AI VIET NAM
@aivietnam.edu.vn

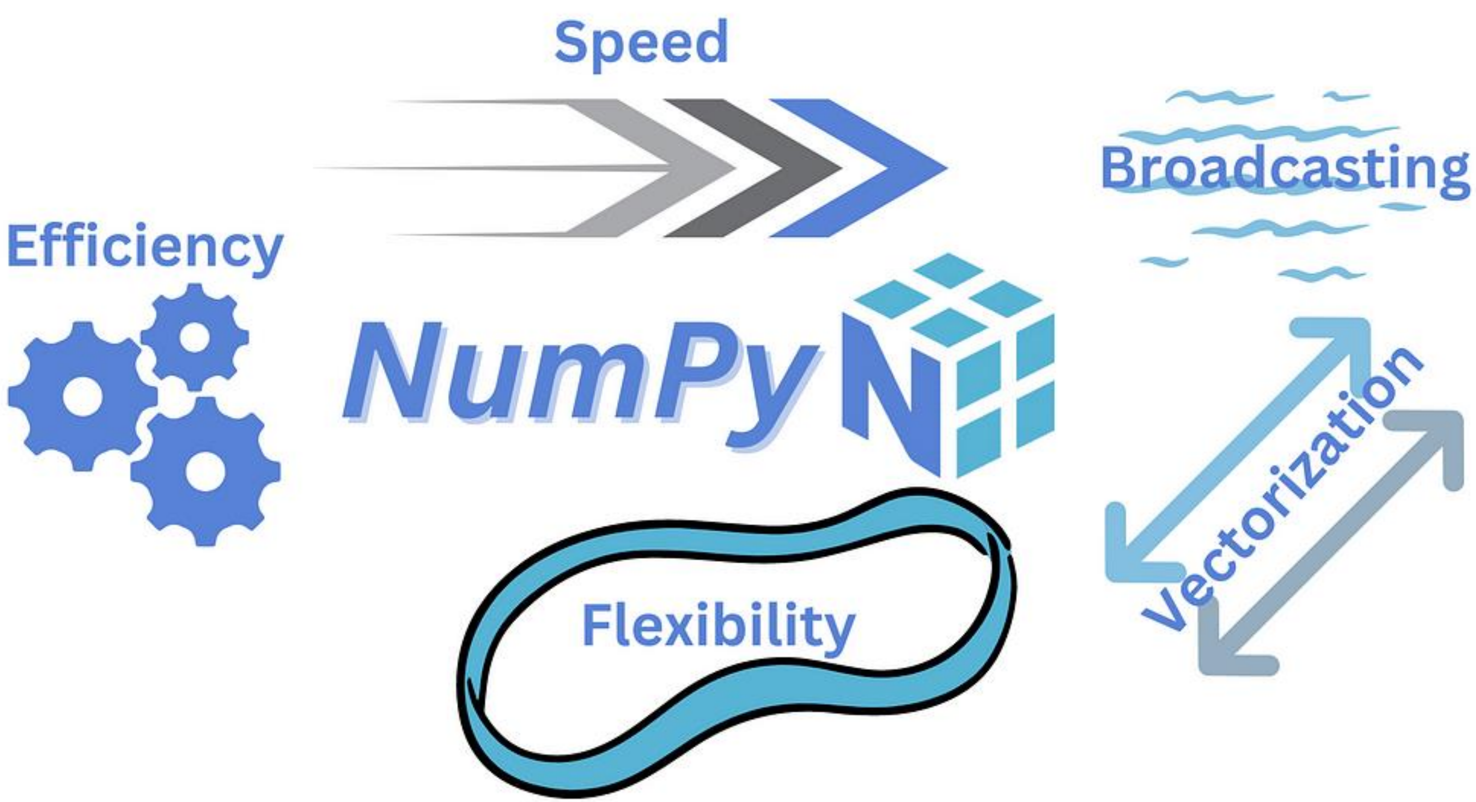


Image taken from Stackademic

NumPy Arrays

Vinh Dinh Nguyen
PhD in Computer Science

Quang-Du Nguyen
PhD in Construction Engineering
(Thesis on AI in Construction Engineering)

Year 2025

[Code&data]

AI VIET NAM
@aivietnam.edu.vn

- List vs NumPy Array
- Advanced Techniques in NumPy
- Practice NumPy Array
- Q&A

NumPy Arrays

Vinh Dinh Nguyen
PhD in Computer Science

QuangDu Nguyen
PhD in Construction Engineering

List vs NumPy Array

What are Arrays?

Vectors are typically introduced in high school math courses, e.g., point (x, y) in the plane, point (x, y, z) in space. In general, a vector v can be mathematically defined as a tuple with n numbers: $v = (v_0, v_1, \ldots, v_{n-1})$ (one index to access elements: e.g., v_1)

Arrays are a generalization of **vectors** where we can have more than one index.
Examples: $A_{i,j}$ (2 indices) and $B_{i,j,k}$ (3 indices)

Complex data represented by arrays:
2D images (height × width); RGB images (channel x height x width)

List	NumPy Array
A sequence of values and values can be **any type** L = [element_1, …, element_n]	A collection of “items” of the same **same types** – All arrays are **homogeneous** A = np.array([element_1,…, element_n])

List vs NumPy Array

How to create Arrays?

List

[9]
```python
# Create a list
lst = ["AIO", 2025, 3.14, -1, True, [1, 2, 3],]
print("List:",lst)
```

String | Integer | Float | Boolean | List (nested list)

[10]
```python
# Create a list using list comprehension
n = 5
lst_zeros = [0]*n
print("Initialised list with zeros:",lst_zeros)
```

```
Initialised list with zeros: [0, 0, 0, 0, 0]
```

NumPy Array

[12]
```python
import numpy as np

arr = np.array([2, 0, 2, 5, 7, 1])
print("Create an array:", arr)
```

```
Create an array: [2 0 2 5 7 1]
```

[17]
```python
import numpy as np

# Create an array of n floats and initialise it with zeros
n = 5
arr_zeros = np.zeros(n)
print("Initialised array with zeros:", arr_zeros)
```

```
Initialised array with zeros: [0. 0. 0. 0. 0.]
```

[14]
```python
import numpy as np

# Convert list into array of integers
lst_int = [2, 0, 2, 5, 7, 1]
arr_int = np.array(lst_int)
print("Array from list:", arr_int)
```

```
Array from list: [2 0 2 5 7 1]
```

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

Arrays Indexing and Slicing?

List

l_data = [4, 5, 6, 7, 8, 9]

Forward index

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

Backward index

l_data[2] = 6

l_data[-4] = 6

NumPy Array

a_data = np.array([4, 5, 6, 7, 8, 9])

Forward index

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

Backward index

a_data[2] = 6

a_data[-4] = 6

List vs NumPy Array

Arrays Indexing and Slicing?

List

l_data = [4, 5, 6, 7, 8, 9]

list[start:end:step]

Forward index

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

l_data[:3] =

4	5	6

l_data[3:] =

7	8	9

l_data[2:4] =

6	7

Start: 0
End: len(list)
Step: 1

NumPy Array

a_data = np.array([4, 5, 6, 7, 8, 9])

array[start:end:step]

Forward index

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

a_data[:3] =

4	5	6

a_data[3:] =

7	8	9

a_data[2:4] =

6	7

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

Arrays Indexing and Slicing?

List	NumPy Array
l_data = [4, 5, 6, 7, 8, 9]	a_data = np.array([4, 5, 6, 7, 8, 9])
list[start:end:step]	array[start:end:step]

List

Forward index

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

l_data[::2] =

4	6	8

l_data[-5::2] =

5	7	9

By default:
Start: 0
End: len(list)
Step: 1

NumPy Array

Forward index

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

a_data[-6::2] =

4	6	8

a_data[1::2] =

7	8	9

List vs NumPy Array

Add an element

List

```
l_data = [4, 5, 6, 7, 8, 9]
```

```
l_data.append(element)
```

NumPy Array

```
a_data = np.array([4, 5, 6, 7, 8, 9])
```

```
np.append(a_data, element)
```

 # thêm 0 vào vị trí cuối list

```
l_data.append(0)
```

4	5	6	7	8	9	0

```
np.append(a_data, 0)
```

4	5	6	7	8	9	0

 # thêm 1 vào vị trí có index = 0

```
l_data.insert(0,1)
```

1	4	5	6	7	8	9

```
np.insert(a_data,0,1)
```

1	4	5	6	7	8	9

List vs NumPy Array

Update an element

List	NumPy Array
l_data = [4, 5, 6, 7, 8, 9]	a_data = np.array([4, 5, 6, 7, 8, 9])
l_data[index] = new element	a_data[index] = new element

 # Update element at index 2 with 1

List	NumPy Array
l_data[2] = 1	a_data[2] = 1
4 5 1 7 8 9	4 5 1 7 8 9

 # Add a list of elements [2, 3] at the end

List	NumPy Array
l_data.extend([2,3])	np.append(a_data,[2,3])
4 5 6 7 8 9 2 3	4 5 6 7 8 9 2 3

List vs NumPy Array

+ and * operators

List	NumPy Array
lst_1 = [6, 8] Lst_2 = [7, 9]	arr_1 = [6, 8] arr_2 = [7, 9]

+ operation - Join two lists/arrays

lst_1 + lst_2	np.append(arr_1, arr_2)
6 8 7 9	6 8 7 9

* operation

lst_1 * 3	arr_1 * 3
6 8 6 8 6 8	18 24

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

Sorting

List	NumPy Array
lst = [2, 1, 6, 8, 3, 9]	arr = [2, 1, 6, 8, 7, 9]

Ascending order

List	NumPy Array
lst.sort()	arr.sort()
1, 2, 3, 6, 8, 9	1, 2, 3, 6, 8, 9

Descending order

List	NumPy Array
lst.sort(reverse=True)	arr.sort() then arr[::-1]
9, 8, 6, 3, 2, 1	9, 8, 6, 3, 2, 1

List vs NumPy Array

Delete an element

List	NumPy Array
lst = [2, 1, 5, 8, 5, 9]	arr = [2, 1, 5, 8, 5, 9]

 # Delete element based on its index

List	NumPy Array
`lst.pop(2)`	`np.delete(arr,2)`
[2, 1, 8, 5, 9]	[2, 1, 8, 5, 9]

 # Delete element(s) based on value

List	NumPy Array
`lst.remove(5)`	`del_indice = np.where(arr == 5)[0]` then `np.delete(arr,del_indice)`
[2, 1, 8, 5, 9]	[2, 1, 8, 9]

List vs NumPy Array

Delete an element

List

lst =

2	1	5	8	5	9

NumPy Array

arr =

2	1	5	8	5	9
0	1	2	3	4	5

Delete elements at index 1 and 2

```
del lst[1:3]
```

2	8	5	9

```
np.delete(arr,np.arange(1,3))
```

2	8	5	9

start_index = 1
end_index = 3

```
np.concatenate((arr[:1],arr[3:]))
```

2	8	5	9

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

index()

List	NumPy Array

lst = | 2 | 1 | 5 | 8 | 5 | 9 |

arr = | 2 | 1 | 5 | 8 | 5 | 9 |
index: | 0 | 1 | 2 | 3 | 4 | 5 |

 # Find the index where the target element first appears

```
lst.index(5)
```

2	1	5	8	5	9

2

```
np.where(arr == 5)[0][0]
```

array([2, 4]) [0]

2	1	5	8	5	9

2

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

index()

List

```
l_x = [0, 0.25, 0.5, 0.75, 1.0]

# Indexing list
l_y = l_x[1:4] #  [0.25, 0.5, 0.75]

#
l_y[-1] = 1000.0
print("Re-Check elements of list y: \n", l_y)
print("Re-Check elements of list x: \n", l_x)
```

```
Re-Check elements of list y:
 [0.25, 0.5, 1000.0]
Re-Check elements of list x:
 [0, 0.25, 0.5, 0.75, 1.0]
```

NumPy Array

```
import numpy as np
x = np.linspace(0.0, 1.0, 5)
print("Array x: \n", x)

# array assignment
y = x[1:4]
print("Check elements of array y: \n", y)
y[-1] = 1000.0

# Check elements of array x
print("Re-Check elements of array y: \n", y)
print("Re-Check elements of array x: \n", x)
```

```
Array x:
 [0.   0.25 0.5  0.75 1.  ]
Re-Check elements of array y:
 [2.5e-01 5.0e-01 1.0e+03]
Re-Check elements of array x:
 [0.0e+00 2.5e-01 5.0e-01 1.0e+03 1.0e+00]
```

List vs NumPy Array

count() and copy()

List	NumPy Array
lst = [2, 1, 5, 8, 5, 9]	arr = [2, 1, 5, 8, 5, 9]

 # Count number of times the target element appears

List	NumPy Array
lst.count(element_to_count)	np.sum(arr == element_to_count)
lst.count(5) → 2	np.sum(arr == 5) → 2

 # Copy

List	NumPy Array
lst.copy()	arr.copy()
Lst_cp = [2, 1, 5, 8, 5, 9]	arr_cp = [2, 1, 5, 8, 5, 9]

List vs NumPy Array

len(), min(), max()

List	NumPy Array
lst = [6, 5, 7, 1, 9, 2]	arr = [6, 5, 7, 1, 9, 2]

List

lst = | 6 | 5 | 7 | 1 | 9 | 2 |

```
# trả về số phần tử
len(lst) = 6

# trả về số phần tử có giá trị nhỏ nhất
min(lst) = 1

# trả về số phần tử có giá trị lớn nhất
max(lst) = 9
```

NumPy Array

arr = | 6 | 5 | 7 | 1 | 9 | 2 |

```
# trả về số phần tử
len(arr) = 6

# trả về số phần tử có giá trị nhỏ nhất
np.min(arr) = 1

# trả về số phần tử có giá trị lớn nhất
np.max(arr) = 9

# trả về số chiều của array
arr. shape = (6,)
```

List vs NumPy Array

sum()

List

lst =

6	5	7	1	9	2

NumPy Array

data =

6	5	7	1	9	2

$$summation = \sum_{i=0}^{n} data_i$$

tính tổng
sum(lst) = 30

```
data = [6, 5, 7, 1, 9, 2]
print(data)

summation = sum(data)
print(summation)
```

```
[6, 5, 7, 1, 9, 2]
30
```

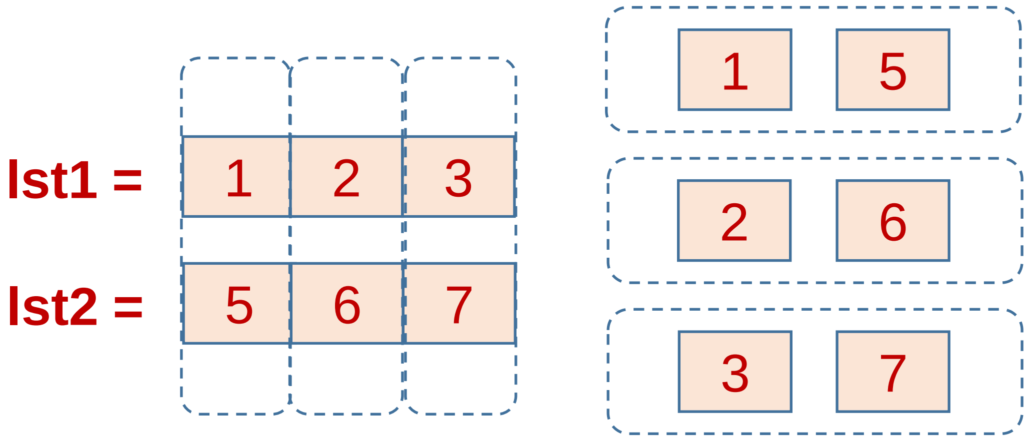
tính tổng
np.sum(data) = 30

```
import numpy as np

data = np.array([6, 5, 7, 1, 9, 2])
print(data)

summation = np.sum(data)
print(summation)
```

```
[6, 5, 7, 1, 9, 2]
30
```

List vs NumPy Array

List - zip()




```python
l1 = [1, 2, 3]
l2 = [5, 6, 7]

# print in pairs
length = len(l1)
for i in range(length):
    print(l1[i], l2[i])
```

```python
l1 = [1, 2, 3]
l2 = [5, 6, 7]

# print in pairs
for v1, v2 in zip(l1, l2):
    print(v1, v2)
```

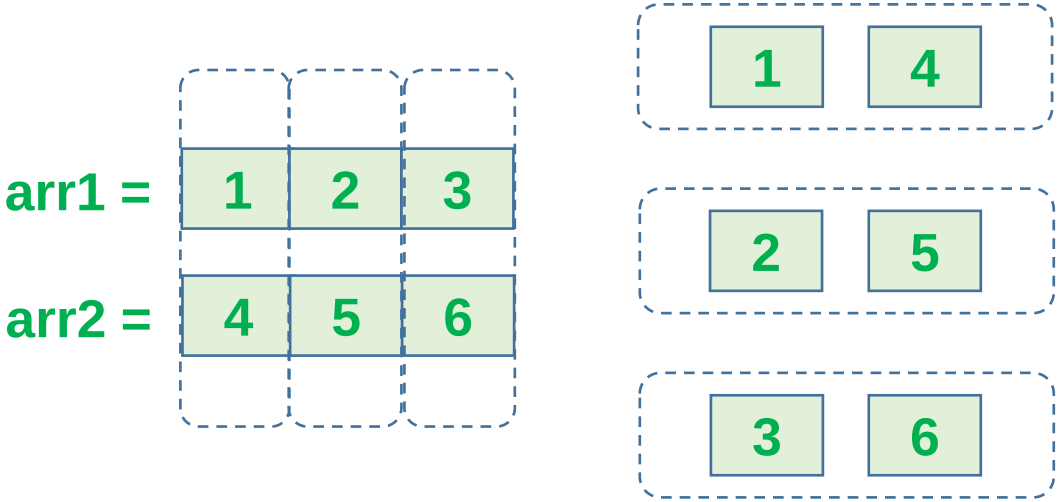
```python
predictions = [1, 0, 1, 1, 0]
ground_truth = [1, 1, 1, 0, 0]

# Use zip to pair predictions with labels
for pred, true in zip(predictions, ground_truth):
    result = "Correct" if pred == true else "Incorrect"
    print(f"Predicted: {pred}, Actual: {true} → {result}")
```

```
Predicted: 1, Actual: 1 → Correct
Predicted: 0, Actual: 1 → Incorrect
Predicted: 1, Actual: 1 → Correct
Predicted: 1, Actual: 0 → Incorrect
Predicted: 0, Actual: 0 → Correct
```

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

Array – No zip() function




```python
import numpy as np

# Example arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Zip-like operation using numpy.dstack()
zipped_arr = np.dstack((arr1, arr2))[0]
print(len(zipped_arr))
for i in range(len(arr1)):
    print(zipped_arr[i])
```

Output

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

List vs NumPy Array

List – enumerate()

lst = | 6 | 1 | 7 |

enumerate(lst) = | 6 | 1 | 7 |

index 0 1 2

```python
# get index and value
data = [6, 1, 7]

length = len(data)
for index in range(length):
    print(index, data[index])
```

```
0 6
1 1
2 7
```

```python
# enumerate
data = [6, 1, 7]
for index, value in enumerate(data):
    print(index, value)
```

```
0 6
1 1
2 7
```

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

Array – np.ndenumerate()

```python
import numpy as np

# create an array
arr = np.array([6, 1, 7])

# Get index and respective element
for idx, val in np.ndenumerate(arr):
    print(idx, val)
```

```
(0,) 6
(1,) 1
(2,) 7
```

arr =

6	1	7

np.ndenumerate(arr) =

6	1	7
(0,)	(1,)	(2,)

index

List vs NumPy Array

List Examples

Sum of even numbers

lst =

6	5	7	1	9	2

```python
# sum of even number
def sum1(data):
    result = 0

    for value in data:
        if value%2 == 0:
            result = result + value

    return result

# test
data = [6, 5, 7, 1, 9, 2]
summation = sum1(data)
print(summation)
```

```
8
```

Sum of elements with even indices

lst =

6	5	7	1	9	2

```python
# sum of numbers with even indices
def sum2(data):
    result = 0

    length = len(data)
    for index in range(length):
        if index%2 == 0:
            result = result + data[index]

    return result


# test
data = [6, 5, 7, 1, 9, 2]
summation = sum2(data)
print(summation)
```

```
22
```

List vs NumPy Array

NumPy Array Examples

Sum of even numbers

```python
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Use boolean indexing to find even numbers
even_numbers = arr[arr % 2 == 0]

# Calculate the sum of even numbers
sum_even_numbers = np.sum(even_numbers)

print("Original array:", arr)
print("Even numbers:", even_numbers)
print("Sum of even numbers:", sum_even_numbers)
```

Sum of elements with even indices

```python
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Select elements at even indices
even_index_elements = arr[::2]  # This selects every second element starting from index 0

# Calculate the sum of elements at even indices
sum_even_indices = np.sum(even_index_elements)

print("Original array:", arr)
print("Elements at even indices:", even_index_elements)
print("Sum of elements at even indices:", sum_even_indices)
```

List vs NumPy Array

2D list vs 2D array

List

```python
# Create a 2D list
list_2d = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
]

print(list_2d)
```

Output

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

NumPy Array

```python
import numpy as np

# Create a 2D NumPy array
array_2d = np.array([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
])

# Accessing elements in a 2D NumPy array
print("\n2D NumPy Array:")
print(array_2d)
```

Output

```
2D NumPy Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

2D list vs 2D array

List

```python
# Create a 2D list
list_2d = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
]

print(list_2d)
```

Output

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```python
# Accessing elements in 2D list
print("Element at (1,2) in list_2d:", list_2d[1][2])
```

NumPy Array

```python
import numpy as np

# Create a 2D NumPy array
array_2d = np.array([
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
])

# Accessing elements in a 2D NumPy array
print("\n2D NumPy Array:")
print(array_2d)
```

Output

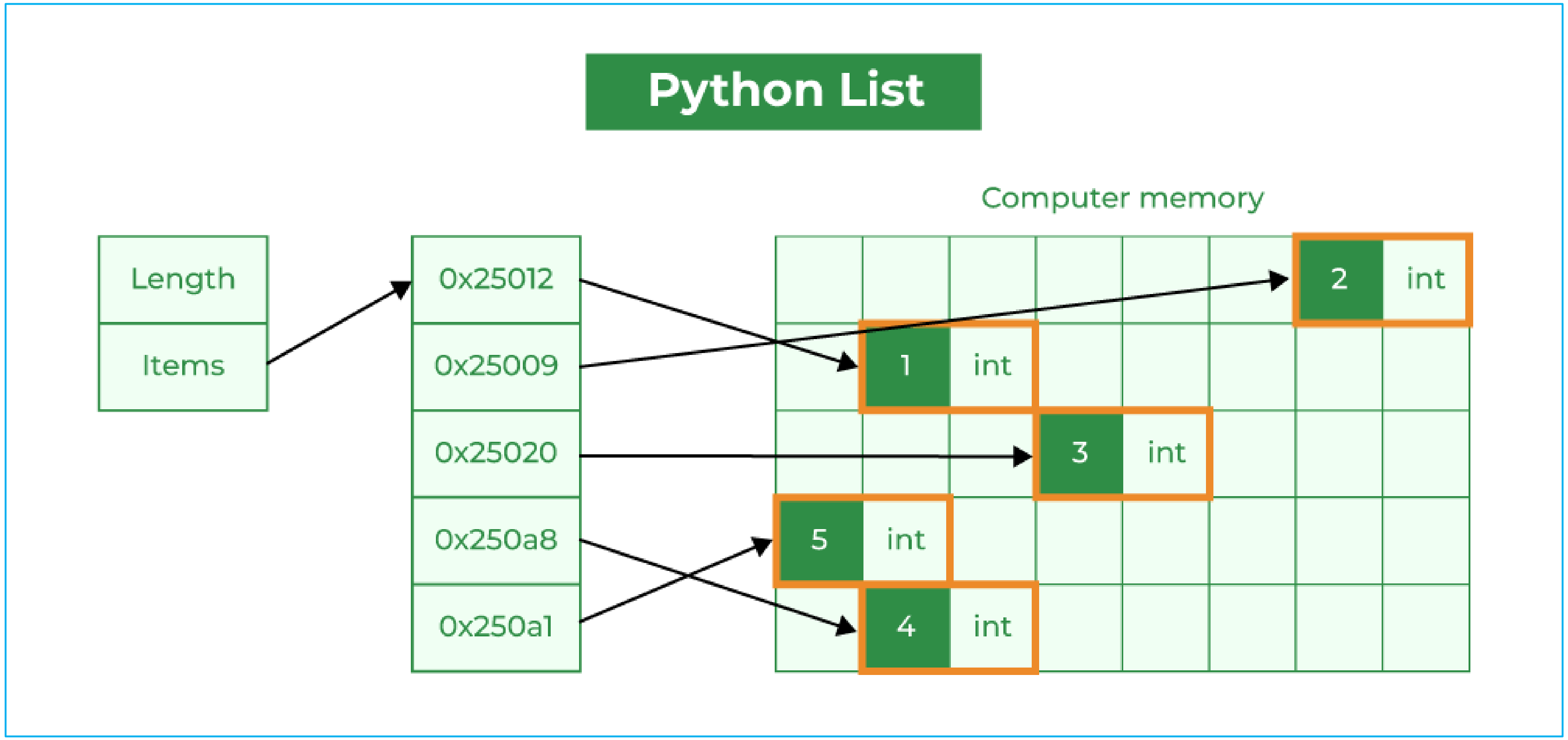
```
2D NumPy Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```python
# Accessing elements in 2D NumPy array
print("Element at (1,2) in array_2d:", array_2d[1, 2])
```

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

Python List Limitations

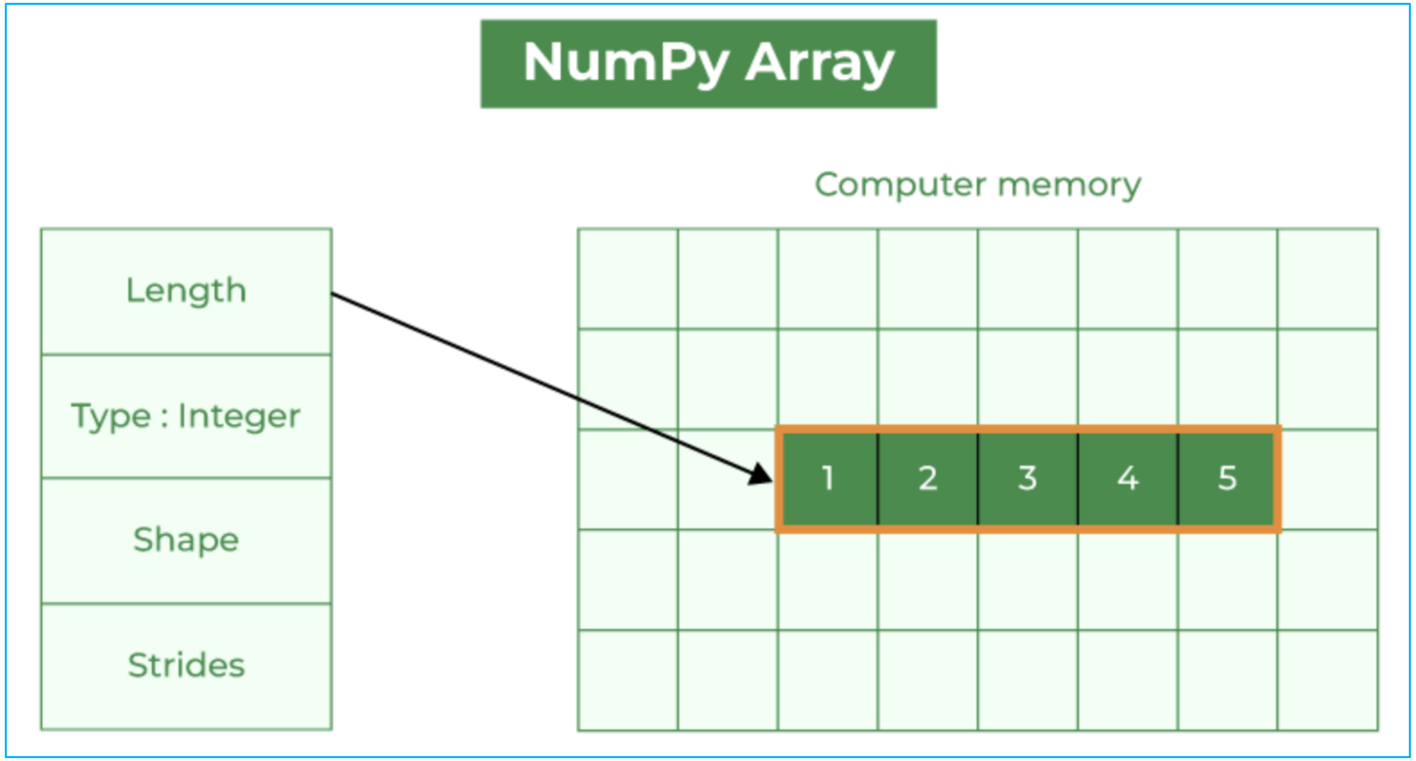


1. Element Overhead
2. Memory Fragmentation
3. Performance
4. Functionality

AI VIET NAM
@aivietnam.edu.vn

List vs NumPy Array

NumPy Array Motivation



1.Homogeneous Data
2.Fixed Data Type
3.Contiguous Memory
4.Performance

- List vs NumPy Array
- Advanced Techniques in NumPy
- Practice NumPy Array
- Q&A

Advanced Techniques in NumPy

Advanced Indexing

Integer Array Indexing

Construct new arrays from another array

```
a = np.array([1, 2, 3, 4, 5])
print(a[[1, 3]])  # Outputs: [2, 4]
```

Boolean Array Indexing

Conditionally select elements

```
a = np.array([1, 2, 3, 4, 5])
print(a[a > 3])  # Outputs: [4, 5]
```

Array Manipulation

[3]

```
# Reshaping
# Change the shape of an array without changing its data
arr = np.array([[1, 2], [3, 4], [5, 6]])
print(arr.reshape(2, 3))
```

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

[7]

```
# Flattening
# Convert a multi-dimensional array into 1D array
arr = np.array([[1, 2], [3, 4]])
print("Array before flattening:\n",arr)
print("Flattened array:", arr.flatten())
```

```
Array before flattening:
 [[1 2]
 [3 4]]
Flattened array: [1 2 3 4]
```

```
# Concatenation
# Joining two or more arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
print("Concatenated array:", np.concatenate((arr1, arr2)))
```

```
Concatenated array: [1 2 3 4 5 6]
```

AI VIET NAM
@aivietnam.edu.vn

Advanced Techniques in NumPy

Broadcasting and Vectorization in NumPy

Broadcasting: The Unsung Hero

Making arrays with mismatched shapes compatible for arithmetic operations.

It “extends” smaller arrays to the shape of the larger array,

Element-wise operations on arrays of different shapes

```
import numpy as np
a = np.array([[1, 2], [3, 4], [5, 6]])
b = np.array([1, 2])

# Broadcasting in action: adding the 1D array 'b' to each row of the 2D array
result = a + b
```

Advanced Techniques in NumPy

Broadcasting and Vectorization in NumPy

Vectorization:
Powering Efficient Computations

The execution of operations on entire arrays, **eliminating** the need for **explicit loops**

Making calculations significantly **faster** than traditional Python loops.

Almost all **NumPy operations** are **inherently vectorized**.

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

# Vectorized multiplication of the two arrays
result = a * b
# Outputs: [5 12 21 32]
```

Advanced Techniques in NumPy

Non-vectorized vs Vectorized code (performance)

Evaluates $f(x) = \sin x e^{-x}$ at 10^6 equispaced points within $[0, 2\pi]$

```python
# Non-vectorized VS vectorized code (performance)
import math
import numpy as np
n = 1_000_000
x = np.linspace(0.0, 2.0*math.pi, n)
y = np.zeros(n)

%timeit for i in range(0,len(x)): \
y[i] = math.sin(x[i])*math.exp(-x[i])
```

```python
# Non-vectorized VS vectorized code (performance)
import math
import numpy as np
n = 1_000_000
x = np.linspace(0.0, 2.0*math.pi, n)
y = np.zeros(n)

%timeit y = np.sin(x)*np.exp(-x)
```

```
266 ms ± 8.03 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
21.4 ms ± 1.93 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

- *Code run on Google Colab*
- *Measurements taken with %timeit magic command in Ipython*
- *266 vs 21.4 millisecs. (vectorized code ≈12.42 times FASTER!)*

IMPORTANT(!): we cannot use math.sin and math.exp on entire arrays, we must use NumPy versions np.sin and np.exp instead

Advanced Techniques in NumPy

Non-vectorized vs Vectorized code (performance)

```python
import numpy as np
import time

# Vectorized version of f(x)
def f(x):
    return x**3 + np.sin(x) * np.exp(-3*x)

# Scalar (loop-based) version of f(x)
def f_scalar(x_array):
    results = []
    for x in x_array:
        results.append(x**3 + np.sin(x) * np.exp(-3*x))
    return np.array(results)

# Array sizes to test
sizes = [10, 100, 1000, 10_000, 100_000]

# Measure and compare performance
for size in sizes:
    x = np.linspace(0, 1, size)

    # Vectorized timing
    start = time.perf_counter()
    y_vec = f(x)
    end = time.perf_counter()
    t_vec = (end - start) * 1000  # convert to ms

    # Scalar timing
    start = time.perf_counter()
    y_sca = f_scalar(x)
    end = time.perf_counter()
    t_sca = (end - start) * 1000  # convert to ms

    speedup = t_sca / t_vec if t_vec > 0 else float('inf')

    print(f"{size:<10}{t_vec:<20.4f}{t_sca:<20.4f}{speedup:.1f}x")
```

Size	Vectorized (ms)	Scalar (ms)	Speedup (x)
10	0.0383	0.0371	1.0x
100	0.0335	0.1866	5.6x
1000	0.5082	2.9573	5.8x
10000	1.3810	17.2290	12.5x
100000	4.2362	173.8604	41.0x

AI VIET NAM
@aivietnam.edu.vn

Advanced Techniques in NumPy

Shape of NumPy Arrays

Python

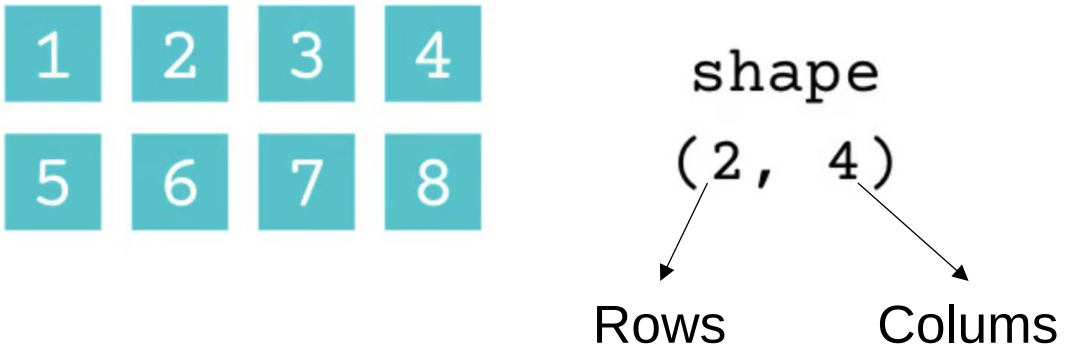
```python
>>> import numpy as np
>>> numbers = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
>>> numbers
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Python

```python
>>> numbers.shape
(2, 4)
```

Python

```python
>>> numbers.ndim
2
```



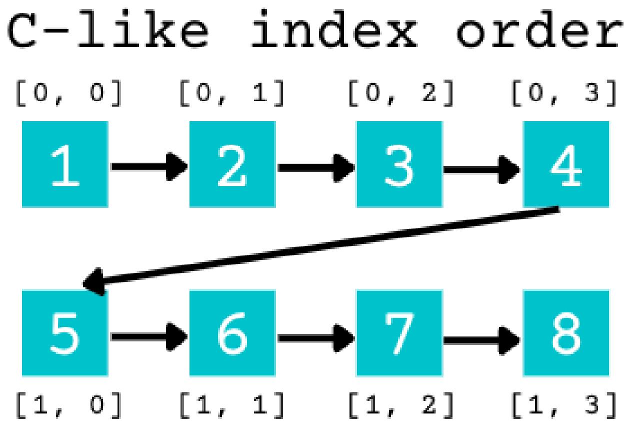
AI VIET NAM
@aivietnam.edu.vn

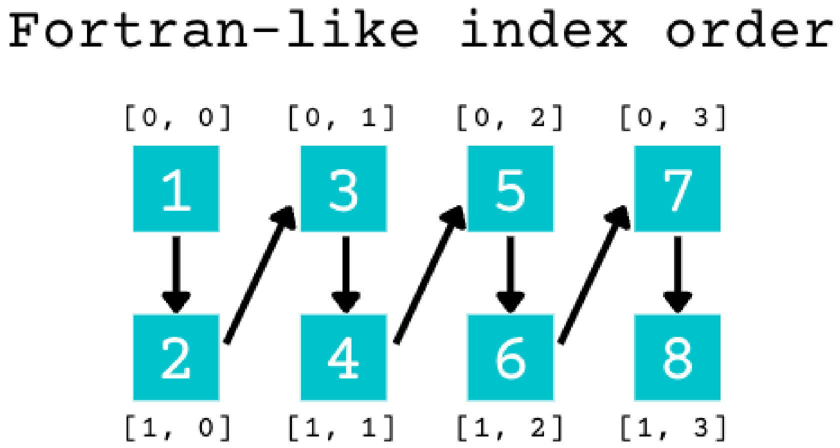
Advanced Techniques in NumPy

Reshape in NumPy Arrays

Python

```
>>> import numpy as np
>>> numbers = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> numbers
array([1, 2, 3, 4, 5, 6, 7, 8])
>>> numbers.shape
(8,)
>>> numbers.ndim
1

>>> numbers.reshape((2, 4), order="C")
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

>>> numbers.reshape((2, 4), order="F")
array([[1, 3, 5, 7],
       [2, 4, 6, 8]])
```





QUIZ TIME

- List vs NumPy Array
- Advanced Techniques in NumPy
- Practice NumPy Array
- Q&A

Practice NumPy Array

Stack two arrays vertically

arr_1 =

0	1	2	3	4
5	6	7	8	9

arr_2 =

8	8	8	8	8
8	8	8	8	8

axis 1

axis 0

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

```python
import numpy as np

# Create the first array
arr_1 = np.arange(10).reshape((2,-1))
print("The first array:\n",arr_1)

# Create the second array
arr_2 = np.repeat(8,10).reshape((2,-1))
print("The second array:\n",arr_2)

```

```
The first array:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
The second array:
 [[8 8 8 8 8]
 [8 8 8 8 8]]
```

```python
# Method 1
new_arr = np.concatenate((arr_1, arr_2), axis=0)
print("Concatenated array:\n",new_arr)

# Method 2
new_arr = np.vstack((arr_1,arr_2))
print("Veritally stack arrays:\n",new_arr)

# Method 3
new_arr = np.r_[arr_1,arr_2]
print("Veritally stack arrays:\n",new_arr)
```

Practice NumPy Array

Stack two arrays horizontally

arr_1 =

0	1	2	3	4
5	6	7	8	9

arr_2 =

8	8	8	8	8
8	8	8	8	8

axis 1

axis 0

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

```python
import numpy as np

# Create the first array
arr_1 = np.arange(10).reshape((2,-1))
print("The first array:\n",arr_1)

# Create the second array
arr_2 = np.repeat(8,10).reshape((2,-1))
print("The second array:\n",arr_2)

```

```python
# Method 1
new_arr = np.concatenate((arr_1, arr_2), axis=1)
print("Concatenated array:\n",new_arr)

# Method 2
new_arr = np.hstack((arr_1,arr_2))
print("Horizontally stack arrays:\n",new_arr)

# Method 3
new_arr = np.c_[arr_1,arr_2]
print("Horizontally stack arrays:\n",new_arr)
```

Practice NumPy Array

 # Take all elements satisfying with a given predefined condition (e.g. less than 7)

arr =

1	8	2	5	4	6	0	7	9	3

arr < 7 =

T	F	T	T	T	T	T	F	F	T

arr[arr < 7] =

1	2	5	4	6	0	3

```python
import numpy as np

# create an array
arr = np.array([1, 8, 2, 5, 4, 6, 0, 7, 9, 3])

```

```python
# Method 1
idx = np.where(arr < 7)
out = arr[idx]
print(out)

# Method 2
out = arr[arr<7]
print(out)
```

Practice NumPy Array

Apply a user-defined function for array

arr_1 =

0	1	2	3	4
5	6	7	8	9

arr_2 =

2	5	1	9	7
3	8	6	0	4

Element-wise compare
& take greater element

2	5	2	9	7
5	8	7	8	9

```python
import numpy as np

# Create the first array
arr_1 = np.arange(10).reshape((2,-1))
print("The first array:\n",arr_1)

# Create the second array
arr_2 = np.array([2,5,1,9,7,3,8,6,0,4]).reshape((2,-1))
print("The second array:\n",arr_2)

# Create a user-defined function
def get_max(x, y):
    if x >= y:
        return x
    else:
        return y
```

```python
# Vectorize the function
element_wise_compare = np.vectorize(get_max, otypes=[int])

# Method 1: Apply the function
out1 = element_wise_compare(arr_1, arr_2)
print("Output using the first method:\n",out1)

# Method 2: Using np.maximum
out2 = np.maximum(arr_1, arr_2)
print("Output using the second method:\n",out2)

# Method 3: Using np.where()
out3 = np.where(arr_1 > arr_2, arr_1, arr_2)
print("Output using the third method:\n",out3)
```

Practice NumPy Array

Length of a Vector

$$\vec{u} = \begin{bmatrix} u_1 \\ \dots \\ u_n \end{bmatrix}$$

$$\|\vec{u}\| = \sqrt{u_1^2 + \cdots + u_n^2}$$

1
2
4
2

```python
import numpy as np

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

# Compute the length of vector u
print("Length of vector u:", np.linalg.norm(u))
```

```
Length of vector u: 5.0
```

$$\|\vec{u}\| = \sqrt{1^2 + 2^2 + 4^2 + 2^2}$$

Practice NumPy Array

Vector addition

$$\vec{v} = \begin{bmatrix} v_1 \\ \dots \\ v_n \end{bmatrix} \quad \vec{u} = \begin{bmatrix} u_1 \\ \dots \\ u_n \end{bmatrix}$$

$$\vec{v} + \vec{u} = \begin{bmatrix} v_1 \\ \dots \\ v_n \end{bmatrix} + \begin{bmatrix} u_1 \\ \dots \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 + u_1 \\ \dots \\ v_n + u_n \end{bmatrix}$$

$$\vec{u} + \vec{v} = \vec{v} + \vec{u}$$

v		u		result
1		5		6
2	+	6	=	8
3		7		10
4		8		12

```python
import numpy as np

# Create two vectors
u = np.array([1, 2, 3, 4])
v = np.array([5, 6, 7, 8])

# Add two vectors
out_1 = u + v
out_2 = np.add(u, v)
```

```
Out 1:
 [ 6  8 10 12]
Out 2:
 [ 6  8 10 12]
```

Practice NumPy Array

Hadamard Product

$$\vec{v} = \begin{bmatrix} v_1 \\ \dots \\ v_n \end{bmatrix} \quad \vec{u} = \begin{bmatrix} u_1 \\ \dots \\ u_n \end{bmatrix}$$

$$\vec{v} \odot \vec{u} = \begin{bmatrix} v_1 \\ \dots \\ v_n \end{bmatrix} \odot \begin{bmatrix} u_1 \\ \dots \\ u_n \end{bmatrix} = \begin{bmatrix} v_1 \times u_1 \\ \dots \\ v_n \times u_n \end{bmatrix}$$

v		u		result
1		5		5
2	*	6	=	12
3		7		21
4		8		32

```python
import numpy as np

# Create two vectors
u = np.array([1, 2, 3, 4])
v = np.array([5, 6, 7, 8])

# Add two vectors
out_1 = u * v
out_2 = v * u
out_3 = np.multiply(u, v)

print("Out 1:\n",out_1)
print("Out 2:\n",out_2)
print("Out 3:\n",out_3)
```

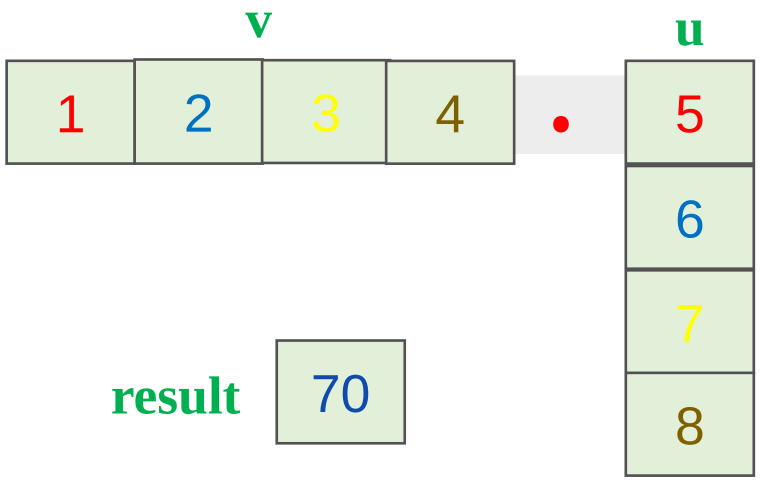
```
Out 1:
 [ 5 12 21 32]
Out 2:
 [ 5 12 21 32]
Out 3:
 [ 5 12 21 32]
```

Practice NumPy Array

Dot Product

$$\vec{v} = \begin{bmatrix} v_1 \\ \dots \\ v_n \end{bmatrix} \quad \vec{u} = \begin{bmatrix} u_1 \\ \dots \\ u_n \end{bmatrix}$$

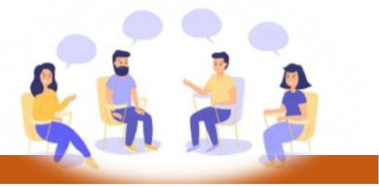
$$\vec{v} \cdot \vec{u} = v_1 \times u_1 + \cdots + v_n \times u_n$$




```python
import numpy as np

# Create two vectors
u = np.array([1, 2, 3, 4])
v = np.array([5, 6, 7, 8])

# Add two vectors
out_1 = u.dot(v)
out_2 = np.dot(v, u)
out_3 = np.dot(u, v)

print("Out 1:\n",out_1)
print("Out 2:\n",out_2)
print("Out 3:\n",out_3)
```

```
Out 1:
 70
Out 2:
 70
Out 3:
 70
```

Practice NumPy Array



1D List/Array Multiplication

```python
import time
import numpy as np

def multiply_lists(size=1000):
    list1 = list(range(size))
    list2 = list(range(size))

    start = time.perf_counter()
    result = [(a * b) for a, b in zip(list1, list2)]
    end = time.perf_counter()

    elapsed_ms = (end - start) * 1000
    print(f"List multiplication time: {elapsed_ms:.4f} ms")

def multiply_arrays(size=1000):
    arr1 = np.arange(size)
    arr2 = np.arange(size)

    start = time.perf_counter()
    result = arr1 * arr2
    end = time.perf_counter()

    elapsed_ms = (end - start) * 1000
    print(f"NumPy array multiplication time: {elapsed_ms:.4f} ms")

# Run both
size = 100000
multiply_lists(size)
multiply_arrays(size)
```

```
List multiplication time: 4.1971 ms
NumPy array multiplication time: 0.1146 ms
```

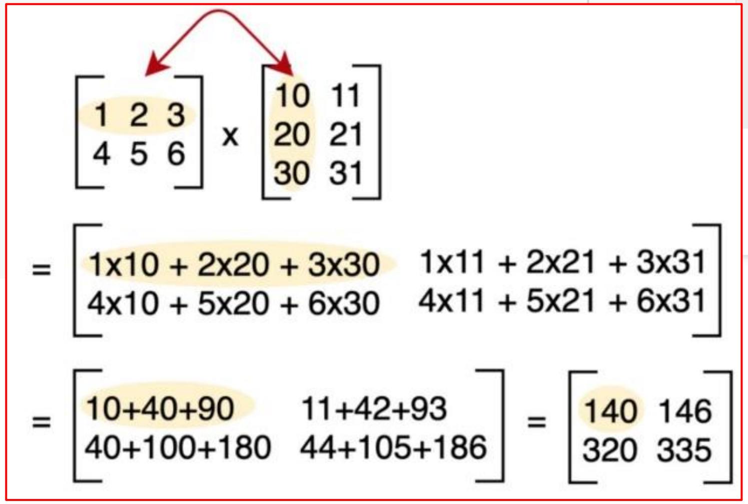
Practice NumPy Array

2D List Multiplication

```python
def matrix_multiply(A, B):
    # Get the number of rows and columns for the input matrices
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])

    # Ensure the number of columns in A is equal to the number of rows in B
    if cols_A != rows_B:
        raise ValueError("Number of columns in A must be equal to number of rows in B")

    # Initialize the result matrix with zeros
    result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]

    # Perform the matrix multiplication
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result
```

```python
# Example matrices
A = [[1, 2, 3],
     [4, 5, 6]]

B = [[7, 8],
     [9, 10],
     [11, 12]]

# Multiply the matrices
result = matrix_multiply(A, B)

# Print the result
for row in result:
    print(row)
```

```
[58, 64]
[139, 154]
```



Practice NumPy Array

2D Array Multiplication

```python
import numpy as np

# Define the matrices
A = np.array([
    [1, 2, 3],
    [4, 5, 6]
])

B = np.array([
    [7, 8],
    [9, 10],
    [11, 12]
])

# Multiply the matrices using np.dot
result = np.dot(A, B)

# Alternatively, you can use the @ operator
# result = A @ B

# Print the result
print(result)
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

$$= \begin{bmatrix} 1\text{x}10 + 2\text{x}20 + 3\text{x}30 & 1\text{x}11 + 2\text{x}21 + 3\text{x}31 \\ 4\text{x}10 + 5\text{x}20 + 6\text{x}30 & 4\text{x}11 + 5\text{x}21 + 6\text{x}31 \end{bmatrix}$$

$$= \begin{bmatrix} 10+40+90 & 11+42+93 \\ 40+100+180 & 44+105+186 \end{bmatrix} = \begin{bmatrix} 140 & 146 \\ 320 & 335 \end{bmatrix}$$

AI VIET NAM
@aivietnam.edu.vn

Take home massages

- **NumPy Arrays** and lists are sequences with different features
- While **NumPy Arrays** are not as flexible as lists, they provide significantly better performance when used wisely.
- **Vectorization** refers to the process of converting an algorithm that uses explicit Python loops to manipulate individual elements into one that relies on array-wide operations, eliminating the need for such loops.

NumPy Arrays

Vinh Dinh Nguyen
PhD in Computer Science

QuangDu Nguyen
PhD in Construction Engineering