42_0_NP_numpy_introduction

December 15, 2023

1 Numpy

1.1 Introduction

NumPy is a Python library created in 2005 that performs numerical calculations. It is generally used for working with arrays.

NumPy also includes a wide range of mathematical functions, such as linear algebra, Fourier transforms, and random number generation, which can be applied to arrays.

1.2 Why NumPy

NumPy means "Numerical Python". It is a Python library that performs numerical calculations. It is generally used for working with arrays.

It's geneerally used for: - Machine Learning - Data Science - Image and Signal Processing - Scientific computing - Quantum computing

1.3 Why NumPy

Some of the major reasons why we should use `NumPy` are:

1. Faster Execution

In Python, we use lists to work with arrays. But when it comes to large array operations, Python lists are not optimized enough.

Numpy arrays are optimized for complex mathematical and statistical operations. Operations on NumPy are up to 50x faster than iterating over native Python lists using loops.

Here're some of the reasons why NumPy is so fast: - Uses specialized data structures called numpy arrays - Uses optimized algorithms written in C

1.4 Why NumPy

Some of the major reasons why we should use `NumPy` are:

2. Used with various Libraries

NumPy is heavly used with various libraries like `Pandas`, `Scipy`, `Scikit-learn`, etc.

1.5 Installation

To install NumPy, use the following command:

```
conda install numpy
```

1.6 Importing NumPy

To use NumPy, we need to import it first. We can import NumPy using the following command:

```
[8]: import numpy as np
```

The code above imports the `numpy` library in our program as an alias `np`.

After this import statement, we can use NumPy functions and objects by calling them with `np`.

1.7 NumPy Array Creation

An array allows us to store a collection of multiple values in a single data structure.

The NumPy array is similar to a list, but with added benefits such as being faster and more memory efficient.

Numpy library provides various methods to work with data. To leverage all those features, we first need to create numpy arrays.

There are multiple techniques to generate arrays in NumPy, and we will explore each of them below.

1.7.1 Create an Array from a List

We can create a NumPy array from a python list using the `array()` function :

```
[ ]: # create a list named list1
     list1 = [2, 4, 6, 8]

     # create numpy array using list1
     array1 = np.array(list1)
     print(array1)

     array2 = np.array([1, 3, 5, "7"])
     print(array2)
```

Note : Unlike `lists` or `series`, arrays can only store data of a similar type.

1.7.2 Create an Array using the np.zeros and np.ones functions

We can create an array of zeros or ones using the `np.zeros` and `np.ones` functions respectively.

```
[13]: z = np.zeros(5)
      print(z)

      o = np.ones(5)
      print(o)
```

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
```

1.7.3 Create an Array using the np.arange function

We can create an array using the `np.arange` function. This function returns an array with evenly spaced values within a given interval.

[14]:
```
import numpy as np

# create an array with values from 0 to 4
array1 = np.arange(5)

print("Using np.arange(5):", array1)

# create an array with values from 1 to 8 with a step of 2
array2 = np.arange(1, 9, 2)

print("Using np.arange(1, 9, 2):",array2)
```

```
Using np.arange(5): [0 1 2 3 4]
Using np.arange(1, 9, 2): [1 3 5 7]
```

In the above example, we have created arrays using the np.arange() function.

- `np.arange(5)` - create an array with 5 elements, where the values range from **0** to **4**
- `np.arange(1, 9, 2)` - create an array with 5 elements, where the values range from **1** to **8** with a step of **2**.

1.7.4 Create an Array using the np.random.rand function

The np.random.rand() function is used to create an array of random numbers.

Let's see an example to create an array of 5 random numbers,

[20]:
```
import numpy as np

# generate an array of 5 random numbers
array1 = np.random.rand(5)

print(array1)
```

```
[0.705403   0.09874786 0.52338173 0.43553374 0.21934742]
```

In the above example, we have used the np.random.rand() function to create an array array1 with 5 random numbers.

Note : This code generates a different output each time we run it.

1.7.5 Create an empty Array using the np.empty function

To create an empty NumPy array, we use the np.empty() function. For example,

[21]:
```
# create an empty array of length 4
array1 = np.empty(4)

