# Numerical Python

Shishir Ahmed Saikat

March 1, 2024

# Contents

# Chapter 1

# Introduction

## 1.1 Introduction

NumPy is a Python library. NumPy is used for working with **arrays**.
It also has functions for working in domain of linear algebra, fourier transform and matrices.

### 1.1.1 Why use NumPy

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.

### 1.1.2 Why is NumPy Faster Than List?

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

### 1.1.3 Which Language is NumPy Written in ?

NumPy is a Python library written partially in Python, but most of the parts that required fast computation are written in C or C++ . The source code for NumPy is located at this https://www.github.com/numpy/num
github repository

Creating Arrays

# Chapter 2

# Creating Arrays

## 2.1 Create a ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called *ndarray*We can create a NumPy *ndar*

### 2.1.1 Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays) .We have several types of array in NumPy. They are;

1. 0-D Arrays → Only one item in arroy
2. 1-D Arrays → Most basic and common arrays
3. 2-D Arrays → An array has 1-D arrays as its element. (Nested array in 1-D)
4. 3-D Arrays → An array has 2-D arrays as its element. (Nested array in 1-D)

**For multidimensional array, the number of elements in each nested array must be identical!**
To return the dimension of the array → *ndim*

### 2.1.2 Higher dimensional Arrays

An array can have any number of dimensions.
When the array is created, we can define the number of dimensions bu using the ***ndim*** argument.
**Example**
arr = np.array([1,2,4,5], ndmin=8)

## 2.2 Array Indexing

Array indexing is the same as accessing an array element.
We can access an array element by referring to its index number.
The indexes in NumPy arrays start with 0, meaning that the index of first element is 0, and the 1 is for the second element and up to up...

### 2.2.1 Access Array Elements

1. To access 1-D Arrays → variable[n]
2. To access 2-D Arrays → variable[n,m]
3. To access 3-D Arrays → variable[n,m,l]
4. For negative Indexing → variable[n,-m]

To access 1-D Arrays → variable[n]

## 2.3 Array Slicing

Slicing 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 considerde 1.
When we select a range such that [m:n] the result returns from index m to (n-1) index. But for nested array this rule is not fllowed. In that case, for [m:n, p:q] , it is taken ,for m:n from index **m to (n-1)** and for p:q **p to q**

### 2.3.1 1-D

variable[start:end]
arr = np.array([1,2,3,4,5,6])
print(arr[1:5])
It will return: 2 3 4
print(arr[1:]) (Slice elements from index 1 to the end of the array)
It will return: 2 3 4 5 6
print(arr[:4]) (Slice elements from the beginnig to index 4 (not included))
It will return: 1 2 3

### 2.3.2 Step

If we don't pass step its considered 1.

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

print(arr[::2])
Result: 1 3 5 7 9

### 2.3.3 2-D

arr = np.array([[1,2,3,4,5],[11,12,13,14,15]])

print(arr[0:2,3])

here,
arr[0:2,3] is saying that go throug the array of index 0 and 1 and slice the element of index 3 from each. Hence, 0:2 indicating the index of main array and 3 is indicating the element of nested array and

Result : 4 14

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

It is saying that go through the array of index 0 and 1 and then slice elements from index 1 to 4

Here, Result: [2 3 4] [12 13 14]

## 2.4 Array Copy vs View

The main difference between **copy** and **view** is that the **copy** is a new array, on the other hand **view** is just a view of original array!

Actually, the behaviour of **copy()** and **view** is as natural. The **copy** method just copy the original array and **view()** method just show us the original array. So if we make change in the original array the copied one will **not be affected** and will return the actual array what we copied. But for **view** we will show the changed array!

