In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

sp = lambda: print("\n\n")
```

numpy Array Attributes

A Numpy array is a row/grid of values, all of the same type, and is indexed by a tuple of non-negative integers.

Each array has the following attributes:

- dtype-the type of data in the array
- ndim-the number of dimensions,
- shape-the size of each dimension, and
- size-the total number of elements in the array

In [2]:
```
#array with 5*5 nums
arr = np.random.rand(5,5)
arr
```

Out[2]:
```
array([[0.26726532, 0.8956987 , 0.83170404, 0.15760709, 0.81149715],
       [0.63294665, 0.57996088, 0.39470832, 0.82151311, 0.90487327],
       [0.72930083, 0.60799535, 0.55765241, 0.6905255 , 0.0586852 ],
       [0.47593344, 0.070155  , 0.47354844, 0.35414013, 0.09322596],
       [0.05291787, 0.94358413, 0.65464733, 0.94278687, 0.49237949]])
```

In [6]:
```
print(f"type of array:{arr.dtype}")
print(f"size of array:{arr.size}")
print(f"shape of array:{arr.shape}")
print(f"number of dimensions:{arr.ndim}")
#
```

```
type of array:float64
size of array:25
shape of array:(5, 5)
number of dimensions:2
```

Functions to Create ndarrays in NumPy

NumPy provides several functions to create arrays (ndarrays). The most commonly used ones are:

- array()

Creates an array from a given sequence (list, tuple, etc.).

- arange()

Creates an array with a specified range of values (similar to Python's range).

- linspace()

Creates an array with n linearly spaced values between two endpoints.

- empty()

Creates an array of a given shape without initializing its values (values are arbitrary / garbage).

- zeros()

Creates an array of a given shape filled with zeros.

- ones()

Creates an array of a given shape filled with ones.

- rand() / randint()

Creates arrays filled with randomly generated values.

rand() → random floats

randint() → random integers

In [14]:
```
#array()
arr = np.array([1,2,3,4,5])
print("array(): ",arr)
sp()
#arange()
arr = np.arange(1,10)
print("arange(): ",arr)
sp()

#linspace()
arr = np.linspace(1,10,5)
print("linspace(): ",arr)
sp()

#empty()
arr = np.empty((1,10))
print("empty(): ",arr)
sp()

#zero
arr = np.zeros((1,10))
print("zeros(): ",arr)
sp()

#ones()
arr = np.ones((1,10))
```

```
print("ones(): ",arr)
sp()

#rand()/randint()
arr = np.random.rand(1,10)
print("rand(): ",arr)
sp()
arr = np.random.randint(1,10,5)
print("randint(): ",arr)
sp()
```

```
array():  [1 2 3 4 5]


arange():  [1 2 3 4 5 6 7 8 9]


linspace():  [ 1.    3.25  5.5   7.75 10.  ]


empty():  [[2.66351057e-315 0.00000000e+000 0.00000000e+000 0.00000000e+000
  0.00000000e+000 1.77668524e+160 3.88601149e-033 5.58649171e-091
  1.71763389e+185 4.57690700e-315]]


zeros():  [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]


ones():  [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]


rand():  [[0.69144138 0.2482599  0.29730229 0.3429867  0.19769871 0.83917906
  0.03134791 0.58099028 0.8922931  0.8114883 ]]


randint():  [6 3 3 6 3]
```

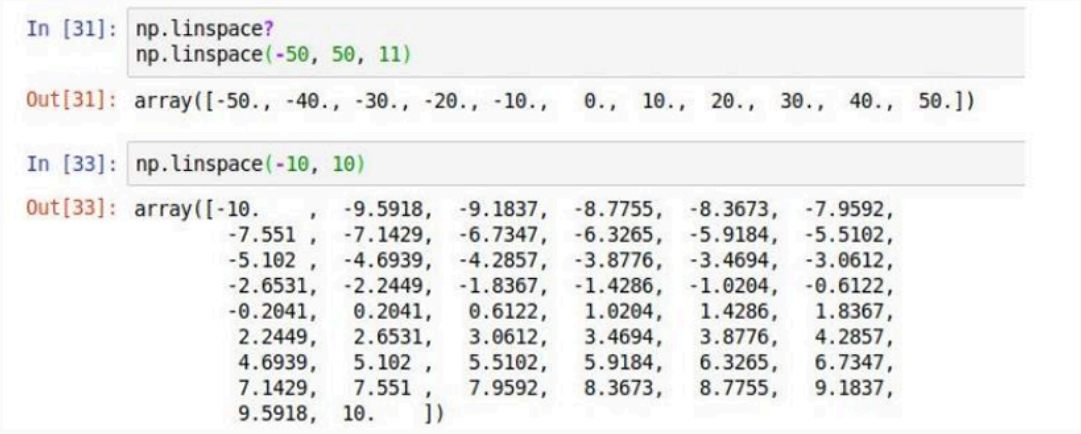
linespace()

linspace()

Function creates evenly spaced float values.

Takes start, end and number of values as parameters.

Generates the given number of values between the specified interval.



