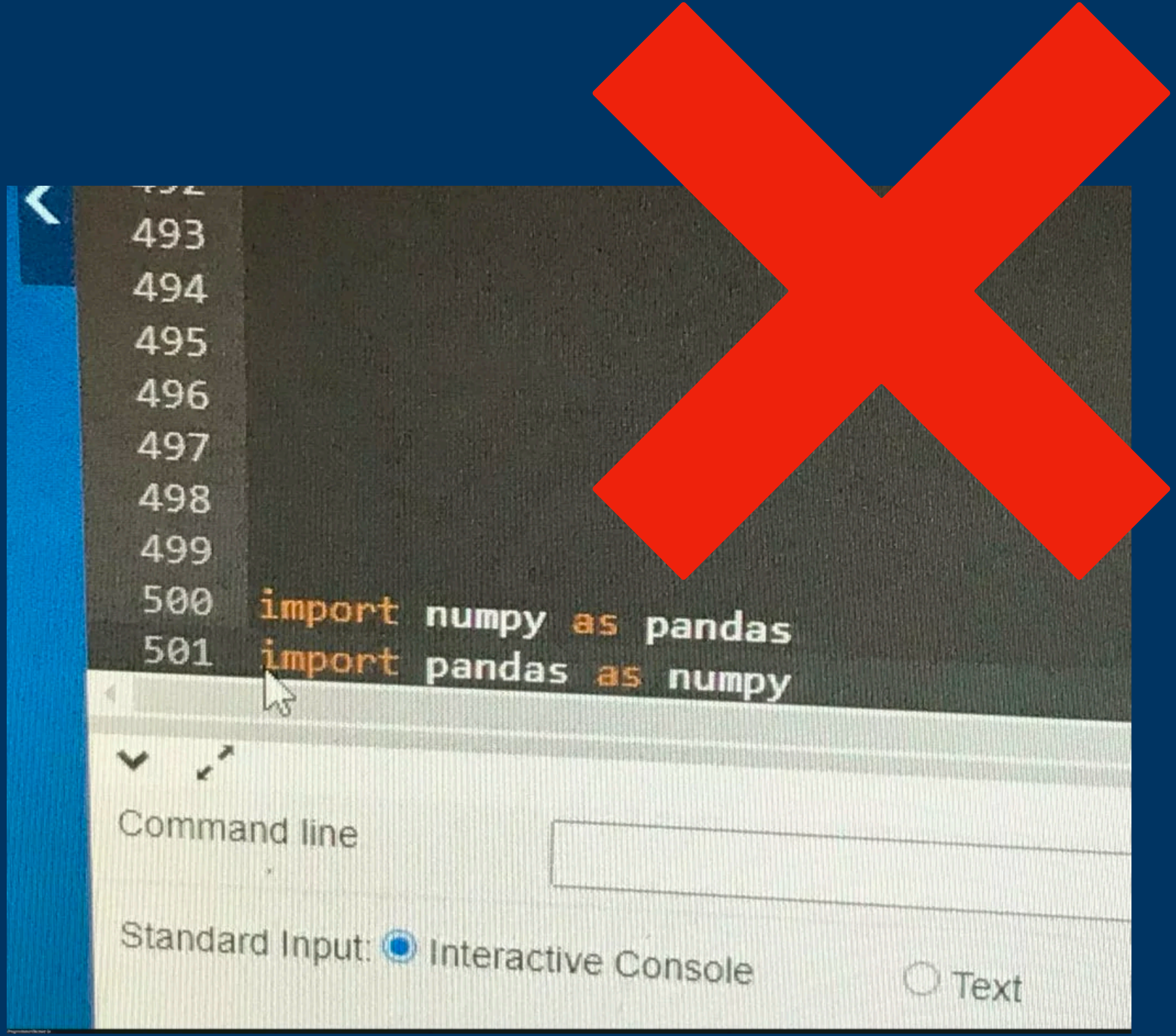
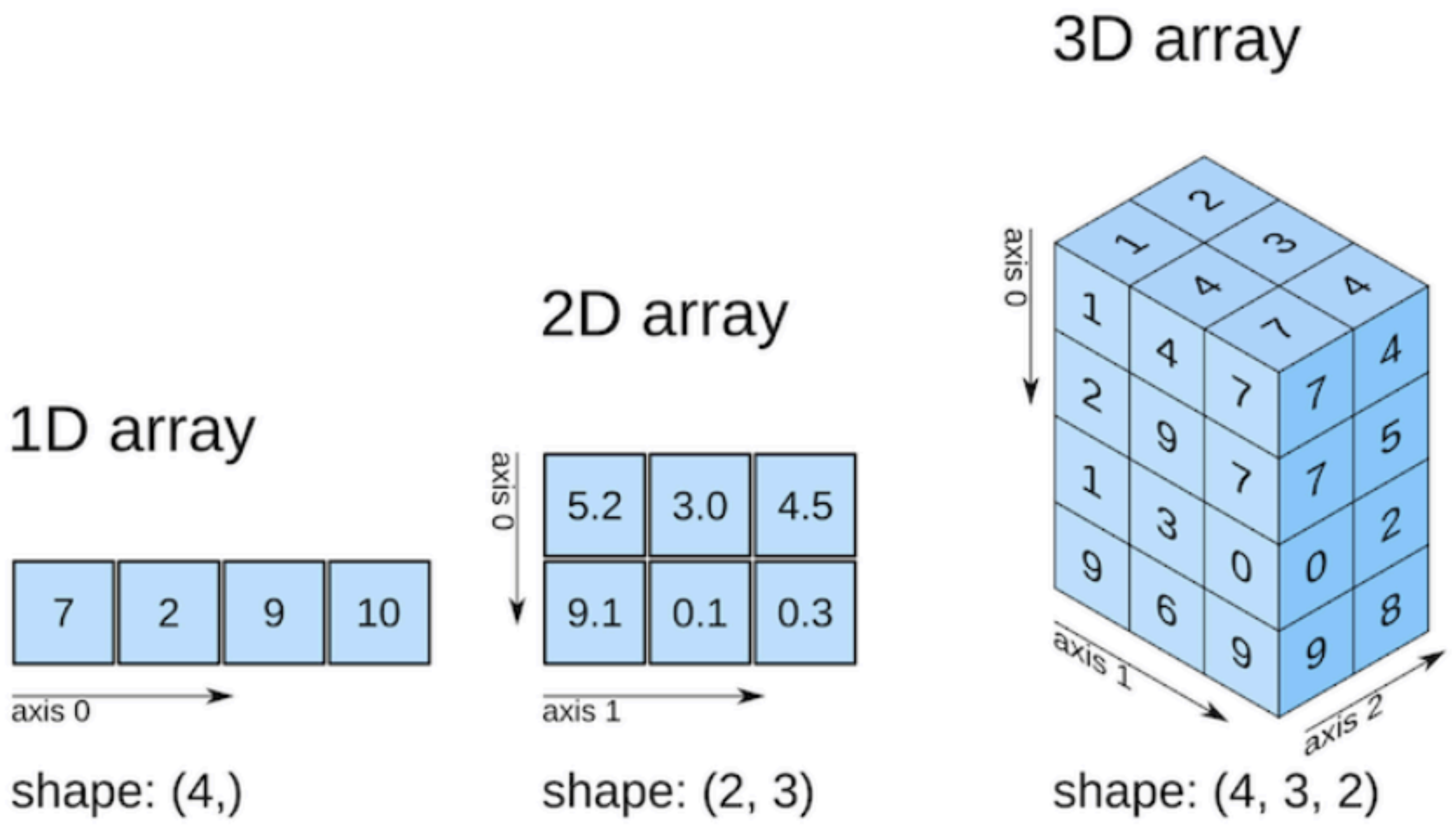
Recitation 4

CS 210



Anamika Lochab

Numpy


3D array
2D array
1D array
7
2
9
10
axis 0
shape: (4,)
axis 0
5.2
3.0
4.5
9.1
0.1
0.3
axis 1
shape: (2, 3)
axis 0
axis 1
axis 2
shape: (4, 3, 2)

List vs NumPy

General Purpose vs Specialized for Numeric Data

- Python Lists (General Purpose) , they can hold heterogeneous data types.

```
mixed_list = ["Hello", 1, 3.14, True]
print(mixed_list)
```

```
['Hello', 1, 3.14, True]
```

- NumPy Arrays (Specialized for Numeric Data), they are specialized for numeric data and are homogeneous.

```
import numpy as np
numeric_array = np.array([1, 2, 3, 4])
print(numeric_array)
```

```
[1 2 3 4]
```

Memory Consumption

- Lists store more than just the data. They also store size, type, and other object overheads.

```
import sys
list_numbers = list(range(1000))
print(sys.getsizeof(list_numbers))
```

```
8056
```

- NumPy arrays are more memory efficient, storing only the data.

```
import numpy as np
array_numbers = np.arange(1000)
print(array_numbers.nbytes)
```

```
8000
```

Performance

- Operations on lists can be slower due to dynamic type checking.

```
list_a = list(range(1000000))
list_b = list(range(1000000, 2000000))
%timeit [a + b for a, b in zip(list_a, list_b)]
```

```
The slowest run took 4.03 times longer than the fastest.
176 ms ± 93.4 ms per loop (mean ± std. dev. of 7 runs, 1
```

- NumPy operations are significantly faster due to fixed type and efficient memory layout.

```
import numpy as np
array_a = np.arange(1000000)
array_b = np.arange(1000000, 2000000)
%timeit array_a + array_b
```

```
3 ms ± 1.09 ms per loop (mean ± std. dev. of 7 runs, 100
```

Functionality

- Lists have basic functionality.

```
list_numbers = [10, 20, 30, 40]
sum_list = sum(list_numbers)
print(sum_list)
```

```
100
```

- NumPy offers a wide range of mathematical and statistical functions.

```
import numpy as np
array_numbers = np.array([10, 20, 30, 40])
mean_val = np.mean(array_numbers)
print(mean_val)
```

```
25.0
```

Flexibility vs Limitations on Resizing

- Lists can be resized easily.

```
list_numbers = [10, 20, 30]
list_numbers.append(40)
print(list_numbers)
```

```
[10, 20, 30, 40]
```

- NumPy arrays can't be resized in place. A new array must be created.

```
import numpy as np
array_numbers = np.array([10, 20, 30])
# To add an element, you need to create a new array
array_numbers = np.append(array_numbers, 40)
print(array_numbers)
```

```
[10 20 30 40]
```

Nested Data and Homogeneity of Size

- Python lists can have nested lists (or lists within lists) of different lengths. This gives them great flexibility but can make certain operations and manipulations challenging.

```
nested_list = [[1, 2, 3], [4, 5], [6]]
print(nested_list)
```

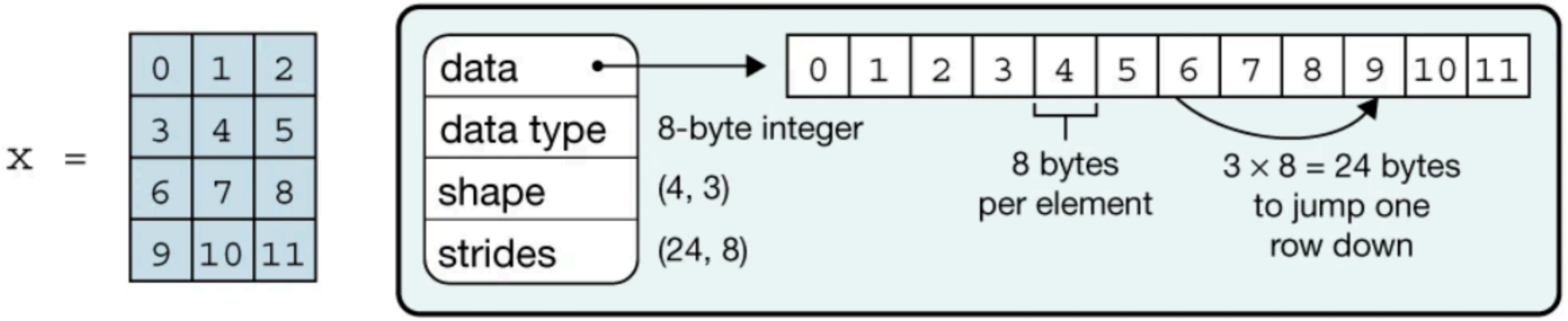
```
[[1, 2, 3], [4, 5], [6]]
```

- NumPy arrays enforce homogeneity, which means every nested sequence must be of the same size. If you're creating a 2D array (matrix), each row must have the same number of columns.

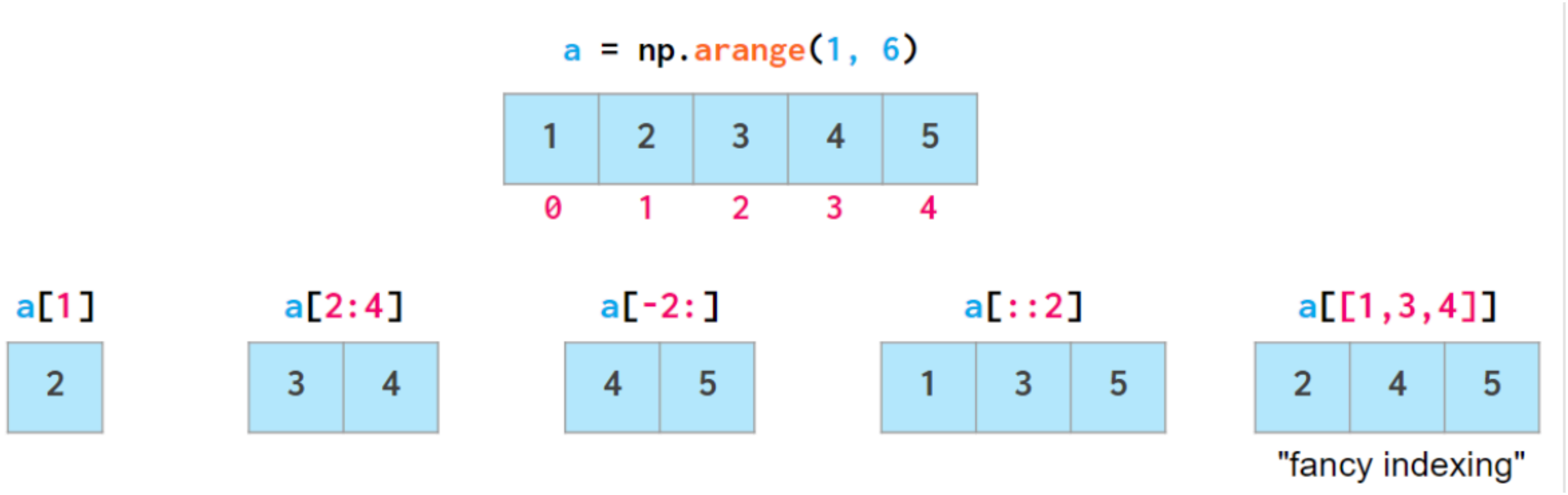
```
import numpy as np

# This will raise an error
invalid_array = np.array([[1, 2, 3], [4, 5], [6]])
```

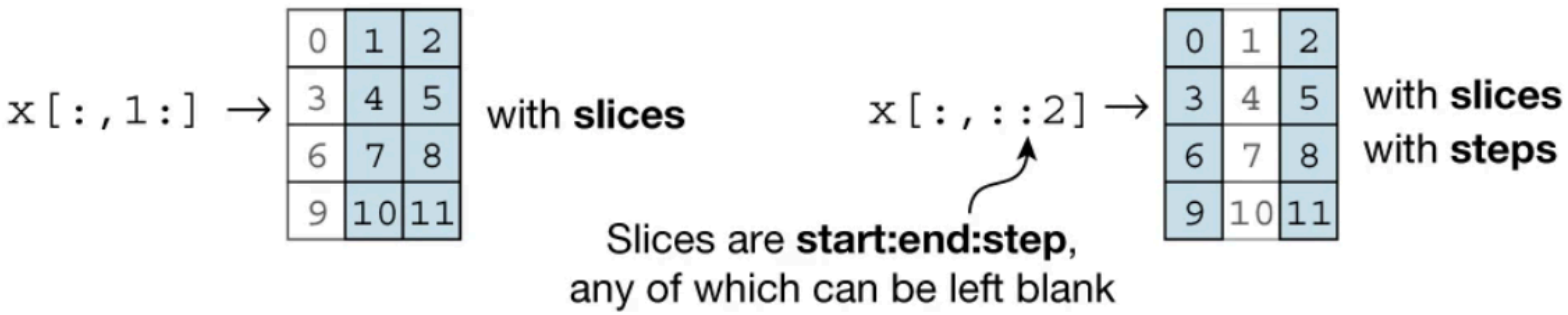
```
<ipython-input-40-a91e67fb2286>:4: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
  invalid_array = np.array([[1, 2, 3], [4, 5], [6]])
```

Array Attributes

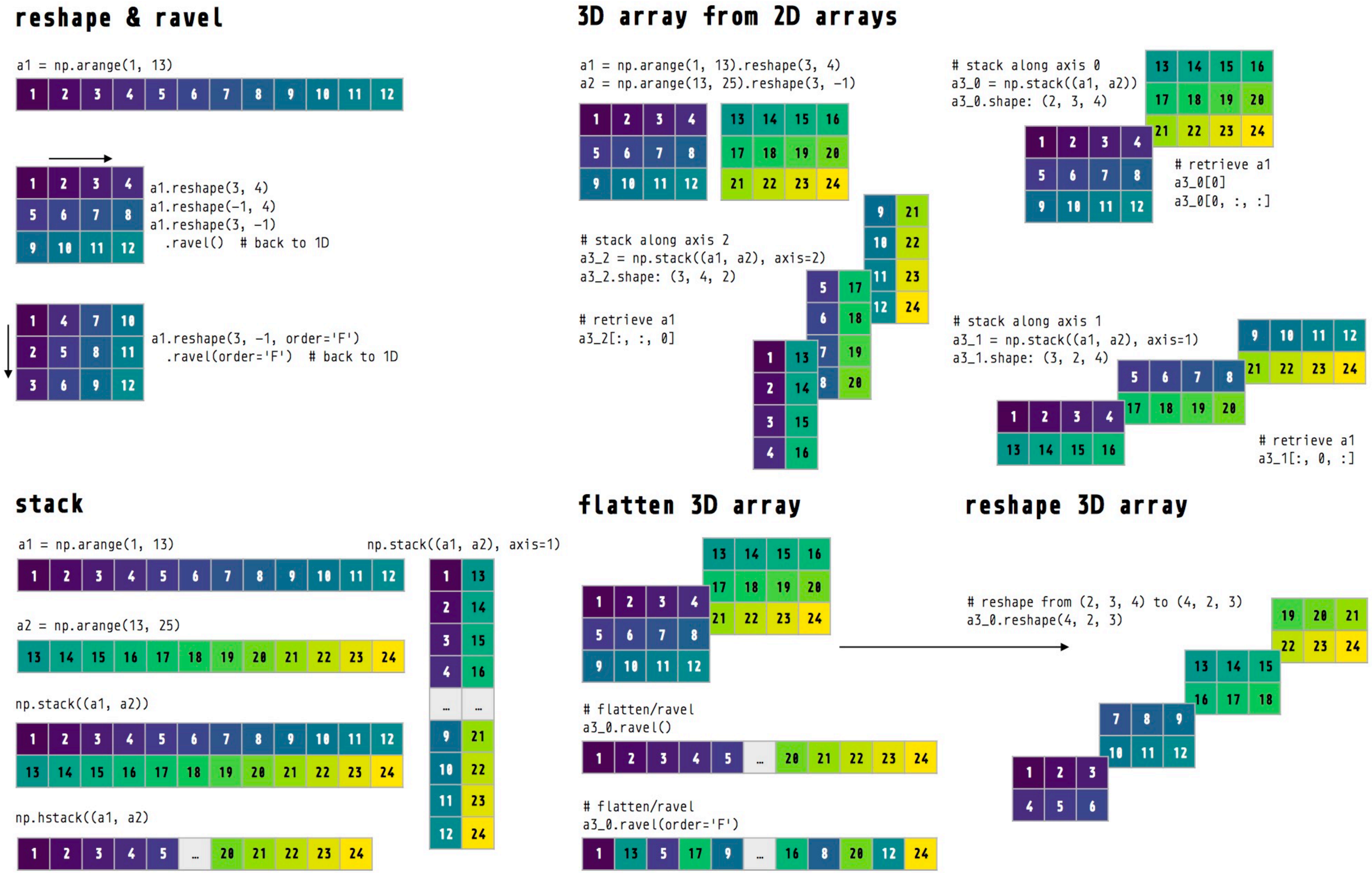


Indexing and slicing





Reshaping


reshape & ravel
a1 = np.arange(1, 13)
a1.reshape(3, 4)
a1.reshape(-1, 4)
a1.reshape(3, -1)
.ravel() # back to 1D
a1.reshape(3, -1, order='F')
.ravel(order='F') # back to 1D
3D array from 2D arrays
a1 = np.arange(1, 13).reshape(3, 4)
a2 = np.arange(13, 25).reshape(3, -1)
# stack along axis 0
a3_0 = np.stack((a1, a2))
a3_0.shape: (2, 3, 4)
retrieve a1
a3_0[0]
a3_0[0, :, :]
# stack along axis 2
a3_2 = np.stack((a1, a2), axis=2)
a3_2.shape: (3, 4, 2)
retrieve a1
a3_2[:, :, 0]
# stack along axis 1
a3_1 = np.stack((a1, a2), axis=1)
a3_1.shape: (3, 2, 4)
retrieve a1
a3_1[:, 0, :]
stack
a1 = np.arange(1, 13)
np.stack((a1, a2), axis=1)
a2 = np.arange(13, 25)
np.stack((a1, a2))
np.hstack((a1, a2))
flatten 3D array
# flatten/ravel
a3_0.ravel()
# flatten/ravel
a3_0.ravel(order='F')
reshape 3D array
# reshape from (2, 3, 4) to (4, 2, 3)
a3_0.reshape(4, 2, 3)

learning numpy axis rules

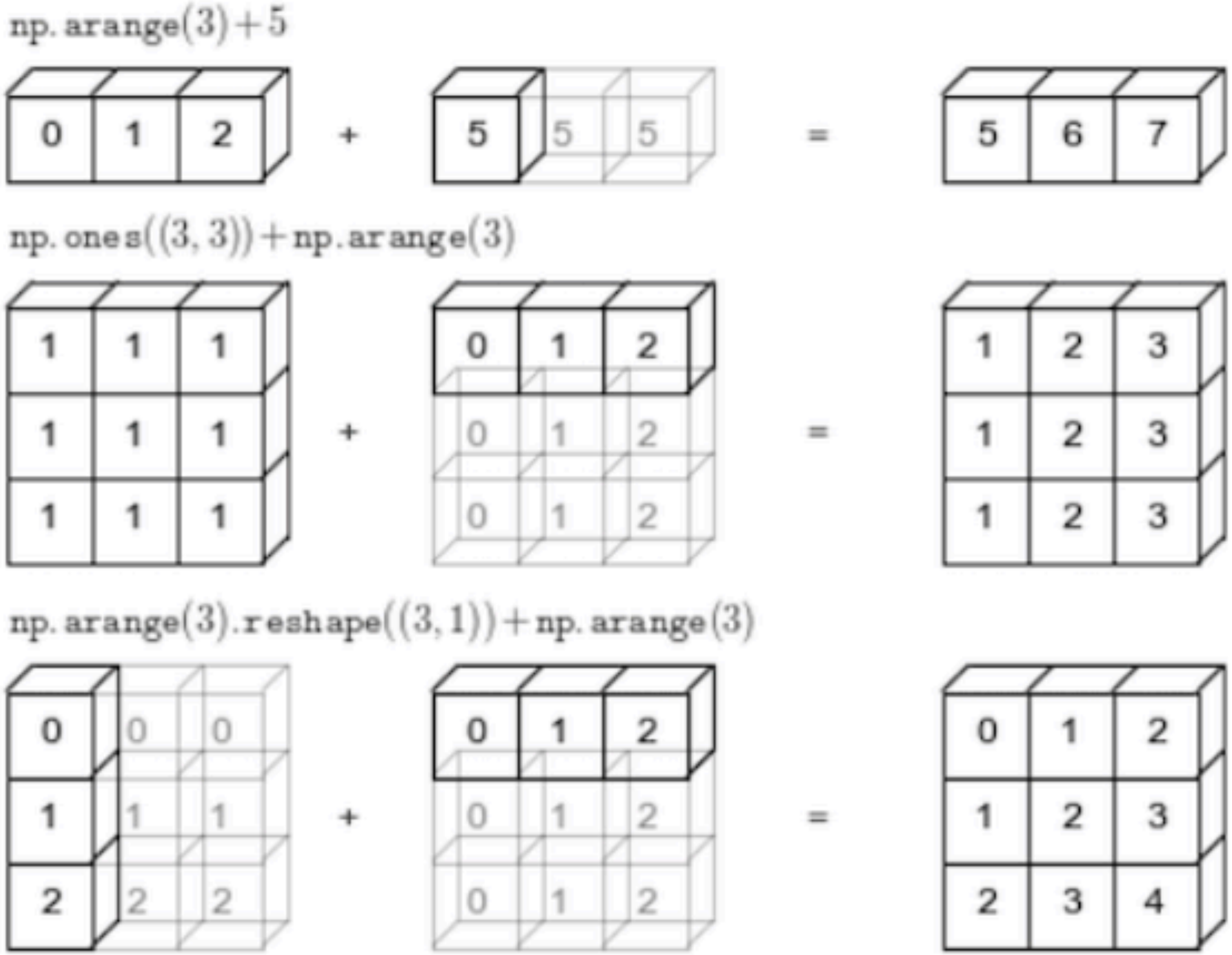
print output array's shape until one of the the axis values works out

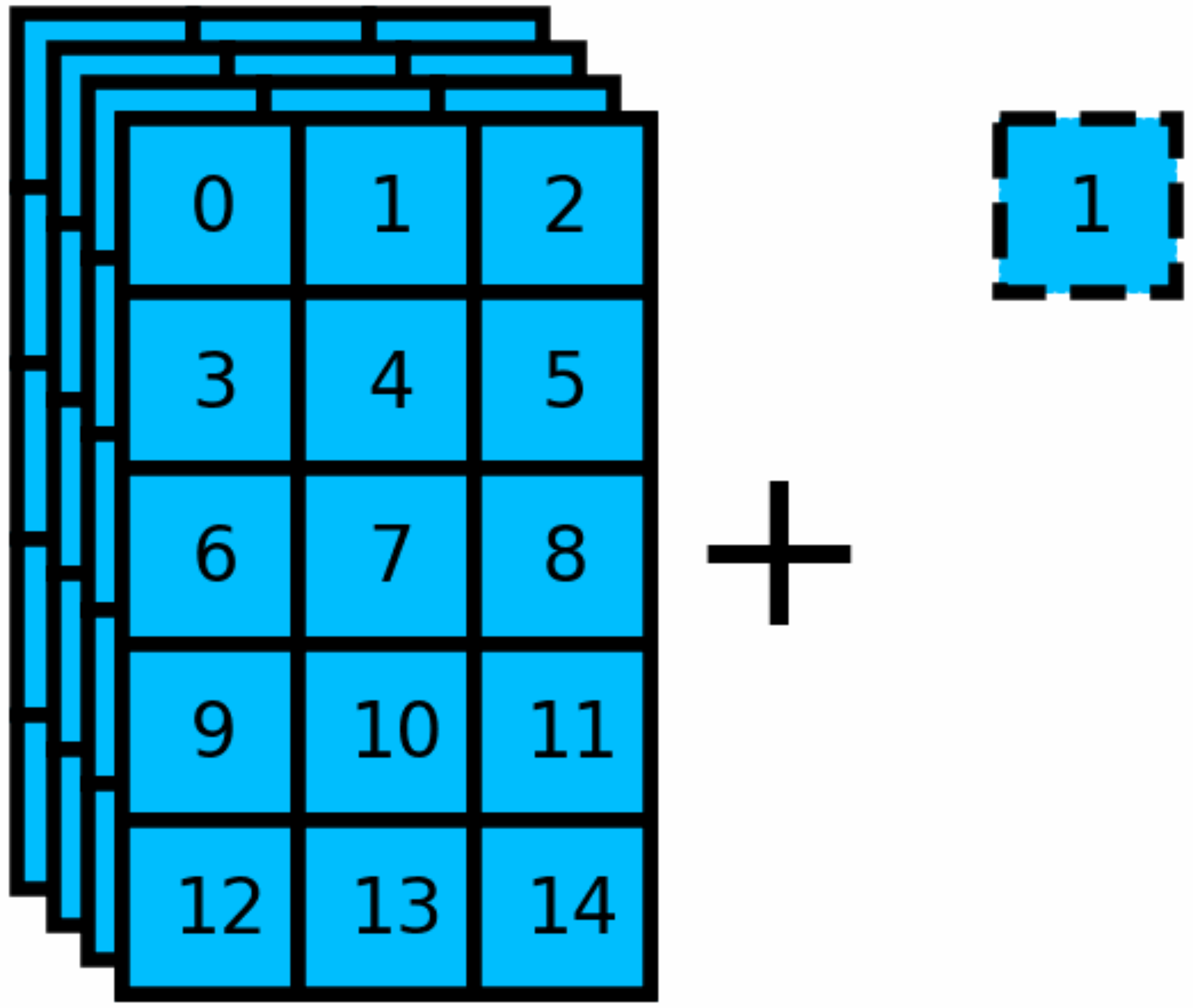
Vectorization

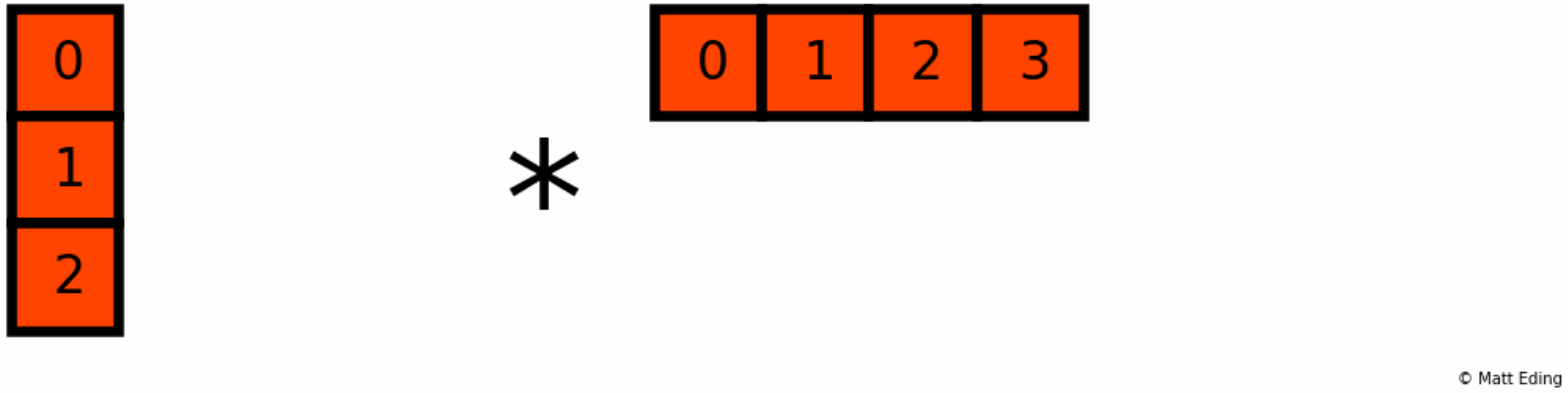
Vectorization refers to the practice of processing entire arrays, rather than their individual elements,

- Instead of using loops to perform operations on each element, NumPy allows for operations on the entire array, making computations faster and more readable.

- The ***np.vectorize()*** function in NumPy is a powerful tool that allows you to make any function work element-wise on an array, making it behave like a NumPy universal function.


np.arange(3)+5
np.ones((3,3))+np.arange(3)
np.arange(3).reshape((3,1))+np.arange(3)


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
+
1


0
1
2
*
0
1
2
3
© Matt Eding

The Essence of Itertools

- The central idea behind ***itertools*** is **lazy evaluation.**
- Python module for creating iterators for efficient looping.
- Provides fast, memory-efficient tools.

Infinite Iterators

- Count :
  - Generates consecutive numbers.
  - Ex- ***count(10, 10)*** starts from 10 and increments by 10.
- Cycle :
  - Loops over an iterable indefinitely.
  - Ex- ***cycle("ABC")***-> A, B, C, A, B, C, ...
- Repeat :
  - Yields an item repeatedly.
  - Ex - repeat("A", 3) -> A,A,A

Combinatoric Iterators

- Product :
    - Cartesian product of iterables.
    - Ex- For 2 sets of ‘AB’: product(‘AB’, repeat = 2)-> AA, AB, BA, BB
- Permutations :
    - Possible arrangements.
    - Ex- With 'ABC' of length 2: permutations(‘ABC’, 2) -> AB, AC, BA, BC, CA, CB
- Combinations :
    - Possible selections.
    - Ex - with 'ABC' of length 2: combinations(‘ABC’, 2) -> AB, AC, BC