Introduction to Numpy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object. In this course, we will be using NumPy for linear algebra.

If you are interested in learning more about NumPy, you can find the user guide and reference at the documentation.

Feel free to alter the code blocks of this notebook in case you want to try something on your own.

Let's first import the NumPy package

In [1]:
```
import numpy as np # we commonly use the np abbreviation when referring to numpy
```

Creating Numpy Arrays

A Numpy array is similar to a Python list. However, it can only contain elements of the same type.

New arrays can be made in several ways. We can take an existing list and convert it to a numpy array:

In [2]:
```
a = np.array([1,2,3])
print(a)
```

```
[1 2 3]
```

In [3]:
```
b = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
print(b)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Note that for 2-dimensional Numpy arrays, all rows must have the same number of elements.

Shape of Numpy Arrays

You can use np.shape to get the shape of a Numpy array.

In [8]:
```
# This is a one dimensional array
a = np.array([1, 2, 3])
print(a.shape)
```

```
(3,)
```

In [5]:
```
# This is a two dimensional array
b = np.array([[1, 2],
              [3, 4],
              [5, 6]])
print(b.shape)
```

```
(3, 2)
```

Note the difference in shape between one dimensional and two dimensional arrays.

For two dimensional arrays, you can consider the first element of the tuple to be the number of rows and the second element to be the number of columns.

Creating Numpy Arrays Based on Shape

You can use the functions np.zeros() and np.ones() to create Numpy arrays of a specified shape. The input of the function is the shape of the Numpy array which you want to get.

In [9]:
```
a = np.zeros((5,))
print(a)
```

```
[0. 0. 0. 0. 0.]
```

In [10]:
```
b = np.zeros((2, 3))
print(b)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

In [11]:
```
b = np.ones((3, 2))
print(b)
```

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

Reshaping Arrays

You can use the function np.reshape() to change the shape of a Numpy array.

One use of this function is to change one dimensional Numpy arrays to two dimensional row vectors or column vectors.

You pass in the required dimension which the array needs to be reshaped to as input to the function. One of the values in the tuple can be -1. This value shall then be determined automatically. You shall get an error if you have more than one -1 in the shape.

Reshape is often used in conjunction with np.arange().

`np.arange(x)` shall return a one dimensional array which contains the elements $[0, 1, \ldots, x-1]$.

In [12]:
```
a = np.arange(3)
print(a)
print(a.shape)
```

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

In [13]:
```
# Reshaping it to a row vector
# The first value of the tuple is 1 because a row vector only has 1 row
a = a.reshape((1, -1))
print(a)
print(a.shape)
```

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

In [14]:
```
# Reshaping it to a column vector
# Why does this shape work?
a = a.reshape((-1, 1))
print(a)
print(a.shape)
```

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

In [15]:
```
# Reshaping an array we get from arange
a = np.arange(6)
print("A: Original")
print(a)
print(a.shape)

a = a.reshape((3, 2))
print("A: Reshaped")
print(a)
print(a.shape)
```

```
A: Original
[0 1 2 3 4 5]
(6,)
A: Reshaped
[[0 1]
 [2 3]
 [4 5]]
(3, 2)
```

Accessing Numpy Arrays

You can use the common square bracket notation for accessing the elements of a Numpy array.

If A is a one dimensional array, then you can use `A[i]` to access the i^{th} element of A.

If A is a two dimensional array, then you can use `A[i, j]` to access element $a_{i,j}$, where i is the row and j is the column.

Note that Numpy arrays are 0 indexed.

In [16]:
```
a = np.arange(4)

print(a)

print("The 2th element of A")
print(a[2])
```

```
[0 1 2 3]
The 2th element of A
2
```

In [17]:
```
a = np.arange(9).reshape(3, 3)

print(a)

print("The 1th row of the array:")
print(a[1])

print("Element at 0th row and 0th column:")
print(a[0, 0])

print("Element at 2th row and 1th column:")
print(a[2, 1])
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
The 1th row of the array:
[3 4 5]
Element at 0th row and 0th column:
0
Element at 2th row and 1th column:
7
```

Array Slicing

You can select more than one element along a certain dimenson by using the slicing notation. Suppose A is a one dimensional array. If you want to get all elements from an array with $index >= start$ but $index < end$, you can use the following: `A[start:end]`

To get all the elements, you can use: `A[:]`

Read more about the slicing notation in the documentation.

