Numpy vs Lists

In [2]:
```
#Let's define a list in python.
heights = [74, 75, 72, 72, 71]
```

In [3]:
```
# Print the heights.
heights
```

Out[3]:
```
[74, 75, 72, 72, 71]
```

In [6]:
```
# Try to multiple heights with a scalar.
heights * 2.54
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-e7573032a4ae> in <module>
      1 # Try to multiple height with a scalar.
----> 2 heights * 2.54

TypeError: can't multiply sequence by non-int of type 'float'
```

In [7]:
```
import numpy as np
```

In [9]:
```
# Deine a NumPy array
np_heights = np.array([74, 75, 72, 72, 71])
```

In [10]:
```
np_heights
```

Out[10]:
```
array([74, 75, 72, 72, 71])
```

In [11]:
```
# Print the type of a NumPy array.
type(np_heights)
```

Out[11]:
```
numpy.ndarray
```

In [12]:
```
# Multiple height (NumPy array) with a scalar.
np_heights * 2.54
```

Out[12]:
```
array([187.96, 190.5 , 182.88, 182.88, 180.34])
```

In [ ]:
```
```

NumPy comes with its own set of methods and operations

In [40]:
```
# Let's define two lists and perform '+' operation on that.
list_1 = [1,2,3]
list_2 = [4,5,6]
list_1 + list_2
```

Out[40]:
```
[1, 2, 3, 4, 5, 6]
```

In [41]:
```
# Let's define two NumPy array and perform '+' operation on that.
np1 = np.array([1,2,3])
np2 = np.array([4,5,6])
np1 + np2
```

Out[41]:
```
array([5, 7, 9])
```

Working with N-D Arrays

In [45]:
```
np_heights
```

Out[45]:
```
array([74, 75, 72, 72, 71])
```

In [46]:
```
type(np_heights)
```

Out[46]:
```
numpy.ndarray
```

In [ ]:

Case Study - Cricket Tournament

A panel wants to select players for an upcoming league match based on their fitness. Players from all significant cricket clubs have participated in a practice match, and their data is collected. Let us now explore NumPy features using the player's data.

Example - 1

Heights of the players is stored as a regular Python list: height_in. The height is expressed in inches. Can you make a numpy array out of it ? ¶

In [5]:
```
# Define list
heights = [74, 74, 72, 72, 73, 69, 69, 71, 76, 71, 73, 73, 74, 74, 69, 70,
```

In [6]:
```
import numpy as np

heights_in = np.array(heights)
```

In [7]:
```
heights_in
```

Out[7]:
```
array([74, 74, 72, ..., 75, 75, 73])
```

In [8]:
```
type(heights_in)
```

Out[8]:
```
numpy.ndarray
```

Example - 2

Count the number of pariticipants

In [9]:
```
len(heights)
```

Out[9]:
```
1015
```

In [10]:
```
heights.size
```

Out[10]:
```
1015
```

In [11]:
```
heights.shape
```

Out[11]:
```
(1015,)
```

Example - 3

Convert the heights from inches to meters

In [12]:
```
heights_m = heights * 0.0254

heights_m
```

Out[12]:
```
array([1.8796, 1.8796, 1.8288, ..., 1.905 , 1.905 , 1.8542])
```

Example - 4

A list of weights (in lbs) of the players is provided. Convert it to kg and calculate BMI

In [13]:
```
weights_lb = [180, 215, 210, 210, 188, 176, 209, 200, 231, 180, 188, 180, 1
```

In [14]:
```
# Converting weights in lbs to kg

weights_kg = np.array(weights_lb) * 0.453592

weights_kg
```

Out[14]:
```
array([81.64656, 97.52228, 95.25432, ..., 92.98636, 86.18248, 88.45044])
```

In [15]:
```
# Calculate the BMI: bmi

bmi = weights_kg / (heights_m ** 2)

bmi
```

Out[15]:
```
array([23.11037639, 27.60406069, 28.48080465, ..., 25.62295933,
       23.74810865, 25.72686361])
```

Sub-Setting Arrays

Fetch the first element from the bmi array

In [16]:
```
bmi[0]
```

Out[16]:
```
23.11037638875862
```

Fetch the last element from the bmi array

In [17]:
```
bmi[-1]
```

Out[17]:
```
25.72686361360713
```

Fetch the first 5 elements from the bmi array

In [18]:
```
bmi[0:5]
```

Out[18]:
```
array([23.11037639, 27.60406069, 28.48080465, 28.48080465, 24.80333518])
```

Fetch the last 5 elements from the bmi array

In [19]:
```
bmi[-5:]
```

Out[19]:
```
array([25.06720044, 23.11037639, 25.62295933, 23.74810865, 25.72686361])
```

Conditional Sub-Setting Arrays

Count the number of pariticipants who are underweight i.e. bmi < 21

In [20]:
```
bmi < 21
```

Out[20]:
```
array([False, False, False, ..., False, False, False])
```

In [21]:
```
bmi [ bmi<21]
```

Out[21]:
```
array([20.54255679, 20.54255679, 20.69282047, 20.69282047, 20.34343189,
       20.34343189, 20.69282047, 20.15883472, 19.4984471 , 20.69282047,
       20.9205219 ])
```

In [22]:
```
underweight_players = bmi [ bmi<21]
underweight_players
```

Out[22]:
```
array([20.54255679, 20.54255679, 20.69282047, 20.69282047, 20.34343189,
       20.34343189, 20.69282047, 20.15883472, 19.4984471 , 20.69282047,
       20.9205219 ])
```

In [23]:
```
underweight_players.size
```

Out[23]:
```
11
```

NumPy Functions

Find the largest BMI value

In [24]:
```
max(bmi)
```

Out[24]:
```
35.26194861031698
```

In [25]:
```
bmi.max()
```

Out[25]:
```
35.26194861031698
```

Find lowest BMI value

In [26]:
```
bmi.min()
```

Out[26]: 19.498447103560874

Find average BMI value

In [27]:
```
bmi.mean()
```

Out[27]: 26.05684565448554

In [ ]:
```

```

Case Study - Cricket Tournament

Example - 1

Players list contain the height(inches) and weight(lbs) data for all the players

In [1]:
```
# list of height and weight of the players.
players = [(74, 180), (74, 215), (72, 210), (72, 210), (73, 188), (69, 176)
```

In [2]:
```
len(players)
```

Out[2]:
```
1015
```

In [2]:
```
players[1][1]
```

Out[2]:
```
215
```

In [3]:
```
import numpy as np

np_players = np.array(players)
```

In [4]:
```
np_players
```

Out[4]:
```
array([[ 74, 180],
       [ 74, 215],
       [ 72, 210],
       ...,
       [ 75, 205],
       [ 75, 190],
       [ 73, 195]])
```

In [5]:
```
type(np_players)
```

Out[5]:
```
numpy.ndarray
```

In [ ]:
```

```

Example - 2 (Numpy Attributes)

Print the structure of the 2-D Array

In [6]:
```
np_players.shape
```

Out[6]:
```
(1015, 2)
```

Print the dimensions of the array

In [7]:
```
np_players.ndim
```

Out[7]:
```
2
```

Print the data type of elements in the array

In [8]:
```
np_players.dtype
```

Out[8]:
```
dtype('int32')
```

Print the size of a single item of the array

In [9]:
```
np_players.itemsize
```

Out[9]:
```
4
```

Example - 3

Convert the heights to meters and weights to kg

In [10]:
```
players_converted = np_players * [0.0254, 0.453592]
```

In [14]:
```
players_converted
```

Out[14]:
```
array([[ 1.8796 , 81.64656],
       [ 1.8796 , 97.52228],
       [ 1.8288 , 95.25432],
       ...,
       [ 1.905  , 92.98636],
       [ 1.905  , 86.18248],
       [ 1.8542 , 88.45044]])
```

Sub-Setting 2-D Arrays

Fetch the first row from the array

In [15]:
```
players_converted[0]
```

Out[15]:
```
array([ 1.8796 , 81.64656])
```

Fetch the first row 2nd element from the array

In [16]:
```
players_converted[0][1]
```

Out[16]:
```
81.64656
```

Fetch the first column from the array

In [17]:
```
players_converted[:, 0]
```
Out[17]:
```
array([1.8796, 1.8796, 1.8288, ..., 1.905 , 1.905 , 1.8542])
```

Fetch the height (1st column) of 125th player from the array

In [18]:
```
players_converted[124][0]
```
Out[18]:
```
1.9811999999999999
```

In [19]:
```
players_converted[124,0]
```
Out[19]:
```
1.9811999999999999
```

Conditional Sub-Setting Arrays

Fetch height and weight of players with height above 1.8m

In [21]:
```
tall_players = players_converted[players_converted[:,0] > 1.8]
```

In [22]:
```
players_converted.shape
```
Out[22]:
```
(1015, 2)
```

In [23]:
```
tall_players.shape
```
Out[23]:
```
(936, 2)
```

Skills Array - holds the player key skills.

