a < b)
print("Maximum:\n", np.maximum(a, b))
```

```
Element-wise comparisons:
a > 2:
 [[False False]
 [ True  True]
 [ True  True]
 [ True  True]]
a == b:
 [[False False]
```

```
 [False False]
 [False False]
 [False False]]
a < b:
 [[ True  True]
 [ True  True]
 [ True  True]
 [ True  True]]
Maximum:
 [[ 9 10]
 [11 12]
 [13 14]
 [15 16]]
```

[44]:
```
# 9. MATRIX OPERATIONS
# ====================
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
v = np.array([9, 10])  # Vector

print("Matrix A:\n", A)
print("Matrix B:\n", B)
print("Vector v:", v)
```

```
Matrix A:
 [[1 2]
 [3 4]]
Matrix B:
 [[5 6]
 [7 8]]
Vector v: [ 9 10]
```

[45]:
```
# Matrix multiplication (dot product)
# This is NOT the same as element-wise multiplication!
C = A.dot(B)  # or np.dot(A, B) or A @ B
print("A dot B (matrix multiplication):\n", C)
print("A * B (element-wise multiplication):\n", A * B)

# numpy uses "dot" for all multiplication of vectors, matrices, in any␣
 ↪dimension, not just
# what mathematicians would call the "dot product"
```

```
A dot B (matrix multiplication):
 [[19 22]
 [43 50]]
A * B (element-wise multiplication):
 [[ 5 12]
 [21 32]]
```

[46]:
```
# Matrix-vector multiplication
Av = A.dot(v)  # or np.dot(A, v) or A @ v
print("A dot v:\n", Av)
```

```
A dot v:
 [29 67]
```

[47]:
```
# You should think of this as a typical column vector, even though it is␣
 ↪printed horizontally
# Why? It's a 1D matrix, which is a vector
print(v)
print(v[0])
print(v[1])
```

```
[ 9 10]
9
10
```

[48]:
```
# Vector dot product (inner product)
inner_product = v.dot(v)  # or np.dot(v, v)
print("v dot v (inner product):", inner_product)
```

```
v dot v (inner product): 181
```

[49]:
```
print("Matrix transpose:\n", A.T)
determinant = np.linalg.det(A)
print("Determinant of A:", determinant)
inverse = np.linalg.inv(A)
print("Inverse of A:\n", inverse)
print("A dot A^(-1) (should be identity):\n", A.dot(inverse))
print("Check if it really is the identity:\n",np.isclose(A.dot(inverse), np.
 ↪eye(2)))  # Check if close to identity matrix
```

```
Matrix transpose:
 [[1 3]
 [2 4]]
Determinant of A: -2.0000000000000004
Inverse of A:
 [[-2.   1. ]
 [ 1.5 -0.5]]
A dot A^(-1) (should be identity):
 [[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
Check if it really is the identity:
 [[ True  True]
 [ True  True]]
```

[50]:
```
# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
```

```
print("\nEigenvalues of A:", eigenvalues)
print("Eigenvectors of A:\n", eigenvectors)

# Solving systems of linear equations: Ax = b
b = np.array([5, 6])
x = np.linalg.solve(A, b)
print("\nSolution to Ax = b:", x)
print("Verification A @ x:", A @ x)  # Should equal b
```

```
Eigenvalues of A: [-0.37228132  5.37228132]
Eigenvectors of A:
 [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]

Solution to Ax = b: [-4.   4.5]
Verification A @ x: [5. 6.]
```

[51]:
```
# 10. BROADCASTING
# ===============

# Broadcasting allows operations between arrays of different shapes

# Example 1: Adding a scalar to an array
A = np.array([[1, 2, 3], [4, 5, 6]])
print("A + 10 =\n", A + 10)  # Scalar is broadcast to array shape
```

```
A + 10 =
 [[11 12 13]
 [14 15 16]]
```

[52]:
```
# Example 2: Adding a vector to each row of a matrix
row_vector = np.array([10, 20, 30])
print("Matrix A:\n", A)
print("Row vector:\n", row_vector)
print("A + row_vector =\n", A + row_vector)
# Broadcasting: row_vector is treated as if it were [[10, 20, 30], [10, 20, 30]]
```

