Data Science

Introduction to Numpy

CS202 09.19.22

Numpy

What is numpy?

- NumPy is a Python library used for working with arrays.
- It has functions for working in the domain of linear algebra (using matrices) and fourier transform.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy stands for Numerical Python.

Numpy

Why use numpy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Installation

We can install by typing “pip install numpy” on your terminal where you have python or you can open your Jupyter notebook and type “pip! install numpy”

Once you finish installing numpy you can write the following:

```
import numpy
```

you can also shorten and write:

```
import numpy as np
```

Numpy

Getting started

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.

```
import numpy as np

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

print(arr)

print(type(arr))
```

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:

```
import numpy as np

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

print(arr)
```

Numpy

Dimensions in arrays

A dimension in arrays is one level of array depth (nested arrays).
0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```python
import numpy as np

arr = np.array(42)

print(arr)
```

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.

```python
import numpy as np

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

print(arr)
```

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.

```python
import numpy as np

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

print(arr)
```

Numpy

Dimensions of 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.

```
      import numpy as np

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

      print(arr)
```

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

```
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)
```

An array can have any number of dimensions.

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

Numpy

Accessing elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

To get first element of the array:

```
import numpy as np

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

print(arr[0])
```

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the row represents the dimension and the index represents the column.

To get element in first row second column of array:

```
import numpy as np

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

print('2nd element on 1st row: ', arr[0, 1])
```

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.
Access the third element of the second array of the first array.

```
import numpy as np

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

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

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 its considered 0

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

If we don't pass step its considered 1

```python
import numpy as np

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

print(arr[1:5])

print(arr[4:])

print(arr[:4])

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

Slicing Array

Slicing 2-D arrays.

From the second element, slice elements 1 through 4 (not included).

```
import numpy as np

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

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

Data Types in Numpy

NumPy has some extra data types, and refer to data types with one character, like `i` for integers, `u` for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

Numpy

Data Types

The NumPy array object has a property called `dtype` that returns the data type of the array:

```
import numpy as np

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

print(arr.dtype)
```

We use the `array()` function to create arrays, this function can take an optional argument: `dtype` that allows us to define the expected data type of the array elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

For `i`, `u`, `f`, `S` and `U` we can define size as well.

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

Numpy

Datatypes in numpy

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.

The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like `'f'` for float, `'i'` for integer etc. or you can use the data type directly like `float` for float and `int` for integer.

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

Copy vs. View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

COPY:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)
```

VIEW:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42

print(arr)
print(x)
```

Numpy

Check if array owns its data

Copies *owns* the data, and views *does not own* the data, but how can we check this?
Every NumPy array has the attribute base that returns None if the array owns the data.
Otherwise, the base attribute refers to the original object.

```
import numpy as np

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

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

Shape

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

```
import numpy as np

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

print(arr.shape)
```

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.

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.

Reshape

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 1-D to 2-D:

```
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)
```

We can only reshape 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.

Iterating

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

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

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

In a 2-D array it will go through all the rows.

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

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

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

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

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

Iterating

The function nditer( ) is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic for loops, iterating through each scalar of an array we need to use *n* for loops which can be difficult to write for arrays with very high dimensionality.

```
import numpy as np

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

for x in np.nditer(arr):
  print(x)
```

We can use op_dtypes argument and pass it the expected datatype to change the datatype of elements while iterating.

NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in nditer( ) we pass flags=[‘buffered’].

```
import numpy as np

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

for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
  print(x)
```

Iterating

We can use filtering and followed by iteration.

```
import numpy as np

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

for x in np.nditer(arr[:, ::2]):
  print(x)
```

Enumeration means mentioning sequence number of somethings one by one.

Sometimes we require corresponding index of the element while iterating, the ndenumerate( ) method can be used for those usecases.

```
import numpy as np

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

for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Join

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.

```python
import numpy as np

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

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

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

print(arr)
```

join 2-D arrays along rows(axis=1).

```python
import numpy as np

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

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

Join via stacking

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

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

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

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

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

NumPy provides a helper function: `hstack()` to stack along rows.

```
arr = np.hstack((arr1, arr2))

print(arr)
```

NumPy provides a helper function: `vstack()` to stack along columns.

```
arr = np.vstack((arr1, arr2))

print(arr)
```

NumPy provides a helper function: `dstack()` to stack along height, which is the same as depth.

```
arr = np.dstack((arr1, arr2))

print(arr)
```

Split

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

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

newarr = np.array_split(arr, 3)

print(newarr)
```

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: arr is as above

```
newarr = np.array_split(arr, 3)

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

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.

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

newarr = np.array_split(arr, 3)

print(newarr)
```

The example above returns three 2-D arrays.

Split

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

another example: each element in the 2-D arrays contains 3 elements.

```
newarr = np.array_split(arr, 3)

print(newarr)
```

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).

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

print(newarr)
```

An alternate solution is using hsplit() opposite of hstack()

```
newarr = np.hsplit(arr, 3)

print(newarr)
```

Numpy Array Search

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.

Find the index where the value is 4:

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

x = np.where(arr == 4)

print(x)
```

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.

There is a method called searchsorted() which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

The searchsorted() is meant to be used on a sorted array.

```
arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)
```

The number 7 should be inserted on index 1 to remain the sort order.

The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.

Numpy Array Sort

Sorting means putting elements in an *ordered sequence*.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

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

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

You can also sort arrays of strings, or any other data type:

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

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

If you use the sort() method on a 2-D array, both arrays will be sorted:

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

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