```
In [31]: np.linspace?
         np.linspace(-50, 50, 11)

Out[31]: array([-50., -40., -30., -20., -10.,   0.,  10.,  20.,  30.,  40.,  50.])

In [33]: np.linspace(-10, 10)

Out[33]: array([-10.    ,  -9.5918,  -9.1837,  -8.7755,  -8.3673,  -7.9592,
                -7.551 ,  -7.1429,  -6.7347,  -6.3265,  -5.9184,  -5.5102,
                -5.102 ,  -4.6939,  -4.2857,  -3.8776,  -3.4694,  -3.0612,
                -2.6531,  -2.2449,  -1.8367,  -1.4286,  -1.0204,  -0.6122,
                -0.2041,   0.2041,   0.6122,   1.0204,   1.4286,   1.8367,
                 2.2449,   2.6531,   3.0612,   3.4694,   3.8776,   4.2857,
                 4.6939,   5.102 ,   5.5102,   5.9184,   6.3265,   6.7347,
                 7.1429,   7.551 ,   7.9592,   8.3673,   8.7755,   9.1837,
                 9.5918,  10.    ])
```

Generating Random Arrays in NumPy

NumPy provides a sub-module called random, which contains functions used to generate random arrays.

- rand(n)

Generates an array of n random floating-point values.

Values are uniformly distributed between 0 and 1.

- randn(n)

Generates an array of n random floating-point values.

Values are drawn from a standard normal distribution (mean = 0, std = 1).

Unlike rand(), values can be negative or greater than 1.

- randint(low, high, size)

Generates an array of random integer values.

Range:

low → inclusive

high → exclusive

size specifies the number (or shape) of generated values.

Function	Distribution	Value Range	Type	Notes
`rand(n)`	Uniform	`0 ≤ x < 1`	Float	Values are evenly spread, **no negatives**
`randn(n)`	Standard Normal	Any real number (mostly -3 to 3)	Float	Values follow a **bell curve**, can be **negative or >1**
`randint(low, high, size)`	Discrete Uniform	`low ≤ x < high`	Integer	Generates random integers, **size** defines shape

In [6]:
```
r_vals = np.random.rand(10)
print("rand(10):\n",r_vals)
sp()
r_vals = np.random.randn(10)
print("randn(10):\n",r_vals)
sp()
r_vals = np.random.randint(1,26,12) #parameter (low,high,size)
print("randint:\n",r_vals)
```

```
rand(10):
 [0.09481387 0.61594601 0.83387542 0.3058312  0.02775456 0.21774403
 0.97487522 0.55435896 0.03848953 0.60886411]


randn(10):
 [ 0.29152037 -1.54270093 -0.49851238 -0.6087686   1.21489849  0.56396883
  0.52231228  0.11061759  1.10801371 -0.86469143]


randint:
 [ 2  5 21  7  9 18  8 19  1  8  5  4]
```

Creating Two-Dimensional Arrays

Two dimensional numpy arrays are just arrays of arrays.

We created the array using the array command and passing a list as a parameter, now we can pass a list of lists.

Example:

```
m = np.array([[3,5,2,4],[7,6,8,8],[1,6,7,7]])
m
Out[35]:
array([ [3, 5, 2, 4],
        [7, 6, 8, 8],
        [1, 6, 7, 7]])
```

Creating 2-D arrays: zeros(), ones()

```
In [1]: import numpy as np

In [3]: np.zeros((3,8))
Out[3]:
array([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])


In [5]: np.ones((3,8))
Out[5]:
array([[1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])


In [6]: np.ones((2,3,2))
Out[6]:
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

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

Slicing NumPy Arrays

Just like Python lists, we can access parts of a NumPy array using the following syntax:

start : stop : step

- start → index to start (included)
- stop → index to stop (excluded)
- step → step size (can be negative!)

Remember: the stop value is not included (up to, but not including).

If any of the three values are omitted, the default values are:

start = 0

stop = end of the array

step = 1

This allows you to easily select any part of the array without deleting or creating a new one.

Examples:

In [7]:
```
z  = np.array([4,7,5,9,2])
z
```

Out[7]:
```
array([4, 7, 5, 9, 2])
```

In [11]:
```
z[1:4:2] #start end step
#note: when say end = 4 mean when arrive index 4 will stop (not take it and stop)
```

Out[11]:
```
array([7, 9])
```

In [10]:
```
z[1::2] #start:1 end:default(end of array) step:1
```

Out[10]:
```
array([7, 9])
```

In [12]:
```
z[:4] #start:default(0) end:4 step:default(1)
```

Out[12]:
```
array([4, 7, 5, 9])
```

Examples with Negative Indices and Negative Step

np.array([4,7,5,9,2])

In [16]:
```
z[-1] #access last element
```

Out[16]:
```
np.int64(2)
```

In [23]:
```
z[4:0:-1] #slice from last to index 2 (step = -1)
```

Out[23]:
```
array([2, 9, 5, 7])
```

In [25]:
```
z[::-1] # Reverse the entire array
```

Out[25]:
```
array([2, 9, 5, 7, 4])
```

In [26]:
```
z[3::-2] # Slice from index 3 backward with step -2
```

Out[26]:
```
array([9, 7])
```

In [29]:
```
# hard
# Slice from last to 4th from end
z[-1:-4:-1]
```

Out[29]:
```
array([2, 9, 5])
```

Slicing 2D NumPy Arrays

A 2D array has rows and columns:

`array[row_start:row_stop:row_step, col_start:col_stop:col_step]`

- First slice → rows
- Second slice → columns

- Step can be positive or negative

Negative indices work the same as in 1D arrays

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

In [7]:
```
#array[row_start:row_stop:row_step, col_start:col_stop:col_step]
arr[:2, :]
```

Out[7]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [9]:
```
# if i want to get [2,4],[6,8]
arr[:2 ,1::2 ]
```

Out[9]:
```
array([[2, 4],
       [6, 8]])
```

In [10]:
```
arr[:, :3]
```

Out[10]:
```
array([[ 1,  2,  3],
       [ 5,  6,  7],
       [ 9, 10, 11]])
```

In [14]:
```
arr[::-1, ::-1] #reverse
```

Out[14]:
```
array([[12, 11, 10,  9],
       [ 8,  7,  6,  5],
       [ 4,  3,  2,  1]])
```

Accessing Elements: Two D Arrays

Syntax: `python arrName[ row, col]` Examples:

In [17]:
```
m = np.linspace(1,10,9).reshape(3,3)
m
```

Out[17]:
```
array([[ 1.   ,  2.125,  3.25 ],
       [ 4.375,  5.5  ,  6.625],
       [ 7.75 ,  8.875, 10.   ]])
```

In [18]:
```
#accessing elements

#method 1
print(m[1][2])

#method 2
print(m[1,2])
```

```
6.625
6.625
```

In [20]:
```
#look at this

m[2,]
```

Out[20]:
```
array([ 7.75 ,  8.875, 10.   ])
```

Element-Wise Operations

Unlike lists, arrays allow us to perform element by element operations without using a loop.

When we apply arithmetic operations to arrays, the operation will be applied to each element of the array.

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

example `python # Element-wise operation array = np.array([1, 2, 3, 4]) result = array * 2 # [2, 4, 6, 8] # No loop needed!`

1. Element-wise Operations (عمليات على كل عنصر)

العملية بتتطبق على كل عنصر لوحده، array ده لما تعمل عملية حسابية على.

```
python
import numpy as np

# عندك array
numbers = np.array([1, 2, 3, 4])

# لو ضربت في 2
result = numbers * 2
```

```
# [8 ,6 ,4 ,2] :النتيجة
# كل رقم اتضرب في 2 لوحده
```

مثال تاني:

```
python

array1 = np.array([10, 20, 30])
array2 = np.array([1, 2, 3])

# جمع
result = array1 + array2
# [33 ,22 ,11] :النتيجة
# 10+1, 20+2, 30+3
```

يعني عنصر بعنصر - أول عنصر مع أول عنصر، تاني مع تاني، وهكذا.

2. Vectorization (التحويل لعمليات متجهة)

ده المفهوم الأكبر! بدل ما تكتب loop:

```
python# الطريقة البطيئة (loop عادي)
result = []
for i in range(len(numbers)):
    result.append(numbers[i] * 2)
```

تستخدم vectorization:

```
python# الطريقة السريعة (vectorized)
result = numbers * 2
```

يلف Python تحت الكابوت. بدل ما (سريعة جداً) C بيستخدم كود مكتوب بلغة NumPy ليه أسرع؟ لأن بيعمل العملية دفعة واحدة بسرعة رهيبة! مثال على الفرق في NumPy، على كل عنصر واحد واحد ببطء السرعة:

```
python

import numpy as np
import time

# array كبير
big_array = np.arange(1000000)  # مليون رقم!

# 1 طريقة: Loop (بطيئة)
start = time.time()
result = [x * 2 for x in big_array]
print(f"Loop: {time.time() - start} ثانية")

# 2 طريقة: Vectorized (سريعة)
```

```
start = time.time()
result = big_array * 2
print(f"Vectorized: {time.time() - start} ثانية")

# Vectorized مرة 100-50 أسرع! 🚀
```

3. Broadcasting (البث/التوسيع التلقائي)

بيكبّر الأصغر تلقائياً NumPy، بأحجام مختلفة arrays لما يكون عندك NumPy! دي حاجة ذكية جداً في عشان يقدر يعمل العملية. مثال بسيط:

```
python# array (,3) بحجم
array = np.array([1, 2, 3])

# رقم واحد (scalar)
number = 10

# لما تجمعهم
result = array + number
# النتيجة: [13 ,12 ,11]
```

python# للرقم 10 وخلاه كأنه [10 ,10 ,10] تلقائياً broadcasting عمل NumPy إيه اللي حصل؟ فهم الموضوع كأنه NumPy: array + np.array([10, 10, 10]) مثال أعقد (2D):

```
python# 3×3 مصفوفة
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# array (3×1) واحد بس
row = np.array([10, 20, 30])

# جمع
result = matrix + row

**النتيجة:**
```

لكل صف في المصفوفة! [10, 20, 30] row كرر الـ NumPy [[11, 22, 33], [14, 25, 36], [17, 28, 39]] قواعد 🎯 Broadcasting:

نفس الحجم → تمام، اشتغل عادي واحد فيهم حجمه 1 → كبّره عشان يساوي التاني أحجام مختلفة تماماً → Error! ❌

```
python# ✅ شغال
np.array([1, 2, 3]) + np.array([10, 20, 30])  # (3,) + (3,)

# ✅ شغال (broadcasting)
np.array([1, 2, 3]) + 5  # (3,) + scalar
```

```
# ✅ شغال (broadcasting)
np.array([[1, 2, 3]]) + np.array([10, 20, 30])  # (1,3) + (3,)

# ❌ مش شغال
np.array([1, 2, 3]) + np.array([1, 2])  # (3,) + (2,) → Error!
```

الخلاصة: Element-wise (أول مع أول، تاني مع تاني) العملية بتحصل عنصر بعنصر Vectorization بيكبّر الـ Broadcasting NumPy! أسرع بكتير → loops كله دفعة واحدة بدون array تشتغل على الـ arrays الصغيرة تلقائياً عشان تناسب الكبيرة

مثال يجمع الكل: python# (3 طلاب × 4 مواد) عندك مصفوفة درجات الطلبة grades = np.array([[80, 70, 90, 85], [75, 85, 80, 90], [90, 95, 85, 88]])

عايز تزود 5 درجات لكل الطلبة

bonus = 5

Broadcasting + Vectorization + Element-wise

new_grades = grades + bonus

كل طالب في كل مادة زادله 5 درجات

بسطر واحد بس! 🎉

Numpy Arrays – built in functions

Numpy arrays have many built in functions for computing statistics on an array.
See: 10_numpyArrayFunctions.py

Function Name	Description
np.sum()	Computes the sum of values in given array.
np.mean()	Computes the mean(average) value in given array.
np.max()	Computes the maximum value in given array.
np.min()	Computes the minimum value in given array.
np.argmax()	Computes the index of the maximum value in given array.
np.argmin()	Computes the index of the minimum value in given array.

numpy Arrays – Matrix Operations

numpy knows how to efficiently do typical matrix operations

Examples:

In [40]:
```
np.random.seed(1) #to always get same random number

a = np.arange(1,10).reshape(3,3)
b = np.random.randint(1,10,9).reshape(3,3)

print("mat a: \n",a)
sp()
print("mat b: \n",b)

#if we use normal multiply
print("\n use *:")
print(a*b)


#matrix vector product
print("\na matrix-vector product using np.dot: ")
print(np.dot(a,b))
```

```
mat a:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]



mat b:
 [[6 9 6]
 [1 1 2]
 [8 7 3]]

 use *:
[[ 6 18 18]
 [ 4  5 12]
 [56 56 27]]

a matrix-vector product using np.dot:
[[ 32  32  19]
 [ 77  83  52]
 [122 134  85]]
```

numpy Arrays – Concatenation

It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays.

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines `np.concatenate`, `np.vstack`, and `np.hstack`.

`np.concatenate()` - takes a tuple or list of arrays as its first argument, returns a new array.

```
import numpy as np

x = np.array([1,2,3])
y = np.array([3,2,1])
print(np.concatenate([x,y]))
```

Output:

```
[1 2 3 3 2 1]
```

numpy Arrays – Concatenation

`np.concatenate()` – can be used to concatenate more than two arrays or two-dimensional arrays as well.

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

Output:

```
[ 1  2  3  3  2  1 99 99 99]
```

```
grid = np.array([[1, 2, 3],[4, 5, 6]])
print(np.concatenate([grid, grid])) #default axis = 0
```

```
[[1 2 3]
 [4 5 6]
 [1 2 3]
 [4 5 6]]
```

```
grid = np.array([[1, 2, 3],[4, 5, 6]])
print(np.concatenate([grid, grid], axis = 1))
```

```
[[1 2 3 1 2 3]
 [4 5 6 4 5 6]]
```

numpy – hstack, vstack

The *concatenate* function can be used when the arrays you are joining have the same number of dimensions (one or two). If you want to join arrays with different dimensions, you should use the functions **vstack** or **hstack**.

vstack: stack two or more arrays vertically and returns a new array

hstack: stack two or more arrays horizontally and returns a new array.

In [52]:
```
array1 = np.array([1, 2, 3])

array2 = np.array([4, 5, 6])

result_conc = np.concatenate([array1, array2])
result_vstack = np.vstack([array1, array2])
result_hstack = np.hstack([array1, array2])

print("\n concatinate result: \n",result_conc)
```

```
print("\n vstack result: \n",result_vstack)
print("\n vstack result:\n",result_hstack)
```

```
concatinate result:
[1 2 3 4 5 6]

vstack result:
[[1 2 3]
[4 5 6]]

vstack result:
[1 2 3 4 5 6]
```

In [57]:
```
a = np.array([10, 20, 30])
b = np.array([40, 50, 60])
c = np.array([70, 80, 90])

result0 = np.concatenate([a, b, c])
result1 = np.vstack([a, b, c])
result2 = np.hstack([a, b, c])

print(result0)
sp()
print(result1)
sp()
print(result2)
```

```
[10 20 30 40 50 60 70 80 90]



[[10 20 30]
 [40 50 60]
 [70 80 90]]



[10 20 30 40 50 60 70 80 90]
```

حاسك توهت بس معنديش دليل

بص معايا على المثال دا كدا

In [65]:
```
x = np.array([[1, 2],
              [3, 4]])

y = np.array([[5, 6],
              [7, 8]])

print(np.concatenate([x, y], axis=0)) #دمج رأسي
sp()
print(np.concatenate((x, y), axis=1)) #دمج افقي
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

```
[[1 2 5 6]
 [3 4 7 8]]
```

بمعنى

```
np.concatenate((a, b), axis=0) شبه np.vstack([a,b])
np.concatenate((a, b), axis=1) شبه np.hstack([a,b])
```

شبه بس مش زي بعض تماما وهتعرف ليه

مقارنة سريعة جدًا

Function	الدمج	يعمل مع	الناتج
`concatenate`	حسب axis	نفس الأبعاد فقط	مرن
`vstack`	رأسي (rows)	1D أو 2D	غالبًا 2D
`hstack`	أفقي (columns)	1D أو 2D	1D أو 2D

The concatenate function can be used when the arrays you are joining have the same number of dimensions (one or two)

يعني اي الكلام دا ؟

اللي هتجمعهم يكون عندهم نفس عدد الأبعاد arrays لازم كل الـ (dimensions). this in concatenate function

خليني ابينلك الفرقات بينهم بأمثله

In [96]:
```python
import numpy as np

def sp():
    print("-" * 40)

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

# ❓ يعني إيه axis هنا؟

# axis = 0 → البعد الوحيد الموجود
```

```python
# axis = 1 → ❌ غير موجود أصلاً

# 📌
# 1 لـ 1D array مفيهاش axis=1


print("concatenate بدون axis:")
print(np.concatenate([a, b]))  # worked (axis = 0)

# الافتراضي:
# axis = 0

sp()

print("hstack:")
print(np.hstack([a, b]))  # worked
sp()


print("concatenate مع axis=1:")
try:
    print(np.concatenate([a, b], axis=1))
except Exception as e:
    print(f"❌ Error: {e}")

sp()

print("vstack:")
print(np.vstack([a, b]))  # worked
sp()
```

```
concatenate بدون axis:
[1 2 3 4 5 6]
----------------------------------------
hstack:
[1 2 3 4 5 6]
----------------------------------------
concatenate مع axis=1:
❌ Error: axis 1 is out of bounds for array of dimension 1
----------------------------------------
vstack:
[[1 2 3]
 [4 5 6]]
----------------------------------------
```

🔹 الحالة عندك

```python
a = np.array([1, 2, 3])   # shape = (3,)
b = np.array([4, 5, 6])   # shape = (3,)
```

1 دي D arrays.

❌ ليه ده مش شغال؟

```python
np.concatenate([a, b], axis=1)
```

السبب الحقيقي

- `concatenate` **صارم جدًا**
- array يكون موجود فعليًا في الـ `axis` لازم الـ
- 1D array عنده:

  `a.ndim == 1`
- يعني:
  - المحاور المتاحة: `axis = 0` فقط
  - `axis = 1` ❌ غير موجود

📌 فـ Python يقولك:

> علشان أدمج عليه axis=1 مفيش

✅ ليه `vstack` اشتغل؟

```
np.vstack([a, b])
```

اللي بيحصل تحت الكواليس 🧠

`vstack` **مش بيشتغل مباشرة على 1D** هو بيعمل خطوة ذكية الأول 👇

1️⃣ يحوّل كل 1D array إلى 2D

```
a → [[1, 2, 3]]
b → [[4, 5, 6]]
```

دلوقتي:

```
shape = (1, 3)
```

2️⃣ بعد كده يعمل:

```
concatenate(axis=0)
```

✔️ ده محور موجود ✔️ فالكود يشتغل

🔍 طيب ليه `concatenate` ما يعملش كده؟

لأن:

- `concatenate` دالة **Low-level**
- بتفترض إنك فاهم الأبعاد كويس
- مش مسؤولة تصلّح أبعادك

لكن:

- `vstack` و `hstack` دوال **High-level helpers**

🧠 تشبيه بسيط

تخيل:

- `concatenate` = موظف صارم

  > "هات الحاجة مظبوطة، أنا مش هظبطها"

- `vstack` = موظف مساعد

  > "ولا يهمك، أظبطهالك وبعدين أشتغل"

😄

🧪 إثبات بالكود

`a.reshape(1, -1)`
بيعمله داخليًا `vstack` ده بالظبط اللي.

✨ الخلاصة الذهبية

Function	بيحوّل الأبعاد؟	ينفع مع axis=1 1 في D؟
concatenate	❌ لا	❌ لا
vstack	✅ آه	✔
hstack	✅ آه	✔

In [82]:
```
#2D

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

y = np.array([[5, 6],
              [7, 8]])

print( np.concatenate([x, y]) ) #axis = 0
sp()
print(np.concatenate((x, y), axis=0) == np.vstack([x,y]))
```

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

```
[[ True  True]
 [ True  True]
 [ True  True]
 [ True  True]]
```

🧠 الخلاصة

1 معD arrays:

Function بيعمل إيه فعليًا

concatenate() axis=0 فقط

hstack() زي concatenate(axis=0)

vstack() 2 يحوّل لـD ثم axis=0

2 معD arrays:

Function Equivalent

vstack concatenate(axis=0)

hstack concatenate(axis=1)

numpy Arrays – Splitting

The opposite of concatenation is splitting, which is implemented by the functions `np.split`, `np.hsplit`, and `np.vsplit`.

For each of these, we can pass a list of indices giving the split points.

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

```
grid = np.arange(16).reshape((4, 4))
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]

[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

In [ ]:

Sorting Arrays

Like lists, numpy arrays can be sorted in place using the `sort()` function.

```
In [67]: arr = np.random.rand(8)

In [68]: arr
Out[68]:
array([0.41857204, 0.52282439, 0.85841995, 0.97523084, 0.52313993,
       0.70206584, 0.37557734, 0.29295251])

In [69]: arr.sort()

In [70]: arr
Out[70]:
array([0.29295251, 0.37557734, 0.41857204, 0.52282439, 0.52313993,
       0.70206584, 0.85841995, 0.97523084])
```

In [ ]:

Unique and Set Logic

For one dimensional arrays, numpy has some basic set operations.

The most used is the `unique()` function, which returns the sorted unique values in an array.

```
In [72]: names = np.array(['Bob','Joe','Will','Bob','Will','Joe','Joe'])

In [73]: np.unique(names)
Out[73]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [74]: ints = np.array([3,3,3,2,2,1,1,4,4])

In [75]: np.unique(ints)
Out[75]: array([1, 2, 3, 4])
```

In [ ]:

Boolean Arrays

A Boolean array is an array that contains True/False values.
A Boolean array may be generated in numpy by using element-wise relational expressions with arrays.

```
In [87]: average_rainfall = np.array([55.0,71.3,10.5,0.0,8.6,30.2,18.8])

In [88]: average_rainfall > 10
Out[88]: array([ True,  True,  True, False, False,  True,  True])

In [89]: (average_rainfall > 10) & (average_rainfall < 20)
Out[89]: array([False, False,  True, False, False, False,  True])
```

Python assumes that True values are 1 and False values are 0. If we `sum()` the elements of a Boolean array, the True values will be counted.

```
In [106]: np.sum(average_rainfall > 10)
Out[106]: 5
```

Boolean indexing and sub arrays

We can use Boolean arrays to select a subset of an array.

For example if we want to find the rainfall for all days of the week in which it was greater than 10, we can do so using logical indexing.

```
In [87]: average_rainfall = np.array([55.0,71.3,10.5,0.0,8.6,30.2,18.8])

In [88]: average_rainfall > 10
Out[88]: array([ True,  True,  True, False, False,  True,  True])

In [90]: over_10 = average_rainfall > 10

In [91]: average_rainfall[over_10]
Out[91]: array([55. , 71.3, 10.5, 30.2, 18.8])
```

where() function – array indexes

In the previous example we used the logical array to select the values/elements that met the given condition(s).

If we want to find the indexes of the elements that meet the given condition(s), we can do so using the where function.

The example below shows that the resulting values are the indexes of the elements in the array that meet the given condition. I.e. elements 0,1,2, etc have values over 10.

```
In [96]: average_rainfall = np.array([55.0,71.3,10.5,0.0,8.6,30.2,18.8])

In [97]: np.where(average_rainfall > 10)
Out[97]: (array([0, 1, 2, 5, 6], dtype=int64),)
```

Subarrays and functions

```
import numpy as np
x = np.array([22,41,58,33,17,32])
#returns array of bool values for each element in x where condition is true
r1 = x > 40
print(r1)

#returns array containing elements in x where condition is true
r2 = x[x > 40]
print(r2)

#returns array containing indices of elements in x where condition is true
r3 = np.where(x > 40)
print(r3)
```

In [ ]:

the end