Numpy-I Notes

Content

- **Installing and Importing Numpy**
- **Introduction to use case**
- **Motivation: Why to use Numpy? - How is it different from Python Lists?**
- **Creating a Basic Numpy Array**
  - From a List - `array()`, `shape`, `ndim`
  - From a range and stepsize - `arange()`
  - From a range and count of elements - `linspace()`
  - `type()` ndarray
- **How numpy works under the hood?**
- **2-D arrays (Matrices)**
  - `reshape()`
  - `resize()`
  - Transpose
  - Converting Matrix back to Vector - `flatten()`
- **Creating some special arrays using Numpy**
  - `zeros()`
  - `ones()`
  - `diag()`
  - `identity()`
- **Indexing and Slicing**
  - Indexing
  - Slicing
  - Masking (Fancy Indexing)

Installation Using `%pip`

```
!pip install numpy
```

```
Collecting nump
  Downloading nump-0.1.tar.gz (604 bytes)
Building wheels for collected packages: nump
  Building wheel for nump (setup.py) ... done
```

```
  Created wheel for nump: filename=nump-0.1-py3-none-any.whl size=1102 sha256=86586a
  Stored in directory: /Users/anantm/Library/Caches/pip/wheels/6e/86/8d/2383b5736f00
Successfully built nump
Installing collected packages: nump
Successfully installed nump-0.1
```

Importing Numpy

```
import numpy as np
```

Use case Introduction: Fitbit

#date	step_count	mood	calories_burned	hours_of_sleep	bool_of_active	weight_kg
06-10-2017	5464	200	181	5	0	66
07-10-2017	6041	100	197	8	0	66
08-10-2017	25	100	0	5	0	66
09-10-2017	5461	100	174	4	0	66
10-10-2017	6915	200	223	5	500	66
11-10-2017	4545	100	149	6	0	66
12-10-2017	4340	100	140	6	0	66
13-10-2017	1230	100	38	7	0	66
14-10-2017	61	100	1	5	0	66
15-10-2017	1258	100	40	6	0	65
16-10-2017	3148	100	101	8	0	65
17-10-2017	4687	100	152	5	0	65
18-10-2017	4732	300	150	6	500	65
19-10-2017	3519	100	113	7	0	65
20-10-2017	1580	100	49	5	0	65
21-10-2017	2822	100	86	6	0	65
22-10-2017	181	100	6	8	0	65
23-10-2017	3158	200	99	5	0	65

Every row is called a record or data point and every column is a feature

What kind of questions can we answer using this data?

- How many records and features are there in the dataset?
- What is the **average step count**?
- On which day the **step count was highest/lowest?**
- What's the **most frequent mood**?

We will try finding

- How daily activity affects mood?

Why use Numpy?

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

```
a = [i**2 for i in a]
```

```
print(a)
```

```
[1, 16, 81, 256, 625]
```

Same operation using NumPy

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

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

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

```
l = range(1000000)
```

```
%timeit [i**2 for i in l]
```

```
241 ms ± 2.58 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
l = np.arange(1000000)
```

```
%timeit l**2
```

```
331 µs ± 2.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Takeaway ?

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

Basic arrays in NumPy

```
array()
```

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

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

What will be the dimension of this array?

```
arr1.ndim
```

```
1
```

Shape of array

```
arr1.shape
```

```
(3,)
```

Sequences in Numpy

From a range and stepsize - `arange()`

- `arange(start, end, step)`

```
[ ] ↳ 4 cells hidden
```

What if we want to generate equally spaced points?

`=> linspace()`

```
[ ] ↳ 5 cells hidden
```

How numpy works under the hood?

Numpy is written in C which allows it to **manage memory very efficiently**

```
↳ 2 cells hidden
```

C type behaviour of Numpy

```
[ ] ↳ 9 cells hidden
```

Working with 2-D arrays (Matrices)

```
m1 = np.array([[1,2,3],[4,5,6]])
m1
# Nicely printing out in a Matrix form
```

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

Shape of a numpy array?

```
m1.shape # arr1 has 3 elements
```

```
(2, 3)
```

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

Tuple

▸ Now, What is the dimension of this array?

```
[ ] ↳ 1 cell hidden
```

▾ How can we create high dimensional arrays using `reshape()` ?

```
m2 = np.arange(1, 13)
m2
```

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

▾ Can we make `m2` a 4×4 array?

```
m2 = np.arange(1, 13)
m2.reshape(4, 4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-25-fc70b006b379> in <module>
      1 m2 = np.arange(1, 13)
----> 2 m2.reshape(4, 4)

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

SEARCH STACK OVERFLOW

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

- 4×3
- 3×4
- 6×2
- 2×6
- 1×12
- 12×1

```
m2 = np.arange(1, 13)
m2.reshape(4, 3)
```

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

```
m2 = np.arange(1, 13)
m2
```

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

```
m2.shape
```

```
(12,)
```

```
m2.reshape(12, 1)
```

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

- `(12,)` means its a **1D array**
- `(12, 1)` means its a **2D array** with 12 rows and 1 column

Resize

```
a = np.arange(4)
a.resize((2,4))
a
```

```
array([[0, 1, 2, 3],
       [0, 0, 0, 0]])
```

difference between resize and reshape?

The difference is that it'll add extra zeros to it if shape exceeds number of elements. However, there is a catch: it'll throw an error if array is referenced somewhere and you try resizing it

```
b = a
a.resize((10,))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-87-6e04a2659d2b> in <module>()
      1 b = a
----> 2 a.resize((10,))

ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
```

SEARCH STACK OVERFLOW

Transpose

- **Change rows into columns and columns into rows**

[ ] ↳ *5 cells hidden*

Flattening of an array

```
A = np.arange(12).reshape(3, 4)
A
```

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

```
A.flatten()
```

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

convert a matrix to 1D array using `reshape()`

What should I pass in `A.reshape()` if I want to use it to convert `A` to 1D vector?

- **(1, 1)?**

[ ] ↳ *5 cells hidden*

What will happen if we pass a negative integer in `reshape()` ?

[ ] ↳ *3 cells hidden*

Special arrays using Numpy

numpy array with all zeros

```
np.zeros(3)
# Pass in how many values you need in array
# All values will be zeroes
```

```
array([0., 0., 0.])
```

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

numpy array with all ones

[ ] ↳ *2 cells hidden*

Now, do we need `np.twos()`, `np.threes()`, `np.fours()`, ... `np.hundreds()`?

- We can just create array using `np.ones()` and multiply with required value

[ ] ↳ *4 cells hidden*

Diagional matrices

```
np.diag([1, 2, 3])
# We pass values for diagonal elements as a list
# All other elements are zero
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Identity matrix

square matrix where **all diagonal values are 1** and **All non-diagonal values are 0**

```
np.identity(3)
# Pass in the single dimension of required square identity matrix
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Indexing and Slicing upon Numpy arrays

```
m1 = np.arange(12)
m1
```

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

Indexing in np arrays

```
m1[0] # gives first element of array
```

```
0
```

```
m1[12] # out of index Error
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-18-24d969f9df5e> in <module>()
----> 1 m1[12] # out of index Error

IndexError: index 12 is out of bounds for axis 0 with size 12
```

SEARCH STACK OVERFLOW

```
m1 = np.arange(1,10).reshape((3,3))
```

```
m1
```

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

```
m1[1][2]
```

```
6
```

```
m1[1, 2] #m1[row, column] (another way of indexing - using comma)
```

```
6
```

list of indexes in numpy

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

```
m1[[2,3,4,1,2,2]]
```

```
array([300, 400, 500, 200, 300, 300])
```

List of indexes in 2D array

```
import numpy as np
```

```
m1 = np.arange(9).reshape((3,3))
```

```
m1
```

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

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

```
array([0, 4, 8])
```

Slicing

```
m1 = np.arange(12)
m1
```

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

```
m1[:5]
```

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

Can we just get this much of our array `m1`?

```
[[5, 6],
 [9, 10]]
```

Remember our `m1` is:

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

```
m1 = np.arange(12).reshape(3,4)
```

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

```
array([[ 5,  6],
       [ 9, 10]])
```

What if I want this much part?

```
[[2, 3],
 [6, 7],
 [10,11]]
```

[ ] ↳ *1 cell hidden*

What if I need 1st and 3rd column?

```
[[1, 3], [5, 7], [9,11]]
```

```
# Get all rows
# Then get columns from 1 to all with step of 2

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

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

```
# Get all rows
# Then get columns 1 and 3

m1[:, (1,3)] #can also pass indices of required columns as a tuple to get the sam result
```

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

Fancy indexing (Masking)

- Numpy arrays can be indexed with boolean arrays (masks).
- It creates copies not views.

```
m1 = np.arange(12).reshape(3, 4)
m1 < 6
```

```
array([[ True,  True,  True,  True],
       [ True,  True, False, False],
       [False, False, False, False]])
```

```
m1[m1 < 6]
# Value corresponding to True is retained
# Value corresponding to False is filtered out
```

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

```
m1
```

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

```
m1[m1%2 == 0]
```

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

Takeaway?

Matrix gets converted into a 1D array after masking because the filtering operation **implicitly converts high-dimensional array into 1D array** as it **cannot retain its 3×4 with lesser number of elements**

Multiple filter conditions

```
a = np.arange(11)
```

```
a[(a %2 == 0) | (a%5 == 0)] # filter elements which are multiple of 2 or 3
```

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