NUMPY

Complete Revision Notes

RAHUL KUMAR

https://www.linkedin.com/in/rahul-kumar-1212a6141/

Installation Using `%pip`

In [ ]:
```
!pip install numpy
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (1.21.6)
```

Importing Numpy

- We'll import `numpy` as its **alias name** `np` for ease of typing

In [2]:
```
import numpy as np
```

Why use Numpy?

Suppose you are given a list of numbers and you have to find square of each number and store it in original list.

In [3]:
```
a = [1,2,3,4,5]
```

Solution: Basic approach iterate over the list and square each element

In [4]:
```
a = [i**2 for i in a]
print(a)
```

```
[1, 4, 9, 16, 25]
```

Lets try the same operation with NumPy

In [5]:
```
a = np.array([1,2,3,4,5])
print(a**2)
```

```
[ 1  4  9 16 25]
```

The biggest benefit of NumPy is that it supports element-wise operation

Notice how easy and clean is the syntax.

But is the clean syntax and ease in writing the only benefit we are getting here?

- To understand this, lets time these operations
- We will use %timeit to measure the time for operations

In [6]:
```
l = range(1000000)
```

In [7]:
```
%timeit [i**2 for i in l]
```

```
322 ms ± 8.89 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It took approx 300 ms sec per loop to iterate and square all elements from 0 to 999,999

Let's peform same operation using numpy arrays

- We will use `np.array()` method for this.
- `np.array()` simply converts a python array to numpy array.
- We can peform element wise operation using numpy

In [8]:
```
l = np.array(range(1000000))
```

In [9]:
```
%timeit l**2
```

```
1.7 ms ± 43.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Notice per loop time for numpy operation: 1.46 micro sec

What is the major reason behind numpy's faster computation?

- The numpy array is densely packed in memory due to it's homogenous type.
- Numpy is able to divide a task into multiple subtasks and process them parallelly.
- Numpy functions are implemented in C. Which again makes it faster compared to Python Lists.

What is the takeaway from this exercise?

- NumPy provides **clean syntax for providing element-wise operations**
- **Per loop time** for numpy to perform operation is much **lesser than list**

Infact, Numpy is one of the most important packages for performing numerical computations

Why?

Most of computations in DS/ML/DA can be broken down into element-wise operations

Let's create some basic arrays in NumPy

First method we'll see in Numpy is `array()`

- We **pass a Python list** into `np.array()`
- It **converts** that Python list into a **numpy array**

In [10]:
```
# Let's create a 1-D array
arr1 = np.array([1, 2, 3])
print(arr1)
print(arr1 * 2)
```

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

- This is **NOT a normal Python list**
- It's a **numpy array** - supports element-wise operation

Question: What will be the dimension of this array?

1 coz it is a 1D array.

We can get the dimension of array using ndim property

In [11]:
```
arr1.ndim
```

Out[11]:
```
1
```

Numpy arrays have an other property called `shape` which can tell us number of elements across every dimension

We can also get the shape of the array.

In [12]:
```
arr1.shape
```

Out[12]:
```
(3,)
```

Let's take another example to understand `shape` and `ndim` better

In [13]:
```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [10, 11, 12]])
print(arr2)
```

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

What do you think will be the dimension of this 2D array?

In [14]:
```
arr2.ndim
```

Out[14]:
```
2
```

And what about the shape?

In [15]:
```
arr2.shape
```

Out[15]:
```
(3, 3)
```

Lets create some sequences in Numpy

From a range and stepsize - `arange()`

- `np.arange()`
- Similar to `range()`
- We can pass **starting** point, **ending** point (not included in array) and **step-size**
- `arange(start, end, step)`

In [16]:
```
arr2 = np.arange(1, 5)
arr2
```

Out[16]:
```
array([1, 2, 3, 4])
```

In [17]:
```
arr2_stepsize = np.arange(1, 5, 2)
arr2_stepsize
```

Out[17]:
```
array([1, 3])
```

- `np.arange()` behaves in same way as `range()` function

But then why not call it np.range?

- In `np.arange()`, we can pass a **floating point number** as **step-size**

In [18]:
```
arr3 = np.arange(1, 5, 0.5)
arr3
```

Out[18]:
```
array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

Lets check the type of a Numpy array

In [19]:
```
type(arr1)
```

Out[19]:
```
numpy.ndarray
```

But why are we calling it an array? Why not a NumPy list?

How numpy works under the hood?

- It's a **Python Library**, will **write code in Python** to use numpy

However, numpy itself is written in C

Allows numpy to **manage memory very efficiently**

But why is C arrays more efficient or faster than Python Lists?

- In **Python List**, we can **store objects of different types together** - int, float, string, etc.
- The **actual values** of objects are **stored somewhere else in the memory**
- Only **References to those objects** (R1, R2, R3, ...) are stored in the Python List.

- So, when we have to access an element in Python List, we **first access the reference** to that element and then that **reference allows us to access the value** of element stored in memory

C array does all this in one step

- **C array stores objects of same data type together**

- **Actual values** are **stored in same contiguous memory**

- So, when we have to access an element in C array, we **access it directly using indices**.

BUT, notice that this would make NumPy array lose the flexibility to store heterogenous data

==> Unlike Python lists, NumPy array can only hold contigous data

- So numpy arrays are **NOT** really **Python lists**
- They are basically **C arrays**

Let's further see the C type behaviour of Numpy

- For this, lets pass a **floating point number** as one of the values in **np array**

In [20]:
```
arr4 = np.array([1, 2, 3, 4])
arr4
```

Out[20]:
```
array([1, 2, 3, 4])
```

In [21]:
```
arr4 = np.array([1, 2, 3, 4.0])
arr4
```

Out[21]:
```
array([1., 2., 3., 4.])
```

- Notice that `int` **is raised to** `float`
- Because **one single C array** can store values of **only one data type** i.e. homogenous data

- If you press **"Shift+tab"** inside `np.array()` function

- You can see **function's signature**
  - **name**
  - **input parameters**
  - **default values** of input parameters

- Look at `dtype=None`
  - `dtype` means **data-type**
  - which is **set to** `None` **by default**

What if we set `dtype` to `float`?

In [22]:
```
arr5 = np.array([1, 2, 3, 4])
arr5
```

Out[22]:
```
array([1, 2, 3, 4])
```

In [23]:
```
arr5 = np.array([1, 2, 3, 4], dtype="float")
arr5
```

Out[23]:
```
array([1., 2., 3., 4.])
```

Conclusion:

- **"nd" in** `ndarray` **stands for n-dimensional** - `ndarray` means an n-dimensional array

Indexing and Slicing upon Numpy arrays

In [24]:
```
m1 = np.arange(12)
m1
```

Out[24]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Indexing in np arrays

- Works same as lists

In [25]:
```
m1[0] # gives first element of array
```

Out[25]:
```
0
```

In [26]:
```
m1[6] # out of index Error
```

Out[26]:
```
6
```

Question: What will be th output of `m1[-1]` ?

In [27]:
```
m1[-1]
```

Out[27]:
```
11
```

Numpy also supports negative indexing.

You can also use list of indexes in numpy

In [28]:
```
m1 = np.array([100,200,300,400,500,600])
```

In [29]:
```
m1[[2,3,4,1,2,2]]
```

Out[29]:
```
array([300, 400, 500, 200, 300, 300])
```

Did you notice how single index can be repeated multiple times when giving list of indexes?

Slicing

- Similar to Python lists
- We can **slice out and get a part of np array**
- Can also **mix Indexing and Slicing**

In [30]:
```
m1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
m1
```

Out[30]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [31]:
```
m1[:5]
```

Out[31]:
```
array([1, 2, 3, 4, 5])
```

Question: What'll be output of `arr[-5:-1]?`

In [32]:
```
m1[-5:-1]
```

Out[32]:
```
array([6, 7, 8, 9])
```

Question: What'll be the output for `arr[-5:-1: -1]` ?

In [33]:
```
m1[-5: -1: -1]
```

Out[33]:
```
array([], dtype=int32)
```

Fancy indexing (Masking)

- Numpy arrays can be indexed with boolean arrays (masks).
- This method is called fancy indexing.

What would happen if we do this?

In [34]:
```
m1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
m1 < 6
```

Out[34]:
```
array([ True,  True,  True,  True,  True, False, False, False, False,
       False])
```

- **Comparison operation also happens on each element**
- All the **values before 6 return** `True` and **all values after 6 return** `False`

Now, Let's use this to filter or mask values from our array

- **Condition will be passed instead of indices and slice ranges**

In [35]:
```
m1[m1 < 6]
```

Out[35]:
```
array([1, 2, 3, 4, 5])
```

Notice that,

- Value corresponding to True is retained
- Value corresponding to False is filtered out

This is similar to filtering using `filter()` function

```
filter(lambda x: x < 6, [...])
```

How can we filter/mask even values from our array?

In [36]:
```
m1[m1%2 == 0]
```

Out[36]:
```
array([ 2,  4,  6,  8, 10])
```

In [37]:
```
m1[m1%2==0].shape
```

Out[37]:
```
(5,)
```

Question: Multiple conditions in numpy

```
Given an array of elements from 0 to 10, filter the elements which are multiple
of 2 or 5.

a = [0,1,2,3,4,5,6,7,8,9,10]

output should be [0,2,4,5,6,8,10]
```

In [38]:
```
a = np.arange(11)
```

In [39]:
```
a[(a %2 == 0) | (a%5 == 0)]
```

Out[39]:
```
array([ 0,  2,  4,  5,  6,  8, 10])
```

(Optional) Why do we use `& , |` instead of `and, or` keywords for writing multiple condition ?

The difference is that

- `and` and `or` gauge the truth of whole object, whereas
- `&` and `|` are bitwise operator and perform operation on each bit

Recall that everything is treated as object in python.

So, when we use `and` or `or`,

- Python will treat object as single Boolean entity.

In [40]:
```
bool(42)
```

Out[40]:
```
True
```

In [41]:
```
bool(0)
```

Out[41]:
```
False
```

In [42]:
```
bool(42 or 0)
```

Out[42]:
```
True
```

In [43]:
```
bool(42 and 0)
```

Out[43]:
```
False
```

Now, when we apply `&` and `|`, it does bitwise `and` and `or` instead of doing on whole object.

In [44]:
```
bin(42)
```

Out[44]:
```
'0b101010'
```

In [45]:
```
bin(50)
```

Out[45]:
```
'0b110010'
```

In [46]:
```
bin(42 & 50)
```

Out[46]:
```
'0b100010'
```

In [47]:
```
bin(42 | 50)
```

Out[47]:
```
'0b111010'
```

Notice that the bits of objects are being compared to get the result.

In similar fashion, you can think of numpy array with boolean values as string of bits

- where 1 = True
- and 0 = False

In [48]:
```
import numpy as np
```

In [49]:
```
arr = np.array([1, 0, 1, 0, 1, 0], dtype = bool)
arr1 = np.array([1, 1, 0, 0, 1, 0], dtype =bool)
```

In [50]:
```
arr
```

Out[50]:
```
array([ True, False,  True, False,  True, False])
```

In [51]:
```
arr1
```

Out[51]:
```
array([ True,  True, False, False,  True, False])
```

In [52]:
```
arr | arr1
```

Out[52]:
```
array([ True,  True,  True, False,  True, False])
```

Using `and` or `or` on arrays will try to evaulate the condition on entire array which is not defined

(as numpy is made for element wise operation)

In [53]:
```
arr and arr1
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [53], in <cell line: 1>()
----> 1 arr and arr1

ValueError: The truth value of an array with more than one element is ambiguous. Use a.a
ny() or a.all()
```

(Optional) Now, What is the dtype of mask?

It is a boolean array. Hence, it can be treated as string of bits and hence, we use & and | operator on it

(Optional) But why do we use `()` when using multiple conditions?

Remember that the precedence of `&, |` is more than `>, <, ==`.

Let's take an example:

`a %2 == 0 | a%5 == 0`

In above mask, it'll end up evaluating `0 | a&5` first which will throw an error.

Operations on Numpy Arrays

We have already seen operations of a Numpy array and a scalar (single value)

In [54]:
```
arr = np.arange(4)
arr
```

Out[54]:
```
array([0, 1, 2, 3])
```

In [55]:
```
arr + 3
```

Out[55]:
```
array([3, 4, 5, 6])
```

Lets see some algerbraic operations on two arrays

In [56]:
```
# Corresponding elements of arrays get added
a = np.array([1, 2, 3])
b = np.array([2, 2, 2])
a + b
```

Out[56]:
```
array([3, 4, 5])
```

In [57]:
```
# Corresponding elements of arrays get multiplied
a * b
```

Out[57]:
```
array([2, 4, 6])
```

Question: What will be the output of the following ?

In [58]:
```
a = np.array([0,2,3])
b = np.array([1,3,5])
```

In [59]:
```
a*b
```

Out[59]:
```
array([ 0,  6, 15])
```

Numpy will do element wise multiplication

Aggregate / Universal Functions on 1D array (`ufunc`)

Numpy provides various universal functions that cover a wide variety of operations.

For example:

- When **addition of constant to array** is performed **element-wise** using `+` operator, then **np.add() is called internally.**

In [60]:
```
import numpy as np
```

In [61]:
```
a = np.array([1,2,3,4])

a+2  # ufunc `np.add()` called automatically
```

Out[61]:
```
array([3, 4, 5, 6])
```

In [62]:
```
np.add(a,2)
```

Out[62]:
```
array([3, 4, 5, 6])
```

- These functions **operate on ndarray (N-dimensional array) i.e Numpy's array class.**
- They perform **fast element-wise array operations**.

Aggregate Functions/ Reduction functions

Now, how would calculate the sum of elements of an array?

`np.sum()`

- It **sums all the values in np array**

In [63]:
```
a = np.arange(1, 11)
a
```

Out[63]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [64]:
```
np.sum(a)  # sums all the values present in array
```

Out[64]:
```
55
```

Now, What if we want to find the average value or median value of all the elements in an array?

`np.mean()`

- `np.mean()` gives **mean of all values in np array**

In [65]:
```
np.mean(a)
```

Out[65]:
```
5.5
```

Now, we want to find the minimum value in the array

`np.min()` function can help us with this

In [66]:
```
a
```

Out[66]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [67]:
```
np.min(a)
```

Out[67]:
```
1
```

We can also find max elements in an array.

`np.max()` function will give us *maximum value in the array*

In [68]:
```
a
```

Out[68]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [69]:
```
np.max(a) # maximum value
```

Out[69]:
```
10
```

Usecase: AirBnB NPS

Imagine you are a Data Analyst @ Airbnb

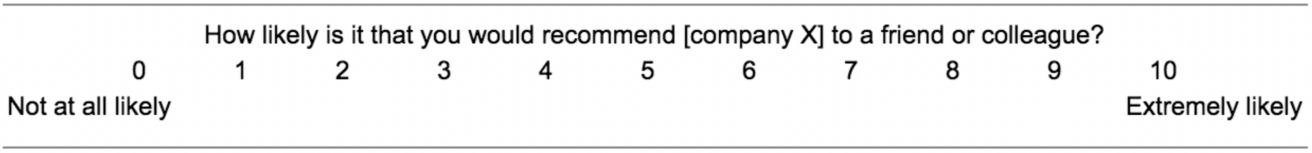
You've been asked to analyze user survey data and report NPS to the management

But, what exactly is NPS?

Have you seen something like this ?

Link: https://drive.google.com/file/d/1-u8e-v_90JdikorKsKzBM-JJqoRtzsN8/view?usp=sharing

---



This is called **Likelyhood to Recommend Survey**

- Responses are given a scale ranging from 0–10,
  - with 0 labeled with “Not at all likely,” and
  - 10 labeled with “Extremely likely.”

Based on this, we calculate the Net Promoter score

How to calculate NPS score?

We label our responses into 3 categories:

- Detractors: Respondents with a score of 0-6
- Passive: Respondents with a score of 7-8
- Promoters: score of 9-10.

And

```
Net Promoter score = % Promoters - % Detractors.
```

How is NPS helpful?

Why would we want to analyse the survey data for NPS?

NPS helps a brand in gauging its brand value and sentiment in the market.

- Promoters are highly likely to recommend your product or sevice. Hence, bringing in more business
- whereas, Detractors are likely to recommend against your product or service’s usage. Hence, bringing the business down.

These insights can help business make customer oriented decision along with product improvisation.

Two third of Fortune 500 companies use NPS

Lets first look at the data we have gathered

Dataset: https://drive.google.com/file/d/1c0ClC8SrPwJq5rrkyMKyPn80nyHcFikK/view?usp=sharing

Notice that the file contains the score for likelyhood to recommend survey

Using NumPy

- we will bin our data into promoters/detractors
- calulate the percentage of promoters/detractors
- calculate NPS

Let's first download the dataset

In [70]:
```
import numpy as np
```

In [71]:
```
!gdown 1c0ClC8SrPwJq5rrkyMKyPn80nyHcFikK
```

```
Downloading...
From: https://drive.google.com/uc?id=1c0ClC8SrPwJq5rrkyMKyPn80nyHcFikK
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\survey.txt

  0%|          | 0.00/2.55k [00:00<?, ?B/s]
100%|##########| 2.55k/2.55k [00:00<?, ?B/s]
```

Let's load the data we saw earlier. For this we will use `.loadtxt() function`

Documentation: https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html

In [72]:
```
score = np.loadtxt('survey.txt', dtype ='int')
```

We provide file name along with the dtype of data we want to load in

Let's see what the data looks like

In [73]:
```
score[:5]
```

Out[73]:
```
array([ 7, 10,  5,  9,  9])
```

Let's check the number of responses

In [74]:
```
score.shape
```

Out[74]:
```
(1167,)
```

There are a total of 1167 responses for the LTR survey

Let's perform some sanity check on data

Let's check the minimum and max value in array

In [75]:
```
score.min()
```

Out[75]:
```
1
```

In [76]:
```
score.max()
```

Out[76]:
```
10
```

Looks like, there are no records with 0 score.

Now, let's calculate NPS using these response.

NPS = % Promoters - % Detractors

Now, in order to calculate NPS, we need to calculate two things:

- % Promoters
- % Detractors

In order to calculate % Promoters and % Detractors, we need to get the count of promoter as well as detractor.

Question: How can we get the count of Promoter/ Detractor ?

We can do so by using fancy indexing (masking)

Let's get the count of promoter and detractors

Detractors have a score <=6

In [77]:
```
detractors = score[score <= 6].shape[0]
```

In [78]:
```
total = score.shape[0]
```

In [79]:
```
percent_detractors = detractors/total*100
```

In [80]:
```
percent_detractors
```

Out[80]:
```
28.449014567266495
```

Similarly, Promoters have a score 9-10

In [81]:
```
promoters = score[score >= 9].shape[0]
```

In [82]:
```
percent_promoters = promoters/total*100
```

In [83]:
```
percent_promoters
```

Out[83]:
```
52.185089974293064
```

Calculating NPS

For calculating NPS, we need to

`% promoters - % detractors`

In [84]:
```
nps = percent_promoters - percent_detractors
nps
```

Out[84]:
```
23.73607540702657
```

In [85]:
```
np.round(nps)
```

Out[85]:
```
24.0
```

Working with 2-D arrays (Matrices)

Question : How do we create a 2D matrix using numpy?

In [86]:
```
m1 = np.array([[1,2,3],[4,5,6]])
m1
```

Out[86]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

How can we check shape of a numpy array?

In [87]:
```
m1.shape
```

Out[87]:
```
(2, 3)
```

Question: What is the type of this result of `arr1.shape`? Which data structure is this?

Tuple

Now, What is the dimension of this array?

In [88]:
```
m1.ndim
```

Out[88]:
```
2
```

Question

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

b = len(a)
```

What'll be the value of b?

```
Ans: 3
```

Explanation: len(nD array) will give you magnitude of first dimension

In [89]:
```
a = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
```

In [90]:
```
a
```

Out[90]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [91]:
```
len(a)
```

Out[91]:
```
3
```

What will be the shape of array `a`?

In [92]:
```
a.shape
```

Out[92]:
```
(3, 3)
```

- So, it is a **2-D array** with **3 rows and 3 columns**

Clearly, if **we have to create high-dimensional arrays, we cannot do this using** `np.arange()` directly

How can we create high dimensional arrays?

- Using `reshape()`

For a 2D array

- **First argument** is **no. of rows**
- **Second argument** is **no. of columns**

In [93]:
```
m2 = np.arange(1, 13)
m2
```

Out[93]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

- We can pass the **desired dimensions** of array in `reshape()`

In what ways can we convert this array with 12 values into high-dimensional array?

Can we make `m2` a 4×4 array?

- Obviously NO
- 4×4 **requires 16 values**, but **we only have 12 in** `m2`

In [94]:
```
m2 = np.arange(1, 13)
m2.reshape(4, 4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [94], in <cell line: 2>()
      1 m2 = np.arange(1, 13)
----> 2 m2.reshape(4, 4)

ValueError: cannot reshape array of size 12 into shape (4,4)
```

So, What are the ways in which we can reshape it?

- 4×3
- 3×4
- 6×2
- 2×6
- 1×12
- 12×1

In [95]:
```
m2 = np.arange(1, 13)
m2.reshape(4, 3)
```

Out[95]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [96]:
```
m2 = np.arange(1, 13)
m2
```

Out[96]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [97]:
```
m2.shape
```

Out[97]:
```
(12,)
```

Lets do some reshaping here

In [98]:
```
m2.reshape(12, 1)
```

Out[98]:
```
array([[ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10],
       [11],
       [12]])
```

Now, What's the difference b/w `(12,)` and `(12, 1)` ?

- `(12,)` means its a **1D array**
- `(12, 1)` means its a **2D array**

Question

What will be output for the following code?

```
a = np.array([[1,2,3],[0,1,4]])
print(a.ndim)
```

Ans: 2

In [99]:
```
a = np.array([[1,2,3],[0,1,4]])
print(a.ndim)
```

```
2
```

Since it is a 2 dimensional array, the number of dimension will be 2.

Transpose

- **Change rows into columns and columns into rows**
- Just use `<Matrix>.T`

In [100…
```
a = np.arange(3)
a
```

Out[100]:
```
array([0, 1, 2])
```

In [101…
```
a.T
```

Out[101]:
```
array([0, 1, 2])
```

Why did Transpose did not work?

- Because **numpy sees `a` as a vector (3,), NOT a matrix**
- We'll have to **reshape the vector `a` to make it a matrix**

In [102…
```
a = np.arange(3).reshape(1, 3)
a
# Now a has dimensions (1, 3) instead of just (3,)
# It has 1 row and 3 columns
```

Out[102]:
```
array([[0, 1, 2]])
```

In [103…
```
a.T
```

Out[103]:
```
array([[0],
       [1],
       [2]])
```

Conclusion

- **Transpose works only on matrices**

Flattening of an array

What if we want to convert this 2D or nD array back to 1D array?

There is a function named `flatten()` to help you do so.

In [104…
```
A = np.arange(12).reshape(3, 4)
A
```

Out[104]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [105…
```
A.flatten()
```

Out[105]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

Indexing and Slicing on 2D

Indexing in np arrays

- Works same as lists

In [106…
```
m1 = np.arange(1,10).reshape((3,3))
m1
```

Out[106]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [107…
```
m1[1][2]
```

Out[107]:
```
6
```

OR

- We just use [0, 0] (**indexes separated by commas**)

What will be the output of this?

In [108…
```
m1[1, 1] #m1[row, column]
```

Out[108]:
```
5
```

We saw how we can use list of indexes in numpy array

In [109…
```
m1 = np.array([100,200,300,400,500,600])
```

In [110…
```
m1[[2,3,4,1,2,2]]
```

Out[110]:
```
array([300, 400, 500, 200, 300, 300])
```

How'll list of indexes work in 2D array ?

In [111…
```
m1 = np.arange(9).reshape((3,3))
m1
```

Out[111]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [112…
```
m1[[0,1,2],[0,1,2]] # picking up element (0,0), (1,1) and (2,2)
```

Out[112]:
```
array([0, 4, 8])
```

Slicing Need to provide two slice ranges - one for row and one for column Can also mix Indexing and Slicing

In [113…
```
m1 = np.arange(12).reshape(3,4)
m1
```

Out[113]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [114…
```
m1[:2] # gives first two rows
```

Out[114]:
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

How can we get columns from 2D array?

In [115…
```
m1[:, :2] # gives first two columns
```

Out[115]:
```
array([[0, 1],
       [4, 5],
       [8, 9]])
```

Question: Given an 2-D array

```
m1 = [[0,1,2,3],
      [4,5,6,7],
      [8,9,10,11]]
```

In [116…
```
# First get rows 1 to all
# Then get columns 1 to 3 (not included)
m1[1:, 1:3]
```

Out[116]:
```
array([[ 5,  6],
       [ 9, 10]])
```

In [117…
```
# Get all rows
# Then get columns from 1 to all with step of 2

m1[:, 1::2]
```

Out[117]:
```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

- **We can also pass indices of required columns as a Tuple** to get the same result

In [118…
```
# Get all rows
# Then get columns 1 and 3

m1[:, (1,3)]
```

Out[118]:
```
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])
```

Fancy indexing (Masking)

What would happen if we do this?

In [119…
```
m1 = np.arange(12).reshape(3, 4)
m1 < 6
```

Out[119]:
```
array([[ True,  True,  True,  True],
       [ True,  True, False, False],
       [False, False, False, False]])
```

- A **matrix having boolean values** `True` and `False` is returned
- **We can use this boolean matrix to filter our array**

Now, Let's use this to filter or mask values from our array

- **Condition will be passed instead of indices and slice ranges**

In [120…
```
m1[m1 < 6]
```

Out[120]:
```
array([0, 1, 2, 3, 4, 5])
```

How can we filter/mask even values from our array?

In [121…
```
m1[m1%2 == 0]
```

Out[121]:
```
array([ 0,  2,  4,  6,  8, 10])
```

But did you notice that matrix gets converted into a 1D array after masking?

In [122…
```
m1
```

Out[122]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [123…
```
m1[m1%2 == 0]
```

Out[123]:
```
array([ 0,  2,  4,  6,  8, 10])
```

It happens because

- To retain matrix shape, it **has to retain all the elements**
- It **cannot retain its 3×4 with lesser number of elements**
- So, this filtering operation **implicitly converts high-dimensional array into 1D array**

If we want, we can reshape the resulting 1D array into 2D

- But, we need to know **beforehand** what is the **dimension or number of elements** in resulting 1D array

In [124…
```
m1[m1%2==0].shape
```

Out[124]:
```
(6,)
```

In [125…
```
m1[m1%2==0].reshape(2, 3)
```

Out[125]:
```
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```

Universal Functions (ufunc) on 2D

Aggregate Functions/ Reduction functions

We saw how aggregate functions work on 1D array in last class

In [126…
```
arr = np.arange(3)
arr
```

Out[126]:
```
array([0, 1, 2])
```

In [127…
```
arr.sum()
```

Out[127]:
```
3
```

Let's apply Aggregate functions on 2D array np.sum()

In [128…
```
a = np.arange(12).reshape(3, 4)
a
```

Out[128]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [129…
```
np.sum(a)  # sums all the values present in array
```

Out[129]:
```
66
```

What if we want to do the elements row-wise or column-wise?

