Lecture6

September 12, 2024

#

Lecture 6: Numpy

The NumPy Reference.

0.0.1 Objective:

This lecture aims to introduce the core concepts of `NumPy`, a fundamental library for numerical computing in Python. We will explore how to create and manipulate multidimensional arrays, perform vectorized operations, and efficiently handle large datasets. By understanding these tools, you'll gain the ability to perform high-performance numerical computations and build the foundation for scientific computing and data analysis using `NumPy`. ****

0.1 What is NumPy?

NumPy (Numerical Python) is the core library for scientific computing in Python, that provide high-performance vector, matrix, and higher-dimensional data structures for Python. It is implemented in `C` and `Fortran`. so when calculations are vectorized (formulated with vectors and matrices), the performance is very good. It provides a high-performance multidimensional array object, and tools for working with these arrays.

This library can be used for different functions in **Linear algebra, Matrix computations, Fourier Transforms**. Alongside Numpy is most suitable for performing basic numerical computations such as *mean, median, range*, etc.

It is an open source module of Python which provides fast mathematical computation on arrays and matrices. Since, arrays and matrices are an essential part of the Machine Learning ecosystem.

`Machine learning` uses vectors, and in that regard, Numpy can be used in all common classes of problem in Machine Learning. Example: **Classification, Regressoin, Clustering** etc.

0.2 NumPy arrays

At the core of the NumPy package, is the `ndarray` object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the *rank* of the array; the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas are built around the NumPy array. This section will present several examples of using NumPy array manipulation to access data and subarrays, and to split, reshape, and join the arrays.

There are a number of ways to initialize new numpy arrays, for example from

- a Python list or tuples;
- using functions that are dedicated to generating numpy arrays, such as arange, linspace, etc.;
- reading data from files.

0.2.1 To use numpy you need to import the module

Import conventions : The recommended convention to import numpy is:

[4]:
```
import numpy as np
```

0.2.2 From lists

For example, to create new vector and matrix arrays from Python lists we can use the `np.array` function.

[5]:
```
# a vector: the argument to the array function is a Python list
v = np.array([1,2,3,4])

v
```

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

[6]:
```
a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
```

[7]:
```
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
(2, 3)
1 2 4
```

0.2.3 Numpy also provides many functions to create arrays:

[8]:
```
import numpy as np


print("------------------------------------------------------------------------")

print("Evenly spaced: \n")

a1 = np.arange(10) # 0 .. n-1  (!). Evenly spaced.
print(a1)

print("\n")

b1 = np.arange(1, 9, 2) # start, end (exclusive), step.
print(b1)


print("========================================================================")

print("Spaced by number of points: \n")


c1 = np.linspace(0, 1, 6)   # start, end, num-points
print(c)


print("\n")

d1 = np.linspace(0, 1, 5, endpoint=False)
print(d1)



print("========================================================================")
print("Common arrays: \n")


a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"
print("\n")

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

print("\n")
```

```
c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"
print("\n")

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"
print("\n")

e = np.random.random((2,2))  # Create an array filled with random values
print("E=",e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

```
------------------------------------------------------------------------
Evenly spaced:

[0 1 2 3 4 5 6 7 8 9]


[1 3 5 7]
========================================================================
Spaced by number of points:
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [8], line 23
     19 print("Spaced by number of points: \n")
     22 c1 = np.linspace(0, 1, 6)   # start, end, num-points
---> 23 print(c)
     26 print("\n")
     28 d1 = np.linspace(0, 1, 5, endpoint=False)

NameError: name 'c' is not defined
```

You can read about other methods of array creation in the documentation.

Exercise: **Creating arrays using functions**

> Experiment with arange, linspace, ones, zeros, eye and diag. Create different kinds of arrays with random numbers. Try setting the seed before creating an array with random

values. Look at the function np.empty. What does it do? When might this be useful?

0.2.4 NumPy Array Attributes

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

[9]:
```
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

[10]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

0.2.5 Basic data types

Different data-types allow us to store data more compactly in memory, but most of the time we simply work with floating point numbers. Note that, in the example above, NumPy auto-detects the data-type from the input.

[11]:
```
a = np.array([1, 2, 3])
print(a.dtype)


b = np.array([1., 2., 3.])
print(b.dtype)
```

```
int64
float64
```

[12]:
```
You can explicitly specify which data-type you want:
```

```
  Cell In [12], line 1
    You can explicitly specify which data-type you want:
        ^
SyntaxError: invalid syntax
```

[13]:
```
c = np.array([1, 2, 3], dtype=float)
print(c.dtype)
```

```
float64
```

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is a more example:

[14]:
```
import numpy as np

x = np.array([1, 2])   # Let numpy choose the datatype
print(x.dtype)         # Prints "int64"

x = np.array([1.0, 2.0])   # Let numpy choose the datatype
print(x.dtype)             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)   # Force a particular datatype
print(x.dtype)                         # Prints "int64"
```

```
int64
float64
int64
```

You can read all about numpy datatypes in the documentation.

[ ]:
```

```

[ ]:
```

```

0.2.6 What is the default data type ?

There are also other types:

0.3 Array Indexing: Accessing Elements/Subarrays

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

[15]:
```
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
```

```
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

print(a)

print('------------------------------------------------------------------------------')

print(b)

print('------------------------------------------------------------------------------')


# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"

print('------------------------------------------------------------------------------')

b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
------------------------------------------------------------------------------
[[2 3]
 [6 7]]
------------------------------------------------------------------------------
2
------------------------------------------------------------------------------
77
```

You can also mix integer indexing with slice indexing. However, doing so will yield an array of lower rank than the original array.