**copy()** method owns the array but **view()** don't.
Example:

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

newarr = arr.copy()
view = arr.view()
arr[0] = 42

print(arr) → [42 2 3 4 5 0]
print(newarr) → [1 2 3 4 5 0]
print(view) → [42 2 3 4 5 0]
```

To check that an array owns the data from another one → variable.base → it will return **None** if it owns otherwise it will return the original object

# Chapter 3

# Data Types

## 3.1 Data Types in Python

By default Python have these data types;

1. string
2. integer
3. float
4. boolean
5. complex

## 3.2 Data Types in NumPy

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

1. i → integer
2. b → boolean
3. u → unsigned integer
4. f → float
5. c → complex float
6. m → timedelta
7. M → datetime
8. O → object
9. S → string
10. U → unicode string
11. V → fixed chunk of memory for other type (void)

We can define or change the data-type of an array simply writing ***dtype='S'*** after the array
We also can define the size of this data type as well! simply writing ***dtype='S4'*** after the array
We know strings like 'hello, a, B etc.' cannot be converted to integer value. So, if we want cast them to integer, will rise an error called ValueError

Here the best way to change the data type of an existing array, is to make a copy of the array with the ***astype()*** method. It created a copy of data and allows to specify the data type as parameter.
e.g: arr = np.array([[1,2,3,4,5],[11,12,13,14,15]],dtype="S")
newarr = arr.astype("i")

# Chapter 4

# Array Shape

## 4.1 Shape of Array

The shape of an array is the number of elements in each dimension.
To get the shape of an array use the attribute *shape*.
If there is an array like : arr = np.array([[1,2,4,5],[1,4,5,6]]) and if we call it by arr.shape then it will return 2,4 . Here 2 is the number of array and 4 is the dimension.

On the other hand, if we create an array like this,
arr = np.array([1,2,3],ndmin=5) and print arr.shape then it will show us 1,1,1,1,3 . Here the written array will be at last!

## 4.2 Array Reshaping

Reshaping means changing the shape of an array, or convert the dimention.
The shape of an array is the number of elements in each dimension.
By reshaping we can add or remobe dimensions or change number of elements in each dimension.
Note that, it is not a copy a copy it is view (See 2.4)

[1,2 ] → 1-D

[1,2 ,[3,4]] → 2-D

[[1,2 ,[2,3]]] → 3-D

Simply, if we count the number of square brackets at begin , it will tell us the dimension ; )

### 4.2.1 From 1-D to 2-D

Suppose we have 12 elements in an array (1-D), now we want to separate them into two array(2-D). To do that ;
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
newArray = arr.reshape(2,6)
now, if we print newArray it will return us [[ 1 3 5 6 7 8] [ 9 3 45 6 4 5]]
arr.reshape(2,6), here 2 indicates the dimension of main array and 6 indicates the dimension of each nested array.

### 4.2.2 From 1-D to 2-D

It is similar as before, but here we will pass 3 argument into *reshape()* instead of 2 argument.
e.g: arr.reshape(2,3,2)
**Let, we have an array of (x) number of elements and we want to reshape into *n* dimension, but do not want to specify the number of element of each nested array then if we use -1 after defining dimention, NumPy will calculate the number of elements for each nested array!**

**Suppose, we have 16 elements in a single array(1-D), now if we want to convert them into 2-D array, then if we write .reshape(2,-1) then it will return us 2-D array [ 2 nested array, each having 8 elements]**

### 4.2.3 From 1-D to 3-D

It is similar as 4.2.2 , additionally we will define a 3-D array into reshape(3,2,2). If the number of elements into an array is 12 then for .reshape(3,2,2) will return us 3* 2-D array each of 2 elements, which is altogether a 3-D array. // Similar as before, if we do not want to define the number of elements in the nested(last) array we can use -1.

### 4.2.4 Multi-D to 1-D

To convert 2-D/3-D/...n-D array into 1-D array, we can use *.reshape(-1).* It is called Flattening.

**NB:** There are a lot of functions for changing the shapes of arrays in numpy flatten, ravel and also for rearranging the elements rot90, flip, fliplr, flipud etc. These fall under Intermediate to Advanced section of numpy.

**NB:** .reshape(n,m,l) ; n*m*l is must be equal to the number of elements in the array!

# Chapter 5

# Array Iteration

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.

In a 2-D array it will go through all the row also the elements. To get only the rows, use only the first iteration.

In a d-D array it will go through all the row also the elements. To get only the rows, use first two iterations.

| 1-D | 2-D | 3-D |
|---|---|---|
| arr = np.array([1, 2, 3])<br>for x in arr:<br>print(x) | arr = np.array([[1, 2, 3], [4, 5, 6]])<br><br>for x in arr:<br>for y in x:<br>print(y) | arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])<br>for x in arr:<br>for y in x:<br>for z in y<br>print(z) |

Here for multidimensional array we need to write the loop again and again. To overcome this problem, we h ave (.nditer()) method. It go throuugh all the array and return all the element at a time.

## 5.1 Iterate with different Data Type

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'], opdtypes=['S']):
```

```
    print(x)
```

## 5.2 Iterating with Different Step size

Iterate through every scalalar element of the 2D array skipping 1 elements.
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

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

## 5.3 Enumerated Iteration

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.
It returns element mentioining the position no. of each element.

```
    import numpy as np

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

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

# Chapter 6

# Array Joining

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

| Method | Example | Result |
|---|---|---|
| General Concatenate | `print("real concatenate", np.concatenate((arr, arr2)))` | [[1 2] [3 4] [5 6] [7 8]] |
| Concatenate | `print("concatenate:", np.concatenate((arr, arr2), axis=1))` | [[1 2 5 6] [3 4 7 8]] |
| Stack | `print("Stack:", np.stack((arr, arr2), axis=1))` | [[[1 2] [5 6]] [[3 4] [7 8]]] |
| hstack | `print("hstack", np.hstack((arr, arr2)))` | [[1 2 5 6] [3 4 7 8]] |
| vstack | `print("vstack", np.vstack((arr, arr2)))` | [[1 2] [3 4] [5 6] [7 8]] |
| dstack | `print("dstack", np.dstack((arr, arr2)))` | [[[1 5] [2 6]] [[3 7] [4 8]]] |

Table 6.1: Methods and their results

# Chapter 7

# In Brief

1. To check dimension/s → variable.ndim
2. To define the dimension of an Array → ndmin = n
3. To check the datatype of an array → variable.dtype
4. To copy,store and convert data type → newArray = prevArray.astype("i/S/bool...")
5. To access 1-D Arrays → variable[n]
6. To access 2-D Arrays → variable[n,m]
7. To access 3-D Arrays → variable[n,m,l]
8. For negative Indexing → variable[n,-m]
9. Slice 1-D Array → variable[n:m] (from n to m-1)
10. To use Step → variable[n:m:a] (a is step)
11. To use Step → variable[::a] (return element from begining to end after "a" number of step)
12. Slice 2-D Array → variable[n:m,p:q] (from n to m-1 and p to q)
13. To copy an array → variable.copy()
14. To show the original array → variable.view()
15. To check that an array owns the data from another one → variable.base → it will return **None** if it owns otherwise it will return the original object
16. To know the shape/size of an array → variable.shape
17. To reshape 2-D array → var.reshape(2,n) or reshahpe(2,-1)
18. To reshape 3-D array → var.reshape(2,2,n) or reshahpe(2,2,-1)
19. To reshape from multi-D to 1-D → var.reshape(-1)
20. Iterating in Multi-D → for x in np.nditer(arr)
21. Iterating with different step size → for x in np.nditer(arr[:,::2])
22. Enumerated iteration → for idx,x in np.ndenumerate(arr)
23. Iterating and also changing data types → for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S'])

## 7.1 Rights