You can conveniently get the i^{th} column of a two dimensional array A by using: `A[:, i]`

Think of this as getting all the rows of the i^{th} column of A. Note that this returns the result as a one dimensional array. You need to reshape it if you want the answer as a column vector.

In [18]:
```
a = np.arange(16).reshape(4, 4)
print(a)
```

```
print("The 1th column of the array:")
print(a[:, 1])

print("The 3th column of the array reshaped into a column vector:")
print(a[:, 3].reshape(-1, 1))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
The 1th column of the array:
[ 1  5  9 13]
The 3th column of the array reshaped into a column vector:
[[ 3]
 [ 7]
 [11]
 [15]]
```

You shall now apply what you have learned so for in the following question.

Finding The Trace of a Matrix

In the next function, you shall be given a square matrix $\boldsymbol{A}$. You need to find the *trace* of $\boldsymbol{A}$, which is defined to be the sum of all elements on the principal diagonal of $\boldsymbol{A}$.

Hint: Each element of the principal diagonal has an index of the form (i, i).

In [24]:
```
def trace(A):
    """
    Find the trace of A

    Arguments:
    A: A two dimensional square matrix

    Returns:
    s: The trace of A
    """

    s = 0

    ### BEGIN SOLUTION

    # Get the number of rows in A
    n = len(A)  # Replace the 0 with the correct code to get the number of rows

    # Loop over all elements of the principal diagonal
    for i in range(n):
        s += A[i][i] # Replace the None with the required element of A

    # END SOLUTION

    return s
```

In order to test your code, we shall be using the testing module of Numpy. More on this later.

In [25]:
```
A = np.array([[3, 2, 7],
              [4, 9, 0],
              [1, 8, 5]])
s_exp = 17
np.testing.assert_allclose(trace(A), s_exp, rtol=1e-5)

A = np.array([[12, -2, 31, 18],
              [32, -77, -24, 19],
              [87, 93, -53, 13],
              [49, 81, 63, 70]])
s_exp = -48
np.testing.assert_allclose(trace(A), s_exp, rtol=1e-5)

print("All tests passed!")
```

```
All tests passed!
```

Operations on Numpy Arrays

You can use the operations '`*`', '`+`', '`-`', '`**`' and '`/`' on numpy arrays and they operate element-wise.

Generally, both the arrays must have the same shape, but see broadcasting later in the notebook.

In [26]:
```
a = np.array([[4, 1, 2],
              [7, 2, 3]])

b = np.array([[3, 6, 9],
              [7, 8, 2]])
```

In [27]:
```
print(a + b)
```

```
[[ 7  7 11]
 [14 10  5]]
```

In [28]:
```
print(a - b)
```

```
[[ 1 -5 -7]
 [ 0 -6  1]]
```

In [29]:
```
print(a * b)
```

```
[[12  6 18]
 [49 16  6]]
```

In [30]:
```
print(a / b)
```

```
[[1.33333333 0.16666667 0.22222222]
 [1.         0.25       1.5       ]]
```

In [31]:
```
print(a ** b)
```

```
[[    64      1    512]
 [823543    256      9]]
```

Some other useful functions is Numpy are np.sqrt(), np.exp(), np.log() which apply the corresponding operation to every element of the inputted Numpy array.