[16]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
```

```
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"

print('------------------------------------------------------------------------------')


print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"

print('------------------------------------------------------------------------------')

print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

```
[5 6 7 8] (4,)
------------------------------------------------------------------------------
[[5 6 7 8]] (1, 4)
[ 2  6 10] (3,)
------------------------------------------------------------------------------
[[ 2]
 [ 6]
 [10]] (3, 1)
```

0.3.1 Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

[17]:
```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

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

Note that for this to work, the size of the initial array must match the size of the reshaped array. Tyr a few examples that you will create.

0.3.2 Fancy indexing

Fancy indexing is the name for when an array or a list is used in-place of an index:

[18]:
```
twenty = (np.arange(4 * 5)).reshape(4, 5)
twenty
```

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

[19]:
```
row_indices = [1, 2, 3]
twenty[row_indices]
```

[19]:
```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

[20]:
```
col_indices = [1, 2, -1] # remember, index -1 means the last element
twenty[row_indices, col_indices]
```

[20]:
```
array([ 6, 12, 19])
```

We can also use index **masks**:

If the index mask is a NumPy array of data type bool, then an element is selected (True) or not

[21]:
```
# 1D array of random integers
# get 10 integers from 0 to 23

num_samples = 10
integers = np.random.randint(23, size=num_samples)
integers
```

[21]:
```
array([15, 20,  3, 12,  4, 20,  8, 14, 15, 20])
```

[22]:
```
# mask has to be of the same shape as the array to be indexed; else IndexError␣
 ↪would be thrown
# mask for indexing alternate elements in the array
row_mask = np.array([True, False, True, False, True, False, True, False, True,␣
 ↪False])

integers[row_mask]
```

[22]:
```
array([15,  3,  4,  8, 15])
```

This feature is very useful to conditionally select elements from an array, using for example comparison operators:

[26]:
```
range_arr = np.arange(0, 10, 0.5)
range_arr
```

[26]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
       6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
```

[27]:
```
mask = (range_arr > 5) * (range_arr < 7.5)
mask
# What is happenning here?
```

[27]:
```
array([False, False, False, False, False, False, False, False, False,
       False, False,  True,  True,  True,  True, False, False, False,
       False, False])
```

[28]:
```
range_arr[mask]
```

[28]:
```
array([5.5, 6. , 6.5, 7. ])
```

0.3.3 Exercise:

Investigate the **Concatenation, Splitting, Copies,repeat, tile, vstack, hstack** functions for numpy arrays.

Investigate Views versus copies in NumPy

0.4 Array math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

[29]:
```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
```

```
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

We instead use the `dot` function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. `dot` is available both as a function in the numpy module and as an instance method of array objects:

[30]:
```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
```

```
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

```
219
219
[29 67]
[29 67]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`

[31]:
```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

You can find the full list of mathematical functions provided by numpy in the documentation.

0.5 Vectorizing functions

As mentioned several times by now, to get good performance we should always try to avoid looping over elements in our vectors and matrices, and instead use vectorized algorithms. The first step in converting a scalar algorithm to a vectorized algorithm is to make sure that the functions we write work with vector inputs.

[35]:
```
def Theta(x):
    """
    scalar implementation of the Heaviside step function.
    """
```

```
    if x >= 0:
        return 1
    else:
        return 0
```

[39]:
```
v1 = np.array([-3,-2,-1,0,1,2,3])

Theta(v1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [39], line 3
      1 v1 = np.array([-3,-2,-1,0,1,2,3])
----> 3 Theta(v1)

Cell In [35], line 5, in Theta(x)
      1 def Theta(x):
      2     """
      3     scalar implementation of the Heaviside step function.
      4     """
----> 5     if x >= 0:
      6         return 1
      7     else:

ValueError: The truth value of an array with more than one element is ambiguous
 ↪Use a.any() or a.all()
```

That didn't work because we didn't write the function Theta so that it can handle a vector input…

To get a vectorized version of Theta we can use the Numpy function vectorize. In many cases it can automatically vectorize a function:

[40]:
```
Theta_vec = np.vectorize(Theta)
Theta_vec(v1)
```

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

On the Other Hand (OTHO), we can also implement the function to accept a vector input from the beginning (requires more effort but might give better performance):

[41]:
```
def Theta(x):
    """
    Vector-aware implementation of the Heaviside step function.
    """
    return 1 * (x >= 0)
```

[42]:
```
Theta(v1)
```

[42]:
```
array([0, 0, 0, 1, 1, 1, 1])
```

[43]:
```
# it even works with scalar input
Theta(-1.2), Theta(2.6)
```

[43]:
```
(0, 1)
```

Numpy provides many more functions for manipulating arrays; you can see the full list in the documentation.

0.6 Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

[44]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector `v`to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

[45]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

[46]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape **(4, 3)** and `v` has shape **(3,)** due to broadcasting; this line works as if `v` actually had shape **(4, 3)**, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

[47]:
```
Here are some applications of broadcasting:
```

```
  Cell In [47], line 1
    Here are some applications of broadcasting:
          ^
SyntaxError: invalid syntax
```

[48]:
```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# giving the following matrix:
# [[2 4 6]
#  [5 7 9]]
print(x + v)
```

```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# Another solution is to reshape w to be a column vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))

# Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

```
[[ 4  5]
 [ 8 10]
 [12 15]]
[[2 4 6]
 [5 7 9]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 5  6  7]
 [ 9 10 11]]
[[ 2  4  6]
 [ 8 10 12]]
```

Broadcasting typically makes your code more concise and faster, so you should strive to use it where possible.