Mathematical Computing with Python
(NumPy)

Learning Objectives

By the end of this lesson, you will be able to:

- Explain NumPy and its importance
- Discuss the basics of NumPy, including its fundamental objects
- Demonstrate how to create and print a NumPy array
- Analyze and perform basic operations in NumPy
- Utilize shape manipulation and copying methods
- Demonstrate how to execute linear algebraic functions
- Build basic programs using NumPy

Quick Recap: Lists

Below are some of the properties of lists:

List

```
distance=[10,15,17,26]
time=[.30,.47,.55,1.20]
```

Collection of values

Multiple types (heterogeneous)

Add, remove, update

Limitations of Lists

You can change individual values in a list, but you cannot apply a mathematical operation over the entire list.

```
distance=[10,15,17,26]
time=[.30,.47,.55,1.20]
```

```
speed=distance/time
```

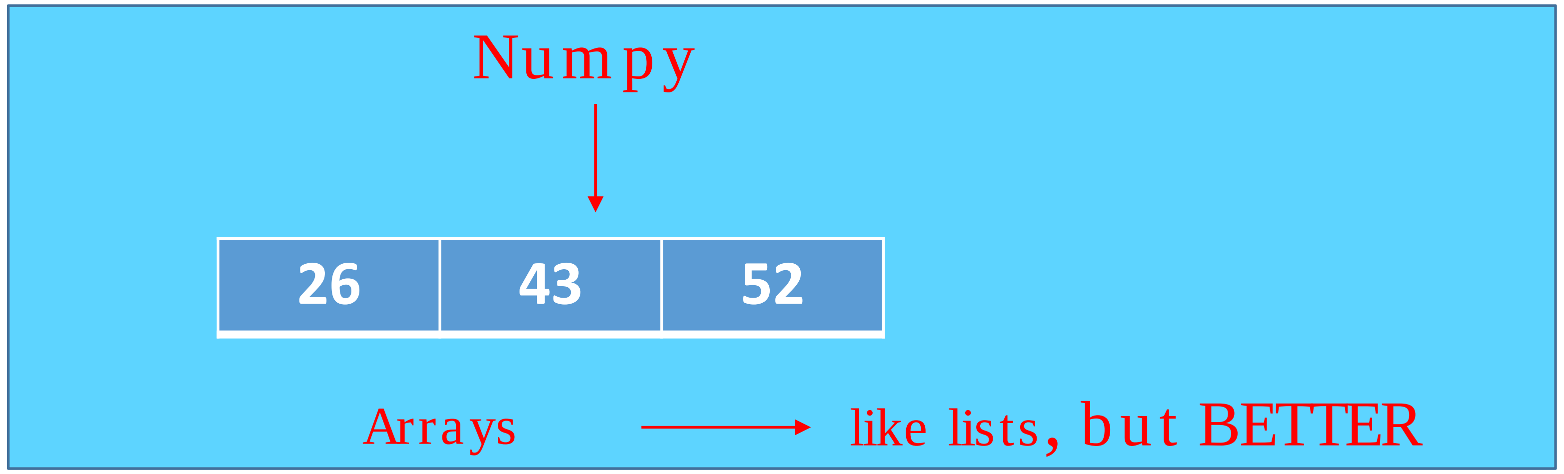
Mathematical operation over the entire “distance” and “time” lists

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-b779bad68500> in <module>()
----> 1 speed=distance/time

TypeError: unsupported operand type(s) for /: 'list' and 'list'
```

Error

Why NumPy

Numerical Python (NumPy) supports multidimensional arrays over which you can easily apply mathematical operations.




```
distance=[10,15,17,26]
time=[.30,.47,.55,1.20]
```

```
import numpy as np
```

Import NumPy

```
np_distance = np.array(distance)
np_time=np.array(time)
speed=np_distance/np_time
```

Create “distance” and “time” NumPy arrays

Mathematical function applied over the entire “distance” and “time” arrays

```
speed
```

```
array([ 33.33333333,  31.91489362,  30.90909091,  21.66666667])
```

Output

NumPy Overview

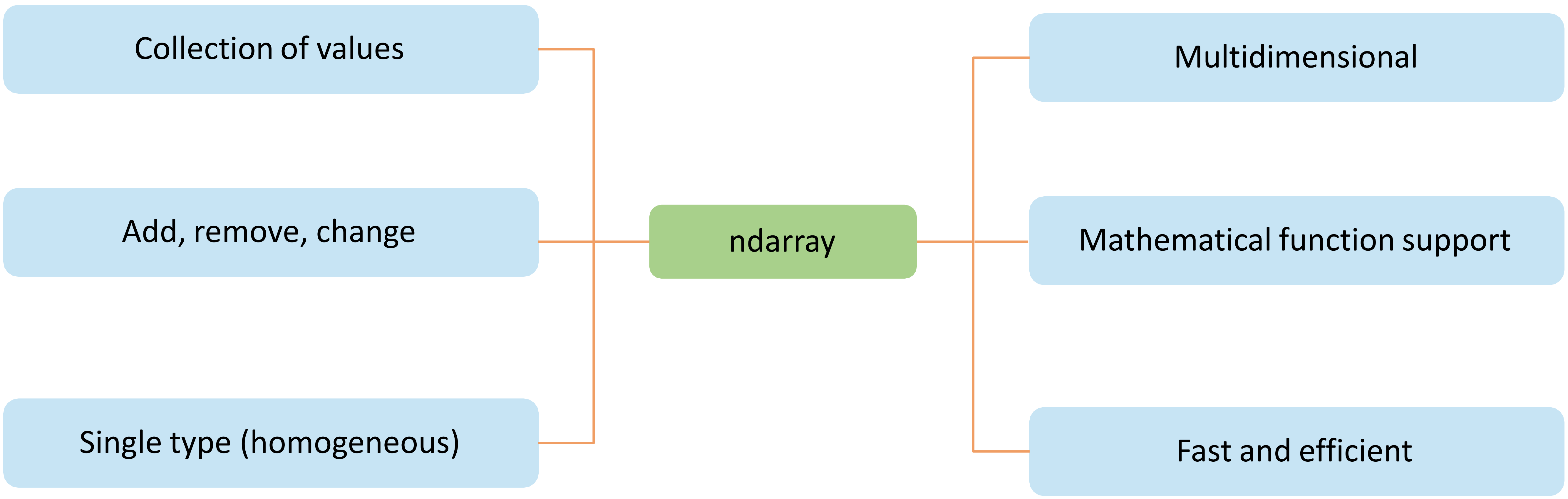
NumPy is the foundational package for mathematical computing in Python.

It has the following properties:

- Supports fast and efficient multidimensional arrays (ndarray)
- Executes element -wise computations and mathematical calculations
- Performs linear algebraic operations, Fourier transforms, and random number generation
- Tools for reading/writing array based datasets to disk
- Efficient way of storing and manipulating data
- Tools for integrating language codes (C, C++)

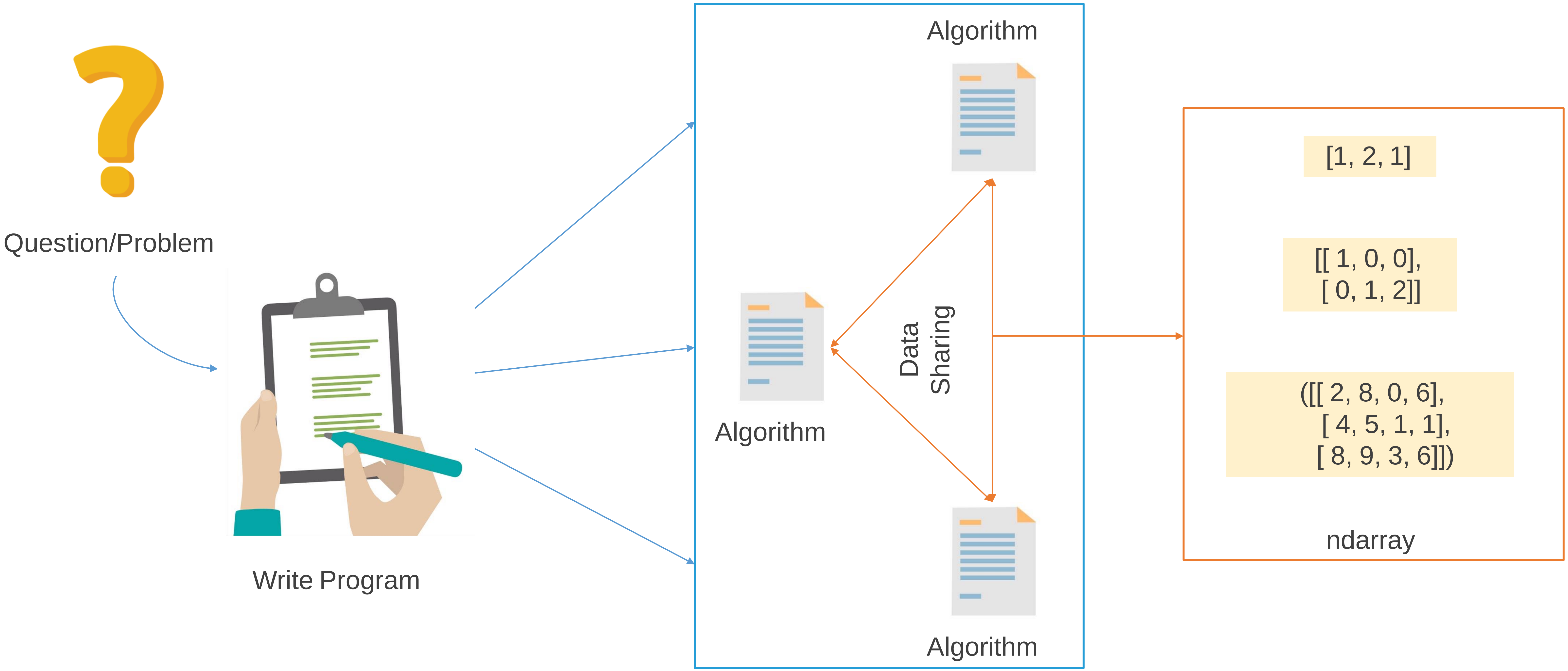
Properties of ndarray

An array in NumPy has the following properties:



Purpose of ndarray

The ndarray in Python is used as the primary container to exchange data between algorithms.



Types of Arrays

Arrays can be one -dimensional, two -dimensional, three -dimensional, or multi -dimensional.

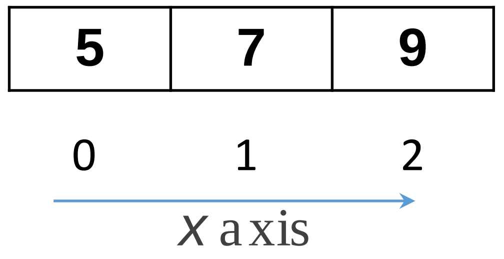
One-Dimensional Array

Printed as rows

array([5, 7,9])

1 axis
rank 1

Length = 3



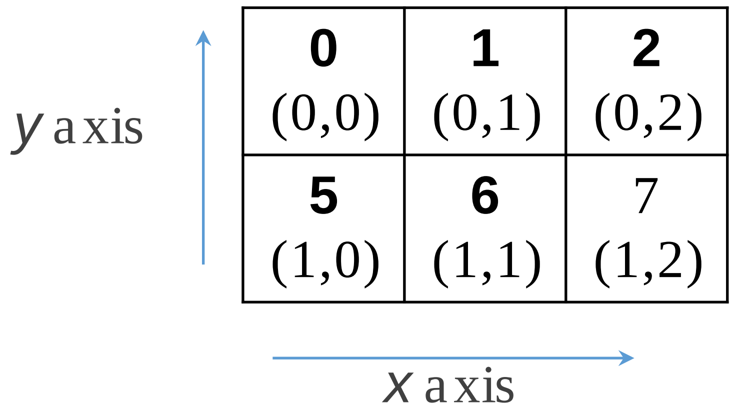
Two-Dimensional Array

Printed as matrices (2x3)

array([[0, 1, 2],
[5, 6, 7]])

2 axes
rank 2

Length = 3



Three-Dimensional Array

Printed as list of matrices (3x3x3)

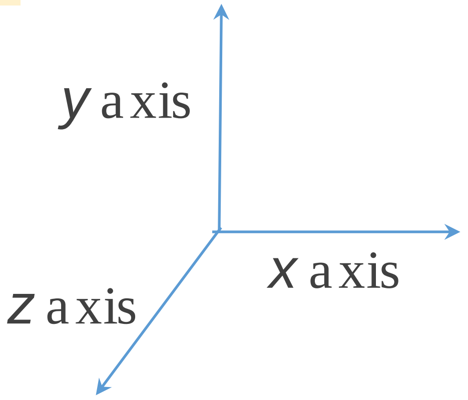
array([[[0, 1, 2],
[3, 4, 5],
[6, 7, 8]],

[[9, 10, 11],
[12, 13, 14],
[15, 16, 17]],

[[18, 19, 20],
[21, 22, 23],
[24, 25, 26]]])

3 axes
rank 3

Length = 3



Activity: Sequence it Right!

The code here is buggy. You must correct its sequence to debug it.

1

```
distance=[10,15,17,26]
time=[.30,.47,.55,1.20]
```

2

```
np_distance = np.array(distance)
np_time=np.array(time)
```

3

```
import numpy as np
```

4

```
speed=np_distance/np_time
```

```
speed
```

```
array([ 33.33333333,  31.91489362,  30.90909091,  21.66666667])
```

Activity: Sequence it Right!

1
```
distance=[10,15,17,26]
time=[.30,.47,.55,1.20]
```

2
```
import numpy as np
```

3
```
np_distance = np.array(distance)
np_time=np.array(time)
```

4
```
speed=np_distance/np_time
```

```
speed
```

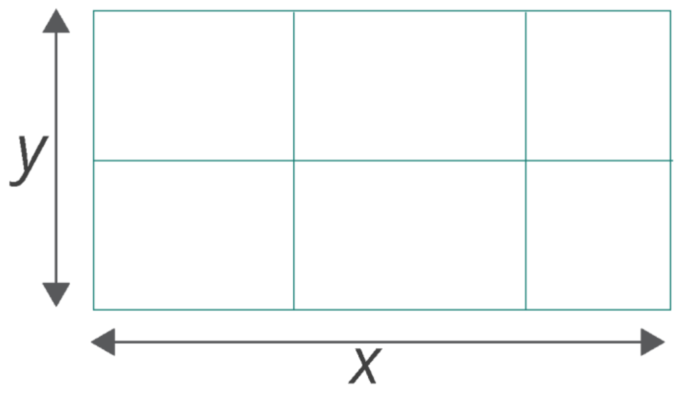
```
array([ 33.33333333,  31.91489362,  30.90909091,  21.66666667])
```

Classes and Attributes of ndarray: .ndim

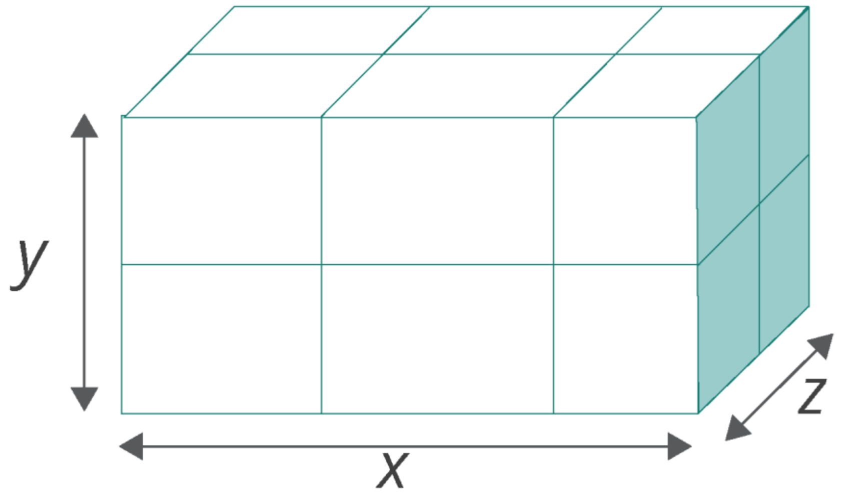
Numpy's array class is **ndarray** , also referred to as **numpy.ndarray** . The attributes of ndarray are:

- ndarray.ndim
- ndarray.shape
- ndarray.size
- ndarray.dtype

This refers to the number of axes (dimensions) of the array. It is also called the rank of the array.



Two axes or 2D array



Three axes or 3D array

Concept | Example

Classes and Attributes of ndarray: .ndim

ndarray.ndim

ndarray.shape

ndarray.size

ndarray.dtype

The array **np_city** is one-dimensional, while the array **np_city_with_state** is two-dimensional.

```
In [108]: np_city = np.array(['NYC', 'LA', 'Miami','Houston'])

In [109]: np_city.ndim
Out[109]: 1

In [110]: np_city_with_state = np.array([['NYC', 'LA', 'Miami','Houston'],['NY', 'CA', 'FL','TX']])

In [111]: np_city_with_state.ndim
Out[111]: 2
```

Concept

Example

Classes and Attributes of ndarray: .shape

Numpy's array class is **ndarray**, also referred to as **numpy.ndarray**. The attributes of ndarray are:

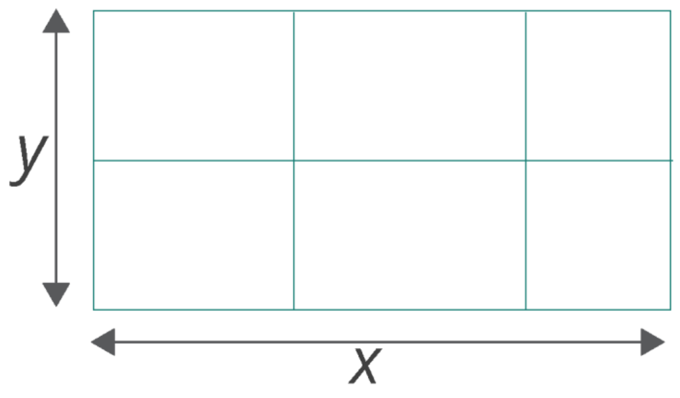
ndarray.ndim

ndarray.shape

ndarray.size

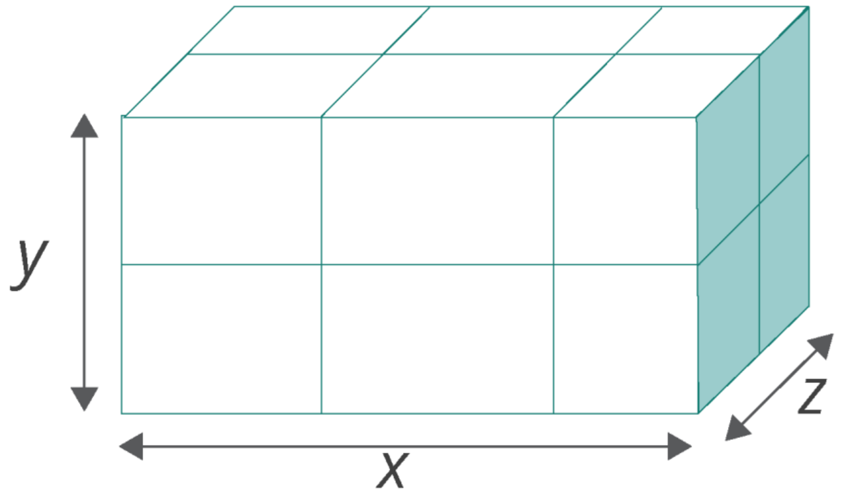
ndarray.dtype

This consists of a tuple of integers showing the size of the array in each dimension. The length of the **shape tuple** is the rank or ndim.



2 rows, 3 columns

Shape: (2, 3)



2 rows, 3 columns, 2 ranks

Shape: (2, 3, 2)

Concept

Example

Classes and Attributes of ndarray: .shape

Numpy's array class is **ndarray**, also referred to as **numpy.ndarray**. The attributes of ndarray are:

- ndarray.ndim
- ndarray.shape
- ndarray.size
- ndarray.dtype

The shape tuple of both the arrays indicate their size along each dimension.

```
In [108]: np_city = np.array(['NYC', 'LA', 'Miami','Houston'])

In [110]: np_city_with_state = np.array([['NYC', 'LA', 'Miami','Houston'],['NY', 'CA', 'FL','TX']])

In [112]: np_city.shape
Out[112]: (4L,)

In [113]: np_city_with_state.shape
Out[113]: (2L, 4L)
```

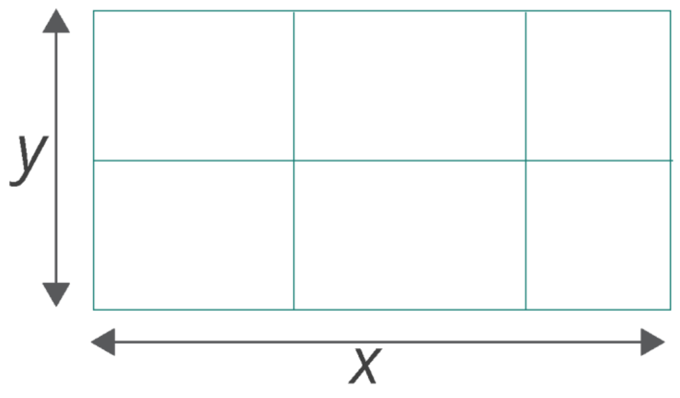
Concept | Example

Classes and Attributes of ndarray: .size

Numpy's array class is **ndarray** , also referred to as **numpy.ndarray** . The attributes of ndarray are:

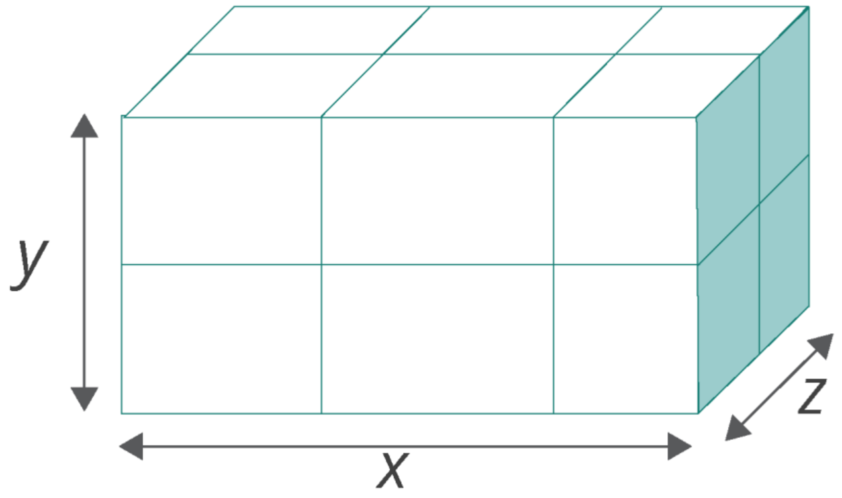
- ndarray.ndim
- ndarray.shape
- **ndarray.size**
- ndarray.dtype

It gives the total number of elements in the array. It is equal to the product of the elements of the shape tuple.



Array contains 6 elements

Array a = (2, 3)
Size = 6



Array contains 12 elements

Array b = (2, 3, 2)
Size = 12

Concept Example

Classes and Attributes of ndarray: .size

Numpy's array class is **ndarray**, also referred to as **numpy.ndarray**. The attributes of ndarray are:

- ndarray.ndim
- ndarray.shape
- ndarray.size
- ndarray.dtype

Look at the examples to see how the shape tuples of the arrays are used to calculate their size.

```
In [112]: np_city.shape
Out[112]: (4L,)

In [113]: np_city_with_state.shape
Out[113]: (2L, 4L)

In [114]: np_city.size
Out[114]: 4

In [115]: np_city_with_state.size
Out[115]: 8
```

Concept   Example

Classes and Attributes of ndarray: .dtype

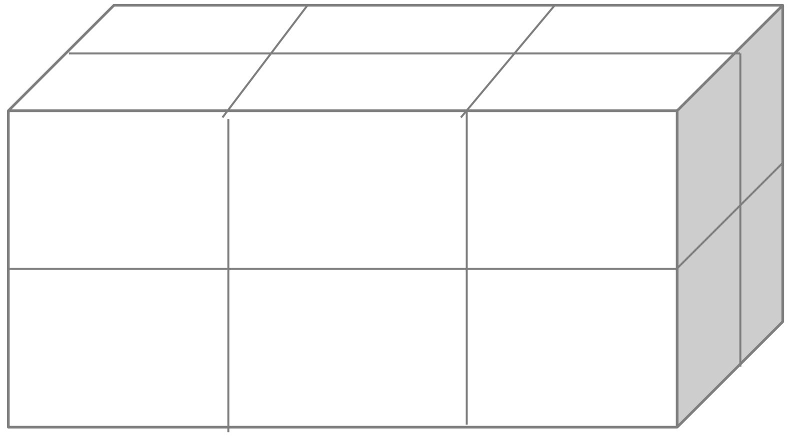
Numpy's array class is **ndarray** , also referred to as **numpy.ndarray** . The attributes of ndarray are:

- ndarray.ndim
- ndarray.shape
- ndarray.size
- ndarray.dtype

It's an object that describes the type of the elements in the array. It can be created or specified using Python.

Array contains integers

Array a = [3, 7, 4]
[2, 1, 0]



Array contains floats

Array b = [1.3, 5.2, 6.7]
[0.2, 8.1, 9.4]

[2.6, 4.2, 3.9]
[7.8, 3.4, 0.8]

Concept | Example

Classes and Attributes of ndarray: .dtype

Numpy's array class is **ndarray** , also referred to as **numpy.ndarray** . The attributes of ndarray are:

- ndarray.ndim
- ndarray.shape
- ndarray.size
- ndarray.dtype

Both the arrays are of **string** data type (dtype) and the longest string is of length 7, which is **Houston** .

```
In [116]: np_city
Out[116]: array(['NYC', 'LA', 'Miami', 'Houston'],
                dtype='|S7')

In [117]: np_city_with_state
Out[117]: array([['NYC', 'LA', 'Miami', 'Houston'],
                 ['NY', 'CA', 'FL', 'TX']],
                dtype='|S7')

In [118]: np_city_with_state.dtype
Out[118]: dtype('S7')
```

Concept Example

Basic Operations

Using the following operands, you can easily apply various mathematical, logical, and comparison operations on an array.

Mathematical Operations

Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

Logical Operations

And	&
Or	|
Not	~

Comparison Operations

Greater	>
Greater or equal	>=
Less	<
Less or equal	<=
Equal	==
Not equal	!=

Basic Operations: Example

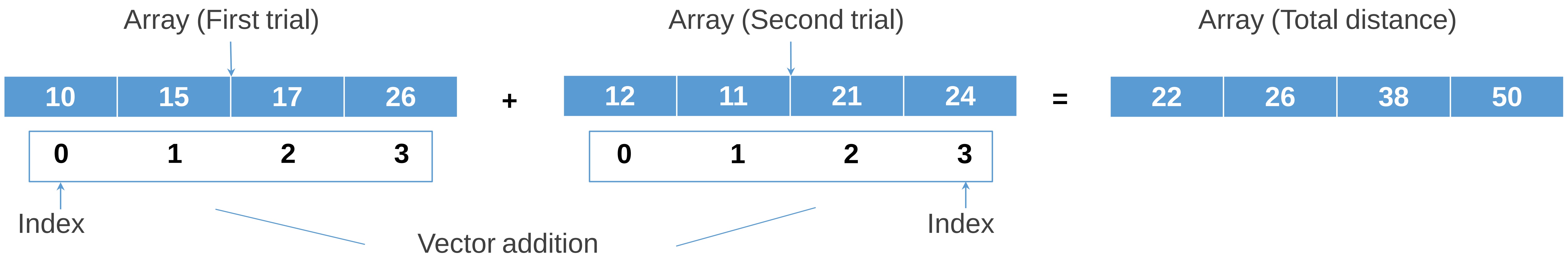
NumPy uses the indices of the elements in each array to carry out basic operations. In this case, where we are looking at a dataset of four cyclists during two trials, vector addition of the arrays gives the required output.

```
In [99]:  first_trial_cyclist =[10,15,17,26]
In [100]: second_trial_cyclist =[12,11,21,24]
In [101]: np_first_trial_cyclist = np.array(first_trial_cyclist)
In [102]: np_second_trial_cyclist = np.array(second_trial_cyclist)
In [103]: np_first_trial_cyclist+np_second_trial_cyclist
Out[103]: array([22, 26, 38, 50])
```

First trial

Second trial

Total distance



Accessing Array Elements: Indexing

You can access an entire row of an array by referencing its axis index.

1st set data   2nd set data

```
In [117]: cyclist_trials = np.array([[10,15,17,26],[12,11,21,24]])
```

Create 2D array using cyclist trial data shown earlier

```
In [118]: first_trial =cyclist_trials[0]
```

First trial data

```
In [119]: first_trial
Out[119]: array([10, 15, 17, 26])
```

```
In [120]: second_trial = cyclist_trials[1]
```

Second trial data

```
In [121]: second_trial
Out[121]: array([12, 11, 21, 24])
```

2D array containing cyclists'data

10	15	17	26	First trial (axis 0)
12	11	21	24	Second trial (axis 1)

Accessing Array Elements: Indexing

You can refer the indices of the elements in an array to access them. You can also select a particular index of more than one axis at a time.

```
In [122]: first_cyclist_firstTrial = cyclist_trials[0][0]
```

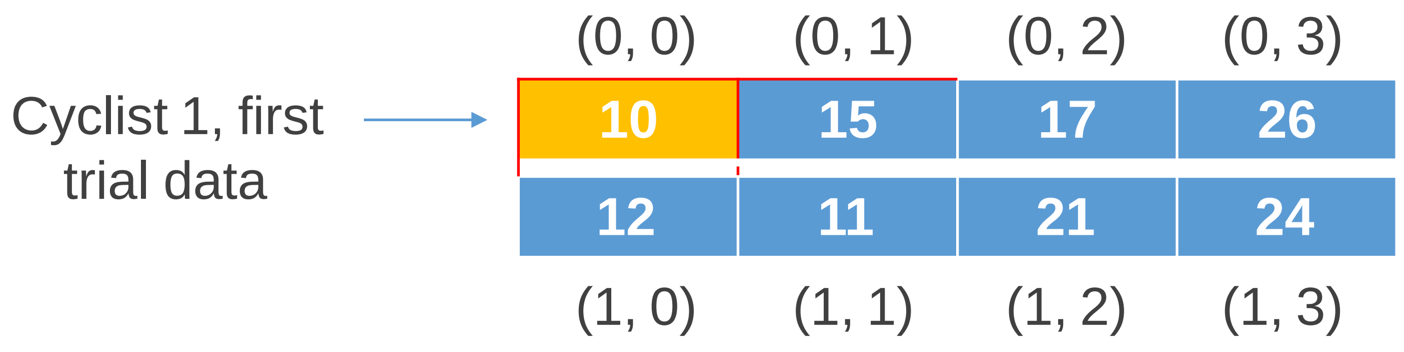
First cyclist: first trial data

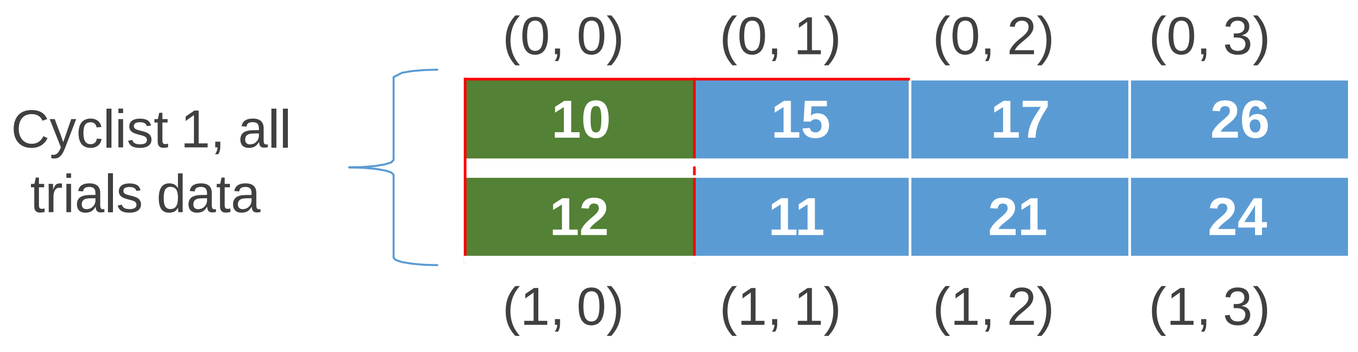
```
In [123]: first_cyclist_firstTrial
Out[123]: 10
```

```
In [124]: first_cyclist_all_trials = cyclist_trials[:,0]
```

First cyclist: all trial data
(Use **:** to select all the rows of an array)

```
In [125]: first_cyclist_all_trials
Out[125]: array([10, 12])
```





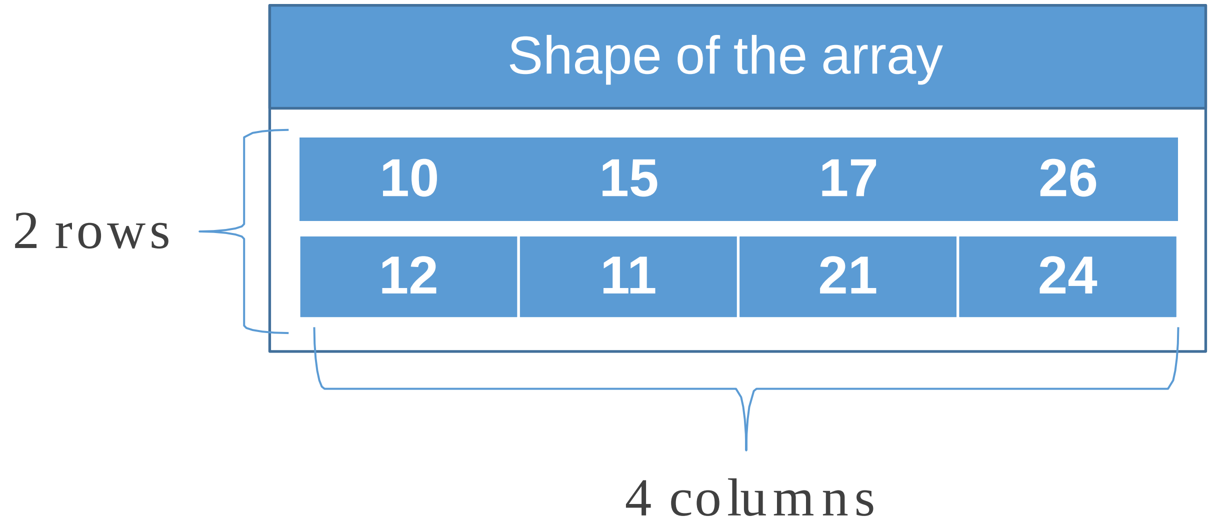
Accessing Array Elements: Slicing

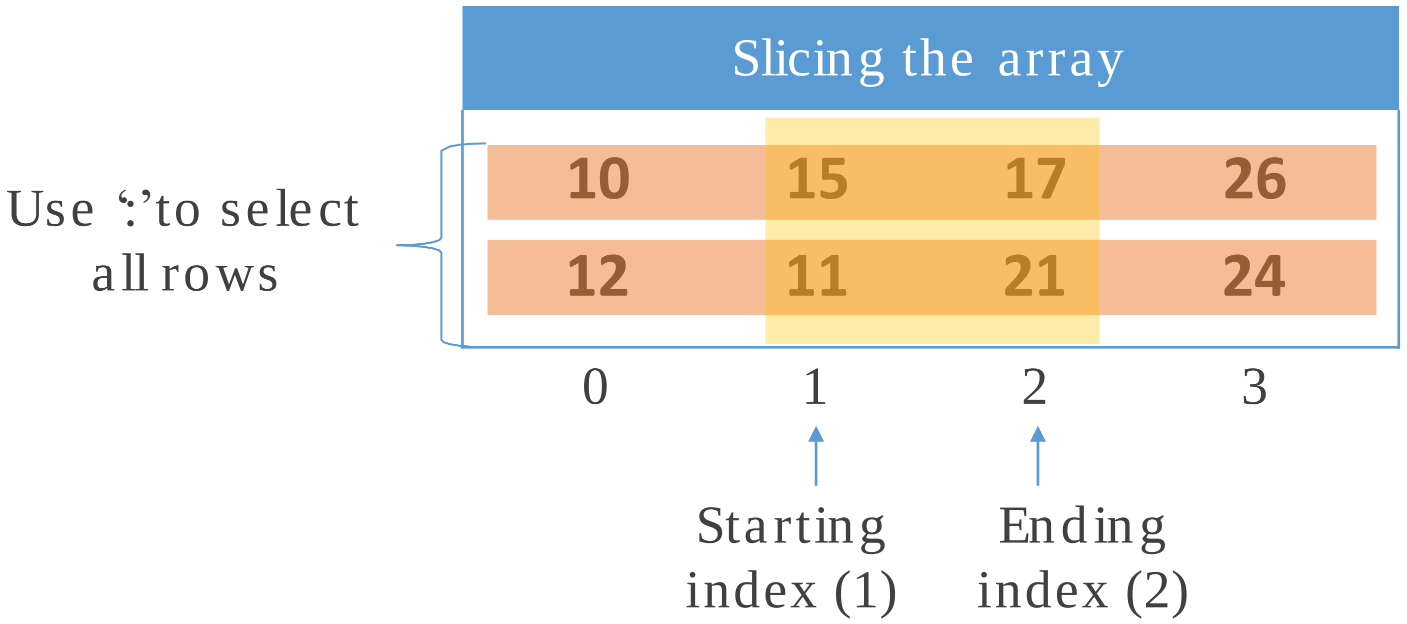
Use the slicing method to access a range of values within an array.

Shape of the array

```
In [152]: cyclist_trials.shape
Out[152]: (2L, 4L)

In [153]: two_cyclist_trial_data=cyclist_trials[:,1:3]

In [154]: two_cyclist_trial_data
Out[154]: array([[15, 17],
                 [11, 21]])
```

Slicing the array data [: , 1 : 3]
where 1 is inclusive but 3 is not





Activity: Slice It!

Select any two elements from the array to see how the statement required to slice the range changes.

Rules of the Game

- Choose the first element of the range. Then, choose the element that ends the range.
- See how the values in the statement change according to your choices.
- Refresh to try again.

```
example_array[1:3]
```

Select any two elements from the array.

Accessing Array Elements: Iteration

Use the iteration method to go through each data element present in the dataset.

```
In [117]: cyclist_trials = np.array([[10,15,17,26],[12,11,21,24]])

In [153]: two_cyclist_trial_data=cyclist_trials[:,1:3]

In [154]: two_cyclist_trial_data

Out[154]: array([[15, 17],
                 [11, 21]])

In [159]: for iterate_cyclist_trials_data in cyclist_trials:
              print (iterate_cyclist_trials_data)

          [10 15 17 26]
          [12 11 21 24]

In [160]: for iterate_two_cyclist_trial_data in two_cyclist_trial_data:
              print (iterate_two_cyclist_trial_data)

          [15 17]
          [11 21]
```

Iterate with **for loop** through entire dataset

Iterate with **for loop** through the **two cyclist** datasets

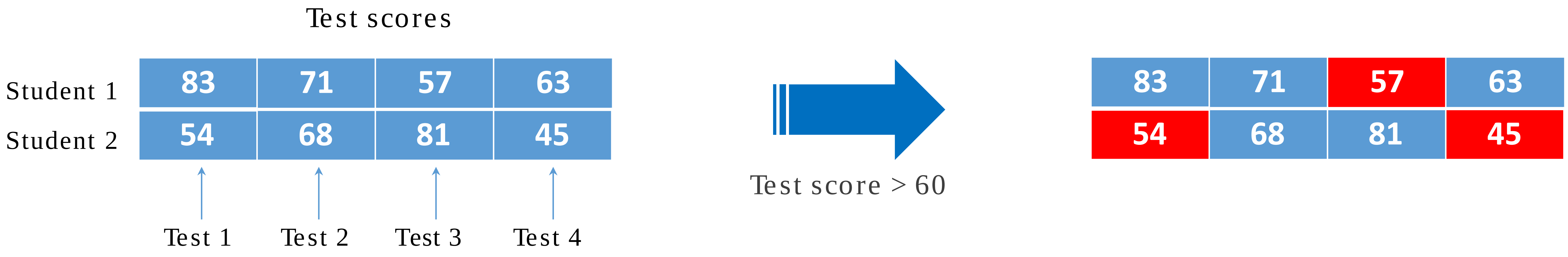
Indexing with Boolean Arrays

Boolean arrays are useful when you need to select a dataset according to set criteria.

True False

Indexing with Boolean Arrays

Here, the original dataset contains test scores of two students. You can use a Boolean array to choose on ly the scores that are above a given value.




```
In [234]: test_scores =np.array([[83,71,57,63],[54,68,81,45]])
```

```
In [235]: passing_score = test_scores>60
```

Setting the passing score

```
In [236]: passing_score
Out[236]: array([[ True,  True, False,  True],
                 [False,  True,  True, False]], dtype=bool)
```

Shows data elements which fit the criteria (Boolean array)

```
In [237]: test_scores[passing_score]
Out[237]: array([83, 71, 63, 68, 81])
```

Send **passing score** as an argument to **test scores** object

Copy and Views

When working with arrays, data is copied into new arrays only in some cases. Following are the three possible scenarios:

- Simple Assignments
- View/Shallow Copy
- Deep Copy

In this method, a variable is directly assigned the value of another variable. No new copy is made.

```
In [303]: NYC_Borough = np.array(['Manhattan','Bronx','Brooklyn','Staten Island','Queens'])

In [294]: NYC_Borough

Out[294]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Queens'],
                dtype='|S13')
```

Original dataset

```
In [295]: Boroughs_in_NYC = NYC_Borough

In [296]: Boroughs_in_NYC

Out[296]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Queens'],
                dtype='|S13')
```

Assigned dataset

```
In [297]: Boroughs_in_NYC is NYC_Borough

Out[297]: True
```

Shows both objects are the same

Copy and Views

Simple Assignments

View/Shallow Copy

Deep Copy

A view, also referred to as a shallow copy, creates a new array object.

```
In [296]: Boroughs_in_NYC
Out[296]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Queens'],
          dtype='|S13')
```

Original dataset

```
In [298]: View_of_Borough_in_NYC = Boroughs_in_NYC.view()

In [299]: len(View_of_Borough_in_NYC)
Out[299]: 5

In [300]: View_of_Borough_in_NYC[4] ='Central Park'
```

Change value in **view** object

```
In [301]: View_of_Borough_in_NYC
Out[301]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Central Park'],
          dtype='|S13')

In [302]: Boroughs_in_NYC
Out[302]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Central Park'],
          dtype='|S13')
```

Original dataset changed

Copy and Views

Simple Assignments

View/Shallow Copy

Deep Copy

Copy is also called **deep copy** because it entirely copies the original dataset. Any change in the copy will not affect the original dataset.

```
In [304]: Copy_of_NYC_Borough = NYC_Borough.copy()

In [305]: Copy_of_NYC_Borough is NYC_Borough
Out[305]: False

In [306]: Copy_of_NYC_Borough.base is NYC_Borough
Out[306]: False

In [307]: Copy_of_NYC_Borough[4]='Central Park'

In [308]: NYC_Borough
Out[308]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Queens'],
      dtype='|S13')

In [309]: Copy_of_NYC_Borough
Out[309]: array(['Manhattan', 'Bronx', 'Brooklyn', 'Staten Island', 'Central Park'],
      dtype='|S13')
```

Shows **copy** and original object are different

Shows **copy** object data is not owned by the original dataset

Change value in **copy**

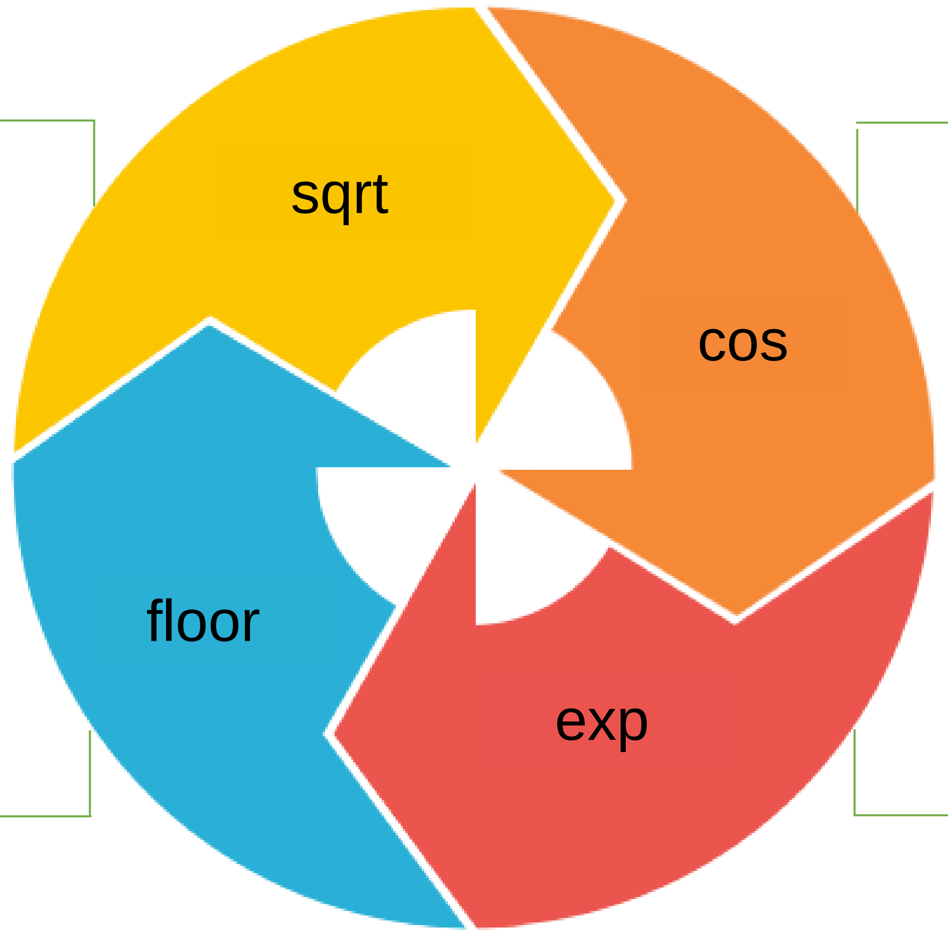
Copy object changed

Original dataset retained

Universal Functions (ufunc)

NumPy provides useful mathematical functions called Universal Functions. These functions operate element-wise on an array, producing another array as output. Some of these functions are:

sqrt function provides the square root of every element in the array.



cos function gives cosine values for all elements in the array.

floor function returns the largest integer value of every element in the array.

exp function performs exponentiation on each element.

Ufunc: Examples

```
In [186]: np_sqrt = np.sqrt([2,4,9,16])
```

Numbers for which square root will be calculated

```
In [187]: np_sqrt
Out[187]: array([ 1.41421356,  2.        ,  3.        ,  4.        ])
```

Square root values

```
In [188]: from numpy import pi
          np.cos(0)
Out[188]: 1.0
```

Import pi*

```
In [189]: np.sin(pi/2)
Out[189]: 1.0

In [190]: np.cos(pi)
Out[190]: -1.0
```

Trigonometric functions

```
In [191]: np.floor([1.5,1.6,2.7,3.3,1.1,-0.3,-1.4])
Out[191]: array([ 1.,  1.,  2.,  3.,  1., -1., -2.])
```

Return the floor of the input element wise

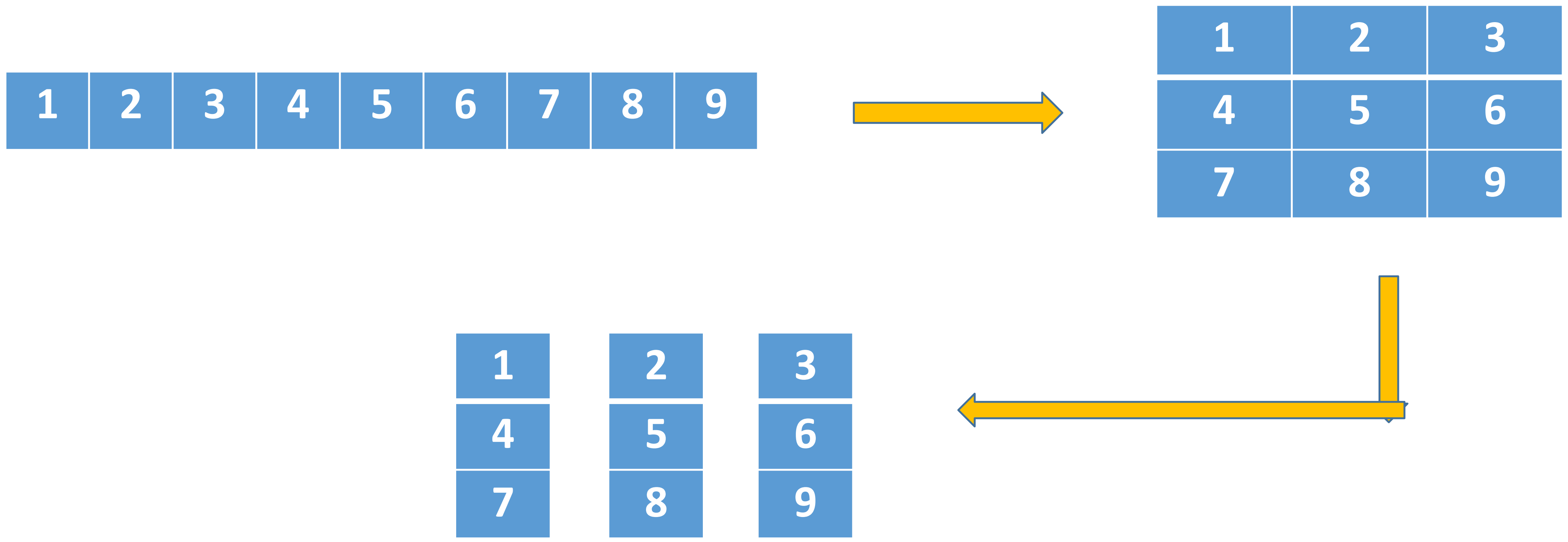
```
In [192]: np.exp([0,1,5])
Out[192]: array([   1.        ,    2.71828183,  148.4131591 ])
```

Exponential functions for complex mathematical calculations

Shape Manipulation

You can use certain functions to manipulate the shape of an array.

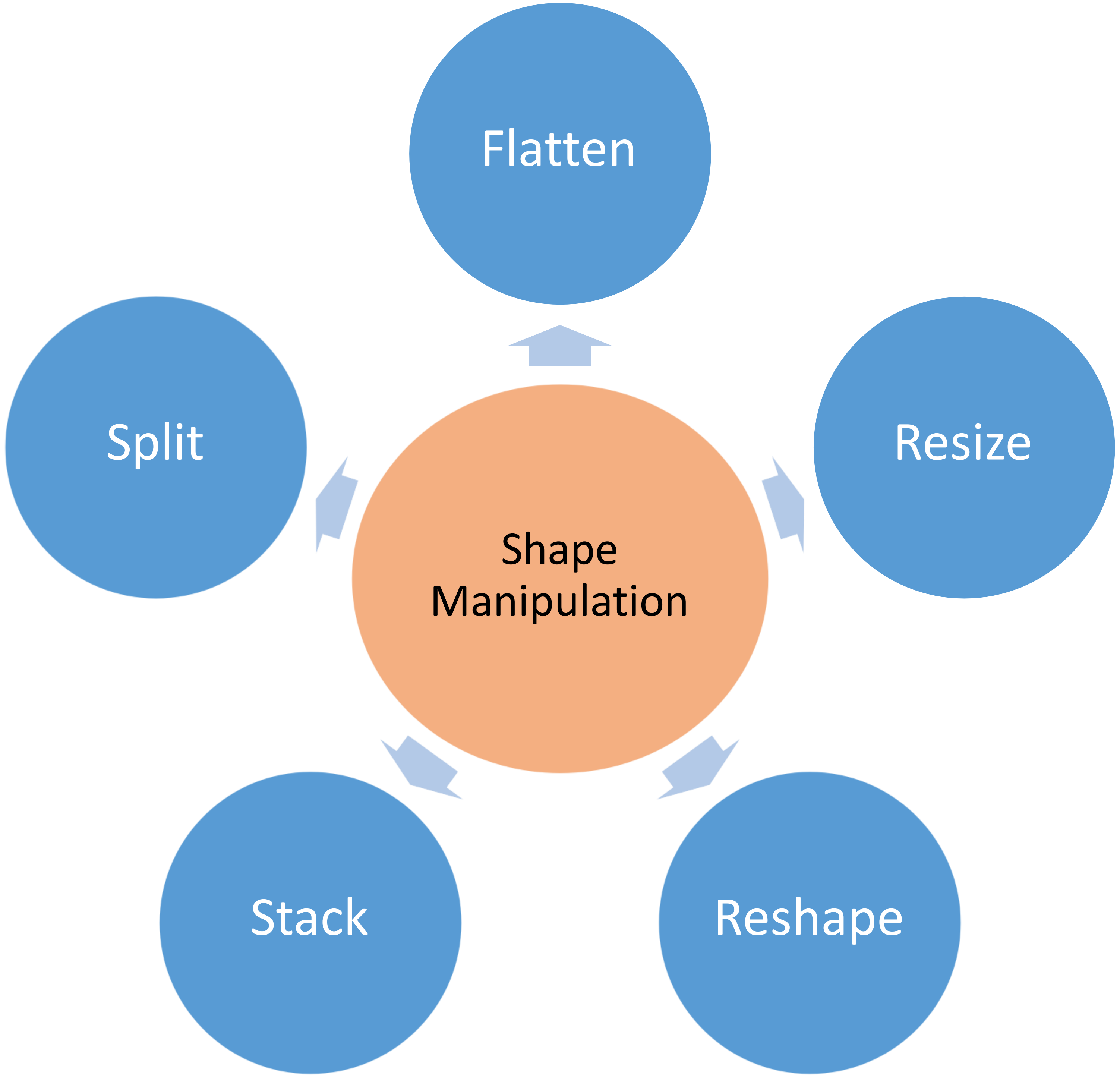
The shape of an array can be changed according to the requirement using the NumPy library functions.



Array shape manipulation methods → Data Wrangling

Shape Manipulation

Some common methods for manipulating shapes are:



Shape Manipulation: Example

You can use certain functions to manipulate the shape of an array to do the following:

```
In [383]: new_cyclist_trials = np.array([[10,15,17,26,13,19],[12,11,21,24,14,23]])
```

```
In [384]: new_cyclist_trials.ravel()
Out[384]: array([10, 15, 17, 26, 13, 19, 12, 11, 21, 24, 14, 23])
```

Flattens the dataset

```
In [385]: new_cyclist_trials.reshape(3,4)
Out[385]: array([[10, 15, 17, 26],
                 [13, 19, 12, 11],
                 [21, 24, 14, 23]])
```

Changes or reshapes the dataset to 3 rows and 4 columns

```
In [386]: new_cyclist_trials.resize(2,6)
```

Resizes again to 2 rows and 6 columns

```
In [387]: new_cyclist_trials
Out[387]: array([[10, 15, 17, 26, 13, 19],
                 [12, 11, 21, 24, 14, 23]])
```

```
In [388]: np.hsplit(new_cyclist_trials,2)
Out[388]: [array([[10, 15, 17],
                  [12, 11, 21]]), array([[26, 13, 19],
                  [24, 14, 23]])]
```

Splits the array into two

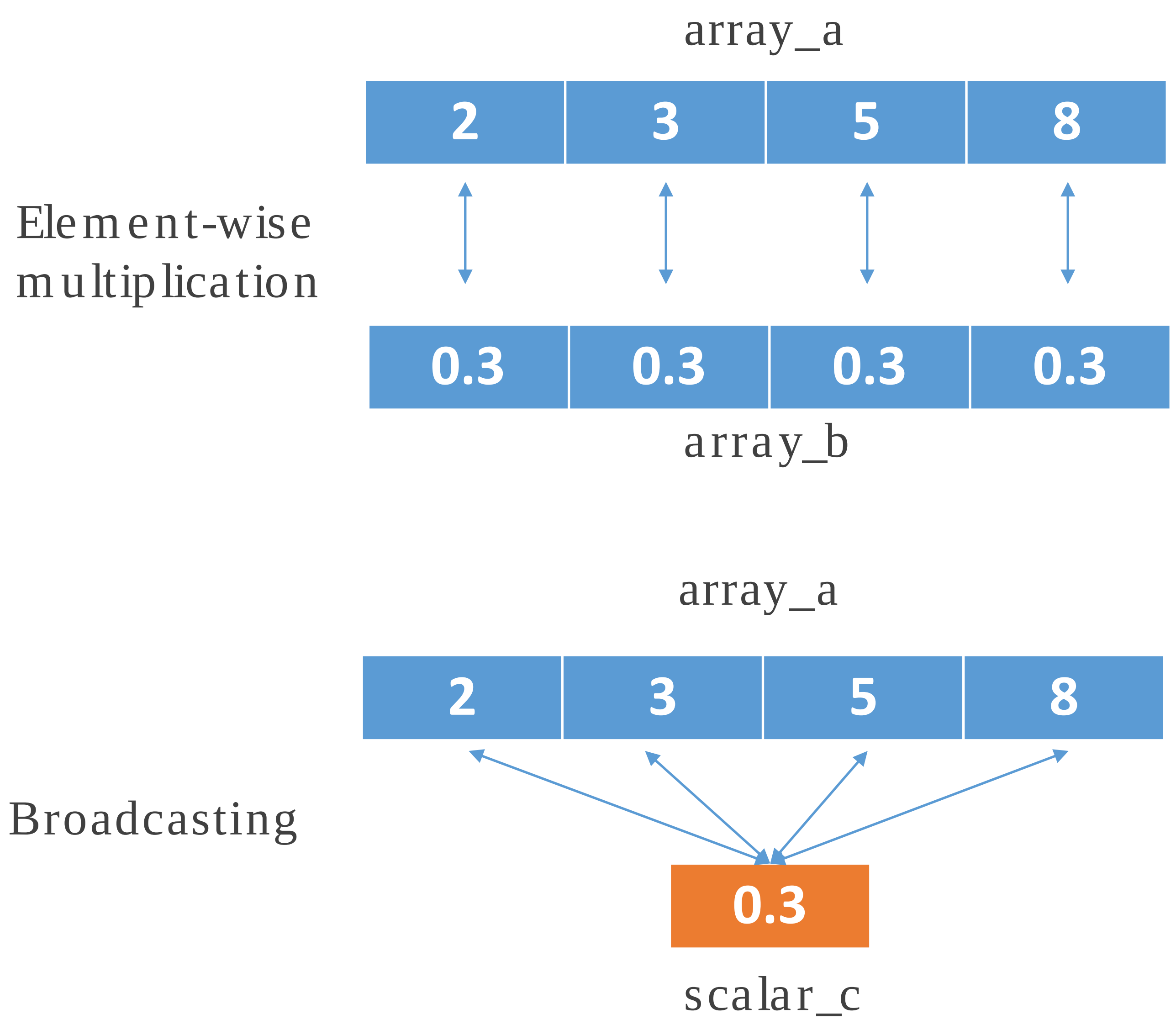
```
In [389]: new_cyclist_1 = np.array([10,15,17,26,13,19])

In [390]: new_cyclist_2 = np.array([12,11,21,24,14,23])

In [391]: np.hstack((new_cyclist_1,new_cyclist_2))
Out[391]: array([10, 15, 17, 26, 13, 19, 12, 11, 21, 24, 14, 23])
```

Stacks the arrays together

Broadcasting

NumPy uses broadcasting to carry out arithmetic operations between arrays of different shapes. In this method, NumPy automatically broadcasts the smaller array over the larger array.

```
In [9]:  import numpy as np

In [10]: #Create two arrays of the same shape
         array_a = np.array([2, 3, 5, 8])
         array_b = np.array([.3, .3, .3, .3])

In [11]: #Multiply arrays
         array_a * array_b

Out[11]: array([ 0.6,  0.9,  1.5,  2.4])

In [12]: #Create a variable with a scalar value
         scalar_c = .3

In [13]: #Multiply 1D array with a scalar value
         array_a * scalar_c

Out[13]: array([ 0.6,  0.9,  1.5,  2.4])
```



If the shape doesn't match with array_a, numpy doesn't have to create copies of scalar value. Instead, broadcast scalar value over the entire array to find the product.

Broadcasting: Constraints

Though broadcasting can help carry out mathematical operations between different -shaped arrays, they are subject to certain constraints as listed below:

```
In [9]: import numpy as np

In [10]: #Create two arrays of the same shape
         array_a = np.array([2, 3, 5, 8])
         array_b = np.array([.3, .3, .3, .3])

In [11]: #Multiply arrays
         array_a * array_b

Out[11]: array([ 0.6,  0.9,  1.5,  2.4])

In [14]: #Create array of a different shape
         array_d = np.array([4, 3])

In [15]: array_a * array_d

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-43adcf6f7a54> in <module>()
----> 1 array_a * array_d

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

- When NumPy operates on two arrays, it compares their shapes element -wise. It finds these shapes compatible only if:
  - Their dimensions are the same or
  - One of them has a dimension of size 1.
- If these conditions are not met, a **ValueError** is thrown, indicating that the arrays have incompatible shapes.

Broadcasting: Example

Let’s look at an example to see how broadcasting works to calculate the number of working hours of a worker per day in a certain week.

```
In [246]: np_week_one =np.array([105, 135, 195, 120, 165])
          np_week_two =np.array([123, 156, 230, 200, 147])
```

Week one earnings

Week two earnings

```
In [247]: total_earning = np_week_one+np_week_two
```

Element -wise operation

```
In [248]: total_earning

Out[248]: array([228, 291, 425, 320, 312])
```

Total earning for 2 weeks

```
In [249]: np_week_one_hrs = np_week_one / 15
```

Calculate week one hours

Hourly wage

```
In [250]: np_week_one_hrs

Out[250]: array([ 7,  9, 13,  8, 11])
```

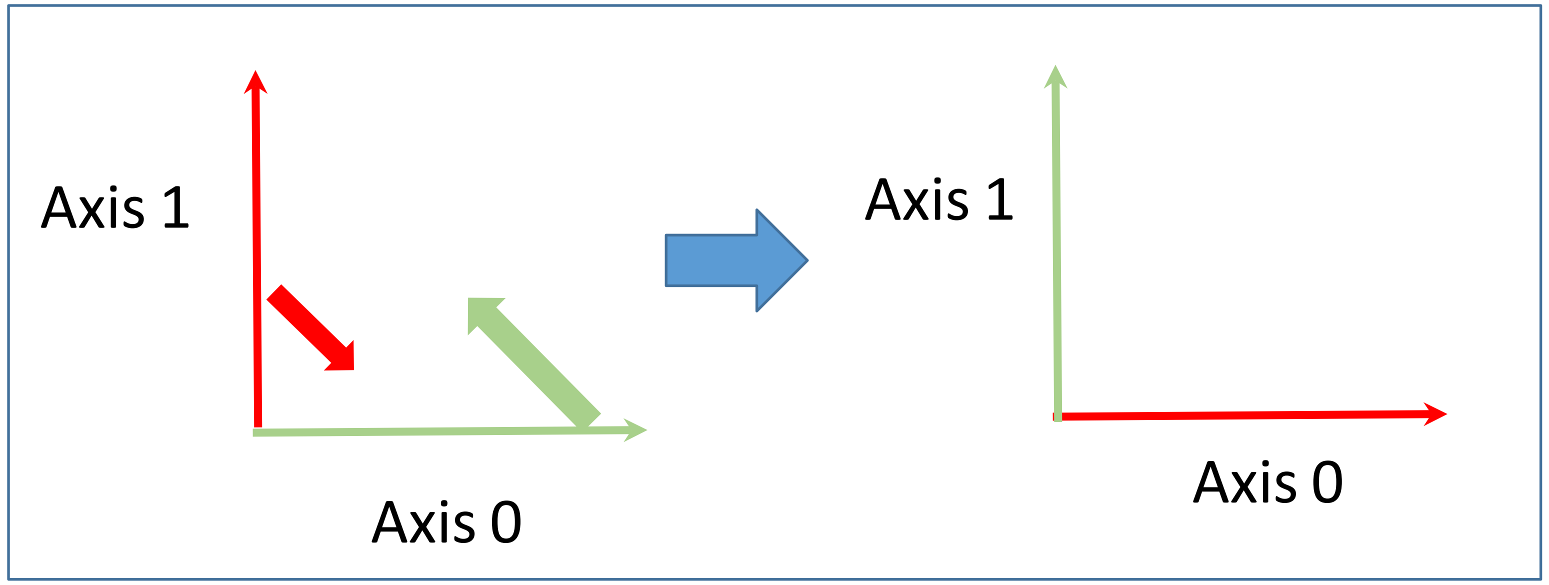
Number of working hours
per day in week one

Linear Algebra: Transpose

NumPy can carry out linear algebraic functions as well. The **transpose()** function can help you interchange rows as columns, and vice -versa.

transpose()

83	71	57	63
54	68	81	45




```
In [397]: test_scores =np.array([[83,71,57,63],[54,68,81,45]])

In [398]: test_scores.transpose()

Out[398]: array([[83, 54],
                 [71, 68],
                 [57, 81],
                 [63, 45]])
```

Linear Algebra: Inverse and Trace Functions

Using NumPy, you can also find the inverse of an array and add its diagonal data elements.

np.linalg.inv()

```
In [411]: inverse_array =np.array([[10,20],[15,25]])

In [412]: np.linalg.inv(inverse_array)
Out[412]: array([[-0.5,  0.4],
                 [ 0.3, -0.2]])
```

Inverse of the given array

* Can be applied **only** on a square matrix

np.trace()

```
In [420]: trace_array =np.array([[10,20],[22,31]])

In [421]: np.trace(trace_array)
Out[421]: 41
```

Sum of diagonal elements **10** and **31**