In [32]:
```
print(np.sqrt(a))
```

```
[[2.         1.         1.41421356]
 [2.64575131 1.41421356 1.73205081]]
```

In [33]:
```
print(np.exp(a))
```

```
[[  54.59815003    2.71828183    7.3890561 ]
 [1096.63315843    7.3890561    20.08553692]]
```

In [34]:
```
print(np.log(a))
```

```
[[1.38629436 0.         0.69314718]
 [1.94591015 0.69314718 1.09861229]]
```

Broadcasting

Broadcasting allows you to perform element wise operations on Numpy arrays that are not of the same dimension but can be strected/duplicated so that they are of the same dimenson.

The simplest example for this is when you have to multiply all elements of a Numpy array with a scaler or add a scaler to all elements of the array.

In [35]:
```
print(a * 2)
```

```
[[ 8  2  4]
 [14  4  6]]
```

In [36]:
```
print(a + 2)
```

```
[[6 3 4]
 [9 4 5]]
```

The way to think about this is to imagine that the number 2 is being duplicated so that it fills all the elements of a Numpy array which has the same shape as that of a. Then, the regular element-wise operation is applicable.

Broadcasting can also be applied during operation between two arrays of different shapes. Suppose that a is an array of shape $(2, 3)$ whereas c is an array of shape $(1, 3)$. Then, when an operation is performed between a and c, you can think of c being duplicated along the axis of the rows so that its shape is $(2, 3)$.

In [37]:
```
c = np.array([5, 3, 4]).reshape((1, 3))

print('a:\n', a)
print('c:\n', c)
print('a + c:\n', a + c)
```

```
a:
 [[4 1 2]
 [7 2 3]]
c:
 [[5 3 4]]
a + c:
 [[ 9  4  6]
 [12  5  7]]
```

You shall get an error if you try to perform an element wise operation on two arrays which cannot be broadcasted together.

In [38]:
```
a = np.arange(6).reshape((2, 3))
print(a)

b = np.arange(4).reshape((2, 2))
print(b)

print(a + b)
```

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

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-38-e2b1a9f1400d> in <module>
      5 print(b)
      6
----> 7 print(a + b)

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

Broadcasting is a very powerful feauture of Numpy which often reduces the complexity of your code.

To know more about broadcasting, you can read the documentation.

Now you will write a function which makes use of broadcasting to perform operations on a Numpy array.

Sigmoid Function

Sigmoid is a very important function in machine learning. Its formula is given by:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Write a function which computes the sigmoid of x. Note that x might be a real number or a Numpy array.

In [49]:
```
def sigmoid(x):
    """
    Computes the sigmoid of x

    Arguments:
    x: A real number or a Numpy array

    Returns:
    s: The sigmoid of x
    """

    ### BEGIN SOLUTION
    s = 1 / (1 + np.exp(-x))
    ### END SOLUTION

    return s
```

In [50]:
```
x = 2.1
expected = 0.8909031
np.testing.assert_allclose(sigmoid(x), expected, rtol=1e-5)

x = np.array([[3.4, -7.1, 9.4],
              [-2.7, 8.882, -2.114]])
expected = np.array([[9.67704535e-01, 8.24424686e-04, 9.99917283e-01],
                     [6.29733561e-02, 9.99861153e-01, 1.07743524e-01]])
np.testing.assert_allclose(sigmoid(x), expected, rtol=1e-5)

print("All tests passed!")
```

```
All tests passed!
```

Performing Operations Along an Axis

Numpy also has other useful functions like np.sum(), np.max() and np.min().

When you just pass the Numpy array to these functions, they simply return the answer of the operation over the entire array. Now suppose you want to find the maximum element of each column of an array. You can do this by passing an additional argument called `axis` to the function.