- By **setting `axis` parameter**

What will `np.sum(a, axis=0)` do?

- **`np.sum(a, axis=0)` adds together values in DIFFERENT rows**
- **`axis = 0` ---> Changes will happen along the vertical axis**
- Summing of values happen **in the vertical direction**
- Rows collapse/merge when we do `axis=0`

In [130…
```
np.sum(a, axis=0)
```

Out[130]:
```
array([12, 15, 18, 21])
```

Now, What if we specify `axis=1` ?

- **`np.sum(a, axis=1)` adds together values in DIFFERENT columns**
- **`axis = 1` ---> Changes will happen along the horizontal axis**
- Summing of values happen **in the horizontal direction**
- Columns collapse/merge when we do `axis=1`

In [131…
```
np.sum(a, axis=1)
```

Out[131]:
```
array([ 6, 22, 38])
```

Now, What if we want to find the average value or median value of all the elements in an array?

In [132…
```
np.mean(a) # no need to give any axis
```

Out[132]:
```
5.5
```

What if we want to find the mean of elements in each row or in each column?

- We can do **same thing with `axis` parameter** like we did for `np.sum()` function

Question: Now you tell What will `np.mean(a, axis=0)` give?

- It will give **mean of values in DIFFERENT rows**
- **`axis = 0` ---> Changes will happen along the vertical axis**
- Mean of values will be calculated **in the vertical direction**
- Rows collapse/merge when we do `axis=0`

In [133…
```
np.mean(a, axis=0)
```

Out[133]:
```
array([4., 5., 6., 7.])
```

How can we get mean of elements in each column?

- **`np.mean(a, axis=1)` will give mean of values in DIFFERENT columns**

- `axis = 1` **---> Changes will happen along the horizontal axis**
- Mean of values will be calculated **in the horizontal direction**
- Columns collapse/merge when we do `axis=1`

In [134…
```
np.mean(a, axis=1)
```

Out[134]:
```
array([1.5, 5.5, 9.5])
```

Now, we want to find the minimum value in the array

`np.min()` function can help us with this

In [135…
```
a
```

Out[135]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [136…
```
np.min(a)
```

Out[136]:
```
0
```

What if we want to find row wise minimum value?

Use `axis` argument!!

In [137…
```
np.min(a, axis = 1 )
```

Out[137]:
```
array([0, 4, 8])
```

We can also find max elements in an array.

`np.max()` function will give us *maximum value in the array*

We can also use `axis` argument to find row wise/ column wise max.

In [138…
```
a
```

Out[138]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [139…
```
np.max(a) # maximum value
```

Out[139]:
```
11
```

In [140…
```
np.max(a, axis = 0) # column wise max
```

Out[140]:
```
array([ 8,  9, 10, 11])
```

Logical Operations

Now, What if we want to check whether "any" element of array follows a specific condition?

Let's say we have 2 arrays:

In [141…
```
a = np.array([1,2,3,4])
b = np.array([4,3,2,1])
a, b
```

Out[141]:
```
(array([1, 2, 3, 4]), array([4, 3, 2, 1]))
```

Let's say we want to find out if any of the elements in array `a` is smaller than any of the corresponding elements in array `b`

`np.any()` can become handy here as well

- `any()` returns `True` if **any of the corresponding elements** in the argument arrays follow the **provided condition**.

In [142…
```
a = np.array([1,2,3,4])
b = np.array([4,3,2,1])
np.any(a<b) # Atleast 1 element in a < corresponding element in b
```

Out[142]:
```
True
```

Let's try the same condition with different arrays:

In [143…
```
a = np.array([4,5,6,7])
b = np.array([4,3,2,1])
np.any(a<b) # All elements in a >= corresponding elements in b
```

Out[143]:
```
False
```

- In this case, **NONE of the elements in `a` were smaller than their corresponding elements in `b`**
- So, `np.any(a<b)` returned `False`

What if we want to check whether "all" the elements in our array are non-zero or follow the specified condition?

`np.all()`

Now, What if we want to check whether "all" the elements in our array follow a specific condition?

Let's say we want to find out if all the elements in array `a` are smaller than all the corresponding elements in array `b`

Again, Let's say we have 2 arrays:

In [144…
```
a = np.array([1,2,3,4])
b = np.array([4,3,2,1])
a, b
```

Out[144]:
```
(array([1, 2, 3, 4]), array([4, 3, 2, 1]))
```

In [145…
```
np.all(a<b) # Not all elements in a < corresponding elements in b
```

Out[145]:
```
False
```

Let's try it with different arrays

In [146…
```
a = np.array([1,0,0,0])
b = np.array([4,3,2,1])
np.all(a<b) # All elements in a < corresponding elements in b
```

Out[146]:
```
True
```

- In this case, **ALL the elements in `a` were smaller than their corresponding elements in `b`**
- So, `np.all(a<b)` returned `True`

Multiple conditions for `.all()` function

In [147…
```
a = np.array([1, 2, 3, 2])
b = np.array([2, 2, 3, 2])
c = np.array([6, 4, 4, 5])
((a <= b) & (b <= c)).all()
```

Out[147]:
```
True
```

What if we want to update an array based on condition ?

Suppose you are given an array of integers and you want to update it based on following condition:

- if element is > 0, change it to +1
- if element < 0, change it to -1.

How will you do it ?

In [148…
```
arr = np.array([-3,4,27,34,-2, 0, -45,-11,4, 0 ])
arr
```

Out[148]:
```
array([ -3,   4,  27,  34,  -2,   0, -45, -11,   4,   0])
```

You can use masking to update the array (as discussed in last class)

In [149…
```
arr[arr > 0]  = 1
arr [arr < 0] = -1
```

In [150…
```
arr
```

Out[150]:
```
array([-1,  1,  1,  1, -1,  0, -1, -1,  1,  0])
```

There is a numpy function which can help us with it.

np.where()

Function signature: `np.where(condition, [x, y])`

This functions returns an ndarray whose elements are chosen from x or y depending on condition.

In [151…
```
arr = np.array([-3,4,27,34,-2, 0, -45,-11,4, 0 ])
```

In [152…
```
np.where(arr > 0, +1, -1)
```

Out[152]:
```
array([-1,  1,  1,  1, -1, -1, -1, -1,  1, -1])
```

In [153…
```
arr
```

Out[153]:
```
array([ -3,   4,  27,  34,  -2,   0, -45, -11,   4,   0])
```

Sorting Arrays

- We can also sort the elements of an array along a given specified axis
- Default axis is the last axis of the array.

np.sort()

In [154…
```
a = np.array([2,30,41,7,17,52])
a
```

Out[154]:
```
array([ 2, 30, 41,  7, 17, 52])
```

In [155…
```
np.sort(a)
```

Out[155]:
```
array([ 2,  7, 17, 30, 41, 52])
```

In [156…
```
a
```

Out[156]:
```
array([ 2, 30, 41,  7, 17, 52])
```

Let's work with 2D array

In [157…
```
a = np.arange(9,0,-1).reshape(3,3)
a
```

Out[157]:
```
array([[9, 8, 7],
       [6, 5, 4],
       [3, 2, 1]])
```

Question: What will be the result when we sort using axis = 0 ?

In [158…
```
np.sort(a, axis = 0)
```

Out[158]:
```
array([[3, 2, 1],
       [6, 5, 4],
       [9, 8, 7]])
```

Recall that when axis =0

- change will happen along vertical axis.

Hence, it will sort out row wise.

In [160…
```
a
```

Out[160]:
```
array([[9, 8, 7],
       [6, 5, 4],
       [3, 2, 1]])
```

- Original array is still the same. It hasn't changed

`np.argsort()`

- Returns the **indices** that would sort an array.
- Performs an indirect sort along the given axis.
- It returns **an array of indices of the same shape as a that index data along the given axis in sorted order**.

In [161…
```
a = np.array([2,30,41,7,17,52])
a
```

Out[161]:
```
array([ 2, 30, 41,  7, 17, 52])
```

In [162…
```
np.argsort(a)
```

Out[162]:
```
array([0, 3, 4, 1, 2, 5], dtype=int64)
```

As you can see:

- The orginal indices of elements are in same order as the orginal elements would be in sorted order

Use Case: Fitness Data analysis

Let's first download the dataset

In [163…
```
!gdown 1vk1Pu0djiYcrdc85yUXZ_Rqq2oZNcohd
```

```
Downloading...
From: https://drive.google.com/uc?id=1vk1Pu0djiYcrdc85yUXZ_Rqq2oZNcohd
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\fit.txt

  0%|          | 0.00/3.43k [00:00<?, ?B/s]
100%|##########| 3.43k/3.43k [00:00<00:00, 3.37MB/s]
```

Let's load the data we saw earlier. For this we will use .loadtxt() function

In [164…
```
data = np.loadtxt('fit.txt', dtype='str')
```

We provide file name along with the dtype of data we want to load in

In [165…
```
data[:5]
```

Out[165]:
```
array([['06-10-2017', '5464', 'Neutral', '181', '5', 'Inactive'],
       ['07-10-2017', '6041', 'Sad', '197', '8', 'Inactive'],
       ['08-10-2017', '25', 'Sad', '0', '5', 'Inactive'],
       ['09-10-2017', '5461', 'Sad', '174', '4', 'Inactive'],
       ['10-10-2017', '6915', 'Neutral', '223', '5', 'Active']],
      dtype='<U10')
```

What's the shape of the data?

In [166…
```
data.shape
```

Out[166]:
```
(96, 6)
```

There are 96 records and each record has 6 features. These features are:

- Date
- Step count
- Mood
- Calories Burned
- Hours of sleep
- activity status

Notice that above array is a homogenous containing all the data as strings

In order to work with strings, categorical data and numerical data, we will have save every feature seperately

How will we extract features in seperate variables?

We can get some idea on how data is saved.

Lets see whats the first element of `data`

In [167…
```
data[0]
```

Out[167]:
```
array(['06-10-2017', '5464', 'Neutral', '181', '5', 'Inactive'],
      dtype='<U10')
```

Hm, this extracts a row not a column

Think about it.

Whats the way to change columns to rows and rows to columns?

