4. Numerical Python (NumPy)

NumPy is a Python library created in 2005 that performs numerical calculations. It is generally used for working with arrays.
NumPy also includes a wide range of mathematical functions, such as linear algebra, Fourier transforms, and random number generation, which can be applied to arrays.

What is NumPy Used for?
NumPy is an important library generally used for:

- Machine Learning
- Data Science
- Image and Signal Processing
- Scientific Computing
- Quantum Computing

Why Use NumPy?
Some of the major reasons why we should use NumPy are:

1. Faster Execution
In Python, we use lists to work with arrays. But when it comes to large array operations, Python lists are not optimized enough.
Numpy arrays are optimized for complex mathematical and statistical operations. Operations on NumPy are up to 50x faster than iterating over native Python lists using loops.

Here're some of the reasons why NumPy is so fast:

- Uses specialized data structures called numpy arrays.
- Created using high-performance languages like C and C++.

2. Used with Various Libraries
NumPy is heavily used with various libraries like Pandas, Scipy, scikit-learn, etc.

4.1 Import NumPy in Python
We can import NumPy in Python using the *import* statement.

- If we import NumPy without an alias using `import numpy`, we can create an array using the `numpy.array()` function.

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

- Using an alias np is a common convention among Python programmers, as it makes it easier and quicker to refer to the NumPy library in your code.

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

4.2 Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

1. **0-D Arrays**
   0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np
arr = np.array(42)
print(arr)
```

2. **1-D Arrays**
   An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
   These are the most common and basic arrays.

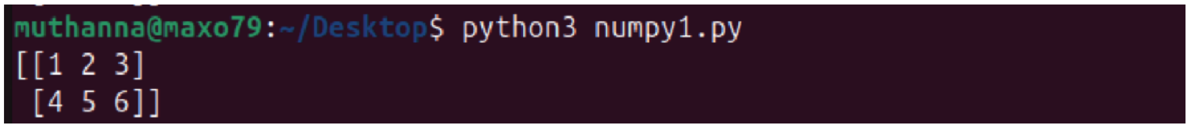
```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

3. **2-D Arrays**
   An array that has 1-D arrays as its elements is called a 2-D array.
   These are often used to represent matrix or 2nd order tensors.

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

Or

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

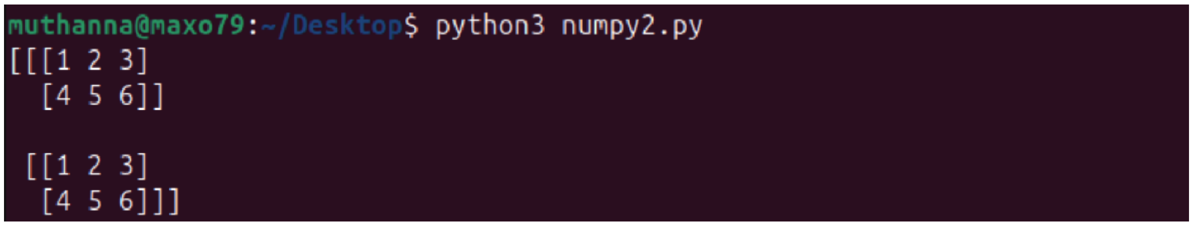


```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[[1 2 3]
 [4 5 6]]
```

4. **3-D Arrays**
   An array that has 2-D arrays (matrices) as its elements is called 3-D array.

These are often used to represent a 3rd order tensor.

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



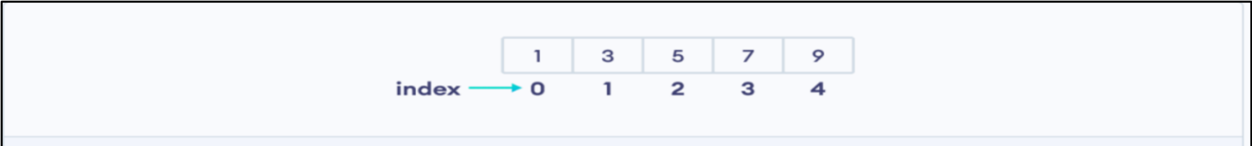
```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
[[[1 2 3]
  [4 5 6]]

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

4.3 Access Array Elements

Array indexing is the same as accessing an array element. You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

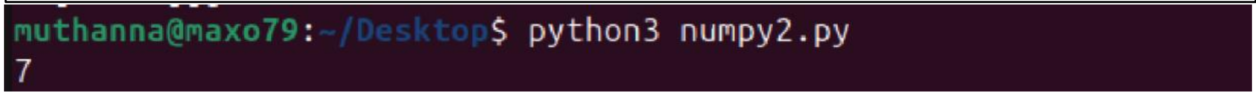


Array Indexing in NumPy

Example #1

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

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

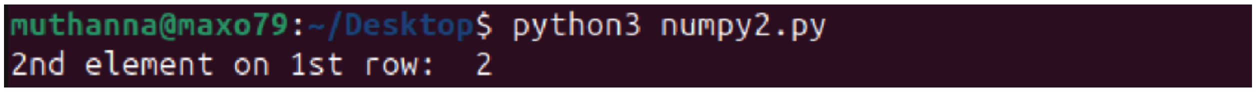


