Day 4: Programming with Python

NumPy

NumPy (https://numpy.org/) is the fundamental package for numeric computing with Python. It is a Python library that provides a multidimensional array object, and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, basic linear algebra, basic statistical operations, random simulation and much more.

Why use NumPy?

Numpy data structures perform better in:

`Size` - Numpy data structures take up less space.

`Performance` - they have a need for speed and are faster than lists.

`Functionality` - NumPy has optimized functions such as linear algebra operations built in.

NumPy arrays are faster and more compact than Python lists. An array consumes less memory and is convenient to use. NumPy uses much less memory to store data and it provides a mechanism of specifying the data types. This allows the code to be optimized even further. Overall a task executed in Numpy is around 5 to 100 times faster than the standard python list.

0. Install and import NumPy

For this unit, we will use the NumPy library.

The most common abbreviation for NumPy is `np`.

In [1]:
```
import numpy as np
```

1. NumPy arrays

Key points:

- **np.array()** - Creates an numpy array
- **np.size** - Returns the number of elements
- **np.shape** - Returns the number of rows and columns
- **np.dtype** - Returns the type of elements
- **np.ndim** - Returns the dimensions of the array
- **np.reshape()** - Converts an array to a new one of the specified dimensions

The NumPy library is based on one main object: `ndarray` (which stands for N-dimensional array). This object is a multidimensional homogeneous array with a predetermined number

of items: homogeneous because virtually all the items in it are of the same type and the same size.

We can create a NumPy `ndarray` object by using the `array()` function. There are several ways to create NumPy arrays, a common way is to pass a list as an argument.

In [2]:
```
A = np.array([1, 2, 3, 4, 5, 6])
print(A)
```

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

We can also create a multi-dimensional array. One way to do that is to pass a list of lists.

For example, we can pass a list with floats, the [1.1, 2.9] and [3.1, 4.2]

In [3]:
```
B = np.array([[1.1, 2.9], [3.1, 4.2]])
print(B)
```

```
[[1.1 2.9]
 [3.1 4.2]]
```

We can check the attributes of the A and B arrays with `ndim`, `dtype`, `size` and `shape` attributes.

In [4]:
```
print("Attributes of A")
print("ndim: ", A.ndim)
print("dtype: ", A.dtype)
print("size: ", A.size)
print("shape: ", A.shape)
print("Attributes of B")
print("ndim: ", B.ndim)
print("dtype: ", B.dtype)
print("size: ", B.size)
print("shape: ", B.shape)
```

```
Attributes of A
ndim:  1
dtype:  int64
size:  6
shape:  (6,)
Attributes of B
ndim:  2
dtype:  float64
size:  4
shape:  (2, 2)
```

Reshape an array

We can change the shape of an array using the `reshape()` method:

In [5]:
```
print('A =', A)
print()

B = np.reshape(A,(3, 2))
#B = A.reshape(2, 3)

print('B =', B)
```

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

B = [[1 2]
```

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

In [6]:
```
print("shape of B:", B.shape)
```

```
shape of B: (3, 2)
```

2. Creation routines

Key points:

- **np.zeros()** - Returns a new array of given shape, filled with zeros
- **np.random.random()** - Returns a new array of filled with random numbers in (0, 1)
- **np.random.randint(N, M)** - Return a new array filled with random integers from N to M
- **np.arange()** - Returns evenly spaced values within a given interval of a specified step

In some cases we know the shape of the array we want to create, but we still do not have data. In this case we can use numpy functions that fill the arrays with filler values.

We can create an array that contains 0 with the `np.zeros` method. This method takes as argument the shape of the array.

Let's create an array of dimensions (2,4) and fill it first with zeros and then with ones.

In [7]:
```
shape = (2, 4)
zeros_array = np.zeros(shape)
print(zeros_array)
```

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

One very useful method is the the `random()` function with which we obtain arrays filled with random values. The function generates samples from the uniform distribution on [0, 1)

In [8]:
```
np.random.random(size = (2, 4))
```

Out[8]:
```
array([[0.98666502, 0.36175769, 0.23567831, 0.6344979 ],
       [0.46453554, 0.03976571, 0.00822334, 0.37481695]])
```

The `randit()` function returns random integers.

In [9]:
```
np.random.randint(100, size = (2, 4))
```

Out[9]:
```
array([[15, 10, 11, 71],
       [40, 11, 49,  8]])
```

If we want to create an array that contains sequence of numbers, we can use the `arange()` method that returns evenly spaced values within a given interval. Values are generated within the half-open interval [start, stop) (in other words, the interval including start but excluding stop). Start and stop are the first two arguments. The third argument is the interval with a default value of 1.

Let's create an array A that contains all the even numbers from 0 to 20

In [10]:
```
A = np.arange(0, 20, 2)
print('A =', A)
```

