NumPy Arrays

NumPy (Numerical Python) is a Python library designed for working with numerical data in Python. It is widely used in scientific computing, data analysis, and machine learning.

It provides:

- N-dimensional array object, or ndarray is a fixed-size and homogeneous (fixed-type) multidimensional array. It contains elements of a single data type, such as all integers, all floating-point numbers
- powerful mathematical functions for operating on those arrays of numbers.
- high-performance array calculations , because a ndarray is an homogeneous block of data

In this course we will focus on 1-D and 2-D arrays

NumPy module

Import NumPy in this way

```
import numpy as np   #generic import and rename the module
```

https://numpy.org/doc/stable/user/basics.html

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

```
import numpy as np
L1=[7,2,9,10]


v1=np.array(L1) #converts a list of numbers to a 1D array
print(v1)
[7 2 9 10]

print(type(v1))
<class 'numpy.ndarray'>

print(v1.dtype) #dtype returns the data type of the elements
int64
```

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

All items of a ndarray must be of the same data type or dtype

In this course we will focus on integer (int64), float (float64) , boolean (bool) types

```
L1=[7.2,2,9,10]
v1=np.array(L1)
print(v1)
[ 7.2  2.   9.  10. ]
print(v1.dtype)
float64
```

```
L1=[7.2,2,9,10,'pop']
v1=np.array(L1)
print(v1)
['7.2' '2' '9' '10' 'pop']
print(v1.dtype)
<U32  #unicode string
```

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

```
mutability

L1=[7.2,2,9,10]
v1=np.array(L1)

L1[0]=10   #[10, 2, 9, 10]
v1[0]=10   #[10  2  9 10]



L1[:3]=[0,0,0]  #[0, 0, 0, 10]
v1[:3]=0  #[ 0  0  0 10]  #the value is propagated to the entire selection. Broadcasting
```

NumPy arrays vs lists

type	items	mutability	size
list	Heterogeneous	mutable	change
numpy. ndarray	Homogeneous	mutable	fixed

```
size

id(L1) #140471496928832

L1.append(100) #[0, 0, 0, 10, 100]
id(L1) #140471496928832
#id is the same. Size can dynamically change.



id(v1) #140471497006512

v1=np.append(v1,[100])  #[  0    0    0  10 100]
id(v1) #140471497048784
#id is different. Size is fixed, and a new ndarray object is
created.
```

NumPy arrays vs lists

type	items	mutability	Size	
list	Heterogeneous	mutable	change	
numpy. ndarray	Homogeneous	mutable	fixed	**High-performance array operation**

numeric calculations much easier and faster with ndarray than list

```
L1=list(range(10)) #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v1=np.arange(10)   #[0 1 2 3 4 5 6 7 8 9]
```

We want to calculate the cube of each element

```
L2=[i**3 for i in L1] #[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
v2=v1**3              #[  0   1   8  27  64 125 216 343 512 729]
```

We want to calculate the root mean square of each element

```
import math
L3=[math.sqrt(i) for i in L1]

v3=np.sqrt(v1) #no need to import math
```

Covert lists/ndarrays

np.array() - lists to ndarrays

```
L1=[7,2,9,10]
v1=np.array(L1) #converts a list of numbers to a 1D array
[ 7  2  9 10]

L2=[[5.2,3,4],[9.1,0.1,0.3]]
M=np.array(L2) #converts a list of lists to a 2D array
[[5.2 3.  4. ]
 [9.1 0.1 0.3]]
```

A method is applied to a ndarray object

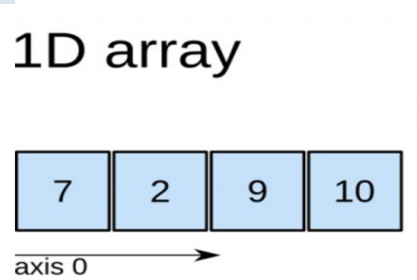
ndarray.tolist() - ndarrays to lists

```
L11=v1.tolist() #converts a 1D array to a list
[7, 2, 9, 10]

L22=M.tolist() #converts a 2D array to a list of lists
[[5.2, 3.0, 4.0], [9.1, 0.1, 0.3]]
```

