Lecture 1

Learning outcomes:

- ➢ *Numerical Calculations with NumPy:*
- ▪ *Introduction to arrays operations*

NumPy (Numerical Python)

- NumPy is an open-source library in Python widely used in science and engineering for scientific computing and data analysis.

- NumPy library is a fundamental tool for performing high-performance mathematical operations on multidimensional arrays and matrices data structures, such as the homogeneous, N-dimensional array.

- It also enables fast and efficient data manipulation on these data structures.

Why use NumPy?

- Python lists are excellent, general-purpose containers. They can be “heterogeneous”, meaning that they can contain elements of a variety of types, and they are quite fast when used to perform individual operations on a handful of elements.
- Depending on the characteristics of the data and the types of operations that need to be performed, other containers may be more appropriate; by exploiting these characteristics, we can improve speed, reduce memory consumption, and offer a high-level syntax for performing a variety of common processing tasks.
- Where NumPy shines when there are large quantities of “homogeneous” (same-type) data to be processed on the CPU.
- One of the key features of NumPy is its ability to perform element-wise operations on arrays. This means that you can perform operations like addition, subtraction, and multiplication on entire arrays with a single command.

What is an “array”?

- In computer programming, an array is a structure for storing and retrieving data. We often talk about an array as if it were a grid in space, with each cell storing one element of the data. For instance, if each element of the data were a number, we might visualize a “one-dimensional” array like a list.

- NumPy arrays are much faster than traditional Python lists and provide a number of advanced features that make working with data much easier.

❑ Most NumPy arrays have some restrictions. For instance:

- All elements of the array must be of the same type of data.
- Once created, the total size of the array can’t change.
- The shape must be “rectangular”, not “jagged”; e.g., each row of a two-dimensional array must have the same number of columns.

✓When these conditions are met, NumPy exploits these characteristics to make the array faster, more memory efficient, and more convenient to use than less restrictive data structures.

❑ How to import NumPy: import numpy as np

- This widespread convention allows access to NumPy features with a short, recognizable prefix (np.) while distinguishing NumPy features from others that have the same name.

Create a NumPy array:

❖ There are several ways to create a NumPy array, including:

▪ From a Python list:

```
import numpy as np
array_numpy = np.array([1, 2, 3, 4, 5])
 array([1, 2, 3, 4, 5])
```

▪ With specific values:

```
array_numpy = np.array([1, 4, 9, 16, 25], dtype=float)
array([ 1.,  4.,  9., 16., 25.])
```

- Using NumPy functions:

➢ The simplest is to use the array function to make a direct definition

```
from numpy import *
a = array([[1., 2., 3.], [4., 5., 6.]])
a
```

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

➢The zeros and ones functions

A couple of other methods for generating arrays are provided by the zeros and ones functions.

✓array_zeros = np.zeros((2, 4))

✓array_ones = np.ones((3, 3, 3))

```
>>> zeros((2,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

```
>>> ones((3,3,3))
array([[[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]],

       [[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]]])
>>>
```

- ✓ The tuple in the argument list of these functions defines the shape of the array. The arrays are filled with the values indicated by the function names.

- Creating one-dimensional sequences

➢The arange function creates a one-dimensional array consisting of a sequence of numbers:

```
>>> b = arange(0,11)
>>> b
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> c = arange(0.,2.,0.4)
>>> c
array([ 0. ,  0.4,  0.8,  1.2,  1.6])
>>>
```

- ✓The first two arguments of arange are the start and stop values. Note that the sequence stops before the stop value, as with the standard Python range function. The third argument is the step value in the sequence with a default value of 1.

- ➢Floating point arithmetic causes a potential pitfall with non-integer values of the step parameter. Look at the following example:

```
>>> arange(0.,2.1,0.3)
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8,  2.1])
```

- ✓One would expect the last value in this array to be 1.8, but it is 2.1. However, with the minuscule inaccuracies in floating point arithmetic, the final value is actually a tiny bit less than 2.1, an inaccuracy that is not represented in the printed output.

- ✓Thus, the rule that the sequence stops before the stop value is reached is technically obeyed, though the result is probably not what was expected.

- ✓Be forewarned and don’t set your stop value to an element of the sequence if you are using floating point numbers! Make it one sequence element beyond the last desired element minus a small number, say, half the step size.

An unambiguous alternative to arange in such situations is the function linspace(start, stop, number). this function returns an array with number evenly spaced elements with the specified starting and stopping values:

```
>>> a = linspace(0.,3.5,8)
>>> a
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5])
>>>
```

- Two-dimensional arrays with meshgrid

✓One often deals with fields in an N-dimensional vector space as approximated by arrays defined over an N-dimensional rectangular grid. With each grid dimension one defines a one-dimensional array containing “position” values along the corresponding axis. It is useful to define N-dimensional arrays over the vector space, one for each dimension, containing these position values. Such meshgrid arrays can be used as inputs to construct fields which are functions of position.

✓For the special case of two dimensions, NumPy provides a convenient function for generating meshgrid arrays, called, appropriately, meshgrid. Meshgrid returns the two desired meshgrid arrays in a tuple:

