P02: INTRODUCTION TO NUMPY

Reference:

The numpy reference and user guide are available in this link (https://docs.scipy.org/doc/)

This tutorial is adapted from SciPy Quickstart Tutorial (https://docs.scipy.org/doc/numpy-dev/user/quickstart.html#fancy-indexing-and-index-tricks). If you need more exercises, you can work on 100 numpy exercises (http://www.labri.fr/perso/nrougier/teaching/numpy.100/)

INTRODUCTION

What is numpy?

NumPy is the fundamental package required for high performance scientific computing and data analysis. It issued by practically all other scientific tools, such as Panda and Scikit. The following code shows how to import the numpy library.

In [1]:

```
import numpy as np          # import numpy library
```

Why numpy?

- Numpy is *much faster* for scientific computation compared to conventional Python methods.
- Provides standard mathematical functions for fast operations on entire arrays of data without having to write loops (easier to write)
- Provides tools for reading / writing array data to disk
- Linear algebra, random number generation capabilities

SECTION 1: ARRAY CREATION

Creating an array from a list or tuple

`np.array` :

Creates a *numpy array* using a normal python *list* or *tuple*.

Creating 1-D array

In [2]:

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

Out[2]:

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

In [3]:

```
list1 = [2,4,6,8]
a = np.array(list1)
a
```

Out[3]:

```
array([2, 4, 6, 8])
```

Displaying the information about the array

In [4]:

```
print('Shape of a =', a.shape)                    # number items in each dimension
print('Number of dimensions of a =', a.ndim)      # number of dimensions
print('Number of items of vector a =', len(a))        # number of items in 1st dimension. S
                                                  #     this corresponds to number of items
```

```
Shape of a = (4,)
Number of dimensions of a = 1
Number of items of vector a = 4
```

Displaying the type of the array or its elements

In [5]:

```
print('Type of a =', type(a))                    # a is a list
print('Type of items in a =', a.dtype)           # the items stored in a is an integer
```

```
Type of a = <class 'numpy.ndarray'>
Type of items in a = int32
```

Creating 2-D array

In [6]:

```
a = np.array([[1, 2, 3, 4, 5],[6, 7, 8, 9, 10], [11, 12, 13, 14, 15]])
a
```

Out[6]:

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

In [7]:

```
print('Shape of a =', a.shape)                   # number of items in each dimension
print('Number of dimensions of a =', a.ndim)     # number of dimensions
print('Number of rows of matrix a =', len(a))    # number of items in 1st dimension. Since
                                                 #     this corresponds to number of rows
```

```
Shape of a = (3, 5)
Number of dimensions of a = 2
Number of rows of matrix a = 3
```

Arrays of different type

The type is automatically inferred by the values in the list

In [8]:

```
a = np.array([(1.5, 2, 3), (4, 5, 6)])
print(a)
print(a.dtype)
```

```
[[1.5 2.  3. ]
 [4.  5.  6. ]]
float64
```

The type of the array can also be explicitly specified at creation time:

In [9]:

```
a = np.array( [ [1,1], [0,1] ], dtype=bool)
a
```

Out[9]:

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

Creating arrays initialized with zeros or ones

np.ones and **np.zeros** :

Creates array initialized to ones and zeros, respectively

In [10]:

```
all_zeros = np.zeros((3, 5))
all_zeros
```

Out[10]:

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

In [11]:

```
all_ones = np.ones(3)
all_ones
```

Out[11]:

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

Generating number sequences

Generating evenly spaced values within an interval

** np.arange **

`np.arange` returns evenly spaced values within a given interval. Values are generated within the half-open interval `[start, stop)` . This means the interval includes `start` but excluding `stop` . For integer arguments, the function is equivalent to the Python built-in `range` function, but returns an `ndarray` (numpy array type)rather than a `list` .

In [12]:

```
a = np.arange( 10, 30, 5 )                  # the parameters are start, end, step. Note tha
print(a)
```

```
[10 15 20 25]
```

`np.arange` can also work with float aargument as well

In [13]:

```
b =  np.arange( 0, 2, 0.3 )                 # it accepts float arguments as well
print(b)
```

```
[0.  0.3 0.6 0.9 1.2 1.5 1.8]
```

`np.arange` also accepts one parameter (`end`). By default, `start` = 0 and `step` = 1

In [16]:

```
c = np.arange(10)                           # starts = 0 and step = 1
c
```

Out[16]:

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

Generating a certain number of items within an interval

`np.linspace`

Returns a particular number of numbers that are evenly spaced over a specified interval.

In [18]:

```
from numpy import pi
a = np.linspace( 0, 2, 9 )                  # generates a total of 9 numbers ranging from fr
print('a = ', a)
```

```
a =  [0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
```

In [22]:

```
a = np.linspace(0,10,1)
print(a)
```

```
[0.]
```

In [23]:

```
x = np.linspace( 0, 2*pi, 10)        # useful to evaluate function at lots of points
print('x = ', x)

f = np.sin(x)
print('f = ', f)
```

```
x =  [0.         0.6981317  1.3962634  2.0943951  2.7925268  3.4906585
 4.1887902  4.88692191 5.58505361 6.28318531]
f =  [ 0.00000000e+00  6.42787610e-01  9.84807753e-01  8.66025404e-01
  3.42020143e-01 -3.42020143e-01 -8.66025404e-01 -9.84807753e-01
 -6.42787610e-01 -2.44929360e-16]
```

Random number generator

`np.random.rand()` :

Creates an array of the given shape and populate it with random samples from a *uniform* distribution over `[0, 1)` (i.e., inclusive of 0 but exclusive of 1)

In [24]:

```
np.random.rand()              # generate a random scalar number uniform in [0,1)
```

Out[24]:

```
0.8605374461200643
```

In [25]:

```
np.random.rand(4)             # generates an array of size 4
```

Out[25]:

```
array([0.73480198, 0.60643378, 0.89824132, 0.09275302])
```

In [26]:

```
np.random.rand(3,2)           # generates a matrix of size 3x2
```

Out[26]:

```
array([[0.14310056, 0.03482878],
       [0.22645686, 0.89446827],
       [0.29574355, 0.63120269]])
```

`np.random.randint(low, high=None, size=None)` :

Returns an array of size `size` . The array is populated with random integers from the *discrete uniform* distribution in the interval `[low, high)` (excluding `high`). If `high` is `None` (the default), then results are from `[0, low)` .

In [28]:

```
np.random.randint(0, 20+1)    # generate a random integer scalar between 0 and 20
```

Out[28]:

```
15
```

In [29]:

```
np.random.randint(0,20)
```

Out[29]:

```
16
```

In [30]:

```
np.random.randint(0, 21, (3,2))# generates a matrix of size (3,2)
```

Out[30]:

```
array([[11,  3],
       [18,  6],
       [ 0,  2]])
```

np.random.randn :

Returns a sample (or samples) from the *standard normal* distribution.

In [31]:

```
np.random.randn()   # univariate 'normal' (Gaussian distribution, with a mean = 0, variance
```

Out[31]:

```
0.7128651022753891
```

The following code show how to create a 2x2 matrix whose numbers are sampled from a normal distribution

In [32]:

```
np.random.randn(2, 2)
```

Out[32]:

```
array([[ 1.01759057, -0.26438428],
       [-1.51491713,  0.46021416]])
```

Seeding the random generator

np.random.seed()

By default, random number generators uses the system time for random seeding. This allows you to generate different numbers each time you run your simultation.

But, sometimes you may want *repeatable* result. To produce repeatable results, seed your generator using a constant number. Try running the following two cells. Notice that when we apply the same seed, the random number generator will generate the same number sequences.

In [33]:

```
np.random.seed(4)
print(np.random.rand(4))
print(np.random.rand(4))
```

```
[0.96702984 0.54723225 0.97268436 0.71481599]
[0.69772882 0.2160895  0.97627445 0.00623026]
```

In [34]:

```
np.random.seed(4)
print(np.random.rand(4))
print(np.random.rand(4))
```

```
[0.96702984 0.54723225 0.97268436 0.71481599]
[0.69772882 0.2160895  0.97627445 0.00623026]
```

Exercise 1

Q1. Create a vector containing the first 10 odd integers using `np.arange` .

Answer:

```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

In [41]:

```
a = np.arange(1,20,2)
a
```

Out[41]:

```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
```

Q2. Create a uniform subdivision of the interval -1.3 to 2.5 with 64 subdivisions using `np.linspace` .

Answer:

```
array([-1.3       , -1.23968254, -1.17936508, -1.11904762, -1.05873016,
       -0.9984127 , -0.93809524, -0.87777778, -0.81746032, -0.75714286,
       -0.6968254 , -0.63650794, -0.57619048, -0.51587302, -0.45555556,
       -0.3952381 , -0.33492063, -0.27460317, -0.21428571, -0.15396825,
       -0.09365079, -0.03333333,  0.02698413,  0.08730159,  0.14761905,
        0.20793651,  0.26825397,  0.32857143,  0.38888889,  0.44920635,
        0.50952381,  0.56984127,  0.63015873,  0.69047619,  0.75079365,
        0.81111111,  0.87142857,  0.93174603,  0.99206349,  1.05238095,
        1.11269841,  1.17301587,  1.23333333,  1.29365079,  1.35396825,
        1.41428571,  1.47460317,  1.53492063,  1.5952381 ,  1.65555556,
        1.71587302,  1.77619048,  1.83650794,  1.8968254 ,  1.95714286,
        2.01746032,  2.07777778,  2.13809524,  2.1984127 ,  2.25873016,
        2.31904762,  2.37936508,  2.43968254,  2.5       ])
```

In [42]:

```
 a = np.linspace(-1.3,2.5,64)
a
```

Out[42]:

```
array([-1.3       , -1.23968254, -1.17936508, -1.11904762, -1.05873016,
       -0.9984127 , -0.93809524, -0.87777778, -0.81746032, -0.75714286,
       -0.6968254 , -0.63650794, -0.57619048, -0.51587302, -0.45555556,
       -0.3952381 , -0.33492063, -0.27460317, -0.21428571, -0.15396825,
       -0.09365079, -0.03333333,  0.02698413,  0.08730159,  0.14761905,
        0.20793651,  0.26825397,  0.32857143,  0.38888889,  0.44920635,
        0.50952381,  0.56984127,  0.63015873,  0.69047619,  0.75079365,
        0.81111111,  0.87142857,  0.93174603,  0.99206349,  1.05238095,
        1.11269841,  1.17301587,  1.23333333,  1.29365079,  1.35396825,
        1.41428571,  1.47460317,  1.53492063,  1.5952381 ,  1.65555556,
        1.71587302,  1.77619048,  1.83650794,  1.8968254 ,  1.95714286,
        2.01746032,  2.07777778,  2.13809524,  2.1984127 ,  2.25873016,
        2.31904762,  2.37936508,  2.43968254,  2.5       ])
```

Q3. Create a matrix of size (4, 5). Each item is a random integer between 0 and 10. Use `np.random.randint` .

Sample solution (Numbers in matrix must be integers between 0 and 10):

```
array([[ 7,  9,  8,  4,  2],
       [ 6, 10,  4,  3,  0],
       [ 7,  5,  5,  9,  6],
       [ 6,  8,  2,  5,  8]])
```

In [99]:

```
np.random.seed(3)
a = np.random.randint(0,11,(4,5))
a
```

Out[99]:

```
array([[10,  8,  9,  3,  8],
       [ 8,  0,  5,  3, 10],
       [ 9,  9, 10,  5,  7],
       [ 6,  0,  4,  7,  8]])
```

Q4. Create a matrix of numbers randomly distributed between 0 and 1, having a size of (3,4). Use `np.random.rand` .

Sample solution (numbers in matrix must be between 0 and 1):

```
array([[ 0.11798675,  0.88034312,  0.57913209,  0.06656284],
       [ 0.66977722,  0.36144647,  0.7793515 ,  0.85881387],
       [ 0.39195763,  0.64347722,  0.17478051,  0.30934148]])
```

In [44]:

```
a = np.random.rand(3,4)
a
```

Out[44]:

```
array([[0.63755707, 0.1914464 , 0.49779411, 0.1824454 ],
       [0.91838304, 0.43182207, 0.8301881 , 0.4167763 ],
       [0.90466759, 0.40482522, 0.3311745 , 0.57213877]])
```

Q5. Create a dataset by sampling from a normal distribution. The dataset contains 1200 samples (rows). Each point has 2 attributes (columns). Use `np.random.randn` .

Sample solution (the number in matrix is sampled from a normal distribution):

```
array([[-0.08722439,  0.2021376 ],
       [-0.68195243,  0.3402292 ],
       [ 0.92526745, -1.51547866],
       ...,
       [-0.81260973,  0.45430727],
       [-0.11204656,  0.72001995],
       [-0.88341863, -0.43912629]])
```

In [102]:

```
a = np.ones((5,4))
a
```

Out[102]:

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

In [47]:

```
a = np.random.rand(1200,2)
a
```

Out[47]:

```
array([[0.70998516, 0.95263681],
       [0.76613034, 0.34290132],
       [0.33850827, 0.59255941],
       ...,
       [0.84550509, 0.14330778],
       [0.84491346, 0.45926751],
       [0.17414001, 0.04181372]])
```

SECTION 2: NUMERICAL OPERATIONS

Arithmetic operations

Typically, arithmetic operators on arrays apply *element-wise*. A *new* array is created and filled with the result.

In [48]:

```
a = np.array ([ 2, 4, 6, 8])
b = np.array ([ 1, 2, 3, 1])
```

In [49]:

```
c = a - b           # element-wise subtraction, a and b will remains unchanged
print(a)
print(b)
print(c)
```

```
[2 4 6 8]
[1 2 3 1]
[1 2 3 7]
```

In [50]:

```
a + b           # element-wise addition
```

Out[50]:

```
array([3, 6, 9, 9])
```

In [51]:

```
a + 2           # element-wise addition with constant
```

Out[51]:

```
array([ 4,  6,  8, 10])
```

In [52]:

```
a / b           # element-wise division
```

Out[52]:

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

In [53]:

```
a / 2           # element-wise division with constant
```

Out[53]:

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

In [54]:

```
a * b           # element-wise multiplication
```

Out[54]:

```
array([ 2,  8, 18,  8])
```

In [55]:

```
a ** b          # Element-wise power
```

Out[55]:

```
array([  2,  16, 216,   8], dtype=int32)
```

In [56]:

```
a += b           # this will change the content of a
a
```

Out[56]:

```
array([3, 6, 9, 9])
```

In [57]:

```
a *= b          # this will change the content of a
a
```

Out[57]:

```
array([ 3, 12, 27,  9])
```

Unlike in many matrix languages, the product operator * performs *elementwise multiplication, NOT matrix multiplication*

In [58]:

```
A = np.array( [[1,1], [0,1]] )
B = np.array( [[2,0], [3,4]] )
A*B
```

Out[58]:

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

Comparison and Logical operations

In [59]:

```
a1 = np.array([13, 22, 37, 42])
a2 = np.array([13, 22, 37, 42])
b  = np.array([2, 4, 7, 5])
```

In [60]:

```
a1 < 35             # element-wise comparison
```

Out[60]:

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

In [61]:

```
a1 == 42
```

Out[61]:

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

In [62]:

```
a1 == a2
```

Out[62]:

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

In [63]:

```
np.array_equal(a1, a2)
```

Out[63]:

```
True
```

In [64]:

```
np.array_equal(a1, b)
```

Out[64]:

```
False
```

In [65]:

```
bool1     = [True,  True, False, False]
bool2     = [False, True, False, True]
all_true  = [True, True, True, True]
all_false = [False, False, False, False]
```

In [66]:

```
np.all(bool1)                          # Checks if every value is True
```

Out[66]:

```
False
```

In [67]:

```
np.all(all_true)
```

Out[67]:

```
True
```

In [68]:

```
np.any(bool1)                          # Checks if any value is True
```

Out[68]:

```
True
```

In [69]:

```
np.any(all_false)
```

Out[69]:

```
False
```

In [70]:

```
np.logical_and(bool1, bool2)           # Element-wise AND operation
```

Out[70]:

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

In [71]:

```
np.logical_or(bool1, bool2)            # Element-wise OR operation
```

Out[71]:

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

Comparing arrays

In [72]:

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

In [73]:

```
a == b
```

Out[73]:

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

In [74]:

```
np.maximum (a, b)    # Get the element-wise maximum of all corresponding items in a and b
```

Out[74]:

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

In [75]:

```
np.minimum (a, b)    # Get the element-wise minimum of all corresponding items in a and b
```

Out[75]:

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

Applying numpy functions

In [76]:

```
a = np.random.randint(-10, 10, 5)
a
```

Out[76]:

```
array([ 8, -9,  4, -8,  7])
```

In [77]:

```
np.sin(a)
```

Out[77]:

```
array([ 0.98935825, -0.41211849, -0.7568025 , -0.98935825,  0.6569866 ])
```

In [78]:

```
np.exp(a)
```

Out[78]:

```
array([2.98095799e+03, 1.23409804e-04, 5.45981500e+01, 3.35462628e-04,
       1.09663316e+03])
```

In [79]:

```
np.abs(a)
```

Out[79]:

```
array([8, 9, 4, 8, 7])
```

In [80]:

```
np.max(a)             # get the item with the biggest value
```

Out[80]:

```
8
```

In [81]:

```
np.min(a)             # get the item with the smallest value
```

Out[81]:

```
-9
```

ndarray functions

All ndarrays come with a set of functions that can be invoked to operate on its data.

In [82]:

```
a = np.random.randint(0, 20, 10)
a
```

Out[82]:

```
array([16,  1, 10, 13,  5, 17,  6,  0, 18, 17])
```

In [83]:

```
print('sum:    ', a.sum())                # compute the sum
print('mean:   ', a.mean())               # compute the mean
print('std:    ', a.std())                # compute the standard deviation
print('min:    ', a.min())                # get the smallest value in array
print('argmin: ', a.argmin())             # find the index of the item with the smallest va
print('max:    ', a.max())                # get the maximum value
print('argmax: ', a.argmax())             # find the index of the item with the biggest val
```

```
sum:     103
mean:    10.3
std:     6.542935121182236
min:     0
argmin:  7
max:     18
argmax:  8
```

ndarray operations on 2-D arrays

In [84]:

```
a = np.random.randint (0, 20, (3, 2))
a
```

Out[84]:

```
array([[ 1,  1],
       [19,  8],
       [18, 11]])
```

In [85]:

```
print('sum:    ', a.sum())                # compute the sum
print('mean:   ', a.mean())               # compute the mean
print('std:    ', a.std())                # compute the standard deviation
print('min:    ', a.min())                # get the smallest value in array
print('argmin: ', a.argmin())             # find the index of the item with the smallest va
print('max:    ', a.max())                # get the maximum value
print('argmax: ', a.argmax())             # find the index of the item with the biggest val
```

```
sum:     58
mean:    9.666666666666666
std:     7.203394261658103
min:     1
argmin:  0
max:     19
argmax:  2
```

By default, these operations apply to the 2-D array as though it were a list of numbers, regardless of its shape.

However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

In [86]:

```
a.sum(axis=0)           # axis = 0 refers to the column dimension (sum all items in each co
```

Out[86]:

```
array([38, 20])
```

In [87]:

```
a.sum(axis=1)         # axis = 1 refers to the row dimension (sum all items in each row)
```

Out[87]:

```
array([ 2, 27, 29])
```

Exercise 2

Q1. Create a random vector with values [0, 1) of size 30. Then compute its summation, mean and standard deviation values.

Sample solution:

```
[ 0.31826591  0.64980313  0.9074009   0.01855129  0.1471393   0.21421904
  0.35889039  0.26098793  0.67206902  0.22928994  0.67655462  0.46969651
  0.59789036  0.77442506  0.786646    0.30591469  0.36607411  0.08582968
  0.82389183  0.93058502  0.25422148  0.24471819  0.90520293  0.48908817
  0.55304988  0.73125874  0.84016257  0.18487058  0.80443297  0.69220994]
summation =  15.2933401657
mean =  0.509778005522
std =  0.273489774136
```

In [116]:

```
a = np.random.random(30)
print(a)
print('summation = ',a.sum())#np.sum(a)
print('mean = ',np.mean(a))
print('std = ',a.std())
```

```
[0.28352508 0.69313792 0.44045372 0.15686774 0.54464902 0.78031476
 0.30636353 0.22195788 0.38797126 0.93638365 0.97599542 0.67238368
 0.90283411 0.84575087 0.37799404 0.09221701 0.6534109  0.55784076
 0.36156476 0.2250545  0.40651992 0.46894025 0.26923558 0.29179277
 0.4576864  0.86053391 0.5862529  0.28348786 0.27797751 0.45462208]
summation =  14.773719802597402
mean =  0.4924573267532467
std =  0.24404685095144643
```

Q2. Create a 3x3 array with random integer values `[0, 10)` . Then, find the following:

1. maximum value in the whole matrix
2. minimum value of each row of the matrix
3. sum of each column of the matrix

Sample solution:

```
[[0 6 6]
 [3 7 4]
 [3 6 5]]

maximum value in the whole matrix: 7
minimum value of each row of the matrix: [0 3 3]
sum of each column of the matrix: [ 6 19 15]
```

In [100]:

```
a = np.random.randint(0,10,(3,3))
print(a)
print('maximum value in the whole matrix: ',a.max())
print('minimum value in the whole matrix: ',a.min(axis=1))
print('sum of each column of the matrix: ',a.sum(axis=0))
```

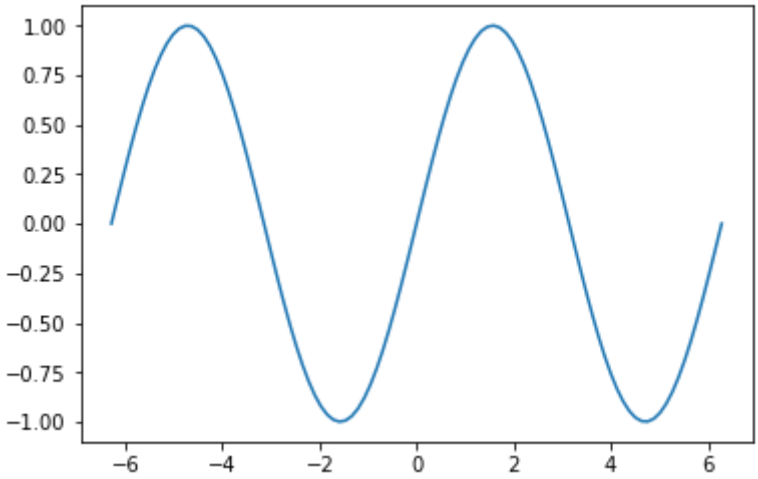
```
[[1 6 2]
 [2 1 3]
 [5 8 1]]
maximum value in the whole matrix:  8
minimum value in the whole matrix:  [1 1 1]
sum of each column of the matrix:  [ 8 15  6]
```

Q3. Generate the array containing the value of $f(x) = sin(x)$ for $-2\pi \leq x \leq 2\pi$. Compute $f(x)$ for 100 evenly spaced points in x.

Then, plot the values using the following command:

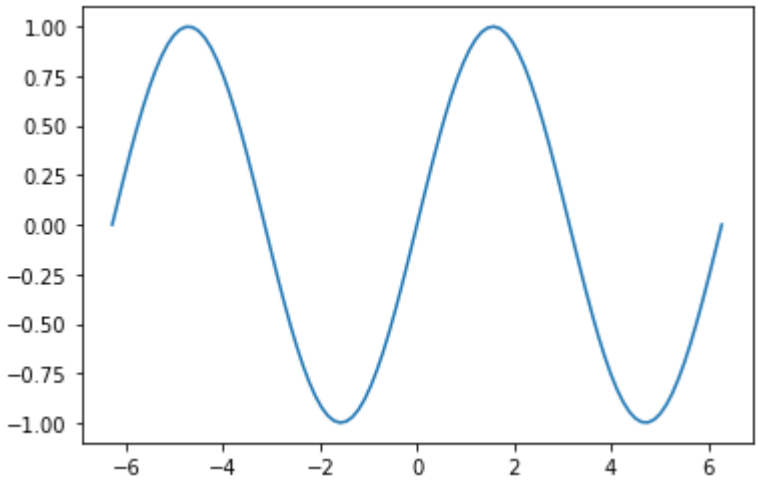
```
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.show()
```

Ans:



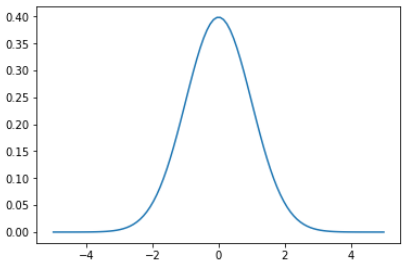
In [119]:

```
import matplotlib.pyplot as plt
x = np.linspace(-2*np.pi,2*np.pi,100)
y = np.sin(x)
plt.plot(x, y)
plt.show()
```



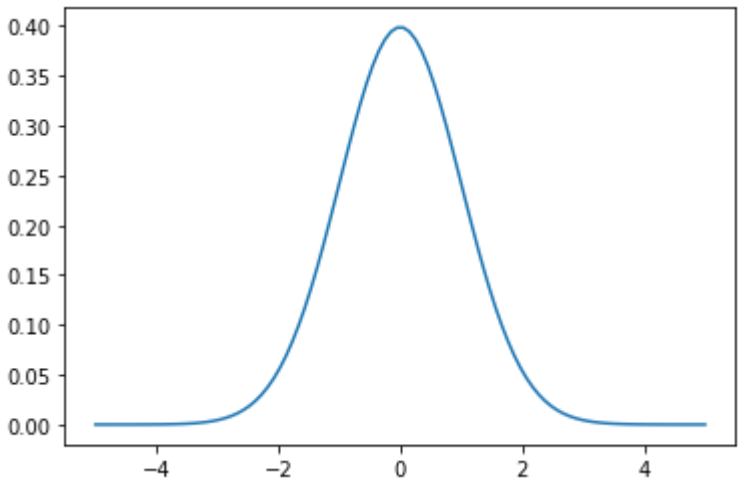
Q4. Generate the array containing the value of $f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$ for $-5 \leq x \leq 5$. The array should contain $f(x)$ for 100 evenly spaced points in x. Then, plot out the computed values. Refer to Q3 on how to plot the graph.

Ans:



In [121]:

```
x = np.linspace(-5,5,100)
y = 1/(np.sqrt(2*pi))*np.exp((-x**2/2))
plt.plot(x,y)
plt.show()
```



Q5. Given the vector `a = np.array([12, 14, 20, 26, 39, 49])` , find the closest value to `query = 27` . The algorithm to achieve this is as follows:

- compute the absolute difference between all items with the query item (Useful command: `np.abs`)
- get the item with the smallest absolute distance (Useful command: `np.argmin`)

Ans:

```
The closest number in a to 27 is 26
```

In [128]:

```
a = np.array([12,14,20,26,39,49])
query = 27
#np.argmin is used to show the location of the value
print('The closest number in a to',query,'is',a[np.argmin(np.abs(a-query))])
```

```
The closest number in a to 27 is 26
```

Q6. Randomly generate a vector `a` of size 10 with values betwen 100 and 200. Compute the L2 norm of `a` Then normalize the vector `a` to get `an` by dividing each item with the L2 norm.

For example, given `a = [4, 3]` , the L2 norm of `a` is $\sqrt{4^2 + 3^2} = 5$ and `an = [0.8, 0.6]`

In [129]:

```
a = np.random.randint(100,200+1,10)
an = a/np.sqrt(np.sum(a**2))
print('a = ',a)
print('an =', an)
```

```
a =  [132 148 109 133 160 188 155 111 184 110]
an = [0.28658836 0.32132635 0.23665251 0.28875949 0.34737984 0.40817131
 0.33652422 0.24099476 0.39948681 0.23882364]
```

Q7. Given two vectors `d1 = (5, 0, 3, 0, 2, 0, 0, 2, 0, 0)` and `d2 = (3, 0, 2, 0, 1, 1, 0, 1, 0, 1)` , compute the cosine similarity between `d1` and `d2` .

$$cos(a, b) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2}\sqrt{\sum_{i=1}^{n} b_i^2}}$$

The steps are as follows:

1. Compute A = $\sum_{i=1}^{n} a_i b_i$ = sum(5(3) + 0(0) + ... (0)(1)) = 25
2. Compute B = $\sqrt{\sum_{i=1}^{n} a_i^2}$ = sqrt(sum(5(5) + 0(0) + ... + 0(0))) = 6.48
3. Compute C = $\sqrt{\sum_{i=1}^{n} b_i^2}$ = sqrt(sum(3(3) + 0(0) + ... + 1(1))) = 4.12
4. cos(d1, d2) = A / (B*C) = 0.9356

In [122]:

```
d1 = (5, 0, 3, 0, 2, 0, 0, 2, 0, 0)
d2 = (3, 0, 2, 0, 1, 1, 0, 1, 0, 1)
a = np.array(d1)
b = np.array(d2)
A = np.sum(a*b)
B = np.sqrt(np.sum(a**2))
C = np.sqrt(np.sum(b**2))
print('cos(d1,d2) = ',A/(B*C))
```

```
cos(d1,d2) =  0.9356014857063997
```

SECTION 3: INDEXING, SLICING AND ITERATING

Simple Indexing and Slicing

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

In [123]:

```
a = np.random.randint(0, 100, 10)
print(a)
```

```
[44 48 59 74 54 91 21 56 39 29]
```

In [124]:

```
a[2]              # get item 2
```

Out[124]:

```
59
```

In [130]:

```
a[2:5]           # start = 2, end = 5 (not inclusive). This gets item 2 up to item 4
```

Out[130]:

```
array([109, 133, 160])
```

In [131]:

```
a[:3]           # equivalent to a[0:3]
```

Out[131]:

```
array([132, 148, 109])
```

In [132]:

```
a[7:]           # gets item at position 7 to the last position
```

Out[132]:

```
array([111, 184, 110])
```

We can change the content of the array using slicing.

In [133]:

```
a[2] = -1         # overwritting the third item
a
```

Out[133]:

```
array([132, 148,  -1, 133, 160, 188, 155, 111, 184, 110])
```

In [134]:

```
a[::2] = -1000    # equivalent to a[0:11:2] = -1000
a
```

Out[134]:

```
array([-1000,   148, -1000,   133, -1000,   188, -1000,   111, -1000,
         110])
```

Negative indexing

Numpy array also supports negative indexing where we index from the end of the list.

In [135]:

```
a = np.random.randint(0, 100, 10)
print(a)
```

```
[80 76 68 44 44 19 16 98 39 50]
```

In [136]:

```
a[-1]
```

Out[136]:

```
50
```

In [137]:

```
a[-3:]             # get the last three items
```

Out[137]:

```
array([98, 39, 50])
```

Multidimensional indexing

The following codes show how to index a 2-D array.

In [138]:

```
a = np.random.randint(0, 100, (3, 5))
a
```

Out[138]:

```
array([[65, 35, 45, 52,  1],
       [18, 63,  2, 87, 99],
       [20, 62, 81, 22, 92]])
```

In [139]:

```
a[2, 3]               # item at row 2, column 3 (third row, fourth column)
```

Out[139]:

```
22
```

In [140]:

```
a[:, 1]               # get the column 1 of a
```

Out[140]:

```
array([35, 63, 62])
```

In [141]:

```
a[:2, 1]              # first two items in column 1 of a
```

Out[141]:

```
array([35, 63])
```

In [142]:

```
a[:, 1:3]            # get column 1 and 2 (not inclusive of 3)
```

Out[142]:

```
array([[35, 45],
       [63,  2],
       [62, 81]])
```

When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

In [143]:

```
a[2]              # get row 2
```

Out[143]:

```
array([20, 62, 81, 22, 92])
```

In [144]:

```
a[1:]             # get row 1 to the last row
```

Out[144]:

```
array([[18, 63,  2, 87, 99],
       [20, 62, 81, 22, 92]])
```

Indexing with Integer Arrays

Indexing a 1-D array

In [145]:

```
a = np.array([10, 11, 13,  0, 10, 10, 16, 12, 18,  9])

selected = [2, 3, 5]   #select third, fourth and sixth item from array
b = a[selected]
b
```

Out[145]:

```
array([13,  0, 10])
```

Indexing a 2-D array

In [3]:

```
import numpy as np
a = np.array([[2, 5, 1, 1, 2],
              [5, 7, 2, 7, 35],
              [1, 9, 8, 49, 9]])

selected_row = (2, 1)
selected_col = (3, 4)

b = a[selected_row, selected_col]             # extracts item at location (2, 3) and (1, 4
b
```

Out[3]:

```
array([49, 35])
```

Indexing with Boolean Arrays

We can also index an array using boolean arrays.

1-D indexing using Boolean Arrays

In [147]:

```
a = np.array([23, -56, 2, 3, -57])
a
```

Out[147]:

```
array([ 23, -56,   2,   3, -57])
```

In [148]:

```
selected = [True, False, True, False, False]
a[selected]
```

Out[148]:

```
array([23,  2])
```

In [149]:

```
all_pos = a > 0
a[all_pos]                # extract all positive numbers from a
```

Out[149]:

```
array([23,  2,  3])
```

In [150]:

```
isEven = a % 2 == 0
a [isEven]                # extract all even numbers from a
```

Out[150]:

```
array([-56,   2])
```

2-D indexing using Boolean Arrays

In [151]:

```
a = np.array([[2, 5, 1, 1, 2],
              [5, 7, 2, 7, 35],
              [1, 9, 8, 49, 9]])
```

In [152]:

```
sel_rows = [False, True, True]
a[sel_rows]                                  # select the last two rows
```

Out[152]:

```
array([[ 5,  7,  2,  7, 35],
       [ 1,  9,  8, 49,  9]])
```

In [153]:

```
sel_cols = [True, True, False, False, False]
a[:, sel_cols]                               # select the first two columns
```

Out[153]:

```
array([[2, 5],
       [5, 7],
       [1, 9]])
```

Exercise 3

Q1. Create a vector of zeros (integers) of size 10. Use slicing to set the first two items with 2, the next 3 items with 4 and the remaining items with 6.

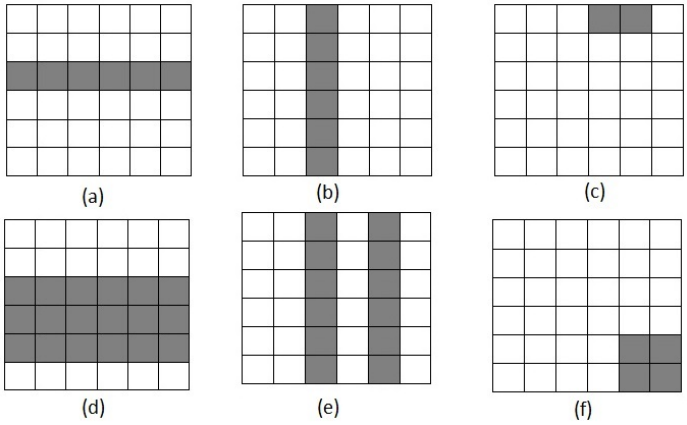
In [156]:

```
a = np.zeros(10)
a[:2] = 2
a[2:5] = 4
a[5:] = 6
a
```

Out[156]:

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

Q2. Generate a random matrix of size 6x6. Then write the codes to extract the following items from the matrix.



In [163]:

```
a = np.random.randint(0,10,(6,6))
print(a)
print('(a)\n',a[2])
print('(b)\n',a[:,2])
print('(c)\n',a[0,3:5])
print('(d)\n',a[2:5])
print('(e)\n',a[:,2:5:2])
print('(f)\n',a[4:,4:])
```

```
[[5 7 5 9 7 2]
 [6 4 8 4 9 3]
 [8 5 6 3 8 4]
 [2 7 7 2 2 0]
 [7 5 0 3 3 5]
 [5 4 7 2 2 6]]
(a)
 [8 5 6 3 8 4]
(b)
 [5 8 6 7 0 7]
(c)
 [9 7]
(d)
 [[8 5 6 3 8 4]
 [2 7 7 2 2 0]
 [7 5 0 3 3 5]]
(e)
 [[5 7]
 [8 9]
 [6 8]
 [7 2]
 [0 3]
 [7 2]]
(f)
 [[3 5]
 [2 6]]
```

Q3. Randomly generate an array comprising all numbers of size 20 with numbers between 0 and 100. Then, extract the array of numbers that are both even and divisible by 3.

In [167]:

```
a = np.random.randint(0,101,20)
print(a)
print(a[np.logical_and(a%2==0,a%3==0)])
```

```
[29 63 21 26 12 23 17 30 36  7 74 25 99 10 98 51  1 84 53 82]
[12 30 36 84]
```

SECTION 4: MATRIX COMPUTATION

Basic matrix computation

In [168]:

```
A = np.array([[1, 0], [2, 5], [3, 1]], dtype = 'int')
print(A)

B = np.array([[4, 0.5], [2, 5], [0, 1]], dtype = 'int')
print(B)
```

```
[[1 0]
 [2 5]
 [3 1]]
[[4 0]
 [2 5]
 [0 1]]
```

In [169]:

```
A + B           # A plus B
```

Out[169]:

```
array([[ 5,  0],
       [ 4, 10],
       [ 3,  2]])
```

In [170]:

```
A - B           # A subtract B
```

Out[170]:

```
array([[-3,  0],
       [ 0,  0],
       [ 3,  0]])
```

Matrix multiplication

`np.dot`：

In numpy, matrix multiplication is performed through `np.dot`. The operator `*` only performs element-wise multiplication.

In [4]:

```
m1 = np.array([[1, 2],[1, 4]])
print (m1)

m2 = np.array([[0, 1],[1, 2]])
print (m2)

v1 = np.array([2, 3])
print(v1)

v2 = np.array([1, 5])
print(v2)
```

```
[[1 2]
 [1 4]]
[[0 1]
 [1 2]]
[2 3]
[1 5]
```

In [5]:

```
np.dot(m1, m2)                    # matrix matrix multiplication
```

Out[5]:

```
array([[2, 5],
       [4, 9]])
```

In [6]:

```
m1*m2
```

Out[6]:

```
array([[0, 2],
       [1, 8]])
```

In [173]:

```
m1.dot(m2)                       # matrix matrix multiplication
```

Out[173]:

```
array([[2, 5],
       [4, 9]])
```

In [174]:

```
np.dot(m1, v1)                    # matrix vector multiplication
```

Out[174]:

```
array([ 8, 14])
```

In [175]:

```
np.dot(v1, v2)                    # vector-vector multiplication (or dot product)
```

Out[175]:

```
17
```

Transpose a matrix

In [176]:

```
A.T
```

Out[176]:

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

Invert a matrix

Inversion can only work on square matrix

In [177]:

```
b = np.random.randint(0, 10, (3, 3))
print(b)
```

```
[[4 1 3]
 [2 1 4]
 [5 8 3]]
```

In [178]:

```
np.linalg.inv(b)
```

Out[178]:

```
array([[ 0.42028986, -0.30434783, -0.01449275],
       [-0.20289855,  0.04347826,  0.14492754],
       [-0.15942029,  0.39130435, -0.02898551]])
```

Exercise 4

Q1. First, set the seed for your random generator to 42. Then, randomly sample from the standard *normal* distribution: (a) predicted output vector `h` and (b) actual output vector `y`, both having a size of (m,) where m = 50 (Use `np.random.randn`).

Then, compute the RMSE.

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(h^{(i)} - y^{(i)})^2}$$

$$= \sqrt{\frac{1}{m}(\mathbf{h} - \mathbf{y})^T(\mathbf{h} - \mathbf{y})}$$

Expected answer: 1.2192

In [183]:

```
np.random.seed(42)
m = 50
h = np.random.randn(m)
y = np.random.randn(m)
#First
rmse1 = np.sqrt(np.sum((h-y)**2)/m)
#Second
rmse2 = np.sqrt(np.dot((h-y).T,(h-y))/m)
print('RMSE1 = ',rmse1,' RMSE2 =',rmse2)
```

```
RMSE1 =  1.2192273122578394  RMSE2 = 1.2192273122578394
```

Q2. Create the matrix `X = np.array([[1, 1],[2, 1],[3, 1]])` and `y = np.array([1, 3, 7])` . Then, compute the normal equation.

$$h = (X^T X)^{-1} X^T y$$

Expected answer: `[3, -2.33333333]`

In [187]:

```
X = np.array([[1, 1],[2, 1],[3, 1]])
y = np.array([1, 3, 7])
h1 = np.linalg.inv(X.T.dot(X))
h2 = h1.dot(X.T)
h = h2.dot(y)
#h = np.dot(np.dot(np.linalg.inv(np.dot(X.T,X)),X.T),y)
h
```

Out[187]:

```
array([ 3.        , -2.33333333])
```

SECTION 5: COPIES vs VIEWS

When operating and manipulating arrays, there are three possible scenarios on how the data is updated or returned:

1. **Shallow copy**: Returns the reference to the same data.
2. **View copy**: Similar to shallow copy where it returns the reference to the same data. However, it may change the view (namely the shape) of the data. However, the underlying data are the same.
3. **Deep copy**: Copies the data into a new array. Any changes to the new array would not affect the old one.

This may be quite confusing for beginners. But, it is important to understand this concepts.

Shallow copy

Returns the reference to the same data. This is done through the operator `=` .

In [195]:

```
a = np.arange(12)
print("a = ", a)
```

```
a =  [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

In [198]:

```
b = a
print("b = ", b)
b is a                    # for simple assignment, a and b are two names for the same ndarra
```

```
b =  [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Out[198]:

```
True
```

Changing the content of `a` affects the content of b

In [190]:

```
a[0] = -99                # change the content of a affects b as well
print(a)
print(b)
```

```
[-99   1   2   3   4   5   6   7   8   9  10  11]
[-99   1   2   3   4   5   6   7   8   9  10  11]
```

In [196]:

```
a.shape
```

Out[196]:

```
(12,)
```

In [199]:

```
b.shape = 3,4               # change the shape of b
print("b = ")
print(b)

print("a = ")               # changes to b affects a as well
print(a)
```

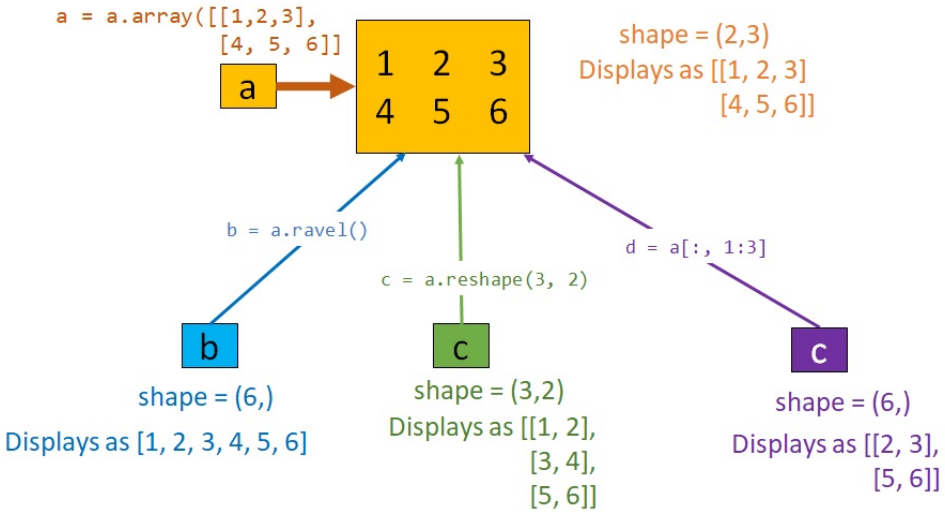
```
b =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
a =
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

View copy

Different array objects can share the *same* data but present it differently as follows:

- Display it in a different shape
- Display only portion of it

The following code creates three different view of the same data `a`.



First, we create our array `a`.

In [201]:

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

Out[201]:

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

When we flatten an array using **`.ravel()`**, it returns a *view* of the original data.

In [202]:

```
b = a.ravel()
b
```

Out[202]:

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

When we reshape an array using **`.reshape()`**, it returns a *view* of the original data.

In [203]:

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

Out[203]:

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

When we **slice** an array, we return a view of it:

In [204]:

```
d = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,1:3]"
d
```

Out[204]:

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

Changing the content in `b` or `c` or `d` will change `a` as well. The changes will be reflected in all views as well.

In [206]:

```
b[-1] = -9999
print('a:\n', a)
print('\nb:\n', b)
print('\nc:\n', c)
print('\nd:\n', d)
```

```
a:
 [[    1     2     3]
 [    4     5 -9999]]

b:
 [    1     2     3     4     5 -9999]

c:
 [[    1     2]
 [    3     4]
 [    5 -9999]]

d:
 [[    2     3]
 [    5 -9999]]
```

Deep copy

The `copy` method makes a complete copy of the array and its data.

In [207]:

```
a = np.arange(15).reshape(3, 5)
d = a.copy()                          # a new array object with new data is created
```

In [208]:

```
d is a                                # d is not the same as a
```

Out[208]:

```
False
```

In [209]:

```
d[0,0] = 9999                          # Changing the content of d does not affect a
print("a = ")
print(a)
print("d = ")
print(d)
```

```
a =
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
d =
[[9999    1    2    3    4]
 [   5    6    7    8    9]
 [  10   11   12   13   14]]
```

SECTION 6: MANIPULATING ARRAYS

We have already seen how we can manipulate shape of the arrays through the commands **`.ravel()`** and **`.reshape()`** .

In this section, we further see how to:

1. combine matrices and
2. create a new dimensions to an array.

In [210]:

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

Out[210]:

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

In [211]:

```
b = np.array([[23,23,34], [43,54,63]])
b
```

Out[211]:

```
array([[23, 23, 34],
       [43, 54, 63]])
```

`hstack`

Stacks two arrays horizontally. The two arrays must have the same number of rows.

In [212]:

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

Out[212]:

```
array([[ 1,  2,  3, 23, 23, 34],
       [ 4,  5,  6, 43, 54, 63]])
```

vstack

Stacks two arrays vertically. The two arrays must have the same number of columns.

In [213]:

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

Out[213]:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [23, 23, 34],
       [43, 54, 63]])
```

Notes: *numpy array is not designed to dynamically add or remove items. If you need to expand or shrink your arrays consistently, it is better to represent the data using a normal Python list.*

newaxis

We can use `newaxis` to convert an array (vector) of size `n` into a 2-D array (matrix) of size `nx1`.

In [214]:

```
a = np.array([4, 2])
print(a)
print(a.shape) #rank-1 array
```

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

In [217]:

```
c = a[np.newaxis,:]
print(c)
print(c.shape)
```

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

In [215]:

```
b = a[:, np.newaxis]
print(b)
print('shape =', b.shape)#vector array
```

```
[[4]
 [2]]
shape = (2, 1)
```

Exercise 5

Q1. Create the following matrix:

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

Use the command `arange` and `reshape` to do this.

In [216]:

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

Out[216]:

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

Q2. Create the matrix `a` as follows: generate a `10x10` matrix of zeros. Then, frame it with a border of ones.

In [273]:

```
a = np.zeros((10,10))
border1 = np.ones((10))
border2 = np.ones((12))
border2 = border2.reshape(12,1)
c = np.vstack((a,border1))
c = np.vstack((border1,c))
c = np.hstack((border2,c))
c = np.hstack((c,border2))
c
```

Out[273]:

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

Q3. Create a 8x8 matrix and the fill it with a checkerboard pattern.

- First, create a zero matrix of size 8x8
- Set the odd rows, even numbers to 1 using slicing
- Set the even rows, odd numbers to 1 using slicing

In [236]:

```
a = np.zeros((8,8))
a[::2,1::2] = 1
a[1::2,::2] = 1
a
```

Out[236]:

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

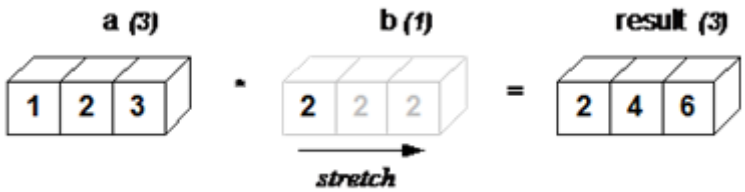
SECTION 7: ARRAY BROADCASTING

By default, numpy performs element-wise operations. This requires the input arrays to be of the same shape.

When the input arrays have different shape, it may perform **broadcasting** before carrying out the operation. Subject to certain constraints, the smaller array is "broadcast" across the larger array so that they have compatible shapes.

2-D array and scalar

Let's take an example when we want multiply an array with a scalar:



In [218]:

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

Out[218]:

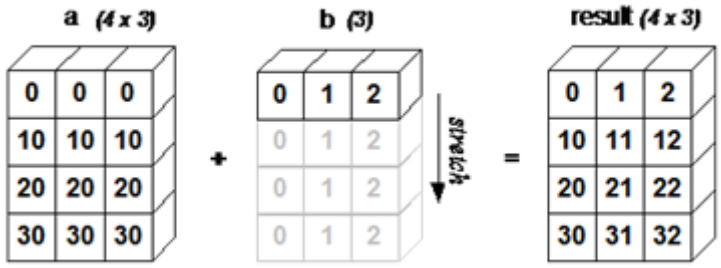
```
array([2, 4, 6])
```

Note that `numpy` does not actually stretches `b` to create a new array but is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

2-D array and 1-D array

Now, let's try to add a 2-D array with a 1-D array. If the size of the first dimension of the arrays are the same, then numpy will peform broadcasting by default.

In the following example,



Note that the shape of `a` is (4, 3) and the shape of `b` is (3,). Since the size of first dimension are the same, i.e., 3, numpy will broadcast `b` to be added to each row of `a`.

In [219]:

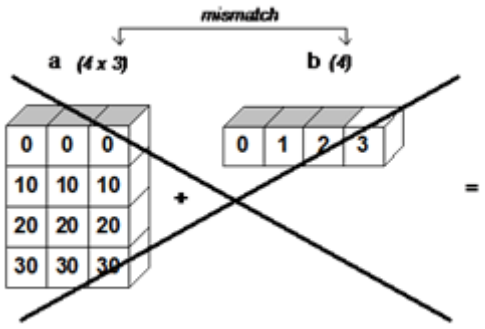
```
a = np.array([[ 0,  0,  0],
              [10, 10, 10],
              [20, 20, 20],
              [30, 30, 30]])

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

Out[219]:

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Reminder: *For broadcasting to work, the number of dimensions must be consistent, i.e., the size of the trailing axes for both arrays in an operation must be either be the same size.*



Exercise 6

Q1. Randomly generate a matrix of size (5, 3). Normalize the *columns* such that the sum of each column is equal to 1. Use broadcasting in your code to simplify your task.

In [241]:

```
a = np.random.rand(5,3)
b = 1
c = b-a
print(a+c)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```