print(array1)
```

```
[1.e-323 2.e-323 3.e-323 4.e-323]
```

Here, we have created an empty array of length 4 using the np.empty() function.

If we look into the output of the code, we can see the empty array is actually not empty, it has some values in it.

It is because although we are creating an empty array, NumPy will try to add some value to it. The values stored in the array are arbitrary and have no significance.

1.8 NumPy N-D Array creation

NumPy is not restricted to 1-D arrays, it can have arrays of multiple dimensions, also known as N-dimensional arrays or ndarrays.

An N-dimensional array refers to the number of dimensions in which the array is organized.

An array can have any number of dimensions and each dimension can have any number of elements.

For example, a 2D array represents a table with rows and columns, while a 3D array represents a cube with width, height, and depth.

There are multiple techniques to create N-d arrays in NumPy, and we will explore each of them below.

1.8.1 N-D Array from list of lists

To create an N-dimensional NumPy array from a Python List, we can use the np.array() function and pass the list as an argument.

Create a 2D NumPy Array Let's create a 2D NumPy array with 2 rows and 4 columns using lists.

[23]:
```
# create a 2D array with 2 rows and 4 columns
array1 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(type(array1))
```

```
<class 'numpy.ndarray'>
```

In the above example, we first created a 2D list (list of lists) [[1, 2, 3, 4], [5, 6, 7, 8]] with 2 rows and 4 columns. We then passed the list to the np.array() function to create a 2D array.

Create a 3D NumPy Array Let's say we want to create a 3-D NumPy array consisting of two "slices" where each slice has 3 rows and 4 columns.

Here's how we create our desired 3-D array,

[24]:
```python
import numpy as np

# create a 3D array with 2 "slices", each of 3 rows and 4 columns
array1 = np.array([[[1, 2, 3, 4],
                    [5, 6, 7, 8],
                    [9, 10, 11, 12]],

                    [[13, 14, 15, 16],
                     [17, 18, 19, 20],
                     [21, 22, 23, 24]]])

print(array1)
```

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

 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]]
```

Here, we created a 3D list [list of lists of lists] and passed it to the np.array() function. This creates the 3-D array named array1.

In the 3D list,

- The outermost list contains two elements, which are lists representing the two "slices" of the array. Each slice is a 2-D array with 3 rows and 4 columns.
- The innermost lists represent the individual rows of the 2-D arrays.

Note : In the context of an N-D array, a slice is like a subset of the array that we can take out by selecting a specific range of rows, columns.

1.8.2 Creatin g a N-D Array from scratch

We saw how to create N-d NumPy arrays from Python lists. Now we'll see how we can create them from scratch.

To create multidimensional arrays from scratch we use functions such as

- np.zeros() / np.ones()
- np.arange()
- np.random.rand()

Create N-D Arrays using np.zeros() / np.ones() The np.zeros() function allows us to create N-D arrays filled with all zeros. For example,

[28]:
```python
import numpy as np

# create 2D array with 2 rows and 3 columns filled with zeros
array1 = np.zeros((2, 3))
```

```
print("2-D Array: ")
print(array1)

# create 3D array with dimensions 2x3x4 filled with zeros
array2 = np.ones((2, 3, 4))

print("\n3-D Array: ")
print(array2)
```

```
2-D Array:
[[0. 0. 0.]
 [0. 0. 0.]]

3-D Array:
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
```

In the above example, we have used the np.zeros() function to create a 2-D array and 3-D array filled with zeros respectively.

- np.zeros((2, 3)) - returns a zero filled 2-D array with 2 rows and 3 columns
- np.zeros((2, 3, 4)) - returns a zero filled 3-D array with 2 slices, each slice having 3 rows and 4 columns.

Note : Similarly we can use np.ones() to create an array filled with values 1.

Create N-D Array with a specified value In NumPy, we can use the np.full() function to create a multidimensional array with a specified value.

For example, to create a 2-D array with the value 5, we can do the following:

[31]:
```
import numpy as np

# Create a 2-D array with elements initialized to 5
numpy_array = np.full((2, 2), 5)

print("Array:", numpy_array)
```

```
Array: [[5 5]
 [5 5]]
```

Here, we have used the np.full() function to create a 2-D array where all elements are initialized to 5.

Create N-D Array with np.random.rand() The np.random.rand() function is used to create an array of random numbers.

Let's see an example to create an array of 5 random numbers,

[34]:
```python
import numpy as np

# create a 2D array of 2 rows and 2 columns of random numbers
array1 = np.random.rand(2, 2)

print("2-D Array: ")
print(array1)

# create a 3D array of shape (2, 2, 2) of random numbers
array2 = np.random.rand(2, 2, 2)

print("\n3-D Array: ")
print(array2)
```

```
2-D Array:
[[0.04776108 0.6829359 ]
 [0.85217264 0.04082834]]

3-D Array:
[[[0.24027892 0.54593305]
  [0.28405552 0.72918824]]

 [[0.81406525 0.3360061 ]
  [0.67886112 0.25663121]]]
```

Here,

- np.random.rand(2, 2) - creates a 2D array of 2 rows and 2 columns of random numbers.
- np.random.rand(2, 2, 2) - creates a 3D array with 2 slices, each slice having 2 rows and 2 columns of random numbers.

Create empty N-D Array To create an empty N-D NumPy array, we use the np.empty() function. For example,

[35]:
```python
import numpy as np

# create an empty 2D array with 2 rows and 2 columns
array1 = np.empty((2, 2))

print("2-D Array: ")
print(array1)

# create an empty 3D array of shape (2, 2, 2)
array2 = np.empty((2, 2, 2))
```

```
print("\n3-D Array: ")
print(array2)
```

```
2-D Array:
[[0.04776108 0.6829359 ]
 [0.85217264 0.04082834]]

3-D Array:
[[[0.24027892 0.54593305]
  [0.28405552 0.72918824]]

 [[0.81406525 0.3360061 ]
  [0.67886112 0.25663121]]]
```

In the above example, we used the np.empty() function to create an empty 2-D array and a 3-D array respectively.

If we look into the output of the code, we can see the empty array is actually not empty, it has some values in it.

It is because although we are creating an empty array, NumPy will try to add some value to it. The values stored in the array are arbitrary and have no significance value.

1.9 NumPy Data Types

A data type is a way to specify the type of data that will be stored in an array. For example,

[36]:
```
array1 = np.array([2, 4, 6])
```

Here, the array1 array contains three integer elements, so the data type is Integer(int64)), by default.

NumPy provides us with several built-in data types to efficiently represent numerical data.

1.9.1 NumPy data types

NumPy offers a wider range of numerical data types than what is available in Python. Here's the list of most commonly used numeric data types in NumPy:

- `int8`, `int16`, `int32`, `int64` - signed integer types with different bit sizes
- `uint8`, `uint16`, `uint32`, `uint64` - unsigned integer types with different bit sizes
- `float32`, `float64` - floating-point types with different precision levels
- `complex64`, `complex128` - complex number types with different precision levels

1.9.2 check data type of a NumPy Array

To check the data type of a NumPy array, we use the `dtype` attribute.

[41]:
```
# create an array of integers
array1 = np.array([2, 4, 6])
```

```
# check the data type of array1
print(array1.dtype)

# Output: int64
```

```
int64
```

In the above example, we have used the dtype attribute to check the data type of the array1 array.

Since array1 is an array of integers, the data type of array1 is inferred as int64 by default.

[44]:
```
# create an array of  integers
int_array = np.array([-3, -1, 0, 1])

# create an array of floating-point numbers
float_array = np.array([0.1, 0.2, 0.3])

# create an array of complex numbers
complex_array = np.array([1+2j, 2+3j, 3+4j])

# check the data type of int_array
print(int_array.dtype)  # prints int64

# check the data type of float_array
print(float_array.dtype)  # prints float64

# check the data type of complex_array
print(complex_array.dtype)  # prints complex128
```

```
int64
float64
complex128
```

Here, we have created types of arrays and checked the default data types of these arrays using the dtype attribute.

- int_array - contains four integer elements whose default data type is int64
- float_array - contains three floating-point numbers whose default data type is float64
- complex_array - contains three complex numbers whose default data type is complex128

1.9.3 Creating NumPy Arrays with a defined data type

In NumPy, we can create an array with a defined data type by passing the dtype parameter while calling the np.array() function. For example,

[50]:
```
# create an array of 32-bit integers
array1 = np.array([1, -3, 7], dtype='int32')

print(array1, array1.dtype)
```

```
[ 1 -3  7] int32
```

[54]:
```python
# create an array of 8-bit integers
array1 = np.array([1, 3, 7], dtype='int8')

# create an array of unsigned 16-bit integers
array2 = np.array([2, 4, 6], dtype='uint16')

# create an array of 32-bit floating-point numbers
array3 = np.array([1.2, 2.3, 3.4], dtype='float32')

# create an array of 64-bit complex numbers
array4 = np.array([1+2j, 2+3j, 3+4j], dtype='complex64')

# print the arrays and their data types
print(array1, array1.dtype)
print(array2, array2.dtype)
print(array3, array3.dtype)
print(array4, array4.dtype)
```

```
[1 3 7] int8
[2 4 6] uint16
[1.2 2.3 3.4] float32
[1.+2.j 2.+3.j 3.+4.j] complex64
```

1.9.4 NumPy Type Conversion

In NumPy, we can convert the data type of an array using the astype() method. For example,

[55]:
```python
# create an array of integers
int_array = np.array([1, 3, 5, 7])

# convert data type of int_array to float
float_array = int_array.astype('float')

# print the arrays and their data types
print(int_array, int_array.dtype)
print(float_array, float_array.dtype)
```

```
[1 3 5 7] int64
[1. 3. 5. 7.] float64
```

Here, int_array.astype('float') converts the data type of int_array from int64 to float64 using astype()

1.10 NumPy Array attributes

In NumPy, attributes are properties of NumPy arrays that provide information about the array's shape, size, data type, dimension, and so on.

For example, to get the dimension of an array, we can use the ndim attribute.

There are numerous attributes available in NumPy

1.10.1 Common NumPy Attributes

Here are some of the commonly used NumPy attributes:

Attribute	Description
shape	The shape of the array.
ndim	The number of dimensions in the array.
size	The total number of elements in the array.
dtype	The data type of the array.
itemsize	The size of each element in the array in bytes.
nbytes	The total number of bytes in the array.
T	The transpose of the array.
data	returns the buffer containing actual elements of the array in memory

[56]:
```
# create a 2-D array
array1 = np.array([[2, 4, 6],
                  [1, 3, 5]])

# check the dimension of array1
print(array1.ndim)

# Output: 2
```

```
2
```

In this example, array1.ndim returns the number of dimensions present in array1. As array1 is a 2D array, we got 2 as an output.

### NumPy Array size Attribute

The size attribute returns the total number of elements in the given array.

Let's see an example.

[57]:
```
array1 = np.array([[1, 2, 3],
                 [6, 7, 8]])

# return total number of elements in array1
print(array1.size)

# Output: 6
```

```
6
```

In this example, array1.size returns the total number of elements in the array1 array, regardless of the number of dimensions.

Since these are a total of 6 elements in array1, the size attribute returns 6.

### NumPy Array shape Attribute

In NumPy, the shape attribute returns a tuple of integers that gives the size of the array in each dimension. For example,

[58]:
```
array1 = np.array([[1, 2, 3],
                   [6, 7, 8]])

# return a tuple that gives size of array in each dimension
print(array1.shape)

# Output: (2,3)
```

```
(2, 3)
```

Here, array1 is a 2-D array that has 2 rows and 3 columns. So array1.shape returns the tuple (2,3) as an output.

1.10.2 NumPy Array dtype Attribute

We can use the dtype attribute to check the datatype of a NumPy array. For example,

[59]:
```
# create an array of integers
array1 = np.array([6, 7, 8])

# check the data type of array1
print(array1.dtype)

# Output: int64
```

```
int64
```

In the above example, the dtype attribute returns the data type of array1.

Since array1 is an array of integers, the data type of array1 is inferred as int64 by default.

1.10.3 NumPy Array itemsize Attribute

In NumPy, the itemsize attribute determines size (in bytes) of each element in the array. For example,

[60]:
```
# create a default 1-D array of integers
array1 = np.array([6, 7, 8, 10, 13])

# create a 1-D array of 32-bit integers
array2 = np.array([6, 7, 8, 10, 13], dtype=np.int32)

# use of itemsize to determine size of each array element of array1 and array2
print(array1.itemsize)  # prints 8
print(array2.itemsize)  # prints 4
```

```
8
4
```

Here,

- array1 is an array containing 64-bit integers by default, which uses 8 bytes of memory per element. So, itemsize returns 8 as the size of each element.
- array2 is an array of 32-bit integers, so each element in this array uses only 4 bytes of memory. So, itemsize returns 4 as the size of each element.

1.10.4 NumPy Array data Attribute

In NumPy, we can get a buffer containing actual elements of the array in memory using the data attribute.

In simpler terms, the data attribute is like a pointer to the memory location where the array's data is stored in the computer's memory.

Let's see an example.

[61]:
```
array1 = np.array([6, 7, 8])
array2 = np.array([[1, 2, 3],
                          [6, 7, 8]])

# print memory address of array1's and array2's data
print("\nData of array1 is: ",array1.data)
print("Data of array2 is: ",array2.data)
```

```

Data of array1 is:  <memory at 0x10a0f8b80>
Data of array2 is:  <memory at 0x105b3fb90>
```

Here, the data attribute returns the memory addresses of the data for array1 and array2 respectively.

1.11 NumPy Input Output

NumPy offers input/output (I/O) functions for loading and saving data to and from files.

Input/output functions support a variety of file formats, including binary and text formats.