```
Matrix A:
 [[1 2 3]
 [4 5 6]]
Row vector:
 [10 20 30]
A + row_vector =
 [[11 22 33]
 [14 25 36]]
```

[53]:
```
# Example 3: Adding a vector to each column of a matrix
col_vector = np.array([[100], [200]])
print("Matrix A:\n", A)
print("Column vector:\n", col_vector)
print("A + col_vector =\n", A + col_vector)
# Broadcasting: col_vector is treated as if it were [[100, 100, 100], [200,␣
 ↪200, 200]]
```

```
Matrix A:
 [[1 2 3]
 [4 5 6]]
Column vector:
 [[100]
 [200]]
A + col_vector =
 [[101 102 103]
 [204 205 206]]
```

[ ]:
```
# Broadcasting rules are complicated, proceed with caution and always check:
# 1. Arrays are compared from the trailing dimensions
# 2. Dimensions must be equal or one must be 1
# 3. Missing dimensions are treated as 1
```

[54]:
```
# 11. AXIS-BASED OPERATIONS
# =========================

# Many NumPy functions accept an 'axis' parameter to operate along
arr = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

print("Array:\n", arr)
print("sum(arr) = ", np.sum(arr))

# Sum along axis 0 (sum the columns into one row)
col_sums = np.sum(arr, axis=0)
print("\nColumn sums:", col_sums)  # [12 15 18]
```

```
Array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
sum(arr) =  45

Column sums: [12 15 18]
```

[55]:
```
# Sum along axis 1 (sum the rows into one column)
row_sums = np.sum(arr, axis=1)
print("Row sums:", row_sums)  # [6 15 24]
```

```
Row sums: [ 6 15 24]
```

[ ]:
```
# Takes a lot of getting used to which axis is which.

# Axis 0 is the rows. Why?
print("second row:\n",arr[1, :]) # gives the second row
# putting different numbers into the first (0th) dimension of the brackets␣
 ↪accesses different rows

# Axis 1 is the columns. Why?
print("second column:\n", arr[:, 1]) # gives the second column
# putting different numbers into the second (1st) dimension of the brackets␣
 ↪accesses different columns
```

[ ]:
```
# summing along axis 0 means collapsing all ROWS together into one ROW, which␣
 ↪means adding the numbers in each column

# summation along axis 1 means collapsing all COLUMNS together into one COLUMN,␣
 ↪which means adding the numbers in each row
```

[56]:
```
# Mean along axes
col_means = np.mean(arr, axis=0)
row_means = np.mean(arr, axis=1)
print("Matrix:\n", arr)
print("Column means:", col_means)
print("Row means:", row_means)
```

```
Matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Column means: [4. 5. 6.]
Row means: [2. 5. 8.]
```

[57]:
```
# Min/max along axes
print("Matrix:\n", arr)
print("Minimum in each row:", np.min(arr, axis=1))
print("Maximum in each column:", np.max(arr, axis=0))
```

```
Matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Minimum in each row: [1 4 7]
Maximum in each column: [7 8 9]
```

[58]:
```
# The keepdims parameter preserves dimensions
print("Matrix:\n", arr)
sum_keepdims = np.sum(arr, axis=0, keepdims=True)
print("Sum with keepdims=True:\n", sum_keepdims)
print("Shape with keepdims:", sum_keepdims.shape)  # (1, 3)

# Without keepdims
sum_no_keepdims = np.sum(arr, axis=0)
print("Sum without keepdims:\n", sum_no_keepdims)
print("Shape without keepdims:", sum_no_keepdims.shape)  # (3,)

print("\n")
# Now summing columns
sum_keepdims = np.sum(arr, axis=1, keepdims=True)
print("Sum with keepdims=True:\n", sum_keepdims)
print("Shape with keepdims:", sum_keepdims.shape)  # (3, 1)
# Without keepdims
sum_no_keepdims = np.sum(arr, axis=1)
print("Sum without keepdims:\n", sum_no_keepdims)
print("Shape without keepdims:", sum_no_keepdims.shape)  # (3,)
```

