NUMPY

What is Numpy?

Numpy stands for numerical python.

It is the core and extremely popular library for scientific computations.

Numpy contains large number of mathematical algebraic functions.

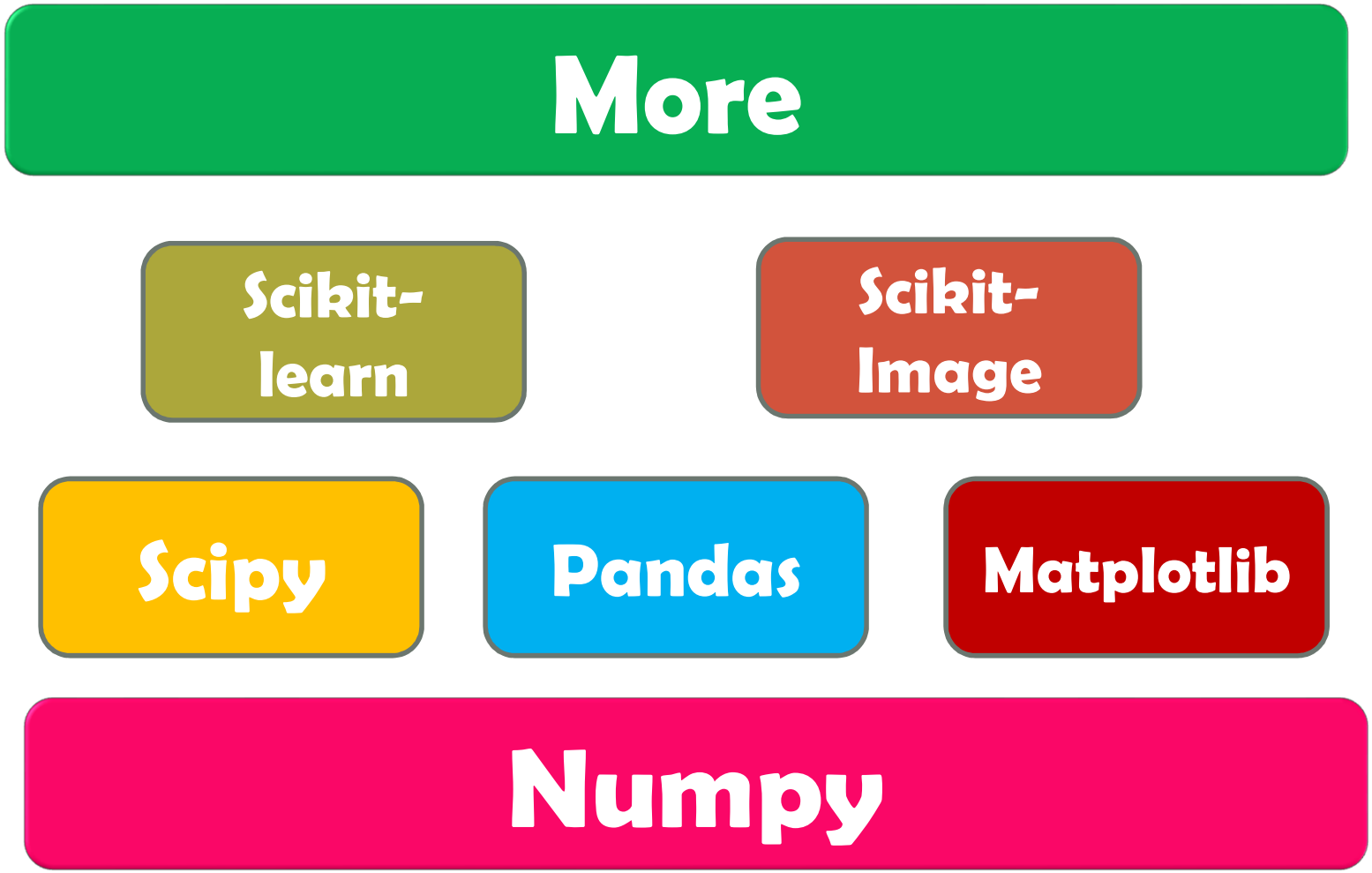
Numpy is a Linear Algebra library for Python.

Numpy is also incredibly fast.

Numpy

- NumPy arrays essentially come in two flavors: Vectors and Matrices.
- Vectors are strictly 1-d arrays and matrices are 2-d or 3-d.

The reason it is so Popular for Data Science with Python is that almost all of the libraries in the PyData Ecosystem rely on Numpy as one of their main building block.


More
Scikit-learn
Scikit-Image
Scipy
Pandas
Matplotlib
Numpy

Properties of Numpy

Mutable.

We can do indexing and slicing.

Arithmetic and Vectorized operations element wise.

Similarities Between Numpy and Python List

Both are mutable.

Both are used to store data.

Both can be indexed and sliced.

Python List Vs. Numpy

- Inside lists we have elements of multiple data type.
- Lists are slow compared to Numpy.
- We can't do element wise operation.
- Takes more size
- List is a part of core Python

- All the elements must be of same data type.
- Numpy arrays are faster than lists.
- We can do element wise operations in numpy.
- Takes less size as compared to list
- Numpy Array is a part of Numpy Library

Numpy

- **Collection of same types of Element**

- **1-D Numpy**

Step :1

```
import numpy as np
```

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

```
oned
```

Only in Jupyter

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

Numpy

- **2-D Numpy**

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

```
twod
```

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

Numpy

- **3-D Numpy**

```
1  import numpy as np
```

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

```
1  threed
```

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

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

Numpy

List -> Numpy

```
list1 = [1,2,3]
print(list1)
print(type(list1))
```

```
[1, 2, 3]
<class 'list'>
```

```
numpy = np.array(list1)
print(numpy)
print(type(numpy))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

Numpy -> List

```
numpy = np.array(list1)
print(numpy)
print(type(numpy))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

```
back_to_list = numpy.tolist()
```

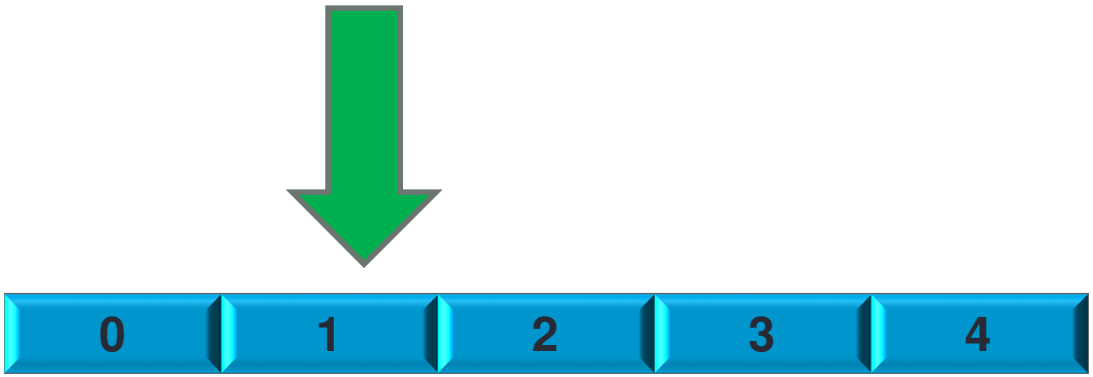
```
print(back_to_list)
print(type(back_to_list))
```

```
[1, 2, 3]
<class 'list'>
```

Numpy : ways to create numpy (allows float)

arange(start,stop,step)

np.arange(5)




```
numpy = np.arange(5)
numpy
```

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

```
numpy = np.arange(10,14)
numpy
```

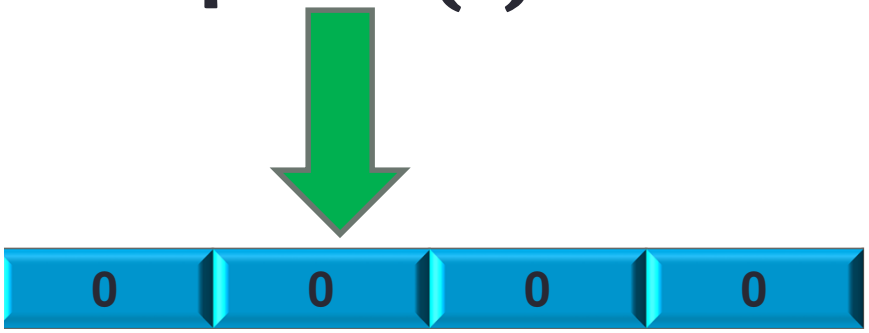
```
array([10, 11, 12, 13])
```

```
numpy = np.arange(10,20,2)
numpy
```

```
array([10, 12, 14, 16, 18])
```

Numpy : ways to create numpy

- **np.zeros(4)**




```
numpy = np.zeros(4,dtype='int')
numpy
```

```
array([0, 0, 0, 0])
```

```
numpy =np.zeros((3,3))
numpy
```

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

- **np.ones(4)**


```
numpy = np.ones(4,dtype='int')
numpy
```

```
array([1, 1, 1, 1])
```

```
numpy =np.ones((3,3))
numpy
```

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

Numpy : ways to create numpy

- linspace() : It is used to create an evenly spaced sequence in a specified interval

- Required Parameters

- Start : Starting value of the sequence

- Stop : end value of the sequence unless endpoint set to false

Numpy : ways to create numpy

- linspace() : It is used to create an evenly spaced sequence in a specified interval
- Optional Parameters:
- num : The number of samples needed to generate within the interval. The default value is 50
- endpoint : if endpoint is set to false, then the end value is not included in the sequence

Numpy : ways to create numpy

- linspace() : It is used to create an evenly spaced sequence in a specified interval
- Optional Parameters:
- retstep : If the retstep is true then (samples, step) is returned. "Step" refers to the spacing between the values in the interval
- dtype : The type of the output array. If `dtype` is not given, infer the data type from the other input arguments
- axis : The axis in the result to store the samples. (Added in version 1.16.0)

Numpy : ways to create numpy

linspace() : It is used to create an evenly spaced sequence in a specified interval

```
1 import numpy as np
2 np.linspace(2.0, 3.0, num=5)
```

```
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

```
1 np.linspace(2.0, 3.0, num=5, endpoint=False)
```

```
array([2. , 2.2, 2.4, 2.6, 2.8])
```

```
1 np.linspace(2.0, 3.0, num=5, retstep=True)
```

```
(array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

Numpy : ways to create numpy (randint)

```
1 x1= np.random.randint(0,10,15)
2 x2=np.random.randint(10,20,15)
3 x3= np.random.randint(20,30,15)
```

```
1 data = np.array([x1,x2,x3])
```

```
1 data
```

```
array([[ 2,  9,  3,  3,  7,  3,  9,  2,  9,  5,  0,  1,  4,  1,  3],
       [10, 13, 10, 11, 18, 11, 12, 11, 12, 10, 15, 16, 13, 10, 13],
       [24, 20, 22, 21, 25, 28, 28, 23, 23, 23, 25, 29, 27, 28, 21]])
```

Numpy : ways to create numpy (eye())

- **Returns a 2-D array with ones on the diagonal and zeros elsewhere.**

```
1  import numpy as np
```

```
1  np.eye(2)
```

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

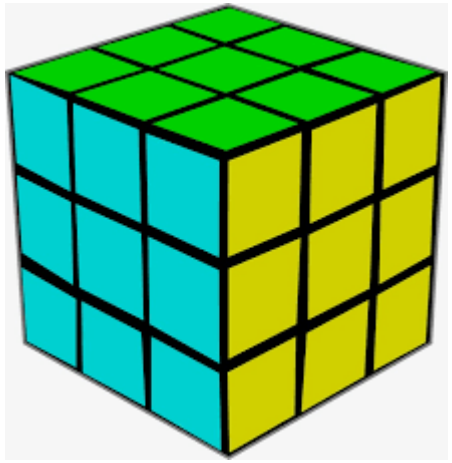
```
1  np.eye(2,3,dtype='int')
```

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

```
1  np.eye(3,k=1)
```

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

Getting Information About Numpy (Attributes)



Find the Dimension

```
a=np.array([5,6,7])
```

```
a.ndim
```

```
1
```

SIZE

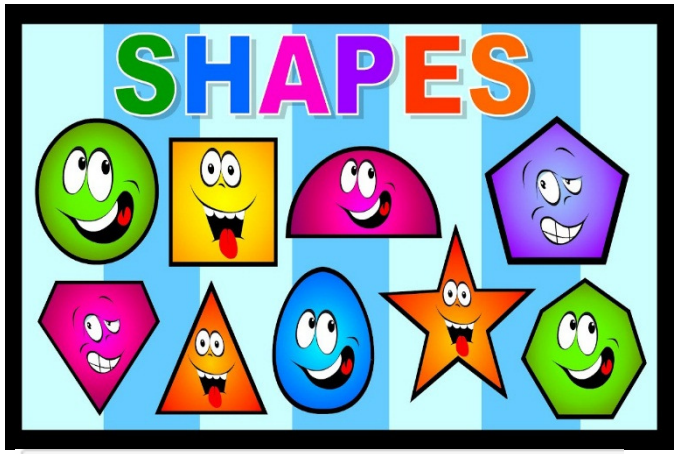
```
numpy  = np.array([1,2,3])
```

```
numpy.itemsize
```

```
4
```

```
numpy.size
```

```
3
```




```
numpy  = np.array([1,2,3])
numpy.shape
```

```
(3,)
```

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

```
(2, 3)
```

```
numpy  = np.array([[1,2,3],
                   [4,5,6],
                   [7,8,9]])
numpy.shape
```

```
(3, 3)
```

Python List Vs. Numpy

- Inside lists we have elements of multiple data type.
- Lists are slow compared to Numpy.
- We can't do element wise operation.
- Takes more size

- All the elements must be of same data type.
- Numpy arrays are faster than lists.
- We can do element wise operations in numpy.
- Takes less size as compared to list

Python List Vs. Numpy

Numpy arrays : Contain only one type

```
import numpy as np
```

```
numpy = np.array([1,3.5,10])
```

```
numpy
```

```
array([ 1. ,  3.5, 10. ])
```

```
import numpy as np
```

```
numpy = np.array([1,"priyang",10])
```

```
numpy
```

```
array(['1', 'priyang', '10'], dtype='<U11')
```

```
import numpy as np
```

```
numpy = np.array([1,'hello',3.4])
```

```
numpy
```

```
array(['1', 'hello', '3.4'], dtype='<U11')
```

```
import numpy as np
```

```
numpy = np.array([1,True,False])
```

```
numpy
```

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

Python List Vs. Numpy

- Inside lists we have elements of multiple data type.
- Lists are slow compared to Numpy.
- We can't do element wise operation.
- Takes more size

- All the elements must be of same data type.
- Numpy arrays are faster than lists.
- We can do element wise operations in numpy.
- Takes less size as compared to list

Python List Vs. Numpy

```
python_list = [1,2,3]
```

```
python_list + python_list
```

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

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

```
numpy_array + numpy_array
```

```
array([2, 4, 6])
```

Sir, Why should I use numpy? I have a List

Numpy Vs. List

Advantages of Numpy over List

Python List Vs. Numpy

- Inside lists we have elements of multiple data type.
- Lists are slow compared to Numpy.
- We can't do element wise operation.
- Takes more size

- All the elements must be of same data type.
- Numpy arrays are faster than lists.
- We can do element wise operations in numpy.
- Takes less size as compared to list

Practical : Memory

List

```
import numpy as np
import time
import sys
```

```
s = range(1000)
```

```
print(sys.getsizeof(5) * len(s))
```

```
28000
```

Numpy

```
numpy = np.arange(1000)
```

```
print(numpy.size)
print(numpy.itemsize)
```

```
1000
4
```

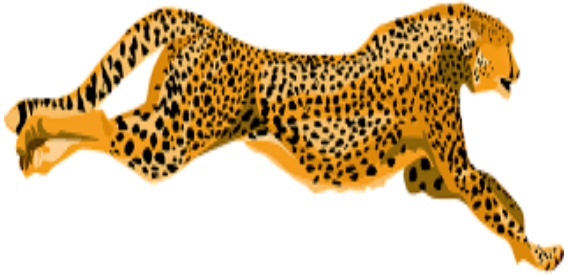
```
print(numpy.size * numpy.itemsize)
```

```
4000
```

Numpy Vs. List

Sir, Why should I use numpy? I have a List

Advantages of Numpy over List



Python List Vs. Numpy

- Inside lists we have elements of multiple data type.
- Lists are slow compared to Numpy.
- We can't do element wise operation.
- Takes more size

- All the elements must be of same data type.
- Numpy arrays are faster than lists.
- We can do element wise operations in numpy.
- Takes less size as compared to list

Practical : Speed

```
import time
```

```
SIZE = 1000000
```

```
L1 =range(SIZE)
L2 =range(SIZE)
```

```
A1 = np.arange(SIZE)
A2= np.arange(SIZE)
```

Practical : Speed

List

```
start = time.time()
result = [(x+y) for x,y in zip(L1,L2)]
print((time.time()-start)*1000)
```

```
260.01477241516113
```

Time ms taken by List

Numpy

```
start = time.time()
result = A1 + A2
print((time.time()-start)*1000)
```

```
47.00279235839844
```

Time ms taken by Numpy

Numpy Vs. List

Sir, Why should I use numpy? I have a List

Advantages of Numpy over List

Python List Vs. Numpy

- Inside lists we have elements of multiple data type.
- Lists are slow compared to Numpy.
- We can't do element wise operation.
- Takes more size

- All the elements must be of same data type.
- Numpy arrays are faster than lists.
- We can do element wise operations in numpy.
- Takes less size as compared to list

Practical :Convenient List

```
height = [1.67,1.74,1.56,1.89,1.71]
```

```
height
```

```
[1.67, 1.74, 1.56, 1.89, 1.71]
```

```
weight = [66.2,56.3, 63.6, 79.4,68.7]
```

```
weight
```

```
[66.2, 56.3, 63.6, 79.4, 68.7]
```


```
bmi = weight/height ** 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-1f23fc3140dd> in <module>()
----> 1 bmi = weight/height ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Practical :Convenient Numpy

```
import numpy as np
```

```
height =np.array([1.67,1.74,1.56,1.89,1.71])
```

```
height
```

```
array([1.67, 1.74, 1.56, 1.89, 1.71])
```

```
weight = np.array([66.2,56.3, 63.6, 79.4,68.7])
```

```
weight
```

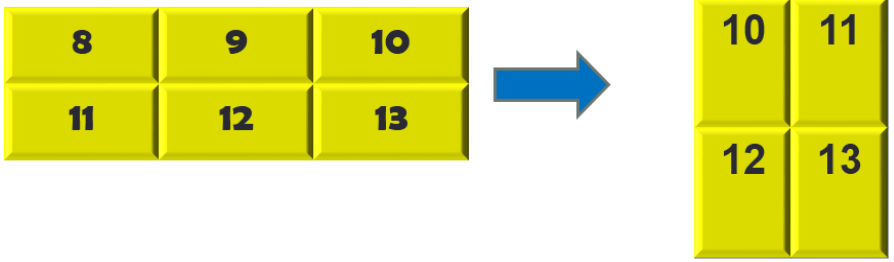
```
array([66.2, 56.3, 63.6, 79.4, 68.7])
```


```
bmi = weight/height ** 2
```

```
bmi
```

```
array([23.73695722, 18.59558726, 26.13412229, 22.22782117, 23.49440854])
```

Numpy Operations




```
a.reshape(3,3)
```

Error !!!

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

```
a
```

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

```
a.reshape(2,3)
```

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

```
a.reshape(6,1)
```

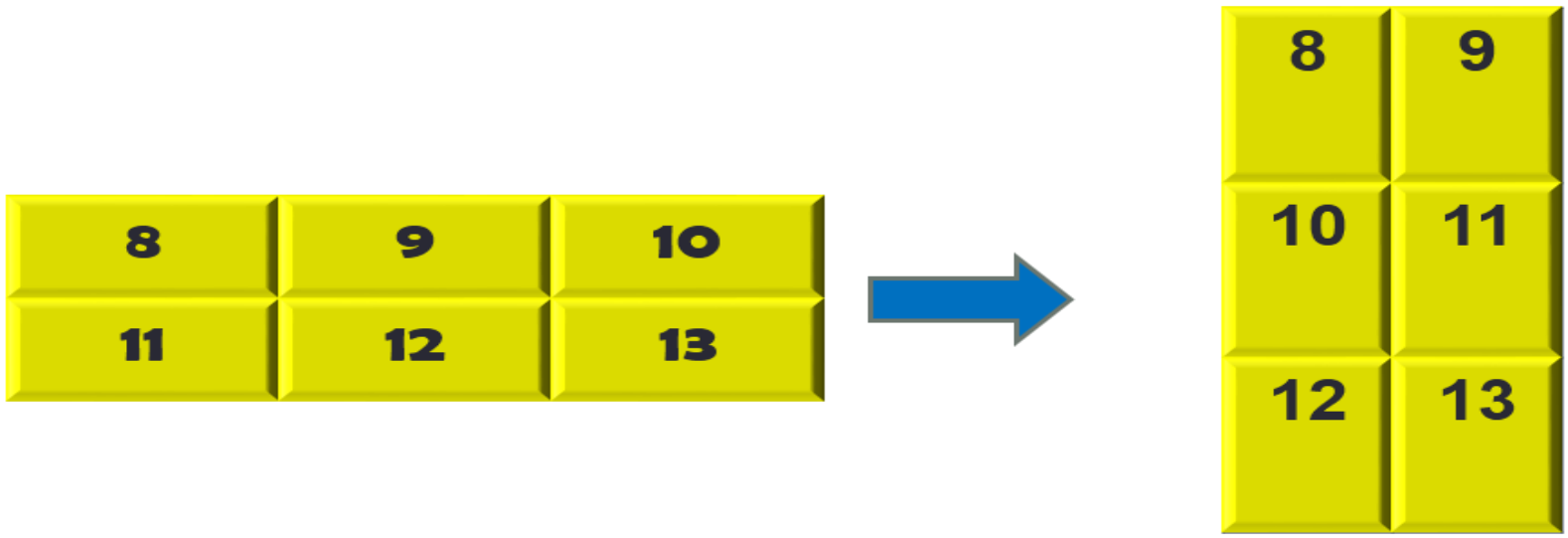
```
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

```
a.reshape(3,3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-9-7a8e5c733146> in <module>()
----> 1 a.reshape(3,3)

ValueError: cannot reshape array of size 6 into shape (3,3)
```

Numpy reshape()
with
!!! Unknown Dimensions !!!
reshape((-1,2)),reshape((-1,-1)),
reshape((-1,1))



Numpy reshape()
with
!!! Unknown Dimension !!!

- We can give an unknown dimension in reshape function.
- Means that we do not have to specify an exact number for one of the dimensions in the reshape() function.
- For that unknown dimension, we have to write "-1" as the value.
- Numpy will calculate this number for you.

Numpy reshape()
with
!!! Unknown Dimension !!!

Alert :

We can not pass “-1” to more than one dimension

Numpy reshape()
with
!!! Unknown Dimension !!!

- **Examples:**

1. a = np.arange(10).reshape((5,2))
2. a.reshape((2, -1))
3. a.reshape((5, -1))
4. a.reshape((-1, 2))
5. a.reshape((-1, 5))
6. a.reshape((-1, -1))
7. a.reshape((2, -1))
8. a.reshape(-1)

Numpy reshape()
with
!!! Unknown Dimension !!!

- Next ,

- reshape(-1,1)

Numpy Operations : convert into 1d (Using Ravel() Vs. Flatten() Vs. Reshape(-1)

Ravel()

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

```
a
```

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

```
b
```

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

```
a[0,0]=10
```

b also changed !!!!

```
a
```

```
array
```

```
b
```

```
array([10,  2,  3,  4,  5,  6])
```

Flatten()

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

```
a
```

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

```
b
```

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

```
a[0,0]
```

b !!!!

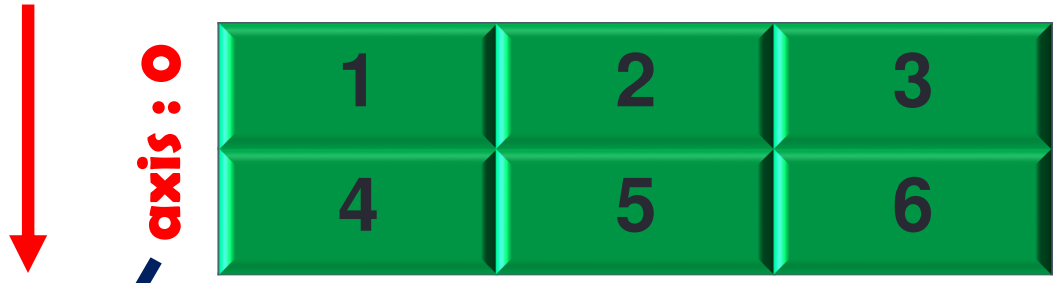
```
a
```

```
arr
```

```
b
```

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

Numpy: Concatenate



axis = 0
Vertically(row)
and
axis = 1
Horizontally
(Column).

axis : None!!!

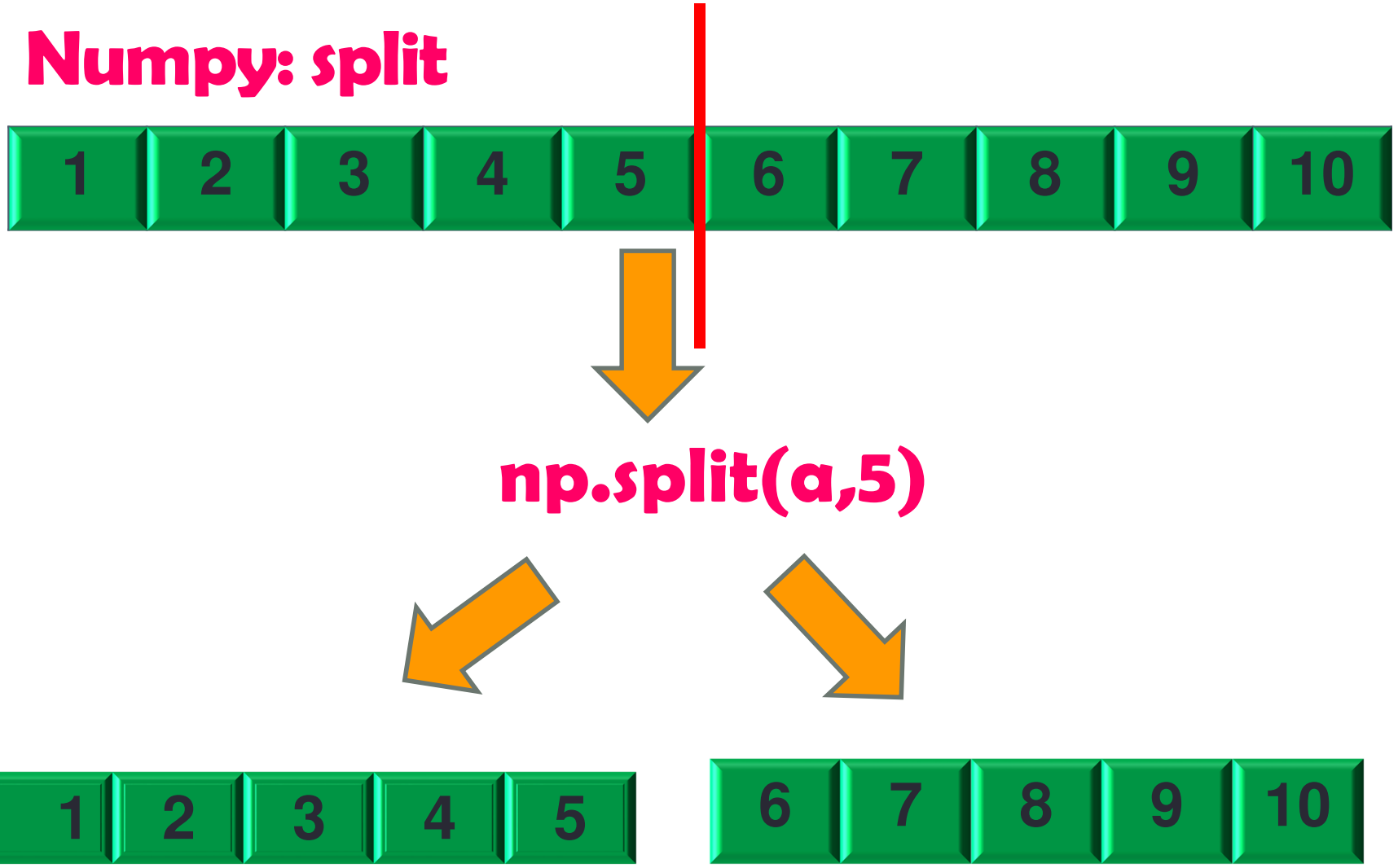
```
x= np.concatenate((a,b),axis=0)
x
```

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

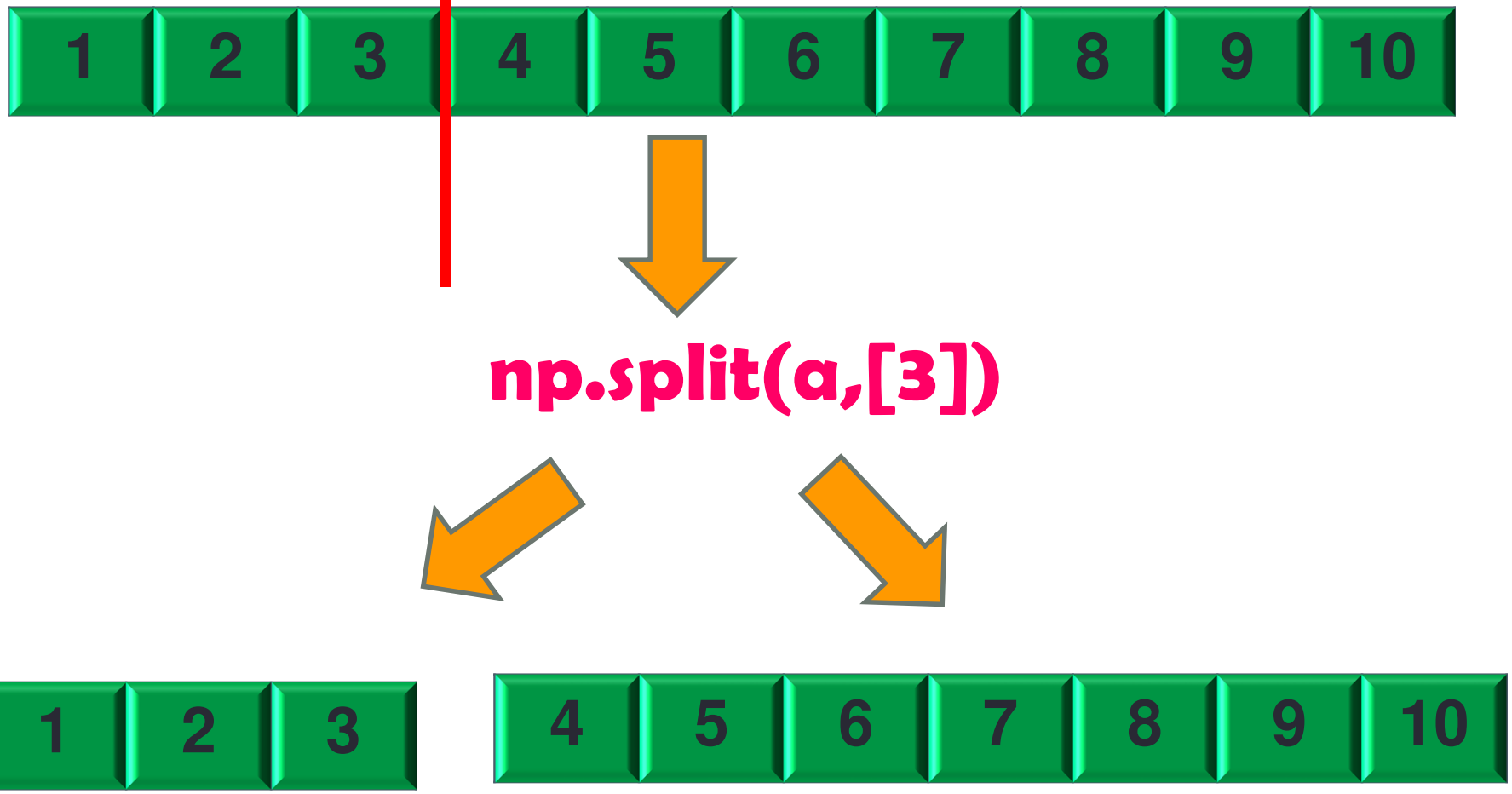
```
x= np.concatenate((a,b),axis=1)
x
```

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

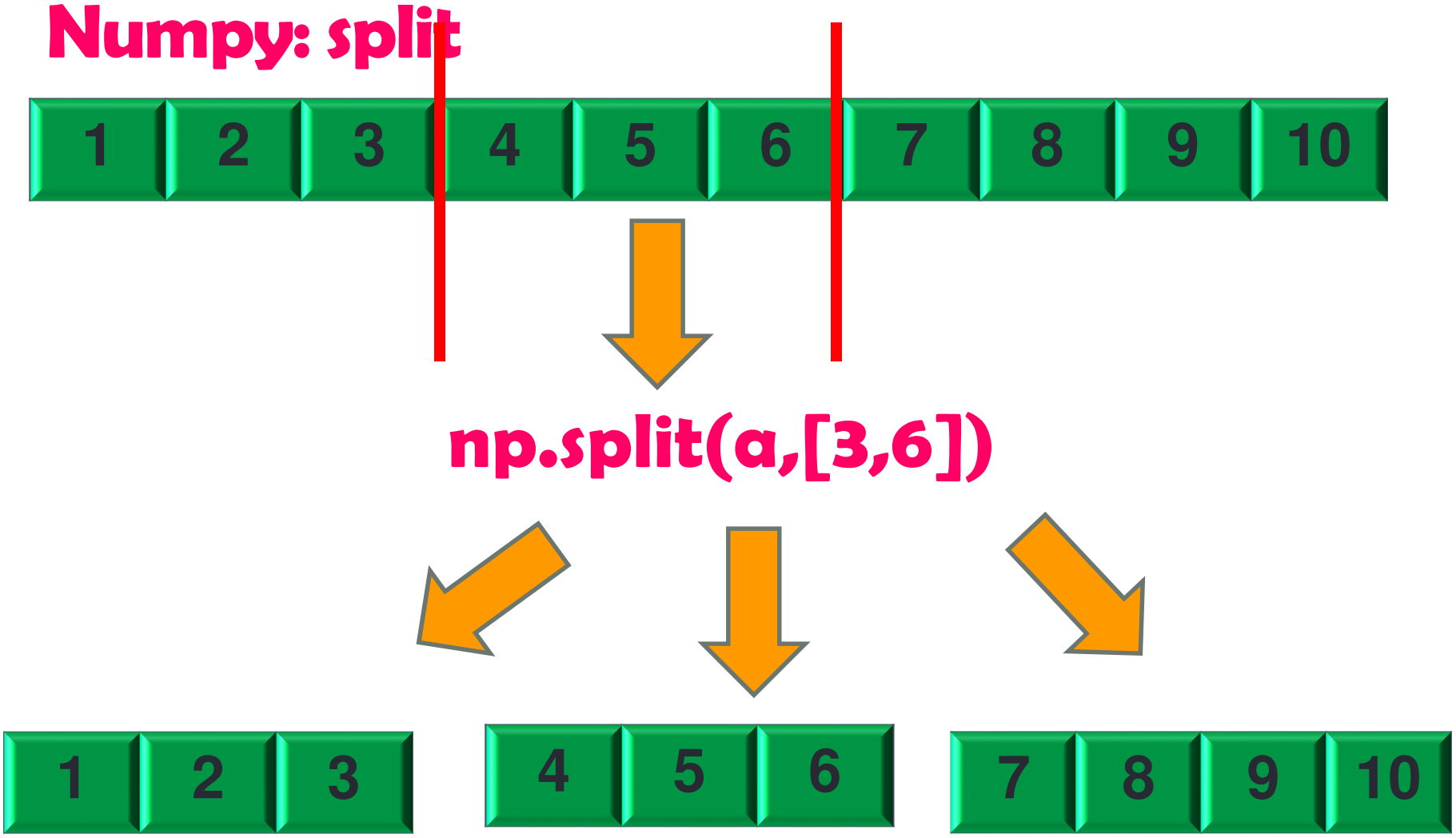
Numpy: split



Numpy: split



Numpy: split



Numpy: split (practical)

```
a= np.arange(1,11)
a
```

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

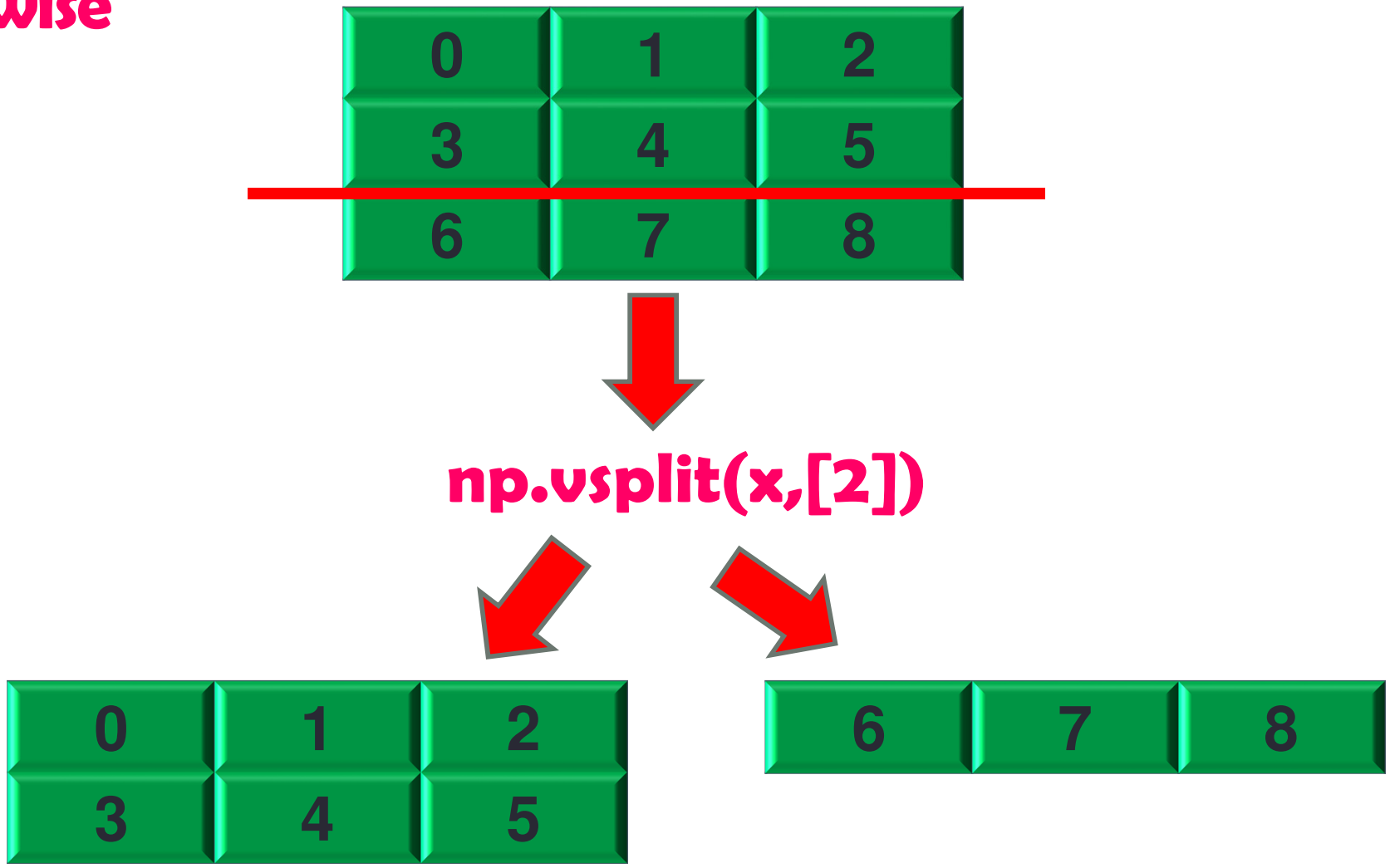
```
np.split(a,[3])
```

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

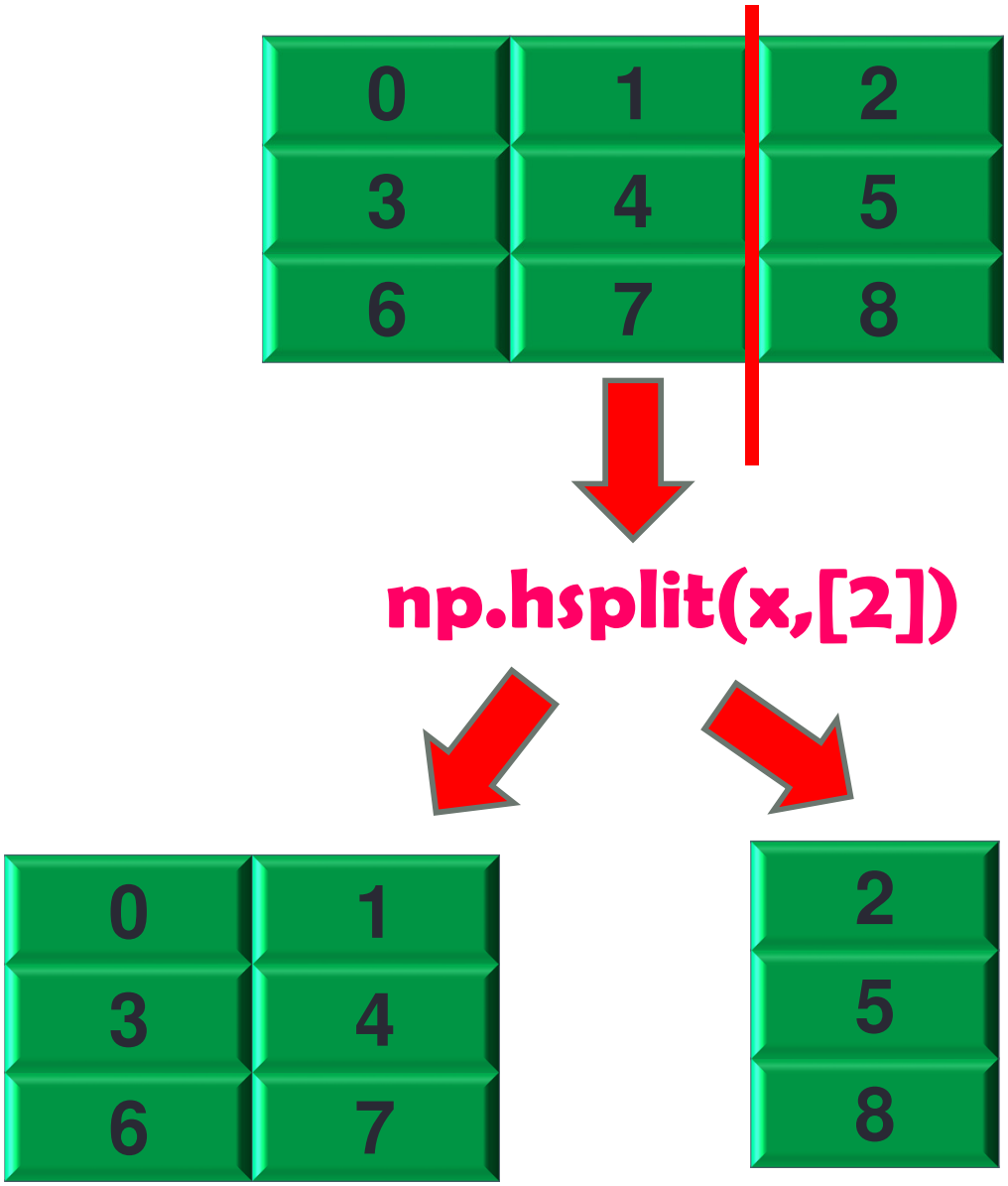
```
np.split(a,[3,6])
```

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

Numpy: np.vsplit == np. Split (axis=0)) row-wise



Numpy: np.hsplit == np. Split (axis=1)) column-wise



Numpy: vsplit (row) Vs. hsplit(column)

```
x= np.arange(9).reshape(3,3)
x
```

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

```
result = np.vsplit(x,[2])
```

```
result[0]
```

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

```
result[1]
```

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

```
x= np.arange(9).reshape(3,3)
x
```

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

```
result = np.hsplit(x,[2])
```

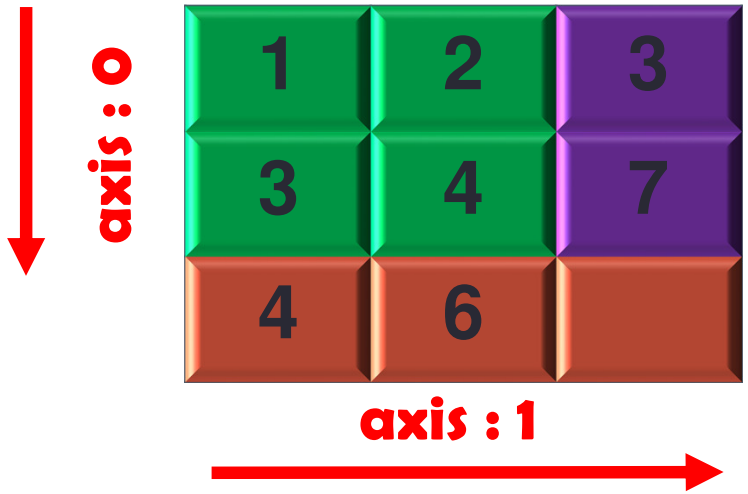
```
result[0]
```

```
array([[0, 1],
       [3, 4],
       [6, 7]])
```

```
result[1]
```

```
array([[2],
       [5],
       [8]])
```

Numpy: sum



axis = 0 means along the column and axis = 1 means working along the row.

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

```
10
```

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

```
10
```

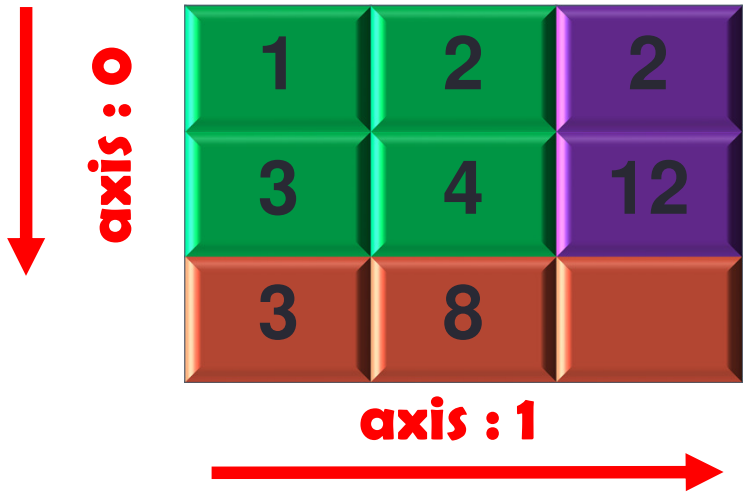
```
a.sum(axis=0)
```

```
array([4, 6])
```

```
a.sum(axis=1)
```

```
array([3, 7])
```

Numpy: product



axis = 0 means along the column and axis = 1 means working along the row.

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

```
a.prod()
```

```
24
```

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

```
a.prod(axis=0)
```

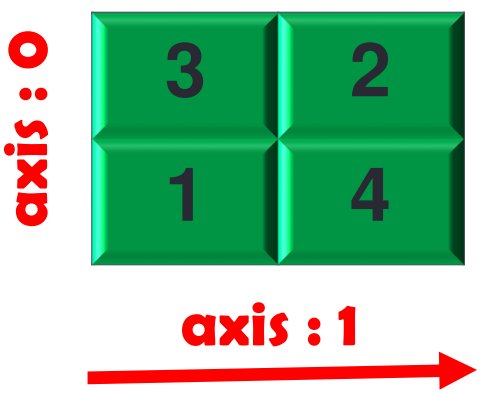
```
array([3, 8])
```

```
a.prod(axis=1)
```

```
array([ 2, 12])
```

Numpy: sort

axis = 0 means along the column and axis = 1 means working along the row.




```
a= np.array([1,10,31,4])
```

```
a.sort()
```

```
a
```

```
array([ 1,  4, 10, 31])
```

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

```
a.sort(axis=0)
```

```
a
```

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

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

```
a.sort(axis=1)
```

```
a
```

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

Numpy: vstack (row-wise)

vstack

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

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

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

vstack

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

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

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

Numpy: vstack

- For a 1D array, the shape would be (n,) : n is the number of elements in your array.
- For a 2D array, the shape would be (n , m) : n is the number of rows, m is the number of columns in your array.

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

```
(3,)
```

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

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

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

```
(3, 1)
```

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

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

Numpy: hstack (column-wise)

hstack

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

```
np.hstack((a,b))
```

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

hstack

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

```
np.hstack((a,b))
```

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

NumPy - Copies & Views

Copy

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

```
a = np.copy(b)
```

```
a
```

```
array([10,  2,  3,  4,  5,  6])
```

```
a[0]=1
```

```
a
```

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

```
b
```

```
array([10,  2,  3,  4,  5,  6])
```

View

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

```
b=a
```

```
b
```

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

```
a[0]=10
```

```
a
```

```
array([10,  2,  3,  4,  5,  6])
```

```
b
```

```
array([10,  2,  3,  4,  5,  6])
```

Numpy: Arithmetic operations (element-wise)

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

np.add(a,b)	a+b	array([5,7,9])
np.subtract(a,b)	a-b	array([-3,-3,-3])
np.multiply(a,b)	a*b	array([4,10,18])
np.divide(a,b)	a/b	array([0.25,0.4,0.5])
np.mod(a,b)	a%b	array([1,2,3])
np.power(a,b)	a**b	array([1,32,729])

Numpy: Arithmetic operations (element-wise)

a= array[[0,1,2], [3,4,5], [6,7,8]]		
1	np.max(a)	8
2	np.min(a)	0
3	np.std(a)	2.58
4	np.sqrt(a)	Square root of Every elements

Numpy: Arithmetic operations (element-wise)

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

b = np.array([4,5])

Result = a + b

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

```
a
```

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

Error !!!

```
b
```

```
array([4, 5])
```

```
a+b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-101-f1d53b280433> in <module>()
----> 1 a+b

ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Numpy: Comparison operations

```
import numpy as np
```

```
a = np.array([1,3,0])
b= np.array([0,3,2])
```

```
a>b
```

```
array([ True, False, False])
```

```
a<b
```

```
array([False, False,  True])
```

```
a==b
```

```
array([False,  True, False])
```

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

```
a>3
```

```
array([False, False, False,  True,  True,  True,  True])
```

```
a[a>3]
```

```
array([4, 5, 6, 7])
```

Numpy: Comparison operations (any())

`any()` returns:

- `True` if at least one element of an iterable is true
- `False` if all elements are false or if an iterable is empty

When	Return Value
All values are true	True
All values are false	False
One value is true (others are false)	True
One value is false (others are true)	True
Empty Iterable	False

```
c = np.array([True,True,False])
```

```
c
```

```
array([ True,  True, False])
```

```
any(c)
```

```
True
```

```
c = np.array([True,True,True])
```

```
c
```

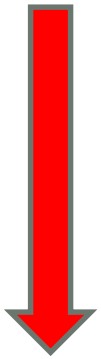
```
array([ True,  True,  True])
```

```
any(c)
```

```
True
```

Numpy: Comparison operations (all())

all(iterable)



List, tuple, Dictionary

When	Return Value
All values are true	True
All values are false	False
One value is true (others are false)	False
One value is false (others are true)	False
Empty Iterable	True

The all() method returns:

- **True** - If all elements in an iterable are true
- **False** - If any element in an iterable is false

```
c = np.array([True,True,True])
```

```
all(c)
```

```
True
```

```
c = np.array([True,True,False])
```

```
all(c)
```

```
False
```

Numpy: Broadcasting

- The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.
- Numpy arrays differ from a normal Python list because of their ability to broadcast.
- If the dimensions of two arrays are dissimilar, element-to-element operations are not possible.
- However, operations on arrays of non-similar shapes is still possible in NumPy, because of the broadcasting capability.
- The smaller array is broadcast to the size of the larger array so that they have compatible shapes.

Numpy: Broadcasting

- **NumPy operations are usually done on pairs of arrays on an element-by-element basis.**
- **In the simplest case, the two arrays must have exactly the same shape, as in the following example:**

```
1  import numpy as np
```

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

```
1  a+b
```

```
array([5, 6, 7])
```

So when Broadcasting comes into the picture?

1. When we are using scalar value with Numpy array.

2. If the dimensions of two arrays are dissimilar, element-to-element operations are not possible.

Numpy: Broadcasting

The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
1  import numpy as np
```

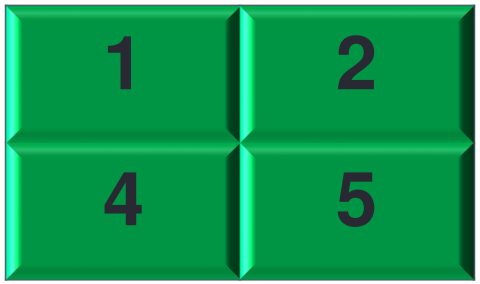
```
1  a=np.array([1,2,3])
2  b=4
```

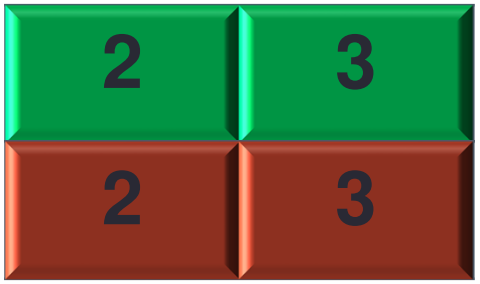
[4,4,4]

```
1  a+b
```

```
array([5, 6, 7])
```

Broadcasting in Numpy



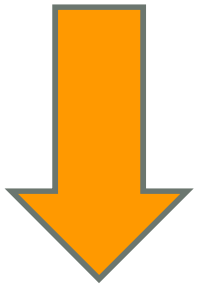


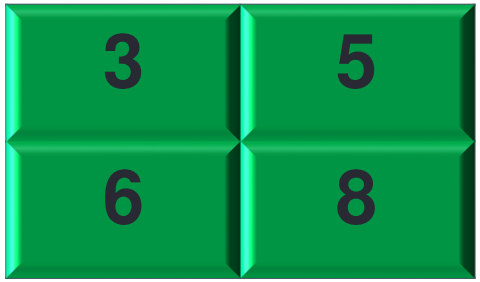

```
import numpy as np
```

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

```
a+b
```

```
array([[ 3,  5],
       [ 6,  8],
       [ 9, 11]])
```






```
import numpy as np
```

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

```
a+b
```

```
array([[3, 5],
       [6, 8]])
```

Numpy: Broadcasting

```
import numpy as np
```

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

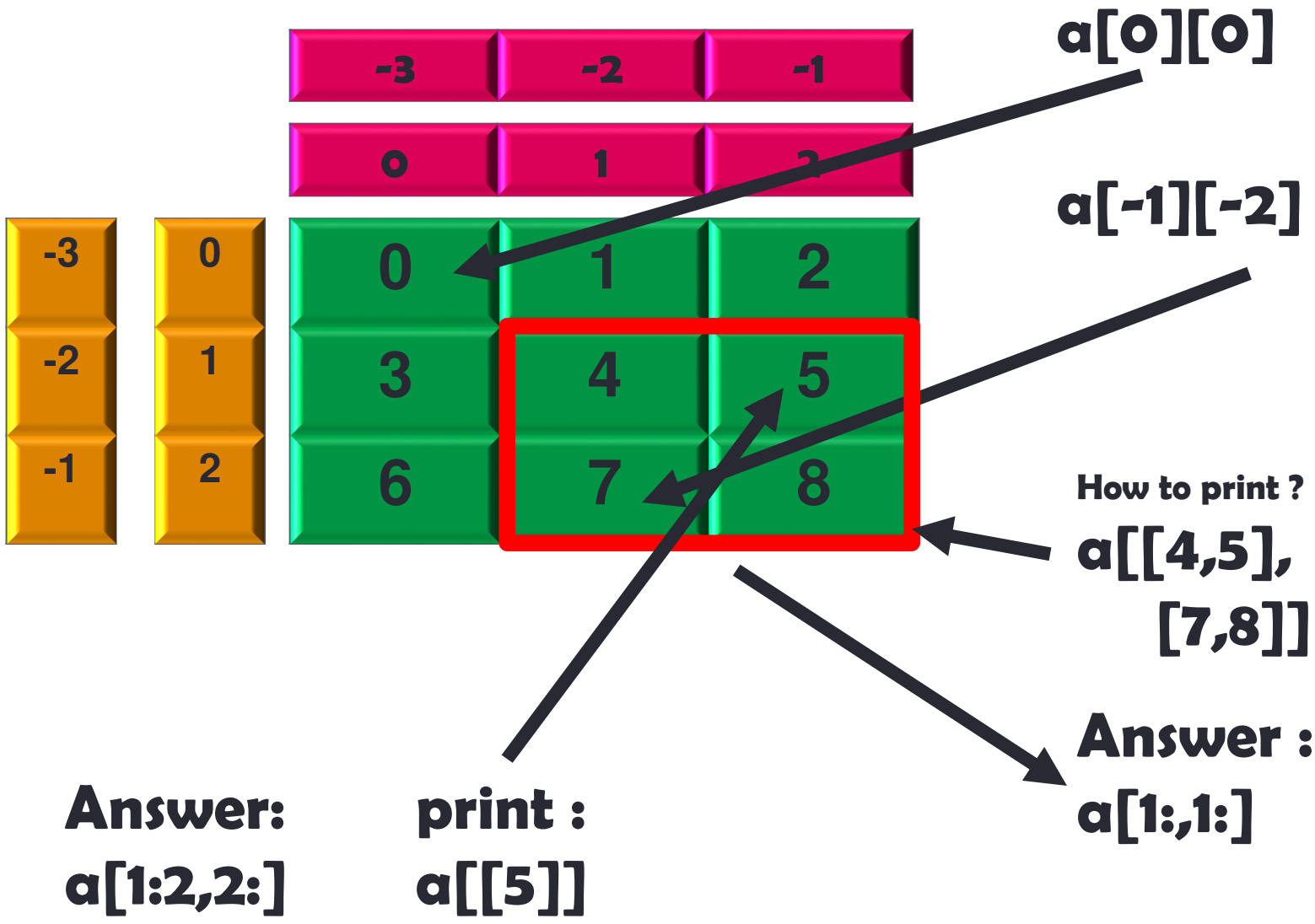
```
a+b
```

Error !!!

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-29-f1d53b280433> in <module>()
----> 1 a+b

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Numpy: indexing and Slicing



Numpy : Matrix Multiplication (a.dot(b))

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

```
a
```

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

```
b = np.array([[11,12],[13,14]])
```

```
b
```

```
array([[11, 12],
       [13, 14]])
```

```
a.dot(b)
```

```
array([[37, 40],
       [85, 92]])
```

```
[[1*11+2*13, 1*12+2*14],
 [3*11+4*13, 3*12+4*14]]
```

```
[[37, 40], [85, 92]]
```

I/O with Numpy

load() and save() functions handle / numpy binary files (with npy extension)

loadtxt() and savetxt() functions handle normal text files

numpy.save()

```
import numpy as np
a = np.array([11,12,13,14,15])
np.save('outfile',a)
```

numpy.load()

```
import numpy as np
b = np.load('outfile.npy')
print(b)
```

I/O with Numpy

The storage and retrieval of array data in simple text file format is done with savetxt() and loadtxt() functions.

np.loadtxt()

```
a = np.array([1,2,3,4,5])
np.savetxt('out.txt',a)
```

np.loadtxt()

```
b = np.loadtxt('out.txt')
print(type(b))
print(b)
```

```
<class 'numpy.ndarray'>
[1. 2. 3. 4. 5.]
```

Complete Machine Learning Course (Introduction)

- Numpy
- Pandas
- Matplotlib
- Sklearn Machine Learning Library
- Model Deployment

Reading Mixed data Using Numpy

```
from numpy import genfromtxt
my_data = genfromtxt("music.csv",delimiter=',')
```

```
my_data
```

```
array([[nan, nan, nan],
       [20.,  1., nan],
       [23.,  1., nan],
       [25.,  1., nan],
       [26.,  1., nan],
       [29.,  1., nan],
       [30.,  1., nan],
       [31.,  1., nan],
       [33.,  1., nan],
       [37.,  1., nan],
       [20.,  0., nan],
       [21.,  0., nan]
```

```
from numpy import genfromtxt
my_data = genfromtxt("music.csv",delimiter=',',skip_header=1,
                     dtype=[('f0', '<i4'), ('f1', '<i4'),('f4', '|U10')])
```

```
my_data
```

```
array([(20, 1, 'HipHop'), (23, 1, 'HipHop'), (25, 1, 'HipHop'),
       (26, 1, 'Jazz'), (29, 1, 'Jazz'), (30, 1, 'Jazz'),
       (31, 1, 'Classical'), (33, 1, 'Classical'), (37, 1, 'Classical'),
       (20, 0, 'Dance'), (21, 0, 'Dance'), (25, 0, 'Dance'),
       (26, 0, 'Acoustic'), (27, 0, 'Acoustic'), (30, 0, 'Acoustic'),
       (31, 0, 'Classical'), (34, 0, 'Classical'), (35, 0, 'Classical')],
      dtype=[('f0', '<i4'), ('f1', '<i4'), ('f4', '<U10')])
```

Introduction to Jupyter Notebook

YouTube video uploading process stuck at 0% or 99%

The standard definition (SD) taking too much time to process your video


Microsoft
MTA
Introduction to Programming
Using Python
2020


scikit
learn


CISCO
CCNA


Red Hat

Numpy.resize() Vs. array.resize()

- np.resize() function is used to create a new array with the specified shape.
- If the new array is larger than the original array, then the new array is filled with repeated copies of elements.

- array.resize() function is used to create a new array with the specified shape.
- If the new array is larger than the original array, then the new array is filled with zeros.

Numpy.resize() Vs. array.resize()

- If the new array is larger than the original array, then the new array is filled with repeated copies of elements.

```
1  import numpy as np
```

```
1  x = np.arange(10)
```

```
1  v= np.resize(x, (5,3))
```

```
1  v
```

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

- If the new array is larger than the original array, then the new array is filled with zeros.

```
1  import numpy as np
```

```
1  x = np.arange(10)
```

```
1  v= x.resize((5,3))
```

```
1  v
```

```
1  x
```

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

Numpy.resize() Vs. array.resize()

- **np.resize() not change the original array.**

```
1  import numpy as np
```

```
1  x = np.arange(10)
```

```
1  v= np.resize(x, (5,3))
```

```
1  v
```

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

- **array.resize() change the original array (Inplace operation).**

```
1  import numpy as np
```

```
1  x = np.arange(10)
```

```
1  v= x.resize((5,3))
```

```
1  v
```

```
1  x
```

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

Numpy.resize() Vs. array.resize()

- **np.resize() if less number of elements , consider from beginning.**

```
1 x = np.arange(10)
```

```
1 v= np.resize(x, (2,3))
```

```
1 v
```

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

- **array.resize() if less number of elements , consider from beginning.**

```
1 y = np.arange(10)
```

```
1 v= y.resize((2,3))
```

```
1 y
```

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

Aliasing and Cloning Numpy Array

Aliasing : The process of giving another reference variable to the existing numpy array is called aliasing.

Aliasing and Cloning Numpy Array

```
1 import numpy as np
```

```
1 x = np.array([10,20,30,40])
```

```
1 y=x
```

```
1 x[1]=999
```

```
1 x
```

```
array([ 10, 999,  30,  40])
```

```
1 y
```

```
array([ 10, 999,  30,  40])
```

Aliasing :

The problem in this approach is by using one reference variable if we are changing content, then those changes will be reflected in the other reference variable.

Aliasing and Cloning Numpy Array

Slicing :

```
1 import numpy as np
```

```
1 x = np.array([10,20,30,40])
```

```
1 y = x[:]
```

```
1 y[2]=888
```

```
1 x
```

```
array([ 10,  20, 888,  40])
```

```
1 y
```

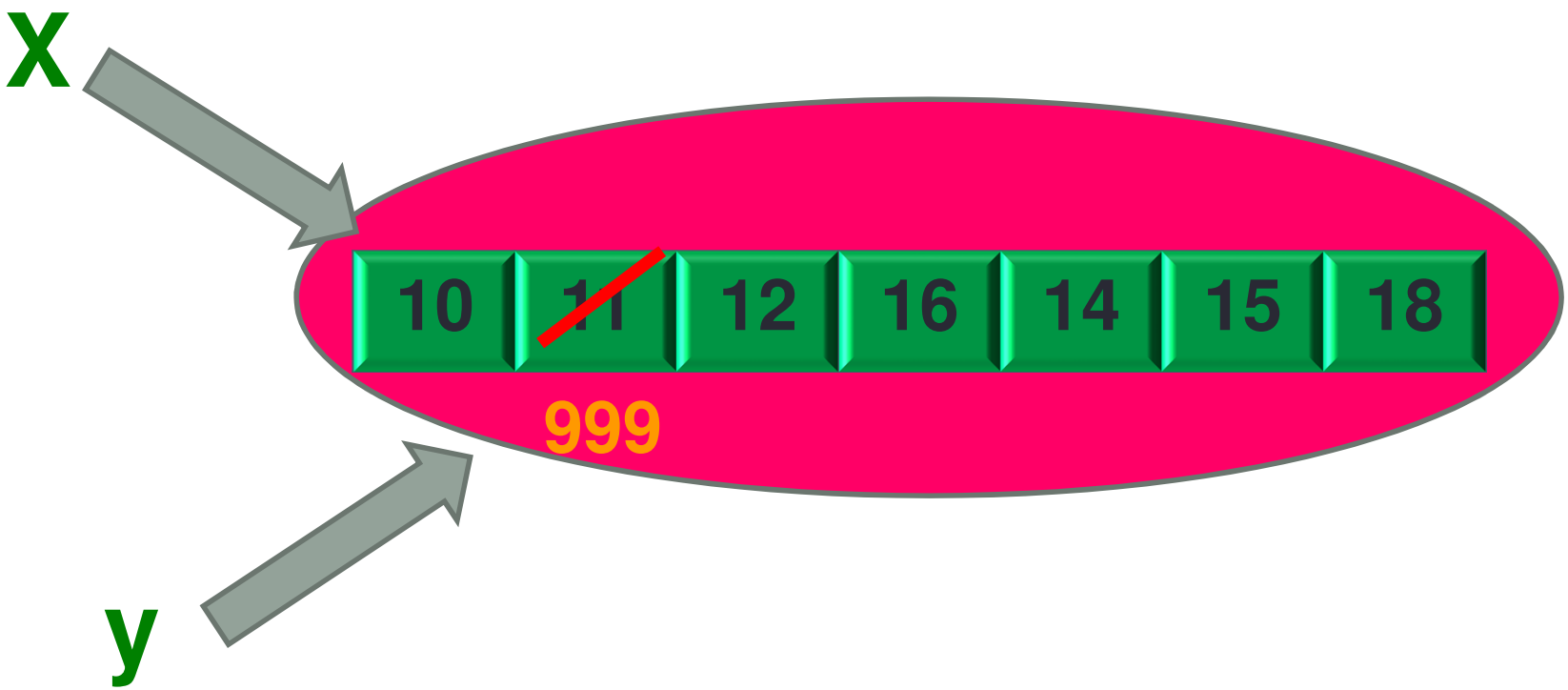
```
array([ 10,  20, 888,  40])
```

Aliasing :

The problem in this approach is by using one reference variable if we are changing content, then those changes will be reflected in the other reference variable.

Aliasing and Cloning Numpy Array

- Aliasing



Aliasing and Cloning Numpy Array

Cloning : To overcome this problem we have to do cloning. The process of creating the exact duplicate independent object is called cloning.

We can implement cloning using a copy() function.

Aliasing and Cloning Numpy Array : copy()

```
1 import numpy as np
```

```
1 x = np.array([10,20,30,40])
```

```
1 y = x.copy()
```

```
1 y[1]=777
```

```
1 x
```

```
array([10, 20, 30, 40])
```

```
1 y
```

```
array([ 10, 777,  30,  40])
```

The copy should not be affected by the changes made to the original array.

Aliasing and Cloning Numpy Array : copy()

```
1 import numpy as np
```

```
1 x = np.array([10,20,30,40])
```

```
1 y = np.copy(x[:])
```

```
1 x[1]=999
```

```
1 x
```

```
array([ 10, 999,  30,  40])
```

```
1 y
```

```
array([10, 20, 30, 40])
```

The copy should not be affected by the changes made to the original array.

Aliasing and Cloning Numpy Array

Alert

- Note that np.copy() is a shallow copy and will not copy object elements within arrays. This is mainly important for arrays containing Python objects.
- **Solution** : deepcopy()

copy() vs. view() : Numpy

- **Copy()**
- **Any changes made to the copy will not affect original array.**

- **View()**
- **Any changes made to the view will affect the original array**

ndarray.base

view()

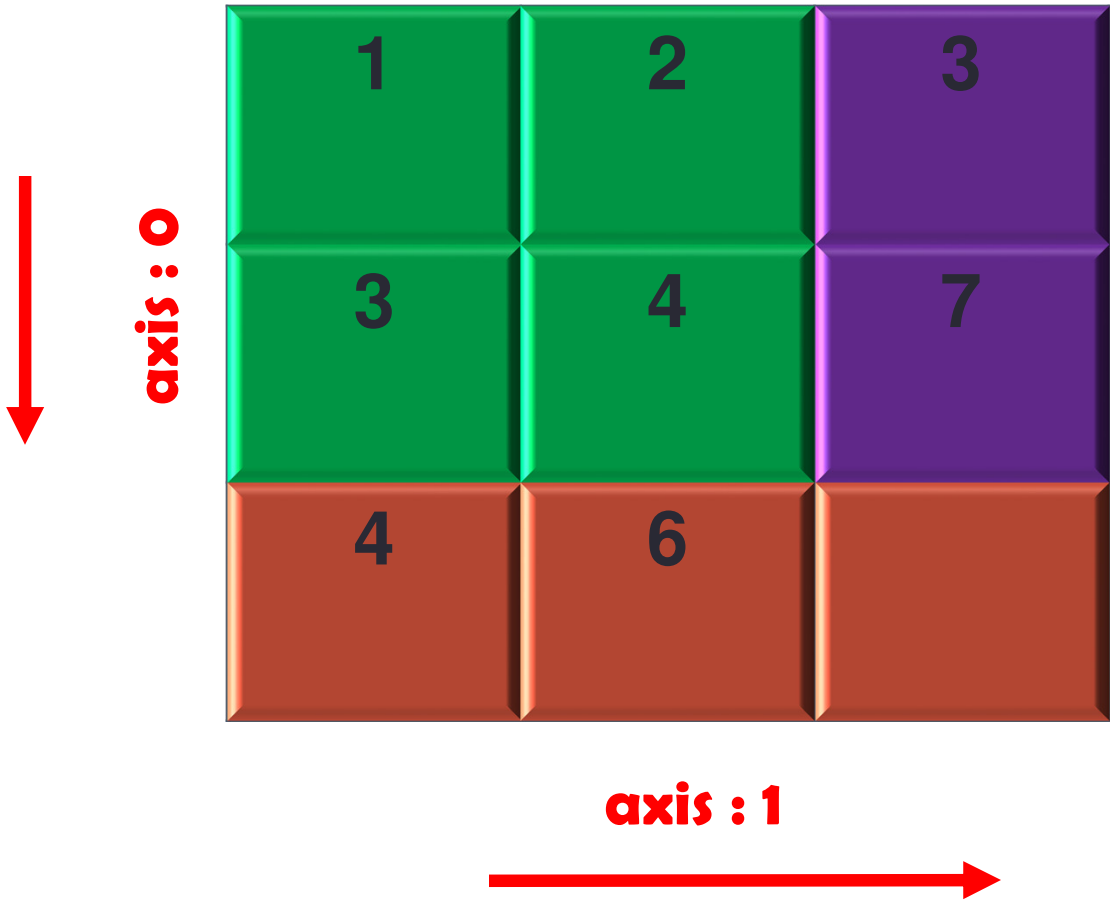
- Changes made to the view will affect the original array.
- View is just a view of the original array.

copy()

- Any changes made to the copy will not affect original array.
- Copy is a new array.

Copy Vs. Deepcopy in Numpy

Axis in numpy



axis = 0 means along the column and axis = 1 means working along the row.

Axis in numpy

Axes are defined for arrays with more than one dimension

Numpy: vstack (row-wise)

vstack

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

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

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

Numpy: hstack (column-wise)

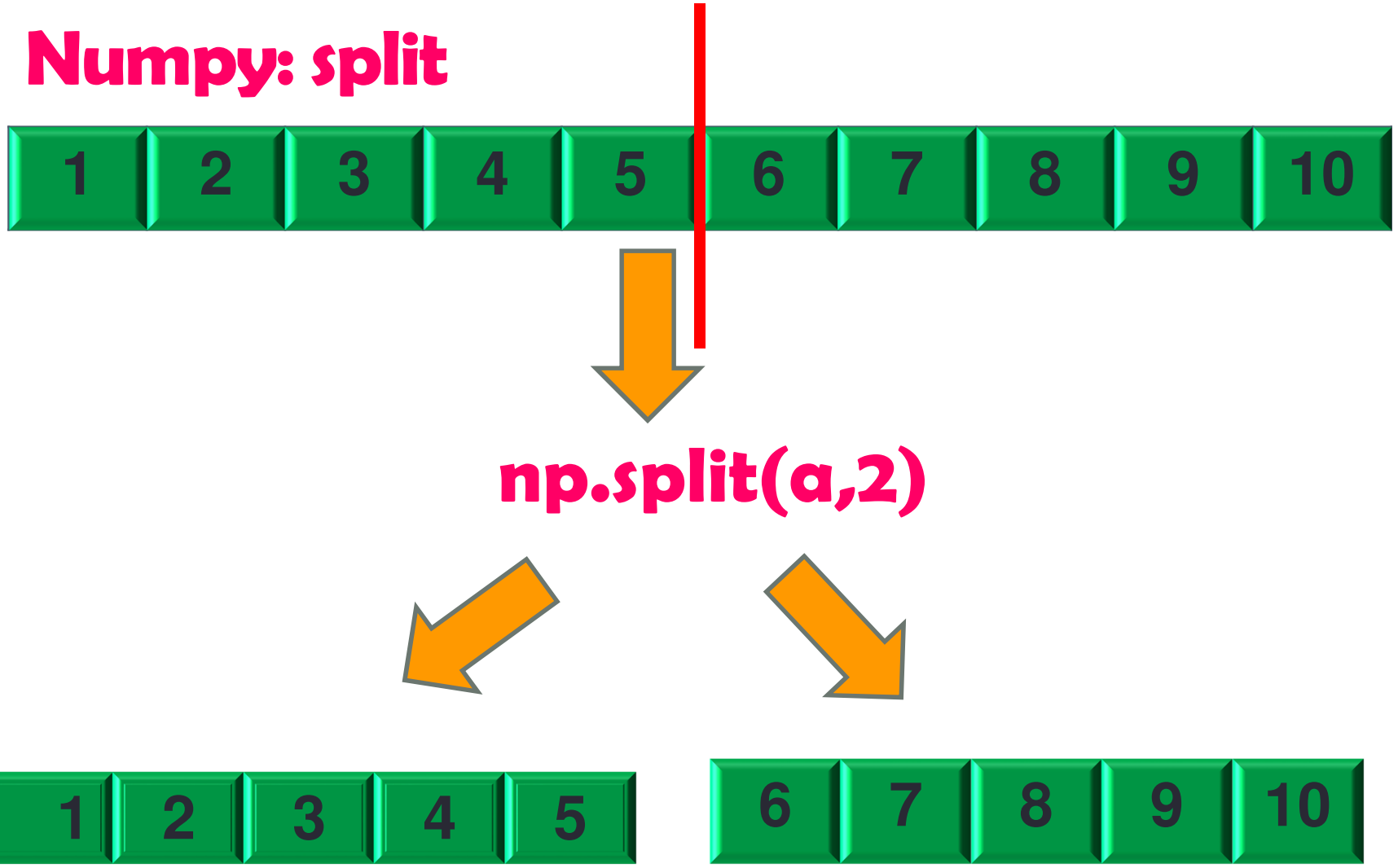
hstack

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

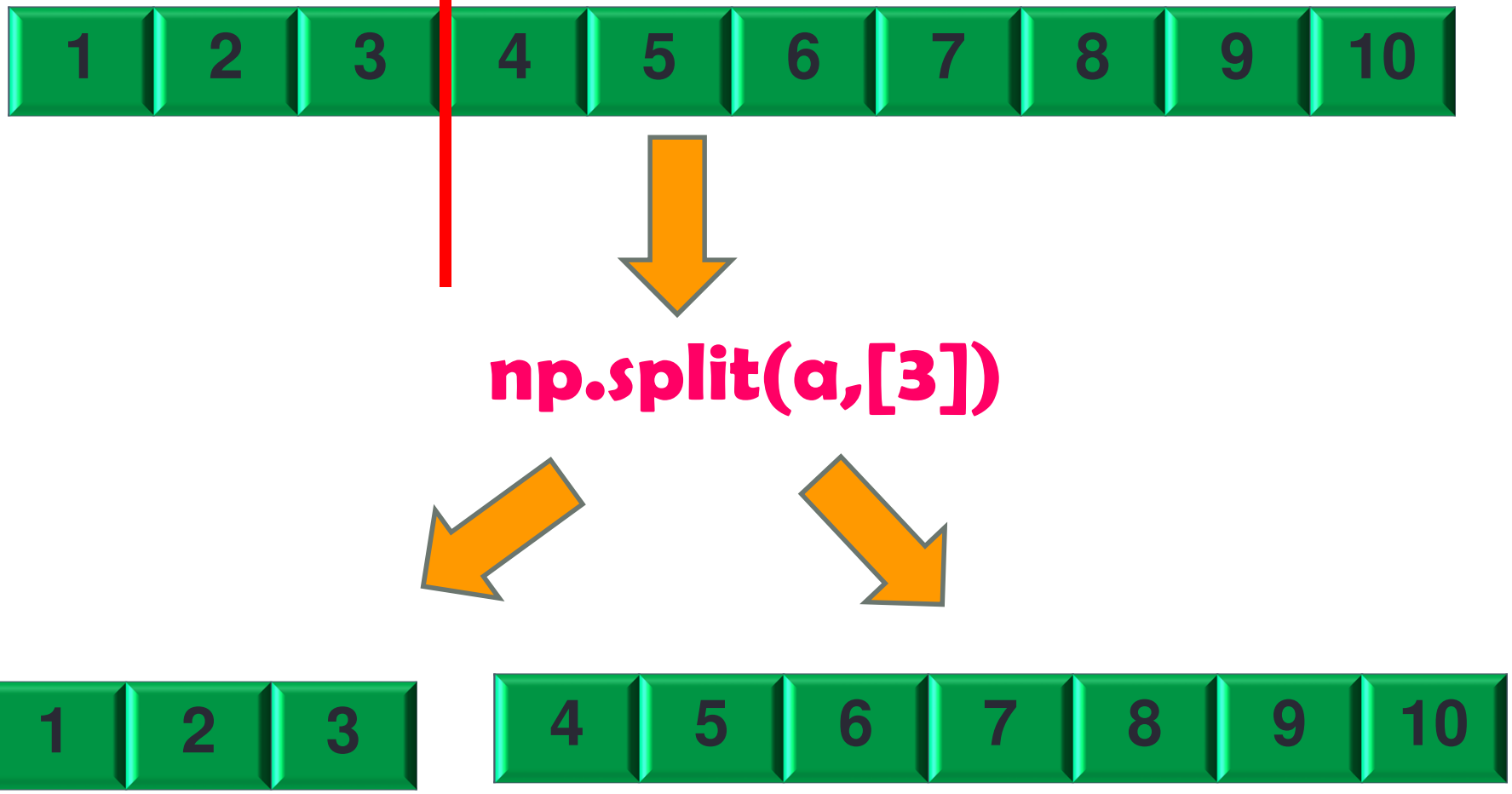
```
np.hstack((a,b))
```

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

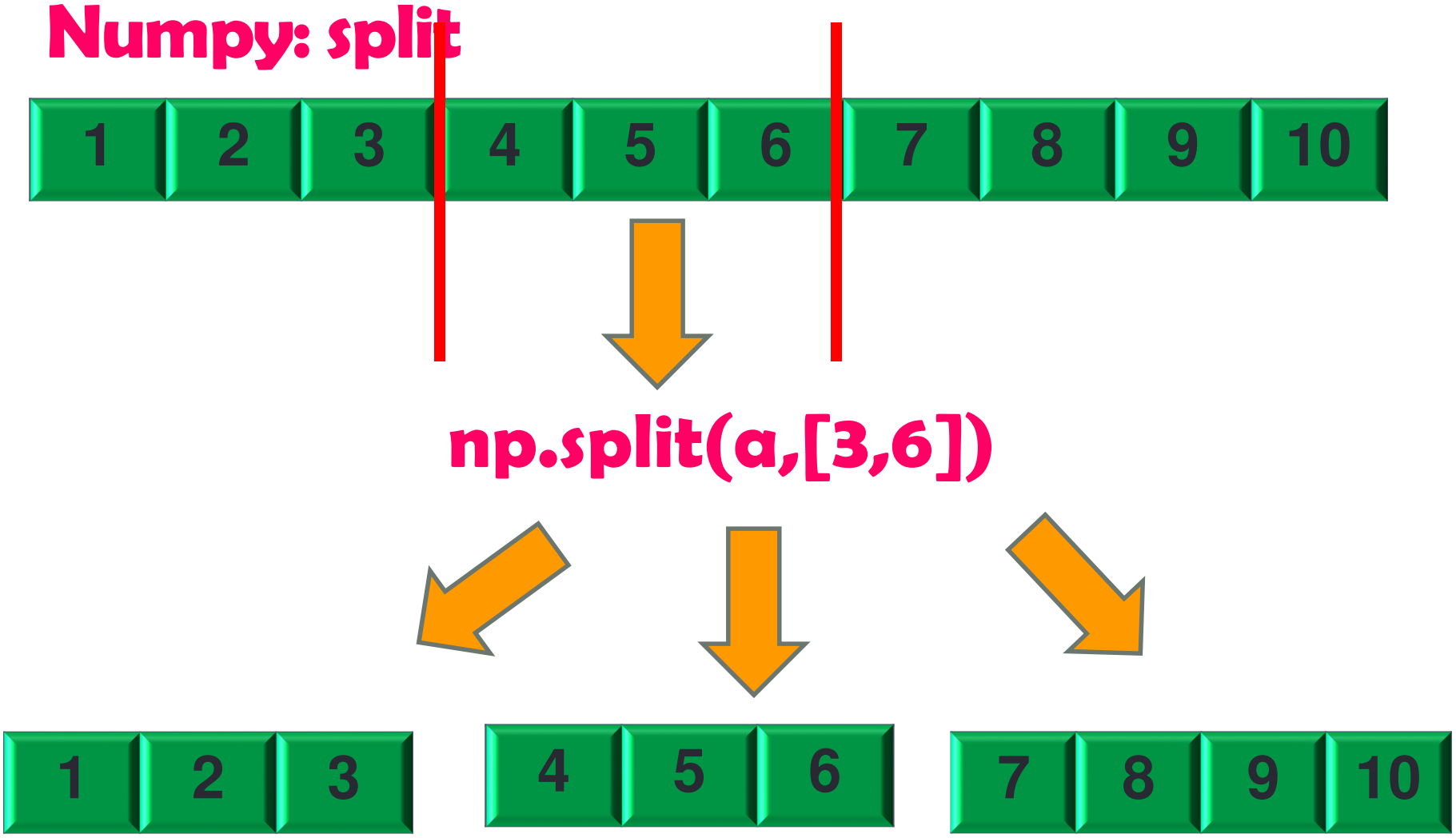
Numpy: split



Numpy: split



Numpy: split



Numpy: split (practical)

```
a= np.arange(1,11)
a
```

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

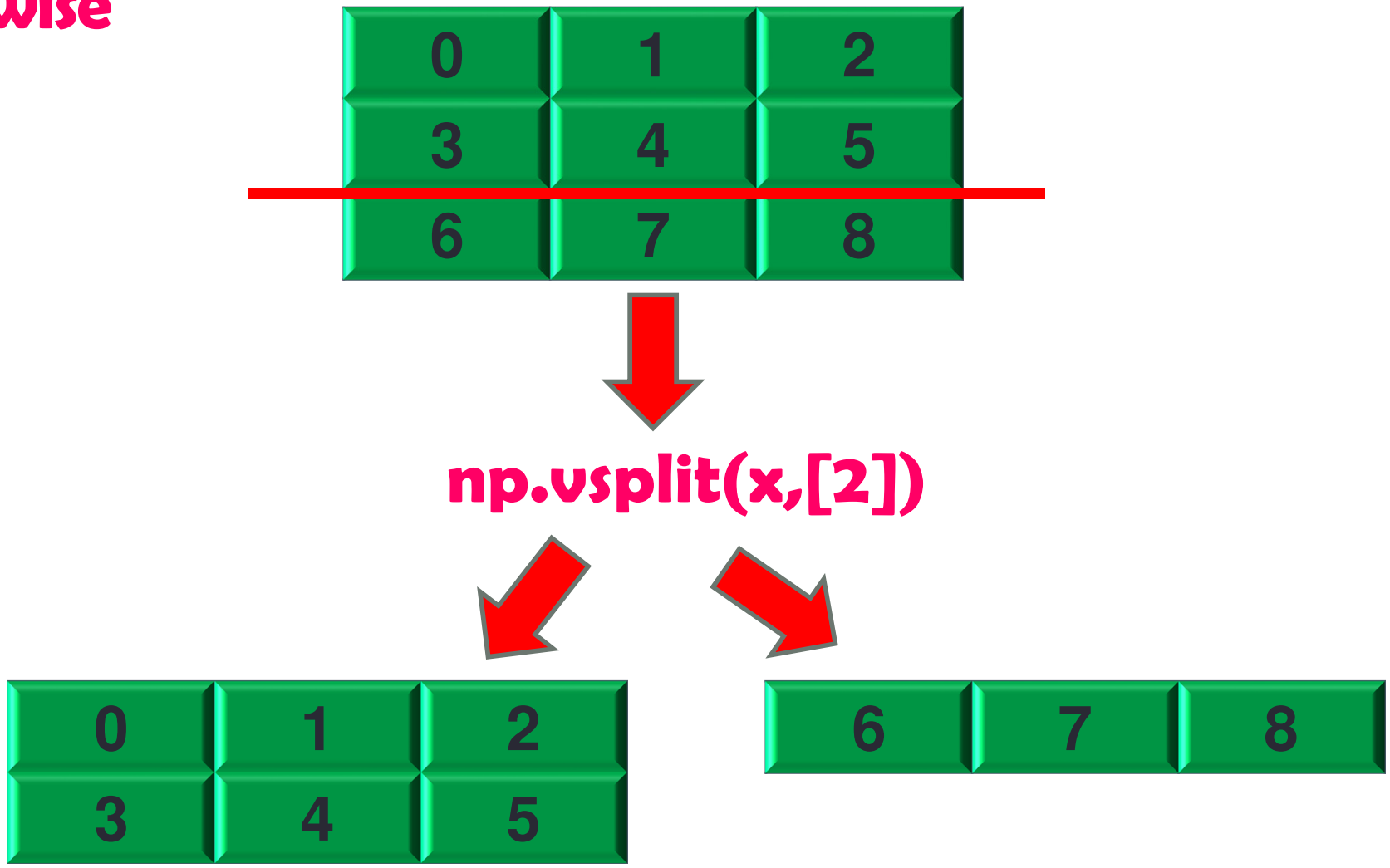
```
np.split(a,[3])
```

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

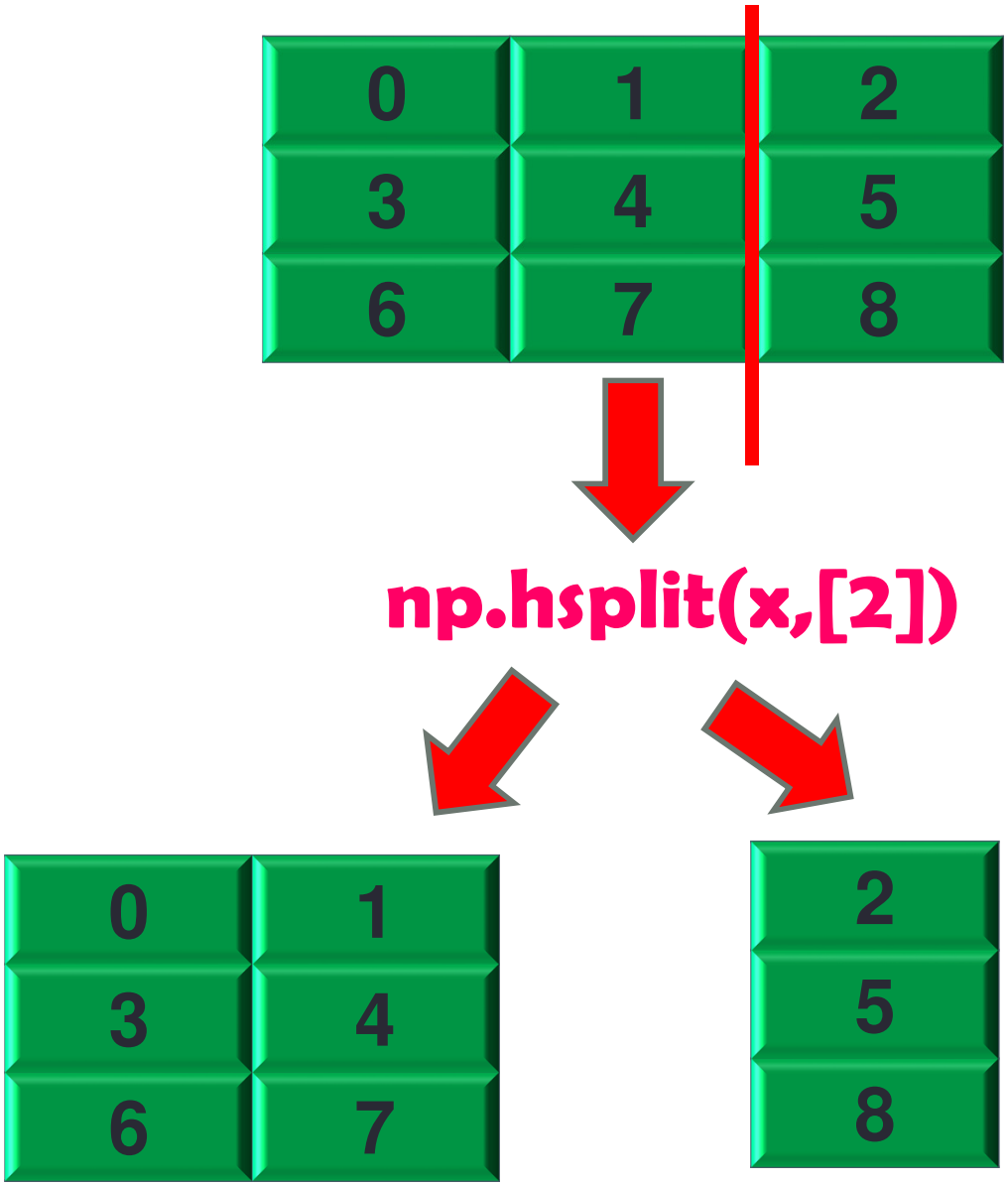
```
np.split(a,[3,6])
```

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

Numpy: np.vsplit == np. Split (axis=0)) row-wise



Numpy: np.hsplit == np. Split (axis=1)) column-wise



Numpy: vsplit (row) Vs. hsplit(column)

```
x= np.arange(9).reshape(3,3)
x
```

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

```
result = np.vsplit(x,[2])
```

```
result[0]
```

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

```
result[1]
```

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

```
x= np.arange(9).reshape(3,3)
x
```

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

```
result = np.hsplit(x,[2])
```

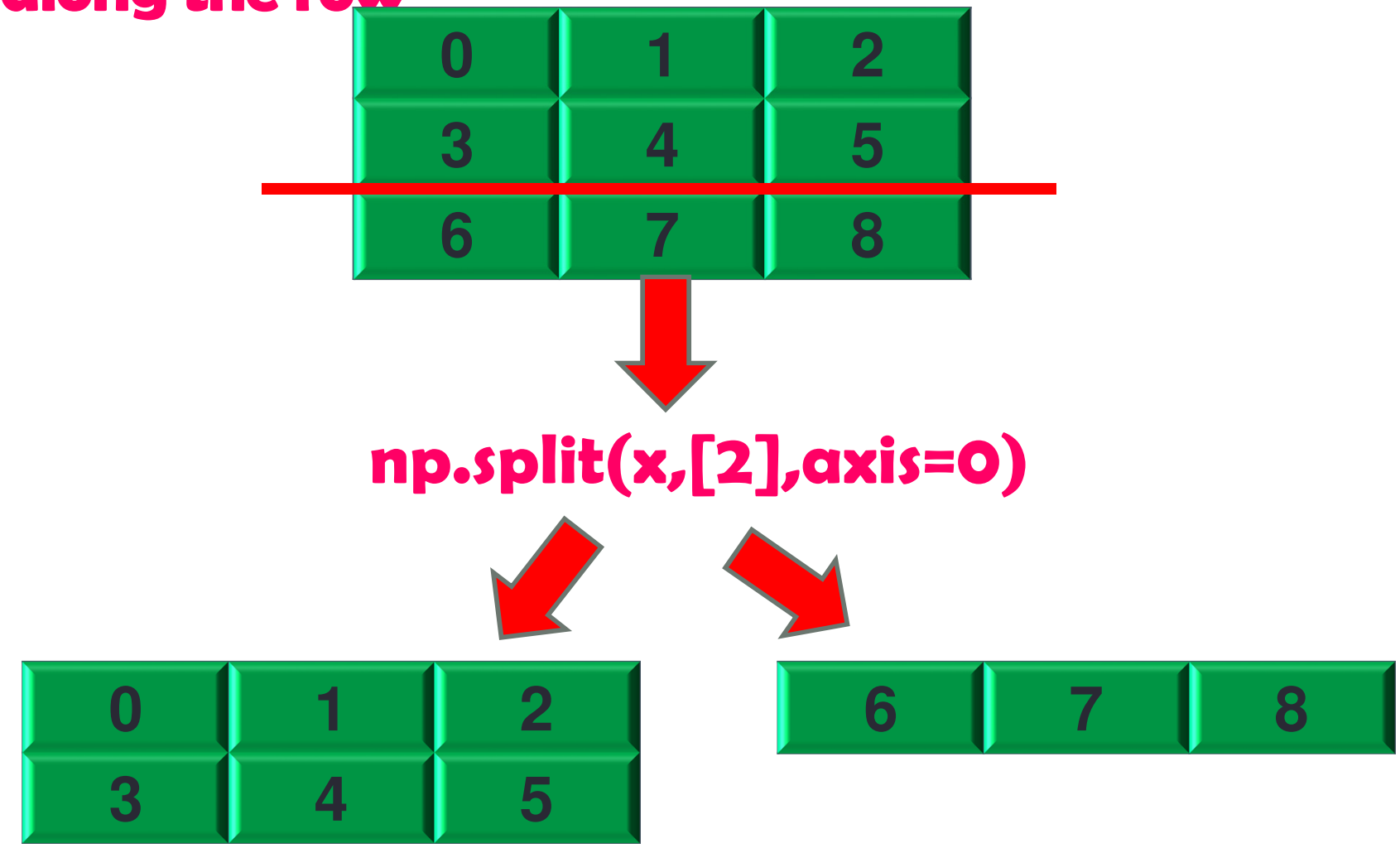
```
result[0]
```

```
array([[0, 1],
       [3, 4],
       [6, 7]])
```

```
result[1]
```

```
array([[2],
       [5],
       [8]])
```

Numpy : np. Split (x, int/position ,axis=0) cut along the row



Numpy: np. Split (x, int/position,axis=1) cut along the column