If $\boldsymbol{A}$ is a two dimensional array, if you want to find the maximum of each column of $\boldsymbol{A}$, you can use: `np.max(A, axis=0)`

If $\boldsymbol{A}$ is a two dimensional array, if you want to find the maximum of each row of $\boldsymbol{A}$, you can use: `np.max(A, axis=1)`

A point to note is that these functions may return one dimensional arrays, which might cause errors with broadcasting. In order to make sure that two dimensional arrays are returned, you must pass the argument `keepdims=True` in the function: `np.max(A, axis=0, keepdims=True)`

In [51]:
```
# Minimum of all elements of array

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

print(a)
print(np.min(a))
```

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

In [52]:
```
# Sum of all columns of an array, returned as one dimensional array
print(np.sum(a, axis=0))
```

```
[6 6 9]
```

In [53]:
```
# Sum of all columns of an array, returned as a two dimensional array
print(np.sum(a, axis=0, keepdims=True))
```

```
[[6 6 9]]
```

In [54]:
```
# Sum of all rows of an array
print(np.sum(a, axis=1, keepdims=True))
```

```
[[ 6]
 [15]]
```

Normalizing All Columns of an Array

Before a machine learning model is applied to data, it is very common to first normalize it. In the next exercise, you will need to normalize all the columns of an array. Suppose that x_i is the i^{th} column of the input array, and c_i is the i^{th} column of the output array, then:

$$c_i = \frac{x_i - \text{mean}(x_i)}{\sigma(x_i)}$$

where $\text{mean}(x_i)$ is the algebraic mean of all the elements of x_i and $\sigma(x_i)$ is the standard deviation of all the elements of x_i.

For this exercise, you may assume that $\sigma(x)$ is never equal to 0.

Try to complete this exercise without using any for loops. Numpy functions make use of a technique called vectorization which make them much faster than for loops. Using vectorized implementations can often make magnitudes of difference in the training time of your models: where your model might initially take a couple of days to train, it would now be done in a couple of hours.

If you are stuck, you may find np.mean() and np.std() useful.

In [61]:
```
def normalize(x):
    """
    Normalize all the columns of x

    Arguments:
    x: A two dimensional Numpy array

    Returns:
    c: The normalized version of x
    """

    ### BEGIN SOLUTION

    # Calculate the mean of all columns
    mean = np.mean(x, axis=0, keepdims=True)

    # Calculate the standard deviation all columns
    sigma = np.std(x, axis=0, keepdims=True)

    # Compute the final answer
    c = (x - mean) / sigma

    ### END SOLUTION

    return c
```

In [62]:
```
x = np.array([[1, 4],
              [3, 2]])
expected = np.array([[-1, 1],
                     [1, -1]])
np.testing.assert_allclose(normalize(x), expected, rtol=1e-5)

x = np.array([[324.33, 136.11, 239.38, 237.17],
              [123.43, 325.24, 243.52, 745.25],
              [856.36, 903.02, 430.25, 531.35]])
expected = np.array([[-0.35694152, -0.97689929, -0.73023324, -1.28391946],
                     [-1.00662188, -0.39712973, -0.68372539,  1.1554411 ],
                     [ 1.3635634 ,  1.37402902,  1.41395863,  0.1284783 ]])
np.testing.assert_allclose(normalize(x), expected, rtol=1e-5)
```

```
print("All tests passed!")
```

```
All tests passed!
```

Linear Algebra

In this course, we use Numpy arrays for linear algebra. We shall use one dimensional arrays or two dimensional column vectors (two dimensional arrays of shape $(d, 1)$) to represent vectors and two dimensional arrays to represent matrices.

In [63]:
```
a = np.arange(6)
print('A Vector:\n', a)

a = np.array([[1, 2, 3],
              [4, 5, 6]]) # Can you think of a better way to create this array?
print('A Matrix:\n', a)
```

```
A Vector:
 [0 1 2 3 4 5]
A Matrix:
 [[1 2 3]
 [4 5 6]]
```

Transposing a Matrix

Suppose `A` is a matrix, then you can take the transpose of `A` with `A.T`

In [64]:
```
A = np.array([[5, 2, 9],
              [6, 1, 0]])

print('A\n', A)
print('A.T\n', A.T)

B = np.arange(9).reshape((3, 3))

print('B\n', B)
print('B.T\n', B.T)
```

```
A
 [[5 2 9]
 [6 1 0]]
A.T
 [[5 6]
 [2 1]
 [9 0]]
B
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
B.T
 [[0 3 6]
 [1 4 7]
 [2 5 8]]
```

Note that taking the transpose of a 1D array has **NO** effect.