Transpose

In [168…
```
data.T[0]
```

Out[168]:
```
array(['06-10-2017', '07-10-2017', '08-10-2017', '09-10-2017',
       '10-10-2017', '11-10-2017', '12-10-2017', '13-10-2017',
       '14-10-2017', '15-10-2017', '16-10-2017', '17-10-2017',
       '18-10-2017', '19-10-2017', '20-10-2017', '21-10-2017',
       '22-10-2017', '23-10-2017', '24-10-2017', '25-10-2017',
       '26-10-2017', '27-10-2017', '28-10-2017', '29-10-2017',
       '30-10-2017', '31-10-2017', '01-11-2017', '02-11-2017',
       '03-11-2017', '04-11-2017', '05-11-2017', '06-11-2017',
       '07-11-2017', '08-11-2017', '09-11-2017', '10-11-2017',
       '11-11-2017', '12-11-2017', '13-11-2017', '14-11-2017',
       '15-11-2017', '16-11-2017', '17-11-2017', '18-11-2017',
       '19-11-2017', '20-11-2017', '21-11-2017', '22-11-2017',
       '23-11-2017', '24-11-2017', '25-11-2017', '26-11-2017',
       '27-11-2017', '28-11-2017', '29-11-2017', '30-11-2017',
       '01-12-2017', '02-12-2017', '03-12-2017', '04-12-2017',
       '05-12-2017', '06-12-2017', '07-12-2017', '08-12-2017',
       '09-12-2017', '10-12-2017', '11-12-2017', '12-12-2017',
       '13-12-2017', '14-12-2017', '15-12-2017', '16-12-2017',
       '17-12-2017', '18-12-2017', '19-12-2017', '20-12-2017',
       '21-12-2017', '22-12-2017', '23-12-2017', '24-12-2017',
       '25-12-2017', '26-12-2017', '27-12-2017', '28-12-2017',
       '29-12-2017', '30-12-2017', '31-12-2017', '01-01-2018',
       '02-01-2018', '03-01-2018', '04-01-2018', '05-01-2018',
       '06-01-2018', '07-01-2018', '08-01-2018', '09-01-2018'],
      dtype='<U10')
```

Great, we could extract first column

Lets extract all the columns and save them in seperate variables

In [169…
```
date, step_count, mood, calories_burned, hours_of_sleep, activity_status = data.T
```

In [170…
```
step_count
```

Out[170]:
```
array(['5464', '6041', '25', '5461', '6915', '4545', '4340', '1230', '61',
       '1258', '3148', '4687', '4732', '3519', '1580', '2822', '181',
       '3158', '4383', '3881', '4037', '202', '292', '330', '2209',
       '4550', '4435', '4779', '1831', '2255', '539', '5464', '6041',
       '4068', '4683', '4033', '6314', '614', '3149', '4005', '4880',
       '4136', '705', '570', '269', '4275', '5999', '4421', '6930',
       '5195', '546', '493', '995', '1163', '6676', '3608', '774', '1421',
       '4064', '2725', '5934', '1867', '3721', '2374', '2909', '1648',
       '799', '7102', '3941', '7422', '437', '1231', '1696', '4921',
       '221', '6500', '3575', '4061', '651', '753', '518', '5537', '4108',
       '5376', '3066', '177', '36', '299', '1447', '2599', '702', '133',
       '153', '500', '2127', '2203'], dtype='<U10')
```

In [171…
```
step_count.dtype
```

Out[171]:
```
dtype('<U10')
```

Notice the data type of step_count and other variables. It's a string type where **U** means Unicode String. and 10 means 10 bytes.

Why? Because Numpy type-casted all the data to strings.

Let's convert the data types of these variables

Step Count

In [172…
```
step_count = np.array(step_count, dtype = 'int')
step_count.dtype
```

Out[172]:
```
dtype('int32')
```

In [173…
```
step_count
```

Out[173]:
```
array([5464, 6041,   25, 5461, 6915, 4545, 4340, 1230,   61, 1258, 3148,
       4687, 4732, 3519, 1580, 2822,  181, 3158, 4383, 3881, 4037,  202,
        292,  330, 2209, 4550, 4435, 4779, 1831, 2255,  539, 5464, 6041,
       4068, 4683, 4033, 6314,  614, 3149, 4005, 4880, 4136,  705,  570,
        269, 4275, 5999, 4421, 6930, 5195,  546,  493,  995, 1163, 6676,
       3608,  774, 1421, 4064, 2725, 5934, 1867, 3721, 2374, 2909, 1648,
        799, 7102, 3941, 7422,  437, 1231, 1696, 4921,  221, 6500, 3575,
       4061,  651,  753,  518, 5537, 4108, 5376, 3066,  177,   36,  299,
       1447, 2599,  702,  133,  153,  500, 2127, 2203])
```

Calories Burned

In [174…
```
calories_burned = np.array(calories_burned, dtype = 'int')
calories_burned.dtype
```

Out[174]:
```
dtype('int32')
```

Hours of Sleep

In [175…
```
hours_of_sleep = np.array(hours_of_sleep, dtype = 'int')
hours_of_sleep.dtype
```

Out[175]:
```
dtype('int32')
```

Mood

`Mood` is a categorical data type. As a name says, categorical data type has two or more categories in it.

Let's check the values of `mood` variable

In [176…
```
mood
```

Out[176]:
```
array(['Neutral', 'Sad', 'Sad', 'Sad', 'Neutral', 'Sad', 'Sad', 'Sad',
       'Sad', 'Sad', 'Sad', 'Sad', 'Happy', 'Sad', 'Sad', 'Sad', 'Sad',
       'Neutral', 'Neutral', 'Neutral', 'Neutral', 'Neutral', 'Neutral',
       'Happy', 'Neutral', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Neutral', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Neutral',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Neutral', 'Sad', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Sad', 'Neutral', 'Neutral',
       'Sad', 'Sad', 'Neutral', 'Neutral', 'Happy', 'Neutral', 'Neutral',
       'Sad', 'Neutral', 'Sad', 'Neutral', 'Neutral', 'Sad', 'Sad', 'Sad',
       'Sad', 'Happy', 'Neutral', 'Happy', 'Neutral', 'Sad', 'Sad', 'Sad',
       'Neutral', 'Neutral', 'Sad', 'Sad', 'Happy', 'Neutral', 'Neutral',
       'Happy'], dtype='<U10')
```

In [177…
```
np.unique(mood)
```

Out[177]:
```
array(['Happy', 'Neutral', 'Sad'], dtype='<U10')
```

Activity Status

In [178…
```
activity_status
```

Out[178]:
```
array(['Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Active', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Active', 'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Active', 'Active', 'Active', 'Active', 'Active', 'Active',
       'Active', 'Active', 'Active', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Active', 'Active', 'Active',
       'Active', 'Active', 'Active', 'Active', 'Active', 'Active',
       'Active', 'Active', 'Active', 'Inactive', 'Active', 'Active',
       'Inactive', 'Active', 'Active', 'Active', 'Active', 'Active',
       'Inactive', 'Active', 'Active', 'Active', 'Active', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Active', 'Active', 'Active',
       'Active', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Active'], dtype='<U10')
```

Let's try to get some insights from the data.

What's the average step count?

How can we calculate average? => `.mean()`

In [179…
```
step_count.mean()
```

Out[179]:
```
2935.9375
```

User moves an average of 2900 steps a day.

On which day the step count was highest?

How will be find it?

First we find the index of maximum step count and use that index to get the date.

How'll we find the index? =>

Numpy provides a function `np.argmax()` which returns the index of maximum value element.

Similarly, we have a function `np.argmin()` which returns the index of minimum element.

In [180…
```
step_count.argmax()
```

Out[180]:
```
69
```

Here 69 is the index of maximum step count element.

In [181…
```
date[step_count.argmax()]
```

Out[181]:
```
'14-12-2017'
```

Let's check the calorie burnt on the day

In [182…
```
calories_burned[step_count.argmax()]
```

Out[182]:
```
243
```

Not bad! 243 calories. Let's try to get the number of steps on that day as well

In [183…
```
step_count.max()
```

Out[183]:
```
7422
```

7k steps!! Sports mode on!

Let's try to compare step counts on bad mood days and good mood days Average step count on Sad mood days

In [184…
```
np.mean(step_count[mood == 'Sad'])
```

Out[184]:
```
2103.0689655172414
```

In [185…
```
np.sort(step_count[mood == 'Sad'])
```

Out[185]:
```
array([  25,   36,   61,  133,  177,  181,  221,  299,  518,  651,  702,
        753,  799, 1230, 1258, 1580, 1648, 1696, 2822, 3148, 3519, 3721,
       4061, 4340, 4545, 4687, 5461, 6041, 6676])
```

In [186…
```
np.std(step_count[mood == 'Sad'])
```

Out[186]:
```
2021.2355035376254
```

Average step count on happy days

In [187…
```
np.mean(step_count[mood == 'Happy'])
```

Out[187]:
```
3392.725
```

In [188…
```
np.sort(step_count[mood == 'Happy'])
```

Out[188]:
```
array([ 153,  269,  330,  493,  539,  546,  614,  705,  774,  995, 1421,
       1831, 1867, 2203, 2255, 2725, 3149, 3608, 4005, 4033, 4064, 4068,
       4136, 4275, 4421, 4435, 4550, 4683, 4732, 4779, 4880, 5195, 5376,
       5464, 5537, 5934, 5999, 6314, 6930, 7422])
```

Average step count on sad days - 2103.

Average step count on happy days - 3392

There may be relation between mood and step count

Let's try to check inverse. Mood when step count was greater/lesser Mood when step count > 4000

In [189…
```
np.unique(mood[step_count > 4000], return_counts = True)
```

Out[189]:
```
(array(['Happy', 'Neutral', 'Sad'], dtype='<U10'),
 array([22,  9,  7], dtype=int64))
```

Out of 38 days when step count was more than 4000, user was feeling happy on 22 days.

Mood when step count <= 2000

In [190…
```
np.unique(mood[step_count < 2000], return_counts = True)
```

Out[190]:
```
(array(['Happy', 'Neutral', 'Sad'], dtype='<U10'),
 array([13,  8, 18], dtype=int64))
```

Out of 39 days, when step count was less than 2000, user was feeling sad on 18 days.

There may be a correlation between Mood and step count

3D Arrays

Vectors, Matrix and Tensors

1. **Vector** ---> **1-Dimensional** Array
2. **Matrix** ---> **2-Dimensional** Array
3. **Tensor** ---> **3 and above Dimensional** Array

`Tensor` is a general term we use

- Tensor can also be less than 3D

- **2D Tensor** is called a **Matrix**
- **1D Tensor** is called a **Vector**

In [191…
```
B = np.arange(24).reshape(2, 3, 4)
B
```

Out[191]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
```

```
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Now, What is happening here?

Question: How many dimensions `B` has?

- 3
- It's a **3-dimensional tensor**

How is `reshape(2, 3, 4)` working?

- If you see, it is giving 2 matrices
- Each matrix has 3 rows and 4 columns

So, that's how reshape() is interpreted for 3D

- **1st argument** gives **depth** (No. of Matrices)
- **2nd agrument** gives **no. of rows** in each depth
- **3rd agrument** gives **no. of columns** in each depth

How can I get just the whole of 1st Matrix?

In [192…
```
B[0]
```

Out[192]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Question: What value will I get if I do `B[0, 0, 0]`?

In [193…
```
B[0, 0, 0]
```

Out[193]:
```
0
```

In [194…
```
#### Question: What value will I get if I do `B[1, 1, 1]`?
B[1, 1, 1]

# It looks at Matrix 1, that is, 2nd Matrix (Not Matrix 0)
# Then it looks at row 1 of matrix 1
# Then it looks at column 1 of row 1 of matrix 1
```

Out[194]:
```
17
```

We can also Slicing in 3-Dimensions

- Works same as in 2-D matrices

Use Case: Image Manipulation using Numpy

- By now, you already have an idea that Numpy is an amazing open-source Python library for **data manipulation** and **scientific computing**.
- It is used in the domain of **linear algebra**, Fourier transforms, **matrices**, and the **data science field**.
- **NumPy arrays are way faster than Python Lists**.

Do you know Numpy can also be used for Image Processing?

- The fundamental idea is that we know **images are made up of Numpy** `ndarrays`.
- So we can **manipulate these arrays and play with images**.
- This use case is to give you a broad overview of **Numpy for Image Processing.**

Make sure the required libraries are imported

In [195…
```
import numpy as np
import matplotlib.pyplot as plt
```

Now, we'll see how we can play with images using Numpy

Opening an Image

- Well, to play with an image, we first need to open it

But, How can we open an image in our code?

- To open an image, we will use the `matplotlib` library to read and show images.
- It offers two useful methods `imread()` **and** `imshow()`.

`imread()` **– to read the images**

`imshow()` **– to display the images**

Now, Let's go ahead and load our image

Drive link for the image:

Download the image `fruits.jpg` from here: https://drive.google.com/file/d/1lHPQUi3wdB6HxN-SNJSBQXK7Z0y0wf32/view?usp=sharing

and place it in your current working directory

Let's download the images first

In [196…
```
#fruits image
!gdown 17tYTDPBU5hpby9t0kGd7w_-zBsbY7sEd
```

```
Downloading...
From: https://drive.google.com/uc?id=17tYTDPBU5hpby9t0kGd7w_-zBsbY7sEd
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\fruits.png

  0%|          | 0.00/4.71M [00:00<?, ?B/s]
 11%|#1        | 524k/4.71M [00:00<00:04, 860kB/s]
```

```
 22%|##2        | 1.05M/4.71M [00:01<00:03, 926kB/s]
 33%|###3       | 1.57M/4.71M [00:01<00:02, 1.13MB/s]
 44%|####4      | 2.10M/4.71M [00:01<00:02, 1.21MB/s]
 56%|#####5     | 2.62M/4.71M [00:02<00:01, 1.30MB/s]
 67%|######6    | 3.15M/4.71M [00:02<00:01, 1.28MB/s]
 78%|#######7   | 3.67M/4.71M [00:03<00:00, 1.17MB/s]
 89%|########8  | 4.19M/4.71M [00:03<00:00, 1.33MB/s]
100%|##########| 4.71M/4.71M [00:03<00:00, 1.42MB/s]
100%|##########| 4.71M/4.71M [00:03<00:00, 1.25MB/s]
```

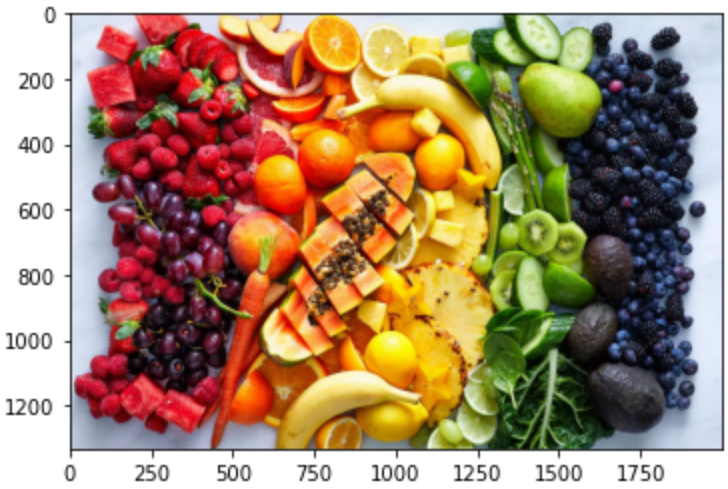
In [197…
```
#emma stone image
!gdown 1o-8yqdTM7cfz_mAaNCi2nH0urFu7pcqI
```

```
Downloading...
From: https://drive.google.com/uc?id=1o-8yqdTM7cfz_mAaNCi2nH0urFu7pcqI
To: C:\Users\kumar\Jupyter Python Files\Scaler Lectures\emma_stone.jpeg

  0%|          | 0.00/80.3k [00:00<?, ?B/s]
100%|##########| 80.3k/80.3k [00:00<00:00, 321kB/s]
100%|##########| 80.3k/80.3k [00:00<00:00, 321kB/s]
```

In [198…
```
img = np.array(plt.imread('fruits.png'))
plt.imshow(img)
```

Out[198]:
```
<matplotlib.image.AxesImage at 0x1bc486c9dc0>
```



Details of an Image

What do you think are the dimensions and shape of this image?

We will see what is the **dimension and shape of this image**, using the `Image.ndim` and `Image.shape` properties.

In [199…
```
print('# of dims: ',img.ndim)      # dimension of an image
print('Img shape: ',img.shape)     # shape of an image
```

```
# of dims:  3
Img shape:  (1333, 2000, 3)
```

How come our 2-D image has 3 dimensions?

- **Coloured images have a 3rd dimension for depth or RGB colour channel**
- Here, the **depth is 3**
- But we will come to what RGB colour channels are in a bit

First, Let's understand something peculiar happening here with the `shape` of image

Do you see something different happening here when we check the `shape` of image?

- When we discussed **3-D Arrays**, we saw that **depth was the first element of the `shape` tuple**
- But when we are loading an image using **matplotlib and getting its 3-D array**, we see that **depth is the last element of the `shape` tuple**

Why is there a difference b/w normal np array and the np array generated from Matplotlib in terms of where the depth part of `shape` appears?

- This is how `matplotlib` reads the image
- It **reads the depth values (R, G and B values) of each pixel one by one** and stacks them one after the other

The shape of imge we read is: (1333, 2000, 3)

- `matplotlib` **first reads that each plane has** 1333×2000 **pixels**
- Then, it **reads depth values (R, G and B values) of each pixel and place the values in 3 separate planes**
- That is why **depth is the last element of `shape` tuple in np array generated from an image read by `matplotlib`**
- Whereas in a **normal np array, depth is the first element of `shape` tuple**

Now, What are these RGB channels and How can we visualize them?

Visualizing RGB Channels

We can split the image into each RGB color channels using only Numpy

But, What exactly RGB values are?

- These are values of each pixel of an image
- Each pixel is made up of **3 components/channels** - **Red, Green, Blue** - which form RGB values
- Coloured images are usually stored as 3-dimensional arrays of **8-bit unsigned integers**
- So, the range of values that each channel of a pixel can take is 0 to $2^8 - 1$
- That is, each pixel's each channel, R, G and B can range from **0 to 255**

Each pixel has these 3 values which combined together forms the colour that the pixel represents

- So, a pixel **[255, 0, 0]** will be **RED** in colour
- A pixel **[0, 255, 0]** will be **GREEN** in colour
- A pixel **[0, 0, 255]** will be **BLUE** in colour

Question: What will be the colour of pixel [0, 0, 0]?

- Black

Question: What will be the colour of pixel [255, 255, 255]?

- White

Now, Let's separate the R, G, B channels in our image:

- We'll make use of **slicing of arrays**
- For **RED** channel, we'll **set values of GREEN and BLUE to 0**

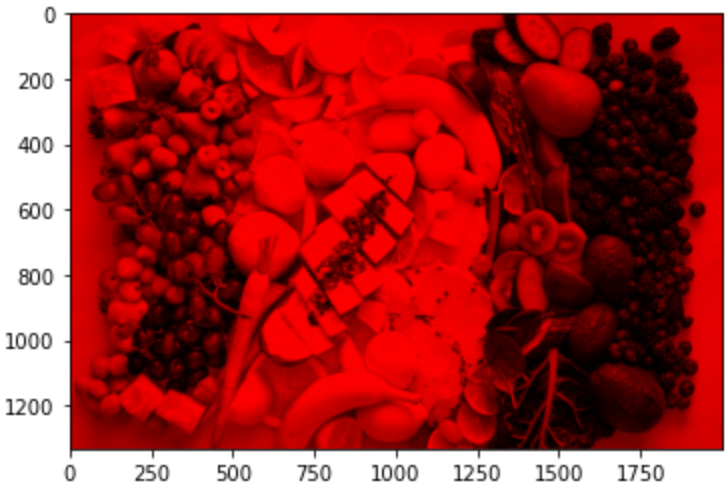
In [200…
```
img = np.array(plt.imread('fruits.png'))

img_R = img.copy()

img_R[:, :, (1, 2)] = 0
```

In [201…
```
plt.imshow(img_R)
```

Out[201]:
```
<matplotlib.image.AxesImage at 0x1bc487cb6d0>
```



Similarly, for GREEN channel, we'll set values of RED and BLUE to 0

... and same for BLUE channel

Rotating an Image (Transpose the Numpy Array)

Now, What if we want to rotate the image?

- Remember **image is a Numpy array**
- **Rotating the image means transposing the array**

For this, we'll use the `np.transpose()` function in numpy

Now, Let's understand `np.transpose()` function first

- It takes 2 arguments

1st argument is obviously the **array that we want to transpose (image array in our case)**

2nd argument is `axes`

- Its a **tuple or list of ints**
- It contains a **permutation of [0,1,..,N-1] where N is the number of axes of array**

Now, our image array has 3 axes (3 dimensions) ---> 0th, 1st and 2nd

- We specify how we want to transpose the array by giving an **order of these axes inside the tuple**
  - **Vertical axis (Row axis) is 0th axis**
  - **Horizontal axis (Column axis) is 1st axis**
  - **Depth axis is 2nd axis**
- **In order to rotate the image, we want to transpose the array**
- That is, we want to **transpose rows into columns and columns into rows**
- So, we want to **interchange the order of row and column axis** ---> **interchange order of 0th and 1st axis**
- We **don't want to change the depth axis (2nd axis)** ---> So, it will **remain at its original order position**

Now, the **order of axes in orginal image is** `(0, 1, 2)`

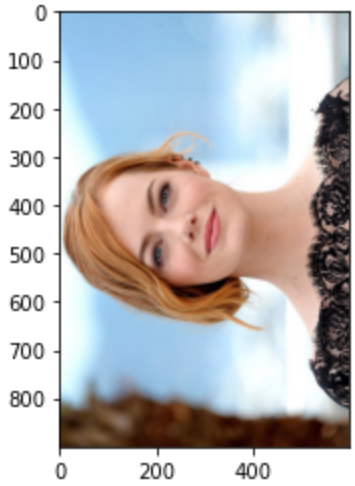
What will be the order of axes rotated image or transposed array?

- The **order of axes in rotated image will be** `(1, 0, 2)`
- **Order (Position) of 0th and 1st column is interchanged**

Let's see it in action:

In [202…
```
img = np.array(plt.imread('emma_stone.jpeg'))
img_rotated = np.transpose(img, (1,0,2))
plt.imshow(img_rotated)
```

Out[202]:
```
<matplotlib.image.AxesImage at 0x1bc489e5370>
```



As you can see:

- We obtained the **rotated image by transposing the np array**

Trim Image

Now, How can we crop an image using Numpy?

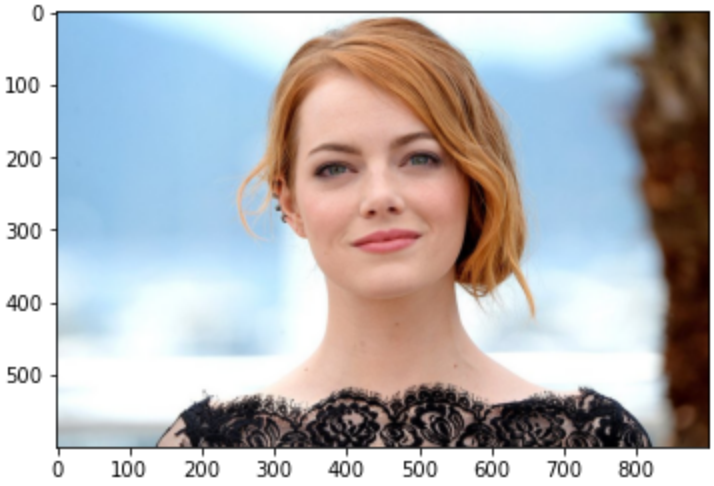
- Remember! Image is a numpy array of pixels
- So, We can trim/crop an image in Numpy using Array using **Slicing**.

Let's first see the original image

In [203…
```
img = np.array(plt.imread('./emma_stone.jpeg'))

plt.imshow(img)
```

Out[203]:
```
<matplotlib.image.AxesImage at 0x1bc48be8e20>
```



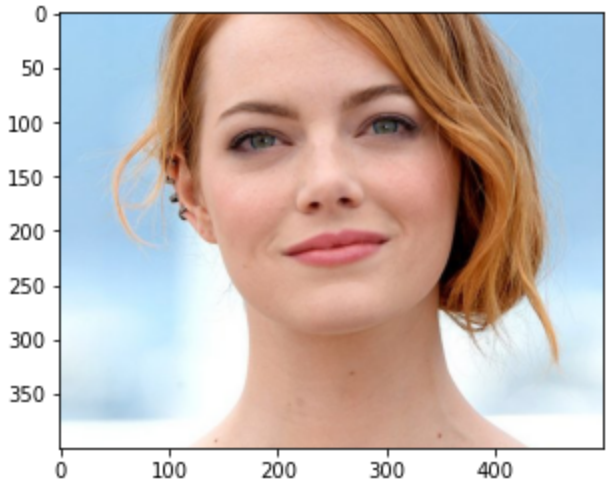
Now, Let's crop the image to get the face only

- If you see x and y axis, the face starts somewhat from ~200 and ends at ~700 on x-axis
  - **x-axis in image is column axis in np array**
  - Columns change along x-axis
- And it lies between ~100 to ~500 on y-axis
  - **y-axis in image is row axis in np array**
  - Rows change along y-axis

We'll use this information to slice our image array

In [204…
```
img_crop = img[100:500, 200:700, :]
plt.imshow(img_crop)
```

Out[204]:
```
<matplotlib.image.AxesImage at 0x1bc48f8a430>
```



Saving Image as `ndarray`

Now, How can we save `ndarray` as Image?

To save a `ndarray` as an image, we can use `matplotlib`'s `plt.imsave()` method.

- **1st agrument** ---> We provide the path and name of file we want to save the image as
- **2nd agrument** ---> We provide the image we want to save

Let's save the cropped face image we obtained previously

In [205…
```
path = 'emma_face.jpg'
plt.imsave(path, img_rotated)
```

Now, if you go and check your current working directory, image would have been saved by the name `emma_face.jpg`

Array splitting and Merging

- In addition to reshaping and selecting subarrays, it is often necessary to split arrays into smaller arrays or merge arrays into bigger arrays,
- **For example**, when joining separately computed or measured data series into a **higher-dimensional array**, such as a matrix.

Splitting

`np.split()`

- Splits an array into multiple sub-arrays as views

It takes an argument `indices_or_sections`

- If `indices_or_sections` is an **integer, n**, the array will be **divided into n equal arrays along axis**.
- If such a split is not possible, an error is raised.

- If `indices_or_sections` is a **1-D array of sorted integers**, the entries indicate **where along axis the array is split**.
- If an index **exceeds the dimension of the array along axis**, an **empty sub-array is returned** correspondingly.

In [211…
```
import numpy as np
```

In [212…
```
x = np.arange(9)
x
```

Out[212]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

In [213…
```
np.split(x, 3)
```

Out[213]:
```
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]
```

In [214…
```
np.split(x, [3, 5, 6])
```

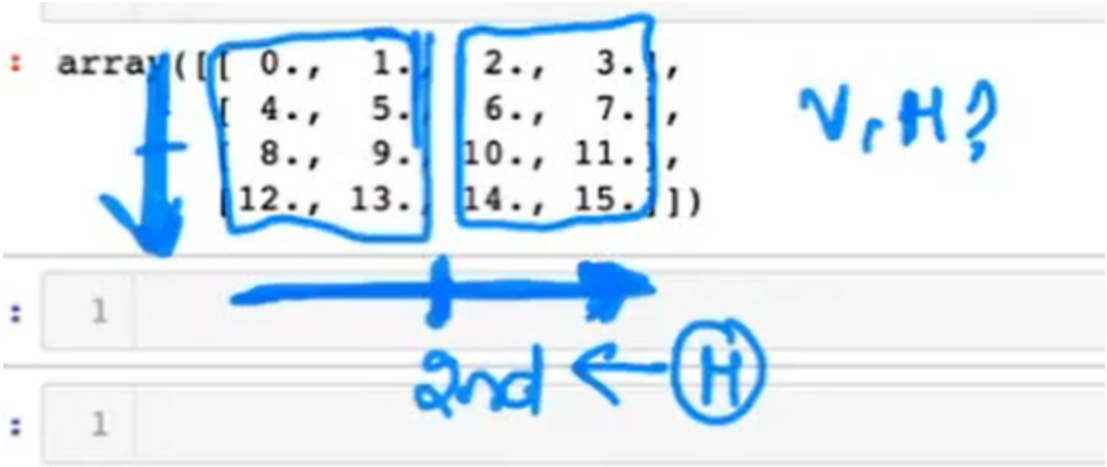
Out[214]:
```
[array([0, 1, 2]), array([3, 4]), array([5]), array([6, 7, 8])]
```

`np.hsplit()`

- Splits an array into multiple sub-arrays **horizontally (column-wise)**.

In [215…
```
x = np.arange(16.0).reshape(4, 4)
x
```

Out[215]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```



Think of it this way:

- There are 2 axis to a 2-D array
  1. **1st axis - Vertical axis**
  2. **2nd axis - Horizontal axis**

Along which axis are we splitting the array?

- The split we want happens across the **2nd axis (Horizontal axis)**

- That is why we use `hsplit()`

So, try to think in terms of "whether the operation is happening along vertical axis or horizontal axis"

- We are splitting the horizontal axis in this case

In [216…
```
np.hsplit(x, 2)
```

Out[216]:
```
[array([[ 0.,  1.],
        [ 4.,  5.],
        [ 8.,  9.],
        [12., 13.]]),
 array([[ 2.,  3.],
        [ 6.,  7.],
        [10., 11.],
        [14., 15.]])]
```

In [217…
```
np.hsplit(x, np.array([3, 6]))
```

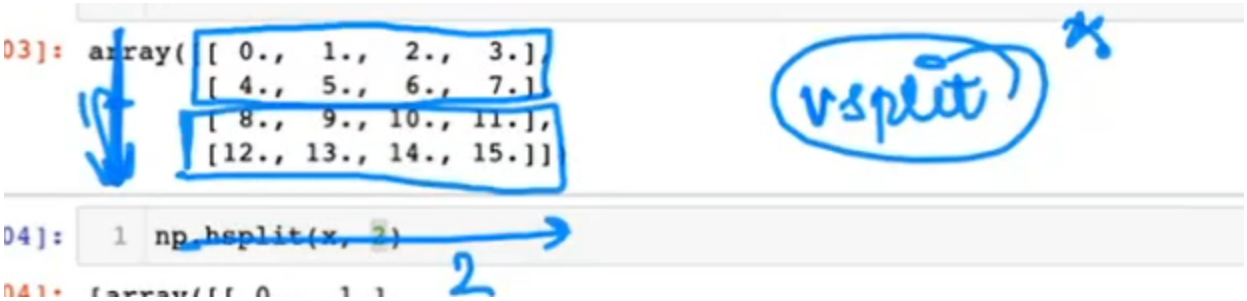
Out[217]:
```
[array([[ 0.,  1.,  2.],
        [ 4.,  5.,  6.],
        [ 8.,  9., 10.],
        [12., 13., 14.]]),
 array([[ 3.],
        [ 7.],
        [11.],
        [15.]]),
 array([], shape=(4, 0), dtype=float64)]
```

`np.vsplit()`

- Splits an array into multiple sub-arrays **vertically (row-wise)**.

In [218…
```
x = np.arange(16.0).reshape(4, 4)
x
```

Out[218]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```



Now, along which axis are we splitting the array?

- The split we want happens across the **1st axis (Vertical axis)**
- That is why we use `vsplit()`

Again, always try to think in terms of "whether the operation is happening along vertical axis or horizontal axis"

- We are splitting the vertical axis in this case

In [219…
```
np.vsplit(x, 2)
```

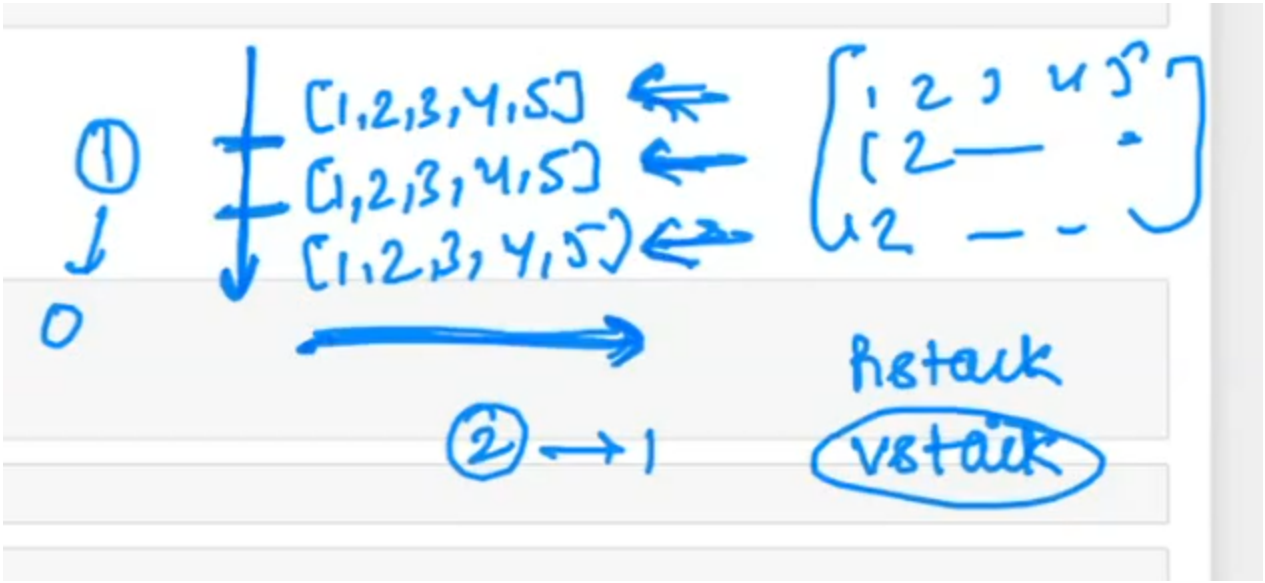
Out[219]:
```
[array([[0., 1., 2., 3.],
        [4., 5., 6., 7.]]),
 array([[ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])]
```

In [220…
```
np.vsplit(x, np.array([3]))
```

Out[220]:
```
[array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]]),
 array([[12., 13., 14., 15.]])]
```

Stacking

Let's say we have an array and we want to stack it like this:



- Operation or **change is happening along horizontal axis**
- So, we'll use `hstack()`

`np.hstack()`

- Stacks a list of arrays horizontally (along axis 1)
- For **example**, **given a list of column vectors, appends the columns to form a matrix**.

In [221…
```
data = np.arange(5).reshape(5,1)
data
```

Out[221]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

In [222…
```
np.hstack((data, data, data))
```

Out[222]:
```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2],
```

```
       [3, 3, 3],
       [4, 4, 4]])
```

Question: Now, What will be the output of this?

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

In [223…
```
a = np.array([[1], [2], [3]])
a
```

Out[223]:
```
array([[1],
       [2],
       [3]])
```

In [224…
```
b = np.array([[4], [5], [6]])
b
```

Out[224]:
```
array([[4],
       [5],
       [6]])
```

In [225…
```
np.hstack((a, b))
```

Out[225]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

This time both `a` and `b` are column vectors

- So, the stacking of `a` and `b` along horizontal axis is more clearly visible

Now, Let's look at a more generalized way of stacking arrays

`np.concatenate()`

- Creates a new array by appending arrays after each other, along a given axis
- Provides similar functionality, but it takes a **keyword argument** **`axis`** that specifies the **axis along which the arrays are to be concatenated**.

Input array to `concatenate()` needs to be of dimensions atleast equal to the dimensions of output array

In [226…
```
z = np.array([[2, 4]])
z
```

Out[226]:
```
array([[2, 4]])
```

In [227…
```
z.ndim
```

Out[227]:
```
2
```

In [228…
```
zz = np.concatenate([z, z], axis=0)
zz
```

Out[228]:
```
array([[2, 4],
       [2, 4]])
```

In [229…
```
zz = np.concatenate([z, z], axis=1)
zz
```

Out[229]:
```
array([[2, 4, 2, 4]])
```

Let's look at a few more examples using `np.concatenate()`

Question: What will be the output of this?

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
```

In [230…
```
a = np.array([[1, 2], [3, 4]])
a
```

Out[230]:
```
array([[1, 2],
       [3, 4]])
```

In [231…
```
b = np.array([[5, 6]])
b
```

Out[231]:
```
array([[5, 6]])
```

In [232…
```
np.concatenate((a, b), axis=0)
```

Out[232]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Now, How did it work?

- Dimensions of `a` is 2×2

What is the dimensions of `b` ?

- 1-D array ?? - **NO**
- Look carefully!!
- **`b` is a 2-D array of dimensions** 1×2

`axis = 0` ---> It's a vertical axis

- So, **changes will happen along vertical axis**
- So, **`b` gets concatenated below `a`**

Now, What if we do NOT provide an axis along which to concatenate?

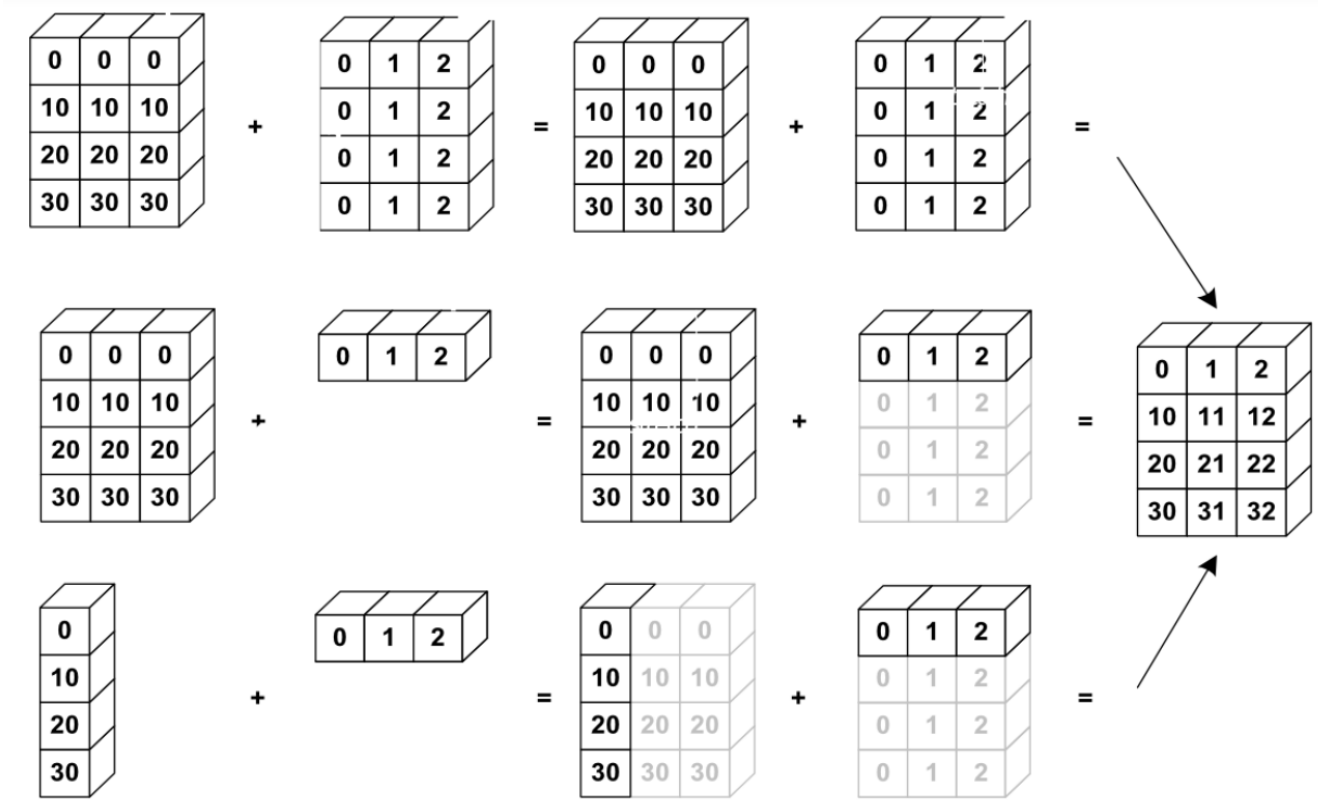
In [233…
```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=None)
```

Out[233]:
```
array([1, 2, 3, 4, 5, 6])
```

Can you see what happened here?

- When we **don't specify the axis (`axis=None`),** `np.concatenate()` **flattens the arrays and concatenates them as 1-D row array**

Broadcasting



Case1:

You are given two 2D array

```
[[0,   0,  0],      [[0, 1, 2],
 [10, 10, 10], and   [0, 1, 2],
 [20, 20, 20],       [0, 1, 2],
 [30, 30, 30]]       [0, 1, 2]]
```

Shape of **first array** is **4x3**

Shape of **second array** is **4x3**.

Will addtion of these array be possible? Yes as the shape of these two array matches.

In [234…
```
a = np.tile(np.arange(0,40,10), (3,1))
a
```

Out[234]:
```
array([[ 0, 10, 20, 30],
       [ 0, 10, 20, 30],
       [ 0, 10, 20, 30]])
```

`np.tile` function is used to repeat the given array multiple times

In [235…
```
np.tile(np.arange(0,40,10), (3,2))
```

Out[235]:
```
array([[ 0, 10, 20, 30,  0, 10, 20, 30],
       [ 0, 10, 20, 30,  0, 10, 20, 30],
```

```
       [ 0, 10, 20, 30,  0, 10, 20, 30]])
```

Now, let's get back to example:

In [236…
```
a
```

Out[236]:
```
array([[ 0, 10, 20, 30],
       [ 0, 10, 20, 30],
       [ 0, 10, 20, 30]])
```

In [237…
```
a = a.T
```

In [238…
```
a
```

Out[238]:
```
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
```

In [239…
```
b = np.tile(np.arange(0,3), (4,1))
```

In [240…
```
b
```

Out[240]:
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

Let's add these two arrays:

In [241…
```
a + b
```

Out[241]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Text book case of element wise addition of two 2D arrays.

Case2 :

Imagine a array like this:

```
[[0,   0,  0],
 [10, 10, 10],
 [20, 20, 20],
 [30, 30, 30]]
```

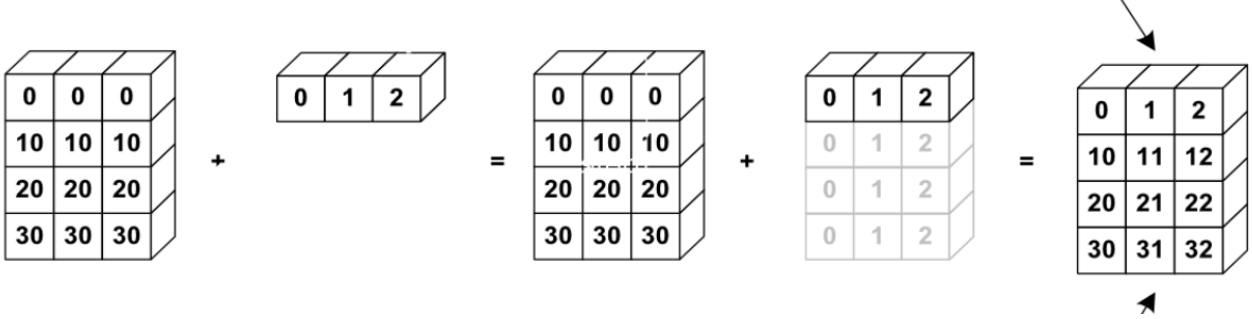
I want to add the following array to it:

```
[[0, 1, 2]]
```

Is it possible? **Yes!**

What broadcasting does is replicate the second array row wise 4 times to fit the size of first array.

Here both array have same number of columns



In [242…
```
a
```

Out[242]:
```
array([[ 0,  0,  0],
       [10, 10, 10],
       [20, 20, 20],
       [30, 30, 30]])
```

In [243…
```
b = np.arange(0,3)
b
```

Out[243]:
```
array([0, 1, 2])
```

In [244…
```
a + b
```

Out[244]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

The **smaller array is broadcasted across the larger array** so that they have **compatible shapes**.

Case 3:

Imagine I have two array like this:

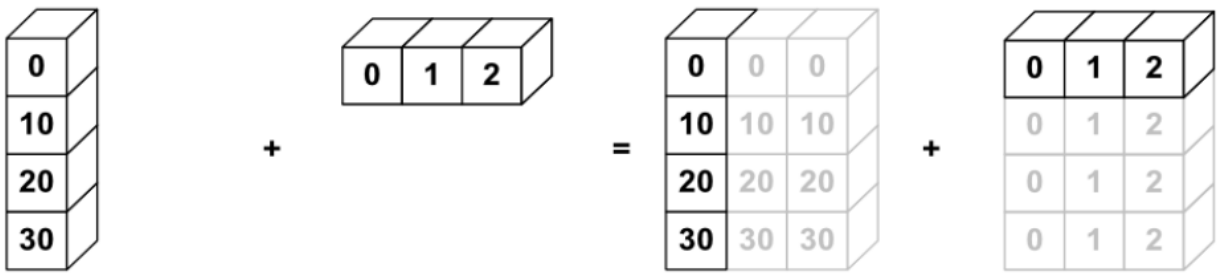
```
[[0],
[10],
[20],
[30]]
```

and

```
[[0, 1, 2]]
```

i.e. one column matrix and one row matrix.

When we try to add these array up, broadcasting will replicate first array column wise 3 time and secord array row wise 4 times to match up the shape.



In [245…
```
a = np.arange(0,40,10)
a
```

Out[245]:
```
array([ 0, 10, 20, 30])
```

This is a 1D row wise array, But we want this array colum wise? How do we do it ? Reshape?

In [246…
```
a = a.reshape(4,1)
a
```

Out[246]:
```
array([[ 0],
       [10],
       [20],
       [30]])
```

In [247…
```
b = np.arange(0,3)
b
```

Out[247]:
```
array([0, 1, 2])
```

In [248…
```
a + b
```

Out[248]:
```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Question: (for general broadcasting rules)

What will be the output of the following?

```
a = np.arange(8).reshape(2,4)
b = np.arange(16).reshape(4,4)

print(a*b)
```

In [249…
```
a = np.arange(8).reshape(2,4)
a
```

Out[249]:
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [250…
```
b = np.arange(16).reshape(4,4)
b
```

Out[250]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [254…
```
a + b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [254], in <cell line: 1>()
----> 1 a + b

ValueError: operands could not be broadcast together with shapes (2,4) (4,4)
```

Why didn't it work?

To understand this, let's learn about some **General Broadcasting Rules**

```
For each dimension ( going from right side)
    1. The size of each dimension should be same OR
    2. The size of one dimension should be 1
```

Rule 1 : If two array differ in the number of dimensions, the shape of one with fewer dimensions is padded with ones on its leading(Left Side).

Rule 2 : If the shape of two arrays doesnt match in any dimensions, the array with shape equal to 1 is stretched to match the other shape.

Rule 3 : If in any dimesion the sizes disagree and neither equal to 1 , then Error is raised.

In the above example, the shapes were (2,4) and (4,4).

Let's compare the dimension from right to left

- First, it will compare the right most dimension (4) which are equal.
- Next, it will compare the left dimension i.e. 2 and 4.
  - Both conditions fail here. They are neither equal nor one of them is 1.

Hence, it threw an error while broadcasting.

Now, Let's take a look at few more examples

Question : Will broadcasting work in this case ?

```
A = np.arange(1,10).reshape(3,3)
B = np.array([-1, 0, 1])
A * B
```

In [255…
```
A = np.arange(1,10).reshape(3,3)
A
```

Out[255]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [256…
```
B = np.array([-1, 0, 1])
B
```

Out[256]:
```
array([-1,  0,  1])
```

In [257…
```
A * B
```

Out[257]:
```
array([[-1,  0,  3],
       [-4,  0,  6],
       [-7,  0,  9]])
```

Why did `A * B` work in this case?

- `A` has 3 rows and 3 columns i.e. (3,3)
- `B` is a 1-D vector with 3 elements (3,)

Now, if you look at **rule 1**

```
Rule 1 : If two array differ in the number of dimensions,
the shape of one with fewer dimensions is padded with ones on its leading( Left
Side).
```

What is the shape of A and B ?

- A has a shape of (3,3)
- B has a shape of (3,)

As per the rule 1,

- the shape of array with fewer dimensions will be prefixed with ones on its leading side.

Here, shape of B will be prefixed with 1

- So, it's shape will become (1,3)

Can we add a (3,3) and (1,3) array ?

We check the validity of broadcasting. i.e. if broadcasting is possible or not.

Checking the dimension from right to left.

- It will compare the right most dimension (3); which are equal
- Now, it compares the leading dimension.
  - The size of one dimension is 1.

Hence, broadcasting condition is satisfied

How will it broadcast?

As per rule 2:

```
Rule 2 :
If the shape of two arrays doesnt match in any dimensions,
the array with shape equal to 1 is stretched to match the other shape.
```

Here, array B (**1**,3) will replicate/stretch its row 3 times to match shape of B

So , **`B` gets broadcasted over `A` for each row of `A`**

Question: Will broadcasting work in following case ?

```
A = np.arange(1,10).reshape(3,3)
B = np.arange(3, 10, 3).reshape(3,1)
C = A + B
```

In [258…
```
A = np.arange(1,10).reshape(3,3)
A
```

Out[258]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [259…
```
B = np.arange(3, 10, 3).reshape(3,1)
B
```

Out[259]:
```
array([[3],
       [6],
       [9]])
```

How did this `A + B` work?

- `A` has 3 rows and 3 columns i.e. shape (3,3)
- `B` has 3 rows and 1 column -i.e. shape (3,1)

Do we need to check rule 1 ?

Since, both arrays have same number of dimensions, we can ignore Rule 1.

Let's check whether broadcasting is possible or not

Now, for each dimension from right to left

- Right most dimension is 1.
- Leading dimension are matching (3)

So, conditions for broadcasting are met.

How will broadcasting happen?

As per rule 2, dimension with value 1 will be streched.

- A.shape => (3,3)
- B.shape => (3,1)

Hence, columns of B will be replicated/streched to match dimensions of A.

- So, **`B` gets broadcasted on every column of `A`**

In [260…
```
C = A + B
np.round(C, 1)
```

Out[260]:
```
array([[ 4,  5,  6],
       [10, 11, 12],
       [16, 17, 18]])
```

Dimension Expansion and Reduction

Recall that we learnt how to convert 1D array to 2D array in previous lectures

In [261…
```
import numpy as np
```

In [262…
```
arr = np.arange(6)
arr
```

Out[262]:
```
array([0, 1, 2, 3, 4, 5])
```

In [263…
```
arr.shape
```

Out[263]:
```
(6,)
```

In [264…
```
arr = arr.reshape(1,-1)
```

In [265…
```
arr.shape
```

Out[265]:
```
(1, 6)
```

This is also know as expanding dimensions

i.e. we expanded our dimension from 1D to 2D

We can also perform same operation using `np.newaxis()`

`np.expand_dims()`

- Expands the shape of an array with axis of length 1.
- Insert a new axis that will appear at the axis position in the expanded array shape.

Function signature: `np.exapnd_dims(arr, axis)`

Documentation:
https://numpy.org/doc/stable/reference/generated/numpy.expand_dims.html#numpy.expand_dims

In [266…
```
arr
```

Out[266]:
```
array([[0, 1, 2, 3, 4, 5]])
```

Let's check the shape of arr

In [267…
```
arr.shape
```

Out[267]:
```
(1, 6)
```

Let's expand the dimensions

In [268…
```
arr1 = np.expand_dims(arr, axis = 0 )
arr1
```

Out[268]:
```
array([[[0, 1, 2, 3, 4, 5]]])
```

In [269…
```
arr1.shape
```

Out[269]:
```
(1, 1, 6)
```

What happened here?

Here, the shape of array is (6,)

- We only have one axis i.e. axis = 0.

When we expand dimension with `axis =0` ,

- it add 1 to dimension @ axis = 0
- Shape becomes (1, 6) from (6,)
- i.e. 1 is padded at the given axis location

Let's expand dims @ axis = 1

In [270…
```
arr2 = np.expand_dims(arr, axis = 1)
arr2
```

Out[270]:
```
array([[[0, 1, 2, 3, 4, 5]]])
```

In [271…
```
arr2.shape
```

Out[271]:
```
(1, 1, 6)
```

Notice that,

- as we provided `axis =1` in argument,
- It expanded the shape along axis =1 i.e 1 was appened @ axis 1.
- Hence, shape become (6,1) from (6,)

We can also do same thing using `np.newaxis`

`np.newaxis`

- passed as a parameter to the array.

Let's see how it works

In [272…
```
arr = np.arange(6)
```

In [273…
```
arr[np.newaxis, :] #equivalent to np.expand_dims(arr, axis =0)
```

Out[273]:
```
array([[0, 1, 2, 3, 4, 5]])
```

We basically passed `np.newaxis` at the axis position where we want to add an axis

- In `arr[np.newaxis, : ]` ,
  - we passed it @ axis =0, hence shape 1 was added @ axis = 0
  - and therefore, shape became (1, 6)

In [274…
```
arr[:, np.newaxis] # equivalent to np.expand_dims(arr, axis = 1 )
```

Out[274]:
```
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])
```

What if we want to reduce the number of dimensions?