```
>>> x = arange(0.,5.1)
>>> y = arange(0.,3.1)
>>> (X,Y) = meshgrid(x,y)
>>> X
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 0.,  1.,  2.,  3.,  4.,  5.]])
>>> Y
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.,  3.]])
```

```
>>> a = X*X + Y*Y
>>> a
array([[  0.,   1.,   4.,   9.,  16.,  25.],
       [  1.,   2.,   5.,  10.,  17.,  26.],
       [  4.,   5.,   8.,  13.,  20.,  29.],
       [  9.,  10.,  13.,  18.,  25.,  34.]])
>>> b = x*x + y*y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
>>>
```

- ✓The one-dimensional arrays x and y contain the position values along the x and y axes, while the two-dimensional arrays X and Y (which are distinct from x and y because Python variable names are case-sensitive) define these position values at each point in the vector space. Note how the array a can be defined in terms of X and Y, but the corresponding attempt to define b in terms of x and y fails.
- ✓Notice that variations in the first argument of meshgrid correspond to variations in the second index of X, Y, and a. This curious inversion probably reflects the influence of Matlab on the world of numerical computing. However, this convention is used consistently in the Matplotlib plotting module discussed in the next section, so there is no getting away from it. It does have the virtue that the first argument of meshgrid (the x axis by convention) is the dimension that corresponds to the horizontal axis in Matplotlib. However, it may not always represent the best way to organize dimensions in numerical computing.

Array data types

- NumPy supports a variety of data types to represent elements in arrays. These data types are more specific than the built-in Python data types, allowing for more efficient memory usage and faster computation. Some commonly used data types in NumPy include integers, floating-point numbers, and complex numbers, along with their variations such as signed and unsigned integers of different sizes.
- Some of the arrays created above contain integers and some contain floating point values. NumPy arrays can be made up of a variety of different numerical types, though all elements of a given array must be of the same type.
- The default float type in Python contains 64 bits (like a C-language double) and the default integer type generally contains 32 or 64 bits, depending on the architecture of the underlying computer. The type of the elements in an array can be defined upon array creation using the above functions by including the positional parameter dtype:

```
>>> a = array([1, 2], dtype = float64)
>>> a
array([ 1.,  2.])
>>> a = ones((2,3), dtype = complex128)
>>> a
array([[ 1.+0.j,  1.+0.j,  1.+0.j],
       [ 1.+0.j,  1.+0.j,  1.+0.j]])
>>> a = array([1.3, 2.7], dtype = int32)
>>> a
array([1, 2])
>>>
```

- It is possible to convert arrays from one type to another using the astype method:

```
>>> from numpy import *
>>> a = arange(0,5)
>>> a.dtype
dtype('int32')
>>> b = a.astype('float32')
>>> b
array([ 0.,  1.,  2.,  3.,  4.], dtype=float32)
>>> c = a.astype('float64')
>>> c
array([ 0.,  1.,  2.,  3.,  4.])
>>>
```

- In many cases data are imported in the form of an array from some outside source. In this case one need not create the array “by hand”.

Array indexing and looping: Access elements of a NumPy array:

❑ Basic indexing

- Individual elements and sets of elements are extracted from an array by indexing. NumPy adopts and extends the indexing methods used in standard Python for strings and lists.

- Individual elements of a NumPy array can be accessed using their row and column indexes. For example:

```
element = array_numpy[1, 2] # Access the element in row 1, column 2
```

```
>>> a = array([[2.1, 3.5, 7.0], [10.4, 12.9, 5.1]])
>>> a
array([[  2.1,   3.5,   7. ],
       [ 10.4,  12.9,   5.1]])
>>> a[0,0]
2.1
>>> a[0,1]
3.5
>>> a[1,0]
10.4
>>> a[0,-1]
7.0
>>>
```

- ✓The above examples show how to extract single elements as in standard Python.

- ✓The extension is that since NumPy arrays can be multi-dimensional, a list of N indices (really, a tuple) is needed for an N-dimensional array.

- ✓As in standard Python, indexing starts at 0 and negative indices index backwards from the end of the array, starting with -1.

Slicing of NumPy arrays:

- The slicing methods used in Python strings and lists also work for NumPy arrays:

- Slicing allows you to extract submatrices from a NumPy array by specifying ranges of rows and columns. For example:

- subarray = array_numpy[1:3, 0:2] # Extract subarray from row 1 to 2 and from column 0 to 1

```
>>> a = array([[2.1, 3.5, 7.0], [10.4, 12.9, 5.1]])
>>> a
array([[  2.1,   3.5,   7. ],
       [ 10.4,  12.9,   5.1]])
>>> b = a[:,1:2]
>>> b
array([[  3.5],
       [ 12.9]])
>>>
```

- ✓The index “:” returns all elements along the corresponding axis. Thus, a[:,1:2] creates an array consisting of the second column (column 1 with zero-based indexing) and all rows of array a.

✓Note that if the number of colons is less than the number of dimensions, then all elements of the remaining dimensions are included:

```
>>> a = zeros((2,2))
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> a[:]
array([[ 0.,  0.],
       [ 0.,  0.]])
>>>
```

✓Thus, as illustrated above, a[:] indicates the entire array regardless of the number of dimensions.

✓Also note that slicing may be used on an array on the left side of the equals sign to facilitate assignment to part of that array:

```
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b = array([1.,2.])
>>> b
array([ 1.,  2.])
>>> a[0,:] = b
>>> a
array([[ 1.,  2.],
       [ 0.,  0.]])
>>>
```

✓Here are some other useful tricks in slicing:

```
>>> a = arange(0,10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[0:10:2]
array([0, 2, 4, 6, 8])
>>> a[-1:0:-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1])
>>> a[-1::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>>
```

✓The first example illustrates the use of a non-default step of 2, while the last two show how to reverse the order of the elements in an array. The second of these demonstrates that by omitting the stop value, all elements up to the beginning of the array (or the end if the start value is positive) are included in the new view.

Array methods

- ❑ Since NumPy arrays are Python objects, they have methods associated with them. Here we describe some of the most useful methods.
- ✓Some methods operate on the array in place and don’t return anything.
- ✓Others return some object, generally a modified form of the original array. Which does which is indicated.

➢ ***General methods:***

- ▪ tolist(): This method converts a NumPy array to an ordinary Python list:

```
>>> a = arange(0.,9.5)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a.tolist()
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
>>> b = a.reshape(2,5)
>>> b
array([[ 0.,  1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.,  8.,  9.]])
>>> b.tolist()
[[0.0, 1.0, 2.0, 3.0, 4.0], [5.0, 6.0, 7.0, 8.0, 9.0]]
>>>
```

- tofile(file=, sep=”, format=”): This writes an array to a file.

✓File is either a string representing a file name, in which case this method opens this file and writes the array to it, or it is the file object for a previously opened file. If the optional keyword parameters sep and format are absent, the array is written in native binary format:

```
>>> a = array([[1., 2., 3.], [9., 8., 7.]])
>>> a.tofile("junk.bin")
```

✓No meta-information about the array is written – just the array element values, as can be inferred by the size of junk.bin; 48 bytes. The parameter sep defines an ascii string (for instance, a space or a tab) to separate the array elements. The elements are converted to ascii string format for writing. If the format command is used, then the elements are written in the specified format, e. g., “%f”, “%g”, or “%d”. For example, using the same array:

```
 a.tofile(sys.stderr, sep = " ")
1.0 2.0 3.0 9.0 8.0 7.0>>>
```

✓The Python prompt appears after the output because no newline is appended to the file.

- astype(): This method converts the type of elements to the type listed in the argument as discussed previously.
- byteswap(): This method swaps bytes as previously discussed. If the argument is False (the default), bytes are swapped in place. If True, the method returns a new array with the swapped bytes.
- fill(scalar): This fills all of the elements of the array with the scalar value and is faster than element-by-element assignment:

```
>>> a = array([1,4,7])
>>> a
array([1, 4, 7])
>>> a.fill(42)
>>> a
array([42, 42, 42])
>>>
```

- NumPy Array Shape

✓ The shape of a NumPy array defines the size of each dimension of the array. The shape attribute of a NumPy array returns a tuple representing the size of each dimension.

```
import numpy as np

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


# Output:

(2, 3)
```

- transpose(tuple of axes), T: This method transposes arrays of two and higher dimensions.

✓Transposing a NumPy array swaps the rows and columns. It can be done using the np.transpose() function:

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

✓For arrays of greater than two dimensions a tuple of axes can be used to show how the existing axes should be ordered in the returned array. a.T is shorthand for a.transpose() with no arguments.

- NumPy Array Sort

✓ You can use the sort() function to sort NumPy arrays. This function sorts the array in ascending or descending order.