In [65]:
```
a = np.ones(3)
print('A:\n', a)
print('Shape of A:\n', a.shape)
print('A.T:\n', a.T)
print('Shape of A.T:\n', a.T.shape)
```

```
A:
 [1. 1. 1.]
Shape of A:
 (3,)
A.T:
 [1. 1. 1.]
Shape of A.T:
 (3,)
```

But it does work if you have a 2D array of shape $(d, 1)$

In [66]:
```
a = np.ones((3,1))
print('A:\n', a)
print('Shape of A:\n', a.shape)
print('A.T:\n', a.T)
print('Shape of A.T:\n', a.T.shape)
```

```
A:
 [[1.]
 [1.]
 [1.]]
Shape of A:
 (3, 1)
A.T:
 [[1. 1. 1.]]
Shape of A.T:
 (1, 3)
```

Dot Product and Matrix Product

You can compute the dot product of two vectors or the matrix product between two matrices using np.dot() or the `@` operator. Going forward, we shall only be using the `@` operator.

In [67]:
```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
print('x:\n', x)
print('y:\n', y)
print('np.dot(x, y):\n', np.dot(x, y))
print('x @ y:\n', x @ y)
```

```
x:
 [1 2 3]
y:
 [4 5 6]
np.dot(x, y):
 32
x @ y:
 32
```

You can use this operator to compute matrix-matrix and matrix-vector product as well.

In [68]:
```
A = np.array([[2, 0, 1],
              [1, 3, 4],
              [0, 2, 1]])

B = np.arange(6).reshape(3, 2)

C = np.array([3, 2, 8]).reshape((-1, 1))

print('A:\n', A)
print('B:\n', B)
print('A @ B:\n', A @ B)

print('A:\n', A)
print('C:\n', C)
print('A @ C:\n', A @ C)
```

```
A:
 [[2 0 1]
 [1 3 4]
 [0 2 1]]
B:
 [[0 1]
 [2 3]
 [4 5]]
A @ B:
 [[ 4  7]
 [22 30]
 [ 8 11]]
A:
 [[2 0 1]
 [1 3 4]
 [0 2 1]]
C:
 [[3]
 [2]
 [8]]
A @ C:
 [[14]
 [41]
 [12]]
```

Note that if you try to multiply two matrices which are not compatible for multiplication, you get an error.

In [69]:
```
print(B.shape)
print(A.shape)
print(B @ A)
```

```
(3, 2)
(3, 3)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-69-6f43b78721c1> in <module>
      1 print(B.shape)
      2 print(A.shape)
----> 3 print(B @ A)

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size
3 is different from 2)
```

The Linalg Library of Numpy

The Numpy library ships with np.linalg package which lets us compute many properties of matrices.

For example, we can compute the determinant of a square matrix by using np.linalg.det().

In [70]:
```
# This computes the determinant
print(np.linalg.det(A))
```

```
-7.999999999999998
```

We can compute the inverse of a matrix by using np.linalg.inv().

In [71]:
```
# This computes the inverse
print(np.linalg.inv(A))

I = np.eye(3) # We can use this function to generate the identity matrix
np.testing.assert_allclose(A @ np.linalg.inv(A), I, atol=1e-10)
```

```
[[ 0.625 -0.25   0.375]
 [ 0.125 -0.25   0.875]
 [-0.25   0.5   -0.75 ]]
```

We can compute the eigenvalues and eigenvectors of a matrix using np.linalg.eig().

In [72]:
```
# This computes the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
print("The eigenvalues are\n", eigenvalues)
print("The eigenvectors are\n", eigenvectors)
```

```
The eigenvalues are
 [ 5.10548262  1.77653793 -0.88202054]
The eigenvectors are
 [[-0.13964316  0.9724502  -0.24499029]
 [-0.8901906  -0.08437307 -0.66441635]
 [-0.43365942 -0.21730574  0.70606705]]
```

You shall now apply all the concepts that you have learned in the next sections.

Diagonalizing a Matrix

In this question, you shall be given a square matrix which you need to diagonalize. In particular, you will be given a diagonalizable matrix A and you need to find matrices S and D such that:

$$A = SDS^{-1}$$

Recall that in order to do this, you must first find all the eigenvalues and eigenvectors of A. Then, S is the matrix of all the eigenvectors arranged as columns,

and D is the matrix of the corresponding eigenvalues arranged along the diagonal.

Suppose $A = \begin{bmatrix} 1 & 5 \\ 2 & 4 \end{bmatrix}$

Then, we can calculate $S = \begin{bmatrix} -2.5 & 1 \\ 1 & 1 \end{bmatrix}$

And $D = \begin{bmatrix} -1 & 0 \\ 0 & 6 \end{bmatrix}$

You might find np.zeros(), np.linalg.eig() and np.linalg.inv() useful. Note that for this exercise, you may assume that A is always diagonalizable.

For testing purposes, each eigenvector in S must be of unit length. This shall always be the case if you use np.linalg.eig(). However, if you do not use this function, then depending on your implementation, you might have to normalize the eigenvectors. Also, the eigenvalues must appear in non decreasing order.

In [121]:
```
def diagonalize(A):
    """
    Diagonalizes the input matrix A

    Arguments:
    A: A two dimensional Numpy array which is guaranteed to be diagonalizable

    Returns:
    S, D, S_inv: As explained above
    """

    ### BEGIN SOLUTION

    # Retrieve the number of rows in A
    n = len(A)

    # Get the eigenvalues and eigenvectors of A
    eig_vals, eig_vecs = np.linalg.eig(A)
    S = eig_vecs

    # Start by initializing D to a matrix of zeros of the appropriate shape
    D = np.diag(eig_vals)

    # Set the diagonal element of D to be the eigenvalues
    for i in range(n):
        pass

    # Compute the inverse of S
    S_inv = np.linalg.inv(S)

    ### END SOLUTION

    return S, D, S_inv
```

In [122]:
```
A = np.array([[1, 5],
              [2, 4]])
S_exp = np.array([[-0.92847669, -0.70710678],
                  [ 0.37139068, -0.70710678]])
D_exp = np.array([[-1, 0],
                  [0, 6]])
S_inv_exp = np.array([[-0.76930926,  0.76930926],
                      [-0.40406102, -1.01015254]])


S, D, S_inv = diagonalize(A)
np.testing.assert_allclose(S_exp, S, rtol=1e-5, atol=1e-10)
np.testing.assert_allclose(D_exp, D, rtol=1e-5, atol=1e-10)
np.testing.assert_allclose(S_inv_exp, S_inv, rtol=1e-5, atol=1e-10)

print("All tests passed!")
```

```
All tests passed!
```

Lastly, you will implement a function to carry out polynomial multiplication. Implementing this would require the application of multiple concepts that you have learned till now.

Polynomial Multiplication (Challenge)

You can challenge yourself by trying to implement this function. It is a good opportunity to practice the concepts introduced in this notebook. Feel free to skip to the next section if you want.

In this function, you shall be implementing polynomial multiplication. You will be given two one dimensional numpy arrays A and B, the coefficients of the two polynomials, where a_i is the coefficient of x^i in A. You must calculate the coefficients of $A \cdot B$.

More formally, if C is the resultant one dimensional array, then

$$c_i = \sum_{j+k=i} a_j * b_k$$

There are multiple ways to do this, and your implementation may require you to use functions which we have not introduced to you. If that is the case, we encourage you to look at the documentation.

Finally, try to implement this function using only a single for loop over i, and try to implement the summation using only inbuilt functions of Numpy. This will lead to much faster code, thanks to vectorization.

We shall not guide you through this function by as much as we did with the others.

Additional hints:

- A and B might be of different sizes. Depending on your implementation, this might have an effect. Pad the end of the smaller array with zeros so that A and B have the same size. You might want to take a look at np.pad().
- For a fixed i, try to see how j and k vary and which elements of A and B can be multiplied together. Does the resultant expression seem similar? Maybe the dot product of two slices?
- You can use np.flip() to reverse a Numpy array.
- Make sure that your answer does not have any zeros at the end. Try to find a function in Numpy which does that for you.

In case you are curious, there are faster ways to implement polynomial multiplication. If you are interested and feel (very) confident about your math and algorithmic skills, take a look at FFT.

In [123]:
```
def multiply(A, B):
    """
    Multiplies two polynomials

    Arguments:
    A: Coefficients of the first polynomial
    B: Coefficients of the second polynomial
```

```
    B: coefficients of the second polynomial

    Returns:
    C: The coefficients of A*B
    """

    ### BEGIN SOLUTION

    # Find the coefficients of both the polynomials
    na = len(A)
    nb = len(B)

    # Initialize the output array with 0s
    C = np.zeros(na + nb - 1)

    # Perform the multiplication
    # You might want to break the loop over i into two separate phases
    for i in range(na):
        for j in range(nb):
            C[i + j] += A[i] * B[j]

    return C
```

In [124]:
```
A = np.array([1, 2])
B = np.array([3, 4])
C_exp = np.array([3, 10, 8])
np.testing.assert_allclose(multiply(A, B), C_exp, rtol=1e-5, atol=1e-10)

A = np.array([5, 6])
B = np.array([1, 3, 5, 9])
C_exp = np.array([5, 21, 43, 75, 54])
np.testing.assert_allclose(multiply(A, B), C_exp, rtol=1e-5, atol=1e-10)
np.testing.assert_allclose(multiply(B, A), C_exp, rtol=1e-5, atol=1e-10)

print("All tests passed!")
```

```
All tests passed!
```

If you could successfully implement this function using a single for loop, well done!

Introduction to Assert Statements and Testing

An assert statement lets you check if a condition in your program evaluates to true.

If the condition does evaluate to true, then nothing happens and the program execution continues normally. However, if it evaluates to false, then the program execution is immediately terminated, an error is raised and an error message is printed.

In [125]:
```
# An assert statement where condition evaluates to true
x = 5
assert x == 5
```

Since the condition evaluates to true, nothing happens.

In [126]:
```
# An assert statement where the condition evaluates to false
x = 5
assert x == 4
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-126-a985966787ee> in <module>
      1 # An assert statement where the condition evaluates to false
      2 x = 5
----> 3 assert x == 4

AssertionError:
```

This time, since the assert statement evaluates to false, an error is thrown and the line where the assert statement failed is printed to the standard output.

You can also print a message which further explains what error took place.

In [127]:
```
# An asseert statement with an error message
x = 5
assert x == 4, "x does not store the intended value"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-127-ee7bee875a10> in <module>
      1 # An asseert statement with an error message
      2 x = 5
----> 3 assert x == 4, "x does not store the intended value"

AssertionError: x does not store the intended value
```

Asserts are a very powerful way to find bugs in your code. Do not hesitate to use these statements in the functions you write to assist you in debugging them. Later, we shall also be using assert statements sometimes to test the functions that you shall write in the later labs.

Let us see an example where using assert statements helps us spot bugs in our code.

In [131]:
```
def test_assert():
    """
    This function demonstrates the use of assert statements in debugging
    """

    A = np.arange(5)
    s = 0

    # We shall first add all elements of A to s
    for i in range(A.shape[0]):
        s += A[i]

    # We shall now subtract all the elements of A in the reverse order
    # Unfortunately, we have a bug
    for i in range(A.shape[0] - 1, -1, -1):
        s -= A[i]

    # Quite certainly, s must be equal to 0 at this point
    # Had our implementation been correct, this assert should pass
    assert s == 0
```

```
test_assert()
```

Can you find the bug in the code and fix it? Have a look at the documentation of range().

The Numpy Testing Module

Numpy has a very useful function called np.testing.assert_allclose() which allows us to test our functions.

The function accepts two numbers or two Numpy arrays and checks them for equality. Note that you cannot compare floating point numbers using the `==` operator as you have to account for a margin of error which can be caused due to rounding. This function allows you to customize the error margin based on your needs.

We highly advice you to read the documentation of this function.

The function takes two compulsory arguments, the first is the array which you want to test, and the second is the array which you want to test the array against. You can also configure the tolerance level if you wish.

You can see examples of how this function is used earlier in the notebook.

In the next code block, we have written a function which tries to compute the inverse of a 2 by 2 matrix. However, the function is incorrect.

In [132]:
```
def inverse(A):
    """
    Computes (incorrectly) the inverse of A

    A must have shape (2, 2)
    """

    return np.array([[A[1, 1], -A[0, 1]],
                     [-A[1, 0], A[0, 0]]])
```

We have written a test for this function but it unfortunately passes. Can you write a test for this function which fails? Then, can you modify the function so that it is now correct and passes the test that you wrote?

In [133]:
```
A = np.array([[3, 5],
              [1, 2]])
A_exp = np.array([[2, -5],
                  [-1, 3]])
np.testing.assert_allclose(inverse(A), A_exp, rtol=1e-5)
np.testing.assert_allclose(inverse(A) @ A, np.eye(2), rtol=1e-5, atol=1e-10)

# Add another test here
```

Debugging Your Code

While you are working through the rest of the labs of this course, you shall come across many situations where your code shall not work correctly. You are not alone if this happens to you. Debugging your code can be a difficult and daunting task, so in this last section, we shall give you some practical guidelines to assist you in debugging your code.

Dimension Errors in Matrix Multiplication

This is one of the most frequent errors that you shall face. Let us first see the error message that Numpy prints if you try to multiply two matrices of incompatible dimensions.

In [134]:
```
A = np.zeros((3, 2))
B = np.zeros((4, 5))
print(A @ B)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-134-c1fcd344e478> in <module>
      1 A = np.zeros((3, 2))
      2 B = np.zeros((4, 5))
----> 3 print(A @ B)

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufunc signature (n?,k),(k,m?)->(n?,m?) (size
4 is different from 2)
```

That was a mouthful! However, in this course, we shall only be performing operations on arrays that have atmost a dimension of 2, which considerably simplifies things.

The only important line in the error message is: `size 4 is different from 2`. This says that the 0th dimension of B is 4 whereas the 1th dimension of A is 2, and hence they are incompatible for matrix multiplication.

One way to debug this is to print the dimensions of all the matrices before and after each matrix multiplication and track them, because a previous error which unfortunately passed dimension checks might be causing the problems here.

Errors can also take place when you try to multiply two one dimensional arrays together or try to multiply a one dimensional array with a two dimensional array.

We would advice you to use np.outer() and np.inner() when computing the dot product of 1D arrays. If X is a vector (represented as a 1D array in this course), then `np.inner(X, X)` calculates $X^T \cdot X$ (the regular dot product) and `np.outer(X, X)` computes $X \cdot X^T$.

If you are performing matrix multiplication between a 2D array and a 1D array, we would advise you to first reshape the 1D array into a 2D array of shape $(d, 1)$.

Practicing Code Debugging

In this question, we were trying to find the sum of the maximum element of each row in a 2D array, but we have unfortunately made a bug. Can you fix it so that our tests pass?

In [138]:
```
def sum_of_max(A):
    """
    Computes the sum of the maximum element of each row of A

    A must be a 2D Numpy array
    """

    return np.sum(np.max(A, axis=1))
```

In [139]:
```
A = np.array([[1, 2],
```

```
              [3, 4]])
np.testing.assert_allclose(sum_of_max(A), 6)

A = np.array([[24, 69, 83],
              [74, 14, 27]])
np.testing.assert_allclose(sum_of_max(A), 157)
```

Congratulations on making it to the end of this really long notebook! Now you are in a good place to proceed with the remaining labs of this course.