Numpy

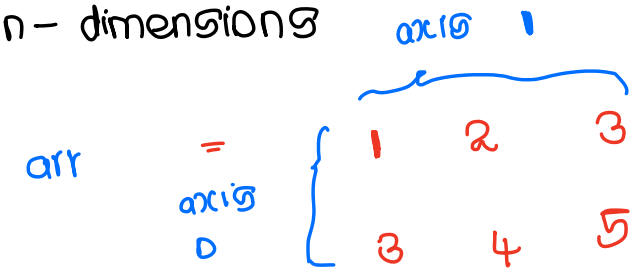
- open source library
- >30% code: written in C
- numpy uses semantic versioning: 1 . 19 . 1
  - major version (changes to API)
  - minor version (new features)
  - patch version
- library to represent & manipulate tensors. (ndarray)
- many libraries built on top of numpy
  ex: scipy, matplotlib, NLTK
- faster and more efficient than python:
  - array operations in numpy use C.
  - since numpy arrays must have homogenous data can use contiguous memory blocks.
  - Easier to cache and look up vs. python lists that allow heterogenous data through pointers

ex: if a list of data of same type (8 bits) & stored contiguously. third item in collection = start + 8 bits x 3
mem address

- The homogenous data requirement in numpy can be a challenge when working with datasets that have different types
- solution: use pandas → convert data → use numpy

Ndarray

- one data type
- fixed size
- n-dimensions




```
arr[1, 2] = 5
```

- automatically infers data type

```
arr.dtype  // int64
```

- can conver to other types

```
float32_arr = arr2d.astype(np.float32)
```

- can check the number of dimensions

```
arr2d.ndim  // 2
arr2d.shape // (2, 3)
```

Numpy Filling Methods

Placeholders

ones : fill with one

zeros : fill with zeros

eye : create an identity matrix

ex: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

diag : create a diagonal given a collection

ex: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$

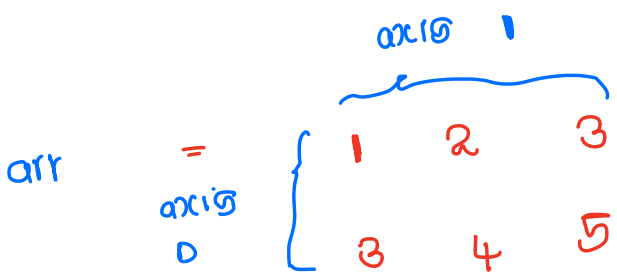
arange : given a starting value, end value and step: creates a nd array

```
np.arrang (0.1, 0.5, 0.2)
array ( [0.1, 0.3, 0.5] )
```

linspace: create evenly spaced intervals:

```
np.linspace (start_value, end_value, num intervals)
```

Array Indexing



arr [1 , 2] = 5 //

additional dimensions after axis 0 denoted by ,

ex:

arr [0, -2] = 2 //

first row, second last element

arr [-1, -1] = 5 //

last row, last column element

arr [0] = [1, 2, 3] //

entire first row

```
arr[:, 0] = [1, 3]
```

entire first column

```
arr[:, :2]
```

first two columns

Numpy Array Math & Universal functions

- ~60 ufunc (universal functions)
- easier and more efficient modications of ndarray. (rather than for loops)
- ufuncs implemented in C

ex:
- basic arithmetic operations:
  - add
  - substract
  - exp
- trigonometric operators:
  - arctan
  - cos

- bitwise operators
  - AND
  - OR
- Get Idx Min / Max elements
- Numpy uses operator overloading when using ndarrays with arithmetic operators

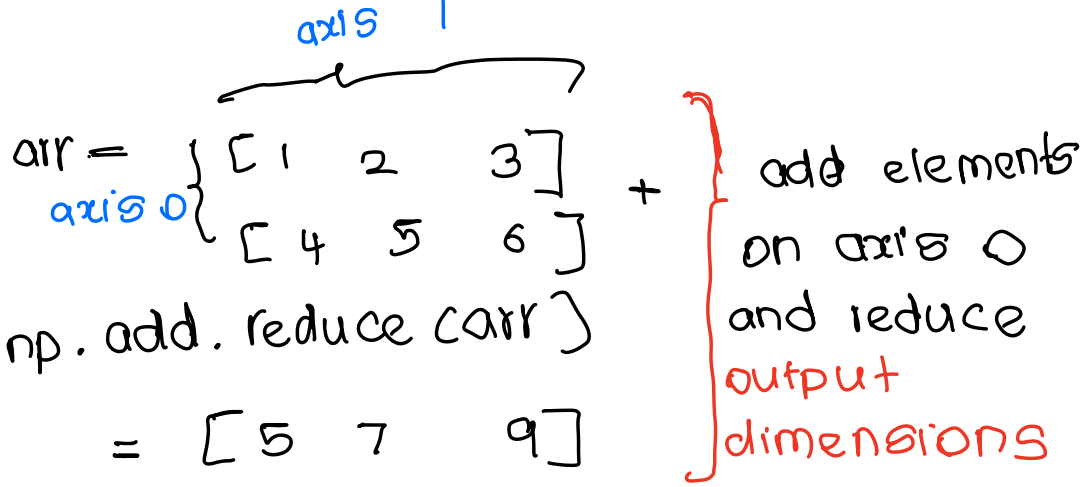
```
np.add(arr, 1) == arr + 1
```

if arr is a ndarray

Reduce :

- To reduce dimensions of a ndarray by applying an operator along an axis.
- Axis 0 by default (↓)

ex:



- performs operations paralelly
- dask: library to scale over multiple computational nodes
- can write reduce more simply

  arr.sum(axis = 0)
- ufuncs for standard deviation and variance: population variance & std.deviation not the sample standard deviation of population variance,

  np.std(arr):
  population std deviation

  np.std(a, ddof = 1):
  sample std deviation

Numpy Broadcasting

How can we perform matrix operations when dimensions don't match

ex: $Arr = \begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{bmatrix}$

$arr2 = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$

$arr + 1 = \begin{matrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{matrix} + \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$

$arr + arr2 = \begin{matrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{matrix} + \begin{matrix} 4 & 5 & 6 \\ 4 & 5 & 6 \end{matrix}$

∴ perform operations on linear algebra

* Memory Views

- slicing, indexing a ndarray creates a memory view.
- helps to save memory
- changing reference → change OG arr

ex:

$$arr = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \end{bmatrix}$$

```
first_row = arr[0]
first_row += 99
```

$$arr = \begin{bmatrix} 100 & 101 & 102 \\ 6 & 7 & 8 \end{bmatrix}$$

//

- avoid changing OG if we use copy()

ex:

$$arr = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \end{bmatrix}$$

```
first_row = arr[0].copy()
first_row += 99
```

$$arr = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \end{bmatrix}$$

* Fancy Indexing

- can use "fancy indexing" to create deep copies
- pass list of indexes we want

ex:

arr = [1 2 4 5 6]

fancy_idx_copy = arr[[0, 2]]

fancy_idx_copy += 99

fancy_idx_copy = [99 101]

arr = [1 2 4 5 6]

arr = $\begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \end{bmatrix}$ //

copy of first and last columns =

arr[:, [2, 0]]

= $\begin{bmatrix} 3 & 1 \\ 8 & 6 \end{bmatrix}$ can rearrange as well

Boolean Masking

Arrays with values: TRUE / FALSE

ex: arr = 1 2 3
4 5 6

greater_3_mask = arr > 3

greater_3_mask = 0 0 0
1 1 1

arr[greater_3_mask] = 4 5 6

can be chained together using logical operators like AND, OR etc:

arr = 1 2 3
4 5 6

combined_mask = arr > 3 AND arr < 6

combined_mask = 0 0 0
1 1 0

arr[combined mask] = 4 5

- np.where(arr > 2, 1, 0): values > 2 → 1
values < 2 → 0

Random Number Generator

* can create random data multiple methods:
  - using std deviation
  - using uniform distribution
  - using Normal distribution
  - etc:
* can pass the seed to produce same random generation

```
np.random.seed(123)
np.random.rand(3)
```

* useful for reproducable results & unit testing
* we can also use default_rng or random state object
* calling sequence matters

ex:

```
rng = np.random.default_rng(seed=123)
val1 = rng.rand(3)
val2 = rng.rand(3)
```

} val 1 ≠ val 2

Reshaping Numpy Array

- Changing shape is important when we want to change input data → specific format
- we can only change shape in a way our data fits

ex: arr = [1, 2, 3, 4, 5, 6]

arr.reshape(2,3) ✓

arr.reshape(1,6) ✓

arr.reshape(4,4) ✗ 4 x 4 = 16 elements //

- reshape returns a shallow copy
- we can use -1 instead of specific values

ex: arr.reshape(2, -1)

-1 = 3 if we have 6 elements

-1 = 4 if we have 8 elements

- for flattening an array

arr.reshape(-1) : 1D array

- other ways to flatten an array:

  arr.flatten() : deep copy,

- concatenation:

  arr = [1 2 3]

  np.concatenate([arr, arr])

  [1 2 3 1 2 3]

- we can also specify axis to concatenate along a specific axis

Linear Algebra:

- $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 10 \\ 36 \end{bmatrix}$

  $2 \times \underline{2} \quad \underline{2} \times 1 \qquad 2 \times 1$

  has to be identical

- numpy can be forgiving as it uses broadcasting
- to get dot product we can:
  - matmul
  - dot (recomended)
- transpose