```
A = [ 0  2  4  6  8 10 12 14 16 18]
```

3. Access matrix elements, rows and columns

Indexing

Similar like lists, we can access matrix elements using index. Let's start with a one-dimensional NumPy array and print the first element.

In [11]:
```
A = np.arange(2, 21, 2)

print("A[0] =", A[0])
print("A[0:2] =", A[0:2])
```

```
A[0] = 2
A[0:2] = [2 4]
```

4. Sort arrays

Key points:

- **np.sort()** - Returns a sorted copy of an array

The method we can use to sort an array is the `np.sort()` .

Let's try to sort the array [2, 1, 5, 3, 7, 4, 6, 8]

In [12]:
```
A = np.array([2, 1, 5, 3, 7, 4, 6, 8])
np.sort(A)
```

Out[12]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Let's create a 2-d array and see the result of sort.

In [13]:
```
A = np.array([[1, 0, 5], [3, 4, 1]])
np.sort(A)
```

Out[13]:
```
array([[0, 1, 5],
       [1, 3, 4]])
```

5. Array operations

With NumPy we can perform matices operations, like addition, subtraction, square and matrix multiplication.

Arithmetic operators

The simplest arithmetic operations that we can perform on arrays are adding, sutracting, multiplying and dividing an array by a scalar. Let's create a list and then add each element by 3.

In [14]:
```
listNumbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
listNumbersNew = []
for l in listNumbers:
```

```
    listNumbersNew.append(l + 3)

print(listNumbersNew)
```

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

Let's create an array A and then add each element by 3.

In [15]:

```
A = np.arange(10)
print("A =",A)
print()

A = A + 3
print("A =", A)
print()
```

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

A = [ 3  4  5  6  7  8  9 10 11 12]
```

The arithmetic operators can also be applied between two arrays. These operations are element-wise, that is, the operators are applied only between corresponding elements.

Let's see add, subtract and multiple two arrays.

In [16]:

```
A = np.array([[1, 2], [3, 4]])
print("A =",A)
print()

B = np.array([[5, 6], [7, 8]])
print("B =", B)
print()

C = A + B
print("C =", C)
```

```
A = [[1 2]
 [3 4]]

B = [[5 6]
 [7 8]]

C = [[ 6  8]
 [10 12]]
```

In [17]:

```
C = A - B
print("A - B =", C)
```

```
A - B = [[-4 -4]
 [-4 -4]]
```

In [18]:

```
C = A * B
print("A * B =", C)
```

```
A * B = [[ 5 12]
 [21 32]]
```

Universal functions

NumPy provides mathematical functions such as `sqrt()` , and `exp()` . Within NumPy, these functions operate elementwise on an array, producing an array as output.

In [19]:
```
A = np.arange(0, 5, 2)
print("A =", A)
print()

print("e^A =", np.exp(A))
print()

print("square-root of A =", np.sqrt(A))
print()
```

```
A = [0 2 4]

e^A = [ 1.          7.3890561  54.59815003]

square-root of A = [0.         1.41421356 2.        ]
```

Aggregate functions

Aggregate functions perform an operation on an array and produce a single result. Let's see some examples.

In [20]:
```
A = np.array([[3, 6, 7], [5, -3, 0]])
print("A =", A)
print()

print ("min =", A.min())
print ("max =", A.max())
```

```
A = [[ 3  6  7]
 [ 5 -3  0]]

min = -3
max = 7
```

With the `ptp()` (peak-to-peak) we can return the range between maximim and minimum

In [21]:
```
print(A.ptp())
```

```
10
```

Other statistics include mean, standard deviation and variance calculated with `mean()`, `std()` and `var()`

In [22]:
```
print("Mean: ", A.mean())
print("Stand. Dev.: ", A.std())
print("Variance: ", A.var())
```

```
Mean:  3.0
Stand. Dev.:  3.5118845842842465
Variance:  12.333333333333334
```

6. Save and load NumPy objects

It is more efficient to save our arrays to a file and load them back without having to re-run the code. This is especially convenient when we have to calculate features from a huge amount of data that takes some time.

The `.npy` stores data, shape, dtype, and other information required to reconstruct the `ndarray` in a way that allows the array to be correctly retrieved, even when the file is on another machine with different architecture.

If we want to store a single `ndarray` object, we can store it as a `.npy` file using `np.save()`.

Let's create a simple array A and save it to a file in the export folder. The file will be automatically saved as type of `.npy`

In [23]:
```
A = np.array([1, 2, 3, 4, 5, 6])
np.save('../data/numpyFile', A)
```

We can use `np.load()` to reconstruct the array.

In [24]:
```
b = np.load('../data/numpyFile.npy')
print(type(b))
```

```
<class 'numpy.ndarray'>
```

7. Pandas and NumPy and Matplotlib all together

Let's finally see how we can put everything together. Let's load the supermarket_sales and print Quantity

In [25]:
```
import pandas as pd

df = pd.read_csv("../data/supermarket_sales.csv")
df.head()

print(df['Quantity'])
```

```
0       7
1       5
2       7
3       8
4       7
       ..
995     1
996    10
997     1
998     1
999     7
Name: Quantity, Length: 1000, dtype: int64
```

Convert Quantity to numpy array

In [26]:
```
npQuantity = (df['Quantity']).to_numpy()
df['newQuantity'] = npQuantity + 5
print(df[['Quantity', 'newQuantity']])
```

```
     Quantity  newQuantity
0           7           12
1           5           10
2           7           12
3           8           13
4           7           12
..        ...          ...
995         1            6
996        10           15
```

```
997         1            6
998         1            6
999         7           12

[1000 rows x 2 columns]
```

Exercises

1. Create an array A that contains all the odd numbers from 20 to 50

In [27]:
```
print(np.arange(21, 50, 2))
```

```
[21 23 25 27 29 31 33 35 37 39 41 43 45 47 49]
```

2. Try to reshape A to a new 2d array with 5 rows and 2 columns. Name the new array B. How can you fix the error?

In [28]:
```
B = np.reshape(A,(5,2))

print(B)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-28-c2234cdfaebe> in <module>
----> 1 B = np.reshape(A,(5,2))
      2
      3 print(B)

<__array_function__ internals> in reshape(*args, **kwargs)

~/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py in resha
pe(a, newshape, order)
    297            [5, 6]])
    298     """
--> 299     return _wrapfunc(a, 'reshape', newshape, order=order)
    300
    301

~/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py in _wrap
func(obj, method, *args, **kwds)
     56
     57     try:
---> 58         return bound(*args, **kwds)
     59     except TypeError:
     60         # A TypeError occurs if the object does have such a method in
 its

ValueError: cannot reshape array of size 6 into shape (5,2)
```

3. Reshape A to a new 2d array with 8 rows and 2 columns. Name the new array B.

In [29]:
```
B = np.reshape(A,(8,2))

print(B)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-29-39e332c6cede> in <module>
```

```
----> 1 B = np.reshape(A,(8,2))
      2
      3 print(B)

<__array_function__ internals> in reshape(*args, **kwargs)

~/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py in resha
pe(a, newshape, order)
    297            [5, 6]])
    298     """
--> 299     return _wrapfunc(a, 'reshape', newshape, order=order)
    300
    301

~/opt/anaconda3/lib/python3.8/site-packages/numpy/core/fromnumeric.py in _wrap
func(obj, method, *args, **kwds)
     56
     57     try:
---> 58         return bound(*args, **kwds)
     59     except TypeError:
     60         # A TypeError occurs if the object does have such a method in
 its

ValueError: cannot reshape array of size 6 into shape (8,2)
```

4. Create a random array (4, 4) with integer values from 100-120

In [ ]:
```
A = np.random.randint(100, 120, (4,4))
```

1. Create two random arrays (4, 4) with integers from 0-50 and then first add and then subtract them

In [ ]:
```
A = np.random.randint(0, 50, (4,4))
B = np.random.randint(0, 50, (4,4))

print(A + B)
print(A - B)
```

6. Create the A = [[3, 6, np.NaN, 7], [5, -3, 0, np.NaN]] that contains two NA values as well. Print the max and the mean of the whole array. What do you notice? Find a function to calculate the max and mean excluding the NaN values

In [ ]:
```
A = np.array([[3, 6, np.NaN, 7], [5, -3, 0, np.NaN]])
```

In [ ]:
```
print(np.amax(A))
print(np.mean(A))
```

In [ ]:
```
print(np.nanmax(A))
print(np.nanmean(A))
```

7. NumPy provides a lot of functions that can be applied on the DataFrame.

a. Read the supermarket_sales.csv file

b. Normalise the Unit price with values from 0 to 1. First turn the `df['Unit price']` to numpy and then create a new column `df['normalisedPrice']` for the normalised values

c. Create a new column `df['NewTotal']` that will contain the value of total after you subtract 2 from it

a. First we read the file

In [ ]:
```
import pandas as pd
df = pd.read_csv("../data/supermarket_sales.csv")
df.head()
```

b. Let's normalise the Unit price with values from 0 to 1.

In [ ]:
```
rating = df['Unit price'].to_numpy()
df['normalisedPrice'] = (rating - rating.min())/rating.ptp()
df['normalisedPrice'][0:5]
```

c. Create a new column `df['NewTotal']` that will contain the value of total after you subtract 2 from it

In [30]:
```
print(df['Total'].head())
df['NewTotal'] = df['Total'] - 2
print(df['NewTotal'].head())
```

```
0    548.9715
1     80.2200
2    340.5255
3    489.0480
4    634.3785
Name: Total, dtype: float64
0    546.9715
1     78.2200
2    338.5255
3    487.0480
4    632.3785
Name: NewTotal, dtype: float64
```

8. Extra Bonus

Create two lists and two numpy arrays of the size 10,000,000 containing all the numbers from 0 to 10,000,000. You can use range for the list and arange for the numpy arrays. Once you created the two lists, multipy them and print the execution time. Do the same for the numpy arrays and compare the diffence in performance.

In [31]:
```
from datetime import datetime

size = 10000000

# the two lists
listA = range(size)
listB = range(size)

# the two arrays
arA = np.arange(size)
arB = np.arange(size)
```

```
# time before the multiplication of Python lists
sTime = datetime.now()
listC = []

# multiplying  elements of the lists
for i in range(0, len(listA)):
    listC.append(listA[i] * listB[i])


# calculating execution time
print("Lists needed ",
      (datetime.now() - sTime),
      "seconds for the multiplication")

# time before the multiplication of Numpy arrays
sTime = datetime.now()

# multiplying  elements of the Numpy arrays
arC = arA * arB

# calculating execution time
print("NumPy Arrays needed ",
      (datetime.now() - sTime),
      "seconds for the multiplication")
```

```
Lists needed  0:00:03.840816 seconds for the multiplication
NumPy Arrays needed  0:00:00.050821 seconds for the multiplication
```

In [ ]: