Creating Numpy Arrays

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the array() function.

In [ ]:
```
import numpy as np

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

print(arr)

print(type(arr))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

In [ ]:
```
import numpy as np

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

print(arr)
```

```
[1 2 3 4 5]
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

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

arr = np.array(42)

print(arr)
```

```
42
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

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

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

print(arr)
```

```
[1 2 3 4 5]
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

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

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

print(arr)
```

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

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

In [ ]:
```
import numpy as np

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

print(arr)
```

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

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

Check Number of Dimensions

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

In [ ]:
```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

In [ ]:
```
import numpy as np

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

print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

NumPy Array Slicing

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start it's considered 0

If we don't pass end it's considered length of array in that dimension

If we don't pass step it's considered 1

In [ ]:
```
import numpy as np

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

print(arr[1:5])
```

```
[2 3 4 5]
```

In [ ]:
```
import numpy as np

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

print(arr[4:])
```

```
[5 6 7]
```

In [ ]:
```
import numpy as np

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

print(arr[:4])
```

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

Negative Slicing

In [ ]:
```
import numpy as np

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

print(arr[-3:-1])
```

```
[5 6]
```

STEP

```
In [ ]: import numpy as np

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

        print(arr[1:5:2])
```

```
[2 4]
```

```
In [ ]: import numpy as np

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

        print(arr[::2])
```

```
[1 3 5 7]
```

Slicing 2-D Arrays

```
In [ ]: import numpy as np

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

        print(arr[1, 1:4])
```

```
[7 8 9]
```

```
In [ ]: import numpy as np

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

        print(arr[0:2, 2])
```

```
[3 8]
```

```
In [ ]: import numpy as np

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

        print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

NumPy Array Shape

The shape of an array is the number of elements in each dimension.

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

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

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

print(arr.shape)
```

```
(2, 4)
```

The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

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

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

print(arr)
print('shape of array :', arr.shape)
```

```
[[[[[1 2 3 4]]]]]
shape of array : (1, 1, 1, 1, 4)
```

Integers at every index tells about the number of elements the corresponding dimension has.

In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

NumPy Array Reshaping

Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

In [ ]:
```
import numpy as np

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

newarr = arr.reshape(4, 3)

print(newarr)
```

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

Reshape From 1-D to 3-D

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

In [ ]:
```
import numpy as np

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

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

```
In [ ]: import numpy as np

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

        newarr = arr.reshape(3, 3)

        print(newarr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-79494d80387a> in <module>
      3 arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
      4 
----> 5 newarr = arr.reshape(3, 3)
      6 
      7 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

Pass -1 as the value, and NumPy will calculate this number for you.

```
In [ ]: import numpy as np

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

        newarr = arr.reshape(2, 2, -1)

        print(newarr)
```

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

 [[5 6]
  [7 8]]]
```

Flattening the arrays

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

We can use reshape(-1) to do this.

```
In [ ]: import numpy as np

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

        newarr = arr.reshape(-1)

        print(newarr)
```

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

NumPy Array Iterating

Iterating 1D Arrays

```
In [ ]: import numpy as np

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

        for x in arr:
          print(x)
```

```
1
2
3
```

Iterating 2-D Arrays

```
In [ ]: import numpy as np

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

        for x in arr:
          print(x)
```

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

If we iterate on a n-D array it will go through n-1th dimension one by one.

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

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

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

for x in arr:
  for y in x:
    print(y)
```

```
1
2
3
4
5
6
```

NumPy Joining Array

Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Concatenating 1D Array

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

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

arr2 = np.array([4, 5, 6])

arr = np.concatenate((arr1, arr2))

print(arr)
```

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

Concatenating 2D Arrays

axis 0

In [ ]:
```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
print('arr1')
print(arr1)

arr2 = np.array([[5, 6], [7, 8]])
print('arr2')
print(arr2)
arr = np.concatenate((arr1, arr2), axis=0)
print('ans')
print(arr)
```

```
arr1
[[1 2]
 [3 4]]
arr2
[[5 6]
 [7 8]]
ans
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

axis 1

In [ ]:
```
import numpy as np

arr1 = np.array([[1, 2], [3, 4]])
print('arr1')
print(arr1)

arr2 = np.array([[5, 6], [7, 8]])
print('arr2')
print(arr2)
arr = np.concatenate((arr1, arr2), axis=1)
print('ans')
print(arr)
```

```
arr1
[[1 2]
 [3 4]]
arr2
[[5 6]
 [7 8]]
ans
[[1 2 5 6]
 [3 4 7 8]]
```

Concatenating 3D Arrays

axis 0

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

arr1 = np.array([[[1, 2], [3, 4]],[[5, 6], [7, 8]]])
print('arr1')
print(arr1)

arr2 = np.array([9,10,11,12,13,14,15,16])

arr2 = arr2.reshape(2,2,2)
print('arr2')

print(arr2)

arr = np.concatenate((arr1, arr2), axis=0)
print('ans')
print(arr)
```

```
arr1
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
arr2
[[[ 9 10]
  [11 12]]

 [[13 14]
  [15 16]]]
ans
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]

 [[13 14]
  [15 16]]]
```

axis 1

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

arr1 = np.array([[[1, 2], [3, 4]],[[5, 6], [7, 8]]])
print('arr1')
print(arr1)

arr2 = np.array([9,10,11,12,13,14,15,16])

arr2 = arr2.reshape(2,2,2)
print('arr2')

print(arr2)

arr = np.concatenate((arr1, arr2), axis=1)
print('ans')
print(arr)
```

```
arr1
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
arr2
[[[ 9 10]
  [11 12]]

 [[13 14]
  [15 16]]]
ans
[[[ 1  2]
  [ 3  4]
  [ 9 10]
  [11 12]]

 [[ 5  6]
  [ 7  8]
  [13 14]
  [15 16]]]
```

axis 2

In [1]:
```
import numpy as np

arr1 = np.array([[[1, 2], [3, 4]],[[5, 6], [7, 8]]])
print('arr1')
print(arr1)

arr2 = np.array([9,10,11,12,13,14,15,16])

arr2 = arr2.reshape(2,2,2)
print('arr2')

print(arr2)

arr = np.concatenate((arr1, arr2), axis = 2)
print('ans')
print(arr)
```

```
arr1
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
arr2
[[[ 9 10]
  [11 12]]

 [[13 14]
  [15 16]]]
ans
[[[ 1  2  9 10]
  [ 3  4 11 12]]

 [[ 5  6 13 14]
  [ 7  8 15 16]]]
```

NumPy Splitting Array

Splitting NumPy Arrays

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

```
In [ ]: import numpy as np

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

        newarr = np.array_split(arr, 3)

        print(newarr)
```

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

If the array has less elements than required, it will adjust from the end accordingly.

```
In [ ]: import numpy as np

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

        newarr = np.array_split(arr, 4)

        print(newarr)
```

```
[array([1, 2]), array([3, 4]), array([5]), array([6])]
```

The return value of the array_split() method is an array containing each of the split as an array.

If you split an array into 3 arrays, you can access them from the result just like any array element:

```
In [ ]: import numpy as np

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

        newarr = np.array_split(arr, 3)

        print(newarr[0])
        print(newarr[1])
        print(newarr[2])
```

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

Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays.

Use the array_split() method, pass in the array you want to split and the number of splits you want to do.

```
In [ ]: import numpy as np

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

        newarr = np.array_split(arr, 3)

        print(newarr)
```

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

The example above returns three 2-D arrays.

Let's look at another example, this time each element in the 2-D arrays contains 3 elements.

```
In [ ]: import numpy as np

        arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17,
        18]])

        newarr = np.array_split(arr, 3)

        print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```

The example above returns three 2-D arrays.

In addition, you can specify which axis you want to do the split around.

The example below also returns three 2-D arrays, but they are split along the row (axis=1).

In [ ]:
```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3, axis=1)

print(newarr)
```

```
[array([[ 1],
       [ 4],
       [ 7],
       [10],
       [13],
       [16]]), array([[ 2],
       [ 5],
       [ 8],
       [11],
       [14],
       [17]]), array([[ 3],
       [ 6],
       [ 9],
       [12],
       [15],
       [18]])]
```

Where function ¶

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

In [ ]:
```
import numpy as np

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

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6]),)
```

The example above will return a tuple: (array([3, 5, 6],)

Which means that the value 4 is present at index 3, 5, and 6.

In [ ]:
```
import numpy as np

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

x = np.where(arr%2 == 0)

print(x)
```

```
(array([1, 3, 5, 7]),)
```

Find the indexes where the values are odd:

```
In [ ]: import numpy as np

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

        x = np.where(arr%2 == 1)

        print(x)
```

```
(array([0, 2, 4, 6]),)
```

NumPy Sorting Arrays

Sorting 1D Arrays

```
In [ ]: import numpy as np

        arr = np.array([3, 2, 0, 1])

        print(np.sort(arr))
```

```
[0 1 2 3]
```

```
In [ ]: import numpy as np

        arr = np.array(['banana', 'cherry', 'apple'])

        print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

```
In [ ]: import numpy as np

        arr = np.array([True, False, True])

        print(np.sort(arr))
```

```
[False  True  True]
```

Sorting 2D Arrays

```
In [ ]: import numpy as np

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

        print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

Sorting 3D Arrays

In [ ]:
```
import numpy as np

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

print(np.sort(arr))
```

```
[[[2 3 4]
  [0 1 5]]

 [[2 3 4]
  [0 1 5]]]
```

Descending Order

In [ ]:
```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr)[::-1])
```

```
[3 2 1 0]
```

In [ ]:
```
import numpy as np

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

print(np.sort(arr)[:, ::-1])
```

```
[[4 3 2]
 [5 1 0]]
```

In [ ]:
```
import numpy as np

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

print(np.sort(arr)[:, :, ::-1])
```

```
[[[4 3 2]
  [5 1 0]]

 [[4 3 2]
  [5 1 0]]]
```