Content

- Introduction to DAV
- Python Lists vs Numpy Array
    - Importing Numpy
    - Why use Numpy?
- Dimension & Shape
- Type Conversion in Numpy Arrays
- Indexing & Slicing
- NPS use case

Introduction to DAV (Data Analysis and Visualization) Module

It will contain 3 sections -

1. DAV-1: Python Libraries

- Numpy
- Pandas
- Matplotlib & Seaborn

2. DAV-2: Probability Statistics
3. DAV-3: Hypothesis Testing

Python Lists vs Numpy Arrays

Homogeneity of data

So far, we've been working with Python lists, that can have **heterogenous data**.

In [1]:

```
a = [1, 2, 3, "Michael", True]
a
```

Out[1]:

```
[1, 2, 3, 'Michael', True]
```

Because of this hetergenity, in Python lists, the data elements are not stored together in the memory (RAM).

- Each element is stored in a different location.
- Only the address of each of the element will be stored together.
- So, a list is actually just referencing to these different locations, in order to access the actual element.

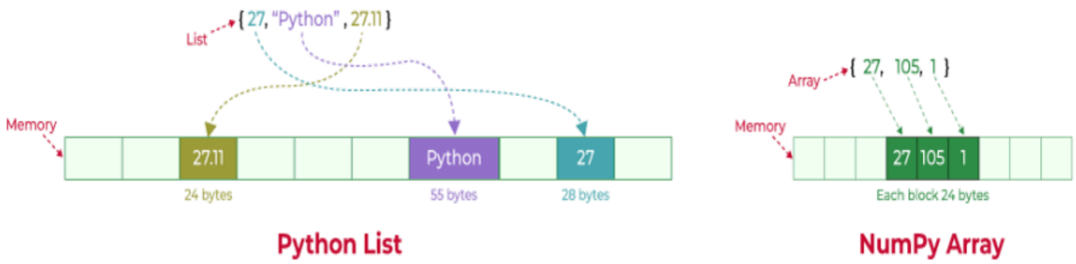
On the other hand, Numpy only stores **homogenous data**, i.e. a numpy array cannot contain mixed data types.

It will either

- ONLY contain integers
- ONLY contain floats
- ONLY contain characters

... and so on.

Because of this, we can now store these different data items together, as they are of the same type.



Speed

Programming languages can also be slow or fast.

In fact,

- Java is a decently fast language.
- Python is a slow language.
- C, one of the earliest available languages, is super fast.

This is because C has concepts like memory allocation, pointers, etc.

How is this possible?

With Numpy, though we will be writing our code using Python, but behind the scene, all the code is written in the **C programming language**, to make it faster.

Because of this, a Numpy Array will be significantly faster than a Python List in performing the same operation.

This is very important to us, because in data science, we deal with huge amount of data.

Properties

- **In-built Functions**
- For a Python list `a` , we had in-built functions like `.sum(a)` , etc.
- For NumPy arrays also, we will have such in-built functions.
- **Slicing**
- Recall that we were able to perform list slicing.
- All of that is still applicable here.

Recall how we used to import a module/library in Python.

- In order to use Python Lists, we do not need to import anything extra.
- However to use Numpy Arrays, we need to import it into our environment, as it is a Library.

Generally, we do so while using the alias **np** .

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

Note:

- In this terminal, we will already have numpy installed as we are working on Google Colab
- However, when working on an evironment that does not have it installed, you'll have to install it the first time working.
- This can be done with the command: `!pip install numpy`

Why use Numpy? - Time Comparison

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

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

```
In [4]: type(a)
```

```
Out[4]: list
```

The basic approach here would be to iterate over the list and square each element.

```
In [5]: res = [i**2 for i in a]
        print(res)
```

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

Let's try the same operation with Numpy.

To do so, first of all we need to define the Numpy array.

We can convert any list `a` into a Numpy array using the `array()` function.

```
In [6]: b = np.array(a)
        b
```

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

```
In [7]: type(b)
```

```
Out[7]: numpy.ndarray
```

- `nd` in `numpy.ndarray` stands for **n-dimensional**

Now, how can we get the square of each element in the same Numpy array?

```
In [8]: b**2
```

```
Out[8]: array([ 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, let's measure the time for these operations.
- We will use `%timeit` .

```
In [9]: l = range(1000000)
```

```
In [10]: %timeit [i**2 for i in l]
```

```
31.3 ms ± 2.68 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

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

Let's peform the same operation using Numpy arrays -

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

```
In [11]: l = np.array(range(1000000))
```

```
In [12]: %timeit l**2
```

```
567 µs ± 29.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loop
s each)
```

Notice that it only took 900 μs per loop time for the numpy operation.

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

- Numpy array is densely packed in memory due to it's **homogenous** type.
- Numpy functions are implemented in **C programming launguage**.
- Numpy is able to divide a task into multiple subtasks and process them **parallelly.**

Dimensions and Shape

We can get the dimension of an array using the `ndim` property.

In [13]:
```
arr1 = np.array(range(1000000))
arr1.ndim
```

Out[13]:
```
1
```

Numpy arrays have another property called `shape` that tells us number of elements across every dimension.

In [14]:
```
arr1.shape
```

Out[14]:
```
(1000000,)
```

This means that the array `arr1` has 1000000 elements in a single dimension.

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

In [15]:
```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [10, 11, 12]])
print(arr2)
```

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

In [16]:
```
arr2.ndim
```

Out[16]:
```
2
```

In [17]:
```
arr2.shape
```

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

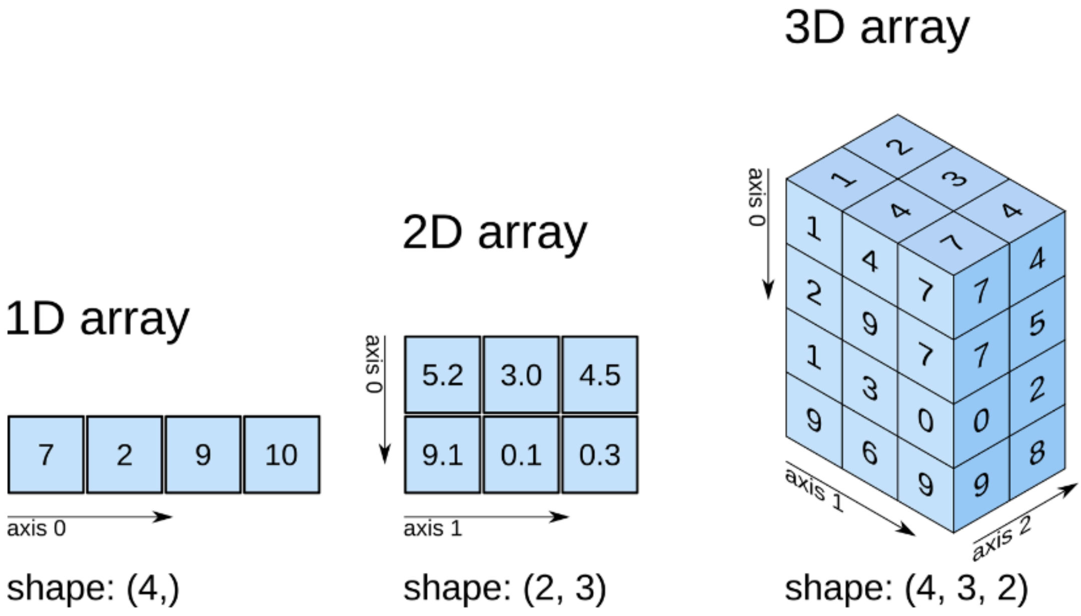
`ndim` specifies the number of dimensions of the array i.e. 1D (1), 2D (2), 3D (3) and so on.

`shape` returns the exact shape in all dimensions, that is (3,3) which implies 3 in axis 0 and 3 in axis 1.

In [18]:

```
from IPython.display import Image
Image(filename='download.png')
```

Out[18]:



`np.arange()`

Let's create some sequences in Numpy.

We can pass **starting** point, **ending** point (not included in the array) and **step-size**.

Syntax:

- `arange(start, end, step)`

In [19]:

```
arr2 = np.arange(1, 5)
arr2
```

Out[19]:

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

In [20]:

```
arr2_step = np.arange(1, 5, 2)
arr2_step
```

Out[20]:

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

`np.arange()` behaves in the 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 [21]: arr3 = np.arange(1, 5, 0.5)
         arr3

Out[21]: array([1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

Type Conversion in Numpy Arrays

For this, let's pass a **float** as one of the values in a **numpy array**.

```
In [22]: arr4 = np.array([1, 2, 3, 4])
         arr4

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

```
In [23]: arr4 = np.array([1, 2, 3, 4.0])
         arr4

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

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

Similarly, what will happen when we run the following code? Will it give an error?

```
In [24]: np.array(["Harry Potter", 1, 2, 3])

Out[24]: array(['Harry Potter', '1', '2', '3'], dtype='<U21')
```

No. It will convert all elements of the array to `char` type.

There's a `dtype` parameter in the `np.array()` function.

What if we set the `dtype` of array containing `integer` values to `float` ?

```
In [25]: arr5 = np.array([1, 2, 3, 4])
         arr5

Out[25]: array([1, 2, 3, 4])
```

```
In [26]: arr5 = np.array([1, 2, 3, 4], dtype="float")
         arr5

Out[26]: array([1., 2., 3., 4.])
```

Question: What will happen in the following code?

```
In [27]: np.array(["Shivank", "Bipin", "Ritwik"], dtype=float)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent
call last)
Cell In[27], line 1
----> 1 np.array(["Shivank", "Bipin", "Ritwik"], dtype=float)

ValueError: could not convert string to float: 'Shivank'
```

Since it is not possible to convert strings of alphabets to floats, it will naturally return an Error.

We can also convert the data type with the `astype()` method.

```
In [28]: arr = np.array([10, 20, 30, 40, 50])
         arr
```

```
Out[28]: array([10, 20, 30, 40, 50])
```

```
In [29]: arr = arr.astype('float64')
         print(arr)
```

```
[10. 20. 30. 40. 50.]
```

Indexing

- Similar to Python lists

```
In [30]: m1 = np.arange(12)
         m1
```

```
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
In [31]: m1[0] # gives first element of array
```

```
Out[31]: 0
```

```
In [32]: m1[-1] # negative indexing in numpy array
```

```
Out[32]: 11
```

You can also use list of indexes in numpy.

```
In [33]: m1 = np.array([100,200,300,400,500,600])
```

```
In [34]: m1[[2,3,4,1,2,2]]
```

```
Out[34]: array([300, 400, 500, 200, 300, 300])
```

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

Note:

- If you want to extract multiple indices, you need to use two sets of square brackets `[[ ]]`
    - Otherwise, you will get an error.
- Because it is only expecting a single index.
- For multiple indices, you need to pass them as a list.

```
In [35]: m1[2,3,4,1,2,2]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[35], line 1
----> 1 m1[2,3,4,1,2,2]

IndexError: too many indices for array: array is 1-dimensional, but 6 were indexed
```

Slicing

- Similar to Python lists

```
In [36]: m1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
         m1
```

```
Out[36]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [37]: m1[:5]
```

```
Out[37]: array([1, 2, 3, 4, 5])
```

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

In [38]:
```
m1[-5:-1]
```

Out[38]:
```
array([6, 7, 8, 9])
```

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

In [39]:
```
m1[-5: -1: -1]
```

Out[39]:
```
array([], dtype=int64)
```

Fancy Indexing (Masking)

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

What would happen if we do this?

In [40]:
```
m1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
m1 < 6
```

Out[40]:
```
array([ True,  True,  True,  True,  True, False, False, False, Fa
lse,
       False])
```

Comparison operation also happens on each element.

- All the values before 6 return `True`
- All the values after 6 return `False`

Question: What will be the output of the following?

In [41]:
```
m1[[True,  True,  True,  True,  True, False, False, False, False,
```

Out[41]:
```
array([1, 2, 3, 4, 5])
```

Notice that we are passing a list of indices.

- For every instance of `True` , it will print the corresponding index.
- Conversely, for every `False` , it will skip the corresponding index, and not print it.

So, this becomes a **filter** of sorts.

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

Condition will be passed instead of indices and slice ranges.

```
In [42]: m1[m1 < 6]
```

```
Out[42]: array([1, 2, 3, 4, 5])
```

This is known as **Fancy Indexing** in Numpy.

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

```
In [43]: m1[m1%2 == 0]
```

```
Out[43]: array([ 2,  4,  6,  8, 10])
```

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?

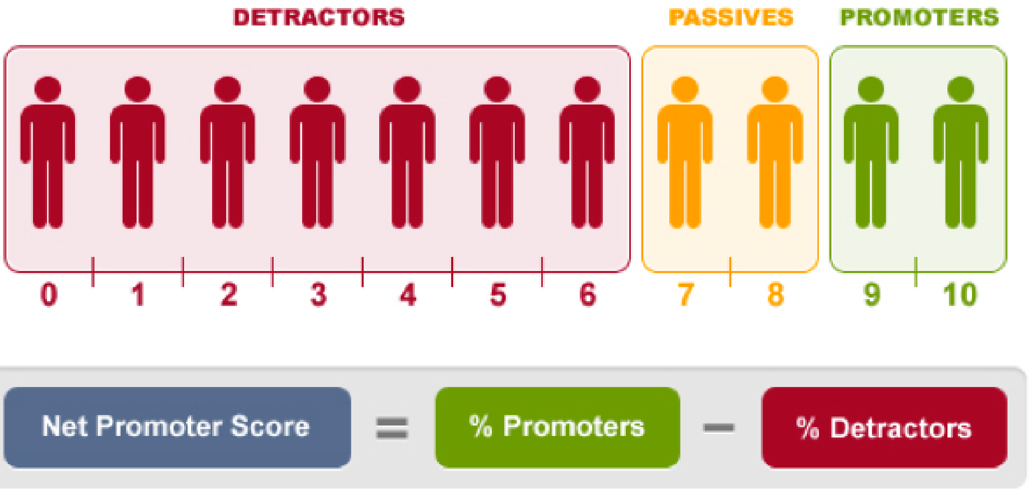
Have you all seen that every month, you get a survey form from airbnb?

- This form asks you to fill in feedback regarding how you are liking the services of airbnb in terms of a numerical score.
- This is known as the **Likelihood to Recommend Survey**.
- It is widely used by different companies and service providers to evaluate their performance and customer satisfaction.
- 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**.

```
In [44]: from IPython.display import Image
         Image(filename='nps.png')
```

Out[44]:



We label our responses into 3 categories:

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

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

## Range of NPS

- If all people are promoters (rated 9-10), we get $100$ NPS
- Conversely, if all people are detractors (rated 0-6), we get $-100$ NPS
- Also, if all people are neutral (rated 7-8), we get a $0$ NPS

Therefore, the range of NPS lies between $[-100, 100]$

Generally, each company targets to get at least a threshold NPS.

- this is a score of 70.
- This means that if $NPS > 70$, it is great performance of the company.

Naturally, this varies from business to business.

## 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.

#### 2/3 of Fortune 500 companies use NPS

\

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

Loading the data -

- For this we will use the `.loadtxt() function`
- We provide file name along with the dtype of data that we want to load.
- Documentation: https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html (https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html)
```

In [45]:
```
score = np.loadtxt('survey.txt', dtype ='int')
```

Let's check the type of this data variable `score` -

In [46]:
```
type(score)
```

Out[46]:
```
numpy.ndarray
```

Let's see what the data looks like -

In [47]:
```
score[:5]
```

Out[47]:
```
array([ 7, 10,  5,  9,  9])
```

Let's check the number of responses -

In [48]:
```
score.shape
```

Out[48]:
```
(1167,)
```

There are a total of 1167 responses for the LTR survey.

Now, let's calculate NPS using these response.

NPS = % Promoters - % Detractors

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 [49]:
```
detractors = score[score <= 6]
```

```
In [50]: # Number of detractors -

         num_detractors = len(detractors)
         num_detractors

Out[50]: 332
```

Promoters have a score >= 9

```
In [51]: promoters = score[score >= 9]
```

```
In [52]: # Number of promoters -

         num_promoters = len(promoters)
         num_promoters

Out[52]: 609
```

```
In [53]: total = len(score)
         total

Out[53]: 1167
```

```
In [54]: # % of detractors -

         percentage_detractors = (num_detractors/total) * 100
         percentage_detractors

Out[54]: 28.449014567266495
```

```
In [55]: # % of promoters -

         percentage_promoters = (num_promoters/total) * 100
         percentage_promoters

Out[55]: 52.185089974293064
```

```
In [56]: nps = percentage_promoters - percentage_detractors
         nps

Out[56]: 23.73607540702657
```

```
In [57]: # Rounding off upto 2 decimal places -

         np.round(nps, 2)

Out[57]: 23.74
```

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 -

```
In [1]: import numpy as np
        a = np.array(range(16))
        a
```

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

What will be it's shape and dimensions?

```
In [2]: a.shape
```

```
Out[2]: (16,)
```

```
In [3]: a.ndim
```

```
Out[3]: 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**

```
In [4]: a.reshape(8, 2)
```

```
Out[4]: 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.

```
In [5]: a.reshape(4, 4)
```

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

```
In [6]: a.reshape(4, 5)
```

```
-----------------------------------------------------------------
----------
ValueError                                Traceback (most recent
call last)
Cell In[6], line 1
----> 1 a.reshape(4, 5)

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

This will give an Error. Why?

- We have 16 elements in `a`, but `reshape(4, 5)` is trying to fill in `4x5 = 20` elements.
- Therefore, whatever the shape we're trying to reshape to, must be able to incorporate the number of elements that we have.

```
In [7]: a.reshape(8, -1)
```

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

- You need to give at least one dimension.

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

```
In [8]: a = a.reshape(8, 2)
```

What will be the length of `a` ?

- It will be 8, since it contains 8 lists as it's 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

```
In [9]: len(a)

Out[9]: 8
```

```
In [10]: len(a[0])

Out[10]: 2
```

Transpose

Let's create a 2D numpy array.

```
In [11]: a = np.arange(12).reshape(3,4)
         a

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

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.

```
In [12]: a.T

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

Let's verify the shape of this transpose array -

In [13]:
```
a.T.shape
```

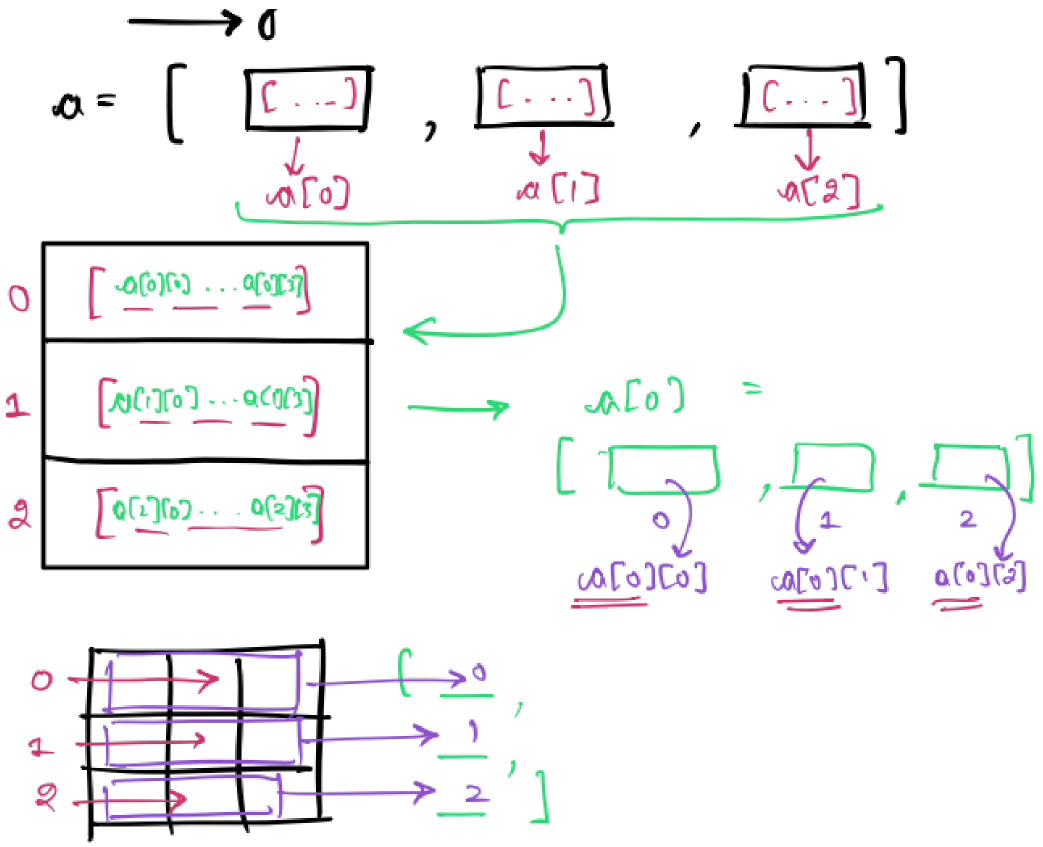
Out[13]:
```
(4, 3)
```

Indexing in 2D arrays

- Similar to Python lists

In [14]:
```
from IPython.display import Image
Image(filename='2dnp.png')
```

Out[14]:


0
a = [[. . .] , [. . .] , [. . .]]
a[0]
a[1]
a[2]
0
[a[0][0] . . . a[0][3]]
1
[a[1][0] . . . a[1][3]]
2
[a[2][0] . . . a[2][3]]
a[0] =
[, ,]
0
1
2
a[0][0]
a[0][1]
a[0][2]
0
1
2
[0 , 1 , 2]

In [15]:
```
a
```

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

Can we extract just the element 6 from a ?

In [16]:
```
# Accessing 2nd row and 3rd col -
a[1, 2]
```

Out[16]:
```
6
```

This can also be written as

```
In [17]: a[1][2]
```

```
Out[17]: 6
```

```
In [18]: m1 = np.arange(1,10).reshape((3,3))
         m1
```

```
Out[18]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

What will be the output of this?

```
In [19]: m1[1, 1] # m1[row,column]
```

```
Out[19]: 5
```

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

```
In [20]: m1 = np.array([100,200,300,400,500,600])
```

Will this work now?

```
In [21]: m1[2, 3]
```

```
-----------------------------------------------------------------
----------
IndexError                          Traceback (most recent
call last)
Cell In[21], line 1
----> 1 m1[2, 3]

IndexError: too many indices for array: array is 1-dimensional, b
ut 2 were indexed
```

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.

```
In [22]: m1[[2, 3]]
```

```
Out[22]: array([300, 400])
```

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

```
In [23]: m1 = np.arange(9).reshape((3,3))
         m1
```

```
Out[23]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

```
In [24]: m1[[0,1,2],[0,1,2]] # picking up element (0,0), (1,1) and (2,2)
```

```
Out[24]: array([0, 4, 8])
```

Index Arrays: When you do m1[[0, 1, 2], [0, 1, 2]], you are providing two lists or arrays for indexing:

The first list [0, 1, 2] specifies the row indices. The second list [0, 1, 2] specifies the column indices.

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 m2[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**

```
In [25]: m1 = np.arange(12).reshape(3,4)
         m1
```

```
Out[25]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

```
In [26]: m1[:2] # gives first two rows
```

```
Out[26]: array([[0, 1, 2, 3],
                [4, 5, 6, 7]])
```

How can we get columns from a 2D array?

```
In [27]: m1[:, :2] # gives first two columns
```

```
Out[27]: array([[0, 1],
                [4, 5],
                [8, 9]])
```

```
In [28]: m1[:, 1:3] # gives 2nd and 3rd col
```

```
Out[28]: 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` -

```
In [29]: m1 = np.arange(12).reshape(3, 4)
         m1
```

```
Out[29]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

What will be output of following?

```
In [30]: m1 < 6
```

```
Out[30]: 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.

```
In [31]: m1[m1 < 6]
```

```
Out[31]: array([0, 1, 2, 3, 4, 5])
```

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

Aggregate Functions

How would calculate the sum of elements of an array?

`np.sum()`

- It sums all the values in a numpy array.

```
In [32]: a = np.arange(1, 11)
         a
```

```
Out[32]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [33]: np.sum(a)
```

```
Out[33]: 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.

```
In [34]: np.mean(a)
```

```
Out[34]: 5.5
```

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

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

```
In [35]: np.min(a)
```

```
Out[35]: 1
```

```
In [36]: np.max(a)
```

```
Out[36]: 10
```

Let's apply aggregate functions on 2D array.

In [37]:

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

Out[37]:

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

In [38]:

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

Out[38]:

```
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` .

In [39]:

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

Out[39]:

```
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` .

In [40]:

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

Out[40]:

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

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` :

In [41]:

```
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.

In [42]:

```


# 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 pl
else:
    print("Finish all your chores before you can play.")
```

```
Finish all your chores before you can play.
```

Multiple conditions for `.all()` function -

In [43]:

```
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[43]:

```
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 [44]:

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

Out[44]:

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

You can use masking to update the array.

In [45]:

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

In [46]:

```
arr
```

Out[46]:

```
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.

In [47]:

```
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 desiging the watch?

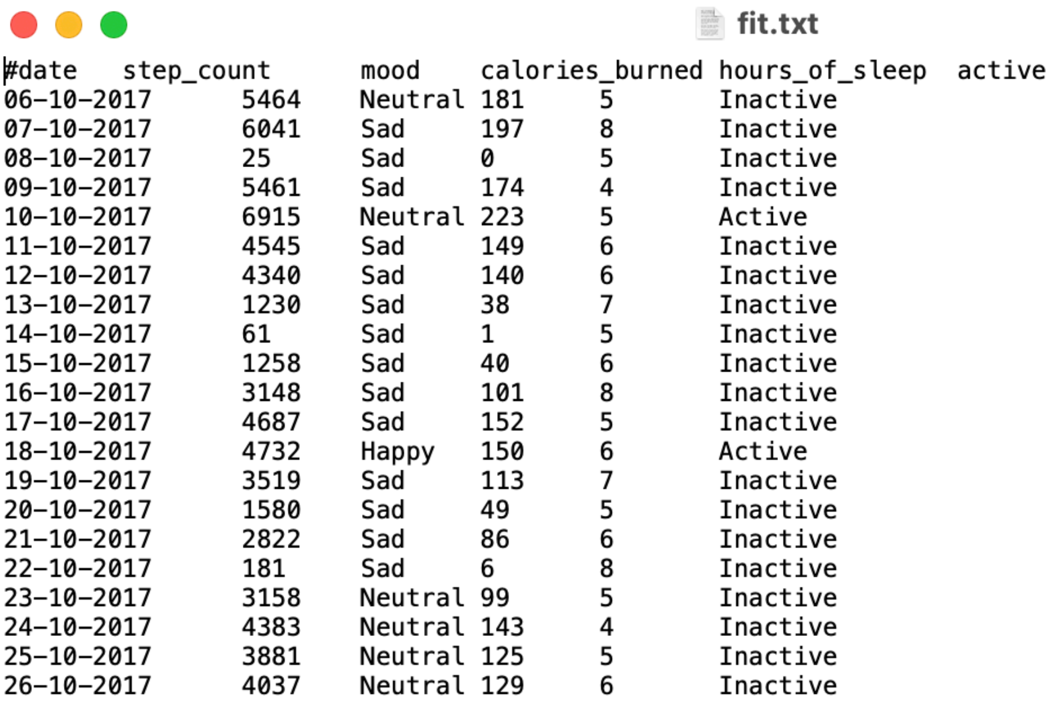
These insights from the user data can help business make customer oriented decision for the product design.

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

In [48]:

```
from IPython.display import Image
Image(filename='fir.png')
```

Out[48]:



```
#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.

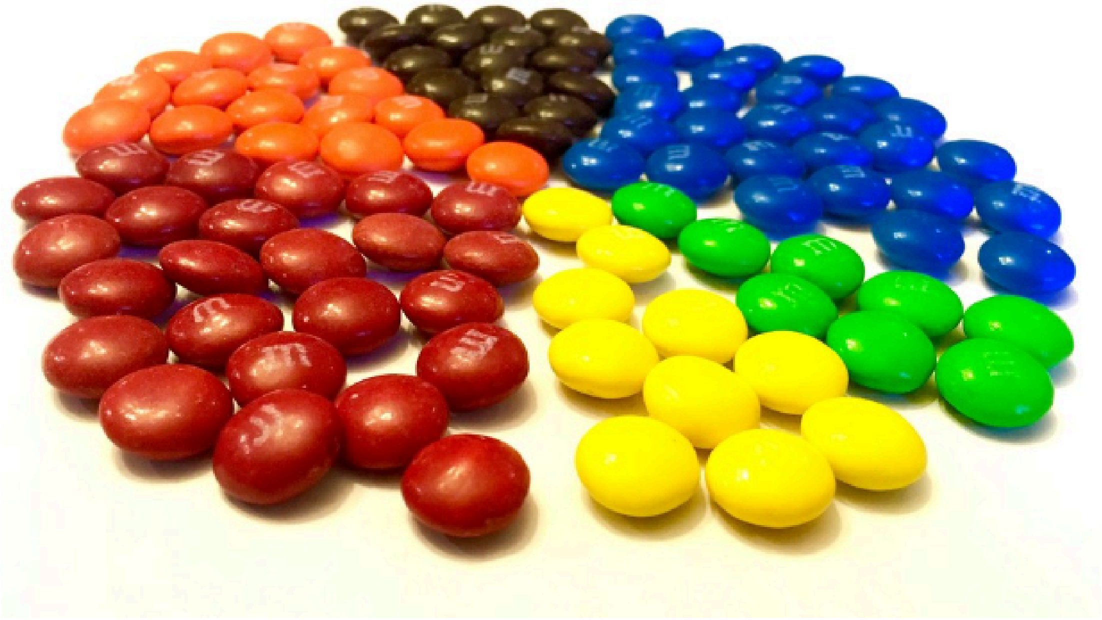
Imagine you have a big box of colorful candies. EDA is like looking at all the candies, counting how many of each color there are, and maybe even making a pretty picture to show which colors you have the most of. This way, you can learn a lot about your candies without eating them all at once!

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.

In [50]:
```
from IPython.display import Image
Image(filename='eda.png')
```

Out[50]:



First, we will import numpy.

In [51]:
```
import numpy as np
```

Let's load the data that we saw earlier.

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

In [52]:

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

```
Out[52]: 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'],
               ['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'],
               ['27-10-2017', '202', 'Neutral', '6', '8', 'Inactive'],
               ['28-10-2017', '292', 'Neutral', '9', '5', 'Inactive'],
               ['29-10-2017', '330', 'Happy', '10', '6', 'Inactive'],
               ['30-10-2017', '2209', 'Neutral', '72', '5', 'Inactive'],
               ['31-10-2017', '4550', 'Happy', '150', '8', 'Active'],
               ['01-11-2017', '4435', 'Happy', '141', '5', 'Inactive'],
               ['02-11-2017', '4779', 'Happy', '156', '4', 'Inactive'],
               ['03-11-2017', '1831', 'Happy', '57', '5', 'Inactive'],
               ['04-11-2017', '2255', 'Happy', '72', '4', 'Inactive'],
               ['05-11-2017', '539', 'Happy', '17', '5', 'Active'],
               ['06-11-2017', '5464', 'Happy', '181', '4', 'Inactive'],
               ['07-11-2017', '6041', 'Neutral', '197', '3', 'Inactive'],
               ['08-11-2017', '4068', 'Happy', '131', '2', 'Inactive'],
               ['09-11-2017', '4683', 'Happy', '154', '9', 'Inactive'],
               ['10-11-2017', '4033', 'Happy', '137', '5', 'Inactive'],
               ['11-11-2017', '6314', 'Happy', '193', '6', 'Active'],
               ['12-11-2017', '614', 'Happy', '19', '4', 'Active'],
               ['13-11-2017', '3149', 'Happy', '101', '5', 'Active'],
               ['14-11-2017', '4005', 'Happy', '139', '8', 'Active'],
               ['15-11-2017', '4880', 'Happy', '164', '4', 'Active'],
               ['16-11-2017', '4136', 'Happy', '137', '5', 'Active'],
               ['17-11-2017', '705', 'Happy', '22', '6', 'Active'],
               ['18-11-2017', '570', 'Neutral', '17', '5', 'Active'],
               ['19-11-2017', '269', 'Happy', '9', '6', 'Active'],
               ['20-11-2017', '4275', 'Happy', '145', '5', 'Inactive'],
               ['21-11-2017', '5999', 'Happy', '192', '6', 'Inactive'],
               ['22-11-2017', '4421', 'Happy', '146', '5', 'Inactive'],
               ['23-11-2017', '6930', 'Happy', '234', '6', 'Inactive'],
               ['24-11-2017', '5195', 'Happy', '167', '5', 'Inactive'],
               ['25-11-2017', '546', 'Happy', '16', '6', 'Inactive'],
               ['26-11-2017', '493', 'Happy', '17', '7', 'Active'],
               ['27-11-2017', '995', 'Happy', '32', '6', 'Active'],
               ['28-11-2017', '1163', 'Neutral', '35', '7', 'Active'],
               ['29-11-2017', '6676', 'Sad', '220', '6', 'Active'],
               ['30-11-2017', '3608', 'Happy', '116', '5', 'Active'],
               ['01-12-2017', '774', 'Happy', '23', '6', 'Active'],
               ['02-12-2017', '1421', 'Happy', '44', '7', 'Active'],
               ['03-12-2017', '4064', 'Happy', '131', '8', 'Active'],
               ['04-12-2017', '2725', 'Happy', '86', '8', 'Active'],
               ['05-12-2017', '5934', 'Happy', '194', '7', 'Active'],
               ['06-12-2017', '1867', 'Happy', '60', '8', 'Active'],
               ['07-12-2017', '3721', 'Sad', '121', '5', 'Active'],
               ['08-12-2017', '2374', 'Neutral', '76', '4', 'Inactive'],
               ['09-12-2017', '2909', 'Neutral', '93', '3', 'Active'],
               ['10-12-2017', '1648', 'Sad', '53', '3', 'Active'],
               ['11-12-2017', '799', 'Sad', '25', '4', 'Inactive'],
               ['12-12-2017', '7102', 'Neutral', '227', '5', 'Active'],
               ['13-12-2017', '3941', 'Neutral', '125', '5', 'Active'],
               ['14-12-2017', '7422', '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']], dtyp
        e='<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?

In [53]:
```
data.shape
```

Out[53]:
```
(96, 6)
```

What's the dimensionality?

In [54]:
```
data.ndim
```

Out[54]:
```
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 seperately.

How will we extract features in seperate variables?

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

Let's see whats the first element of the `data`.

```
In [55]: data[0]
```

```
Out[55]: 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.

```
In [56]: data[1]
```

```
Out[56]: array(['07-10-2017', '6041', 'Sad', '197', '8', 'Inactive'], dtyp
         e='<U10')
```

We can also use slicing.

```
In [57]: data[:5]
```

```
Out[57]: 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

```
In [58]: data[:, 0]
```

```
Out[58]: 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

```
In [59]: 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` ?

```
In [60]: data_t[0]
```

```
Out[60]: 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` ?

```
In [61]: data_t.shape
```

```
Out[61]: (6, 96)
```

Let's extract all the columns and save them in seperate variables.

```
In [62]: date, step_count, mood, calories_burned, hours_of_sleep, activity_
In [63]: step_count
```

```
Out[63]: array(['5464', '6041', '25', '5461', '6915', '4545', '4340', '123
         0', '61',
                '1258', '3148', '4687', '4732', '3519', '1580', '2822', '1
         81',
                '3158', '4383', '3881', '4037', '202', '292', '330', '220
         9',
                '4550', '4435', '4779', '1831', '2255', '539', '5464', '60
         41',
                '4068', '4683', '4033', '6314', '614', '3149', '4005', '48
         80',
                '4136', '705', '570', '269', '4275', '5999', '4421', '693
         0',
                '5195', '546', '493', '995', '1163', '6676', '3608', '77
         4', '1421',
                '4064', '2725', '5934', '1867', '3721', '2374', '2909', '1
         648',
                '799', '7102', '3941', '7422', '437', '1231', '1696', '492
         1',
                '221', '6500', '3575', '4061', '651', '753', '518', '553
         7', '4108',
                '5376', '3066', '177', '36', '299', '1447', '2599', '702',
         '133',
                '153', '500', '2127', '2203'], dtype='<U10')
```

```
In [64]: step_count.dtype
```

```
Out[64]: 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 [65]: step_count = np.array(step_count, dtype='int')
         step_count.dtype
```

```
Out[65]: dtype('int64')
```

In [66]:
```
step_count
```

Out[66]:
```
array([5464, 6041,   25, 5461, 6915, 4545, 4340, 1230,   61, 125
8, 3148,
       4687, 4732, 3519, 1580, 2822,  181, 3158, 4383, 3881, 403
7,  202,
        292,  330, 2209, 4550, 4435, 4779, 1831, 2255,  539, 546
4, 6041,
       4068, 4683, 4033, 6314,  614, 3149, 4005, 4880, 4136,  70
5,  570,
        269, 4275, 5999, 4421, 6930, 5195,  546,  493,  995, 116
3, 6676,
       3608,  774, 1421, 4064, 2725, 5934, 1867, 3721, 2374, 290
9, 1648,
        799, 7102, 3941, 7422,  437, 1231, 1696, 4921,  221, 650
0, 3575,
       4061,  651,  753,  518, 5537, 4108, 5376, 3066,  177,   3
6,  299,
       1447, 2599,  702,  133,  153,  500, 2127, 2203])
```

What will be shape of this array?

In [67]:
```
step_count.shape
```

Out[67]:
```
(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

In [68]:
```
calories_burned = np.array(calories_burned, dtype='int')
calories_burned.dtype
```

Out[68]:
```
dtype('int64')
```

Hours of Sleep

In [69]:
```
hours_of_sleep = np.array(hours_of_sleep, dtype='int')
hours_of_sleep.dtype
```

Out[69]:
```
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 -

```
In [70]: mood
```

```
Out[70]: array(['Neutral', 'Sad', 'Sad', 'Sad', 'Neutral', 'Sad', 'Sad',
         'Sad',
                'Sad', 'Sad', 'Sad', 'Sad', 'Happy', 'Sad', 'Sad', 'Sad',
         'Sad',
                'Neutral', 'Neutral', 'Neutral', 'Neutral', 'Neutral', 'Ne
         utral',
                'Happy', 'Neutral', 'Happy', 'Happy', 'Happy', 'Happy', 'H
         appy',
                'Happy', 'Happy', 'Neutral', 'Happy', 'Happy', 'Happy', 'H
         appy',
                'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Neu
         tral',
                'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Happy', 'Hap
         py',
                'Happy', 'Happy', 'Neutral', 'Sad', 'Happy', 'Happy', 'Hap
         py',
                'Happy', 'Happy', 'Happy', 'Happy', 'Sad', 'Neutral', 'Neu
         tral',
                'Sad', 'Sad', 'Neutral', 'Neutral', 'Happy', 'Neutral', 'N
         eutral',
                'Sad', 'Neutral', 'Sad', 'Neutral', 'Neutral', 'Sad', 'Sa
         d', 'Sad',
                'Sad', 'Happy', 'Neutral', 'Happy', 'Neutral', 'Sad', 'Sa
         d', 'Sad',
                'Neutral', 'Neutral', 'Sad', 'Sad', 'Happy', 'Neutral', 'N
         eutral',
                'Happy'], dtype='<U10')
```

```
In [71]: np.unique(mood)
```

```
Out[71]: array(['Happy', 'Neutral', 'Sad'], dtype='<U10')
```

Activity Status

In [72]:
```
activity_status
```

Out[72]:
```
array(['Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactiv
e',
       'Inactive', 'Inactive', 'Active', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactiv
e',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactiv
e',
       'Active', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
'Active',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Inactiv
e',
       'Active', 'Active', 'Active', 'Active', 'Active', 'Activ
e',
       'Active', 'Active', 'Active', 'Inactive', 'Inactive', 'Ina
ctive',
       'Inactive', 'Inactive', 'Inactive', 'Active', 'Active', 'A
ctive',
       'Active', 'Active', 'Active', 'Active', 'Active', 'Activ
e',
       'Active', 'Active', 'Active', 'Inactive', 'Active', 'Activ
e',
       'Inactive', 'Active', 'Active', 'Active', 'Active', 'Activ
e',
       'Inactive', 'Active', 'Active', 'Active', 'Active', 'Inact
ive',
       'Inactive', 'Inactive', 'Inactive', 'Active', 'Active', 'A
ctive',
       'Active', 'Inactive', 'Inactive', 'Inactive', 'Inactive',
       'Inactive', 'Inactive', 'Inactive', 'Inactive', 'Active',
       'Inactive', 'Active'], dtype='<U10')
```

Since we've extracted form 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?

In [73]:
```
step_count_happy = step_count[mood == 'Happy']
step_count_happy
```

Out[73]:
```
array([4732,  330, 4550, 4435, 4779, 1831, 2255,  539, 5464, 406
8, 4683,
       4033, 6314,  614, 3149, 4005, 4880, 4136,  705,  269, 427
5, 5999,
       4421, 6930, 5195,  546,  493,  995, 3608,  774, 1421, 406
4, 2725,
       5934, 1867, 7422, 5537, 5376,  153, 2203])
```

```
In [74]: len(step_count_happy)

Out[74]: 40
```

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

```
In [75]: step_count_sad = step_count[mood == 'Sad']
         step_count_sad

Out[75]: array([6041,   25, 5461, 4545, 4340, 1230,   61, 1258, 3148, 468
         7, 3519,
                1580, 2822,  181, 6676, 3721, 1648,  799, 1696,  221, 406
         1,  651,
                 753,  518,  177,   36,  299,  702,  133])
```

```
In [76]: len(step_count_sad)

Out[76]: 29
```

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

```
In [77]: step_count_neutral = step_count[mood == 'Neutral']
         step_count_neutral

Out[77]: array([5464, 6915, 3158, 4383, 3881, 4037,  202,  292, 2209, 604
         1,  570,
                1163, 2374, 2909, 7102, 3941,  437, 1231, 4921, 6500, 357
         5, 4108,
                3066, 1447, 2599,  500, 2127])
```

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

```
In [78]: step_count_happy_or_neutral = step_count[(mood == 'Neutral') | (mo
         step_count_happy_or_neutral

Out[78]: array([5464, 6915, 4732, 3158, 4383, 3881, 4037,  202,  292,  33
         0, 2209,
                4550, 4435, 4779, 1831, 2255,  539, 5464, 6041, 4068, 468
         3, 4033,
                6314,  614, 3149, 4005, 4880, 4136,  705,  570,  269, 427
         5, 5999,
                4421, 6930, 5195,  546,  493,  995, 1163, 3608,  774, 142
         1, 4064,
                2725, 5934, 1867, 2374, 2909, 7102, 3941, 7422,  437, 123
         1, 4921,
                6500, 3575, 5537, 4108, 5376, 3066, 1447, 2599,  153,  50
         0, 2127,
                2203])
```

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

```
In [79]: # Average step count on Sad mood days -

         np.mean(step_count_sad)
```

```
Out[79]: 2103.0689655172414
```

```
In [80]: # Average step count on Happy days -

         np.mean(step_count_happy)
```

```
Out[80]: 3392.725
```

```
In [81]: # Average step count on Neutral days -

         np.mean(step_count_neutral)
```

```
Out[81]: 3153.777777777778
```

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.

```
In [82]: # mood when step count > 4000

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

```
Out[82]: (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.

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

Numpy 3

Content

- Sorting
- Matrix Multiplication
  - `np.dot`
  - `@` operator
  - `np.matmul`
- Vectorization
- Broadcasting

Sorting

- `np.sort` returns a sorted copy of an array.

```
In [1]: import numpy as np
```

```
In [2]: a = np.array([4, 7, 0, 3, 8, 2, 5, 1, 6, 9])
        a
```

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

```
In [3]: b = np.sort(a)
        b
```

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

```
In [4]: a # no change is reflected in the original array
```

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

We can directly call `sort` method on array but it can change the original array as it is an inplace operation.

```
In [5]: a.sort() # sorting is performed inplace
        a
```

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

Sorting in 2D array

```
In [6]: a = np.array([[1,5,3], [2,5,7], [400, 200, 300]])
        a
```

```
Out[6]: array([[  1,   5,   3],
               [  2,   5,   7],
               [400, 200, 300]])
```

```
In [7]: np.sort(a, axis=0) # sorting every column
```

```
Out[7]: array([[  1,   5,   3],
               [  2,   5,   7],
               [400, 200, 300]])
```

```
In [8]: np.sort(a, axis=1) # sorting every row
```

```
Out[8]: array([[  1,   3,   5],
               [  2,   5,   7],
               [200, 300, 400]])
```

Note: By default, the `np.sort()` functions sorts along the last axis.

```
In [9]: a = np.array([[23,4,43], [12, 89, 3], [69, 420, 0]])
```

```
In [10]: np.sort(a) # default axis = -1 (last axis)
```

```
Out[10]: array([[  4,  23,  43],
                [  3,  12,  89],
                [  0,  69, 420]])
```

Element-Wise Multiplication

Element-wise multiplication in NumPy involves multiplying corresponding elements of two arrays with the same shape to produce a new array where each element is the product of the corresponding elements from the input arrays.

```
In [11]: a = np.arange(1, 6)
         a
```

```
Out[11]: array([1, 2, 3, 4, 5])
```

```
In [12]: a * 5
```

```
Out[12]: array([ 5, 10, 15, 20, 25])
```

In [13]:

```
b = np.arange(6, 11)
b
```

Out[13]:

```
array([ 6,  7,  8,  9, 10])
```

In [14]:

```
a * b
```

Out[14]:

```
array([ 6, 14, 24, 36, 50])
```

Both arrays should have the same shape.

In [15]:

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

In [16]:

```
a * c
```

```
-------------------------------------------------------------------
----------
ValueError                                Traceback (most recent
call last)
Cell In[16], line 1
----> 1 a * c

ValueError: operands could not be broadcast together with shapes
(5,) (3,)
```

In [17]:

```
d = np.arange(12).reshape(3, 4)
e = np.arange(13, 25).reshape(3, 4)
```

In [18]:

```
print("d=", d)
print("e=", e)
```

```
d= [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
e= [[13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]]
```

In [19]:

```
d * e
```

Out[19]:

```
array([[  0,  14,  30,  48],
       [ 68,  90, 114, 140],
       [168, 198, 230, 264]])
```

Takeaway:

- Array * Number -> WORKS
- Array * Array (same shape) -> WORKS
- Array * Array (different shape) -> DOES NOT WORK

Matrix Multiplication

Rule: Number of columns of the first matrix should be equal to number of rows of the second matrix.

- (A,B) * (B,C) -> (A,C)
- (3,4) * (4,3) -> (3,3)

Visual Demo: https://www.geogebra.org/m/ETHXK756 (https://www.geogebra.org/m/ETHXK756)

In [20]:
```
a = np.arange(1,13).reshape((3,4))
c = np.arange(2,14).reshape((4,3))
```

In [21]:
```
a.shape, c.shape
```

Out[21]:
```
((3, 4), (4, 3))
```

a is of shape (3,4) and c is of shape (4,3). The output will be of shape (3,3).

In [22]:
```
# Using np.dot
np.dot(a,c)
```

Out[22]:
```
array([[ 80,  90, 100],
       [184, 210, 236],
       [288, 330, 372]])
```

In [23]:
```
# Using np.matmul
np.matmul(a,c)
```

Out[23]:
```
array([[ 80,  90, 100],
       [184, 210, 236],
       [288, 330, 372]])
```

In [24]:
```
# Using @ operator
a@c
```

Out[24]:
```
array([[ 80,  90, 100],
       [184, 210, 236],
       [288, 330, 372]])
```

In [25]:

```
a@5
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[25], line 1
----> 1 a@5

ValueError: matmul: Input operand 1 does not have enough dimensions (has 0, gufunc core with signature (n?,k),(k,m?)->(n?,m?) requires 1)
```

In [26]:

```
np.matmul(a, 5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[26], line 1
----> 1 np.matmul(a, 5)

ValueError: matmul: Input operand 1 does not have enough dimensions (has 0, gufunc core with signature (n?,k),(k,m?)->(n?,m?) requires 1)
```

In [27]:

```
np.dot(a, 5)
```

Out[27]:

```
array([[ 5, 10, 15, 20],
       [25, 30, 35, 40],
       [45, 50, 55, 60]])
```

Important:

- `dot()` function supports the vector multiplication with a scalar value, which is not possible with `matmul()` .
- `Vector * Vector` will work for `matmul()` but `Vector * Scalar` won't.

Vectorization

Vectorization in NumPy refers to performing operations on entire arrays or array elements simultaneously, which is significantly faster and more efficient than using explicit loops.

In [28]:

```
a = np.arange(10)
a
```

Out[28]:

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

Note:

- 1d np array --> vector
- 2d np array --> matrix
- 3d onwards --> tensors

In [29]:
```
def random_operation(x):
    if x % 2 == 0:
        x += 2
    else:
        x -= 2

    return x
```

In [30]:
```
random_operation(a)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent
call last)
Cell In[30], line 1
----> 1 random_operation(a)

Cell In[29], line 2, in random_operation(x)
      1 def random_operation(x):
----> 2     if x % 2 == 0:
      3         x += 2
      4     else:

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

In [31]:
```
cool_operation = np.vectorize(random_operation)
```

In [32]:
```
type(cool_operation)
```

Out[32]:
```
numpy.vectorize
```

np.vectorize()

- It is a generalised function for vectorization.
- It takes the function and returns an object (which acts like function but can take an array as input and perform the operations).

In [33]:
```
cool_operation(a)
```

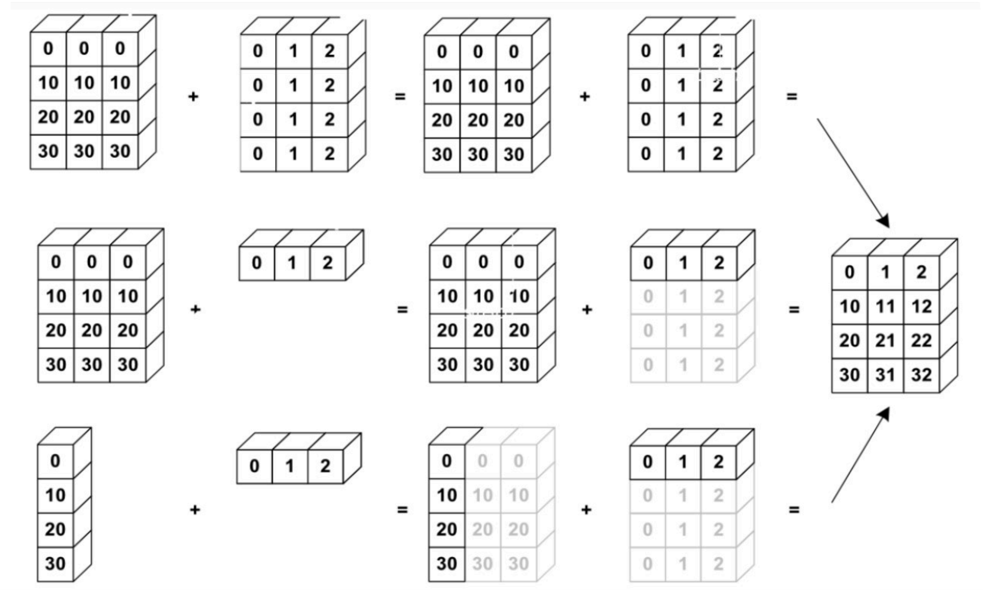
Out[33]:
```
array([ 2, -1,  4,  1,  6,  3,  8,  5, 10,  7])
```

Broadcasting

Broadcasting in NumPy is the automatic and implicit extension of array dimensions to enable element-wise operations between arrays with different shapes.

```
In [34]: from IPython.display import Image
         Image(filename='broadcasting.jpeg')
```

Out[34]:



Case 1: If dimension in both matrix is equal, element-wise addition will be done.

```
In [35]: a = np.tile(np.arange(0,40,10), (3,1))
         a
```

```
Out[35]: array([[ 0, 10, 20, 30],
                [ 0, 10, 20, 30],
                [ 0, 10, 20, 30]])
```

Note:

- `numpy.tile(array, reps)` constructs an array by repeating A the number of times given by reps along each dimension.
- `np.tile(array, (repetition_rows, repetition_cols))`

```
In [36]: a=a.T
         a
```

```
Out[36]: array([[ 0,  0,  0],
                [10, 10, 10],
                [20, 20, 20],
                [30, 30, 30]])
```

```
In [37]: b = np.tile(np.arange(0,3), (4,1))
         b
```

```
Out[37]: array([[0, 1, 2],
                [0, 1, 2],
                [0, 1, 2],
                [0, 1, 2]])
```

```
In [38]: print(a.shape, b.shape)
```

```
(4, 3) (4, 3)
```

Since a and b have the same shape, they can be added without any issues.

```
In [39]: a+b
```

```
Out[39]: array([[ 0,  1,  2],
                [10, 11, 12],
                [20, 21, 22],
                [30, 31, 32]])
```

Case 2: Right array should be of 1-D and number of columns should be same of both the arrays and it will automatically do n-tile.

```
In [40]: a
```

```
Out[40]: array([[ 0,  0,  0],
                [10, 10, 10],
                [20, 20, 20],
                [30, 30, 30]])
```

```
In [41]: c = np.array([0,1,2])
         c
```

```
Out[41]: array([0, 1, 2])
```

```
In [42]: print(a.shape, c.shape)
```

```
(4, 3) (3,)
```

```
In [43]: a + c
```

```
Out[43]: array([[ 0,  1,  2],
                [10, 11, 12],
                [20, 21, 22],
                [30, 31, 32]])
```

- c was broadcasted along rows (vertically)
- so that a and c can be made compatible

Case 3: If the left array is column matrix (must have only 1 column) and right array is row matrix, then it will do the n-tile such that element wise addition is possible.

```
In [44]: d = np.array([0,10,20,30]).reshape(4,1)
         d
```

```
Out[44]: array([[ 0],
                [10],
                [20],
                [30]])
```

```
In [45]: c = np.array([0,1,2])
         c
```

```
Out[45]: array([0, 1, 2])
```

```
In [46]: print(d.shape, c.shape)
```

```
         (4, 1) (3,)
```

```
In [47]: d + c
```

```
Out[47]: array([[ 0,  1,  2],
                [10, 11, 12],
                [20, 21, 22],
                [30, 31, 32]])
```

- d was stacked (broadcasted) along columns (horizontally)
- c was stacked (broadcasted) along rows (vertically)

Will broadcasting work in this case?

```
In [48]: a = np.arange(8).reshape(2,4)
         a
```

```
Out[48]: array([[0, 1, 2, 3],
                [4, 5, 6, 7]])
```

```
In [49]: b = np.arange(16).reshape(4,4)
         b
```

```
Out[49]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

```
In [50]: a+b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent
call last)
Cell In[50], line 1
----> 1 a+b

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

Broadcasting in 2D Arrays

- A + A (same shape)-> Works
- A + A (1D) -> Works
- A + number -> Works
- A + A (different shape but still 2D) -> DOES NOT WORK

Is broadcasting possible in this case?

```
In [51]: A = np.arange(1,10).reshape(3,3)
         A
```

```
Out[51]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [52]: B = np.array([-1, 0, 1])
         B
```

```
Out[52]: array([-1,  0,  1])
```

```
In [53]: A*B
```

```
Out[53]: array([[-1,  0,  3],
                [-4,  0,  6],
                [-7,  0,  9]])
```

Yes! Broadcasting is possible for all the operations.

```
In [54]: A = np.arange(12).reshape(3, 4)
         A
```

```
Out[54]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

```
In [55]: B = np.array([1, 2, 3])
         B
```

```
Out[55]: array([1, 2, 3])
```

```
In [56]: A + B
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[56], line 1
----> 1 A + B

ValueError: operands could not be broadcast together with shapes (3,4) (3,) 
```

Why did it throw an error?

Are the number of dimensions same for both array? No.

- Shape of A ⇒ (3,4)
- Shape of B ⇒ (3,)

So, `Rule 1` will be invoked to pad 1 to the shape of B.

So, the shape of B becomes **(1,3)**.

Now, we check whether broadcasting conditions are met or not?

Starting from the right most side,

- Right most dimension is not equal (4 and 3).

Hence, broadcasting is not possible as per `Rule 3`.

Question: Given two arrays,

1. Array A of shape (8, 1, 6, 1)
2. Array B of shape (7, 1, 5)

Is broadcasting possible in this case? If yes, what will be the shape of output?

Answer: Broadcasting possible; Shape will be (8, 7, 6, 5)

Explanation:

As number of dimensions are not equal, `Rule 1` is invoked.

The shape of B becomes (1, 7, 1, 5)

Next, it checks whether broadcasting is possible.

A ⇒ (8 , 1, 6, 1)
B ⇒ (1, 7, 1, 5)

- Right most dimension, one of the dimension is 1 (1 vs 5)
- Next, comparing 6 and 1, We have one dimension as 1
- Similarly, we have one of the dimension as 1 in both leading dimensions.

Hence, broadcasting is possible.

Now, as per `Rule 2`, dimension with value 1 is streched to match dimension of other array.

- Right most dimension of array is streched to match 5
- Leading dimension of array B (1) is streched to match array A dim (6)

So, the output shape becomes : `(8, 7, 6, 5)`.