```
Matrix:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Sum with keepdims=True:
 [[12 15 18]]
Shape with keepdims: (1, 3)
Sum without keepdims:
 [12 15 18]
Shape without keepdims: (3,)


Sum with keepdims=True:
 [[ 6]
 [15]
 [24]]
Shape with keepdims: (3, 1)
Sum without keepdims:
 [ 6 15 24]
Shape without keepdims: (3,)
```

[59]:
```
# 12. UNIVERSAL FUNCTIONS (UFUNCS)
# ================================

# NumPy's universal functions (ufuncs) operate element-wise on arrays
# They are optimized and much faster than Python loops
```

```
# Example: Element-wise operations using ufuncs
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

# These are all ufuncs
print("Addition:", np.add(a, b))        # [6 8 10 12]
print("Subtraction:", np.subtract(a, b)) # [-4 -4 -4 -4]
print("Multiplication:", np.multiply(a, b)) # [5 12 21 32]
print("Division:", np.divide(a, b))     # [0.2 0.33 0.43 0.5]

# When you do a+b, a-b, etc, you are implicitly already using these
```

```
Addition: [ 6  8 10 12]
Subtraction: [-4 -4 -4 -4]
Multiplication: [ 5 12 21 32]
Division: [0.2        0.33333333 0.42857143 0.5       ]
```

[60]:
```
# Other examples of ufuncs:
print("Square root:", np.sqrt(a))
print("Exponential:", np.exp(a))
print("Natural log:", np.log(a))
print("Sine:", np.sin(a))
print("Cosine:", np.cos(a))
print("Tangent:", np.tan(a))
print("Hyperbolic sine:", np.sinh(a))
print("Hyperbolic cosine:", np.cosh(a))
print("Hyperbolic tangent:", np.tanh(a))
```

```
Square root: [1.         1.41421356 1.73205081 2.        ]
Exponential: [ 2.71828183  7.3890561  20.08553692 54.59815003]
Natural log: [0.         0.69314718 1.09861229 1.38629436]
Sine: [ 0.84147098  0.90929743  0.14112001 -0.7568025 ]
Cosine: [ 0.54030231 -0.41614684 -0.9899925  -0.65364362]
Tangent: [ 1.55740772 -2.18503986 -0.14254654  1.15782128]
Hyperbolic sine: [ 1.17520119  3.62686041 10.01787493 27.2899172 ]
Hyperbolic cosine: [ 1.54308063  3.76219569 10.067662   27.30823284]
Hyperbolic tangent: [0.76159416 0.96402758 0.99505475 0.9993293 ]
```

[61]:
```
# Performance comparison: ufunc vs. loop
import time

large_array = np.random.rand(10_000_000)
print(large_array)


# Using ufunc
startu = time.time()
result_ufunc = np.exp(large_array) # e to the power of 10m things, all at once
```

```
endu = time.time()
print(f"\nUfunc time: {endu - startu:.6f} seconds")

# Using Python loop (much slower!)
import math
startl = time.time()
result_loop = np.zeros_like(large_array) # make an array of zeros the same size␣
 ↪as large_array
for i in range(len(large_array)):
    result_loop[i] = math.exp(large_array[i]) # e to the power of 10m things,␣
 ↪one by one
endl = time.time()
print(f"Loop time: {endl - startl:.6f} seconds")
print(f"Speedup factor: {(endl - startl) / (endu - startu):.1f}x")
```

```
[0.456509   0.68552101 0.79670502 … 0.07308556 0.26015935 0.67267452]

Ufunc time: 0.027297 seconds
Loop time: 1.605944 seconds
Speedup factor: 58.8x
```

[ ]:
```
# 13. ACTIVATION FUNCTIONS FOR NEURAL NETWORKS
# ============================================

# Generate sample data for visualization
x = np.random.randn(6)
print(x)

# Common activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

# Calculate activations
sigmoid_y = sigmoid(x)
tanh_y = tanh(x)
relu_y = relu(x)

print("\nSigmoid function:\n", sigmoid_y)
print("\nTanh function:\n", tanh_y)
print("\nReLU function:\n", relu_y)
```

[ ]:

[62]:
```
# 14. NEURAL NETWORK FORWARD PASS
# ==================================================

# Let's implement a simple neural network forward pass using NumPy

# Network architecture: 2 hidden layers (5 neurons each), 3 output neurons
# Input (2) -> Hidden1 (5) -> Hidden2 (5) -> Output (3)

# Generate a single random input sample
x = np.random.randn(2)  # A single sample with 2 features

# Initialize weights and biases
# First hidden layer
W1 = np.random.randn(5, 2) # Input -> Hidden1
b1 = np.random.randn(5)  # Hidden1 bias
print("Weights from input -> Hidden1:\n", W1)
print("Bias for Hidden1:\n", b1)

# Second hidden layer
W2 = np.random.randn(5, 5)  # Hidden1 -> Hidden2
b2 = np.random.randn(5)  # Hidden2 bias
print("Weights from Hidden1 -> Hidden2:\n", W2)
print("Bias for Hidden2:\n", b2)

# Output layer
W3 = np.random.randn(3, 5) # Hidden2 -> Output
print("Weights from Hidden2 -> Output:\n", W3)
b3 = np.random.randn(3)  # Output bias
```

```
Weights from input -> Hidden1:
 [[ 0.39540412 -1.54457654]
 [ 0.21079528  0.08108668]
 [-1.08421425 -0.30025125]
 [ 0.60049643  1.10687026]
 [ 0.64407532  2.24876093]]
Bias for Hidden1:
 [-1.5079567  -0.76530859  0.09408864  0.59949571 -0.3835397 ]
Weights from Hidden1 -> Hidden2:
 [[-0.962587   -0.4089846   1.35100511 -0.29872797  0.28262419]
 [-0.79924032 -1.92604787 -0.40446541 -0.45967687  1.45522762]
 [-1.35184078 -0.78216472 -0.2253223  -0.13624768 -0.09657366]
 [ 1.12453774 -0.05663148 -0.99379591 -1.65298972 -0.17084719]
 [ 1.60746298 -1.72873137  0.76868029  0.643563   -0.68328433]]
Bias for Hidden2:
 [ 0.41548549  0.21989559  1.59086791  0.85340636 -1.53896785]
Weights from Hidden2 -> Output:
```

```
 [[ 0.82937984  1.00554818 -0.22041615  1.30573366 -0.21235591]
 [-1.50683202 -0.08078727  0.92206944  0.15847318 -0.21280159]
 [-0.75551364  1.18495302 -1.6994446  -1.18202946  0.45552452]]
```

[63]:
```
# Forward pass
print("\nPerforming forward pass for a single sample:")

print("Input data:\n", x)

# First hidden layer with ReLU activation
z1 = W1.dot(x) + b1  # W1 @ x + b1
print("Data in Hidden1 pre-activation:\n", z1)
a1 = np.maximum(0, z1)  # ReLU activation
print("Data in Hidden1 post-activation:\n", a1)

# Second hidden layer with ReLU activation
z2 = W2.dot(a1) + b2  # W2 @ a1 + b2
print("Data in Hidden2 pre-activation:\n", z2)
a2 = np.maximum(0, z2)  # ReLU activation
print("Data in Hidden2 post-activation:\n", a2)

# Output layer (no activation)
z3 = W3.dot(a2) + b3  # W3 @ a2 + b3
print("Data in Output layer:\n", z3)
```

```
Performing forward pass for a single sample:
Input data:
 [0.7239625  0.80927786]
Data in Hidden1 pre-activation:
 [-2.47169054 -0.54707905 -0.93382851  1.92999821  1.90261911]
Data in Hidden1 post-activation:
 [0.         0.         0.         1.92999821 1.90261911]
Data in Hidden2 pre-activation:
 [ 0.37666724  2.10146395  1.14416723 -2.66191796 -1.59692223]
Data in Hidden2 post-activation:
 [0.37666724 2.10146395 1.14416723 0.         0.        ]
Data in Output layer:
 [2.15830766 1.62116441 0.61219735]
```

[ ]:

[ ]: