qwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmrtyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnmqwertyuiopasdfghjklzxcvbnm

**To instal**l:

**pip install numpy**

**Array**:

An array is basically nothing but pointers.

It’s a combination of a memory address, a data type, a shape and strides:

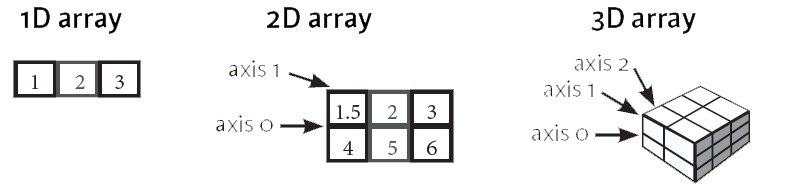
The data pointer indicates the memory address of the first byte in the array,

The data type or dtype pointer describes the kind of elements that are contained within the array,

The shape indicates the shape of the array, and

The strides are the number of bytes that should be skipped in memory to go to the next element.

If your strides are (10,1), you need to proceed one byte to get to the next column and 10 bytes to locate the next row.

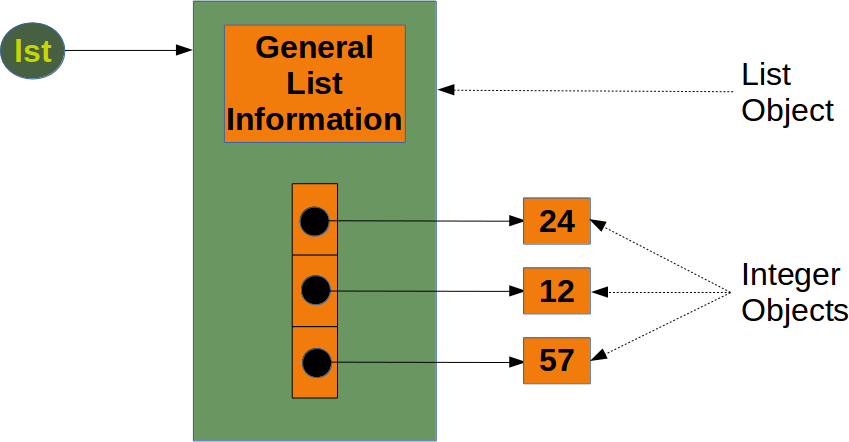


Numpy data structures perform better in:

* **Size** - Numpy data structures take up less space
* **Performance** - they have a need for speed and are faster than lists
* **Functionality** - SciPy and NumPy have optimized functions such as linear algebra operations built in.

**Memory**

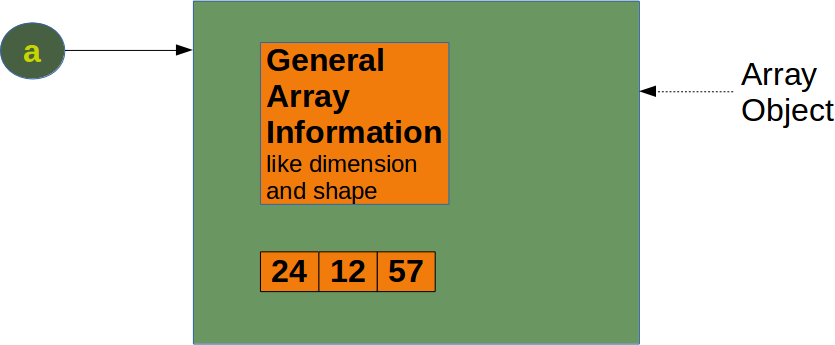
* The main benefits of using NumPy arrays should be smaller memory consumption and better runtime behavior.
* For Python Lists -  We can conclude from this that for every new element, we need another eight bytes for the reference to the new object. The new integer object itself consumes 28 bytes.
* The size of a list "lst" without the size of the elements can be calculated with:
* 64 + 8 \* len(lst) + + len(lst) \* 28



NumPy takes up less space. This means that an arbitrary integer array of length "n" in numpy needs

96 + n \* 8 Bytes

whereas a list of integer



So the more numbers you need to store - the better you do.

P: Pyton vs Numpy: Multiplying each element with 2

**Logic**:

Loops in Python are slow.

**Using python array:**

A1=[1,2,3,4]

A2=[]

for e in A1:

A2.append(e\*2)

print(A2)

**Using numpy array**:

import numpy as np;

A1=np.array([1,2,3,4]);

print(A1\*2);

**Output**:

[2, 4, 6, 8]

P:Basic 1d array

**Code**:

import numpy as np;

np=np.array([1,2,3,4,5]);

print(np);

**Output**:

[1 2 3 4 5]

P:More Than one dimensions

**Code**:

import numpy as np

a = np.array([[1, 2], [3, 5]])

print (a)

**Output**:



import numpy as np

a = np.array([[1, 2], [c, b]])

print (a)

**Output**:

**NameError**: name 'c' is not defined

P:nmin

**Logic**:

**Code**:

# minimum dimensions

import numpy as np

a = np.array([1, 2, 3,4,5], ndmin = 2)

print (a)

**Output**:

[[1 2 3 4 5]]

Data Types

P:Data Type

**Logic**:

**Code**:

# dtype parameter

import numpy as np

a = np.array([1, 2, 3], dtype = complex)

print a

**Output**:

[ 1.+0.j, 2.+0.j, 3.+0.j]

*Indexing & Slicing*

P:

**Logic**:

**Code**:

import numpy as np

a = np.arange(10)

s = slice(2,7,2)

print (a[s]);

**Output**:

[2,4,6]

P:

**Logic**:

**Code**:

import numpy as np

a = np.arange(10)

b = a[2:7:2]

print (b)

**Output**:

[2,4,6]

P:Slicing single item

**Logic**:

**Code**:

# slice single item

import numpy as np

a = np.arange(10)

b = a[5]

print(b)

**Output**:

5

P:Slice items starting from index

**Logic**:

**Code**:

# slice items starting from index

import numpy as np

a = np.arange(10)

print (a[2:])

**Output**:

[2 3 4 5 6 7 8 9]

P:Slice items between indexes

**Logic**:

**Code**:

import numpy as np

a = np.arange(10)

print (a[2:5])

**Output**:

[2,3,4]

P:

**Logic**:

**Code**:

import numpy as np

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

print (a)

# slice items starting from index

print ('Now we will slice the array from the index a[1:]')

print (a[1:])

**Output**:

[[1 2 3]

[3 4 5]

[4 5 6]]

Now we will slice the array from the index a[1:]

[[3 4 5]

[4 5 6]]

P:

**Logic**:

**Code**:

# array to begin with

import numpy as np

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

print ('Our array is:')

print (a)

print ('\n')

# this returns array of items in the second column

print ('The items in the second column are:')

print (a[...,1])

print ('\n')

# Now we will slice all items from the second row

print ('The items in the second row are:')

print (a[1,...])

print ('\n')

# Now we will slice all items from column 1 onwards

print ('The items column 1 onwards are:')

print (a[...,1:])

**Output**:

Our array is:

[[1 2 3]

[3 4 5]

[4 5 6]]

The items in the second column are:

[2 4 5]

The items in the second row are:

[3 4 5]

The items column 1 onwards are:

[[2 3]

[4 5]

[5 6]]

**Integer Indexing**:

This mechanism helps in selecting any arbitrary item in an array based on its Ndimensional index.

P:Integer Indexing

**Logic**:

**Code**:

import numpy as np

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

y = x[[0,1,2], [0,1,0]]

print (y)

**Output**:

[1 4 5]

P:

**Logic**:

**Code**:

import numpy as np

x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])

print ('Our array is:')

print (x)

print ('\n')

rows = np.array([[0,0],[3,3]])

cols = np.array([[0,2],[0,2]])

y = x[rows,cols]

print ('The corner elements of this array are:')

print (y)

**Output**:

Our array is:

[[ 0 1 2]

[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

The corner elements of this array are:

[[ 0 2]

[ 9 11]]

## Boolean Array Indexing:

This type of advanced indexing is used when the resultant object is meant to be the result of Boolean operations, such as comparison operators.

P:

**Logic**:

**Code**:

import numpy as np

x = np.array([[ 0, 1, 2],[ 3, 4, 5],[ 6, 7, 8],[ 9, 10, 11]])

print ('Our array is:')

print (x)

print ('\n')

# Now we will print the items greater than 5

print ('The items greater than 5 are:')

print (x[x > 5])

**Output**:

Our array is:

[[ 0 1 2]

[ 3 4 5]

[ 6 7 8]

[ 9 10 11]]

The items greater than 5 are:

[ 6 7 8 9 10 11]

*Broadcasting*

The term **broadcasting** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations.

Arithmetic operations on arrays are usually done on corresponding elements.

If two arrays are of exactly the same shape, then these operations are smoothly performed.

P: Smooth Addition

**Logic**:

**Code**:

import numpy as np

a = np.array([1,2,3,4])

b = np.array([10,20,30,40])

c = a \* b

print (c)

**Output**:

[ 10 40 90 160]

Broadcasting is possible if the following rules are satisfied −

* Array with smaller **ndim** than the other is prepended with '1' in its shape.
* Size in each dimension of the output shape is maximum of the input sizes in that dimension.
* An input can be used in calculation, if its size in a particular dimension matches the output size or its value is exactly 1.
* If an input has a dimension size of 1, the first data entry in that dimension is used for all calculations along that dimension.

A set of arrays is said to be **broadcastable** if the above rules produce a valid result and one of the following is true −

* Arrays have exactly the same shape.
* Arrays have the same number of dimensions and the length of each dimension is either a common length or 1.
* Array having too few dimensions can have its shape prepended with a dimension of length 1, so that the above stated property is true.

P:

**Logic**:

**Code**:

import numpy as np

a = np.array([[0.0,0.0,0.0],[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,30.0]])

b = np.array([1.0,2.0,3.0])

print ('First array:')

print (a)

print ('\n')

print ('Second array:')

print (b)

print ('\n')

print ('First Array + Second Array')

print (a + b)

**Output**:

First array:

[[ 0. 0. 0.]

[10. 10. 10.]

[20. 20. 20.]

[30. 30. 30.]]

Second array:

[1. 2. 3.]

First Array + Second Array

[[ 1. 2. 3.]

[11. 12. 13.]

[21. 22. 23.]

[31. 32. 33.]]

