Flow:

1. Numpy import convention
2. Why Numpy?
3. Initializing array
    a. .array()
    b. .arange()
    c. .linspace()
    d. .zeros() / .ones()
4. Properties of array
    a. .ndim
    b. .shape
    c. .dtype
5. Accessing Elements
    a. Indexing
    b. Slicing
    c. Masking/ Fancy indexing
6. Operations
    a. Arithmetic
    b. Comparison
    c. Matrix Multiplication
7. Axis (show it diagrammatically)
8. Universal Functions
    a. Aggregate
    b. Logical
9. Array Manipulation
    a. Reshaping
    b. Sorting
    c. Splitting
    d. Merging
10. Argument based function (topic name needs some work)
    a. Argwhere
    b. argmin/argmax
    c. argsort
11. Broadcasting
12. Copying Array
13. Misc

Numpy

Import Convention

Import numpy as np

Why Numpy ?

- Supports element wise operation + Vectorization
  - arr = np.array([1, 2, 3, 4])
    - Output: array([1, 2, 3, 4])
  - arr * 2
    - output: array([2, 4, 6, 8])

- Faster Execution speed
  - Python list (took **300 milli sec** for squaring elements)

```
l = range(1000000)
```

```
%timeit [i**2 for i in l]
```

```
339 ms ± 14.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

  - Numpy array (took **2 ms**)
    - Numpy internally uses C arrays.

```
l = np.array(range(1000000))
```

```
%timeit l**2
```

```
2.92 ms ± 418 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Initializing Array

Initializing array using python list	arr1= np.array([4, 11, 53, 2, 9,]) type(arr1) > numpy.ndarray
Creates 2-D array of float data type array([1., 2., 3.],	arr2 = np.array([[2, 7 , 11], [4, 8 , 2]], dtype= float)

[4., 5., 6.]])	
Initializing numpy array range. Similar to python range(). Step size can be float.	arr3 = np.arange(stop= 5) # array([0, 1, 2, 3, 4]) arr4 = np.arange(start = 2, stop = 10,step = 1.5) # array([2. , 3.5, 5. , 6.5, 8. , 9.5])
Returns evenly spaced numbers over specified interval	np.linspace(start = 0, stop = 100,num =5) # array([0, 25., 50. , 75., 100.]) np.linspace(0, 10, 5) # array([0. , 2.5, 5. , 7.5, 10.])
Creates an array with all elements as 0 similar func: np.ones()	zero_arr = np.zeros(3) # [0., 0., 0.] zero_arr_2d= np.zeros((2,3)) # [[0., 0., 0.], [0., 0., 0.]]

Generates array with elements belonging to continuous uniform distribution
Range: [0, 1)
	np.random.rand(3, 2)
	
	# array([[0.81595852, 0.59222987],
	  [0.30536743, 0.27175429],
	  [0.03835455, 0.27976716]])

Properties of Array

number of dimensions of the array.	**arr.ndim** arr1.ndim # returns 1 arr2.ndim # returns 2
shape of array	**arr.shape** arr1.shape # returns (5,) arr2.shape # returns (2, 3)
datatype of array.	**arr.dtype** arr1.dtype # returns int64

Accessing Elements

Accessing Single element (Indexing)

access element present at that index. Index start from 0	arr1[2] # returns 53 arr2[1, 2] # returns 6
Negative index based indexing	arr1[-1] # returns 6

Accessing Sequence(Slicing)

Slice out and get part of the numpy array. Can use negative indexes for slicing as well. Slicing returns View not copy.	arr1[3:] # returns [2, 9] arr1[:4] # returns [4, 11, 53, 2] arr[1: 4: 2] #returns array([11, 2]) arr1[-4: -1] # returns [11, 53, 2]
	arr2[:1, :] # fetches first row # [[2., 7. , 11.]] arr2[:, 2:] # fetches third column # [[11.], [2.]]

Accessing based on condition (Masking)

Indexing based on condition. Masking creates a copy of the array not a view.	arr1[arr1 > 8] # returns [11, 53, 9] arr1[(arr1 >5) & (arr1 <=11)] # returns [11, 9]

Operations

Arithmetic

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

Element wise Addition	a + b [or np.add (a, b)] # [2, 3, 5, 6]

Element wise Subtraction	a - b [or np.subtract(a,b)] # [0, 1, 1, 2]
Element wise Multiplication	a * b [or np.multiply(a, b)] # [1, 2, 6, 8]
Element wise Division	a/b [pr np.divide(a, b)] # [1., 2., 1.5., 2.]

Comparison

Element wise comparison Returns bool array	a==b, a>=b, a<=b
Array wise equality Returns True/False	np.array_equal(a, b)

Matrix Multiplication

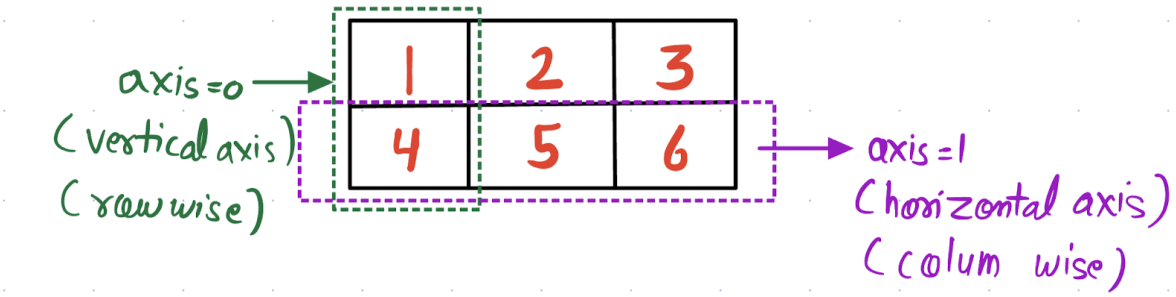
mat1 = np.array([[2], [1]])
mat2 = np.array([[2, 4]])

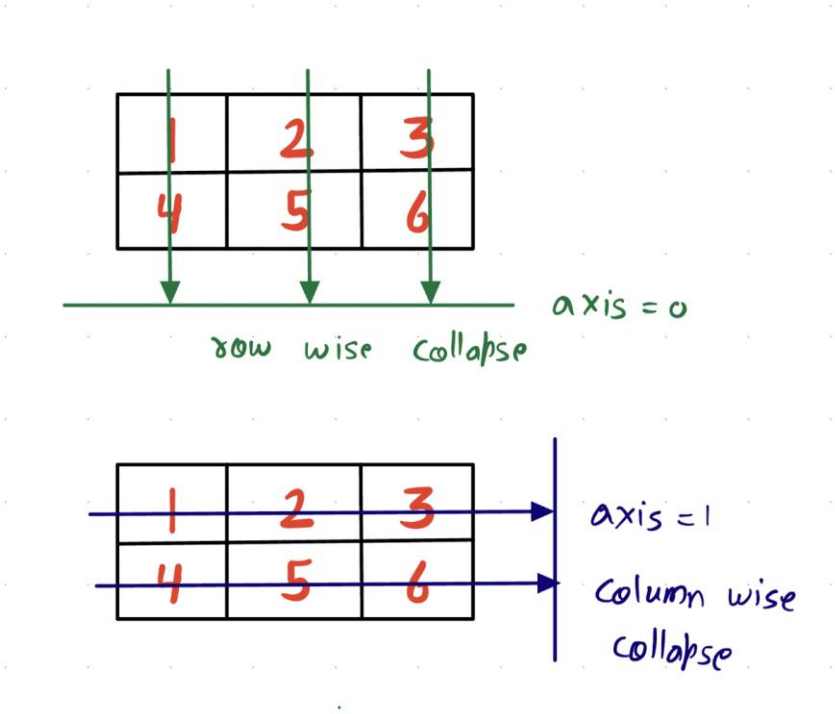
Returns matrix mul. of arrays provided the condition for matrix multiplication is satisfied.	**np.matmul**(mat1, mat2) # array([[4, 8], [2, 4]])
	mat1 @ mat2 # array([[4, 8], [2, 4]])
Performs matrix multiplication if both	**np.dot()**

inputs are 2D. Other cases for np.dot(): **Case1:** It performs dot product if both inputs are scalar **Case2:** Perform simple multiplication if both inputs are scalars	np.dot(mat1, mat2) # returns ([[4, 8], [2, 4]] np.dot(a, b) # [2, 4, 6] =< 1*2 + 2*2 + 3*2 np.dot(2, 3) # returns 6

Axis

Diagram ref: [Cheat sheet Numpy Python copy.indd]





Universal Functions

Aggregate

Sums all the elements of array Sums the elements along the vertical axis(rowwise). We can also take sum along horizontal axis (axis =1)	**np.sum()** np.sum(arr2) # retuns 34.0 np.sum(arr2, axis = 0) # returns [6., 15., 3.]
Takes mean of all the elements of array Takes mean along the horizontal axis (column wise)	np.mean(arr2) # returns 5.666667 np.mean(arr2, axis = 1) # [6.6667, 4.66667]
Returns element with minimum value. Can also find mean row wise/ column	np.min(arr2) # returns 2.0

wise using axis 0/1 Other similar func: np.max()	

Logical

Returns True if the any of the corresponding elements in the array follow provided condition	np.any(arr1 < arr3) # returns True
Returns True only if the all of the corresponding elements in the array follow provided condition	np.all(arr1 < arr3) # returns False
Function signature: np.where(condition, [x, y]) Vectorized if else over an array. Returns an array where value = x if the condition is True else y.	np.where(arr1 > 2, 1, 0) # array([1, 1, 1, 0, 1])

Array Manipulation

Reshaping

```
array3d = np.array([[[1 , 2]],
                    [[5, 6]]])
```

Reshapes the array Can use -ve index in reshape	array3d.reshape(2, 2) # returns [[1, 2], [5, 6]] arr5.reshape(2, -1) # [[1, 2], [5, 6]]

Returns a flattened array i.e. 1D array. Returns copy of array	array3d.flatten() # returns array([1, 2, 5, 6])
Returns a flattened array. Returns view of array	array3d.ravel()
Transposes the array	reshaped_arr.T np.transpose(array3d, axis = [2, 1, 0]) # array([[[1, 5]], [[2, 6]]])

Sorting

Sorts original array. Doesn't return anything	arr2.sort()
Returns sorted array. Doesn't make changes to original array	np.sort(arr) # array([[2., 7., 11.], [2., 4., 8.]])
Sorts array row wise i.e. along the vertical axis Sorts array along the horizontal axis i.e. column wise	np.sort(arr, axis = 0) # array([[2., 7., 2.], [4., 8., 11.]]) np.sort(arr, axis = 1) # array([[2., 7., 11.], [2., 4., 8.]])
Returns indices that would sort the array	np.argsort(arr) # [4, 0, 3, 1, 2]

Splitting

Split array into multiple sub arrays	np.split(arr, indices_or_sections= 2) # [array([1, 2]), array([3, 4])]

	np.split(arr, indices_or_sections = [1, 3]) # `[array([1]), array([2, 3]), array([4])]`
Split along horizontal axis i.e. column wise	np.hsplit(arr2, [1, 2]) `# [array([[2.], [4.]]), array([[7.], [8.]]), array([[11.], [ 2.]])]`
Split along vertical axis i.e. row wise	np.vsplit(arr2, 2) `# [array([[ 2., 7., 11.]]), array([[4., 8., 2.]])]`

Merging

Stacks array vertically ie. row wise append (axis = 0)	**np.vstack()** np.vstack((arr, arr, arr)) # [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
Stacks array horizontally i.e. column wise append (axis = 1)	**np.hstack()** arr = arr.reshape(4,1) # [[0], [1], [2]] np.hstack((arr, arr, arr)) # [[0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3]]
Concatenate two or more array along the given axis	np.concatenate([arr, arr]) # [1, 2, 3, 4, 1, 2, 3, 4]

	arr_2d = arr.reshape(1,-1) np.concatenate([arr, arr], axis = 0) #[[1, 2, 3, 4], [1, 2, 3, 4]] np.concatenate([arr, arr], axis = 1) array([[1, 2, 3, 4, 1, 2, 3, 4]])

Argument based function

Get indices of non zero elements	np.argwhere(arr1)
	
	# array([[0],
	  [1],
	  [2],
	  [3]
	  [4]])

Get indices minimum value. Other similar func: np.argmax()	np.argmin(arr1) # returns 3 np.argmin(arr2) # array([0, 0, 1])
Returns indices that would sort the array	np.argsort(arr) # array([3, 0, 4, 1, 2])

Broadcasting

For each dimension (going from right side)
  1. The size of each dimension should be same OR
  2. The size of one dimension should be 1
 	
Rule 1 : If two arrays differ in the number
	Shape of arr1= (1, 2)
	Shape of arr2 = (2, 2)
	
	Shape of [arr1 + arr2] = (2, 2) # **Rule 2**
	
	arr1 shape = (2, 1)
	arr2 shape = (2, 2)

of dimensions, the shape of one with fewer dimensions is padded with ones on its leading(Left Side). **Rule 2 :** If the shape of two arrays does not match in any dimensions, the array with shape equal to 1 is stretched to match the other shape i.e. broadcasted. **Rule 3 :** If in any dimension the sizes disagree and neither equal to 1 , then Error is raised.	arr1 + arr2 shape = (2,2) # **Rule 2** arr1 shape = (2, 4) arr2 shape = (4, 4) arr1 - arr2 shape = Error # **Rule 3** arr1 shape = (15, 3, 5) arr2 shape = (3, 1) arr1 + arr2 shape = (15, 3, 5) # **Rule1 + Rule2**

Copying Array

Creates a copy of an array. Masking and array op creates copy.	copy = arr1.copy()
Creates View of an array. Slicing creates view	view = arr1.view()

Misc

Change the datatype of an array	arr.astype(int)