In [24]:
```
skills = np.array(['Keeper', 'Batsman', 'Bowler', 'Keeper-Batsman', 'Batsma
skills
```
Out[24]:
```
array(['Keeper', 'Batsman', 'Bowler', ..., 'Batsman', 'Bowler',
       'Keeper-Batsman'], dtype='<U14')
```

Fetch Heights of the Batsmen

In [25]:
```
batsmen = players_converted[skills == 'Batsman']
```

In [27]:
```
batsmen.shape
```
Out[27]:
```
(323, 2)
```

In [28]:
```
batsmen[:, 0]
```

Out[28]:
```
array([1.8796, 1.8542, 1.7526, 1.8034, 1.9304, 1.8542, 1.8542, 1.778 ,
       2.0066, 1.8288, 1.8034, 1.905 , 1.9558, 1.8542, 1.905 , 1.8796,
       1.8034, 1.8542, 1.8796, 1.9304, 1.905 , 1.9304, 1.8288, 1.905 ,
       1.8542, 1.778 , 1.778 , 1.8034, 1.8288, 1.905 , 1.9812, 1.8034,
       1.8542, 1.8542, 1.9304, 1.8796, 1.8542, 1.8288, 1.8542, 1.8288,
       1.8542, 1.8288, 1.905 , 1.905 , 1.8288, 1.8288, 1.9558, 1.9558,
       1.905 , 1.9304, 2.032 , 1.905 , 1.8542, 1.8796, 1.905 , 1.8034,
       1.9304, 1.8796, 1.8542, 1.8542, 1.8034, 1.8542, 1.8542, 1.8288,
       1.905 , 1.778 , 1.8034, 1.8288, 1.905 , 1.8542, 1.9304, 1.905 ,
       1.9304, 1.8288, 1.8542, 1.905 , 1.8796, 1.8034, 1.9558, 1.9812,
       1.905 , 1.905 , 1.9304, 1.8288, 1.8288, 1.8542, 1.8796, 1.8796,
       1.905 , 1.8542, 1.8796, 1.9558, 1.9812, 1.9812, 1.8796, 1.9812,
       1.8796, 1.8288, 1.9304, 1.8542, 1.8542, 1.9558, 1.9558, 1.8034,
       1.9812, 1.778 , 1.8796, 1.8288, 1.8542, 1.905 , 1.8796, 1.8542,
       1.8796, 1.8542, 1.9812, 1.9304, 1.8542, 1.905 , 1.9812, 1.9558,
       1.8288, 1.7526, 1.8796, 1.778 , 1.8796, 1.9304, 1.905 , 1.8542,
       1.8542, 1.8542, 1.8796, 1.8796, 1.778 , 1.8796, 1.905 , 1.8288,
       1.9558, 1.8542, 1.9304, 1.8542, 1.905 , 1.8796, 1.8542, 1.8034,
       1.9304, 1.905 , 1.8542, 1.8542, 1.9304, 1.8542, 1.905 , 1.905 ,
       1.9558, 1.8796, 1.8034, 1.8796, 1.8796, 1.905 , 1.8288, 1.8542,
       1.9304, 1.9558, 1.8542, 1.778 , 1.8542, 1.8796, 1.9558, 1.905 ,
       1.8542, 1.9558, 1.9558, 1.8796, 1.8796, 1.905 , 1.8034, 1.778 ,
       2.0066, 1.8796, 1.8288, 2.0828, 1.8796, 1.8796, 1.8288, 1.9304,
       1.8542, 1.8288, 1.8288, 1.778 , 1.8034, 1.905 , 1.9304, 1.9304,
       1.9812, 1.905 , 1.9304, 1.8288, 1.8542, 1.778 , 1.8796, 1.8542,
       1.8542, 1.905 , 1.778 , 2.0066, 1.905 , 1.905 , 1.8542, 1.778 ,
       1.8034, 1.905 , 1.8288, 1.8288, 1.9304, 1.905 , 1.7526, 1.8288,
       1.9304, 1.8034, 1.905 , 1.9558, 1.778 , 1.8288, 1.8034, 1.8796,
       1.9304, 1.8288, 1.8796, 1.8288, 1.8034, 1.778 , 1.8288, 1.8796,
       1.8796, 1.905 , 1.8796, 1.8034, 1.8034, 1.9304, 1.8034, 1.8796,
       1.8288, 1.9304, 1.9812, 1.8288, 1.9304, 1.778 , 1.7272, 1.8034,
       1.9558, 1.7526, 1.905 , 1.905 , 1.9304, 1.8288, 1.9558, 1.778 ,
       2.0066, 1.8796, 1.7272, 1.905 , 1.8288, 1.8288, 1.8542, 1.8796,
       1.8288, 1.905 , 1.8288, 1.8542, 1.9304, 1.8796, 1.905 , 1.9304,
       1.8796, 1.8288, 1.8542, 1.8288, 1.8542, 1.8288, 1.8542, 1.778 ,
       1.8288, 1.905 , 1.8542, 1.9304, 1.9558, 1.9558, 1.905 , 1.905 ,
       1.9304, 1.8288, 1.8542, 1.8796, 1.8288, 1.8796, 1.8796, 1.905 ,
       1.8034, 1.9304, 1.8542, 1.7272, 1.8288, 1.7526, 1.8542, 1.905 ,
       1.8796, 1.8796, 1.8796, 1.8542, 1.8796, 1.905 , 1.8796, 1.8542,
       1.9304, 1.9812, 1.8542, 1.905 , 1.7018, 1.778 , 1.778 , 2.0066,
       1.9304, 1.8288, 1.905 ])
```

In [ ]:

Creating NumPy Arrays

The following ways are commonly used when you know the size of the array beforehand:

- `np.ones()` : Create array of 1s
- `np.zeros()` : Create array of 0s
- `np.random.random()` : Create array of random numbers
- `np.arange()` : Create array with increments of a fixed step size
- `np.linspace()` : Create array of fixed length

In [1]:
```
import numpy as np
```

Tip: Use help to see the syntax when required

In [2]:
```
help(np.ones)
```

```
Help on function ones in module numpy:

ones(shape, dtype=None, order='C')
    Return a new array of given shape and type, filled with ones.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    dtype : data-type, optional
        The desired data-type for the array, e.g., `numpy.int8`.  Default
is
        `numpy.float64`.
    order : {'C', 'F'}, optional, default: C
        Whether to store multi-dimensional data in row-major
        (C-style) or column-major (Fortran-style) order in
        memory.

    Returns
    -------
    out : ndarray
        Array of ones with the given shape, dtype, and order.

    See Also
    --------
    ones_like : Return an array of ones with shape and type of input.
    empty : Return a new uninitialized array.
    zeros : Return a new array setting values to zero.
    full : Return a new array of given shape filled with value.


    Examples
    --------
    >>> np.ones(5)
    array([1., 1., 1., 1., 1.])

    >>> np.ones((5,), dtype=int)
    array([1, 1, 1, 1, 1])

    >>> np.ones((2, 1))
    array([[1.],
           [1.]])

    >>> s = (2,2)
    >>> np.ones(s)
    array([[1.,  1.],
           [1.,  1.]])
```

Creating a 1 D array of ones

In [3]:
```
arr = np.ones(5)
arr
```

Out[3]:
```
array([1., 1., 1., 1., 1.])
```

Notice that, by default, numpy creates data type = float64

In [4]:
```
arr.dtype
```

Out[4]:
```
dtype('float64')
```

Can provide dtype explicitly using dtype

In [5]:
```
arr = np.ones(5, dtype=int)
arr
```

Out[5]:
```
array([1, 1, 1, 1, 1])
```

In [6]:
```
arr.dtype
```

Out[6]:
```
dtype('int64')
```

Creating a 5 x 3 array of ones

In [7]:
```
np.ones((5,3))
```

Out[7]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Creating array of zeros

In [8]:
```
np.zeros(5)
```

Out[8]:
```
array([0., 0., 0., 0., 0.])
```

In [9]:
```
# convert the type into integer.
np.zeros(5, dtype=int)
```

Out[9]:
```
array([0, 0, 0, 0, 0])
```

In [12]:
```
# Create a list of integers range between 1 to 5.
list(range(1,5))
```

Out[12]:
```
[1, 2, 3, 4]
```

In [13]:
```
np.arange(3)
```

Out[13]:
```
array([0, 1, 2])
```

In [14]:
```
np.arange(3.0)
```

Out[14]:
```
array([0., 1., 2.])
```

Notice that 3 is included, 35 is not, as in standard python lists

From 3 to 35 with a step of 2

In [20]:
```
np.arange(3,35,2)
```

Out[20]:
```
array([ 3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33])
```

Array of random numbers

In [21]:
```
np.random.randint(2, size=10)
```

Out[21]:
```
array([0, 1, 0, 1, 1, 1, 0, 0, 0, 0])
```

In [24]:
```
np.random.randint(3,5, size=10)
```

Out[24]:
```
array([3, 3, 3, 3, 4, 3, 3, 3, 3, 4])
```

2D Array of random numbers

In [25]:
```
np.random.random([3,4])
```

Out[25]:
```
array([[0.37947795, 0.50446351, 0.76204337, 0.23268129],
       [0.49530063, 0.37298231, 0.17830691, 0.9400508 ],
       [0.18746889, 0.99395211, 0.03729134, 0.16021317]])
```

Sometimes, you know the length of the array, not the step size

Array of length 20 between 1 and 10

In [27]:
```
np.linspace(1,10,20)
```

Out[27]:
```
array([ 1.        ,  1.47368421,  1.94736842,  2.42105263,  2.89473684,
        3.36842105,  3.84210526,  4.31578947,  4.78947368,  5.26315789,
        5.73684211,  6.21052632,  6.68421053,  7.15789474,  7.63157895,
        8.10526316,  8.57894737,  9.05263158,  9.52631579, 10.        ])
```

Exercises

Apart from the methods mentioned above, there are a few more NumPy functions that you can use to create special NumPy arrays:

- `np.full()` : Create a constant array of any number ‘n’
- `np.tile()` : Create a new array by repeating an existing array for a particular number of times
- `np.eye()` : Create an identity matrix of any dimension

- `np.random.randint()` : Create a random array of integers within a particular range

In [ ]:

Operations on NumPy Arrays

The learning objectives of this section are:

- Manipulate arrays
    - Reshape arrays
    - Stack arrays
- Perform operations on arrays
    - Perform basic mathematical operations
    - Apply built-in functions
    - Apply your own functions
    - Apply basic linear algebra operations

In [12]:
```
import numpy as np
```

Example - 1 (Arithmatric Operations)

In [13]:
```
array1 = np.array([10,20,30,40,50])
array2 = np.arange(5)
```

In [14]:
```
array1
```

Out[14]:
```
array([10, 20, 30, 40, 50])
```

In [16]:
```
array2
```

Out[16]:
```
array([0, 1, 2, 3, 4])
```

In [17]:
```
# Add array1 and array2.
array3 = array1 + array2
```

In [18]:
```
array3
```

Out[18]:
```
array([10, 21, 32, 43, 54])
```

Example - 2

In [20]:
```
array4 = np.array([1,2,3,4])
```

In [21]:
```
array4 + array1
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-2811f702eb3f> in <module>
----> 1 array4 + array1

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

In [22]:
```
print (array1.shape)
```

```
(5,)
```

In [23]:
```
print (array4.shape)
```

```
(4,)
```

Example - 3

In [24]:
```
array = np.linspace(1, 10, 5)
array
```

Out[24]:
```
array([ 1.  ,  3.25,  5.5 ,  7.75, 10.  ])
```

In [25]:
```
array*2
```

Out[25]:
```
array([ 2. ,  6.5, 11. , 15.5, 20. ])
```

In [26]:
```
array**2
```

Out[26]:
```
array([  1.    ,  10.5625,  30.25  ,  60.0625, 100.    ])
```

Stacking Arrays

`np.hstack() and n.vstack()`

Stacking is done using the `np.hstack()` and `np.vstack()` methods. For horizontal stacking, the number of rows should be the same, while for vertical stacking, the number of columns should be the same.

In [27]:
```
# Note that np.hstack(a, b) throws an error - you need to pass the arrays a
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])

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

Out[27]:
```
array([1, 2, 3, 2, 3, 4])
```

In [28]:
```
np.vstack((a,b))
```

Out[28]:
```
array([[1, 2, 3],
       [2, 3, 4]])
```

In [29]:
```
np.arange(12)
```

Out[29]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [30]:
```
np.arange(12).reshape(3,4)
```

Out[30]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [31]:
```
array1 = np.arange(12).reshape(3,4) #3x4
array2 = np.arange(20).reshape(5,4) #5x4
```

In [33]:
```
print (array1, '\n', array2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]
```

In [34]:
```
np.vstack((array1,array2))
```

Out[34]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

Example - 4 (Numpy Built-in functions)

In [35]:
```
array1
```

Out[35]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [36]:
```
np.power(array1, 3)
```

Out[36]:
```
array([[   0,    1,    8,   27],
       [  64,  125,  216,  343],
       [ 512,  729, 1000, 1331]])
```

In [38]:
```
np.arange(9).reshape(3,3)
```

Out[38]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [39]:
```
x = np.array([-2,-1, 0, 1,2])
x
```

Out[39]:
```
array([-2, -1,  0,  1,  2])
```

In [40]:
```
abs(x)
```

Out[40]:
```
array([2, 1, 0, 1, 2])
```

In [41]:
```
np.absolute(x)
```

Out[41]:
```
array([2, 1, 0, 1, 2])
```

Example - 5 (Trignometric functions)

In [42]:
```
np.pi
```

Out[42]:
```
3.141592653589793
```

In [43]:
```
theta = np.linspace(0, np.pi, 5)
```

In [44]:
```
theta
```

Out[44]:
```
array([0.        , 0.78539816, 1.57079633, 2.35619449, 3.14159265])
```

In [45]:
```
np.sin(theta)
```

Out[45]:
```
array([0.00000000e+00, 7.07106781e-01, 1.00000000e+00, 7.07106781e-01,
       1.22464680e-16])
```

In [46]:
```
np.cos(theta)
```

Out[46]:
```
array([ 1.00000000e+00,  7.07106781e-01,  6.12323400e-17, -7.07106781e-01,
       -1.00000000e+00])
```

In [47]:
```
np.tan(theta)
```

Out[47]:
```
array([ 0.00000000e+00,  1.00000000e+00,  1.63312394e+16, -1.00000000e+00,
       -1.22464680e-16])
```

Example - 6 (Exponential and logarithmic functions)

In [48]:
```
x = [1, 2, 3, 10]
x = np.array(x)
```

In [49]:
```
np.exp(x) # e=2.718...
```

Out[49]:
```
array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 2.20264658e+04])
```

In [50]:
```
# 2^1, 2^2, 2^3, 2^10
np.exp2(x)
```

Out[50]:
```
array([   2.,    4.,    8., 1024.])
```

In [51]:
```
np.power(x,3)
```

Out[51]:
```
array([   1,    8,   27, 1000])
```

In [52]:
```
np.log(x)
```

Out[52]:
```
array([0.        , 0.69314718, 1.09861229, 2.30258509])
```

In [53]:
```
np.log2(x)
```

Out[53]:
```
array([0.        , 1.        , 1.5849625 , 3.32192809])
```

In [54]:
```
np.log10(x)
```

Out[54]:
```
array([0.        , 0.30103   , 0.47712125, 1.        ])
```

In [ ]:
```
np.log
```

Example - 7

In [57]:
```
x = np.arange(5)
x
```

Out[57]:
```
array([0, 1, 2, 3, 4])
```

In [59]:
```
y = x * 10
y
```

Out[59]:
```
array([ 0, 10, 20, 30, 40])
```

In [58]:
```
y = np.empty(5)
y
```

Out[58]:
```
array([ 1.00000000e+00,  7.07106781e-01,  6.12323400e-17, -7.07106781e-01,
       -1.00000000e+00])
```

In [61]:
```
np.multiply(x, 12, out=y)
```

Out[61]:
```
array([ 0, 12, 24, 36, 48])
```

In [62]:
```
y
```

Out[62]:
```
array([ 0, 12, 24, 36, 48])
```

In [63]:
```
y = np.zeros(10)
y
```

Out[63]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [65]:
```
np.power(2, x, out=y[::2])
```

Out[65]:
```
array([ 1.,  2.,  4.,  8., 16.])
```

In [66]:
```
y
```

Out[66]:
```
array([ 1.,  0.,  2.,  0.,  4.,  0.,  8.,  0., 16.,  0.])
```

Example - 8 (Aggregates)

In [67]:
```
x = np.arange(1,6)
x
```

Out[67]:
```
array([1, 2, 3, 4, 5])
```

In [69]:
```
sum(x)
```

Out[69]:
```
15
```

In [68]:
```
np.add.reduce(x)
```

Out[68]:
```
15
```

In [70]:
```
np.add.accumulate(x)
```

Out[70]:
```
array([ 1,  3,  6, 10, 15])
```

In [72]:
```
np.multiply.accumulate(x)
```

Out[72]:
```
array([  1,   2,   6,  24, 120])
```

In [ ]:
```

```

Apply Basic Linear Algebra Operations

NumPy provides the `np.linalg` package to apply common linear algebra operations, such as:

- `np.linalg.inv` : Inverse of a matrix
- `np.linalg.det` : Determinant of a matrix
- `np.linalg.eig` : Eigenvalues and eigenvectors of a matrix

Also, you can multiple matrices using `np.dot(a, b)` .

In [73]:
```
# np.linalg documentation
help(np.linalg)
```

```
Help on package numpy.linalg in numpy:

NAME
    numpy.linalg

DESCRIPTION
    ``numpy.linalg``
    ================

    The NumPy linear algebra functions rely on BLAS and LAPACK to provi
de efficient
    low level implementations of standard linear algebra algorithms. Th
ose
    libraries may be provided by NumPy itself using C versions of a sub
set of their
    reference implementations but, when possible, highly optimized libr
aries that
    take advantage of specialized processor functionality are preferre
d. Examples
```

In [74]:
```
A = np.array([[6, 1, 1],
              [4, -2, 5],
              [2, 8, 7]])
```

In [75]:
```
A
```

Out[75]:
```
array([[ 6,  1,  1],
       [ 4, -2,  5],
       [ 2,  8,  7]])
```

Rank of a matrix

In [76]:
```
np.linalg.matrix_rank(A)
```

Out[76]:
```
3
```

Trace of matrix A

In [77]:
```
np.trace(A)
```

Out[77]:
```
11
```

Determinant of a matrix

In [78]:
```
np.linalg.det(A)
```

Out[78]:
```
-306.0
```

Inverse of matrix A

In [87]:
```
A
```

Out[87]:
```
array([[ 6,  1,  1],
       [ 4, -2,  5],
       [ 2,  8,  7]])
```

In [79]:
```
np.linalg.inv(A)
```

Out[79]:
```
array([[ 0.17647059, -0.00326797, -0.02287582],
       [ 0.05882353, -0.13071895,  0.08496732],
       [-0.11764706,  0.1503268 ,  0.05228758]])
```

In [84]:
```
B = np.linalg.inv(A)
```

In [85]:
```
np.matmul(A,B) #actual matrix multiplication
```

Out[85]:
```
array([[ 1.00000000e+00,  0.00000000e+00,  2.77555756e-17],
       [-1.38777878e-17,  1.00000000e+00,  1.38777878e-17],
       [-4.16333634e-17,  1.38777878e-16,  1.00000000e+00]])
```

In [86]:
```
A * B
```

Out[86]:
```
array([[ 1.05882353, -0.00326797, -0.02287582],
       [ 0.23529412,  0.26143791,  0.4248366 ],
       [-0.23529412,  1.20261438,  0.36601307]])
```

Matrix A raised to power 3

In [88]:
```
np.linalg.matrix_power(A,3) # matrix multiplication A A A
```

Out[88]:
```
array([[336, 162, 228],
       [406, 162, 469],
       [698, 702, 905]])
```

In [ ]: