

## ✓ Numpy 2

---

### ✓ Content

- Working with 2D arrays (Matrices)
    - Transpose
    - Indexing
    - Slicing
    - Fancy Indexing (Masking)
  - Aggregate Functions
  - Logical Operations
    - `np.any()`
    - `np.all()`
    - `np.where()`
  - Use Case: Fitness data analysis
- 

### ✓ Working with 2D arrays (Matrices)

Let's create an array -

```
import numpy as np
a = np.array(range(16))
a

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

What will be its shape and dimensions?

```
a.shape

(16,)
```

```
a.ndim

1
```

#### ✓ How can we convert this array to a 2-dimensional array?

- Using `reshape()`

For a 2D array, we will have to specify the followings :-

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

Let's try converting it into a 8x2 array.

```
a.reshape(8, 2)

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

Let's try converting it into a 4x4 array.

```
a.reshape(4, 4)
```

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

```
a.reshape(4, 5)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-7-05ad01dfd0f5> in <cell line: 1>()
----> 1 a.reshape(4, 5)

ValueError: cannot reshape array of size 16 into shape (4,5)
```

SEARCH STACK OVERFLOW

### This will give an Error. Why?

- We have 16 elements in `a`, but `reshape(4, 5)` is trying to fill in  $4 \times 5 = 20$  elements.
- Therefore, whatever the shape we're trying to reshape to, must be able to incorporate the number of elements that we have.

```
a.reshape(8, -1)
```

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

Notice that Python automatically figured out what should be the replacement of `-1` argument, given that the first argument is `8`.

We can also put `-1` as the first argument. As long as one argument is given, it will calculate the other one.

### What if we pass both args as `-1`?

```
a.reshape(-1, -1)
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-9-decf4fe03d74> in <cell line: 1>()
----> 1 a.reshape(-1, -1)

ValueError: can only specify one unknown dimension
```

SEARCH STACK OVERFLOW

- You need to give at least one dimension.

Let's save `a` as a  $8 \times 2$  array (matrix) for now.

```
a = a.reshape(8, 2)
```

### What will be the length of `a`?

- It will be 8, since it contains 8 lists as its elements.
- Each of these lists have 2 elements, but that's a different thing.

**Explanation:** `len(nd array)` will give you the magnitude of first dimension

```
len(a)
```

```
8
```

```
len(a[0])
```

```
2
```

## ✓ Transpose

Let's create a 2D numpy array.

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

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

```
a.shape
```

```
(3, 4)
```

There is another operation on a multi-dimensional array, known as **Transpose**.

It basically means that the no. of rows is interchanged by no. of cols, and vice-versa.

```
a.T
```

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

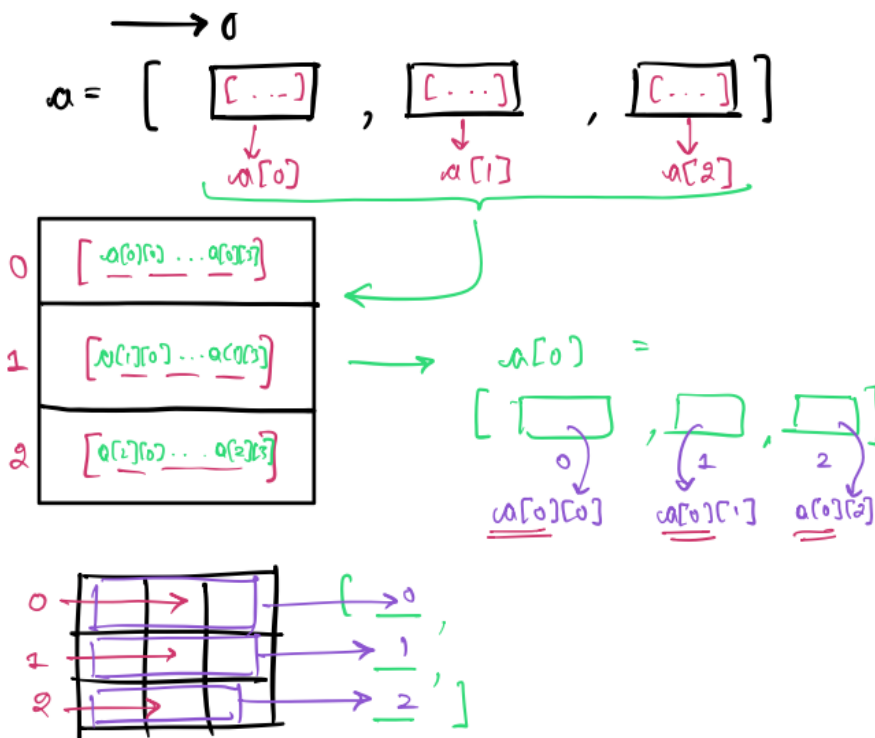
Let's verify the shape of this transpose array -

```
a.T.shape
```

```
(4, 3)
```

## ✓ Indexing in 2D arrays

- Similar to Python lists



a

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

**Can we extract just the element 6 from a ?**

```
# Accessing 2nd row and 3rd col -
a[1, 2]
```

6

This can also be written as

```
a[1][2]
```

6

---

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

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

**What will be the output of this?**

```
m1[1, 1] # m1[row,column]
```

5

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

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

**Will this work now?**

```
m1[2, 3]
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-23-963ce94bbe14> in <cell line: 1>()
----> 1 m1[2, 3]
```

**IndexError:** too many indices for array: array is 1-dimensional, but 2 were indexed

SEARCH STACK OVERFLOW

Note:

- Since m1 is a 1D array, this will not work.
- This is because there are no row and column entity here.

Therefore, you cannot use the same syntax for 1D arrays, as you did with 2D arrays, and vice-versa.

However with a little tweak in this code, we can access elements of m1 at different positions/indices.

```
m1[[2, 3]]
```

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

---

✓ **How will you print the diagonal elements of the following 2D array?**

```
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])
```

When list of indexes is provided for both rows and cols, for example: `m1[[0,1,2],[0,1,2]]`

It selects individual elements i.e. `m1[0][0]`, `m1[1][1]` and `m1[2][2]`.

---

## ✓ Slicing in 2D arrays

- We need to **provide two slice ranges**, one for **row** and one for **column**.
- We can also **mix Indexing and Slicing**

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

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

```
m1[:2] # gives first two rows
```

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

## ✓ How can we get columns from a 2D array?

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

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

```
m1[:, 1:3] # gives 2nd and 3rd col
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

---

## ✓ Fancy Indexing (Masking) in 2D arrays

We did this for one dimensional arrays. Let's see if those concepts translate to 2D also.

Suppose we have the matrix `m1` -

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

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

**What will be output of following?**

```
m1 < 6
```

```
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.**

**Condition(s) will be passed instead of indices and slice ranges.**

```
m1[m1 < 6]
array([0, 1, 2, 3, 4, 5])
```

- Values corresponding to True are retained
  - Values corresponding to False are filtered out
- 

## ✓ Aggregate Functions

Numpy provides various universal functions that cover a wide variety of operations and perform **fast element-wise array operations**.

### ✓ How would calculate the sum of elements of an array?

```
np.sum()
```

- It sums all the values in a numpy array.

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

```
np.sum(a)
55
```

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

```
np.mean()
```

- It gives the us mean of all values in a numpy array.

```
np.mean(a)
5.5
```

### ✓ Now, we want to find the minimum / maximum value in the array.

```
np.min() / np.max()
```

```
np.min(a)
1
```

```
np.max(a)
10
```

Let's apply aggregate functions on 2D array.

```
a = np.arange(12).reshape(3, 4)
a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.sum(a) # sums all the values present in the array

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**
- Summation of values happen **in the vertical direction**.
- Rows collapse/merge when we do axis=0.

```
np.sum(a, axis=0)

array([12, 15, 18, 21])
```

#### ✓ 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**
- Summation of values happen **in the horizontal direction**.
- Columns collapse/merge when we do axis=1.

```
np.sum(a, axis=1)

array([ 6, 22, 38])
```

---

### ✓ Logical Operations

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

```
np.any()
```

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

Imagine you have a shopping list with items you need to buy, but you're not sure if you have enough money to buy everything.

You want to check if there's at least one item on your list that you can afford.

In this case, you can use np.any:

```
import numpy as np

# Prices of items on your shopping list
prices = np.array([50, 45, 25, 20, 35])

# Your budget
budget = 30

# Check if there's at least one item you can afford
can_afford = np.any(prices <= budget)

if can_afford:
    print("You can buy at least one item on your list!")
else:
    print("Sorry, nothing on your list fits your budget.")

    You can buy at least one item on your list!
```

---

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

`np.all()`

- returns True if **all the elements** in the argument arrays follow the **provided condition**.

Let's consider a scenario where you have a list of chores, and you want to make sure all the chores are done before you can play video games.

You can use `np.all` to check if all the chores are completed.

```
import numpy as np

# Chores status: 1 for done, 0 for not done
chores = np.array([1, 1, 1, 1, 0])

# Check if all chores are done
all_chores_done = np.all(chores == 1)

if all_chores_done:
    print("Great job! You've completed all your chores. Time to play!")
else:
    print("Finish all your chores before you can play.")
```

Finish all your chores before you can play.

**Multiple conditions for `.all()` function -**

```
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()

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?**

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

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

You can use masking to update the array.

```
arr[arr > 0] = 1
arr [arr < 0] = -1

arr

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

There's also a numpy function which can help us with it.

✓ `np.where()`

- Syntax: `np.where(condition, [x, y])`
- returns an ndarray whose elements are chosen from `x` or `y` depending on condition.

Suppose you have a list of product prices, and you want to apply a **10%** discount to all products with prices above **\$50**.



You can use `np.where` to adjust the prices.

```
import numpy as np

# Product prices
prices = np.array([45, 55, 60, 75, 40, 90])

# Apply a 10% discount to prices above $50
discounted_prices = np.where(prices > 50, prices * 0.9, prices)

print("Original prices:", prices)
print("Discounted prices:", discounted_prices)

Original prices: [45 55 60 75 40 90]
Discounted prices: [45.  49.5 54.  67.5 40.  81. ]
```

**Notice that it didn't change the original array.**

---

## ✓ Use Case: Fitness data analysis


Imagine you are a Data Scientist at Fitbit

You've been given a user data to analyse and find some insights which can be shown on the smart watch.

But why would we want to analyse the user data for designing the watch?

These insights from the user data can help business make customer oriented decision for the product design.

Let's first look at the data we have gathered.

 **fit.txt**

#date	step_count	mood	calories_burned	hours_of_sleep	active
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
11-10-2017	4545	Sad	149	6	Inactive
12-10-2017	4340	Sad	140	6	Inactive
13-10-2017	1230	Sad	38	7	Inactive
14-10-2017	61	Sad	1	5	Inactive
15-10-2017	1258	Sad	40	6	Inactive
16-10-2017	3148	Sad	101	8	Inactive
17-10-2017	4687	Sad	152	5	Inactive
18-10-2017	4732	Happy	150	6	Active
19-10-2017	3519	Sad	113	7	Inactive
20-10-2017	1580	Sad	49	5	Inactive
21-10-2017	2822	Sad	86	6	Inactive
22-10-2017	181	Sad	6	8	Inactive
23-10-2017	3158	Neutral	99	5	Inactive
24-10-2017	4383	Neutral	143	4	Inactive
25-10-2017	3881	Neutral	125	5	Inactive
26-10-2017	4037	Neutral	129	6	Inactive

Notice that our data is structured in a tabular format.

- Each column is known as a feature.
- Each row is known as a record.

## ✓ Basic EDA

Performing **Exploratory Data Analysis (EDA)** is like being a detective for numbers and information.

Imagine you have a big box of colorful candies. EDA is like looking at all the candies, counting how many of each color there are, and maybe even making a pretty picture to show which colors you have the most of. This way, you can learn a lot about your candies without eating them all at once!



So, EDA is about looking at your things, which is data in this case, to understand them better and find out interesting stuff about them.

Formally defining, Exploratory Data Analysis (EDA) is a process of **examining**, **summarizing**, and **visualizing** data sets to understand their main characteristics, uncover patterns that helps analysts and data scientists gain insights into the data, make informed decisions, and guide further analysis or modeling.

First, we will import numpy.

```
import numpy as np
```

Let's load the data that we saw earlier.

- For this, we will use the `.loadtxt()` function.

```
!gdown https://drive.google.com/uc?id=1vk1Pu0djiYcrdc85yUXZ_Rqq2oZNcohd
```

Downloading...

From: [https://drive.google.com/uc?id=1vk1Pu0djiYcrdc85yUXZ\\_Rqq2oZNcohd](https://drive.google.com/uc?id=1vk1Pu0djiYcrdc85yUXZ_Rqq2oZNcohd)

To: /content/fit.txt

100% 3.43k/3.43k [00:00<00:00, 11.3MB/s]

```
data = np.loadtxt('/content/fit.txt', dtype='str')
data
```

```
[ '14-12-2017', '1422', 'happy', '243', '5', 'Active'],
[ '15-12-2017', '437', 'Neutral', '14', '3', 'Active'],
[ '16-12-2017', '1231', 'Neutral', '39', '4', 'Active'],
[ '17-12-2017', '1696', 'Sad', '55', '4', 'Inactive'],
[ '18-12-2017', '4921', 'Neutral', '158', '5', 'Active'],
[ '19-12-2017', '221', 'Sad', '7', '5', 'Active'],
[ '20-12-2017', '6500', 'Neutral', '213', '5', 'Active'],
[ '21-12-2017', '3575', 'Neutral', '116', '5', 'Active'],
[ '22-12-2017', '4061', 'Sad', '129', '5', 'Inactive'],
[ '23-12-2017', '651', 'Sad', '21', '5', 'Inactive'],
[ '24-12-2017', '753', 'Sad', '28', '4', 'Inactive'],
[ '25-12-2017', '518', 'Sad', '16', '3', 'Inactive'],
[ '26-12-2017', '5537', 'Happy', '180', '4', 'Active'],
[ '27-12-2017', '4108', 'Neutral', '138', '5', 'Active'],
[ '28-12-2017', '5376', 'Happy', '176', '5', 'Active'],
[ '29-12-2017', '3066', 'Neutral', '99', '4', 'Active'],
[ '30-12-2017', '177', 'Sad', '5', '5', 'Inactive'],
[ '31-12-2017', '36', 'Sad', '1', '3', 'Inactive'],
[ '01-01-2018', '299', 'Sad', '10', '3', 'Inactive'],
[ '02-01-2018', '1447', 'Neutral', '47', '3', 'Inactive'],
[ '03-01-2018', '2599', 'Neutral', '84', '2', 'Inactive'],
[ '04-01-2018', '702', 'Sad', '23', '3', 'Inactive'],
[ '05-01-2018', '133', 'Sad', '4', '2', 'Inactive'],
[ '06-01-2018', '153', 'Happy', '0', '8', 'Inactive'],
[ '07-01-2018', '500', 'Neutral', '0', '5', 'Active'],
[ '08-01-2018', '2127', 'Neutral', '0', '5', 'Inactive'],
[ '09-01-2018', '2203', 'Happy', '0', '5', 'Active']], dtype='<U10')

```

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

What's the shape of this data?

```
data.shape

(96, 6)
```

What's the dimensionality?

```
data.ndim

2
```

We can see that this is a 2-dimensional list.

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 homogenous containing all the data as strings.**

In order to work with strings, categorical data and numerical data, we'll have to save every feature separately.

**How will we extract features in separate variables?**

For that, we first need some idea on how data is saved.

Let's see what's the first element of the data.

```
data[0]

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

Hmm.. this extracts a row, not a column.

Similarly, we can extract other specific rows.

```
data[1]
```

```
array(['07-10-2017', '6041', 'Sad', '197', '8', 'Inactive'], dtype='<U10')
```

We can also use slicing.

```
data[:5]

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')
```

Now, we want to place all the **dates** into a single entity.

#### How to do that?

- One way is to just go ahead and fetch the column number 0 from all rows.
- Another way is to, take a transpose of data .

Let's see them both -

#### Approach 1

```
data[:, 0]

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')
```

This gives all the dates.

#### Approach 2

```
data_t = data.T
```

Don't you think all the dates will now be present in the first (i.e. index 0th element) of data\_t ?

```
data_t[0]

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')
```

Also, what will be the shape of data\_t ?

```
data_t.shape
```

```
(6, 96)
```

✓ **Let's extract all the columns and save them in separate variables.**

```
date, step_count, mood, calories_burned, hours_of_sleep, activity_status = data.T
```

```
step_count
```

```
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')
```

```
step_count.dtype
```

```
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**

```
step_count = np.array(step_count, dtype='int')
```

```
step_count.dtype
```

```
dtype('int64')
```

```
step_count
```

```
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])
```

**What will be shape of this array?**

```
step_count.shape
```

```
(96,)
```

- We saw in last class that since it is a 1D array, its shape will be (96, ).

- If it were a 2D array, its shape would've been (96, 1) .

### Calories Burned

```
calories_burned = np.array(calories_burned, dtype='int')
calories_burned.dtype

dtype('int64')
```

### Hours of Sleep

```
hours_of_sleep = np.array(hours_of_sleep, dtype='int')
hours_of_sleep.dtype

dtype('int64')
```

### Mood

Mood belongs to categorical data type. As the name suggests, categorical data type has two or more categories in it.

Let's check the values of mood variable -

```
mood

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', 'Happy', 'Neutral', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', 'Sad', 'Happy', 'Happy',
       'Happy', 'Happy', 'Happy', 'Happy', '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', 'Sad', 'Happy', 'Neutral', 'Happy', 'Neutral', 'Sad', 'Sad',
       'Sad', 'Neutral', 'Neutral', 'Sad', 'Sad', 'Happy', 'Neutral',
       'Neutral', 'Happy'], dtype='<U10')
```

```
np.unique(mood)

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

### Activity Status

```
activity_status

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')
```

Since we've extracted from the same source array, we know that

- mood[0] and step\_count[0]
- There is a connection between them, as they belong to the same record.

Also, we know that their length will be the same, i.e. 96

Now let's look at something really interesting.

**Can we extract the step counts, when the mood was Happy?**

```
step_count_happy = step_count[mood == 'Happy']
```

```
len(step_count_happy)
```

```
40
```

Let's also find for when the mood was Sad.

```
step_count_sad = step_count[mood == 'Sad']
```

```
step_count_sad
```

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

```
len(step_count_sad)
```

```
29
```

Let's do the same for when the mood was Neutral.

```
step_count_neutral = step_count[mood == 'Neutral']
```

```
step_count_neutral
```

```
array([5464, 6915, 3158, 4383, 3881, 4037, 202, 292, 2209, 6041, 570,
       1163, 2374, 2909, 7102, 3941, 437, 1231, 4921, 6500, 3575, 4108,
       3066, 1447, 2599, 500, 2127])
```

```
len(step_count_neutral)
```

```
27
```

**How can we collect data for when the mood was either happy or neutral?**

```
step_count_happy_or_neutral = step_count[(mood == 'Neutral') | (mood == 'Happy')]
```

```
step_count_happy_or_neutral
```

```
array([5464, 6915, 4732, 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, 3608, 774, 1421, 4064,
       2725, 5934, 1867, 2374, 2909, 7102, 3941, 7422, 437, 1231, 4921,
       6500, 3575, 5537, 4108, 5376, 3066, 1447, 2599, 153, 500, 2127,
       2203])
```

```
len(step_count_happy_or_neutral)
```

```
67
```

**Let's try to compare step counts on bad mood days and good mood days.**

```
# Average step count on Sad mood days -
```

```
np.mean(step_count_sad)
```

```
2103.0689655172414
```

```
# Average step count on Happy days -
```

```
np.mean(step_count_happy)
```

```
3392.725
```

```
# Average step count on Neutral days -
```

```
np.mean(step_count_neutral)
```

```
3153.7777777777778
```

As you can see, this data tells us a lot about user behaviour.

This way we can analyze data and learn.

This is just the second class on numpy, we will learn many more concepts related to this, and pandas also.

**Let's try to check the mood when step count was greater/lesser.**

```
# mood when step count > 4000
np.unique(mood[step_count > 4000], return_counts = True)
(array(['Happy', 'Neutral', 'Sad'], dtype='<U10'), array([22,  9,  7]))
```

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

```
# mood when step count <= 2000
np.unique(mood[step_count <= 2000], return_counts = True)
(array(['Happy', 'Neutral', 'Sad'], dtype='<U10'), array([13,  8, 18]))
```

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

✓ **This suggests that there may be a correlation between the Mood and Step Count .**

---