```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
7
```

Example #2

Access the element on the first row, second column:

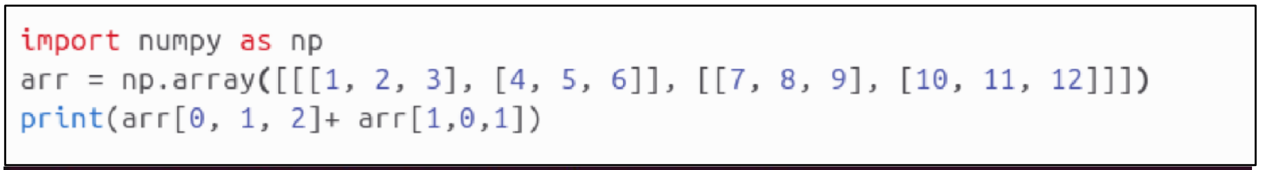
```python
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```



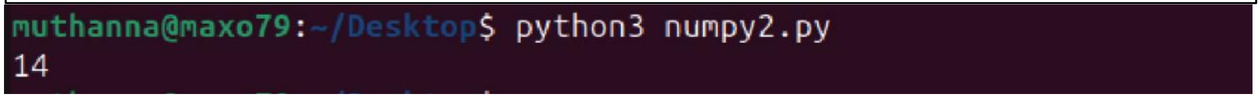
```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
2nd element on 1st row:  2
```

Example #3

Get the third element of the second array of the first array and add to the second element of the first array of the second array



```python
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2]+ arr[1,0,1])
```



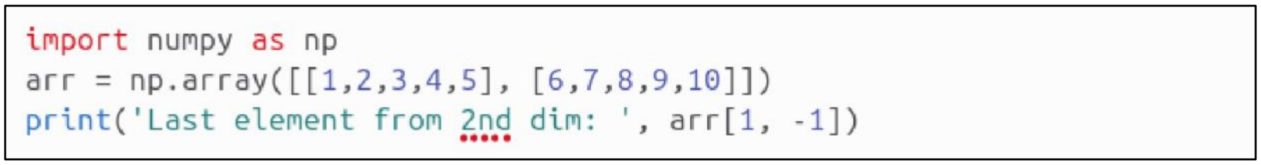
```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
14
```

Negative Indexing

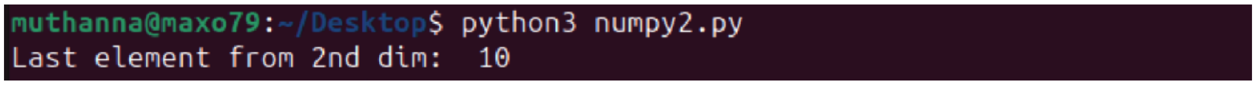
Use negative indexing to access an array from the end.

Example #4

Print the last element from the 2nd dim:



```python
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```



```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
Last element from 2nd dim:  10
```

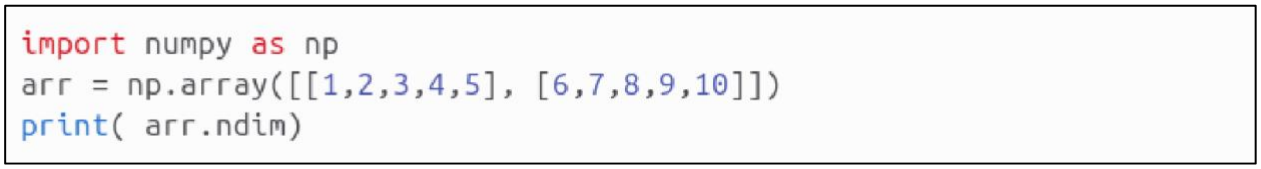
4.4 NumPy Array Attributes

In NumPy, attributes are properties of NumPy arrays that provide information about the array's shape, size, data type, dimension, and so on.
For example, to get the dimension of an array, we can use the ndim attribute.
There are numerous attributes available in NumPy, which we'll learn below.\

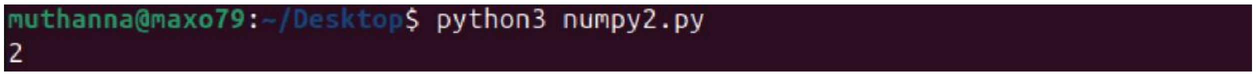
Attributes	Description
`ndim`	returns number of dimension of the array
`size`	returns number of elements in the array
`dtype`	returns data type of elements in the array
`shape`	returns the size of the array in each dimension.
`itemsize`	returns the size (in bytes) of each elements in the array
`data`	returns the buffer containing actual elements of the array in memory

Example #1

The ndim attribute returns the number of dimensions in the numpy array.

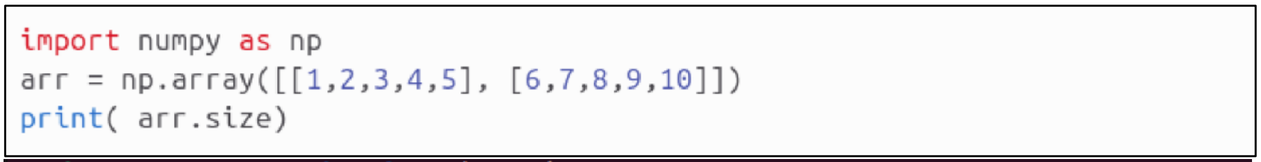