We can use `np.squeeze` for reducing the dimensions

`np.sqeeze()`

- It removes the axis of length 1 from array.
- Inverse of `expand_dims`

Function signature: `np.squeeze(arr, axis)`

Documentation: https://numpy.org/doc/stable/reference/generated/numpy.squeeze.html

In [275…
```
arr = np.arange(9).reshape(1,1,9)
arr
```

Out[275]:
```
array([[[0, 1, 2, 3, 4, 5, 6, 7, 8]]])
```

In [276…
```
arr.shape
```

Out[276]:
```
(1, 1, 9)
```

In [277…
```
arr1 = np.squeeze(arr)
arr1
```

Out[277]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

In [278…
```
arr1.shape
```

Out[278]:
```
(9,)
```

Notice that

- it reduced the shape from (1,1,9) to (9,)
- it did so by removing the axis of length 1
- i.e. it removed axis 0 and 1.

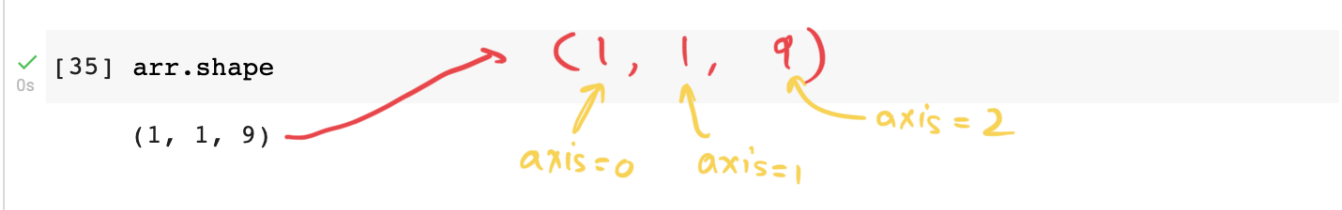
We can also remove specific axis using the `axis` argument

In [279…
```
arr
```

Out[279]:
```
array([[[0, 1, 2, 3, 4, 5, 6, 7, 8]]])
```

In [280…
```
arr.shape
```

Out[280]:
```
(1, 1, 9)
```



Let's remove axis = 1

In [281…
```
arr1 = np.squeeze(arr, axis = 1 )
arr1
```

Out[281]:
```
array([[0, 1, 2, 3, 4, 5, 6, 7, 8]])
```

In [282…
```
arr1.shape
```

Out[282]:
```
(1, 9)
```

What if we try to remove 2nd axis?

In [283…
```
np.squeeze(arr, axis = 2 )
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [283], in <cell line: 1>()
----> 1 np.squeeze(arr, axis = 2 )

File <__array_function__ internals>:5, in squeeze(*args, **kwargs)

File ~\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:1508, in squeeze(a, axis)
   1506     return squeeze()
   1507 else:
-> 1508     return squeeze(axis=axis)

ValueError: cannot select an axis to squeeze out which has size not equal to one
```

It'll throw an error

- as we are trying to remove non- one length axis

Shallow vs Deep Copy

- Numpy **manages memory very efficiently**
- Which makes it really **useful while dealing with large datasets**

But how does it manage memory so efficiently?

- Let's create some arrays to understand what's happening in memory while using Numpy

In [284…
```
# We'll create np array

a = np.arange(4)
a
```

Out[284]:
```
array([0, 1, 2, 3])
```

In [285…
```
# Reshape array `a` and store in b

b = a.reshape(2, 2)
b
```

Out[285]:
```
array([[0, 1],
       [2, 3]])
```

Now we will make some changes to our original array `a`

In [286…
```
a[0] = 100
```

```
a
```

Out[286]:
```
array([100,   1,   2,   3])
```

What will be values if we print array b ?

In [287…
```
b
```

Out[287]:
```
array([[100,   1],
       [  2,   3]])
```

Surprise Surprise!!

- Array `b` **got automatically updated**

This is an example of Numpy using "Shallow Copy" of data

Now, What happens here?

- Numpy **re-uses data** as much as possible **instead of duplicating** it
- This helps Numpy to be efficient

When we created `b = a.reshape(2, 2)`

- Numpy **did NOT make a copy of `a` to store in `b`**, as we can clearly see
- It is **using the same data as in `a`**
- It **just looks different (reshaped)** in `b`
- That is why, **any changes in `a` automatically gets reflected in `b`**

How data is stored using Numpy?

- Variable **does NOT directly point to data** stored in memory
- There is something called **Header** in-between

What does Header do?

- **Variable points to header** and **header points to data** stored in memory
- Header stores **information about data** - called **Metadata**

`a` is pointing to Metadata about our data `[0, 1, 2, 3]`, which may include:

- **How many values** we have --> 4
- What is the **Data Type** of data --> `int`
- What's the **Shape** --> `(4,)`
- What's the **stride** i.e. step size --> 1

When we do `b = a.reshape(2, 2)`

- Numpy **does NOT duplicate the data** pointed to by `a`
- It **uses the same data**
- And **create a New header for** `b` that **points to the same data** as pointed to by `a`

`b` points to a new Header having different values of Metadata of the same data:

- **Number of values** --> 4
- **Data Type** --> `int`
- **Shape** --> `(2, 2)`
- **Stride** i.e. step size --> 1

That is why:

- When data is accessed using `a`, it gives data in shape `(4,)`
- And when data is accessed using `b`, it gives same data in shape `(2, 2)`

This helps Numpy to save time and space - Making it efficient

Now, Let's see an example where Numpy will create a "Deep Copy" of data

Now, What if we do this?

Numpy metadata internals

In [288…
```
a = np.arange(4)
a
```

Out[288]:
```
array([0, 1, 2, 3])
```

In [289…
```
# Create `c`

c = a + 2
c
```

Out[289]:
```
array([2, 3, 4, 5])
```

In [290…
```
# We make changes in a

a[0] = 100
a
```

Out[290]:
```
array([100,   1,   2,   3])
```

In [291…
```
c
```

Out[291]:
```
array([2, 3, 4, 5])
```

As we can see, `c` did not get affected on changing `a`

- Because it is an operation

- A more **permanent change in data**
- So, Numpy **had to create a separate copy for `c`** - i.e., **deep copy of array `a` for array `c`**

Conclusion:

- Numpy is able to **use same data** for **simpler operations** like **reshape** ---> **Shallow Copy**
- It creates a **copy of data** where operations make **more permanent changes** to data ---> **Deep Copy**

Be careful about this while writing code using Numpy

Is there a way to check whether two arrays are sharing memory or not? Yes, there is `np.shares_memory()` function to the rescue!!

In [292…
```
a= np.arange(10)
a
```

Out[292]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [293…
```
b = a[::2]
b
```

Out[293]:
```
array([0, 2, 4, 6, 8])
```

In [294…
```
np.shares_memory(a,b)
```

Out[294]:
```
True
```

Notice that Slicing creates shallow copies.

Why does slicing create shallow copies ?

Rememeber the stride param of the header.

- Stride is nothing but the step size.

For Array `a`, we have a stride of 1.

For creating array b,

- we are slicing array a by 2 i.e. stride 2.
- So, it creates a new header for array b with stride = 2 while pointing to the original data

In [295…
```
b[0] = 2
b
```

Out[295]:
```
array([2, 2, 4, 6, 8])
```

In [296…
```
a
```

Out[296]:
```
array([2, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Notice how change in b also changed the value in array a

Let's check with deep copy

In [297…
```
a
```

Out[297]:
```
array([2, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [298…
```
b = a +2
```

In [299…
```
np.shares_memory(a,b)
```

Out[299]:
```
False
```

We learnt how `.reshape` and **Slicing returns a view** of the original array

- i.e. Any changes made in original array will be reflected in the new array.

However, we saw that creating new array using

- **masking** or **array operation returns deep copy** of the array.
- Any changes made in new array are not reflected in the original array.

Numpy also provides us with few functions to make shallow/ deep copy

How to make shallow copy?

Numpy provides us with `.view()` function which returns view of an array

`.view()`

Returns view of the original array

- Any changes made in new array will be reflected in original array.

Function documentation: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.view.html

In [300…
```
arr = np.arange(10)
arr
```

Out[300]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [301…
```
view_arr = arr.view()
view_arr
```

Out[301]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [302…
```
view_arr[4] = 420
view_arr
```

Out[302]:
```
array([  0,   1,   2,   3, 420,   5,   6,   7,   8,   9])
```

In [303…
```
arr
```

Out[303]:
```
array([  0,   1,   2,   3, 420,   5,   6,   7,   8,   9])
```

Notice that changes in view array are reflected in original array.

How do we make deep copy ?

Numpy has `.copy()` function for that purpose

`.copy()`

Returns copy of the array.

Documentation (`.copy()`): https://numpy.org/doc/stable/reference/generated/numpy.ndarray.copy.html#numpy.ndarray.copy

Documentation: (`np.copy()`): https://numpy.org/doc/stable/reference/generated/numpy.copy.html

In [304…
```
arr = np.arange(10)
arr
```

Out[304]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [305…
```
copy_arr = arr.copy()
copy_arr
```

Out[305]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Let's modify the content of copy_arr and check whether it modified the original array as well

In [306…
```
copy_arr[3] = 45
copy_arr
```

Out[306]:
```
array([ 0,  1,  2, 45,  4,  5,  6,  7,  8,  9])
```

In [307…
```
arr
```

Out[307]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Notice that

- The content of original array were not modified as we changed our copy array.

What are object arrays ?

Object arrays are basically array of any python datatype.

Documentation: https://numpy.org/devdocs/reference/arrays.scalars.html#numpy.object_

In [308…
```
arr = np.array([1, 'm', [1,2,3]], dtype = 'object')
arr
```

Out[308]:
```
array([1, 'm', list([1, 2, 3])], dtype=object)
```

But arrays are suppoed to be homogeous data. How is it storing data of various types?

Remember that everything is object in python.

Just like python list,

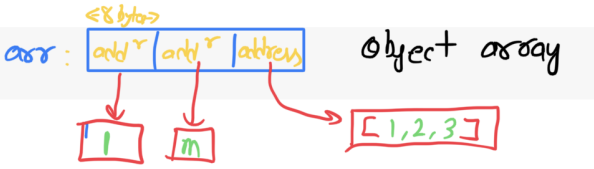
- The data actually **stored** in object arrays are **references to Python objects**, not the objects themselves.

Hence, their elements need not be of the same Python type.

As every element in array is an object. Hence, the dtype = object.

```
[ ] arr = np.array([1, 'm', [1,2,3]], dtype = 'object')
    arr

    array([1, 'm', list([1, 2, 3])], dtype=object)
```



Let's make a copy of object array and check whether it returns a shallow copy or deep copy.

In [309…
```
copy_arr = arr.copy()
```

In [310…
```
copy_arr
```

Out[310]:
```
array([1, 'm', list([1, 2, 3])], dtype=object)
```

Now, let's try to modify the list elements in copy_arr

In [311…
```
copy_arr[2][0] = 999
```

In [312…
```
copy_arr
```

Out[312]:
```
array([1, 'm', list([999, 2, 3])], dtype=object)
```

Let's see if it changed the original array as well

In [313…
```
arr
```

Out[313]:
```
array([1, 'm', list([999, 2, 3])], dtype=object)
```

It did change the original array.

Hence, `.copy()` **will return shallow copy when copying elements of array in object array.**

Any change in the 2nd level elements of array will be reflected in original array as well.

So, how do we create deep copy then ?

We can do so using `copy.deepcopy()` method

`copy.deepcopy()`

Returns the deep copy of array

Documentation: https://docs.python.org/3/library/copy.html#copy.deepcopy

In [314…
```
import copy
```

In [315…
```
arr = np.array([1, 'm', [1,2,3]], dtype = 'object')
arr
```

Out[315]:
```
array([1, 'm', list([1, 2, 3])], dtype=object)
```

Let's make a copy using `deepcopy()`

In [316…
```
copy = copy.deepcopy(arr)
```

In [317…
```
copy
```

Out[317]:
```
array([1, 'm', list([1, 2, 3])], dtype=object)
```

Let's modify the array inside copy array

In [318…
```
copy[2][0] = 999
```

In [319…
```
copy
```

Out[319]:
```
array([1, 'm', list([999, 2, 3])], dtype=object)
```

In [ ]:
```

```

In [320…
```
arr
```

Out[320]:
```
array([1, 'm', list([1, 2, 3])], dtype=object)
```

Notice that,

- the changes in copy array didn't reflect back to original array.

`copy.deepcopy()` **returns deep copy of an array.**

Summarizing

- `.view()` returns shallow copy of array
- `.copy()` returns deep copy of an array except for object type array
- `copy.deepcopy()` returns deep copy of an array.

Thank You!

In [ ]:
```

```

In [ ]:
```

```