```
import numpy as np
arr = np.array([3, 1, 2, 5, 4])

# Sort the array in ascending order
result = np.sort(arr)
print(result)
# Output: [1, 2, 3, 4, 5]

# Sort the array in descending order
result = np.sort(arr)[::-1]
print(result)
# Output: [5, 4, 3, 2, 1]
```

- NumPy Copy vs View

✓NumPy arrays support both copying and viewing operations.

✓A copy creates a new array with its own data, while a view creates a new array object that refers to the same data as the original array.

✓Understanding the difference between copy and view is crucial to avoid unintended side effects when working with NumPy arrays.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Create a copy of the array
arr_copy = arr.copy()
```

```
# Modify the first element of the copy
arr_copy[0] = 100

# Print the original array
print(arr)
# Output should be: [1 2 3 4 5]

# Print the modified copy
print(arr_copy)
# Output should be: [100 2 3 4 5]
```

```
view_arr = arr.view()

view_arr[0] = 100  # Modify view
print("Original:", arr)
print("View:", view_arr)

# Output should be:
Original: [100  2  3  4  5]
View: [100  2  3  4  5]
```

- NumPy Array Reshape

✓Reshaping a NumPy array means changing the shape of the array while keeping the same data. The reshape() function is used to reshape NumPy arrays.

✓Reshape changes the number and size of dimensions of an array, returning a new array (actually a new view – additional data memory isn’t allocated). The new shape is given by the newshape tuple. The total size of the array cannot change.

```
import numpy as np
array = np.array([1, 2, 3, 4, 5, 6])
reshaped_array = array.reshape(2, 3)
print(reshaped_array)
# Output:
[[1 2 3] [4 5 6]]
```

- NumPy Array Iterating

✓NumPy arrays can be iterated over using loops such as for loops. However, NumPy provides optimized functions like nditer() to iterate over arrays efficiently.

```
import numpy as np
arr = np.array([[1, 2], [3, 4]])
# Iterating over the array using nditer()
for x in np.nditer(arr):
    print(x)

# Expected Output:
# 1
# 2
# 3
# 4
```

- NumPy Array Join

Joining means putting contents of two or more arrays in a single array.

We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

result = np.concatenate((arr1, arr2))
print(result)
# Output:
[1 2 3 4 5 6]
```

```
# Joining arrays along the specified axis
# Join two 2-D arrays along rows (axis=1):

import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)
```

✓Output:

```
print(arr)
[[1 2 5 6]
 [3 4 7 8]]
```

- NumPy Array Split

✓The split() function in NumPy splits a NumPy array into two or more sub-arrays along a specified axis. The splitting operation is performed along the specified axis.

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

# Split the array into multiple sub-arrays
result = np.split(arr, 3)

print(result)
# Output:
[array([1, 2]), array([3, 4]), array([5, 6])]
```

- NumPy Array Search

✓ To search for an element in a NumPy array, you can use functions like where() or argwhere(). These functions return the indices of elements that satisfy a specific condition.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])

# Locate the index of the element that meets a specific condition
result = np.where(arr == 3)

print(result)
# Expected Output:
(array([2]))
```

Mathematical methods

- NumPy provides a wide range of functions for performing mathematical operations on arrays. Some of the most common operations include:

- Addition:

```
array_sum = array1 + array2
```

- Subtraction:

```
array_subtraction = array1 - array2
```

- Multiplication:

```
product_array = array1 * array2
```

- Division:

```
array_division = array1/array2
```

- Empowerment:

```
array_power = array1 ** 2
```

- max(axis=None): For no arguments, this returns the scalar value which is the largest element in the entire array. If an axis is defined for an N-dimensional array, the maximum values along that axis are returned as an array with N - 1 dimensions:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.max()
4
>>> a.max(0)
array([3, 4])
>>> a.max(1)
array([2, 4])
>>>
```

- min(): Works analogously to max().
- clip(min=, max=): Returns an array in which elements larger than the value assigned to max are set to that value and those smaller than min are set to min.
- conj(): Returns an array with all (complex) elements conjugated.
- trace(): Returns the trace of a two-dimensional array. With optional arguments, works for higher-dimensioned arrays as well, but this is probably not too useful.

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.trace()
5
>>>
```

- sum(axis = None): With no argument, returns the sum of all the elements in the array. With an axis argument, sums along the specified axis, returning an array of N - 1 dimensions for an N-dimensional initial array:

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.sum()
10
>>> a.sum(axis = 0)
array([4, 6])
>>> a.sum(axis = 1)
array([3, 7])
>>>
```

- cumsum(): With no argument, reshape the array to a single dimension and perform a cumulative sum along that dimension. With an optional axis, perform cumulative sums along that axis.

```
>>> a
array([[1, 2],
       [3, 4]])
>>> a.cumsum()
array([ 1,  3,  6, 10])
>>> a.cumsum(axis = 0)
array([[1, 2],
       [4, 6]])
>>> a.cumsum(axis = 1)
array([[1, 3],
       [3, 7]])
>>>
```

- cumprod(): Like cumsum() but computes the cumulative product.
- mean(axis = None): Like sum() except computes the mean.
- var(axis = None): Like sum() except computes the variance.
- std(axis = None): Like sum() except computes the standard deviation.
- prod(axis = None): Like sum() except computes the product.