What is Numpy?

Numpy allows users to build multi-dimensional arrays and high-level mathematical functions.

- Fast numerical computations
- Takes up lesser storage than lists
- High-level math functions

```
In [20]: import numpy as np


In [21]: harray = np.array(range(100))


In [22]: type(harray)
Out[22]: numpy.ndarray


In [23]: harray.shape
Out[23]: (100,)


In [24]: harray.size
Out[24]: 100
```

Why do we need Numpy?

1

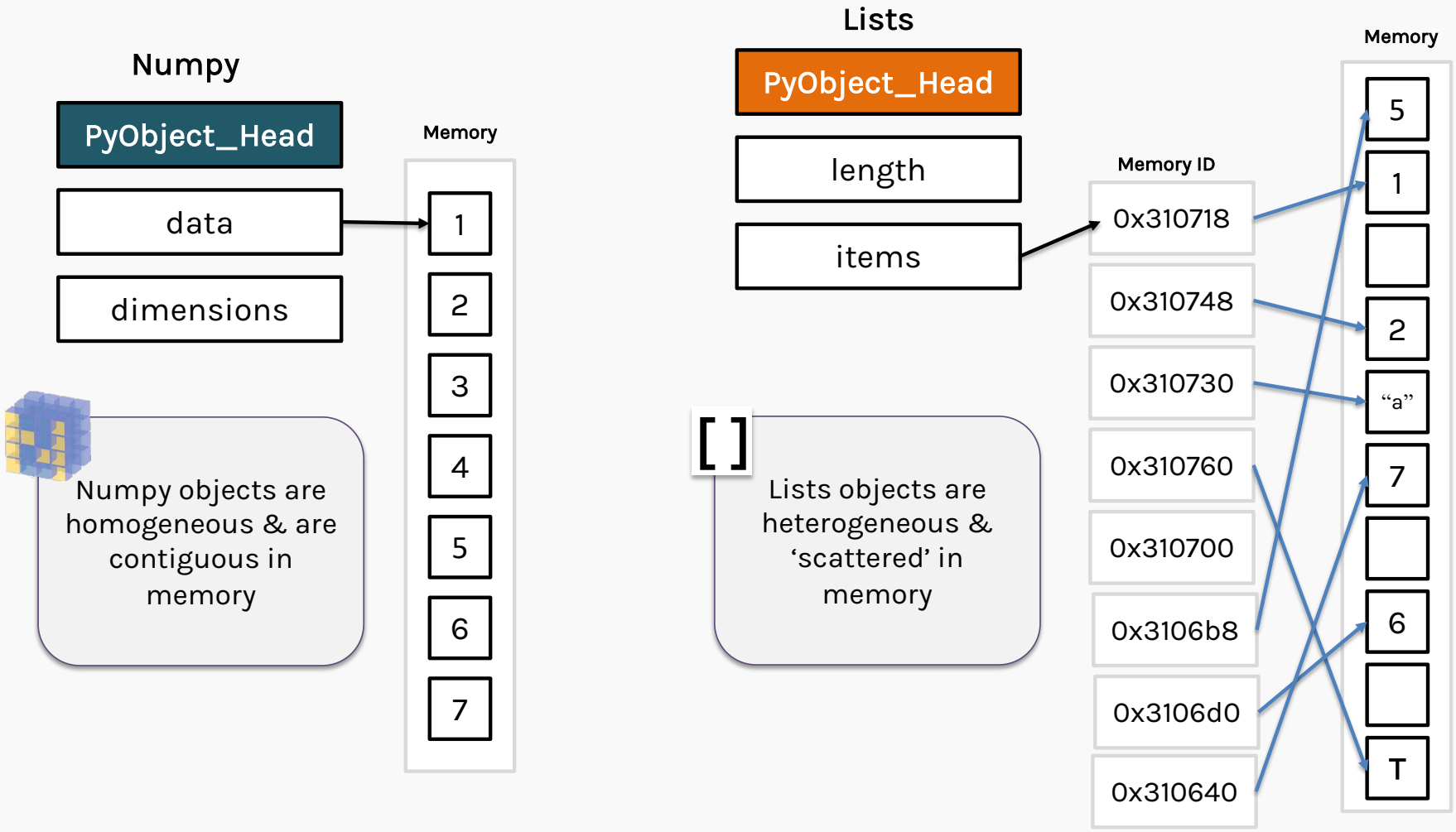
MEMORY EFFICIENT

1. Densely packed in memory
2. Data type pre-defined
3. Optimized for access

```
In [34]: harlist = [1,2,3,4,5,6]

In [35]: numpyarray =
np.array(harlist,dtype='int8')

In [36]: #helper function to
calculate size of python list
    ...: size(harlist)
Out[36]: 168

In [37]: numpyarray.nbytes
Out[37]: 6
```

Jake Vanderplas post on numpy and python

Lists vs. Numpy



Why is Numpy so fast ?

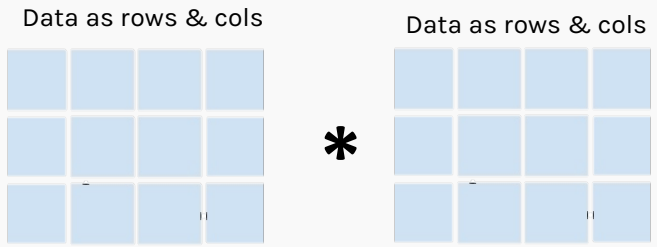
FAST

1. Use universal functions

   (implemented in C)

2. No type conversion

3. No type checking




```
In [1]: %%time
   ...: hadamard = []
   ...: for i in range(len(A)):
   ...:     mini = []
   ...:     for j in range(len(A[0])):
   ...:         mini.append(A[i][j]*B[i][j])
   ...:     hadamard.append(mini)
   ...:
CPU times: user 505 ms, sys: 13.1 ms, total:
518 ms
Wall time: 517 ms
In [2]: %time A*B
CPU times: user 2.83 ms, sys: 3.83 ms, total:
6.66 ms
Wall time: 5.36 ms
```

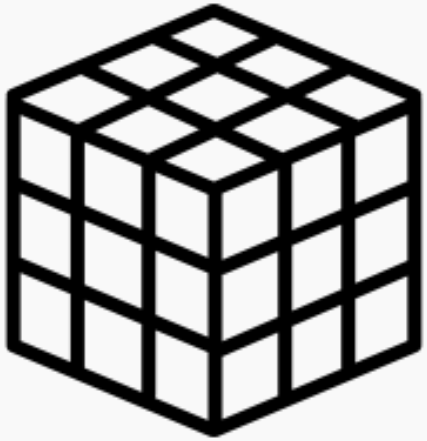
Numpy benefits

MATH-LIKE

- Universal operations
- Intuitive syntax
- No loops needed

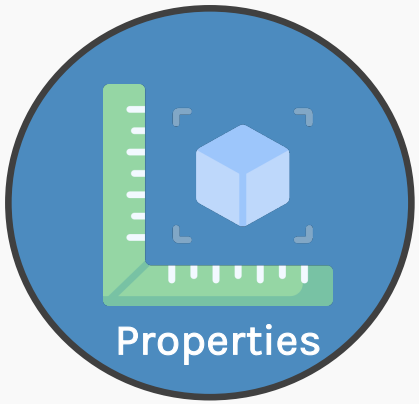
```
In [1]: # Using pythonic loops
   ...: hadamard = []
   ...: for i in range(len(A)):
   ...:     mini = []
   ...:     for j in range(len(A[0])):
   ...:         mini.append(A[i][j]*B[i][j])
   ...:     hadamard.append(mini)


In [2]:# Using Numpy universal functions
      hadamard = A*B
```

Let's begin

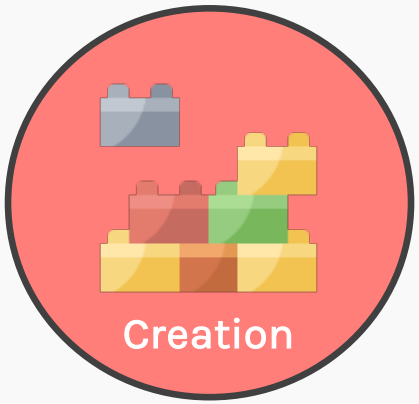


np.array

np.array



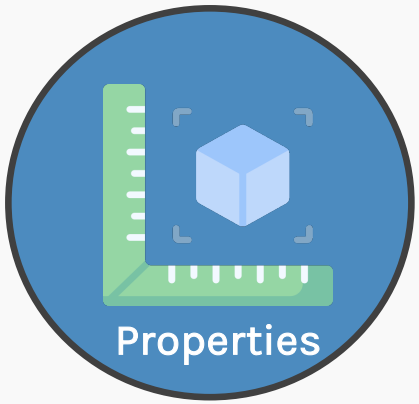
- Shape
- size
- axis



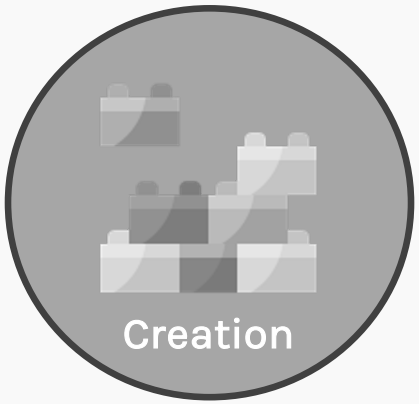
- zeros,ones
- arange,linspace
- vstack,hstack

- Indexing & Slicing
- Reshape
- Broadcasting

np.array



- Shape
- size
- axis

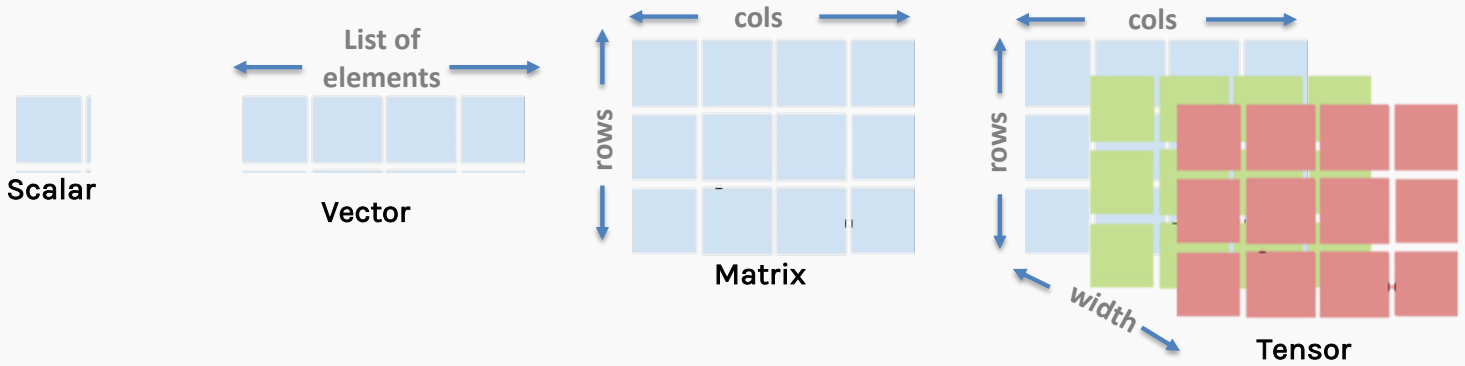


- zeros,ones
- arange,linspace
- vstack,hstack

- Indexing & Slicing
- Reshape
- Broadcasting

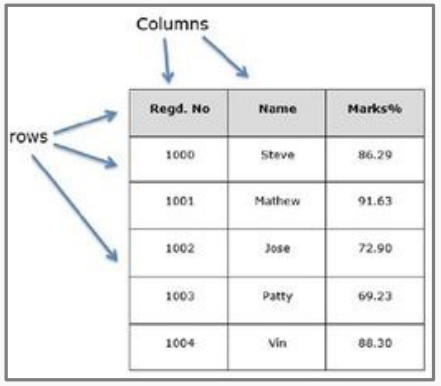
Properties & Attributes

The **array** is a central datastructure of the Numpy library.



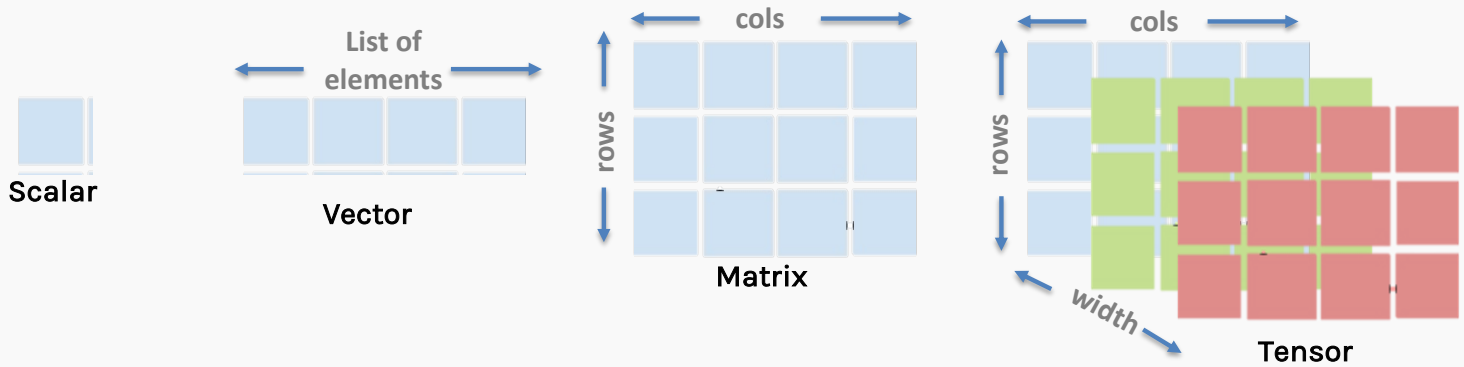
EXAMPLES

Your age	list of ages of PyDS team	A pandas DataFrame	A picture of the Great Astronomer!
25	25 29 22 27		



Properties & Attributes

The **ndarray** is a central datastructure of the Numpy library.



EXAMPLES

Your age

```
In [5]: age = 34
In [6]:
np.array(age)
Out[6]: array(34)
```

list of ages of PyDS team

```
In [7]: age_list =
[25,22,24,27]
In [8]:
np.array(age_list)
Out[8]:
array([25, 22, 24,
27])
```

A pandas DataFrame

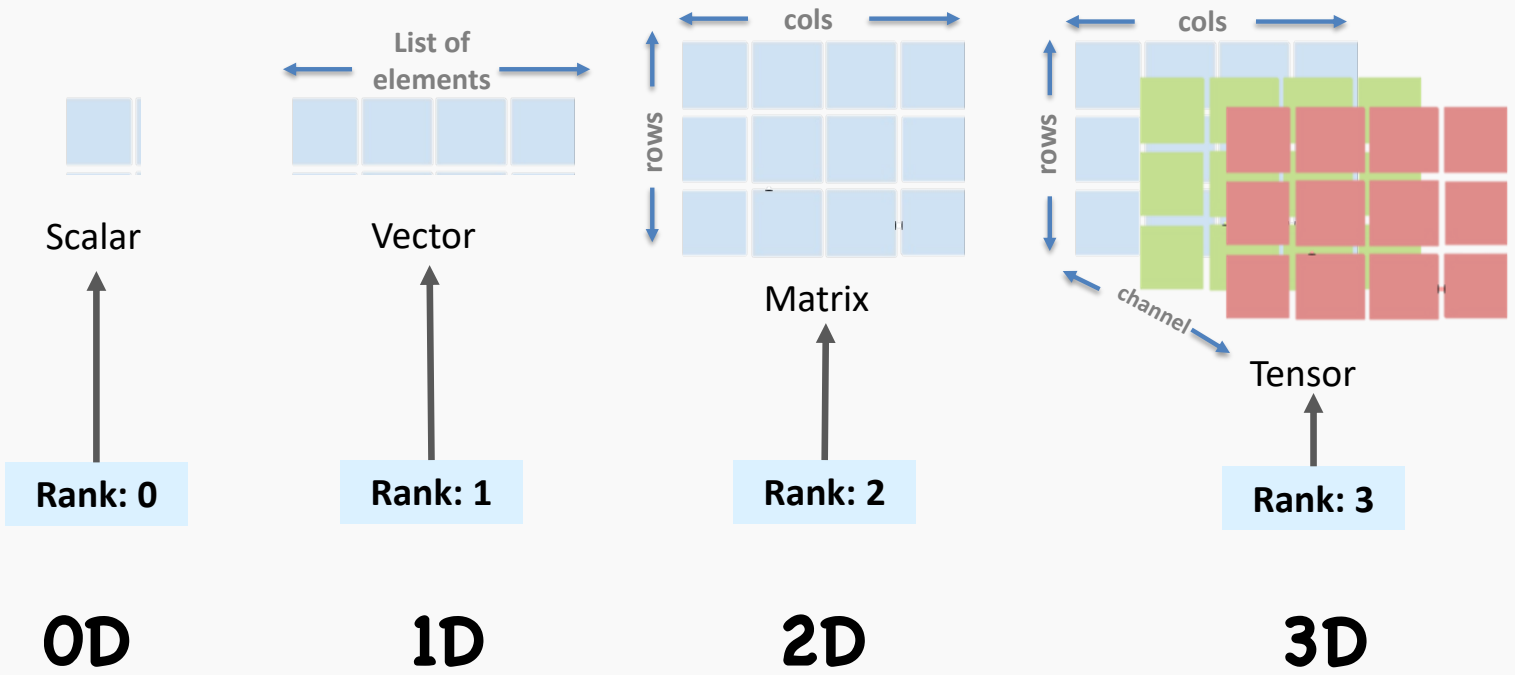
```
In [12]: a = np.array(
[[1, 2, 3, 4],
[5, 6, 7, 8],
[1, 2, 4, 3],
[4, 5, 6, 3]])
```

A picture of the Great Astronomer!

```
In [14]: a = np.array
(
[[[1,2,3,4],[5,6,7,8]
,[1,2,4,3],[1,2,4,3]]
,[[1,2,3,4],[5,6,7,8]
,[1,2,4,3],[1,2,4,3]]
,
[[1,2,3,4],[5,6,7,8],
[1,2,4,3],[1,2,4,3]],
] )
```

RANK

Properties & Attributes

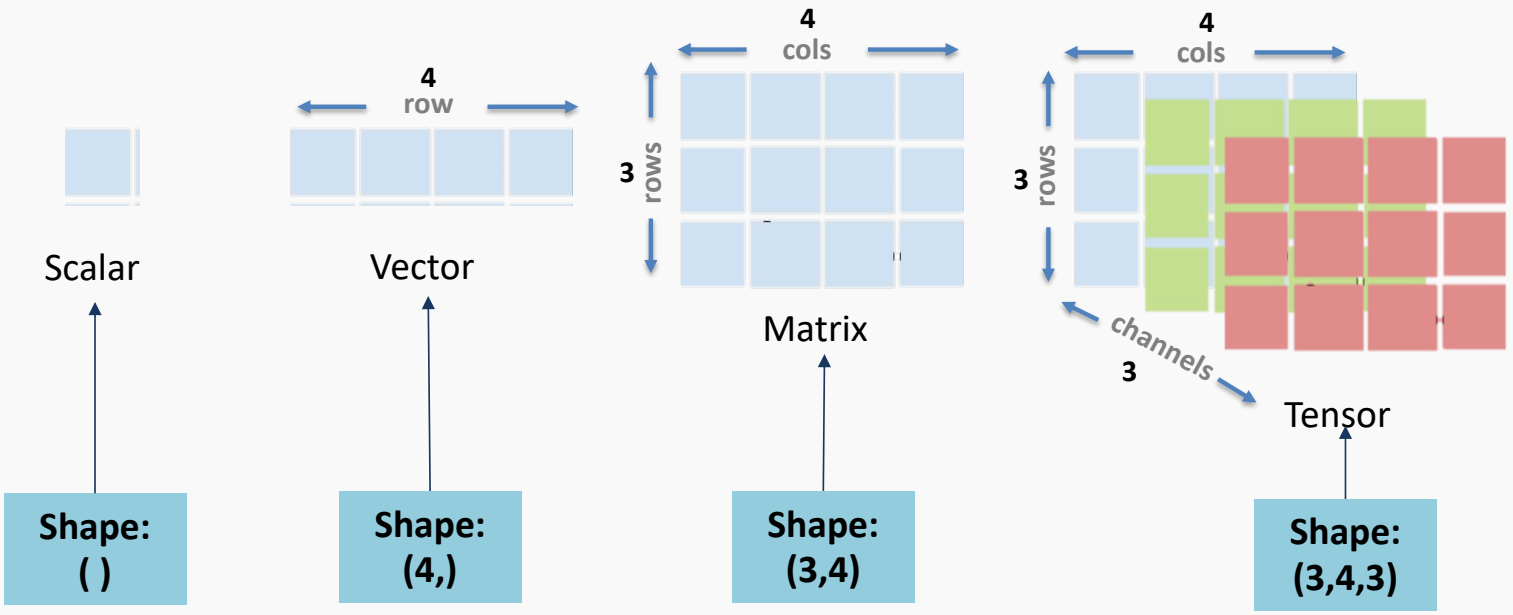


The rank of the array is the number of dimensions.
`ndarray.ndim` will tell you the number of axes, or dimensions, of the array.

SHAPE

Properties & Attributes

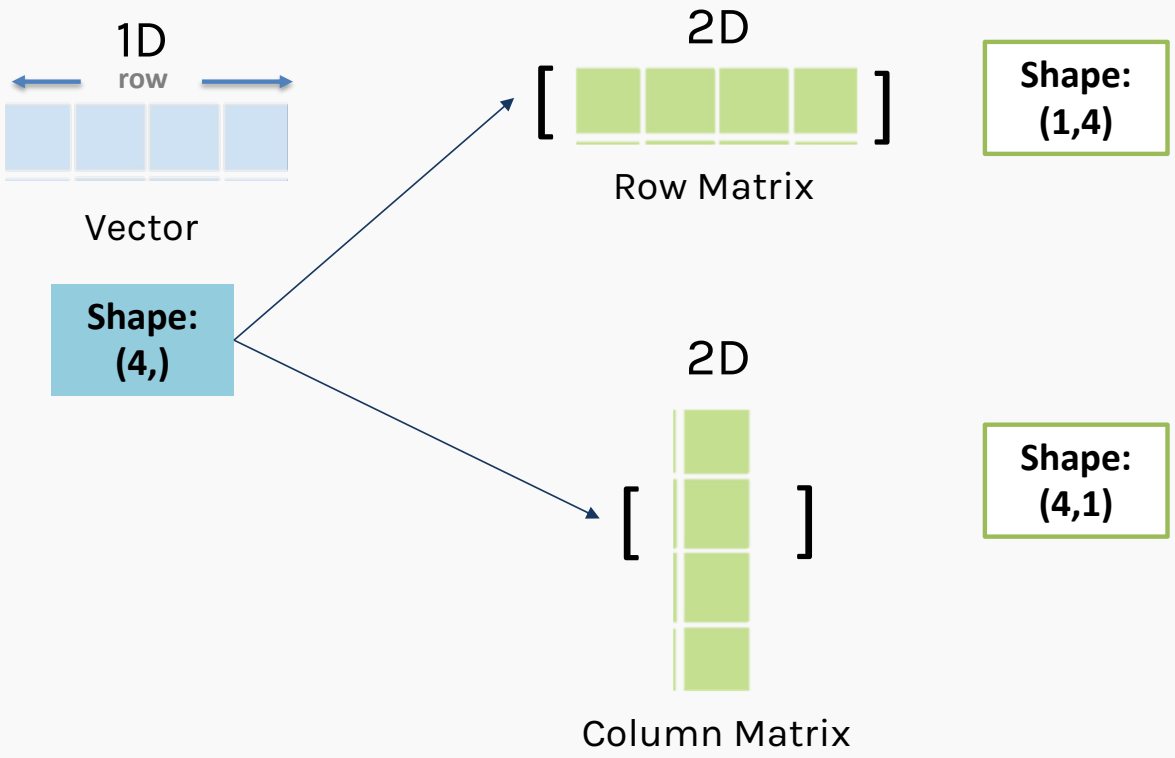
BEFORE AXIS The **ndarray** is a central datastructure of the Numpy library.



The shape of the array is the number of elements present in each dimension. **`ndarray.shape`** will display a tuple of integers that indicate the number of elements stored along each dimension of the array.

Properties & Attributes

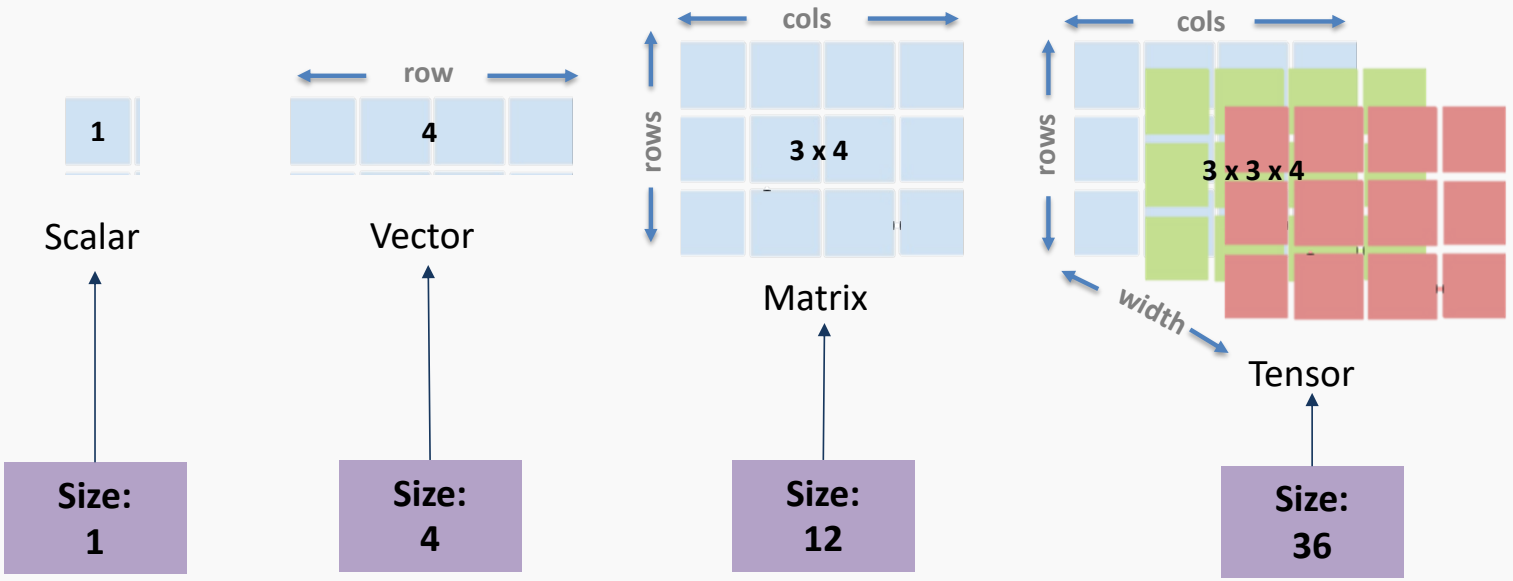
Important to Note! A vector can be converted into a column or row matrix.



It is very easy to get confused with (n,) and (1,n) or (1,n). Remember, that the main difference lies in the DIMENSION.

Properties & Attributes

The **ndarray** is a central datastructure of the Numpy library.

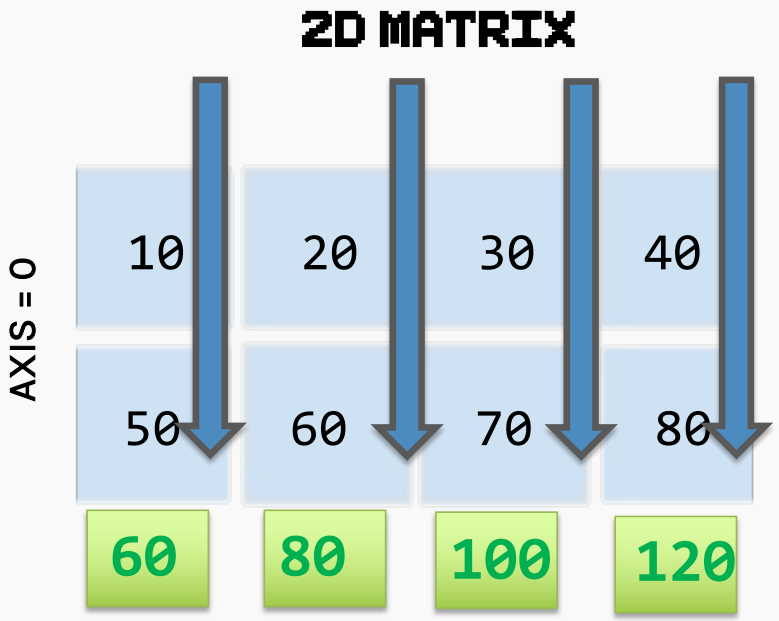


The size of the array is the number of elements present. It is the product of elements in all dimensions. **`ndarray.size`** will tell you the total number of elements of the array.

AXIS

Properties & Attributes

An **axis** is akin to a dimension.
For a 2-dimensional array, there are 2 axes: vertical and horizontal.



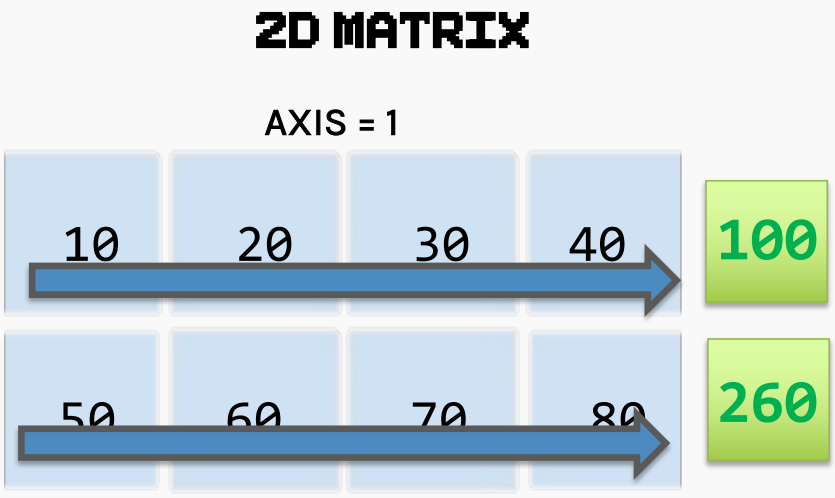
AXIS = 0

Specifying axis = 0 implies "perform operation **vertically**"

```
In [29]: # Given the following 2-dimensional array
    ...: values = np.array([[10, 20, 30, 40],
    ...:                    [50, 60, 70, 80]])
    ...: # Axis=0 # along each "column"


In [30]: np.sum(values,axis=0)
Out[30]: array([ 60,  80, 100, 120])
```

Properties & Attributes

An **axis** is akin to a dimension.
For a 2-dimensional array, there are 2 axes: vertical and horizontal.



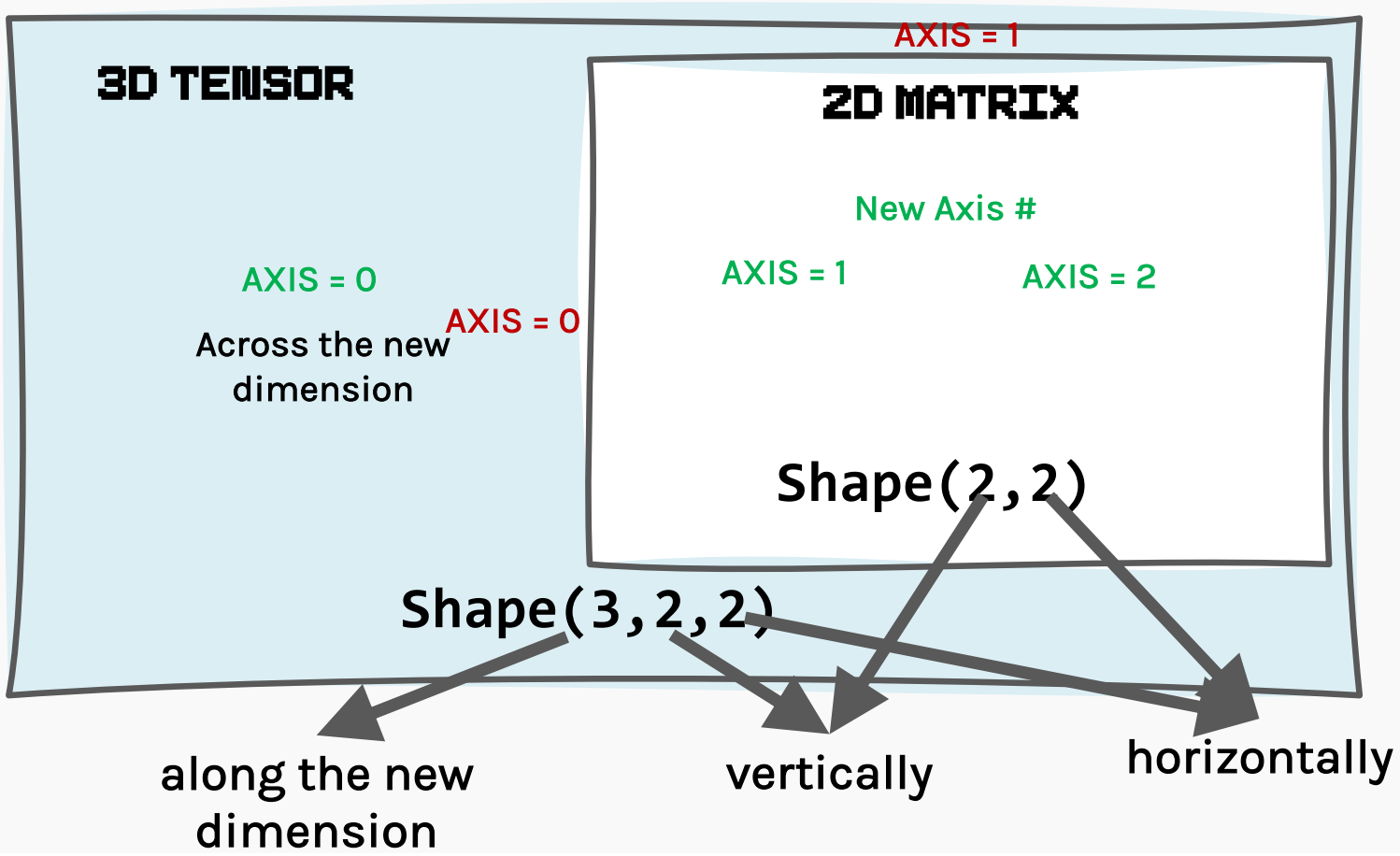
AXIS = 1

Specifying axis = 1 implies "perform operation **horizontally**

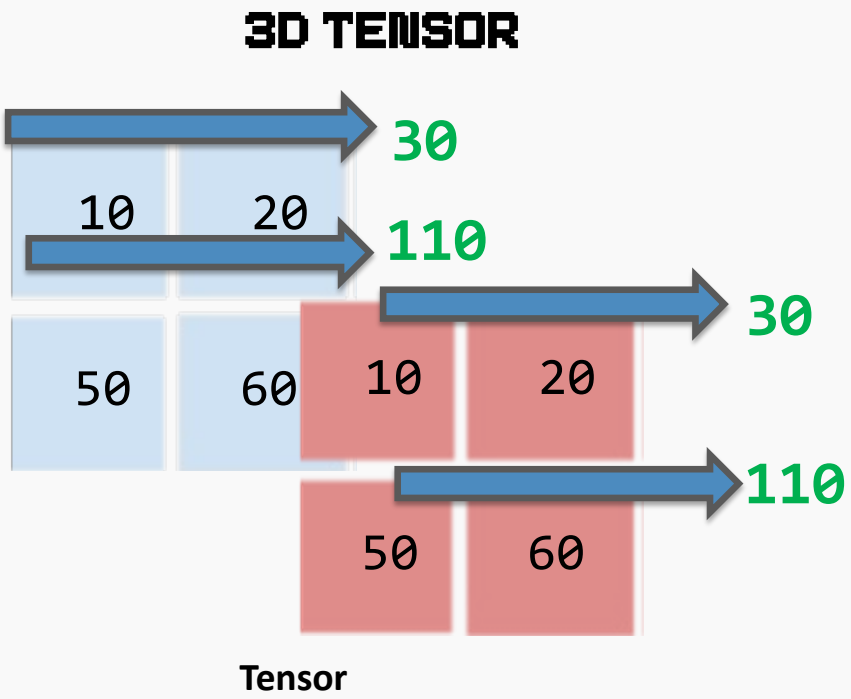
```
In [32]: # Given the following 2-dimensional array
    ...: values = np.array([[10, 20, 30, 40],
    ...:                    [50, 60, 70, 80], ])
    ...:
    ...: # Axis=1 # along each "row"
    ...: np.sum(values, axis=1)
Out[32]: array([100, 260])
```

1) Properties/ Attributes of Numpy objects

For more than 2D cases



Properties & Attributes

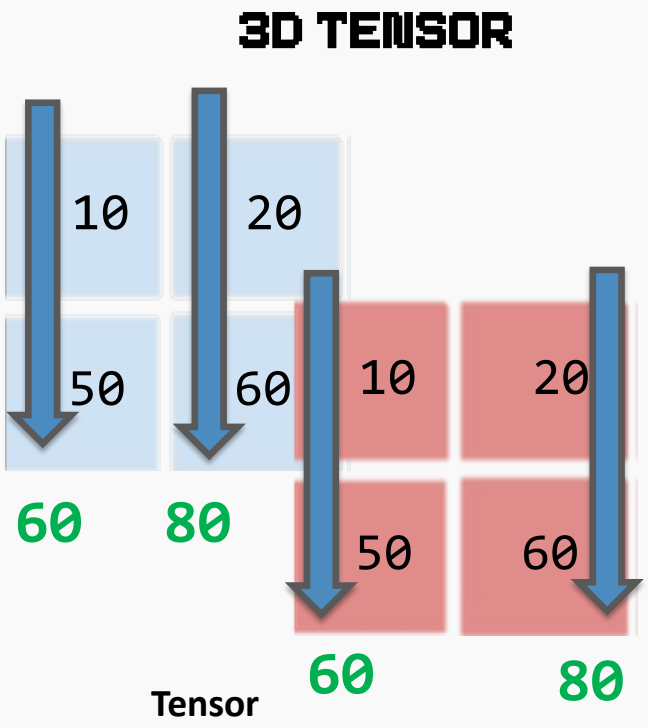


For more than 2D cases

Specifying axis = 2 implies "**perform operations horizontally**"

```
In [50]: # Given the following 2-dimensional array
    ...: values = np.array([[[10, 20],
    ...:                     [50, 60]],
    ...:
    ...:                    [[10, 20],
    ...:                     [50, 60]]])


In [51]: # Axis=2 along each value inside the innermost
layer
    ...: np.sum(values,axis=2)
Out[51]:
array([[ 30, 110],
       [ 30, 110]])
```

Properties & Attributes

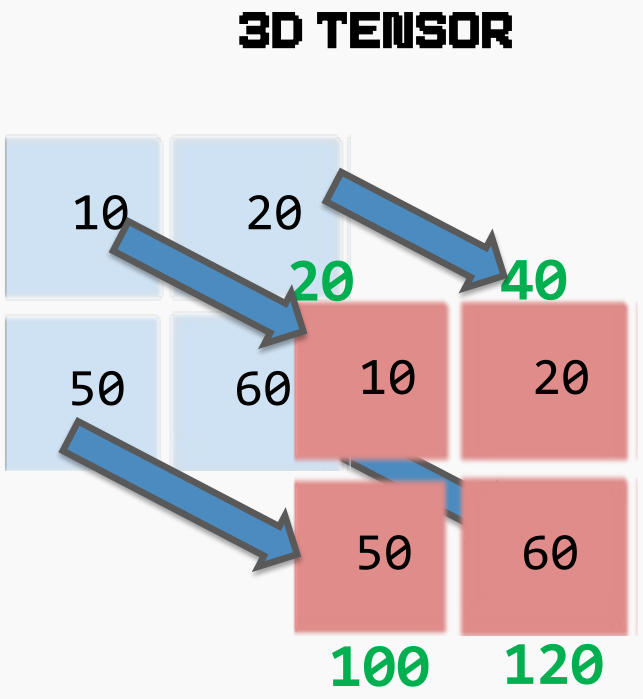


For more than 2D cases

Specifying axis = 1 implies "**perform operations vertically**"

```
In [50]: # Given the following 2-dimensional array
    ...: values = np.array([[[10, 20],
    ...:                     [50, 60]],
    ...:
    ...:                    [[10, 20],
    ...:                     [50, 60]]])


In [51]: # Axis=2 along each value inside the innermost
layer
    ...: np.sum(values,axis=1)
Out[51]:
array([[ 60, 80],
       [ 30, 80]])
```

Properties & Attributes

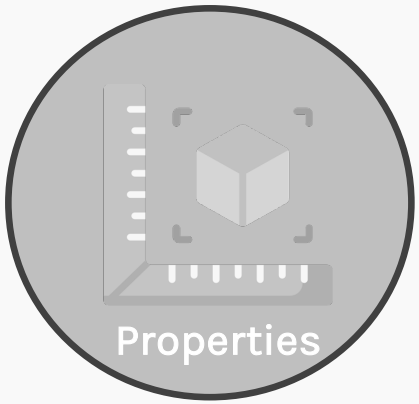


For more than 2D cases

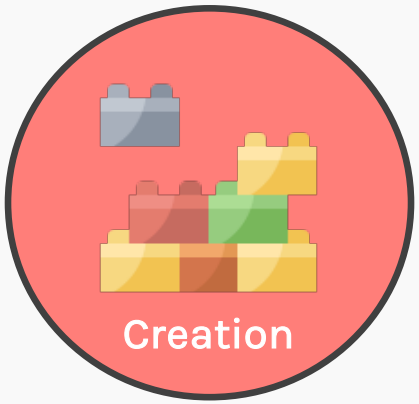
Specifying axis = 0 implies "**perform operations along the new dimension**"

```
In [50]: # Given the following 2-dimensional array
    ...: values = np.array([[[10, 20],
    ...:                     [50, 60]],
    ...:
    ...:                    [[10, 20],
    ...:                     [50, 60]]])


In [51]: # Axis=2 along each value inside the innermost
layer
    ...: np.sum(values,axis=0)
Out[51]:
array([[ 20, 40],
       [ 100, 120]])
```

np.array

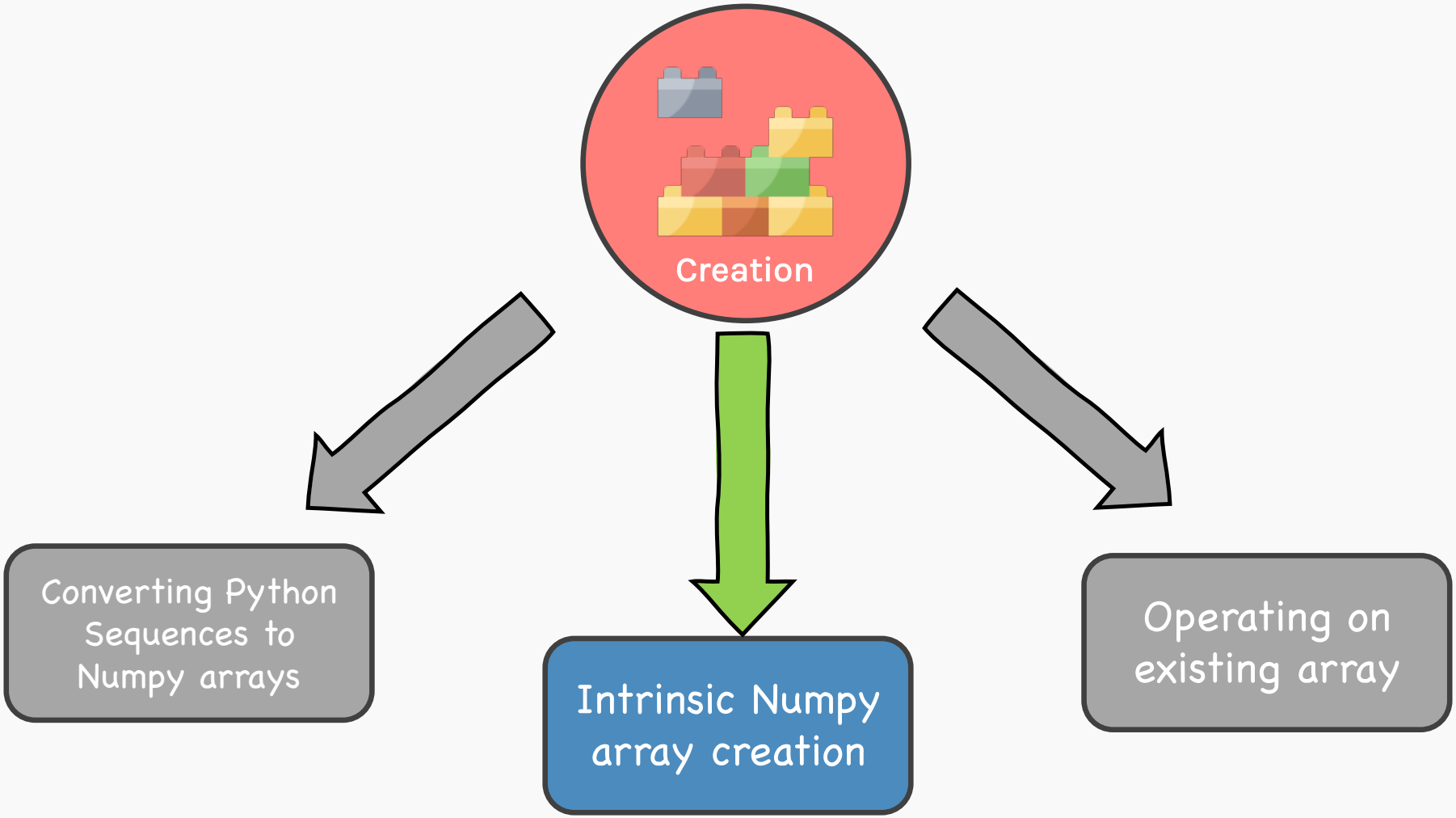


- ~~Shape~~
- ~~size~~
- ~~axis~~



- zeros,ones
- arange,linspace
- vstack,hstack

- Indexing & Slicing
- Reshape
- Broadcasting


Creation
Converting Python Sequences to Numpy arrays
Intrinsic Numpy array creation
Operating on existing array

Python Sequences to Numpy arrays

Converting Python Sequences to Numpy arrays

numpy.array(*object, dtype=None*)

```
In [59]: harlist = [1,2,3,4,5]


In [60]: harray = np.array(harlist)


In [61]: harray
Out[61]: array([1, 2, 3

In [62]: harray =
np.array(harlist,dtype='int8')


In [63]: harray
Out[63]: array([1, 2, 3, 4, 5], dtype=int8)
```

You can explicitly mention the data type at the time of initialization

Python Sequences to Numpy arrays

Intrinsic Numpy array creation

```
In [78]: np.zeros((2,2))
Out[78]:
array([[0., 0.],
       [0., 0.]])
In [79]: np.ones((2,2))
Out[79]:
array([[1., 1.],
       [1., 1.]])
In [80]: np.eye(2)
Out[80]:
array([[1., 0.],
       [0., 1.]])
In [81]: np.diag((-1,1))
Out[81]:
array([[-1,  0],
       [ 0,  1]])
In [82]: np.arange(1,10,2)
Out[82]: array([1, 3, 5, 7, 9])

In [84]: np.linspace(1,9,5)
Out[84]: array([1., 3., 5., 7., 9.])
```

Intrinsic Numpy array creation functions

np.zeros

create an array filled with 0 values with the specified shape

`np.zeros(`*`shape, dtype=float`*`)`

```
In [89]: np.zeros((3,3))
Out[89]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

np.ones

create an array filled with 1 values with the specified shape

`np.ones(`*`shape, dtype=float`*`)`

```
In [90]: np.ones((3,3))
Out[90]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Intrinsic Numpy array creation functions

np.eye

defines a 2D identity matrix

numpy.eye(n, m) where n-> #rows & m ->#cols

```
In [5]: np.eye(3)
Out[5]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

np.diag

defines a square 2D array with given values along the diagonal

numpy.diag(v, k = 0) where v -> array, k-> int

```
In [16]: np.diag([1,2,3])
Out[16]:
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Intrinsic Numpy array creation functions

np.arange

creates arrays with regularly incrementing values

`np.arange([start, stop, step], dtype=None)`

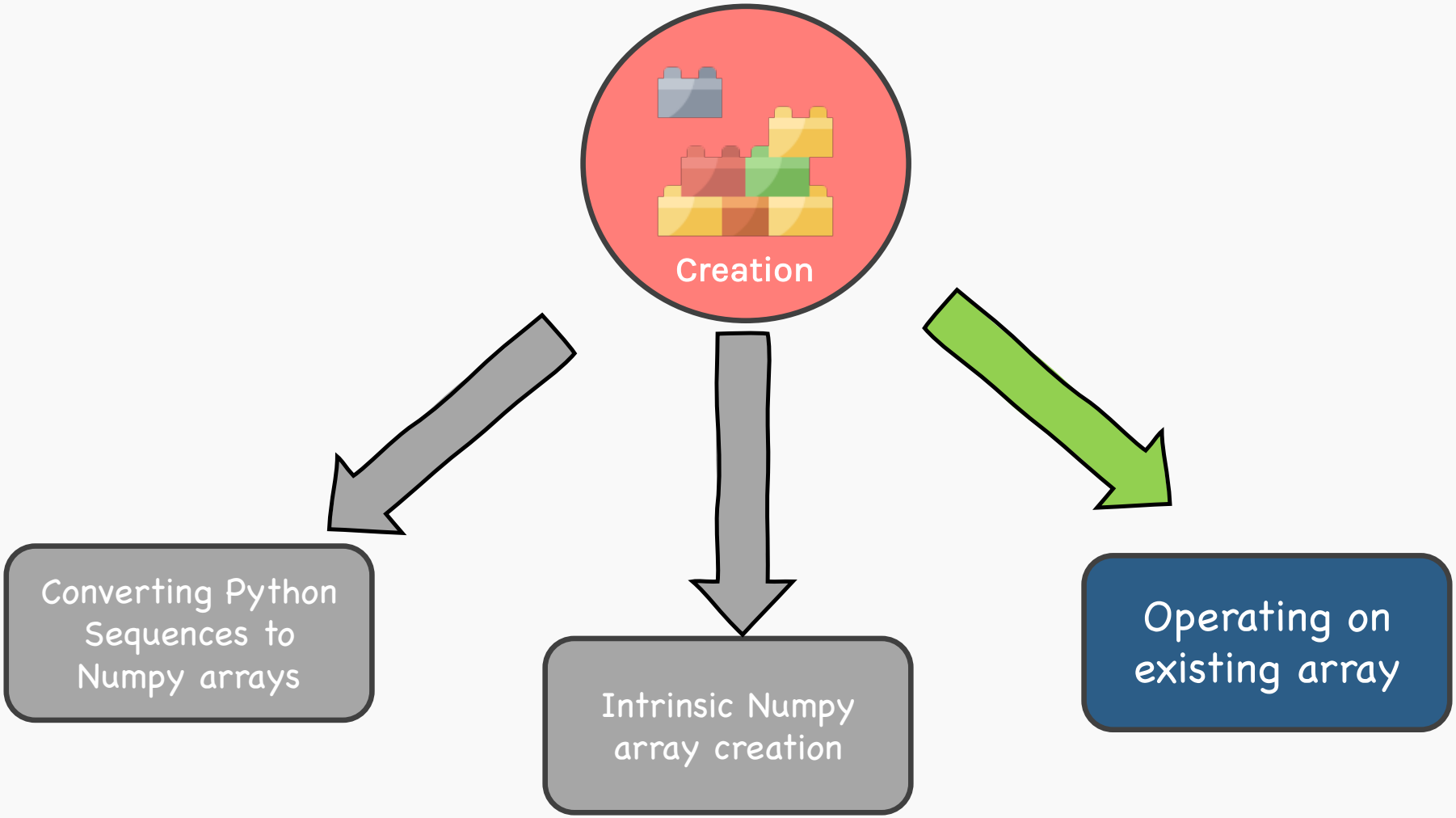
```
In [23]: np.arange(0,10,1)
Out[23]: array([0, 1, 2, 3, 4,
5, 6, 7, 8, 9])


In [24]: np.arange(0,10,2)
Out[24]: array([0, 2, 4, 6, 8])
```

np.linspace

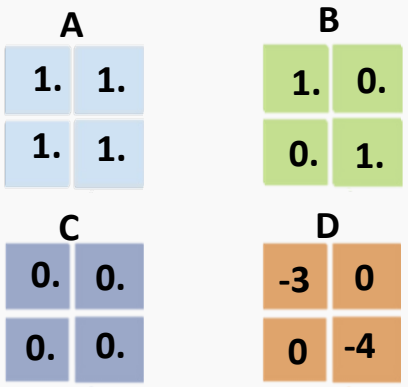
spaced equally between the specified beginning and end values.

`np.linspace(start, stop, num=50, endpoint=True, dtype = None)`

```
In [90]: np.ones((3,3))
Out[90]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

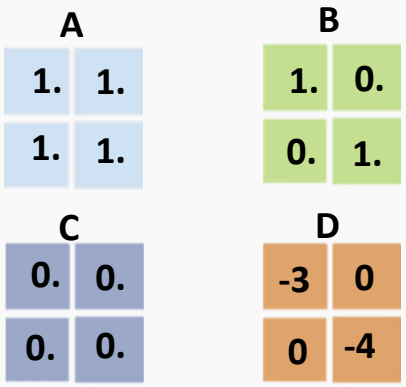

Creation
Converting Python Sequences to Numpy arrays
Intrinsic Numpy array creation
Operating on existing array

Joining Existing Arrays




```
In [61]: a = np.ones((2,2))

In [62]: b = np.eye(2)

In [63]: c = np.zeros((2,2))

In [64]: d = np.diag((-3,-4))
```

Joining Existing Arrays



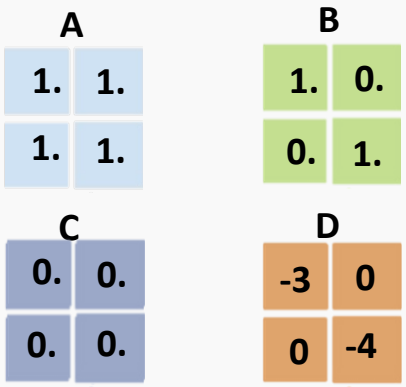
np.hstack()

Stacks matrices along axis=1

```
In [61]: a = np.ones((2,2))

In [62]: b = np.eye(2)

In [63]: c = np.zeros((2,2))

In [64]: d = np.diag((-3,-4))
```

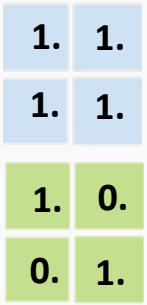
```
In [84]: np.hstack((a,b))
Out[84]:
array([[1, 1, 1, 0],
       [1, 1, 0, 1]])
```

Joining Existing Arrays




```
In [61]: a = np.ones((2,2))

In [62]: b = np.eye(2)

In [63]: c = np.zeros((2,2))

In [64]: d = np.diag((-3,-4))
```

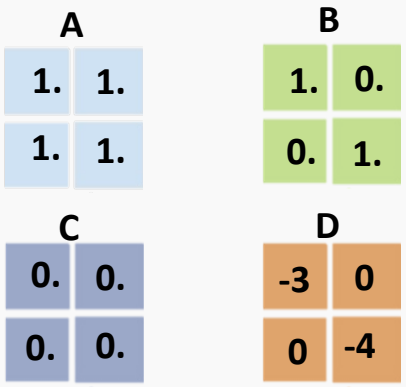
np.vstack()



Stacks matrices along axis=0

```
In [85]: np.vstack((a,b))
Out[85]:
array([[1, 1],
       [1, 1],
       [1, 0],
       [0, 1]])
```

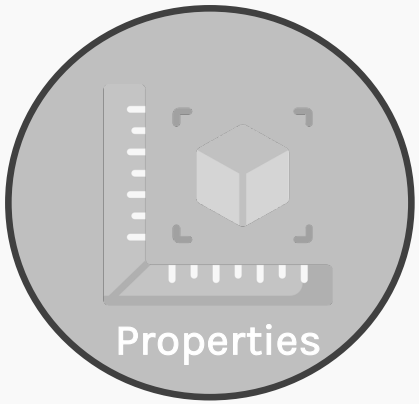
2.3) Replicating/Mutating & Joining Existing Arrays



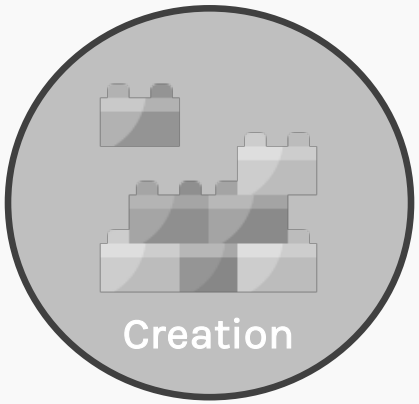

```
In [61]: a = np.ones((2,2))

In [62]: b = np.eye(2)

In [63]: c = np.zeros((2,2))

In [64]: d = np.diag((-3,-4))
```


```
In [100]: ab = np.hstack((a,b))

In [101]: cd = np.hstack((c,d))

In [102]: block = np.vstack((ab,cd))
Out[103]:
array([[ 1,  1,  1,  0],
       [ 1,  1,  0,  1],
       [ 0,  0, -3,  0],
       [ 0,  0,  0, -4]])
```

np.array

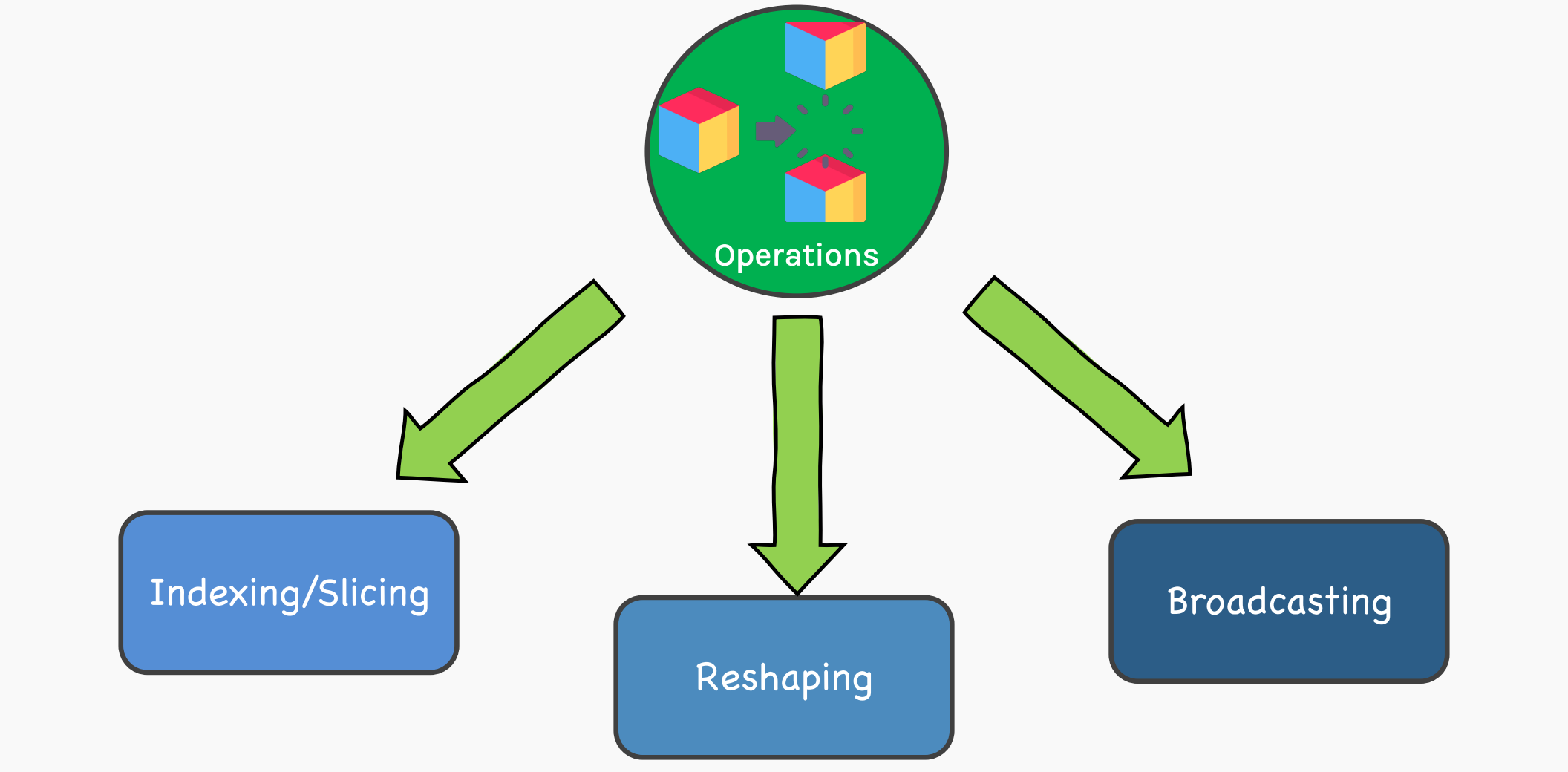


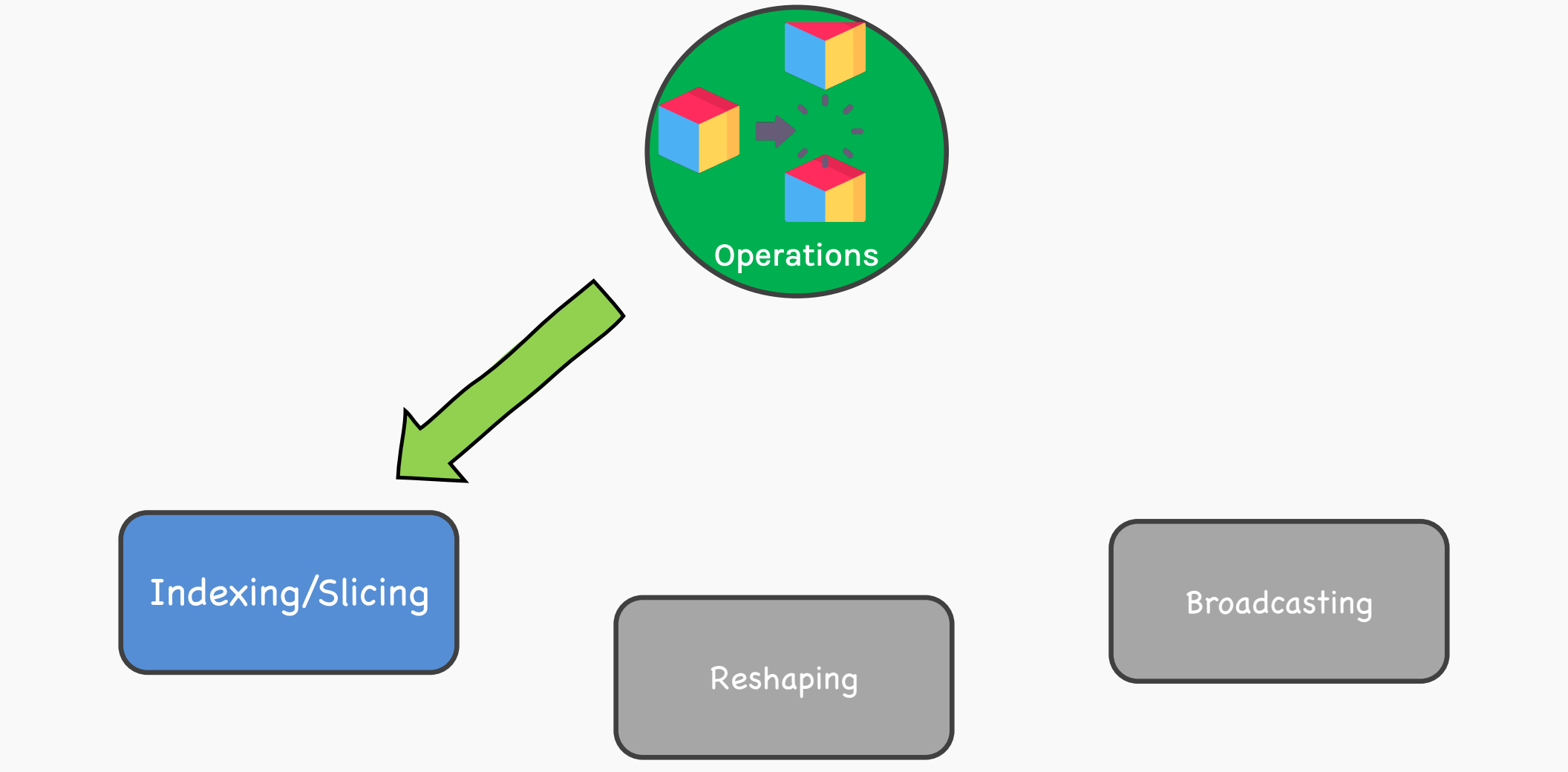
- ~~Shape~~
- ~~size~~
- ~~axis~~



- ~~zeros,ones~~
- ~~arange,linspace~~
- ~~vstack,hstack~~

- Indexing & Slicing
- Reshape
- Broadcasting


Operations
Indexing/Slicing
Reshaping
Broadcasting


Operations
Indexing/Slicing
Reshaping
Broadcasting

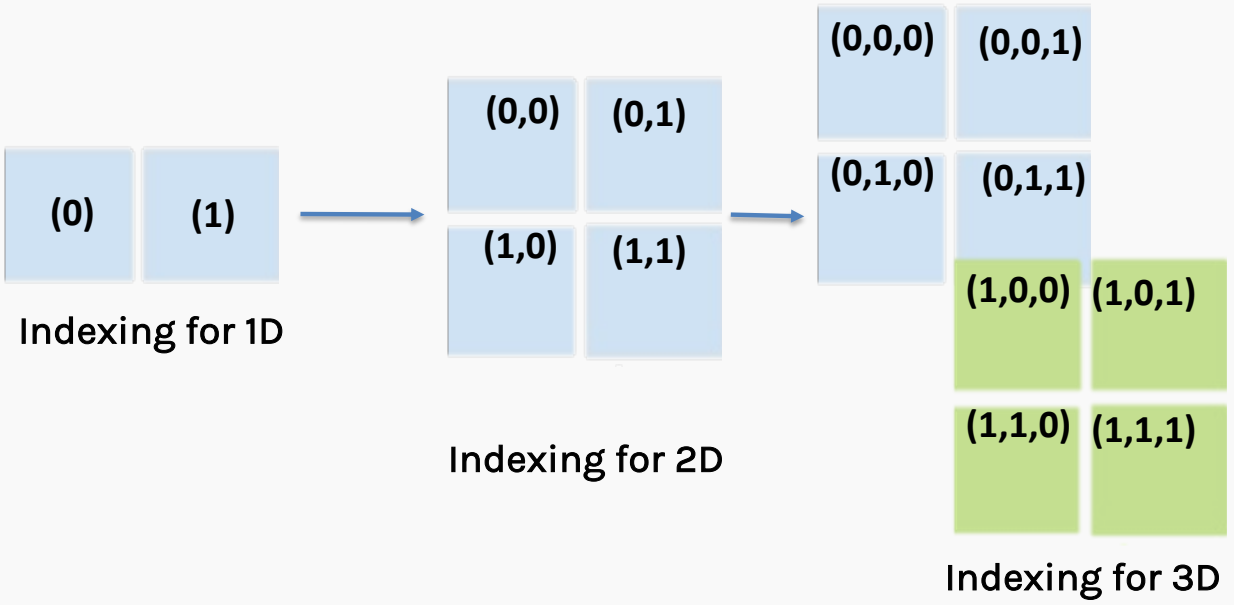
Indexing

Using general rules of Python indexing you can access elements within a ndarray.

General Syntax

```
arr_obj[row index, col index]
```

Indexing/Slicing



Remember, negative index is allowed, it simply means offset from end of the list.

Slicing

Indexing/Slicing

General Syntax

arr_obj[start: stop: step]

One-dimensional Slicing

You can access all data in an array dimension by specifying the slice ‘ : ’ with no indexes.

```
In [16]: #Accessing all elements
    ...: data = np.array([11, 22, 33, 44,55])
    ...: print(data[:])
[11 22 33 44 55]
```

```
#Accessing last two elements
In [14]: data[-2:]
Out[14]: array([44, 55])

#Accessing last two elements
In [15]: data[-2::-1]
Out[15]: array([44, 33, 22, 11])
```

Slicing

Indexing/Slicing

General Syntax

arr_obj[start: stop: step]

Two-Dimensional Slicing

```
    ...: data = np.array(
    ...:                 [
    ...:                     [11, 22, 33],
    ...:                     [44, 55, 66],
    ...:                     [77, 88, 99]
    ...:                 ])
In [20]: #Slice the data
    ...: X,y = data[:,:-1], data[:,-1]

In [21]: X
Out[21]:
array([[11, 22],
       [44, 55],
       [77, 88]])

In [22]: y
Out[22]: array([33, 66, 99])
```

Get all rows until the last column (excluded)

Get all the rows of the last column

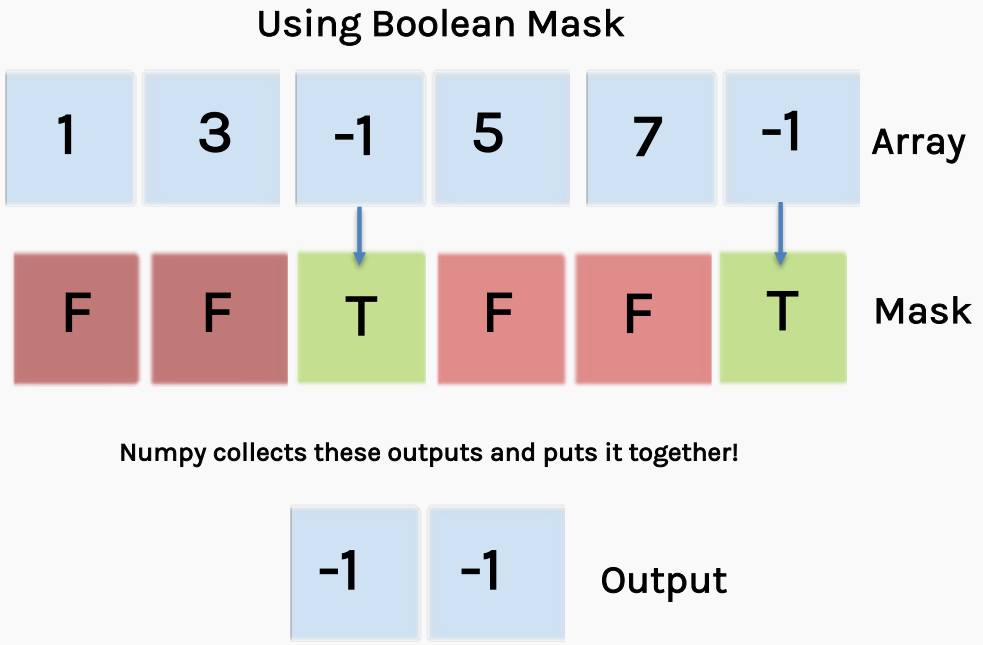
11	22	33
44	55	66
77	88	99

Numpy Indexing

Indexing/Slicing

Boolean Masks

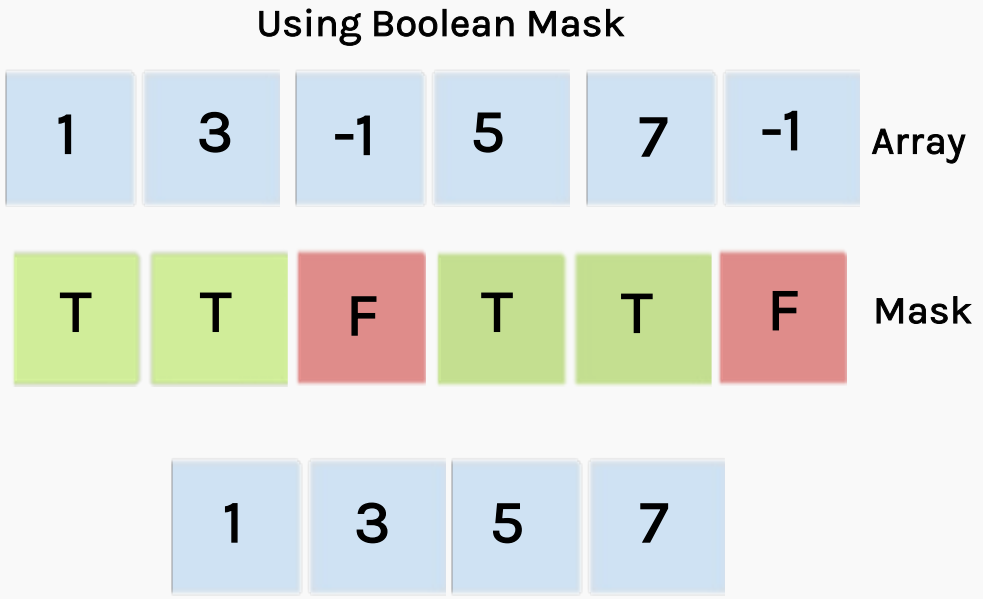
```
In [26]: # Define numpy array
    ...: x = np.array([1,3,-1, 5, 7, -1])
    ...:
    ...: # create mask
    ...: mask = (x < 0)
    ...:
    ...: #accessing/slicing elements using mask
    ...: print(x[mask])
[-1 -1]

In [27]: mask
Out[27]: array([False, False,  True, False,
False,  True])
```



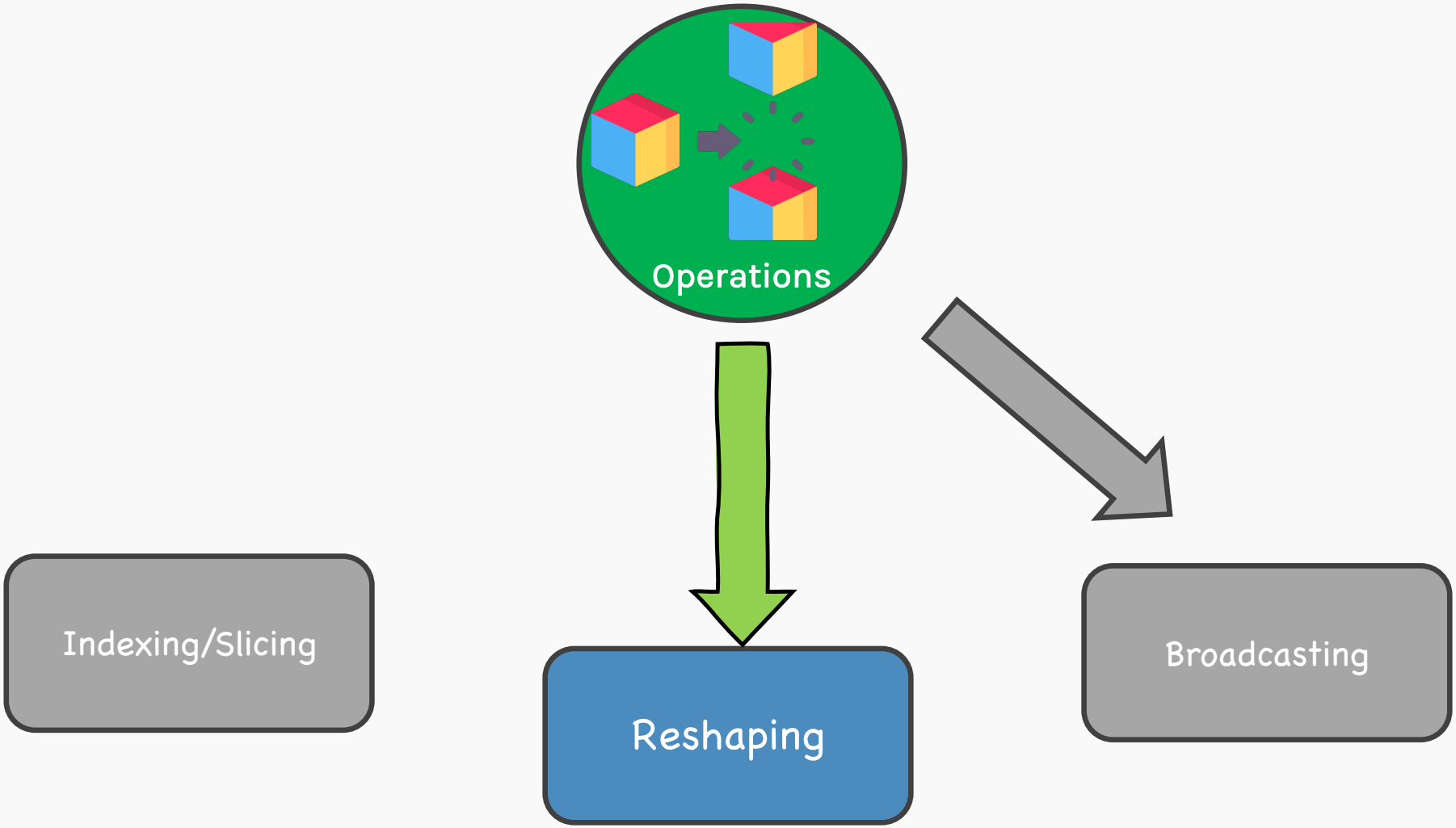
You can even get the opposite boolean array by using the tilde (~) operator

Boolean Masks

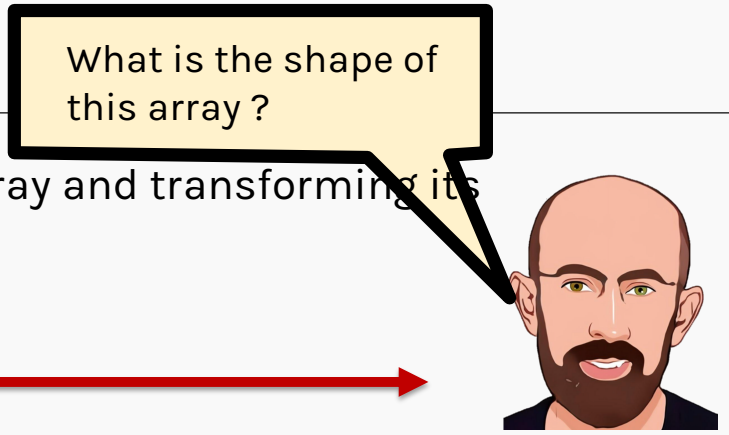
```
In [26]: # Define numpy array
    ...: x = np.array([1,3,-1, 5, 7, -1])
    ...:
    ...: # create mask
    ...: mask = (x > 0)
    ...:
    ...: #accessing/slicing elements using mask
    ...: print(x[mask])
([1, 3, 5, 7])

In [27]: mask
Out[27]: array([True,True, False, True,
True,  False])
```



Numpy does not allow use of conditional operators with numpy arrays. Use bitwise operators instead of comparisions

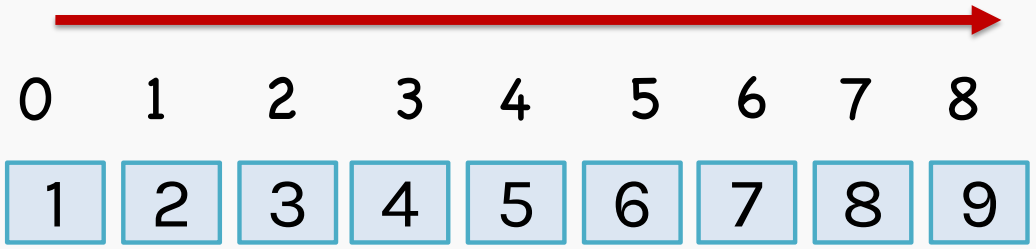

Operations
Indexing/Slicing
Reshaping
Broadcasting

Reshaping



Reshaping implies taking the elements of an existing array and transforming its dimensions into a new one.

Reshaping




```
In [47]: a = np.arange(1,10,1)

In [48]: a
Out[48]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [49]: a.shape
Out[49]: (9,)
```

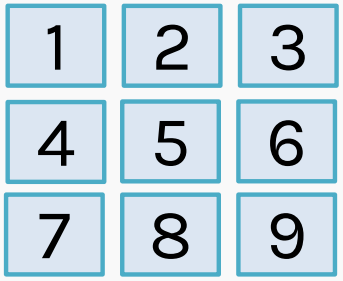
General Syntax

new_array = old_array.reshape((ndim1, ndim2…))

Reshaping

Reshaping implies taking the elements of an existing array and transforming its dimensions into a new one.

Reshaping




```
In [54]: b = a.reshape(3,3)
In [55]: b
Out[55]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

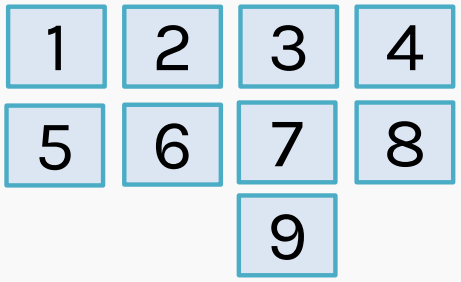
General Syntax

new_array = old_array.reshape((ndim1, ndim2…))

Reshaping

Reshaping implies taking the elements of an existing array and transforming its dimensions into a new one.

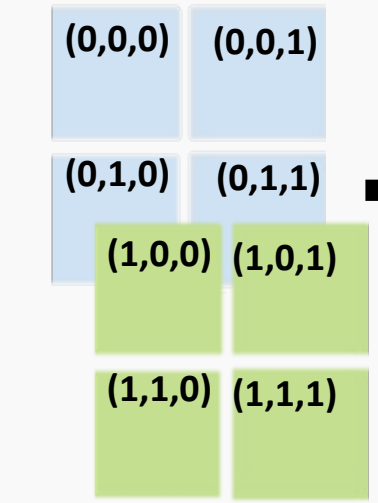
Reshaping




```
In [57]: b = a.reshape(4,2)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-57-cec4c3749f64> in <module>
----> 1 b = a.reshape(4,2)

ValueError: cannot reshape array of size 9 into shape (4,2)
```

Reshaping

Reshaping implies taking the elements of an existing array and transforming its dimensions into a new one.



Size – 8; Shape – (2,2,2)

The size of the new array MUST be the same as the PREV array.

a.reshape((2,4))

a.reshape((8,1))

a.reshape((1,8))

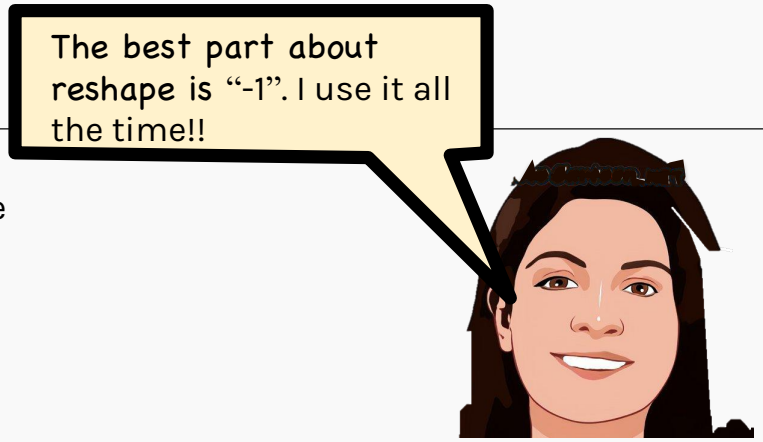
a.reshape((4,2))

a.reshape((4,2))

.shape & .reshape() are going to be your best friends

Reshaping

Numpy allows you to use "-1" when you don't want to spend your time trying to figure out compatible dimensions.



Let the size of the array be M

Reshape parameters given be $a, b, c, \ldots, -1, n$

Missing dimension will be assumed as $\frac{\mathrm{M}}{a*b*\cdots*n}$

GAME TIME!!

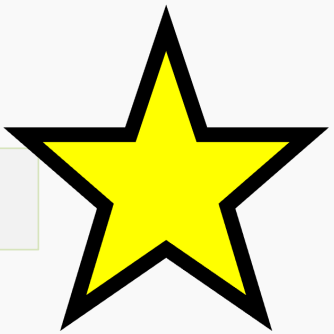
Reshaping

What will the new shape be?

1	2	3	4
5	6	7	8
9	10	11	12

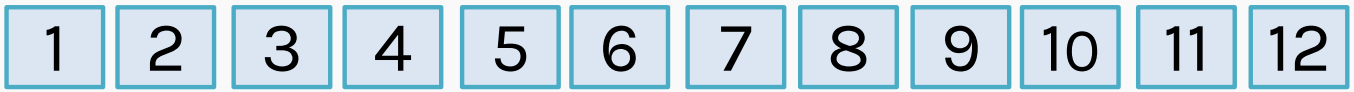
`reshape(-1)`

(12,)



Reshaping

`reshape(-1)`

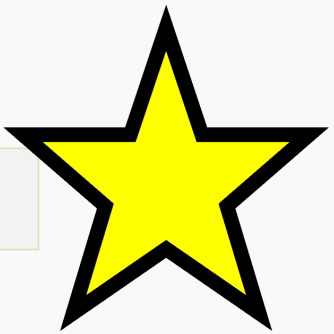


Reshaping

What will the new shape be?

1	2	3	4
5	6	7	8
9	10	11	12

`reshape(-1,1)` (12,1)



Reshaping

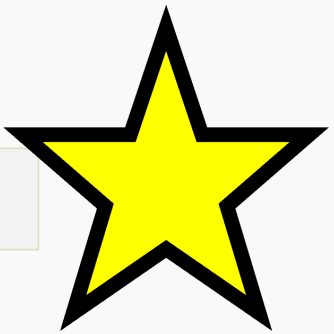
`reshape(-1,1)`

Reshaping

What will the new shape be?

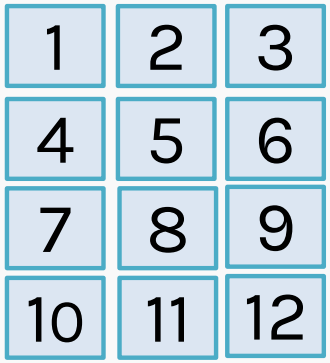
1	2	3	4
5	6	7	8
9	10	11	12

`reshape(4,-1)` (4,3)



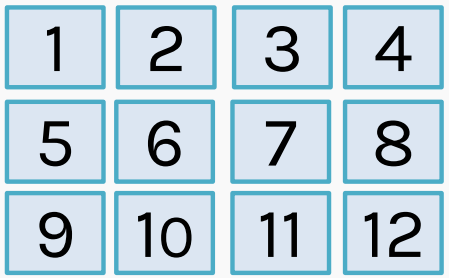
Reshaping

```
reshape(4,-1)
```



Reshaping

What will the new shape be?

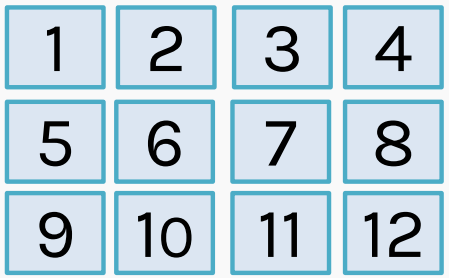



```
reshape(-1,8)
```

The unknown dimension MUST be an integral number else you will get a `ValueError`

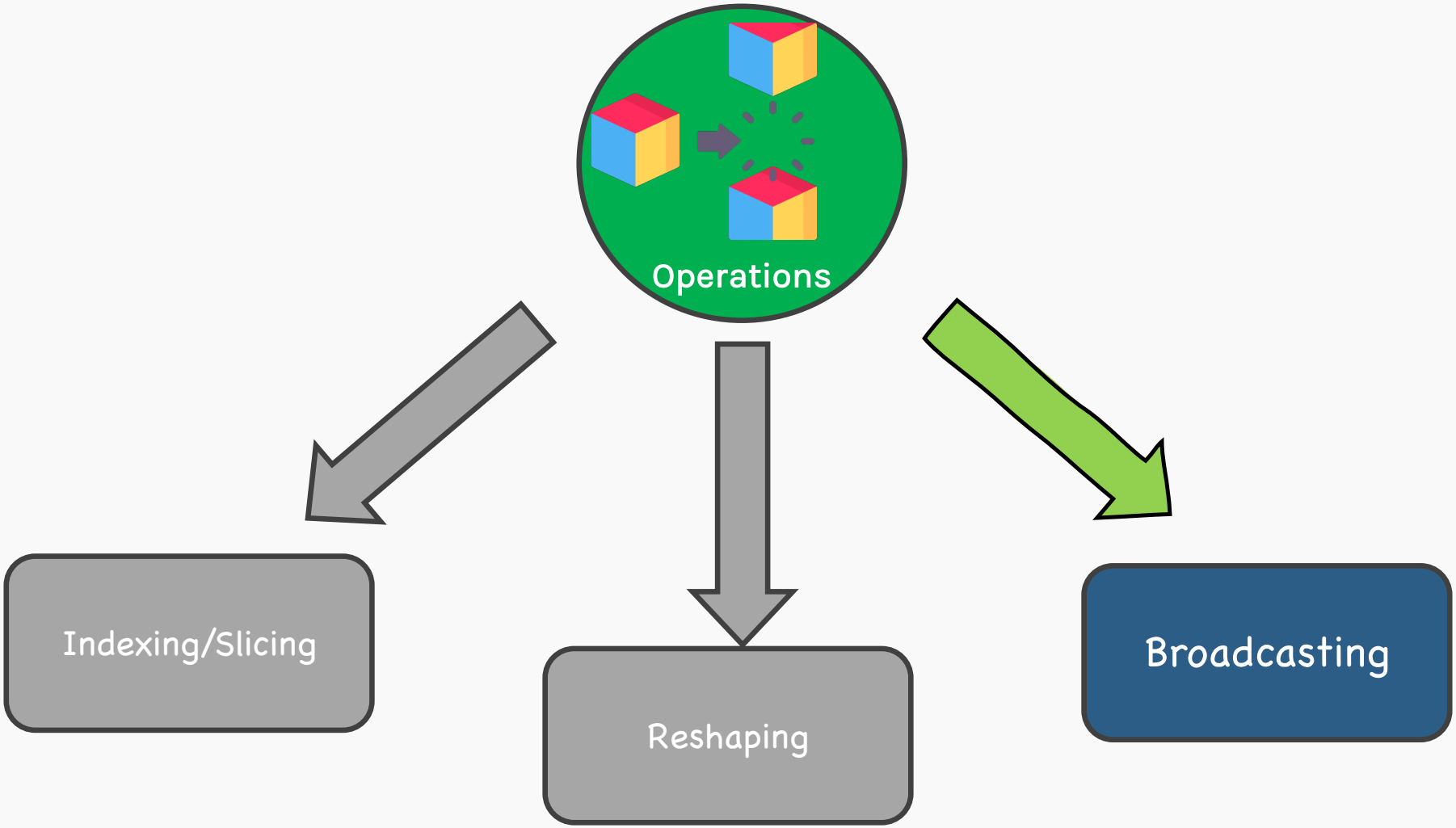
Reshaping

What will the new shape be?



`reshape(-1,-1)`

You can only specify one unknown dimension at a time, else you will get a `ValueError`

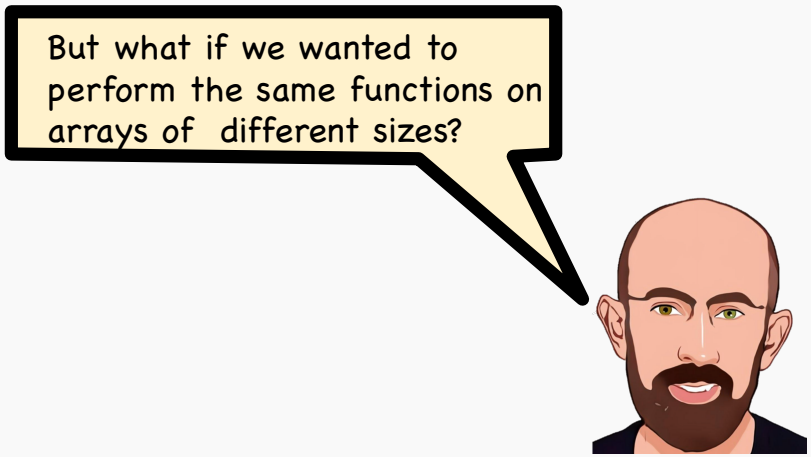

Operations
Indexing/Slicing
Reshaping
Broadcasting

Broadcasting

For arrays of the same size, binary operations are performed on an element-by-element basis.

Broadcasting

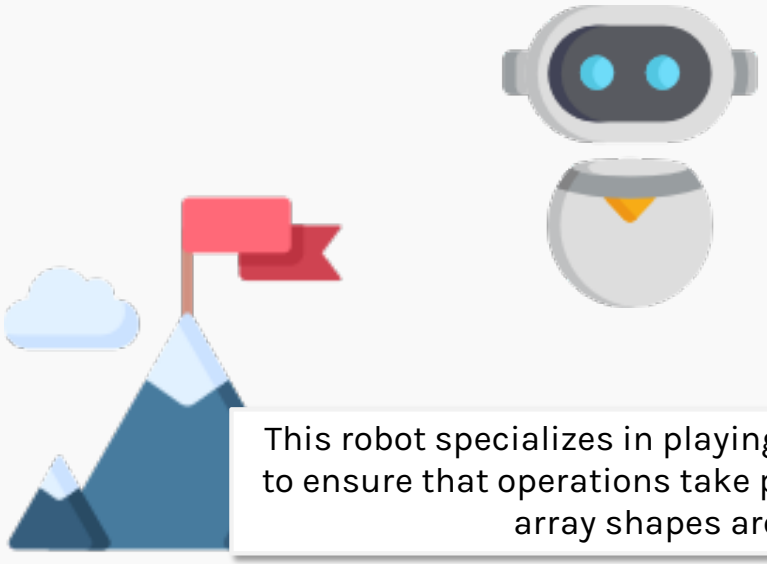
```
>>> a = np.array([0, 1, 2])
>>> b = np.array([5, 5, 5])
>>> a + b
array([5,6,7])
```



Broadcasting

This is where broadcasting comes in! Broadcasting allows these types of binary operations to be performed on arrays of different sizes.

Let this robot be a
“Broadcasting” specialist.



This robot specializes in playing a game where the goal is to ensure that operations take place between arrays if the array shapes are different!

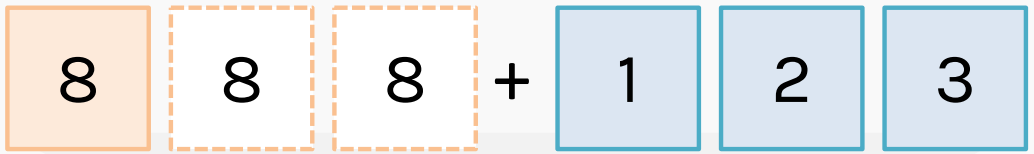
Broadcasting

Game #1

```
>>> a = np.array([0, 1, 2])
>>> a + 8
```

Shape is (3,)

Shape of scaler 8 is ()



The "robot" duplicates 8 to make it the same size as array a.

(3,) (3,)

Output

Broadcasting

Game #2

```
>>> a = np.array([0, 1, 2])
>>> m = np.ones((3,3))
>>> a + m
```

Shape is (3,)

Shape is (3,3)

1	2	3
1	2	3
1	2	3

+

1	1	1
1	1	1
1	1	1

(3,3)

The “robot” duplicates [1,2,3] to make it the same size as array m.

(3,3)

Output

2	3	4
2	3	4
2	3	4

Broadcasting

Example 1

adding a two-dimensional array to a one-dimensional array

```
M = np.ones((2, 3))
a = np.arange(3)
```

M.shape = (2, 3)

a.shape = (3,)

M.shape = (2, 3)

a.shape = (1,3)

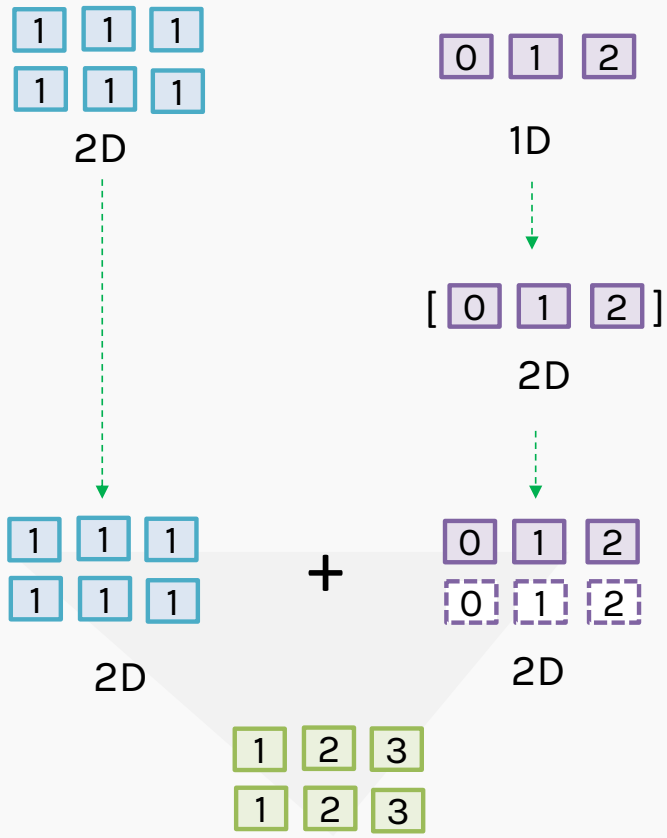
M.shape = (2, 3)

a.shape = (2,3)

OUTPUT

```
>>> M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

The shapes match, and we see that the final shape will be (2, 3)



Broadcasting

Example 2

adding two arrays that need to be broadcast

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

a.shape = (3, 1)

a.shape = (3, 1)

a.shape = (3, 3)

b.shape = (3,)

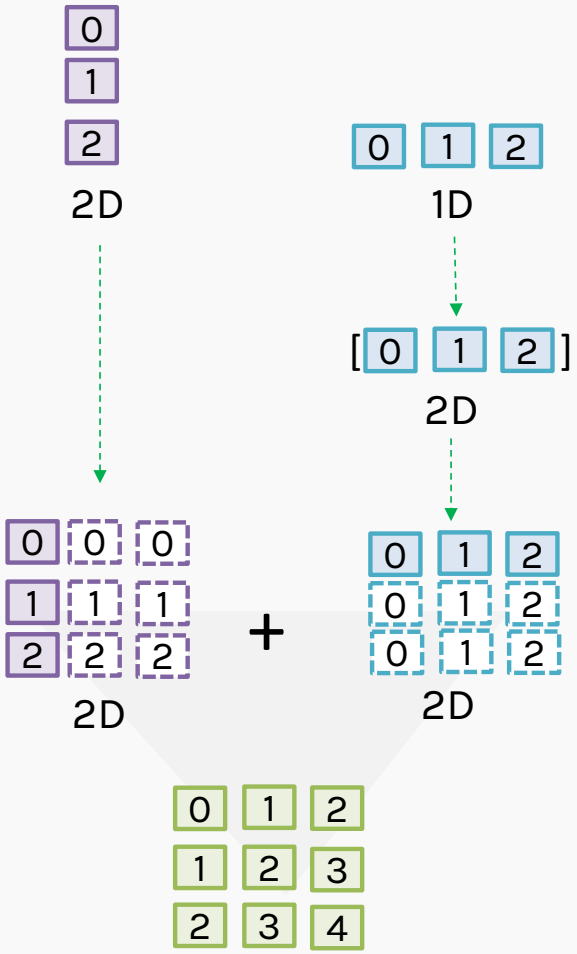
b.shape = (1,3)

b.shape = (3,3)

OUTPUT

```
>>> a + b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

The shapes match, and we see that the final shape will be (3, 3)



3) Broadcasting

Example 3

Trying to broadcast incompatible shapes

```
M = np.ones((4, 2))
a = np.arange(4)
```

M.shape = (4, 2)

M.shape = (4, 2)

M.shape = (4, 2)

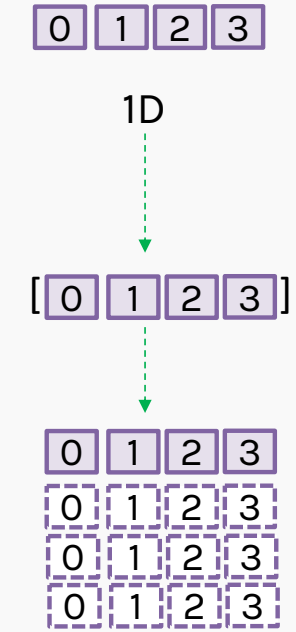
a.shape = (4,)

a.shape = (1,4)

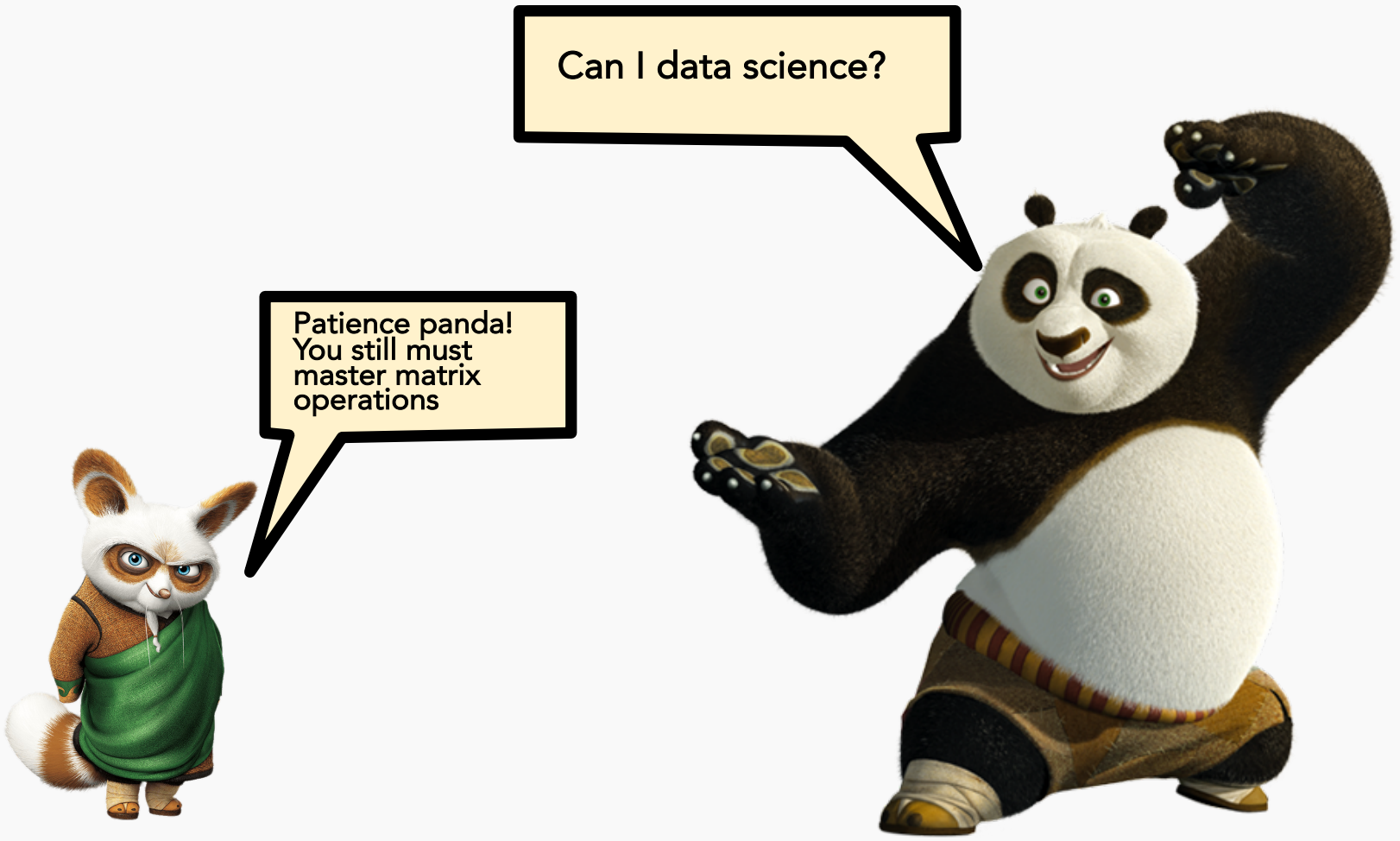
a.shape = (4,4)

OUTPUT

```
>>> M + a
ValueError: operands could not be broadcast together with shapes (4,2) (4,)
```



The shapes do not match and the broadcasting cannot be applied to M since the size to be stretched is NOT a 1.


Can I data science?
Patience panda! You still must master matrix operations

Matrix Operations

Element-wise multiplication

st operation of two matrices is the element-wise

tion of two matrices of the same dimensions.

This is called the hadamard product

1	2	1	-5
4	3	2	6
4	2	1	4
0	0	0	1

$\odot$

0	0	4	1
-1	1	2	1
0	2	4	0
1	4	7	4

$=$

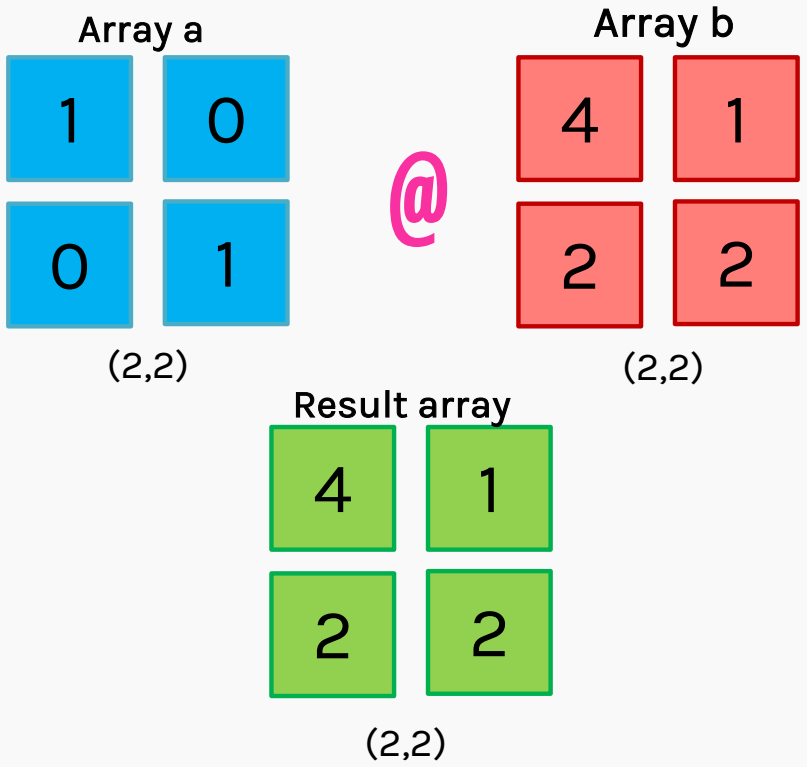
0	0	4	-5
-4	3	4	6
0	4	4	0
0	0	0	4

```
In [4]: a = np.random.randint(-5,5,(4,4))
In [5]: b = np.random.randint(-5,5,(4,4))
In [6]: a*b
Out[6]:
array([[-12,   0,  15,   6],
       [ 15, -20,  12, -16],
       [  4,  -4,   0,   0],
       [ 10,   1, -10,   5]])
```

Matrix Multiplication

Numpy's **np.matmul()** & **@** is used to perform matrix multiplication of two arrays.

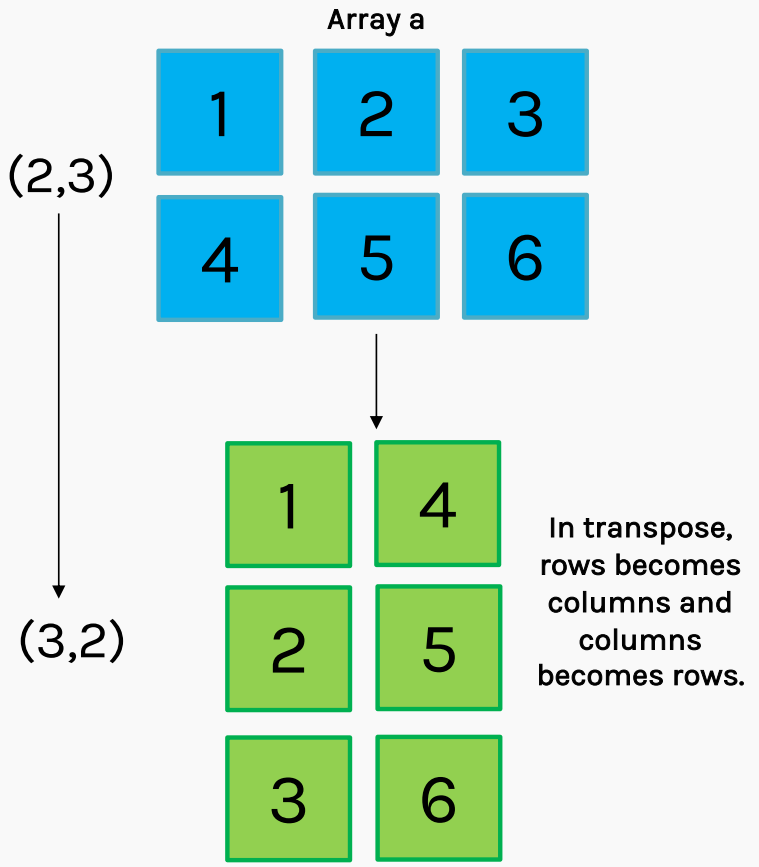
```
In [7]: #Defining array a
   ...: a = np.array([[1, 0],
   ...:                [0, 1]])
   ...: #Defining array b
   ...: b = np.array([[4, 1],
   ...:                [2, 2]])
   ...:
   ...: #Performing matrix multiplication
   ...: np.matmul(a, b)
Out[7]:
array([[4, 1],
       [2, 2]])

In [8]: a@b
Out[8]:
array([[4, 1],
       [2, 2]])
```



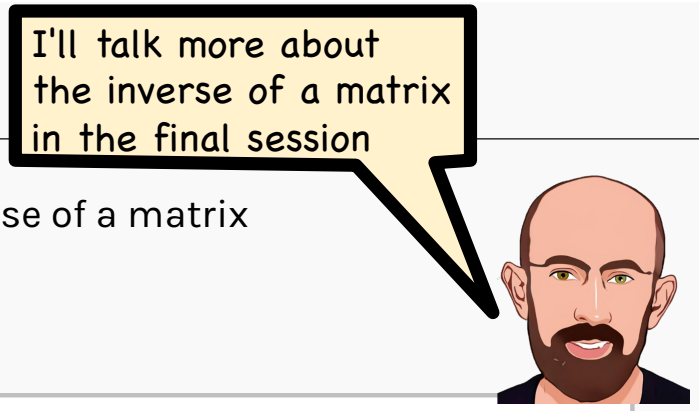
Transpose

You can perform transpose over numpy objects by calling **np.transpose()** or **ndarray.T**

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

#Before transpose
print(a)
array([[1, 2, 3],
       [4, 5, 6]])

#After transpose
a.transpose()
array([[1, 4],
       [2, 5],
       [3, 6])

a.T
array([[1, 4],
       [2, 5],
       [3, 6])
```



Inverse of a matrix



numpy.linalg.inv() is used to calculate the inverse of a matrix (if it exists!)

When we multiply a number by its reciprocal we get 1.

$$n * \frac{1}{n} = 1$$

When we multiply a matrix by its inverse, we get the Identity Matrix

$$A\,A^{-1} = I$$

```
In [16]: x = np.array([[1,2],[3,4]])
    ...:
    ...: #Inverse array x
    ...: invX = np.linalg.inv(x)
    ...: print(invX)
    ...:
    ...: #Verifying
    ...: print(np.dot(x, invX))
[[-2.   1. ]
 [ 1.5 -0.5]]
[[1.00000000e+00 1.11022302e-16]
 [0.00000000e+00 1.00000000e+00]]
```

Resource for Linear Algebra basics

Dot product

To compute dot product of numpy ndarrays, you can use **numpy.dot()** function.

```
a = 5
b = 6

#Performing dot product between scalars
output = np.dot(a,b)
print(output)
30
```

CASE 1:

Applying dot product between 2 scalars

Results in normal multiplication.

5 * 6 = 30

Dot product

To compute dot product of numpy ndarrays, you can use **numpy.dot()** function.

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

#Performing dot product between vectors
output = np.dot(a,b)
print(output)
77
```

CASE 2:

Applying dot product between 2 1D arrays (or) vectors

Results in Inner product.

= [2, 1, 5, 4].[3, 4, 7, 8]
= 2*3 + 1*4 + 5*7 + 4*8
= 77

Dot product

To compute dot product of numpy ndarrays, you can use **numpy.dot()** function.

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

#Performing dot product between vectors
output = np.dot(a,b)
print(output)
[[13 16]
 [43 52]]
```

CASE 3:

Applying dot product between 2 2D arrays (or) matrices

Results matrix multiplication.

= [[2, 1], [5, 4]].[[3, 4], [7, 8]]
= [[2*3+1*7, 2*4+1*8], [5*3+4*7, 5*4+4*8]]
= [[13, 16], [43, 52]]

Dot product

To compute dot product of numpy ndarrays, you can use **numpy.dot()** function.

Dimension of a, b	Output
Scalars (0D)	Normal multiplication
Vectors (1D)	Inner Product
Matrices (2D)	Matrix Multiplication
A: n-dim array B: 1D array	Sum product over the last axis of a and b
A: n-dim array B: m-dim array (m>=2)	Sum product over the last axis of a and second-to-last axis of b