

Lecture 2 - Part 1: Introduction to numpy and scipy

1 What's Numpy?

- **NumPy** is a Python library used for scientific computing and data analysis.
- It provides efficient tools for working with large arrays and matrices of numerical data. In particular, NumPy provides objects such as n-dimensional arrays.
- Libraries written in lower-level languages, such as C, can operate on data stored in Numpy 'ndarray' without copying any data.
- NumPy also includes a wide range of mathematical functions for performing various mathematical operations, such as linear algebra, Fourier analysis, and statistics.
- NumPy facilitates and optimizes the storage and manipulation of numerical data, especially when dealing with large arrays. This is known as "array-oriented computing".
- In summary, the NumPy module is the basic tool used in all scientific and numerical calculations in Python due to its power, speed, and flexibility.

2 How to install NumPy:

You can use the pip package manager to install NumPy by running the command

```
pip install numpy
```

in the command prompt or Anaconda prompt. # How to import NumPy: To use NumPy in your Python code, you first need to import it using the import statement. You can import NumPy by adding the following line at the top of your Python script or Jupyter Notebook:

```
import numpy as np
```

This statement creates an alias "np" for NumPy, which is a common convention used by most developers working with NumPy. # Documentation - You can use <https://docs.scipy.org/doc/>. - Asking For Help:

```
np.info(np.ndarray.dtype)
```

- For interactive help, you can use the symbol ?, for example:

```
np.array?
```

```
1 Docstring:
```

```
2 array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0, like=None)
```

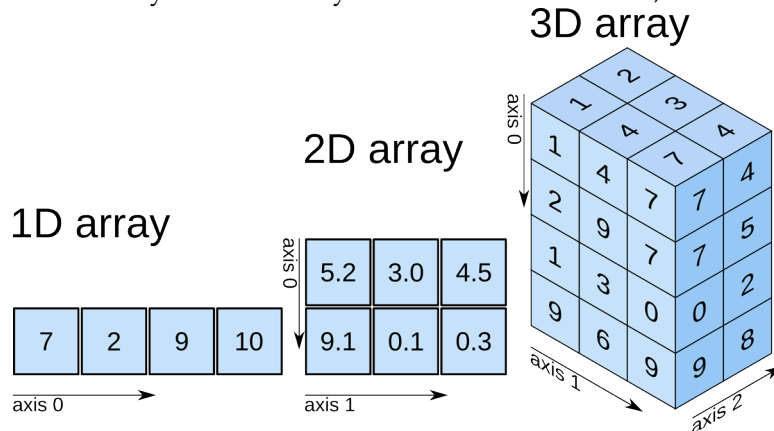
```
3
```

```
4 Create an array.
```

```
5 ...
```

3 NumPy N-dimensional array (ndarray):

- NumPy's ndarray is a multi-dimensional container for homogeneous numerical data, meaning that all elements in the array should be of the same data type.
- An ndarray can have any number of dimensions, and each dimension is called an "axis".



3.1 How to create ndarray:

Ndarrays are created using the `numpy.array` function, which takes a list or tuple of values as input and returns an ndarray object.

```
a = np.array([1,2,3]) #1D array
b = np.array([(1.5,2,3), (4,5,6)], dtype = float) #2D array
c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]], dtype = float) #3D array
```

3.2 Some important NumPy array creation functions:

function	Description
<code>np.zeros</code>	Produce an array of all 0s with the given shape and data type
<code>np.ones</code>	Produce an array of all 1s with the given shape and data type
<code>np.arange</code>	Like the built-in range but returns an ndarray instead of a list
<code>np.linspace</code>	Create an array of evenly spaced values (number of samples)
<code>np.full</code>	Produce an array of the given shape and data type with all values set to the indicated 'fill value';
<code>np.eye</code> or <code>np.identity</code>	Create a square $N \times N$ identity matrix (1s on the diagonal and 0s elsewhere)
<code>np.random.random</code>	Create an array with random values
<code>np.empty</code>	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros

examples:

```
np.zeros((3,4)) #Create an array of zeros
np.ones((2,3,4),dtype=np.int16) #Create an array of ones
np.arange(10,25,5) #Create an array of evenly spaced values (step value)
np.linspace(0,2,9) #Create an array of evenly spaced values (number of samples)
np.full((2,2),7) #Create a constant array
np.eye(2) #Create a 2X2 identity matrix
```

```
np.random.random((2,2)) #Create an array with random values
np.random.randint(2, size=(3,3))
np.empty((3,2)) #Create an empty array
```