```python
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print( arr.ndim)
```

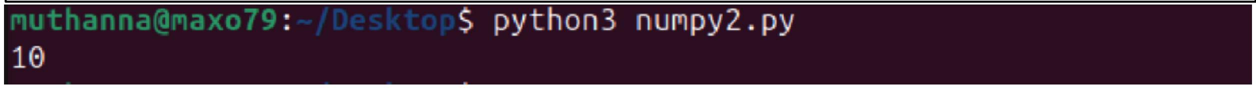


```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
2
```

Example #2
The size attribute returns the total number of elements in the given array.

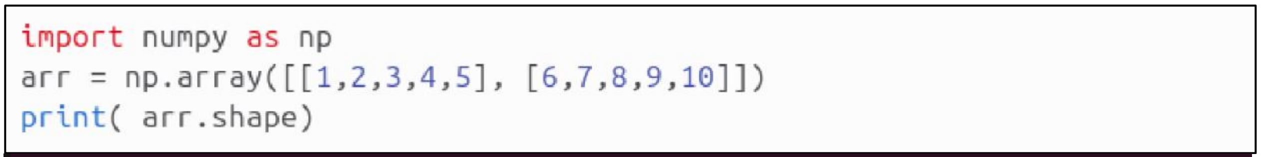


```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print( arr.size)
```

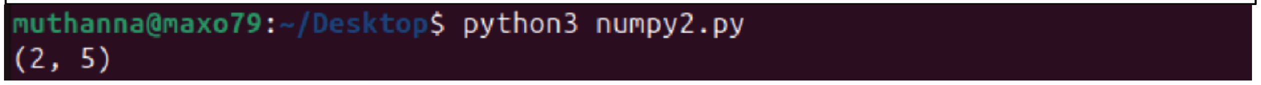


```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
10
```

Example #3
The shape attribute returns a tuple of integers that gives the size of the array in each dimension.



```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print( arr.shape)
```



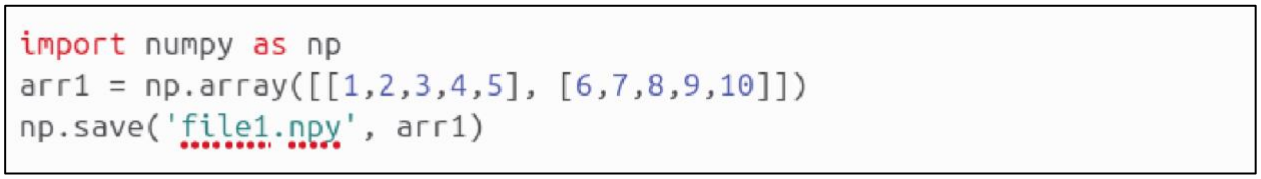
```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
(2, 5)
```

4.5 NumPy Input Output

NumPy offers input/output (I/O) functions for loading and saving data to and from files.
Input/output functions support a variety of file formats, including binary and text formats.
The binary format is designed for efficient storage and retrieval of large arrays.
The text format is more human-readable and can be easily edited in a text editor.
Here are some of the commonly used NumPy Input/Output functions:

Function	Description
`save()`	saves an array to a binary file in the NumPy .npy format.
`load()`	loads data from a binary file in the NumPy .npy format
`savetxt()`	saves an array to a text file in a specific format
`loadtxt()`	loads data from a text file.

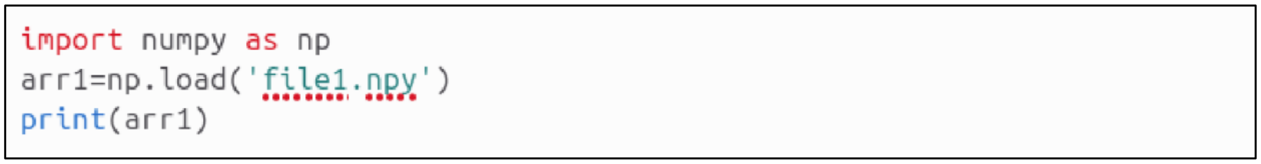
Example #1
The save() function is used to save an array to a binary file in the NumPy .npy format.



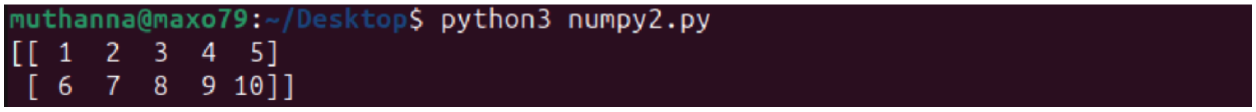
```
import numpy as np
arr1 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
np.save('file1.npy', arr1)
```

Here, we saved the NumPy array named arr1 to the binary file named file1.npy in our current directory.

Example #2
In the previous example, we saved an array to a binary file. Now we'll load that saved file using the load() function.

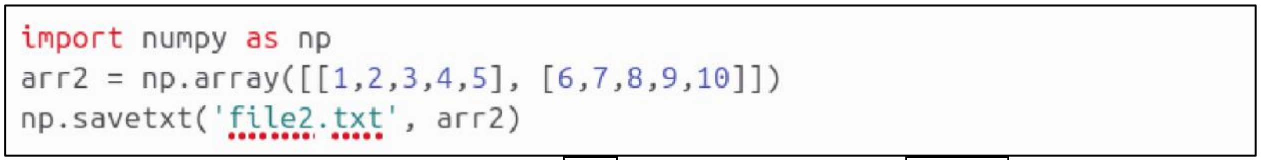


```
import numpy as np
arr1=np.load('file1.npy')
print(arr1)
```



```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
```

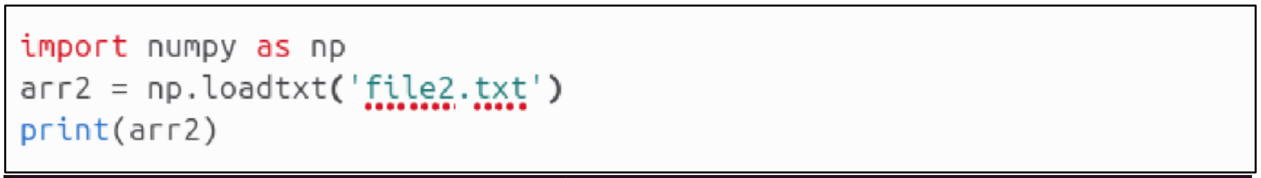
Example #3
In NumPy, we use the savetxt() function to save an array to a text file.



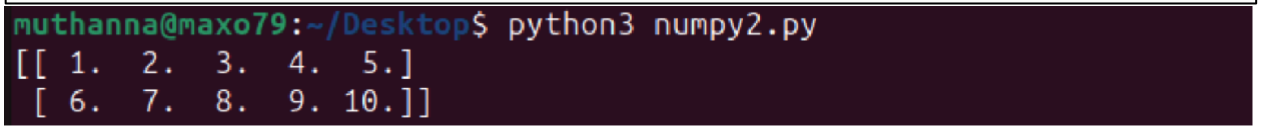
```
import numpy as np
arr2 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
np.savetxt('file2.txt', arr2)
```

The code above will save the NumPy array arr2 to the text file named file2.txt in our current directory.

Example #4
We use the loadtxt() function to load the saved txt file.
Let's see an example to load the file2.txt file that we saved earlier.



```
import numpy as np
arr2 = np.loadtxt('file2.txt')
print(arr2)
```



```
muthanna@maxo79:~/Desktop$ python3 numpy2.py
[[ 1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10.]]
```

4.6 NumPy Arithmetic Array Operations

NumPy provides a wide range of operations that can perform on arrays, including arithmetic operations.
NumPy's arithmetic operations are widely used due to their ability to perform simple and efficient calculations on arrays.
In this tutorial, we will explore some commonly used arithmetic operations in NumPy and learn how to use them to manipulate arrays.

List of Arithmetic Operations

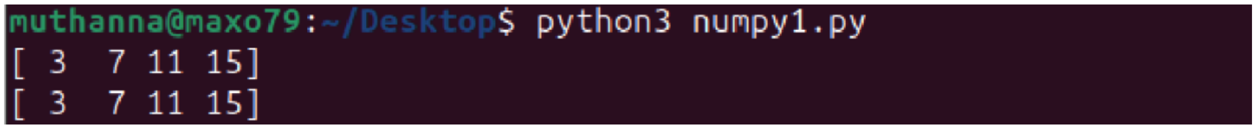
Here's a list of various arithmetic operations along with their associated operators and built-in functions:

lement-wise Operation	Operator	Function
Addition	`+`	`add()`
Subtraction	`-`	`subtract()`
Multiplication	`*`	`multiply()`
Division	`/`	`divide()`
Exponentiation	`**`	`power()`
Modulus	`%`	`mod()`

1. NumPy Array Element- Addition

We can use the both `+` operator and the built-in function `add()` to perform element- addition between two NumPy arrays. For example,