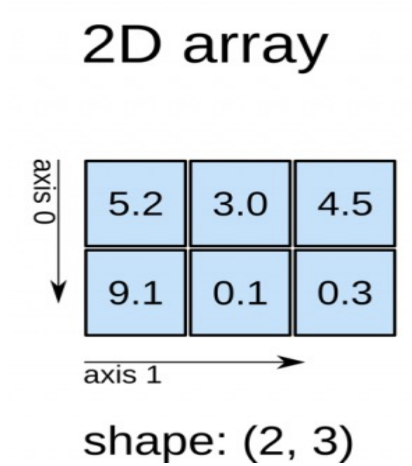
video Attributes-ndim-shape-size

Attributes of an array: shape, size and axis, dtype

`ndarray.ndim`	number of axes, or dimensions, of the array
`ndarray.shape`	tuple of integers that indicate the number of elements stored along each dimension of the array.
`ndarray.size`	total number of elements of the array.
`ndarray.dtype`	the type of the elements in the array




```
print(v1)
 [7  2  9  10]         #vector
print(v1.ndim)   # 1 axis
print(v1.shape) # (4,) 4 elements in one dimention
print(v1.size)   # 4 elements
print(v1.dtype) # int64
```




```
print(M)
[[5.2 3.   4.5] #matrix
[9.1 0.1 0.3]]
print(M.ndim)   # 2 axes
print(M.shape) # (2, 3) 2 rows and 3 columns
print(M.size)   # 6 elements
print(v1.dtype) # float64
```

Create 1D and 2D array of integer random numbers

No need to import the random module

```
np.random.randint(low, high=None, size=None, dtype=int)
```

It generates random integers in range [*low*, *high*).

If *high* is None (the default), then results are from [0, *low*). If size is None returns one value.

```
v=np.random.randint(1,10, size=10)  #1D array, 1 axis
[7 6 3 2 3 9 9 7 4 7]
np.ndim(v)
1

vr=np.random.randint(1,10, size=(1,3)) #2D array, a row vector
[[1 7 4]]

vc=np.random.randint(1,10, size=(3,1)) #2D array, a column vector
[[3]
 [1]
 [1]]

M=np.random.randint(5, size=(2,4)) #2D array, a matrix
[[3 1 4 4]
[4 2 0 4]]
```

look at the doc page https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html

Create 1D and 2D array of real random numbers

To generate random floats in the half-open interval [0.0, 1.0)

```
v1=np.random.random(3)
[0.5769529  0.42219241 0.89814836]  #1D array
```

```
M1=np.random.random((2,3))  #2D array - a matrix
[[0.97055086 0.31126001 0.55647421]
 [0.16726144 0.99078792 0.75163153]]
```

Create 1D array with arange()

np.arange() function is used to generate an array with evenly spaced values within a specified interval. You can define the size step between elements. It returns a 1D array

```
arange(stop)                values generated within [0, stop)
arange(start, stop)         values generated within  [start, stop)
arange(start, stop, step)   values generated within  [start, stop)
with spacing between values given by step.
```

```
a=np.arange(3) #for arrays of integers works as the range function
[0 1 2]

b=np.arange(3,7)
[3 4 5 6]

c=np.arange(1,10,2)
[1 3 5 7 9]

#create 1D array of float
d=np.arange(1,3,0.3)
[1.  1.3 1.6 1.9 2.2 2.5 2.8]

e=np.arange(1,3,0.5)
[1.  1.5 2.  2.5]
```

Create 1D array with linspace()

np.linspace() function is used to create an array with a defined number of elements evenly spaced within a specified range. You can specify the number of elements.
It returns a 1D array

```
linspace(start, stop) creates arrays of 50 (default) evenly spaced
                      numbers over the interval [start, stop]

linspace(start, stop, num=N)   creates N evenly spaced numbers over
                               the interval [start, stop]
```

```
a=np.linspace(0, 1) #create an array of 50 elements in range
[0.1]


b=np.linspace(2.0, 3.0, num=5) #create an array of 5 elements
in range [2.0, 3.0]
[2.   2.25 2.5  2.75 3.  ]
```

video create-1D-2D-zeros-ones-eye

Create 1D and 2D arrays with built-in functions: zeros, eye, ones

```
v = np.zeros((3,)) #tuple (3,) defines the shape, 1D array
print(vr)
[0. 0. 0.] # vector


vc = np.zeros((3,1)) #tuple (3,1) defines the shape 3 rows, 1 column
print(vc)
[[0.]  #column vector
[0.]
[0.]]


M=np.zeros((2, 3)) #tuple (2,3) defines the shape 2 rows, 3 columns
print(M1)
[[0. 0. 0.] #matrix
[0. 0. 0.]]
```

Explore functions ones and eye
https://numpy.org/doc/stable/reference/routines.array-creation.html

Create an array from existing arrays: concatenation

```
np.vstack(tup) Stack arrays in sequence vertically (row wise).
np.hstack(tup) Stack arrays in sequence horizontally (column wise).
tup is sequence type of ndarrays, like a tuple or a list of ndarrays
```

```
A=np.ones((2,3))
[[1. 1. 1.]
[1. 1. 1.]]

B=np.random.randint(10,size=(2,3))
[[5 7 5]
[0 1 5]]

H = np.hstack( (A, B) ) # horizontal concatenation
[[1. 1. 1. 9. 1. 6.]
[1. 1. 1. 0. 1. 4.]]

V = np.vstack( [A, B] ) # vertical concatenation
[[1. 1. 1.]
[1. 1. 1.]
[9. 1. 6.]
[0. 1. 4.]]
```

Shape manipulation: reshape, transpose, flatten

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


M = np.reshape(v, (2,5))   #change shape of an array
[[ 1  2  3  4  5]
[ 6  7  8  9 10]]


Mt=np.transpose(M)              #transpose a matrix
[[ 1  6]
[ 2  7]
[ 3  8]
[ 4  9]
[ 5 10]]


v1=M.flatten()   #method applied to a ndarray returns a 1D array
[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]
```

Indexing on ndarrays

ndarrays can be indexed using the standard Python syntax
https://numpy.org/doc/stable/user/basics.indexing.html

`arr[obj]`

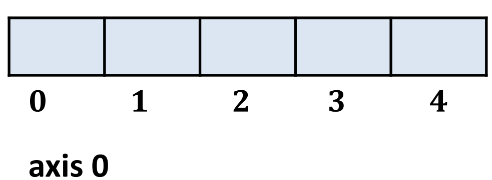
where *arr* is the array and *obj is* the selection.

There are different kinds of indexing available depending on *obj*:

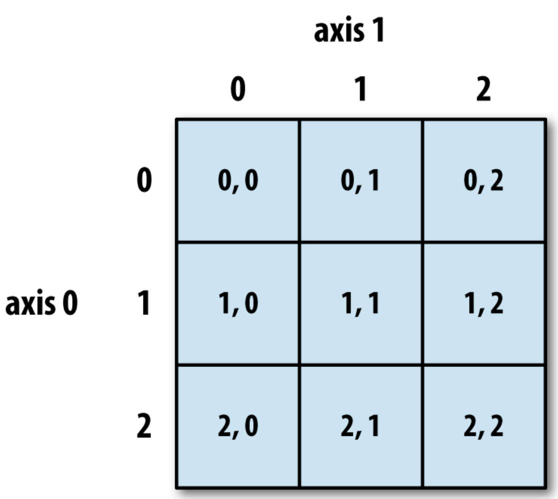
Basic indexing: Single element indexing
Slicing

Advanced indexing: Integer array indexing (Fancy Indexing)
Boolean array indexing (Masking)

1D array



2D array



Basic indexing - 1D array

Single element indexing and slicing work exactly like for other standard Python sequences.

```
arr[index]            # select one element at index
arr[start:end]        # slice from index start through index end-1
arr[start:]           # slice from index start through last index
arr[:end]             # slice from index 0 through index end-1
arr[start:end:step] #slice from index start through not past end by step
```

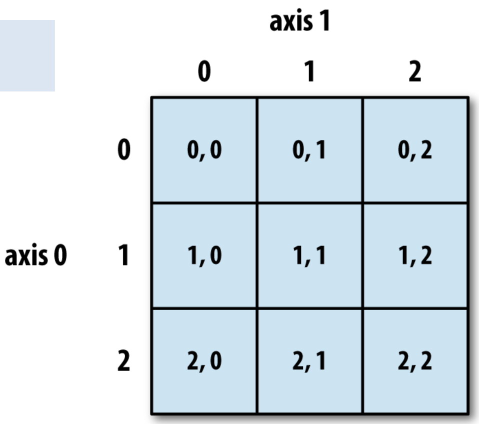
```
arr1 = np.arange(1,11)
[ 1  2  3  4  5  6  7  8  9 10]

arr1[0]    # 1

arr1[-1]   # 10

arr1[3:8] # [4 5 6 7 8]

arr1[5:]   # [ 6  7  8  9 10]

arr1[:5]   # [1 2 3 4 5]

arr1[1:8:2] # [2 4 6 8]
```

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Single element Indexing - 2D array

In a 2D array, to access one element, you need two indices, one for selecting the row and another for selecting the column

```
arr[index_row, index_col]   # select one element
```




```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
[[1 2 3]
[4 5 6]
[7 8 9]]
```

```
arr2[0,0] #same as arr2[0][0] as in nested lists
1
```

You can select a specific row

```
arr2[0] #first row
[1 2 3]

arr2[-1] #last row
[7 8 9]
```

Slicing - 2D array

The standard rules of list slicing apply to basic slicing on a per-dimension basis.

```
arr2d = np.array([[1, 2, 3,4], [5, 6, 7,8], [9, 10, 11,12]])
[[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]]
```

```
arr2d[:,1]  #second column
[ 2  6 10]
```

1	2	3	4
5	6	7	8
9	10	11	12

```
arr2d[0,:]  #first row, same as arr2d[0]
[1 2 3 4]
```

1	2	3	4
5	6	7	8
9	10	11	12

```
arr2d[0,::2]
[1 3]
```

1	2	3	4
5	6	7	8
9	10	11	12

```
arr2d[1:,1:]
[[ 6  7  8]
[10 11 12]]
```

1	2	3	4
5	6	7	8
9	10	11	12

Integer array indexing - 1D

Indexing with an integer array or list of integers allows selection of arbitrary items in the array. This method is also called **fancy indexing**. It is like the simple **indexing** we've already seen, but we pass arrays of **indices** in place of single scalars

```
arr[arr_indices]   # selection of multiple arbitrary elements
```

1	2	3	4	**5**	6	**7**	8	9	10	**arr1**
0	1	2	3	**4**	5	**6**	7	8	10	**indices**

```
arr1[[0,4,6]] # [0,4,6] is a list of indices
[1 5 7]
```

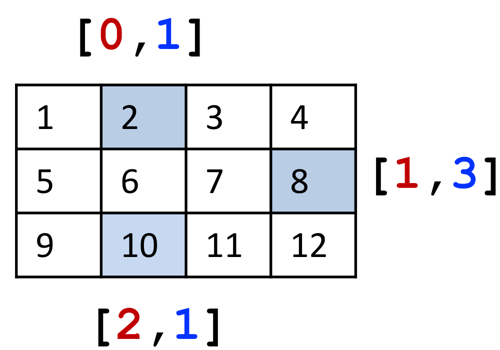
```
indarr=np.array([0,4,6]) #1D array of indices
arr1[indarr]
[1 5 7]
```

Integer array Indexing - 2D array

Integer array Indexing allows selection of arbitrary items in the array.
For 2D array, two integer 1D arrays (or two lists) are needed, one for each dimension.

```
arr[arr_ind_row, arr_ind_col]   # selection of multiple arbitrary elements
```

```
arr2d
[[ 1  2  3  4]
[ 5  6  7  8]
[ 9 10 11 12]]

arr2d[[0,1,2],[1,3,1]]   #provide two lists, one for the rows and another for the columns
[2 8 10]

arr2d[np.array([0,1,2]),np.array([1,3,1])
[2 8 10]
```



Replacing values in 1D array

by using an indexing method on the left side of the equal sign, you can *replace* selected elements of an array.

```
arr[obj]=value
```

The value being assigned to the indexed array must be shape consistent (the same shape or broadcastable to the shape the index produces).

```
arr1=np.arange(1,11)
 [ 1  2  3  4  5  6  7  8  9 10]

arr1[0]=100
[ 100   2   3   4   5   6   7   8   9  10]

arr1[3:7]=12 #a scaler is broadcasted to the entire selection
[ 1  2  3 12 12 12 12  8  9 10]

arr1[5:]=arr1[5:]**2
[  1    2    3    4    5   36   49   64   81  100]
```

Replacing values in 2D array


```
arr2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
[[1 2 3]
[4 5 6]
[7 8 9]]
```

```
arr2[[0,1],[0,1]]=[100,200] #shape consistent
[[100   2   3]
 [  4 200   6]
 [  7   8   9]]
```

```
arr2[[0,1],[0,1]]=0 #a scaler is broadcasted to the entire selection
[[0 2 3]
 [4 0 6]
 [7 8 9]]
```

Summary

Attributes of a NumPy Array

```
ndarray.ndim     number of axes, or dimensions, of the array
ndarray.shape    tuple of integers that indicate the number of
elements
                 stored along each dimension of the array.
ndarray.size     total number of elements of the array.
ndarray.dtype    the type of the elements in the array
```

type conversion

```
ndarray.tolist()       method to convert ndarrays to lists
np.array()             function to convert lists to ndarrays
```

Summary- Create ndarrays using built-in functions

https://numpy.org/doc/stable/reference/routines.array-creation.html

Only 1D array

`np.arange()` function is used to create a 1D array with evenly spaced values within a specified interval.

```
arange(stop)                values generated within [0, stop)
arange(start, stop)         values generated within  [start, stop)
arange(start, stop, step)   values generated within  [start, stop)
with spacing between values given by step.
```

`np.linspace()` function is used to create an array with a defined number of elements evenly spaced within a specified range. You can specify the number of elements.
It returns a 1D array

```
linspace(start, stop)     creates arrays of 50 (default) evenly spaced
                          numbers over the interval [start, stop]

linspace(start, stop, num=N)   creates N evenly spaced numbers over
                               the interval [start, stop]
```

Summary- Create ndarrays using built-in functions

`np.random.randint(low, high=None, size=None, dtype=int)`
generates random integers in range [low, high).
If high is None (the default), then results are from [0, low).

`np.random.random(size=None)` generates random floats in the half-open interval [0.0, 1.0) see also random_samples()

If **size** is None returns one value. To generate a 1D array size=number of elements, to generate a 2D array size=(num rows, num columns)

`np.zeros(`*`shape, dtype=float`*`)` returns a new array of given shape and type, filled with zeros.

`np.ones(`*`shape, dtype=None)`* *r*eturns a new array of given shape and type, filled with ones.

`np.eye(`*`N, M=None`*`)` returns a 2-D array with ones on the diagonal and zeros elsewhere.

Summary - built-in functions for array manipulation

https://numpy.org/doc/stable/reference/routines.array-manipulation.html

`np.vstack(tup)` Stack arrays in sequence vertically (row wise).
`np.hstack(tup)` Stack arrays in sequence horizontally (column wise).
tup is sequence type of ndarrays, like a tuple or a list of ndarrays

`np.reshape(`*`a, newshape`*`)` returns an array with a new shape without changing its data.

`np.transpose(`*`a`*`)` returns an array with axes transposed.

A method is applied to a ndarray object

`ndarray.flatten()` returns a copy of the array collapsed into one dimension.