Intro to NumPy

NumPy (short for *Numerical Python*) is a Python library fundamental for scientific computing. It supports a variety of high-level mathematical functions and is broadly used in data science, machine learning, and big data applications. With its help, you will be able to perform linear algebra calculations easily, as well as statistical, logical, and other operations, making use of numerous built-in functions.

Most parts of NumPy that require high execution speed are written in C, which makes the operations much faster than the corresponding ones in Python itself. Owing to its efficiency and convenience, the library has gained vast popularity among data scientists who work with large datasets and need to perform fast computations.

Installation

Firstly, to start working with NumPy, we need to install it, which can be easily done with pip:

```
pip install numpy
```

You can read more about the installation on the official page of the scientific python distribution.

Then, import the library before starting your work:

```
import numpy as np
```

NumPy arrays

The core NumPy object is an n-dimensional **array**, also known as **ndarray**. The simplest way to create a NumPy array is to convert a Python list:

```
first = np.array([1, 2, 3, 4, 5])
print(first)                          # [1 2 3 4 5]
print(type(first))                    # <class 'numpy.nd
```

In the example above, `first` is a one-dimensional array that is treated as a **vector**. As you can see, when printed, it is rendered without commas, as opposed to Python lists.

You can also use not only integers in the list but any objects (strings, lists, etc.) by specifying the `dtype` argument as `object` (Python object, since in this example you have `int`, `str`, and `list` as element types):

```
first_modified = np.array(['1', 2, [3], 4, [5]], dtyp
print(first_modified)  # ['1' 2 list([3]) 4 list([5])
```

Something to consider here is that if there are multiple fundamental Python types in the array, like in `first_modified`, `numpy` will treat them as the same minimal type required to store all objects in the sequence(the `object`), since `numpy` arrays should be homogenous(that is, all elements have the same type).

Similarly, we can create a two-dimensional Numpy array from the corresponding Python list. Two- and more dimensional arrays are treated as **matrices**.

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

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

If you try to create a two-dimensional Numpy array from a list with sublists of two different lengths, you will obtain a one-dimensional array(in the one-dimensional array, there will be only one value in the shape tuple). This occurs because the elements of an array are stored contiguously in memory.

```
1 second_modified = np.array([[1, 2, 3],
2                             [4, 5]],  dtype=object)
3 print(second_modified)  # [list([1, 2, 3]) list([4, 5
4 print(second_modified.shape)  #(2,)
```

NumPy arrays vs Python lists

As you can see, NumPy arrays resemble a Python built-in list data type. However, there are a few crucial differences:

- Unlike Python lists, which can contain objects of different types(a property known as **heterogeneity**), the objects in NumPy arrays with different Python types will be of the **same type** — the `dtype`. This property is referred to as **homogeneity**, where all elements in the array have the same type, and homogeneity is enforced for performance optimization(operations would be inefficient otherwise). It means that you still can have multiple Python datatypes inside a NumPy array, but they all will be of the `object` type we mentioned earlier. By default, the `dtype` will be set to the minimum type required to hold all objects in the sequence.
- NumPy arrays are much more **memory-efficient** and much **faster** than Python lists when working with large datasets due to various optimizations.
- **Arithmetic operations** differ when executed on Python lists or NumPy arrays.

Let's take a look at arithmetic operations that can be applied both to

arrays and to lists. All differences in them can be explained by the fact that Numpy arrays are created for computing and treated as vectors and matrices, while Python lists are a datatype made just to store data.

To illustrate it, we'll create two lists and two arrays containing the same elements:

```
list_a = [1, 2, 3, 4]
array_a = np.array(list_a)

list_b = [11, 22, 33, 44]
array_b = np.array(list_b)
```

First, let's find their sum. The addition of two arrays returns their sum as when we add two vectors.

```
print(array_a + array_b)  # [12 24 36 48]
```

For this reason, we can't add up arrays of different lengths, a `ValueError` will appear.

```
array_c = np.array([111, 222])
print(array_a + array_c)        # ValueError
```

When we try to add a list and an array, the former is converted to an array, so the result is also a sum of vectors.

```
print(list_a + array_a)   # [2 4 6 8]
```

However, when applied to lists, addition just merges them together.

```
print(list_a + list_b)    # [1, 2, 3, 4, 11, 22, 33,
```

Similarly, if we try to multiply a list by `n`, we'll get the list repeated *n* times, while with an array, each element will be multiplied by *n*:

```
print(list_a * 3)   # [1, 2, 3, 4, 1, 2, 3, 4, 1, 2,
print(array_a * 3)  # [3 6 9 12]
```

Learning sizes

There're a number of ways to learn more about an array without printing it.

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

To find out the dimension size, use `shape`. The first number of the output indicates the number of rows and the second — the number of columns if we are talking about 2-dimensional arrays. For a more general case, when we consider a higher number of dimensions, the length of the `arr.shape` tuple will show the number of dimensions, and each element in the tuple will specify the element count for each dimension.

```
print(second.shape)  # (2, 3)
```

In the example above, there are 2 dimensions(`len(second.shape)==second.ndim` is 2), `second.shape[0]` — the first dimension has 2 elements, and `second.shape[1]` tells us that there are 3 elements in the second dimension.

If the NumPy array has only one dimension, the result will be a bit different:

```
print(first.shape)   # (5,)
```

In this case, the first number is not the number of rows but rather the number of elements in the only dimension, and the empty place after

the comma means that there's no second dimension.

The length of the `shape` tuple is the number of dimensions, `ndim`.

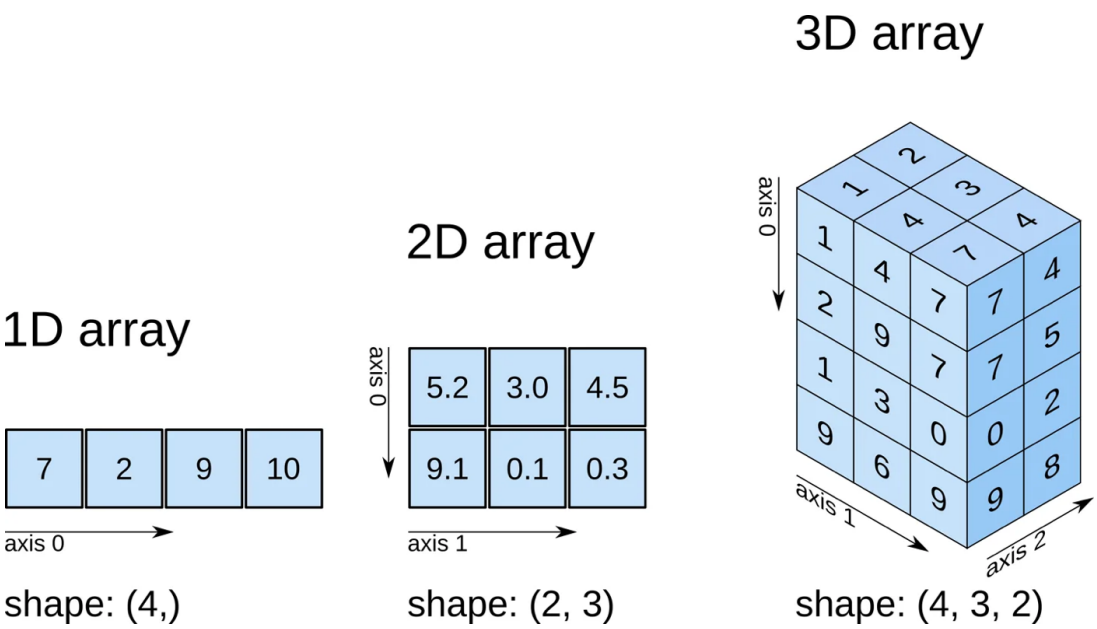
```
print(first.ndim)   # 1
```



```
print(len(first), first.size)    # 5 5
print(len(second), second.size)  # 2 6
```

Note that in the first case they return the same value, while in the second case the numbers differ. The thing is, `len()` works as it would work with regular lists, so if we regard the two-dimensional array above as a list containing two nested lists, it becomes clear that for finding its length only the nested lists are counted. Size, on the contrary, counts each single element in all nested lists.

Another thing to point out is that both the length and the size of an array can also be found from its shape: length is actually the element count in the first dimension, so it equals `shape[0]` (note that here we are talking length like we would see from the `len()` function), and size is the total number of elements, which equals the product of all elements in the shape tuple.



Source: NumPy documentation

This illustration helps to understand the logic behind *there are 4 elements in the first dimension (axis 0), 3 elements in the second dimension (axis 1), and 2 elements in the 3rd dimension (axis 2),* because it looks like the Cartesian coordinates we are all used to. However, NumPy can deal with many more dimensions (these objects are called *tensors*, and vectors and matrices are just the case of a one-dimensions and the two-dimension tensors, respectively), but the 3D case is the simplest one for us to comprehend in terms of visualizations.

You can create and try to run some operations on the 3D array case for the better understanding with the code below:

```
import numpy as np
arr = np.zeros((4, 3 ,2), dtype=int)          #3D arr
rnd_arr = np.random.randint(0, 20, (4, 3 ,2)) #3D arr
```

Conclusion

In this topic, we've learned the basics of NumPy:

- what is NumPy and what it can be used for,
- how to install and import the library,
- arrays, the basic datatype of Numpy,
- the difference between NumPy arrays and Python lists,