Python LIST vs NumPy ARRAY

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

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

```
type(a)
```

```
list
```

Importing NumPy

```
import numpy as np
```

Comparison

Simplicity

```
a = [1, 2, 3, 4, 5]
res = [i**2 for i in a]
print(res)
```

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

```
import numpy as np
b = np.array(a)
print(b**2)
```

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

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

Notice how easy and clean is the syntax.

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

- To understand this, let's measure the time for these operations.
- We will use `%timeit`.

Speed

Let’s square 1 million numbers using both methods:

```
l = range(1000000)
%timeit [i**2 for i in l]  # ~300 ms

l_np = np.array(range(1000000))
%timeit l_np**2            # ~900 µs
```

```
105 ms ± 33.7 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
935 µs ± 141 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
type(l_np)
```

Dimensions and Shape

Dimension (ndim)

- Tells how many directions (axes) the data has
- Also called number of dimensions

Shape

- Tells how many elements are in each direction

```
a = np.array([10, 20, 30])

# a.ndim
a.shape
```

```
(3,)
```

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

# print(b.ndim, b.shape)
b
```

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

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

# print(c.ndim, c.shape)

c
```

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

       [[5, 6],
        [7, 8]]])
```

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

```
1 (8,)
```

Arange

`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)`

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

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

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

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

`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**.

```
arr3 = np.arange(1, 5, 0.5)
arr3
```

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

Type Conversion in Numpy Arrays

```
arr1 = np.array([1, 2, 3, True])
print(arr1)
```

```
[1 2 3 1]
```

```
arr2 = np.array(["Harry Potter", 1, 2, 3])
print(arr2)
```

```
['Harry Potter' '1' '2' '3']
```

```
arr3 = np.array([1, 2, 3, 4], dtype='float')
print(arr3)
```

```
[1. 2. 3. 4.]
```

```
np.array(["Shivank", "Bipin"], dtype=float)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-4244221729.py in <cell line: 0>()
----> 1 np.array(["Shivank", "Bipin"], dtype=float)

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

```
arr4 = np.array([10, 20, None, 30])
arr4 = arr4.astype('float64')
print(arr4)
```

```
[10. 20. nan 30.]
```

```
arr = np.array([1.5, 2+3j])
print(arr)         # [1.5+0.j 2. +3.j]
```

```
print(arr.dtype)   # complex128
```

bool ⟶ int ⟶ float ⟶ complex ⟶ string (object)

Indexing

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

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

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

```
np.int64(11)
```

```
m1[11] # positive indexing in numpy array
```

```
np.int64(11)
```

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

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

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

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

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

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

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

```
m1[[5,7,3,15]]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipython-input-3963483333.py in <cell line: 0>()
----> 1 m1[[5,7,3,15]]

IndexError: index 15 is out of bounds for axis 0 with size 10
```

```
m1[[True,  True,  True,  True,  True, False, False, True, False, False]]
```

```
m1[m1 < 6]
```

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

Slicing

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

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

```
m1[:5]
```

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

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

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

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

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

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

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

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[::2])  # Output: [10, 30, 50]  # every 2nd element
```

`array[start : stop : step]`

Indexing accesses a single element, while slicing accesses a range or subset of elements in arrays or lists.

```
import numpy as np
a = np.array([0,1,2,3,4,5])
a[4:] = 10
print(a)
```

Working with 2D arrays (Matrices)

```
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])
```

```
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]])
```

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2674185757.py in <cell line: 0>()
----> 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.

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

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

```
len(a)
```

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

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]])
```

```
# print(a.ndim, a.shape)
```

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]])
```

```
# print(a.T.ndim, a.T.shape)
```

Indexing in NumPy 2D Arrays

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

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

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

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

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

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

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

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

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

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

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

```
array([1, 5, 9])
```

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

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

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

How can we get columns from a 2D array?

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

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

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

```
import numpy as np
# Remove None
data = [25, 28, 28, 30, 33, 36, 37, 37, 100, 38, 39, 41, 44, 47]
```

```
Q1 = np.percentile(data, 25)  # Lower quartile
Q1
```

```
np.float64(30.75)
```

```
Q3 = np.percentile(data, 75)  # Upper quartile
Q3
```

```
np.float64(40.5)
```

```
IQR = Q3 - Q1
IQR
```

```
np.float64(9.75)
```

```
print("Q1:", Q1)     # 30.75
print("Q3:", Q3)     # 40.50
print("IQR:", IQR)   # 9.75
```

```
Q1: 30.75
Q3: 40.5
IQR: 9.75
```

```
import statistics

data = [18, 22, 22, 25, 27, 28, 29, 30, 31, 95]

mean = statistics.mean(data)            # 32.7
median = statistics.median(data)        # 27.5
mode = statistics.mode(data)            # 22
std_dev = statistics.stdev(data)        # ≈ 22.6
range_val = max(data) - min(data)       # 77

print("mean:", mean)
```

```
print("median:", median)
print("mode:", mode)
print("std_dev:", std_dev)
print("range:", range_val)
```

```
mean: 32.7
median: 27.5
mode: 22
std_dev: 22.271306901731453
range: 77
```