3.3 Why not just use Python lists for calculations instead of creating a new type of array using numpy?

There are several reasons for this: - Python lists are very general (also called high-level objects). They can contain any object \Rightarrow dynamic typing. They do not support mathematical operations. - NumPy arrays are statically typed and homogeneous. - The type of the elements is determined when the array is created \Rightarrow no more dynamic typing. - Similarly, the size of the array is fixed at creation \Rightarrow optimized memory storage. - Due to static typing, mathematical functions such as matrix multiplication and addition can be implemented using a compiled language (C and Fortran).

Example:

```
%timeit [i**2 for i in range(1000)]
a = np.arange(1000)
%timeit a**2
```

3.4 Data types in numpy:

NumPy provides several data types that can be used to create arrays.

Type	Code	Description
<i>bool</i>	<i>?</i>	Boolean (True or False) stored as a byte
<i>int8</i>	<i>i1</i>	Byte (-128 to 127)
<i>int16</i>	<i>i2</i>	Integer (-32768 to 32767)
<i>int32</i>	<i>i4</i>	Integer (-2^{31} to $2^{31} - 1$)
<i>int64</i>	<i>i8</i>	Integer (-2^{63} to $2^{63} - 1$)
<i>uint8</i>	<i>u1</i>	Unsigned integer (0 to 255)
<i>uint16</i>	<i>u2</i>	Unsigned integer (0 to 65535)
<i>uint32</i>	<i>u4</i>	Unsigned integer (0 to $2^{32} - 1$)
<i>uint64</i>	<i>u8</i>	Unsigned integer (0 to $2^{64} - 1$)
<i>float16</i>	<i>f2</i>	Half precision float
<i>float32</i>	<i>f4</i>	Single precision float
<i>float64</i>	<i>f8</i>	Double precision float
<i>complex64</i>	<i>c8</i>	Complex number, represented by two 32-bit floats
<i>complex128</i>	<i>c16</i>	Complex number, represented by two 64-bit floats
<i>string_</i>	<i>S</i>	Fixed-length ASCII string type (1 byte per character); for example, to create a string data type with length 10, use 'S10'

Example:

```
a = np.array([1, 2, 3], dtype=?)
print(a)
a = np.array([1, 2, 3], dtype=f4)
print(a)
```

3.5 Getting arrays dimensions and Data type in numpy

- To get the dimensions of an array, we can use the `shape` attribute. For example, for an array `a`, we can get its dimensions as follows:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.shape)
```

- To get the number of dimensions of an array, we can use the `ndim` attribute. For example:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.ndim)
```

- To get the data type of an array, we can use the `dtype` attribute. For example:

```
a = np.array([1, 2, 3], dtype=np.float32)
print(a.dtype)
```

- We can also explicitly convert the data type of an array using the `astype()` method. For example:

```
a = np.array([1, 2, 3], dtype=np.int32)
b = a.astype(np.float32)
print(b.dtype)
```

3.6 Array mathematics in numpy:

3.6.1 Arithmetic Operations

Subtraction, Addition, Division, Multiplication (item per item): NumPy provides element-wise arithmetic operations for arrays. You can perform addition, subtraction, multiplication, and division on arrays by simply using the appropriate operator (+, -, *, /) between them. The operations are performed on each corresponding element of the arrays. For example:

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

```
# Addition
c = a + b
print(c) # Output: [5 7 9]
```

```
# Subtraction
d = a - b
print(d) # Output: [-3 -3 -3]
```

```
# Multiplication
e = a * b
print(e) # Output: [ 4 10 18]
```

```
# Division
f = a / b
print(f) # Output: [0.25 0.4 0.5 ]
```

element-wise mathematical functions: NumPy provides several mathematical functions that operate element-wise on arrays. For example,

```
import numpy as np

a = np.array([1, 4, 9])

# Square root
b = np.sqrt(a)
print(b) # Output: [1.  2.  3.]

# Sine
c = np.sin(a)
print(c) # Output: [ 0.84147098 -0.7568025  0.41211849]

# Cosine
d = np.cos(a)
print(d) # Output: [ 0.54030231 -0.65364362 -0.91113026]
```

