**Numpy-I**

**Content**

* **Installing and Importing Numpy**
* **Motivation: Why to use Numpy? - How is it different from Python Lists?**
* **Creating a Basic Numpy Array**
  + From a List - array()
  + From a range with some step size - arange()
  + From a range with certain number of elements - linspace()
  + shape
* **How numpy works under the hood?**
* **2-D arrays (Matrices)**
  + reshape()
* **Creating some special arrays using Numpy**
  + zeros()
  + ones()
  + diag()
  + identity()
* **Indexing and Slicing**
  + Indexing
  + Slicing using indices
  + split(), hsplit(), vsplit()
* **Operations on Numpy Arrays**
  + Algebric operations on np arrays with single numbers
  + Masking and Filtering
  + Vectorization - np.vectorize()
  + Operations on two or more np arrays
  + Matrix Multiplication - matmul(), @, dot()
  + Transpose
  + Converting Matrix back to Vector - flatten(), ravel()
  + Merging Arrays - hstack(), vstack(), concatenate()
  + Sorting Arrays - sort(), argsort()
  + 3 dimensions: Tensors
* **Broadcasting**

**Importing Numpy**

* You should be able to import Numpy after installing it
* We’ll import numpy as its **alias name np** for ease of typing
* Otherwise, every time we want to use something from Numpy,
  + we’ll have to **keep writing whole** numpy.<something>

import numpy as np

**Motivation: Why to use Numpy?**

**Do you remember lists?**

list1 = [1, 2, 3]

list1

[1, 2, 3]

**Do you remember what happens if we multiply list with a positive integer?**

list1 \* 2

[1, 2, 3, 1, 2, 3]

* It **doesn’t give the element-wise mathematical behaviour** that we expect
* It gives a **concatenation of same list repeated n times**
* list1\*2 is same as list1 + list1

**So, we need to use a mathematical library like Numpy**

* That can give us **desired mathematical behaviour**

**Creating a Basic Numpy Array**

**First method we’ll see in Numpy is array()**

* We **pass a Python list** into np.array()
* It **converts** that Python list into a **numpy array**

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

arr1

array([1, 2, 3])

* This is **NOT a normal Python list**
* It’s a **numpy array**

**Now, Let’s multiply it by 2 and see what happens**

arr1 \* 2

array([2, 4, 6])

* With **numpy array**, we **get the mathematical behaviour we desire**
* In DS and ML, we’ll mostly be working with numpy arrays
* Very rarely we’ll use lists directly

**How numpy works under the hood?**

* It’s a **Python Library**
* We’ll **write code in Python** to use numpy

**However, numpy itself is written in C**

* Allows numpy to **manage memory very efficiently**
* Languages like **C are closer to hardware**
  + Because **data types are pre-defined**
  + C is a **statically-typed language**
* **Python is slower than C**
  + Because it is a **dynamically-typed langauge**
  + Data types of variables are figured out dynamically

**Why do we need this?**

* To perform high-processing tasks in Data Science and Machine Learning
* We need to use libraries which are **fast and efficient** in terms of **memory** and **processing**
* So we write code in Python to use Numpy
* But **under the hood**, it uses **C**
* So numpy arrays are **NOT** really **Python lists**
* They are basically **C arrays**

**Now Let’s check type of arr1**

type(arr1)

numpy.ndarray

* So arr1 is numpy.ndarray
* **ndarray means an n-dimensional array**

**So, What are the some advantages of Numpy?**

* Numpy **makes use of array features** to **perform mathematical operations**
* Processing speed of numpy arrays is much **faster than lists**
* It **manages memory very well**
* Mostly used to work with **matrices, vectors, linear algebra**.

**How can we check dimensions of a numpy array?**

arr1.shape

# arr1 has 3 elements

(3,)

**Creating arrays within a range using Numpy**

**np.arange()**

* Similar to range()
* We can pass **starting** point, **ending** point (not included in array) and **step-size**
* arange(start, end, step)

arr2 = np.arange(1, 5)

arr2

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

arr2\_stepsize = np.arange(1, 5, 2)

arr2\_stepsize

array([1, 3])

* np.arange() behaves in same way as range() function

**A small difference is that:**

* 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])

**np.linspace()**

* **Three arguments**, where the **first two arguments** are the **start and end values**.
* **Third argument** of **np.arange() is the increment**, while for **np.linspace() it is the total number of points in the array**.
* We already saw np.arange() in previous class.

**Let’s see np.linspace() now:**

np.linspace(0, 10, 11)

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

It created an array having 11 elements where start value is 0 and end value is 10.

* Now, whether to use np.arange or np.linspace is mostly a matter of **personal preference**
* But it is **generally recommended to use np.linspace() whenever the increment is a non-integer**.

**For example:**

np.linspace(1, 10, 20)

array([ 1. , 1.47368421, 1.94736842, 2.42105263, 2.89473684,

3.36842105, 3.84210526, 4.31578947, 4.78947368, 5.26315789,

5.73684211, 6.21052632, 6.68421053, 7.15789474, 7.63157895,

8.10526316, 8.57894737, 9.05263158, 9.52631579, 10. ])

**We got 20 evenly spaced values b/w 1 and 10**

**Let’s see the C type behaviour of Numpy**

* Pass a **floating point number** as one of the values in **np array**
* **All values will become floating point**
* Because **one single C array** can store values of **only one data type**
* **int is raised to float**

arr4 = np.array([1, 2, 3, 4])

arr4

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

arr4 = np.array([1, 2, 3, 4.0])

arr4

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

* If you press **“Shift+tab”** inside np.array() function
* You can see **function’s signature**
  + **name**
  + **input parameters**
  + **default values** of input parameters
* Look at **dtype=None**
  + dtype means **data-type**
  + which is **set to None by default**

**What if we set dtype to float?**

arr5 = np.array([1, 2, 3, 4])

arr5

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

arr5 = np.array([1, 2, 3, 4], dtype="float")

arr5

# We can enforce data-type of values in np array

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

**Another way np array behaves like C arrays and not Python lists**

* In Python lists, number values can be **arbitrarily large or small**
* There’s **no limit** to how large or small a value can be in Python lists
* In Python, there’s **usually no overflow of number values**

**However, in C, C++ and Java, there’s overflow of values**

* As soon as a **number crosses the max possible** value
* The number gets **wrapped around to a smaller value**
* This type of **behaviour** is **exhibited by np arrays**

# In python

100\*\*10

# no overflowing

100000000000000000000

arr6 = np.array([0, 10, 100])

arr6 \*\* 10 # raising powers of each value to 10

# 100^10 is overflowing, so we get incorrect answer for it

array([ 0, 10000000000, 7766279631452241920])

**Working with 2-D arrays (Matrices)**

**How do we create a matrix using numpy?**

m1 = np.array([[1,2],[3,4]])

m1

# Nicely printing out in a Matrix form

array([[1, 2],

[3, 4]])

# Multiplication with integer works the same

m1 = np.array([[1,2],[3,4]])

m1 \* 2

array([[2, 4],

[6, 8]])

**How can we create high dimensional arrays using np.arange()?**

* Using reshape()
* **First argument** is **no. of rows**
* **Second argument** is **no. of columns**

m2 = np.arange(1, 13)

m2

array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

* We can pass the **desired dimensions** of array in reshape()
* **But there’s a catch here**

**In what ways can we convert this array with 12 values into high-dimensional array?**

**Can we make m2 a**4×44×4**array?**

* Obviously NO
* 4×44×4**requires 16 values**, but **we only have 12 in m2**

m2 = np.arange(1, 13)

m2.reshape(4, 4)

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-33-fc70b006b379> in <module>

1 m2 = np.arange(1, 13)

----> 2 m2.reshape(4, 4)

ValueError: cannot reshape array of size 12 into shape (4,4)

**So, What are the ways in which we can reshape it?**

* 4×34×3
* 3×43×4
* 6×26×2
* 2×62×6
* 1×121×12
* 12×112×1

m2 = np.arange(1, 13)

m2.reshape(4, 3)

array([[ 1, 2, 3],

[ 4, 5, 6],

[ 7, 8, 9],

[10, 11, 12]])

m2 = np.arange(1, 13)

m2.reshape(1, 12)

array([[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]])

**How many values will m3 have?**

m3 = np.arange(9)

m3

# 9 values - because 0 is default start

array([0, 1, 2, 3, 4, 5, 6, 7, 8])

**Let’s reshape m3 to**3×33×3

m3 = np.arange(9)

m3 = m3.reshape(3, 3)

m3

array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

**How many dimensions m3 has?**

* 9 ?? - NO
* 3 ?? - NO

**It has 2 dimensions**

* m3 is a **2D array**
* The **magnitude of those two dimensions is 3 and 3**

**So, DO NOT confuse the dimensions of array with no. of rows and columns**

m3.shape

(3, 3)

**Now, What’s the difference b/w (9,) and (9, 1)?**

* **(9,)** means its a **1D array**
* **(9, 1)** means its a **2D array**

m3 = np.arange(9)

m3

array([0, 1, 2, 3, 4, 5, 6, 7, 8])

m3.shape # 1D array

(9,)

m3 = np.arange(9)

m3 = m3.reshape(9, 1)

m3

array([[0],

[1],

[2],

[3],

[4],

[5],

[6],

[7],

[8]])

m3.shape # 2D array - 9 rows & 1 column

(9, 1)

**Remember! Size and Dimensions are two different things**

**Creating some special arrays using Numpy**

**A great way to create an array and initialize it with zeroes**

np.zeros(3)

# Pass in how many values you need in array

# All values will be zeroes

array([0., 0., 0.])

* We can also **pass in shape** in case we want to create 2D array
* **Pass shape as a Tuple**

np.zeros((2, 3))

array([[0., 0., 0.],

[0., 0., 0.]])

**Initializing values to 1**

# Just like np.zeroes, but initialize all values to 1

np.ones(3)

array([1., 1., 1.])

# 2D

np.ones((2,3))

array([[1., 1., 1.],

[1., 1., 1.]])

**Now, do we need np.twos(), np.threes(), np.fours(), … np.hundreds()?**

* NO
* We can just create array using np.ones() and multiply with required value

np.ones((2, 3)) \* 5

array([[5., 5., 5.],

[5., 5., 5.]])

**Creating a Diagonal Matrix**

np.diag([1, 2, 3])

# We pass values for diagonal elements as a list

# All other elements are zero

array([[1, 0, 0],

[0, 2, 0],

[0, 0, 3]])

**Creating an Identity Matrix**

* It’s a **square matrix**
* Where **all diagonal values are 1**
* **All non-diagonal values are 0**

np.identity(3)

# Pass in the single dimension of required square identity matrix

array([[1., 0., 0.],

[0., 1., 0.],

[0., 0., 1.]])

**Indexing and Slicing**

* Now we will look at some of the operations we can perform on np arrays
* Let’s start by creating a 3×43×4 matrix

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

m1

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

**Indexing in np arrays**

* Works same as lists

m1[0] # gives entire first row

array([0, 1, 2, 3])

m1[0][0] # gives value in first row and first column

0

* There’s a **simpler way** to do this in numpy arrays
* We just use [0, 0] (**indexes separated by commas**)

**What will be the output of this?**

m1[1, 1]

5

**Slicing**

* Similar to Python lists
* We can **slice out and get a part of np array**
* Need to **provide two slice ranges** - one for **row** and one for **column**
* Can also **mix Indexing and Slicing**

# First get row 0

# Then get columns 1 to 3 (Column 3 not included)

m1[0, 1:3]

array([1, 2])

**Can you just get this much of our array m1?**

[[5, 6], [9, 10]]

**Remember our m1 is:**

m1 = [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]

# First get rows 1 to all

# Then get columns 1 to 3 (not included)

m1[1:, 1:3]

array([[ 5, 6],

[ 9, 10]])

**What if we want this much part?**

[[2, 3], [6, 7], [10,11]]

# First get all rows

# Then get columns 2 to all

m1[:, 2:]

array([[ 2, 3],

[ 6, 7],

[10, 11]])

**What if we need 1st and 3rd column?**

[[1, 3], [5, 7], [9,11]]

# Get all rows

# Then get columns from 1 to all with step of 2

m1[:, 1::2]

array([[ 1, 3],

[ 5, 7],

[ 9, 11]])

* **We can also pass indices of required columns as a Tuple** to get the same result

# Get all rows

# Then get columns 1 and 3

m1[:, (1,3)]

array([[ 1, 3],

[ 5, 7],

[ 9, 11]])

**np.split()**

* Splits an array into multiple sub-arrays as views

**It takes an argument indices\_or\_sections**

* If indices\_or\_sections is an **integer, n**, the array will be **divided into n equal arrays along axis**.
* If such a split is not possible, an error is raised.
* If indices\_or\_sections is a **1-D array of sorted integers**, the entries indicate **where along axis the array is split**.
* If an index **exceeds the dimension of the array along axis**, an **empty sub-array is returned** correspondingly.

x = np.arange(9.0)

x

array([0., 1., 2., 3., 4., 5., 6., 7., 8.])

np.split(x, 3)

[array([0., 1., 2.]), array([3., 4., 5.]), array([6., 7., 8.])]

np.split(x, [3, 5, 6, 10])

[array([0., 1., 2.]),

array([3., 4.]),

array([5.]),

array([6., 7., 8.]),

array([], dtype=float64)]

**np.hsplit()**

* Splits an array into multiple sub-arrays **horizontally (column-wise)**.

x = np.arange(16.0).reshape(4, 4)

x

array([[ 0., 1., 2., 3.],

[ 4., 5., 6., 7.],

[ 8., 9., 10., 11.],

[12., 13., 14., 15.]])

np.hsplit(x, 2)

[array([[ 0., 1.],

[ 4., 5.],

[ 8., 9.],

[12., 13.]]),

array([[ 2., 3.],

[ 6., 7.],

[10., 11.],

[14., 15.]])]

np.hsplit(x, np.array([3, 6]))

[array([[ 0., 1., 2.],

[ 4., 5., 6.],

[ 8., 9., 10.],

[12., 13., 14.]]),

array([[ 3.],

[ 7.],

[11.],

[15.]]),

array([], shape=(4, 0), dtype=float64)]

**np.vsplit()**

* Splits an array into multiple sub-arrays **vertically (row-wise)**.

x = np.arange(16.0).reshape(4, 4)

x

array([[ 0., 1., 2., 3.],

[ 4., 5., 6., 7.],

[ 8., 9., 10., 11.],

[12., 13., 14., 15.]])

np.vsplit(x, 2)

[array([[0., 1., 2., 3.],

[4., 5., 6., 7.]]),

array([[ 8., 9., 10., 11.],

[12., 13., 14., 15.]])]

np.vsplit(x, np.array([3, 6]))

[array([[ 0., 1., 2., 3.],

[ 4., 5., 6., 7.],

[ 8., 9., 10., 11.]]),

array([[12., 13., 14., 15.]]),

array([], shape=(0, 4), dtype=float64)]

**Operations on Numpy Arrays**

**Algebric operations on np arrays with single numbers**

* **Happens on each element of np array**
* Just like multiplication works on each element
* Similarly **addition/subtraction** of np array with a number **works on each element of np array**

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

m1 + 2

array([[ 2, 3, 4, 5],

[ 6, 7, 8, 9],

[10, 11, 12, 13]])

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

m1 \* 2

array([[ 0, 2, 4, 6],

[ 8, 10, 12, 14],

[16, 18, 20, 22]])

* Similarly, we can do **subtraction**, **raise-to-power**, …and so on

**This is similar to mapping in map() function**

* Each value gets mapped to a new value based on the operation performed

map(lambda x: x \* 2, [...])

**Masking**

**What would happen if we do this?**

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

m1 < 6

array([[ True, True, True, True],

[ True, True, False, False],

[False, False, False, False]])

* Same thing as algebric operations
* All the **values before 6 return True** and **all values after 6 return False**
* **Comparison operation also happens on each element**
* A **matrix having boolean values** True and False is returned
* Based on the given **condition**

**Now, Let’s use this to filter or mask values from our array**

* **Condition will be passed instead of indices and slice ranges**

m1[m1 < 6]

# Value corresponding to True is retained

# Value corresponding to False is filtered out

array([0, 1, 2, 3, 4, 5])

**This is similar to filtering using filter() function**

filter(lambda x: x < 6, [...])

**How can we filter/mask even values from our array?**

m1[m1%2 == 0]

array([ 0, 2, 4, 6, 8, 10])

**But did you notice that matrix gets converted into a 1D array after masking?**

m1

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

m1[m1%2 == 0]

array([ 0, 2, 4, 6, 8, 10])

**It happens because**

* To retain matrix shape, it **has to retain all the elements**
* It **cannot retain its**3×43×4**with lesser number of elements**
* So, this filtering operation **implicitly converts high-dimensional array into 1D array**

**If we want, we can reshape the resulting 1D array into 2D**

* But, we need to know **beforehand** what is the **dimension or number of elements** in resulting 1D array

m1[m1%2==0].shape

(6,)

m1[m1%2==0].reshape(2, 3)

array([[ 0, 2, 4],

[ 6, 8, 10]])

**Vectorization**

* We have already seen vectorization some time ago

**Remember doing scaler operations on np arrays?**

A \* 2

**That’s vectorization**

* Vectorization helps us to **perform operations directly on Arrays instead of scaler**.
* Operation gets performed on each element of np array

**Revisiting the example:**

A \* 2

array([[ 0, 2, 4, 6],

[ 8, 10, 12, 14],

[16, 18, 20, 22]])

**np.vectorize()**

* np.vectorize() defines a **vectorized function**
* It **takes numpy arrays as inputs** and **returns a single numpy array or a tuple of numpy arrays**.
* The vectorized function **evaluates element by element of the input arrays** like the python map function

**Let’s plot graph for y = log(x) (Log function) using np.vectorize()**

* We will pass in a numpy array, as **it can then take a vector/array/list as input**
* It will **return the vectorized form of math.log() function**

import math

import matplotlib.pyplot as plt

x = np.arange(1, 101)

y = np.vectorize(math.log)(x)

plt.plot(x, y)

plt.show()

**Vectors, Matrix and Tensors**

1. **Vector** —> **1-Dimensional** Array
2. **Matrix** —> **2-Dimensional** Array
3. **Tensor** —> **3 and above Dimensional** Array

**Tensor is a general term we use**

* Tensor can also be less than 3D
* **2D Tensor** is called a **Matrix**
* **1D Tensor** is called a **Vector**

**Operations on two or more np arrays**

**Algebric operations on two np arrays**

# Corresponding elements of arrays get added

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

b = np.array([2, 2, 2])

a + b

array([3, 4, 5])

# Corresponding elements of arrays get multiplied

a \* b

array([2, 4, 6])

**Why did it not do Matrix Multiplication?**

* **Check dimensions of a and b**
* Both **a and b are 1D arrays** - shape is **(3,))**
* a and b are **NOT matrices** - **Dimension is NOT**1×31×3
* That is why **a \* b** simply does **element-wise multiplication**

**Now what if we use multiplication operator on matrices created using numpy?**

* Let’s create 2 matrices A and B
* Each of **dimension**3×43×4

A = np.arange(12).reshape(3, 4)

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

B = np.arange(12).reshape(3, 4)

B

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

**Now, Will A \* B actually do Matrix Multiplication?**

Let’s see

A \* B

array([[ 0, 1, 4, 9],

[ 16, 25, 36, 49],

[ 64, 81, 100, 121]])

* It **did NOT do Matrix Multiplication**
* Rather, it **again did element-wise multiplication**

**Why?**

* Check dimensions of matrices A and B
* **Dimensions of both A and B are**3×43×4

**Does this allow Matrix Multiplication?**

**What is the requirement of dimensions of 2 matrices for Matrix Multiplication?**

* **Columns of A = Rows of B** (A **Must condition** for Matric Multiplication)
* **If A is**3×43×4**, B can be**4×34×3… or 4×(SomethingElse)4×(SomethingElse)

**What will be dimensions of resulting matrix?**

* Rows of A ×× Columns of B
* 3×33×3

**So, lets reshape B to**4×34×3**instead**

B = B.reshape(4, 3)

B

array([[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8],

[ 9, 10, 11]])

**Will now the A \* B give Matrix Multiplication?**

Let’s see

A \* B

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-80-a4cedde81ed0> in <module>

----> 1 A \* B

ValueError: operands could not be broadcast together with shapes (3,4) (4,3)

**Uh oh! We got an error**

* Turns out **\* operator only does Element-Wise Multiplication of 2 Matrices**

**For actual Matrix Multiplication, We have a different method/operator**

np.matmul()

np.matmul(A, B)

array([[ 42, 48, 54],

[114, 136, 158],

[186, 224, 262]])

* **We are getting a**3×33×3**matrix as output**
* So, this is doing Matrix Multiplication

**There’s a direct operator as well for Matrix Multiplication**

@

A @ B

array([[ 42, 48, 54],

[114, 136, 158],

[186, 224, 262]])

**What will be the dimensions of Matrix Multiplication B @ A?**

* 4×44×4

B @ A

array([[ 20, 23, 26, 29],

[ 56, 68, 80, 92],

[ 92, 113, 134, 155],

[128, 158, 188, 218]])

**There is another method in np for doing Matrix Multiplication**

* np.dot()
* It is **NOT the dot product** that we have **in Physics**
* Dot product in Physics is just the element-wise multiplication
* **np.dot() performs the actual Matrix Multiplication**

np.dot(A, B)

array([[ 42, 48, 54],

[114, 136, 158],

[186, 224, 262]])

**Now, Let’s try multiplication of a mix of matrices and vectors**

* So that we are clear on Matrix Multiplication

A = np.arange(12).reshape(3, 4) # A is a 3x4 Matrix

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

a = np.array([1, 2, 3]) # a is a 1x3 Vector

a

array([1, 2, 3])

**Will A \* a work?**

A \* a

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-87-920aa4e58700> in <module>

----> 1 A \* a

ValueError: operands could not be broadcast together with shapes (3,4) (3,)

**Will a \* A work?**

a \* A

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-88-d157f8e14faf> in <module>

----> 1 a \* A

ValueError: operands could not be broadcast together with shapes (3,) (3,4)

**Why does it not work for either cases?**

* Because **\* operator just performs element-wise multiplication**
* For this, **both A and a should have same shape**

**However, A \* A and a \* a work**

A \* A # Multiplied with itself

array([[ 0, 1, 4, 9],

[ 16, 25, 36, 49],

[ 64, 81, 100, 121]])

a \* a

array([1, 4, 9])

**Now Let’s experiment with np.matmul()?**

**Will this work?**

np.matmul(A, a)

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-93-76efef6bd8e9> in <module>

----> 1 np.matmul(A, a)

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 4)

* NO
* A is 3×43×4 and a is 1×31×3
* **Columns of A**≠≠**Rows of a**

**Will this work?**

np.matmul(a, A)

array([32, 38, 44, 50])

* YES
* **Columns of a (3) = Rows of A (3)**
* Dimensions of result is 1×41×4

**Same applies to operator @ and method np.dot()**

* A @ a will NOT work
* a @ A will work
* np.dot(A, a) will NOT work
* np.dot(a, A) will work

**Conclusion:**

* np.matmul(a, A), a @ A and np.dot(a, A) work
* Because **a is implicitly assumed to be a**1×31×3**matrix**
* So, the dimensions of a and A follow the rule for Matrix Maultiplication

**Transpose**

* **Change rows into columns and columns into rows**
* Just use <Matrix>.T

a = np.arange(3)

a

array([0, 1, 2])

a.T

array([0, 1, 2])

**Why did Transpose did not work?**

* Because **numpy sees a as a vector (3,), NOT a matrix**
* We’ll have to **reshape the vector a to make it a matrix**

a = np.arange(3).reshape(1, 3)

a

# Now a has dimensions (1, 3) instead of just (3,)

# It has 1 row and 3 columns

array([[0, 1, 2]])

a.T

# It has 3 rows and 1 column

array([[0],

[1],

[2]])

* Now the transpose is working :)

**Conclusion:**

* **Transpose works only on matrices**
* It has **no meaning for a vector**

**Converting Matrix back to a Vector**

**How can we convert our**3×43×4**matrix A back to a vector?**

* We can use flaten()
* We can use ravel()
* We can even use reshape() again

**Let’s take our**3×43×4**matrix A**

A = np.arange(12).reshape(3, 4)

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

A.flatten()

# Gives 1D vector

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

A.ravel()

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

**What should we pass in A.reshape() if I want to use it to convert A to 1D vector?**

* **(1, 1)?** - **NO**
* It means we only have a single element
* But **we don’t have a single element**

A.reshape(1, 1)

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-101-902e5c35e0d3> in <module>

----> 1 A.reshape(1, 1)

ValueError: cannot reshape array of size 12 into shape (1,1)

* So, **(1, 12)?** - **NO**
* It will **still remain a 2D Matrix with dimensions**1×121×12

A.reshape(1, 12)

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

* **Correct answer is (12)**
* We need a vector of dimension (12,)
* So we need to pass only 1 dimension in reshape()

A.reshape(12)

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

**So, Be careful while using reshape() to convert a Matrix into a 1D vector**

**What will happen if we pass a negative integer in reshape()?**

A.reshape(6, -1)

array([[ 0, 1],

[ 2, 3],

[ 4, 5],

[ 6, 7],

[ 8, 9],

[10, 11]])

**Surprisingly, it did not give an error**

* It is able to **figure out on its own** what should be the **value in-place of negative integer**
* Since **no. of elements in our matrix is 12**
* And **we passed 6 as no. of rows**
* It is **able to figure out** that **no. of columns should be 2**

**Same thinhg happens with this:**

A.reshape(-1, 6)

array([[ 0, 1, 2, 3, 4, 5],

[ 6, 7, 8, 9, 10, 11]])

**Merging Arrays**

* We have up to now looked at methods to rearrange arrays.
* We also looked at how to extract subarrays using various indexing techniques.
* In addition to reshaping and selecting subarrays, it is often necessary to merge arrays into bigger arrays,
* **For example**, when joining separately computed or measured data series into a **higher-dimensional array**, such as a matrix.

**Numpy provides certain functions for the task of stacking/merging of arrays together**

**np.vstack()**

* Stacks a list of arrays vertically (along axis 0)
* For **example**, **given a list of row vectors, appends the rows to form a matrix**.

data = np.arange(5)

data

array([0, 1, 2, 3, 4])

np.vstack((data, data, data))

array([[0, 1, 2, 3, 4],

[0, 1, 2, 3, 4],

[0, 1, 2, 3, 4]])

**np.hstack()**

* Stacks a list of arrays horizontally (along axis 1)
* For **example**, **given a list of column vectors, appends the columns to form a matrix**.

data = np.arange(5)

data

array([0, 1, 2, 3, 4])

np.hstack((data, data, data))

array([0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4])

**np.concatenate()**

* Creates a new array by appending arrays after each other, along a given axis
* Provides similar functionality, but it takes a **keyword argument axis** that specifies the **axis along which the arrays are to be concatenated**.

z = np.array([[2, 4]])

z

array([[2, 4]])

zz = np.concatenate([z, z], axis=0)

zz

array([[2, 4],

[2, 4]])

zz = np.concatenate([z, z], axis=1)

zz

array([[2, 4, 2, 4]])

**Sorting Arrays**

* We can sort the elements of an array along a given specified axis
* Default axis is the last axis of the array.

**np.sort()**

* Returns a **sorted copy of an array**.

a = np.array([2,30,41,7,17,52])

a

array([ 2, 30, 41, 7, 17, 52])

np.sort(a)

array([ 2, 7, 17, 30, 41, 52])

a

array([ 2, 30, 41, 7, 17, 52])

* Original array is still the same. It hasn’t changed

**np.ndarray.sort()**

* The np.ndarray.sort() method **performs the sorting in place, modifying the input array**.
* It **changes the orginal array**

a = np.array([2,30,41,7,17,52])

a

array([ 2, 30, 41, 7, 17, 52])

np.ndarray.sort(a)

a

array([ 2, 7, 17, 30, 41, 52])

**np.argsort()**

* Returns the **indices** that would sort an array.
* Performs an indirect sort along the given axis.
* It returns **an array of indices of the same shape as a that index data along the given axis in sorted order**.

a = np.array([2,30,41,7,17,52])

a

array([ 2, 30, 41, 7, 17, 52])

np.argsort(a)

array([0, 3, 4, 1, 2, 5])

**As you can see:**

* The orginal indices of elements are in same order as the orginal elements would be in sorted order

**Broadcasting**

**Do you remember the error thrown when a \* A was not possible?**

* A term called **broadcast** was coming up

**Let’s create 2 matrices and see what happens when we operate on them**

A = np.arange(12).reshape(3, 4) # 12 values, shape 3x4

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

B = np.arange(3).reshape(1, 3) # 2 values, shape 1x3

B

array([[0, 1, 2]])

# Let's check their shapes

print(A.shape, B.shape)

(3, 4) (1, 3)

**What would happen if I add A and B?**

A + B

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-122-151064de832d> in <module>

----> 1 A + B

ValueError: operands could not be broadcast together with shapes (3,4) (1,3)

**So, What is going on here?**

**What is broadcast?**

* **Length of row** in **B is 3**
* **Length of each row** in **A is 4**
* It trying to add rows of different lengths, but it is not able to

**Let’s change the shape of A to**3×33×3**and see what happens**

A = np.arange(9).reshape(3, 3) # 9 values, shape 3x3

A

array([[0, 1, 2],

[3, 4, 5],

[6, 7, 8]])

B = np.arange(3).reshape(1, 3) # 2 values, shape 1x3

B

array([[0, 1, 2]])

# Let's check their shapes

print(A.shape, B.shape)

(3, 3) (1, 3)

A + B

array([[ 0, 2, 4],

[ 3, 5, 7],

[ 6, 8, 10]])

**Now it is able to add A and B**

* It **added the row of B to each row of A**
* Such operations on np arrays **require a part of the shape to match**
  + Either **no. of rows or columns**
* So that **smaller array can be operated on larger array element-wise**

**This is called “Broadcasting”**

* Broadcasting **lets two arrays of different shapes to do some operations.**
* The **smaller array** will **repeat itself**,
* and get **converted to the same shape as of larger array**.

**So, smaller array gets boradcasted over larger array again and again**

* Let’s check it out again using a different shape for A and B

A = np.arange(12).reshape(4, 3) # 12 values, shape 4x3

A

array([[ 0, 1, 2],

[ 3, 4, 5],

[ 6, 7, 8],

[ 9, 10, 11]])

B = np.arange(3).reshape(1, 3) # 2 values, shape 1x3

B

array([[0, 1, 2]])

print(A.shape, B.shape)

(4, 3) (1, 3)

A + B

array([[ 0, 2, 4],

[ 3, 5, 7],

[ 6, 8, 10],

[ 9, 11, 13]])

* It works!!
* **Now B repeats itself 4 times** instead of 3 times when shape of A was (3, 3)

**We can also use Transpose in some cases**

* To **match the shapes for broadcasting**

A = np.arange(12).reshape(3, 4)

A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11]])

B = np.arange(3).reshape(1, 3)

B.T

array([[0],

[1],

[2]])

A + B.T

array([[ 0, 1, 2, 3],

[ 5, 6, 7, 8],

[10, 11, 12, 13]])

Last changed by

**Scaler Topics**Scaler Topics

 454 [4](https://hackmd.io/@scaler-topics-main/HkH5nw8x9)

Published on [**HackMD**](https://hackmd.io/)