```
import numpy as np
arr1 = np.array([1, 3, 5, 7])
arr2= np.array([2, 4, 6, 8])
x=arr1+arr2
print(x)
y=np.add(arr1,arr2)
print(y)
```

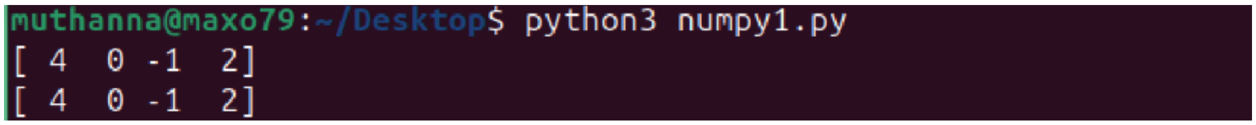


```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[ 3  7 11 15]
[ 3  7 11 15]
```

2. NumPy Array Element- Subtraction

In NumPy, we can either use the `-` operator or the `subtract()` function to perform element- subtraction between two NumPy arrays. For example,

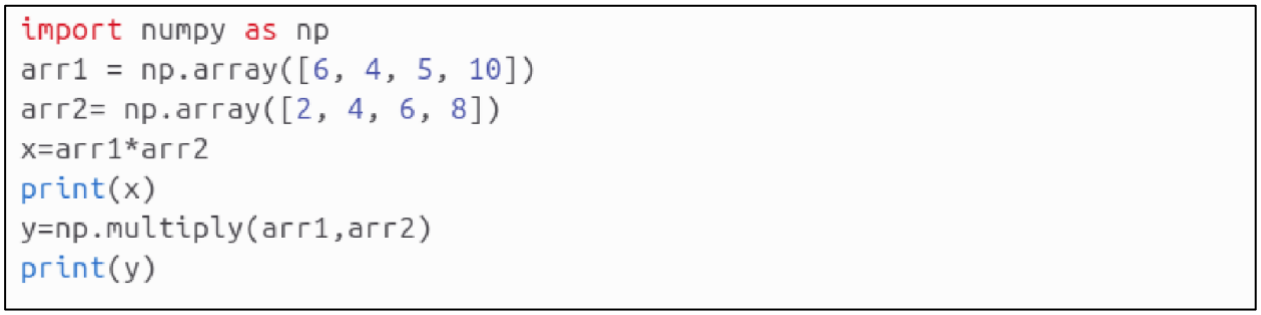
```
import numpy as np
arr1 = np.array([6, 4, 5, 10])
arr2= np.array([2, 4, 6, 8])
x=arr1-arr2
print(x)
y=np.subtract(arr1,arr2)
print(y)
```



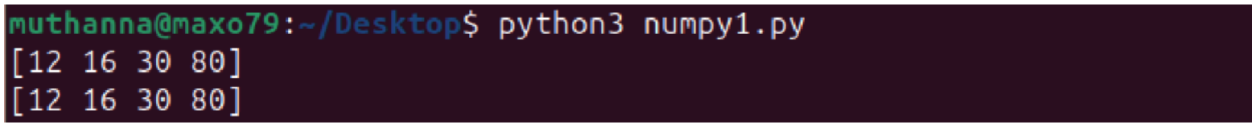
```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[ 4  0 -1  2]
[ 4  0 -1  2]
```

3. NumPy Array Element-Multiplication

For element- multiplication, we can use the $\boxed{*}$ operator or the $\boxed{\text{multiply()}}$ function. For example,



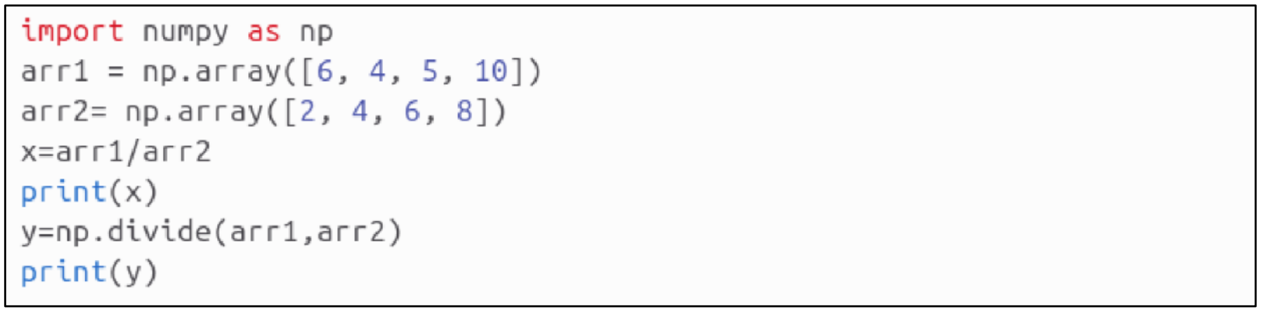
```python
import numpy as np
arr1 = np.array([6, 4, 5, 10])
arr2= np.array([2, 4, 6, 8])
x=arr1*arr2
print(x)
y=np.multiply(arr1,arr2)
print(y)
```



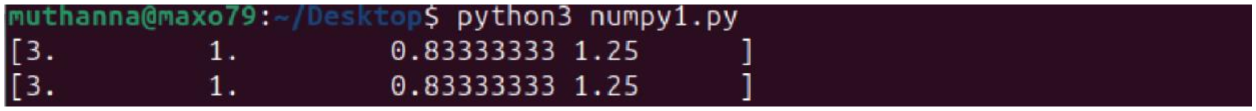
```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[12 16 30 80]
[12 16 30 80]
```

4. NumPy Array Element-Division

We can use either the $\boxed{/}$ operator or the $\boxed{\text{divide()}}$ function to perform element- division between two numpy arrays. For example,



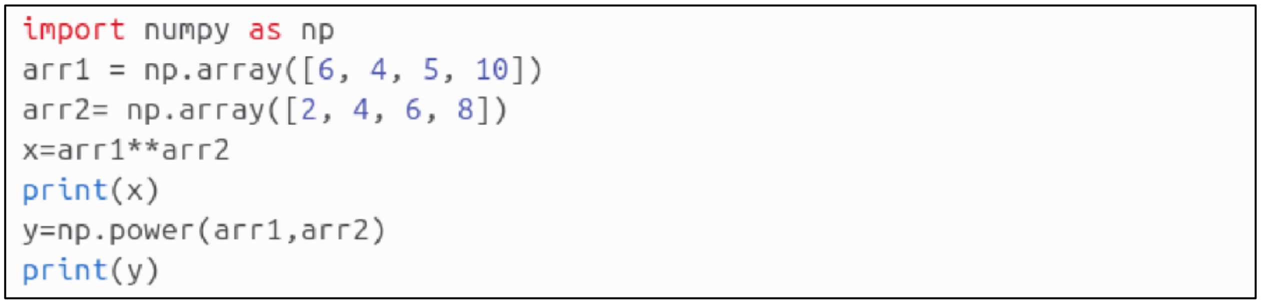
```python
import numpy as np
arr1 = np.array([6, 4, 5, 10])
arr2= np.array([2, 4, 6, 8])
x=arr1/arr2
print(x)
y=np.divide(arr1,arr2)
print(y)
```



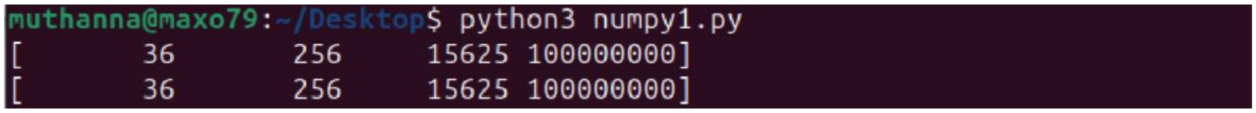
```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[3.         1.         0.83333333 1.25      ]
[3.         1.         0.83333333 1.25      ]
```

5. NumPy Array Element-Exponentiation

Array exponentiation refers to raising each element of an array to a given power.
In NumPy, we can use either the $\boxed{**}$ operator or the $\boxed{\text{power()}}$ function to perform the element-exponentiation operation. For example,



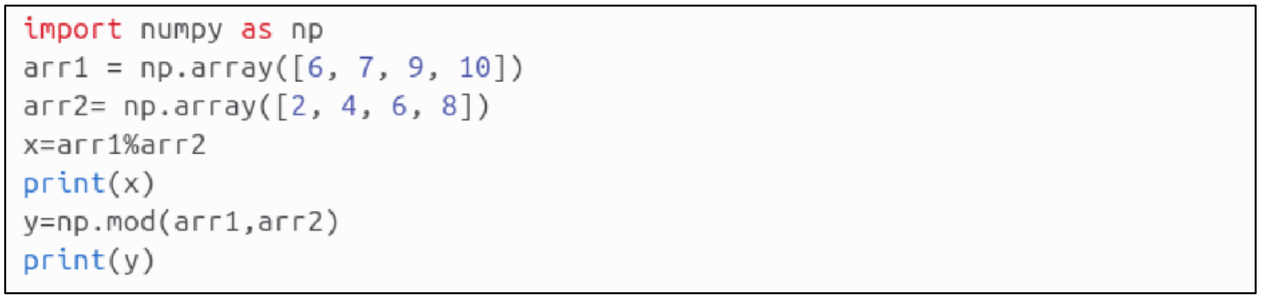
```python
import numpy as np
arr1 = np.array([6, 4, 5, 10])
arr2= np.array([2, 4, 6, 8])
x=arr1**arr2
print(x)
y=np.power(arr1,arr2)
print(y)
```



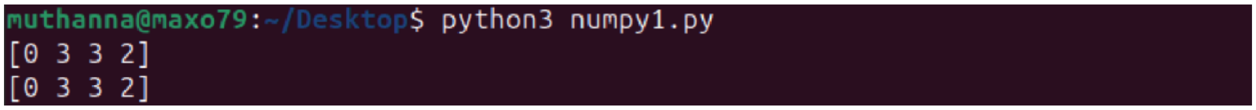
```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[        36        256      15625 100000000]
[        36        256      15625 100000000]
```

6. NumPy Array Element- Modulus

We can perform a modulus operation in NumPy arrays using the % operator or the mod() function. This operation calculates the remainder of element-division between two arrays. Let's see an example.



```
import numpy as np
arr1 = np.array([6, 7, 9, 10])
arr2= np.array([2, 4, 6, 8])
x=arr1%arr2
print(x)
y=np.mod(arr1,arr2)
print(y)
```



```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[0 3 3 2]
[0 3 3 2]
```

4.7 Numpy Comparison/Logical Operations

NumPy provides several comparison and logical operations that can be performed on NumPy arrays.

NumPy's comparison operators allow for element- comparison of two arrays.

Similarly, logical operators perform boolean algebra, which is a branch of algebra that deals with True and False statements.

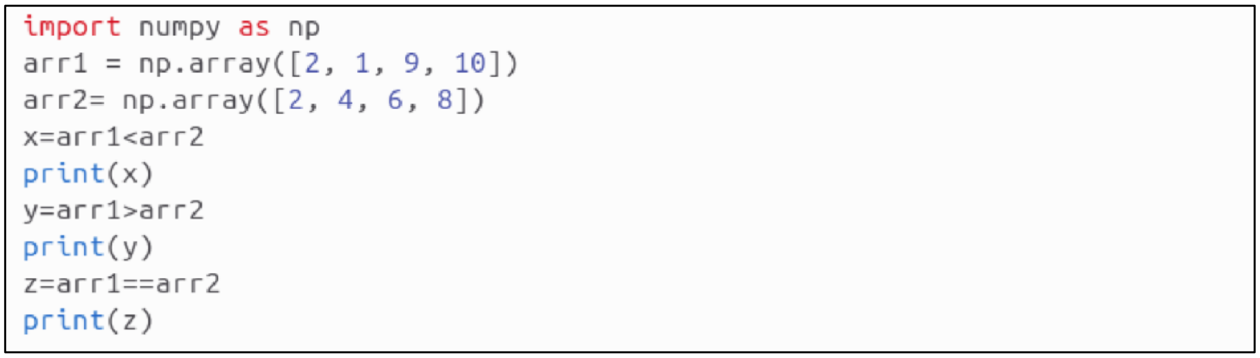
<u>NumPy Comparison Operators</u>

NumPy provides various element-wise comparison operators that can compare the elements of two NumPy arrays.

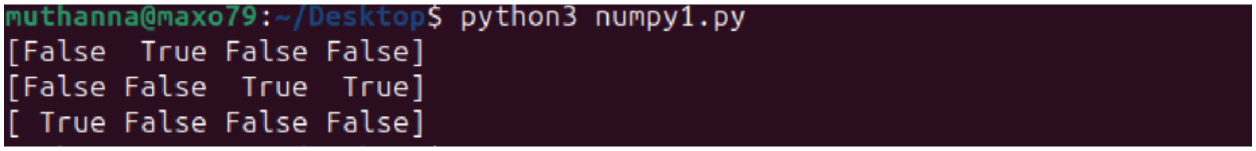
Here's a list of various comparison operators available in NumPy.

Operators	Descriptions
`<` (less than)	returns `True` if element of the first array is less than the second one
`<=` (less than or equal to)	returns `True` if element of the first array is less than or equal to the second one
`>` (greater than)	returns `True` if element of the first array is greater than the second one
`>=` (greater than or equal to)	returns `True` if element of the first array is greater than or equal to the second one
`==` (equal to)	returns `True` if the element of the first array is equal to the second one
`!=` (not equal to)	returns `True` if the element of the first array is not equal to the second one

Example 1: NumPy Comparison Operators



```
import numpy as np
arr1 = np.array([2, 1, 9, 10])
arr2= np.array([2, 4, 6, 8])
x=arr1<arr2
print(x)
y=arr1>arr2
print(y)
z=arr1==arr2
print(z)
```



```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[False  True False False]
[False False  True  True]
[ True False False False]
```

Here, we can see that the output of the comparison operators is also an array, where each element is either True or False based on the array element's comparison.

4.8 NumPy Comparison Functions

NumPy also provides built-in functions to perform all the comparison operations.

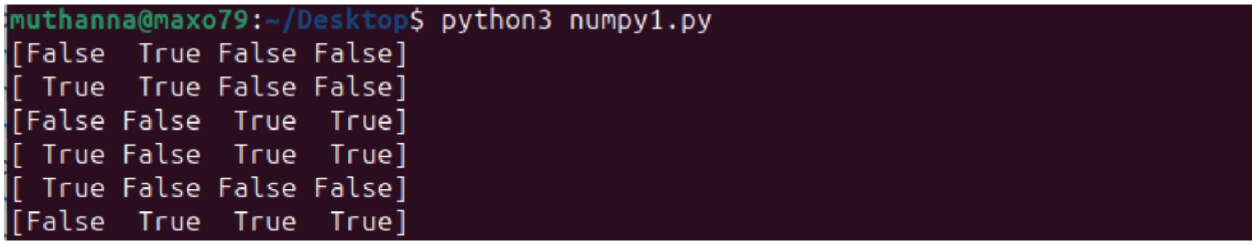
For example, the less() function returns True if each element of the first array is less than the corresponding element in the second array.

Here's a list of all built-in comparison functions.

unctions	Descriptions
`less()`	returns element-wise `True` if the first value is less than the second
`less_equal()`	returns element-wise `True` if the first value is less than or equal to second
`greater()`	returns element-wise `True` if the first value is greater then second
`greater_equal()`	returns element-wise `True` if the first value is greater than or equal to second
`equal()`	returns element-wise `True` if two values are equal
`not_equal()`	returns element-wise `True` if two values are not equal

Example 2: NumPy Comparison Functions

```
import numpy as np
arr1 = np.array([2, 1, 9, 10])
arr2= np.array([2, 4, 6, 8])
x=np.less(arr1,arr2)
print(x)
x1=np.less_equal(arr1,arr2)
print(x1)
y=np.greater(arr1,arr2)
print(y)
y1=np.greater_equal(arr1,arr2)
print(y1)
z=np.equal(arr1,arr2)
print(z)
z1=np.not_equal(arr1,arr2)
print(z1)
```



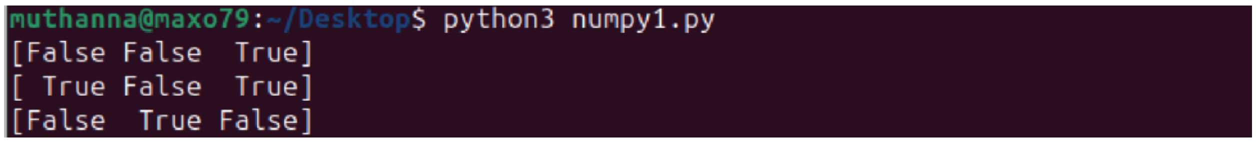
```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[False  True False False]
[ True  True False False]
[False False  True  True]
[ True False  True  True]
[ True False False False]
[False  True  True  True]
```

4.9 NumPy Logical Operations

As mentioned earlier, logical operators perform Boolean algebra; a branch of algebra that deals with True and False statements.
Logical operations are performed element-wise. For example, if we have two arrays x1 and x2 of the same shape, the output of the logical operator will also be an array of the same shape.
Here's a list of various logical operators available in NumPy:

Operators	Descriptions
logical_and	Computes the element-wise truth value of **x1** AND **x2**
logical_or	Computes the element-wise truth value of **x1** OR **x2**
logical_not	Computes the element-wise truth value of NOT **x**

```
import numpy as np
arr1 = np.array([True, False, True])
arr2 = np.array([False, False, True])
print(np.logical_and(arr1, arr2))
print(np.logical_or(arr1, arr2))
print(np.logical_not(arr1))
```



```
muthanna@maxo79:~/Desktop$ python3 numpy1.py
[False False  True]
[ True False  True]
[False  True False]
```