3.6.2 Comparison

In NumPy, we can compare arrays element-wise using comparison operators such as `==`, `!=`, `<`, `>`, `<=`, and `>=`. When performing element-wise comparisons, the result is a boolean array of the same shape as the original arrays. For example:

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

print(a == b) # [ True  True False]
print(a != b) # [False False  True]
print(a < b)  # [False False  True]
print(a > b)  # [False False False]
print(a <= b) # [ True  True  True]
print(a >= b) # [ True  True False]
```

we can also perform element-wise comparisons between a scalar value and an array:

```
a = np.array([1, 2, 3])
b = 2

print(a == b) # [False  True False]
print(a != b) # [ True False  True]
print(a < b)  # [ True False False]
print(a > b)  # [False False  True]
print(a <= b) # [ True  True False]
print(a >= b) # [False  True  True]
```

any(), all(), and sum() with condition:

- `all()` returns True if all the elements in the input array evaluate to True. Otherwise, it returns False.
- `any()` returns True if any of the elements in the input array evaluate to True. Otherwise, it returns False.

Here are some examples:

```
import numpy as np
```

```
a = np.array([1, 2, 3, 4])
b = np.array([0, 2, 4, 6])
c = np.array([-1, 2, -3, 4])
```

```
# using all() method
print(np.all(a > 0))    # True
print(np.all(b > 0))    # False
print(np.all(c > 0))    # False
```

```
# using any() method
print(np.any(a == 2))   # True
print(np.any(b == 1))   # False
print(np.any(c == -3))  # True
```

- In NumPy, we can perform sum with condition using the `np.sum()` function with a boolean mask as input. The boolean mask is created using a comparison operator (`>`, `<`, `==`, etc.) on an array, which creates a boolean array of the same shape with True where the condition is met and False where it is not.

```
import numpy as np
```

```
# Create an array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
# Sum all elements that are greater than 5
sum_greater_than_5 = np.sum(arr[arr > 5])
```

```
print(sum_greater_than_5) # Output: 30
```

3.6.3 Arrays statistics functions:

NumPy provides many built-in statistical functions that can be used to perform various statistical operations on arrays. Here are some of the commonly used statistical functions in NumPy:

<i>function</i>	<i>Description</i>
<code>np.mean()</code>	Calculates the arithmetic mean of an array along a specified axis.
<code>np.median()</code>	Computes the median of an array along a specified axis.
<code>np.std()</code>	Computes the standard deviation of an array along a specified axis.
<code>np.var()</code>	Computes the variance of an array along a specified axis.
<code>np.min()</code>	Computes the minimum value of an array along a specified axis.
<code>np.max()</code>	Computes the maximum value of an array along a specified axis.
<code>np.argmin()</code>	Return the index of the minimum value
<code>np.argmax()</code>	Return the index of the maximum value
<code>np.sum()</code>	Computes the sum of all the elements in an array along a specified axis.
<code>np.prod()</code>	Computes the product of all the elements in an array along a specified axis.
<code>np.cumsum()</code>	Computes the cumulative sum of the elements in an array along a specified axis.

Here's an example of how to use some of these functions:

```
import numpy as np

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

print(np.mean(a))           # 5.0
print(np.median(a))         # 5.0
print(np.std(a))            # 2.581988897471611
print(np.var(a))            # 6.666666666666667
print(np.min(a,axis=1))     # [1 4 7]
print(np.max(a))            # 9
print(np.sum(a))            # 45
print(np.prod(a))           # 362880
print(np.cumsum(a))         # [ 1  3  6 10 15 21 28 36 45]
```

3.6.4 Arrays algebraic and form operations:

Dot product The `dot()` function in NumPy performs matrix multiplication or dot product of two arrays.