- The binary format is designed for efficient storage and retrieval of large arrays.
- The text format is more human-readable and can be easily edited in a text editor.

1.11.1 Most Commonly Used I/O Functions

Here are some of the commonly used NumPy Input/Output functions:

Function	Description
np.save()	Saves a NumPy array to a binary file.
np.load()	Loads a NumPy array from a binary file.
np.savetxt()	Saves data in a NumPy array to a text file.
np.loadtxt()	Loads data from a text file and stores it in a NumPy array.

1.11.2 NumPy save() Function

In NumPy, the save() function is used to save an array to a binary file in the NumPy .npy format.

Here's the syntax of the save() function,

`np.save(file, array)`

- `file` - specifies the name of the file to which the array will be saved.
- `array` - specifies the array to be saved.

[63]:
```
# create a NumPy array
array1 = np.array([[1, 3, 5],
                   [7, 9, 11]])

# save the array to a file
np.save('./data/np_save/file1.npy', array1)
```

Here, we saved the NumPy array named array1 to the binary file named file1.npy in our current directory.

1.11.3 NumPy load() Function

In the previous example, we saved an array to a binary file. Now we'll load that saved file using the load() function.

Let's see an example.

[66]:
```
# load the saved NumPy array
loaded_array = np.load('./data/np_save/file1.npy')

# display the loaded array
print((loaded_array.ndim))
```

```
2
```

Here, we have used the load() function to read a binary file named file1.npy. This is the same file we created and saved using save() function in the last example.

NumPy savetxt() Function

In NumPy, we use the savetxt() function to save an array to a text file.

Here's the syntax of the savetxt() function:

`np.save(file, array)`

file - specifies the file name array - specifies the NumPy array to be saved

[68]:
```
# create a NumPy array
array2 = np.array([[1, 3, 5],
                   [7, 9, 11]])

# save the array to a file
```

```
np.savetxt('./data/np_save/file2.txt', array2)
```

The code above will save the NumPy array array2 to the text file named file2.txt in our current directory.

1.11.4 NumPy loadtxt() Function

We use the loadtxt() function to load the saved txt file.

Let's see an example to load the file2.txt file that we saved earlier.

[69]:
```
# load the saved NumPy array
loaded_array = np.loadtxt('./data/np_save/file2.txt')

# display the loaded array
print(loaded_array)
```

```
[[ 1.  3.  5.]
 [ 7.  9. 11.]]
```

Here, we have used the loadtxt() function to load the text file named file2.txt that was previously created and saved using the savetxt() function.

Note : The values in the loaded array have dots . because loadtxt() reads the values as floating point numbers by default.

1.12 Numpy Array Indexing

In NumPy, each element in an array is associated with a number. The number is known as an array index.

Now, we'll see how we can access individual items from the array using the index number.

1.12.1 Access Array Elements Using Index

We can use indices to access individual elements of a NumPy array.

Suppose we have a NumPy array:

`array1 = np.array([1, 3, 5, 7, 9])`

Now, we can use the index number to access array elements as:

array1[0] - to access the first element, i.e. 1 array1[2] - to access the third element, i.e. 5 array1[4] - to access the fifth element, i.e. 9

[70]:
```
array1 = np.array([1, 3, 5, 7, 9])

# access numpy elements using index
print(array1[0])    # prints 1
print(array1[2])    # prints 5
print(array1[4])    # prints 9
```

```
1
5
9
```

Note : Since the last element of array1 is at index 4, if we try to access the element beyond that, say index 5, we will get an index error: IndexError: index 5 is out of bounds for axis 0 with size 5

1.12.2 Modify Array Elements Using Index

We can use indices to change the value of an element in a NumPy array. For example

[71]:
```
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10])

# change the value of the first element
numbers[0] = 12
print("After modifying first element:",numbers)    # prints [12 4 6 8 10]

# change the value of the third element
numbers[2] = 14
print("After modifying third element:",numbers)    # prints [12 4 14 8 10]
```

```
After modifying first element: [12  4  6  8 10]
After modifying third element: [12  4 14  8 10]
```

In the above example, we have modified elements of the numbers array using array indexing.

- numbers[0] = 12 - modifies the first element of numbers and sets its value to 12
- numbers[2] = 14 - modifies the third element of numbers and sets its value to 14

1.12.3 NumPy Negative Array Indexing

NumPy allows negative indexing for its array. The index of -1 refers to the last item, -2 to the second last item and so on.

[72]:
```
# create a numpy array
numbers = np.array([1, 3, 5, 7, 9])

# access the last element
print(numbers[-1])    # prints 9

# access the second-to-last element
print(numbers[-2])    # prints 7
```

```
9
7
```

[ ]:
```
# create a numpy array
numbers = np.array([2, 3, 5, 7, 11])

# modify the last element
```

```
numbers[-1] = 13
print(numbers)    # Output: [2 3 5 7 13]

# modify the second-to-last element
numbers[-2] = 17
print(numbers)    # Output: [2 3 5 17 13]
```

Here, numbers[-1] = 13 modifies the last element to 13 and numbers[-2] = 17 modifies the second-to-last element to 17.

Note : Unlike regular indexing, negative indexing starts from -1 (not 0) and it starts counting from the end of the array

1.12.4 2-D NumPy Array Indexing

Array indexing in NumPy allows us to access and manipulate elements in a 2-D array.

To access an element of array1, we need to specify the row index and column index of the element. Suppose we have following 2-D array,

```
array1 = np.array([[1, 3, 5],
                   [7, 9, 2],
                   [4, 6, 8]])
```

Now, say we want to access the element in the third row and second column we specify the index as:

```
array1[2, 1] # returns 6
```

Since we know indexing starts from 0. So to access the element in the third row and second column, we need to use index 2 for the third row and index 1 for the second column respectively.

[75]:
```
# create a 2D array
array1 = np.array([[1, 3, 5, 7],
                      [9, 11, 13, 15],
                      [2, 4, 6, 8]])



# access the element at the second row and fourth column
element1 = array1[1, 3]  # returns 15
print("4th Element at 2nd Row:",element1)

# access the element at the first row and second column
element2 = array1[0, 1]  # returns 3
print("2nd Element at First Row:",element2)
```

```
4th Element at 2nd Row: 15
2nd Element at First Row: 3
```

1.12.5 Access Row or Column of 2D Array Using Indexing

In NumPy, we can access specific rows or columns of a 2-D array using array indexing.

[ ]:
```python
# create a 2D array
array1 = np.array([[1, 3, 5],
             [7, 9, 2],
             [4, 6, 8]])

# access the second row of the array
second_row = array1[1, :]
print("Second Row:", second_row)  # Output: [7 9 2]

# access the third column of the array
third_col = array1[:, 2]
print("Third Column:", third_col)  # Output: [5 2 8]
```

Here,

- array1[1, :] - access the second row of array1
- array1[:, 2] - access the third column of array1

1.12.6 3-D NumPy Array Indexing

We learned how to access elements in a 2D array. We can also access elements in higher dimensional arrays.

To access an element of a 3D array, we use three indices separated by commas.

- The first index refers to the slice
- The second index refers to the row
- The third index refers to the column.

Note : In 3D arrays, slice is a 2D array that is obtained by taking a subset of the elements in one of the dimensions.

[77]:
```python
# create a 3D array with shape (2, 3, 4)
array1 = np.array(
[
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12]
    ],
    [
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]
    ]
])

# access a specific element of the array
element = array1[1, 2, 1]
```

```
# print the value of the element
print(element)

# Output: 22
```

```
22
```

Here, we created a 3D array called array1 with shape (2, 3, 4). This array contains 2 2D arrays, each with 3 rows and 4 columns.

Then, we used indexing to access a specific element of array1. Notice the code,

```
array1[1, 2, 1]
```

Here,

array1[1, , ,] - access the second 2D array, i.e.

```
[13, 14, 15, 16],
[17, 18, 19, 20],
[21, 22, 23, 24]
```

array1[,2,] - access the third row of the 2D array, i.e.

```
[21, 22, 23, 24]
```

array1[, ,1] - access the second element of the third row, i.e.

```
[22]
```

1.13 NumPy Array Slicing

Array Slicing is the process of extracting a portion of an array.

With slicing, we can easily access elements in the array. It can be done on one or more dimensions of a NumPy array.

Syntax of NumPy Array Slicing

Here's the syntax of array slicing in NumPy:

```
array[start:stop:step]
```

Here,

- start - index of the first element to be included in the slice
- stop - index of the last element (exclusive)
- step - step size between each element in the slice

Note : When we slice arrays, the start index is inclusive but the stop index is exclusive.

