Numpy Notes for ProgSD Exam

Contents

Chapter 1 Numpy

1.1 Imports

- To work with NumPy effectively, it is essential to import the library using standard conventions.
- The most common imports for NumPy are:
  - `import numpy as np`: This is the standard convention for importing NumPy. The alias `np` is widely recognized and saves typing.
  - `import numpy.random as rnd`: This allows easy access to the `random` module for generating random numbers.

Example 1: Standard NumPy Import.

```
# Import NumPy with standard alias
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Example 2: Importing the Random Module.

```
# Import NumPy and the random module
import numpy as np
import numpy.random as rnd

# Generate a random array of 5 numbers between 0 and 1
random_array = rnd.rand(5)
print(random_array)
```

1.2 Basic Array Creation

- Use `array()` to create NumPy arrays.
- Accepts sequences like lists or tuples.
- Use `dtype` to specify the element type.

Code:

```python
import numpy as np
# Create an array with dtype
    specified
arr = np.array([1, 2, 3, 4],
              dtype=np.float32)
print(arr)
```

Output:

```
[1. 2. 3. 4.]
```

1.3 Dtype Option and Array Creation

- Use `array()` to create arrays from sequences like lists or tuples.
- Specify `dtype` to control the data type, including integers, floats, and complex numbers.
- Functions like `zeros()`, `ones()`, and `arange()` help create arrays programmatically.
- Arrays can contain non-numeric data or be constructed from mixed types.

Code:

```
import numpy as np
# Create a 2x2 array with non-numeric
    data
arr_non_numeric = np.array([['a', 'b'],
                            ['c', 'd']])
print(arr_non_numeric)
```

Output:

```
[['a' 'b']
 ['c' 'd']]
```

Code:

```
import numpy as np
# Create an array from a tuple
    with an interconnected list
arr_tuple = np.array(((1, 2),
    [3, 4]))
print(arr_tuple)
```

Output:

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

Code:

```
import numpy as np
# Create an array with dtype as
    complex
arr_complex = np.array([1, 2, 3],
    dtype=np.complex)
print(arr_complex)
```

Output:

```
[1.+0.j 2.+0.j 3.+0.j]
```

Code:

```
import numpy as np
# Create a range of numbers
    using arange()
arr_range = np.arange(0, 10, 2)
print(arr_range)
```

Output:

```
[0 2 4 6 8]
```

Code:

```
import numpy as np
# Reshape a range of numbers
    into a 2x5 array
arr_reshaped =
    np.arange(10).reshape(2, 5)
print(arr_reshaped)
```

Output:

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

1.4 Generating Numerical Sequences

- `arange()` generates sequences with steps.
- `reshape()` reshapes arrays into new dimensions.
- `linspace()` creates evenly spaced values.

Code:

```python
import numpy as np
# Create a range of values from 0 to 9
arr = np.arange(10)
print(arr)
```

Output:

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

Code:

```python
import numpy as np
# Reshape a linear array into 2x5
arr = np.arange(10).reshape(2, 5)
print(arr)
```

Output:

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

Code:

```python
import numpy as np
# Create 5 equally spaced values between 0 and 1
arr = np.linspace(0, 1, 5)
print(arr)
```

Output:

```
[0.   0.25 0.5  0.75 1.  ]
```

1.5 Arithmetic Operations on Arrays

- Perform element-wise arithmetic operations using operators like `+`, `-`, `*`, and functions like `np.sin()`.
- `np.arange()`: Generate sequences for operations.
- `np.ones()`: Generate an array of ones to use in arithmetic.
- Supports element-wise addition, subtraction, multiplication, and applying mathematical functions like `np.sin()`.

Code:

```
import numpy as np

# Create sequences using
    arange() and ones()
a = np.arange(5)
b = np.ones(5)

# Perform arithmetic operations
print("a + b:")
print(a + b)

print("a - b:")
print(a - b)

print("a * b:")
print(a * b)

# Apply mathematical functions
c = np.sin(a)
print("sin(a):")
print(c)
```

Output:

```
a + b:
[1. 2. 3. 4. 5.]

a - b:
[-1.  0.  1.  2.  3.]

a * b:
[0. 1. 2. 3. 4.]

sin(a):
[ 0.         0.84147098
    0.90929743 0.14112001
    -0.7568025 ]
```

1.6 The Matrix Product

- Use `@` or `np.dot()` for matrix multiplication.
- Key differences between `@` and `np.dot()`:
  - **Operator**: `@` performs matrix multiplication; `np.dot()` performs the dot product or matrix multiplication.
  - **1D Arrays**: `@` treats 1D arrays as vectors for matrix multiplication; `np.dot()` computes the scalar dot product.
  - **Higher-dimensional Arrays**: `@` uses matrix multiplication semantics; `np.dot()` uses dot product semantics.

Code:

```
import numpy as np

# Create matrices
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])

# Show A and B
print("Matrix A:")
print(A)

print("Matrix B:")
print(B)

# Matrix product using @ operator
C_at = A @ B

# Matrix product using np.dot()
C_dot = np.dot(A, B)

print("Result using @:")
print(C_at)

print("Result using np.dot():")
print(C_dot)
```

Output:

```
Matrix A:
[[1 2]
 [3 4]]

Matrix B:
[[2 0]
 [1 2]]

Result using @:
[[ 4  4]
 [10  8]]

Result using np.dot():
[[ 4  4]
 [10  8]]
```

1.7 Transpose, Trace, and Inverse of a Matrix

- `transpose()`: Flips the rows and columns of a matrix.
- `trace()`: Calculates the sum of the diagonal elements.
- `linalg.inv()`: Computes the inverse of a square matrix.

Code:

```
import numpy as np
# Create a 2x2 matrix
A = np.array([[1, 2],
              [3, 4]])
# Transpose
transpose_A = A.transpose()
print(transpose_A)
```

Output:

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

Code:

```
import numpy as np

# Trace of the matrix
trace_A = np.trace(A)
print(trace_A)
```

Output:

```
5
```

Code:

```
import numpy as np

# Inverse of the matrix
inverse_A = np.linalg.inv(A)
print(inverse_A)
```

Output:

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

1.8 Increment and Decrement Operators

- Python does not have ++ or - - operators.
- Use += or -= for incrementing/decrementing values in arrays.
- Operations modify the array in-place.

Code:

```
import numpy as np

# Create an array
arr = np.array([1, 2, 3])
print("Original array:")
print(arr)

# Increment all elements by 2
arr += 2
print("Incremented array:")
print(arr)

# Decrement all elements by 1
arr -= 1
print("Decremented array:")
print(arr)
```

Output:

```
Original array:
[1 2 3]

Incremented array:
[3 4 5]

Decremented array:
[2 3 4]
```

1.9 Shape Manipulation

- `reshape()`: Changes the shape of an array without altering data.
- `ravel()`: Converts a multi-dimensional array to a 1D array.
- `transpose()`: Switches rows and columns.

Example 1: Reshaping an Array
Code:

```
import numpy as np

# Create a 2x3 array
arr = np.array([[1, 2, 3], [4,
    5, 6]])
print("Original array:")
print(arr)

# Reshape to 3x2
reshaped = arr.reshape(3, 2)
print("Reshaped array:")
print(reshaped)
```

Output:

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

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

Example 2: Flattening an Array
Code:

```
import numpy as np

# Flatten the array
flattened = arr.ravel()
print("Flattened array:")
print(flattened)
```

Output:

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

Example 3: Transposing an Array

Code:

```
import numpy as np

# Transpose the array
transposed = arr.transpose()
print("Transposed array:")
print(transposed)
```

Output:

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

1.10 Array Manipulation: Joining Arrays

- `vstack()` and `hstack()` combine arrays vertically or horizontally.
- `column_stack()` and `row_stack()` stack 1D arrays as columns or rows.

Example 1: Vertical Stack (`vstack`)
Code:

```
import numpy as np

# Create two 1D arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

print("Original arrays:")
print("a:", a)
print("b:", b)

# Vertical stack
v_stacked = np.vstack((a, b))
print("Vertical stack (vstack):")
print(v_stacked)
```

Output:

```
Original arrays:
a: [1 2 3]
b: [4 5 6]

Vertical stack (vstack):
[[1 2 3]
 [4 5 6]]
```

Example 2: Horizontal Stack (`hstack`)
Code:

```
import numpy as np

# Create two 1D arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Horizontal stack
h_stacked = np.hstack((a, b))
print("Horizontal stack
    (hstack):")
print(h_stacked)
```

Output:

```
Horizontal stack (hstack):
[1 2 3 4 5 6]
```

Example 3: Column Stack (`column_stack`) Code:

```
import numpy as np

# Create two 1D arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Column stack
c_stacked = np.column_stack((a,
    b))
print("Column stack:")
print(c_stacked)
```

Output:

```
Column stack:
[[1 4]
 [2 5]
 [3 6]]
```

Example 4: Row Stack (`row_stack`) Code:

```
import numpy as np

# Create two 1D arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Row stack
r_stacked = np.row_stack((a, b))
print("Row stack:")
print(r_stacked)
```

Output:

```
Row stack:
[[1 2 3]
 [4 5 6]]
```

1.11 Array Manipulation: Splitting Arrays

- `hsplit()` splits arrays horizontally (columns).
- `vsplit()` splits arrays vertically (rows).
- `split()` divides arrays into custom partitions along any axis.

Code:

```
import numpy as np

# Create a 2x6 array
arr = np.array
([[1, 2, 3, 4, 5, 6],
[7, 8, 9, 10, 11, 12]])

print("Original array:")
print(arr)

# Horizontal split into 3 parts
h_split = np.hsplit(arr, 3)
print("Horizontal split:")
print(h_split)
```

Output:

```
Original array:
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]]

Horizontal split:
[array([[1, 2],
       [7, 8]]),
 array([[ 3, 4],
       [ 9, 10]]),
 array([[ 5, 6],
       [11, 12]])]
```

Code:

```
import numpy as np

# Create a 2x6 array
arr = np.array
([[1, 2, 3, 4, 5, 6],
[7, 8, 9, 10, 11, 12]])

# Vertical split into 2 parts
v_split = np.vsplit(arr, 2)
print("Vertical split:")
print(v_split)
```

Output:

```
Vertical split:
[array([[1, 2, 3, 4, 5, 6]]),
 array([[ 7, 8, 9, 10, 11, 12]])]
```

Code:

```python
import numpy as np
# Create a 2x6 array
arr = np.array
([[1, 2, 3, 4, 5, 6],
7, 8, 9, 10, 11, 12]])

# Custom vertical split
[A1, A2, A3] = np.split(arr,
    [1, 3], axis=0)

print("Custom split:")
print("A1:")
print(A1)

print("A2:")
print(A2)

print("A3:")
print(A3)
```

Output:

```
Custom split:
A1:
[[1 2 3 4 5 6]]
A2:
[[ 7  8  9 10 11 12]]
A3:
[]
```

Custom Split (`split`) Code:

```python
import numpy as np

# Create a 2x6 array
arr = np.array
([[1, 2, 3, 4, 5, 6],
[7, 8, 9, 10, 11, 12]])

# Custom vertical split
[A1, A2, A3] = np.split(arr,
    [1, 3], axis=0)

print("Custom split:")
print("A1:")
print(A1)

print("A2:")
print(A2)

print("A3:")
print(A3)
```

Output:

```
Custom split:
A1:
[[1 2 3 4 5 6]]
A2:
[[ 7  8  9 10 11 12]]
A3:
[]
```

1.12 Array Indexing

- Use square brackets (`[ ]`) to index elements of an array.
- Negative indices count from the end of the array.
- Use pairs of indices `[row, column]` for multi-dimensional arrays.
- Pass an array of indices to select multiple elements at once.
- For 2D arrays (`arr_2d`), the first index specifies the row, and the second index specifies the column.

Example 1: Indexing Elements in a 1D Array
Code:

```
import numpy as np

# Create a 1D array
arr = np.array([10, 20, 30, 40,
    50])

# Access elements using positive
    and negative indices
print("Element at index 1:")
print(arr[1])   # Positive index

print("Element at index -1
    (negative index):")
print(arr[-1]) # Negative index
```

Output:

```
Element at index 1:
20

Element at index -1
    (negative index):
50
```

Example 2: Selecting Multiple Elements in a 1D Array
Code:

```
import numpy as np

# Select multiple elements
print("Elements at indices [0,
    2, -1]:")
print(arr[[0, 2, -1]])
```

Output:

```
Elements at indices [0, 2, -1]:
[10 30 50]
```

Example 3: Indexing Elements in a 2D Array
Code:

```
import numpy as np
# Create a 2D array
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

print("Original 2D array:")
print(arr_2d)

# Access a specific element in
    2D array
print("\nElement at row 1,
    column 2 (0-based
    indexing):")
print(arr_2d[1, 2]) # Access 6
```

Output:

```
Original 2D array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Element at row 1, column 2
    (0-based indexing):
6
```

Example 4: Selecting Multiple Elements in a 2D Array
Code:

```
import numpy as np
# Select multiple elements in 2D
    array
rows = [0, 2] # Rows to access
cols = [1, 0] # Columns to access
print("\nElements at positions
    (0, 1) and (2, 0):")
print(arr_2d[rows, cols]) #
    Access 2 and 7
```

Output:

```
Elements at positions (0, 1)
    and (2, 0):
[2 7]
```

1.13 Array Slicing

- Slicing uses `start:stop:step` syntax inside square brackets.
- Omitting `start`, `stop`, or `step` defaults to the array's bounds or step of 1.
- Supports slicing for rows and columns in multi-dimensional arrays.

Code:

```
import numpy as np

# Create an array
arr = np.array([0, 1, 2, 3, 4,
    5, 6])

# Slice from index 2 to 5
print("Slice from index 2 to 5:")
print(arr[2:6])

# Slice every 2nd element
print("Slice every 2nd element:")
print(arr[::2])
```

Output:

```
Slice from index 2 to 5:
[2 3 4 5]

Slice every 2nd element:
[0 2 4 6]
```

Code:

```
import numpy as np

# Create a 2D array
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Slice first two rows and
    columns 1-2
print("Slice first two rows and
    columns 1-2:")
print(arr_2d[:2, 1:3])
```

Output:

```
Slice first two rows and
    columns 1-2:
[[2 3]
 [5 6]]
```

1.14 Iterating Over Arrays

- Use `for` loops to iterate through rows or elements.
- Use `apply_along_axis()` for axis-wise operations.
- Define custom functions or use `ufuncs` to perform element-wise operations.

Example 1: Iterating Through Rows
Code:

```
import numpy as np

# Create a 2D array
arr = np.array([[1, 2, 3],
                [4, 5, 6]])

# Iterate through rows
print("Iterating through rows:")
for row in arr:
    print(row)
```

Output:

```
Iterating through rows:
[1 2 3]
[4 5 6]
```

Example 2: Element-wise Iteration Using `flat`
Code:

```
import numpy as np

# Create a 2D array
arr = np.array([[1, 2, 3],
                [4, 5, 6]])

# Element-wise iteration
print("Element-wise iteration:")
for val in arr.flat:
    print(val)
```

Output:

```
Element-wise iteration:
1
2
3
4
5
6
```

Example 3: Applying Function Using `apply_along_axis`

Code:

```
import numpy as np

# Create a 2D array
arr = np.array([[20, 22, 24],
               [21, 23, 25],
               [22, 24, 26]])

# Apply numpy mean along axis 0
result_axis0 =
    np.apply_along_axis(np.mean,
    axis=0, arr=arr)

print("Mean along axis 0:")
print(result_axis0)

# Apply numpy mean along axis 1
result_axis1 =
    np.apply_along_axis(np.mean,
    axis=1, arr=arr)
print("Mean along axis 1:")
print(result_axis1)
```

Output:

```
Mean along axis 0:
[21. 23. 25.]

Mean along axis 1:
[22. 23. 24.]
```

1.15 Element-Wise Comparisons

- Use comparison operators (>, <, >=, <=) for element-wise comparisons.
- Functions like `np.all()` and `np.any()` test conditions across elements.
- Element-wise comparison results in a boolean array.

Example 1: Element-Wise Comparison Code:

```
import numpy as np

# Create two arrays
a = np.array([1, 2, 3])
b = np.array([2, 2, 4])

# Element-wise comparison
print("Element-wise comparison
    (a > b):")
print(a > b)
```

Output:

```
Element-wise (a > b):
[False False False]
```

Example 2: Using `np.all()` Code:

```
import numpy as np

# Check if all elements satisfy
    a condition
a = np.array([1, 2, 3])
b = np.array([2, 2, 4])

print("Do all elements of a
    satisfy a <= b?")
print(np.all(a <= b))
```

Output:

```
Do all elements of a
    satisfy a <= b?

True
```

Example 3: Using `np.any()`
Code:

```
import numpy as np

# Check if any element satisfies
    a condition
a = np.array([1, 2, 3])
b = np.array([2, 2, 4])

print("Does any element of a
    satisfy a > b?")
print(np.any(a > b))
```

Output:

```
Does any element of a
    satisfy a > b?

False
```

1.16 Using Matplotlib

- Compute x and y coordinates for points on a sine curve.
- Use `np.arange()` for x values and `np.sin()` for y values.
- Plot the points using `matplotlib.pyplot`.

Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Compute x and y coordinates
x = np.arange(0, 5 * np.pi, 0.2)
y = np.sin(x)

# Plot the sine curve
plt.plot(x, y)
plt.title("Sine Curve")

plt.xlabel("x")
plt.ylabel("sin(x)")

plt.show()
```

Output:

```
[Graph of sine curve
    displayed]
```

1.17 Saving and Loading Arrays

- Use `np.save()` and `np.load()` for binary files.
- Use `np.savetxt()` and `np.loadtxt()` for text files.

Example 1: Saving and Loading Text Files
Code:

```
import numpy as np

# Create an array
arr = np.array([1, 2, 3, 4])
print("Original array:")
print(arr)

# Save array to a text file
np.savetxt('array.txt', arr)

# Load array from text file
loaded_txt_array =
    np.loadtxt('array.txt')

print("Array after loading from
    text file:")
print(loaded_txt_array)
```

Output:

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

Array after loading
    from text file:
[1. 2. 3. 4.]
```

Example 2: Saving and Loading Binary Files
Code:

```
import numpy as np

# Create an array
arr = np.array([5, 6, 7, 8])

print("Original array:")
print(arr)

# Save array to a binary file
np.save('array.npy', arr)

# Load array from binary file
loaded_bin_array =
    np.load('array.npy')

print("Array after loading from
    binary file:")
print(loaded_bin_array)
```

Output:

```
Original array:
[5 6 7 8]

Array after loading
    from binary file:
[5 6 7 8]
```

1.18 Copies or Views of Objects

- Assigning an array creates a view (modifications affect both).
- Use `copy()` to create an independent copy of the array.

Code:

```
import numpy as np

# Create an array
arr = np.array([1, 2, 3])

# Create a view
# Points to 'arr'
view_arr = arr

# Modifying 'arr'
view_arr[0] = 10

print(arr) # Modified due to view

# Create a copy
copy_arr = arr.copy()

# Copied array modified
copy_arr[1] = 20

print(arr) # Unchanged
print(copy_arr) # Changed
```

Output:

```

Original Array:
[1, 2, 3]

Modified Due to View
# arr Array:
[10  2  3]

# arr Array unchanged:
[10  2  3]

# copy_arr Array:
[10 20  3]
```