```
A = np.array([[1, 2], [3, 4], [5, 6]])
B = np.array([[7, 8], [9, 10]])

# Matrix multiplication of A and B
C = np.dot(A, B) # or C = A @ B
print(C)
```

Transposition The `transpose()` function can be called on a NumPy array to create a new array with the rows and columns flipped. For example, the following code creates a 2D array `arr` and then transposes it using the `transpose()` function:

```
arr = np.array([[1, 2], [3, 4]])
transposed_arr = np.transpose(arr)
print(transposed_arr)
```

Alternatively, the T attribute can be used to transpose an array. For example:

```
arr = np.array([[1, 2], [3, 4]])
transposed_arr = arr.T
print(transposed_arr)
```

Reshaping: Changing the shape of an array while maintaining the same number of elements.

```
import numpy as np
a = np.array([[1, 2], [3, 4], [5, 6]])
print("Original array:\n", a)
# Output:
# Original array:
# [[1 2]
#  [3 4]
#  [5 6]]
```

```
b = a.reshape((2, 3))
print("Reshaped array:\n", b)
# Output:
# Reshaped array:
# [[1 2 3]
#  [4 5 6]]
```

Broadcasting: Performing operations on arrays with different shapes and sizes.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([[4], [5], [6]])
c = a + b
print("Result array:\n", c)
# Output:
# Result array:
# [[5 6 7]
#  [6 7 8]
#  [7 8 9]]
```

Concatenation: Joining two or more arrays along a specified axis.

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
c = np.concatenate((a, b), axis=0)
print("Concatenated array:\n", c)
# Output:
```



```
# Concatenated array:
# [[1 2]
#  [3 4]
#  [5 6]]
```

Stacking: Joining two or more arrays along a new axis.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.stack((a, b), axis=0)
print("Stacked array:\n", c)
# Output:
# Stacked array:
# [[1 2 3]
#  [4 5 6]]
```

3.6.5 Splitting:

Splitting an array into smaller arrays along a specified axis.

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
b, c = np.split(a, 2, axis=0)
print("First split array:\n", b)
print("Second split array:\n", c)
# Output:
# First split array:
# [[1 2 3]
#  [4 5 6]]
# Second split array:
# [[ 7  8  9]
#  [10 11 12]]
```

3.6.6 flattened array

`ravel()` is a function in NumPy that returns a flattened array. It means that it returns a 1-dimensional array that contains all the elements of the input array. The returned array is always a copy of the original array, so modifying it does not affect the original array.

Here is an example of how to use `ravel()`:

```
import numpy as np

# Create a 2-dimensional array
arr = np.array([[1, 2], [3, 4]])

# Flatten the array using ravel()
flat_arr = arr.ravel()
```

```
# Print the original and flattened arrays
print("Original array:\n", arr)
print("Flattened array:\n", flat_arr)
```

3.7 Copying Arrays

In NumPy, arrays can be copied in different ways, depending on the desired behavior. Here are some ways to copy arrays in NumPy:

1- **Shallow copy**: In this type of copy, a new array object is created, but the data is not copied. Instead, a reference to the original data is used. Shallow copy can be made using the `view()` method.

2- **Deep copy**: In this type of copy, a completely new array and data are created. Deep copy can be made using the `copy()` method. Example:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = arr1.view()

print(arr1)
print(arr2)

arr2[0] = 0
print("Shallow copy")
print(arr1)
print(arr2)

arr2=arr1.copy()
arr2[2] = 4
print("Deep copy")
print(arr1)
print(arr2)
```

3.8 Sorting Arrays

Sorting an array in numpy can be done using the `sort()` function. By default, it sorts the array in ascending order.

Here is an example:

```
import numpy as np

arr = np.array([3, 2, 1, 4, 5])
arr.sort()

print(arr)
#output
#[1 2 3 4 5]
```

If you want to sort a two-dimensional array, you can use the `sort()` function with the `axis` parameter. By default, it sorts each row independently.

Here is an example:

```
import numpy as np

arr = np.array([[3, 2, 1],
               [6, 5, 4],
               [9, 8, 7]])

arr.sort(axis=1)
#output
#[[1 2 3]
#  [4 5 6]
#  [7 8 9]]
print(arr)
```