EDUNET FOUNDATION-Class Exercise Notebook

LAB 8 - Implementing Numpy Concepts in Python

Basics of Numpy

This part of the book will cover NumPy in detail. NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. In some ways, NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you.

If you followed the advice outlined in the Preface and installed the Anaconda stack, you already have NumPy installed and ready to go. If you're more the do-it-yourself type, you can go to http://www.numpy.org/ (http://www.numpy.org/) and follow the installation instructions found there. Once you do, you can import NumPy and double-check the version:

Topics covered include:

- How to use Numpy functions in a Jupyter Notebook cell
- Using indexing to explore multi-dimensional Numpy array data
- Numpy data types, broadcasting and booleans

In [1]:
```
import numpy
numpy.__version__
```

Out[1]:
```
'1.23.5'
```

For the pieces of the package discussed here, I'd recommend NumPy version 1.8 or later. By convention, you'll find that most people in the SciPy/PyData world will import NumPy using np as an alias:

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

Throughout this chapter, and indeed the rest of the book, you'll find that this is the way we will import and use NumPy.

Now, we have access to all the functions available in numpy by typing np.name_of_function . For example, the equivalent of 1 + 1 in Python can be done in numpy:

In [3]:
```
np.add(1,1)
```

Out[3]: 2

Arrays

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.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

In [ ]:

In [ ]:

In [8]:

In [9]:
```
import numpy as np

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]"

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"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

In [19]:
```
import numpy as np

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

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

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

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

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

```
[[0. 0.]
 [0. 0.]]
[[1. 1.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.07888374 0.46726271]
 [0.99165976 0.70408017]]
```

In [18]:
```
e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

```
[[0.91704832 0.58112602]
 [0.61840851 0.0665395 ]]
```

In [ ]:
```
```

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

Array indexing

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:

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

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

print(arr[0])
```

```
1
```

Get third and fourth elements from the following array and add them.

In [8]:
```
import numpy as np

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

print(arr[2] + arr[3])
```

```
7
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

In [10]:
```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

```
2nd element on 1st row:  2
```

In [11]:
```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd row: ', arr[1, 4])
```

```
5th element on 2nd row:  10
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

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

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

```
6
```

Example Explained
arr[0, 1, 2] prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays: [[1, 2, 3], [4, 5, 6]] and: [[7, 8, 9], [10, 11, 12]] Since we selected 0, we are left with the first array: [[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains two arrays: [1, 2, 3] and: [4, 5, 6] Since we selected 1, we are left with the second array: [4, 5, 6]

The third number represents the third dimension, which contains three values: 4 5 6 Since we selected 2, we end up with the third value: 6

Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

In [13]:
```
import numpy as np

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

print(arr[1:5])
```

```
[2 3 4 5]
```

In [14]:
```
import numpy as np

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

print(arr[4:])
```

```
[5 6 7]
```

In [15]:
```
import numpy as np

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

print(arr[:4])
```

```
[1 2 3 4]
```

Negative Slicing

Use the minus operator to refer to an index from the end:

In [17]:
```
import numpy as np

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

print(arr[-3:-1])
```

```
[5 6]
```

STEP

Use the step value to determine the step of the slicing:

In [18]:
```
import numpy as np

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

print(arr[1:5:2])
```

```
[2 4]
```

In [19]:
```
import numpy as np

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

print(arr[::2])
```

```
[1 3 5 7]
```

From the second element, slice elements from index 1 to index 4 (not included):

In [20]:
```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

```
[7 8 9]
```

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

The NumPy array object has a property called dtype that returns the data type of the array:

Get the data type of an array object:

In [22]:
```
import numpy as np

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

print(arr.dtype)
```

```
int32
```

In [23]:
```
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)
```

```
<U6
```

Creating Arrays With a Defined Data Type

We use the `array()` function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements:

In [24]:
```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4']
|S1
```

For i, u, f, S and U we can define size as well.

In [25]:
```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

```
[1 2 3 4]
int32
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.

The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like `'f'` for float, `'i'` for integer etc. or you can use the data type directly like float for float and int for integer.

In [26]:
```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

Change data type from float to integer by using int as parameter value:

In [27]:
```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype(int)

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

Change data type from integer to boolean:

In [28]:
```
import numpy as np

arr = np.array([1, 0, 3])

newarr = arr.astype(bool)

print(newarr)
print(newarr.dtype)
```

```
[ True False  True]
bool
```

Get the Shape of an Array

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Print the shape of a 2-D array:

In [30]:
```
import numpy as np

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

print(arr.shape)
```

```
(2, 4)
```

The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

In [31]:
```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

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

Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1-D to 2-D

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

In [33]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

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

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

In [34]:
```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.

We can use reshape(-1) to do this.

Convert the array into a 1D array:

In [36]:
```
import numpy as np

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

newarr = arr.reshape(-1)

print(newarr)
```

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

Iterating Arrays

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

Iterate on the elements of the following 1-D array:

In [37]:
```
import numpy as np

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

for x in arr:
  print(x)
```

```
1
2
3
```

In a 2-D array it will go through all the rows.

Iterate on the elements of the following 2-D array:

In [38]:
```
import numpy as np

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

for x in arr:
  print(x)
```

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

If we iterate on a n-D array it will go through n-1th dimension one by one.

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Iterate on each scalar element of the 2-D array:

In [39]:
```
import numpy as np

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

for x in arr:
  for y in x:
    print(y)
```

```
1
2
3
4
5
6
```

Iterating 3-D Arrays

In a 3-D array it will go through all the 2-D arrays.

In [40]:
```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
  for y in x:
    for z in y:
      print(z)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

Iterating Arrays Using nditer()

The function `nditer()` is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

Iterate through the following 3-D array:

In [41]:
```
import numpy as np

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

for x in np.nditer(arr):
  print(x)
```

```
1
2
3
4
5
6
7
8
```

What is a Random Number?

Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

Random numbers generated through a generation algorithm are called pseudo random.

Can we make truly random numbers?

Yes. In order to generate a truly random number on our computers we need to get the random data from some outside source. This outside source is generally our keystrokes, mouse movements, data on network etc.

We do not need truly random numbers, unless it is related to security (e.g. encryption keys) or the basis of application is the randomness (e.g. Digital roulette wheels).

In this tutorial we will be using pseudo random numbers.

Generate Random Number

NumPy offers the random module to work with random numbers.

Generate a random integer from 0 to 100:

In [42]:
```
from numpy import random

x = random.randint(100)

print(x)
```

```
79
```

Generate Random Float

The random module's rand() method returns a random float between 0 and 1.

Generate a random float from 0 to 1:

In [44]:
```
from numpy import random

x = random.rand()

print(x)
```

```
0.10886693671938563
```

Generate Random Array

In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

Integers

The randint() method takes a size parameter where you can specify the shape of an array.

Generate a 1-D array containing 5 random integers from 0 to 100:

In [45]:
```
from numpy import random

x=random.randint(100, size=(5))

print(x)
```

```
[71  8 50 43 68]
```

Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:

In [46]:
```
from numpy import random

x = random.randint(100, size=(3, 5))

print(x)
```

```
[[ 9 23 29 58 54]
 [67 44 42 52 25]
 [20  0 10 18 66]]
```

Floats The rand() method also allows you to specify the shape of the array.

Generate a 1-D array containing 5 random floats:

In [47]:
```
from numpy import random

x = random.rand(5)

print(x)
```

```
[0.79310998 0.44835227 0.05494187 0.66941683 0.21945707]
```

Generate Random Number From Array

The `choice()` method allows you to generate a random value based on an array of values.

The `choice()` method takes an array as a parameter and randomly returns one of the values.

Return one of the values in an array:

In [48]:
```
from numpy import random

x = random.choice([3, 5, 7, 9])

print(x)
```

```
7
```

The `choice()` method also allows you to return an array of values.

Add a `size` parameter to specify the shape of the array.

In [49]:
```
from numpy import random

x = random.choice([3, 5, 7, 9], size=(3, 5))

print(x)
```

```
[[3 7 5 7 9]
 [7 7 9 9 3]
 [5 7 3 3 9]]
```

In [ ]:
```

```