- If we omit start, slicing starts from the first element
- If we omit stop, slicing continues up to the last element
- If we omit step, default step size is 1

1.14 1D NumPy Array Slicing

In NumPy, it's possible to access the portion of an array using the slicing operator :. For example

[78]:
```python
# create a 1D array
array1 = np.array([1, 3, 5, 7, 8, 9, 2, 4, 6])

# slice array1 from index 2 to index 6 (exclusive)
print(array1[2:6])  # [5 7 8 9]

# slice array1 from index 0 to index 8 (exclusive) with a step size of 2
print(array1[0:8:2])  # [1 5 8 2]

# slice array1 from index 3 up to the last element
print(array1[3:])  # [7 8 9 2 4 6]

# items from start to end
print(array1[:])   # [1 3 5 7 8 9 2 4 6]
```

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

In the above example, we have created the array named array1 with 9 elements.

Then, we used the slicing operator : to slice array elements.

- array1[2:6] - slices array1 from index 2 to index 6, not including index 6
- array1[0:8:2] - slices array1 from index 0 to index 8, not including index 8
- array1[3:] - slices array1 from index 3 up to the last element
- array1[:] - returns all items from beginning to end

Modify Array Elements Using Slicing

With slicing, we can also modify array elements using:

start parameter stop parameter start and stop parameter start, stop, and step parameter

1. Using start Parameter

[79]:
```python
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify elements from index 3 onwards
numbers[3:] = 20
print(numbers)
```

```
[ 2  4  6 20 20 20]
```

Here, numbers[3:] = 20 replaces all the elements from index 3 onwards with new value 20.

2. Using stop Parameter

[80]:
```
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify the first 3 elements
numbers[:3] = 40
print(numbers)
```

```
[40 40 40  8 10 12]
```

Here, numbers[:3] = 20 replaces the first 3 elements with the new value 40

3. Using start and stop parameter

[81]:
```
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify elements from indices 2 to 5
numbers[2:5] = 22
print(numbers)
```

```
[ 2  4 22 22 22 12]
```

Here, numbers[2:5] = 22 selects elements from index 2 to index 4 and replaces them with new value 22.

4. Using start, stop, and step parameter

[ ]:
```
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify every second element from indices 1 to 5
numbers[1:5:2] = 16
print(numbers)

# Output: [ 2 16  6 16 10 12]
```

In the above example,

```
numbers[1:5:2] = 16
```

modifies every second element from index 1 to index 5 with a new value 16.

1.14.1 NumPy Array Negative Slicing

We can also use negative indices to perform negative slicing in NumPy arrays. During negative slicing, elements are accessed from the end of the array.

[ ]:
```
# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# slice the last 3 elements of the array
# using the start parameter
```

```
print(numbers[-3:])     # [8 10 12]

# slice elements from 2nd-to-last to 4th-to-last element
# using the start and stop parameters
print(numbers[-5:-2])     # [4 6 8]

# slice every other element of the array from the end
# using the start, stop, and step parameters
print(numbers[-1::-2])    # [12 8 4]
```

Here,

- numbers[-3:] - slices last 3 elements of numbers
- numbers[-5:-2] - slices numbers elements from 5th last to 2nd last(excluded)
- numbers[-1::-2] - slices every other numbers elements from the end with step size 2

2D NumPy Array Slicing

A 2D NumPy array can be thought of as a matrix, where each element has two indices, row index and column index.

To slice a 2D NumPy array, we can use the same syntax as for slicing a 1D NumPy array. The only difference is that we need to specify a slice for each dimension of the array.

Syntax of 2D NumPy Array Slicing

`array[row_start:row_stop:row_step, col_start:col_stop:col_step]`

Here,

- row_start,row_stop,row_step - specifies starting index, stopping index, and step size for the rows respectively
- col_start,col_stop,col_step - specifies starting index, stopping index, and step size for the columns respectively

[86]:
```
# create a 2D array
array1 = np.array([[1, 3, 5, 7],
                   [9, 11, 13, 15]])

print(array1[:2, :2])
```

```
[[ 1  3]
 [ 9 11]]
```

Here, the , in [:2, :2] separates the rows of the array.

The first :2 returns first 2 rows i.e., entire array1. This results in

```
[1  3]
```

The second :2 returns first 2 columns from the 2 rows. This results in

```
[9 11]
```

[89]:
```
# create a 2D array
array1 = np.array([[1, 3, 5, 7],
                      [9, 11, 13, 15],
                      [2, 4, 6, 8]])


# slice the array to get the first two rows and columns
subarray1 = array1[:2, :2]

# slice the array to get the last two rows and columns
subarray2 = array1[1:3, 2:4]

# print the subarrays
print("First Two Rows and Columns: \n",subarray1)
print("Last two Rows and Columns: \n",subarray2)
```

```
First Two Rows and Columns:
 [[ 1  3]
 [ 9 11]]
Last two Rows and Columns:
 [[13 15]
 [ 6  8]]
```

Here,

- array1[:2, :2] - slices array1 that starts at the first row and first column (default values), and ends at the second row and second column (exclusive)
- array1[1:3, 2:4] - slices array1 that starts at the second row and third column (index 1 and 2), and ends at the third row and fourth column (index 2 and 3)

## NumPy Array Reshaping

NumPy array reshaping simply means changing the shape of an array without changing its data.

Let's say we have a 1D array.

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

We can reshape this 1D array into N-d array as

```
# reshape 1D into 2D array
# with 2 rows and 4 columns
[[1 3 5 7]
 [2 4 6 8]]

# reshape 1D into 3D array
# with 2 rows, 2 columns, and 2 layers
[[[1 3]
  [5 7]]

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

Here, we can see that the 1D array has been reshaped into 2D and 3D arrays without altering its data.

1.14.2 Syntax NumPy Array Reshaping

The syntax of NumPy array reshaping is

```
np.reshape(array, newshape, order = 'C')
```

Here,

- array - input array that needs to be reshaped,
- newshape - desired new shape of the array
- order (optional) - specifies the order in which the elements of the array should be arranged. By default it is set to 'C'

Note : The order argument can take one of three values: 'C', 'F', or 'A'. We will discuss them later.

1.15 Reshape 1D Array to 2D Array in NumPy

We use the reshape() function to reshape a 1D array into a 2D array. For example,

[102]:
```
array1 = np.array([1, 3, 5, 7, 2, 4, 6, 8])

# reshape a 1D array into a 2D array
# with 2 rows and 4 columns
result = np.reshape(array1, (2, 4), order="C")
print(result)
```

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

In the above example, we have used the reshape() function to change the shape of the 1D array named array1 into the 2D array. Notice the use of the reshape() function,

```
np.reshape(array1, (2, 4))
```

Here, reshape() takes two parameters,

- array1 - array to be reshaped
- (2, 4) - new shape of array1 specified as a tuple with 2 rows and 4 columns.

[113]:
```
# reshape a 1D array into a 2D array
# with 4 rows and 2 columns
array1 = np.array([1, 3, 5, 7, 2, 4, 6, 8])
result1 = np.reshape(array1, (4, 2))
print("With 4 rows and 2 columns: \n",result1)


# reshape a 1D array into a 2D array with a single row
result2 = np.reshape(array1, (1, 8))
print("\nWith a single row: \n",result2)
```

```
With 4 rows and 2 columns:
 [[1 3]
 [5 7]
 [2 4]
 [6 8]]

With a single row:
 [[1 3 5 7 2 4 6 8]]
```

Note :

- We need to make sure that the new shape in reshape() has the same number of elements as the original array, else a ValueError will be raised.
- For example, reshaping an 8 element 1D array into a 2D array of 2 rows and 4 columns is possible, but reshaping it into a 2D array of 3 rows and 3 columns is not possible as that would require 3x3 = 9 elements.

1.16 Reshape 1D Array to 3D Array in NumPy

In NumPy, we can reshape a 1D NumPy array into a 3D array with a specified number of rows, columns, and layers. For example

[92]:
```
# create a 1D array
array1 = np.array([1, 3, 5, 7, 2, 4, 6, 8])

# reshape the 1D array to a 3D array
result = np.reshape(array1, (2, 2, 2))

# print the new array
print("1D to 3D Array: \n",result)
```

```
1D to 3D Array:
 [[[1 3]
  [5 7]]

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

Here, since there are 8 elements in the array1 array, np.reshape(array1, (2, 2, 2)) reshapes array1 into a 3D array with 2 rows, 2 columns and 2 layers.

1.16.1 Flatten N-d Array to 1-D Array Using reshape()

Flattening an array simply means converting a multidimensional array into a 1D array.

To flatten an N-d array to a 1-D array we can use reshape() and pass "-1" as an argument.

[105]:
```
# flatten 2D array to 1D
array1 = np.array([[1, 3], [5, 7], [9, 11]])
result1 = np.reshape(array1, -1)
print("Flattened 2D array:", result1)
```

```python
# flatten 3D array to 1D
array2 = np.array([[[1, 3], [5, 7]], [[2, 4], [6, 8]]])
result2 = np.reshape(array2, -1)
print("Flattened 3D array:", result2)
```

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

Here, reshape(array1, -1) and reshape(array2, -1) convert 2-D array and 3-D array into a 1-D array by collapsing all dimensions.