Note: The version of Numpy I'm using is **1.19.1**. The version of Python I'm using is **3.7.3**.

Arrays:

- **Introduction:**
- NumPy's main object is the homogeneous multidimensional array.
- In NumPy dimensions are called **axes**.
- For example, the coordinates of a point in 3D space **[1, 2, 1]** has one axis. That axis has 3 elements in it, so we say it has a length of 3.
- The array, **[[1, 0, 0], [0, 1, 2]]**, has 2 axes. The first axis has a length of 2, the second axis has a length of 3.
- NumPy's array class is called ndarray. It is also known by the alias array.
  Note: numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.
- **Basic Numpy Array Methods:**
- Here are some basic numpy array methods:
  - **ndarray.ndim:** The number of axes (dimensions) of the array.
  - **ndarray.shape:** The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.
  - **ndarray.size:** The total number of elements of the array. This is equal to the product of the elements of shape.
  - **ndarray.dtype:** An object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
  - **ndarray.itemsize:** The size in bytes of each element of the array.
  - **ndarray.data:** The buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.
- E.g.

```
a = np.array([[1,2,3], [4,5,6]])
print(a)
print(a.shape)
print(a.dtype)
print(a.size)
print(a.itemsize)
```

```
[[1 2 3]
 [4 5 6]]
(2, 3)
int64
6
8
```

```
a = np.array(42) # 0 Dimension Array
b = np.array([1, 2, 3, 4, 5]) # 1 Dimension Array
c = np.array([[1, 2, 3], [4, 5, 6]]) # 2 Dimension Array
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]]) # 3 Dimension Array

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

- **Ndmin:**
  When creating an array, you can use the **ndmin** argument to specify the number of dimensions.
- E.g.

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

print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4 5 6 7 8 9]]]]]
number of dimensions : 5
```

- **Creating Zero Arrays:**
- To create a zero array, an array full of 0's, use the function **np.zeros**.
- E.g.

```
zero_arr = np.zeros(10)
print(zero_arr)
```

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

Here, I'm creating a 1-D array of length 10, full of 0's.

```
zero_arr = np.zeros(shape=(5, 2))
print(zero_arr)
```

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

Here, I'm creating a 2-D array, 5 rows and 2 columns, full of 0's.

- **Creating One Arrays:**
- To create a one array, an array full of 1's, use the function **np.ones**.
- E.g.

```
zero_arr = np.ones(10)
print(zero_arr)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

```
zero_arr = np.ones(shape=(5, 2))
print(zero_arr)
```

```
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]
```

- **Creating Identity Matrices:**
- To create an identity array, use the function **np.eye**.
- E.g.

```
identity = np.eye(10)
print(identity)
```

```
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

- **Arange:**
- You can use the **ndarray.arange** function to create an array from 0 to n-1, where n is the input you pass in.
- E.g.

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- You can also add start and “jump by” value.
- E.g.

```
print(np.arange(0, 10, 2))
```

```
[0 2 4 6 8]
```

Here, I’m starting at 0, ending at 9 (10-1), and jumping by 2 values each time.

```
print(np.arange(3, 10, 2))
```

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

Here, I’m starting at 3, ending at 9 (10-1), and jumping by 2 values each time.

- **Reshape:**
- Reshaping means changing the shape of an array.
- By reshaping we can add or remove dimensions or change the number of elements in each dimension.
- We can reshape into any shape as long as the number of elements match.
  For example, if we have a 1-D array of length 8, we can't reshape it into a 3*3 matrix as that needs 9 elements.
- E.g.

```
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]]
```

- We can flatten any multi-dimensional array into a 1-D array by doing **reshape(-1)**.
- E.g.

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

newarr = arr.reshape(-1)

print(newarr)
```

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

- **Copy vs View:**
- The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
- The copy owns the data and any changes made to the copy will not affect the original array, and any changes made to the original array will not affect the copy.
- The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.
- To create a copy, use the **copy()** method.
- To create a view, use the **view()** method.
- E.g.

```
a = np.array([[1,2,3], [4,5,6]])
b = a.copy()
b[0,0] = 10
print(b)
print(a)
```

```
[[10  2  3]
 [ 4  5  6]]
[[1 2 3]
 [4 5 6]]
```

Here, I'm making a copy of a, called b, and then changing the first element of the first subarray of b to be 10. Notice that this did not affect a.

```
a = np.array([[1,2,3], [4,5,6]])
b = a.copy()
a[0,0] = 10
print(b)
print(a)
```

```
[[1 2 3]
 [4 5 6]]
[[10  2  3]
 [ 4  5  6]]
```

- Here, I'm making a copy of a, called b, and then changing the first element of the first subarray of a to be 10. Notice that this did not affect b.

```
a = np.array([[1,2,3], [4,5,6]])
b = a.view()
b[0,0] = 10
print(b)
print(a)
```

```
[[10  2  3]
 [ 4  5  6]]
[[10  2  3]
 [ 4  5  6]]
```

Here, I'm making a view of a, called b, and then changing the first element of the first subarray of b to be 10. Notice that this did affect a.

```
a = np.array([[1,2,3], [4,5,6]])
b = a.view()
a[0,0] = 10
print(b)
print(a)
```

```
[[10  2  3]
 [ 4  5  6]]
[[10  2  3]
 [ 4  5  6]]
```

Here, I'm making a view of a, called b, and then changing the first element of the first subarray of a to be 10. Notice that this did affect b.

- As mentioned above, copies own the data, and views do not own the data, but how can we check this?
- Every NumPy array has the attribute **base** that returns None if the array owns the data. Otherwise, the base attribute refers to the original object.

- E.g.

```
a = np.array([[1,2,3], [4,5,6]])
b = a.copy()
c = a.view()
print(a.base)
print(b.base)
print(c.base)
```

```
None
None
[[1 2 3]
 [4 5 6]]
```

Here, since a owns the data, a.base returns None. Likewise, b.base also returns None. However, c is a view of a, so it doesn't own the data, and hence the array is returned.

- **Array Indexing:**
- To do indexing for a 1-D array, you do it like: **a[i]** where i is the index.
- To do indexing for a 2-D array, you do it like: **a[i, j]** where i is the dimension and j is the index.
- To do indexing for a 2-D array, you do it like: **a[i, j, k]** where i is the first dimension, j is the second dimension and k is the index.
- E.g.

```
a = np.array([1,2,3,4,5])
b = np.array([[1,2,3],[4,5,6]])
c = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(a[0])
print(b[0,0])
print(c[0,0,0])
```

```
1
1
1
```

Here, b[0][0] returns 1 because b[0] returns the first dimension ([1,2,3]), and so b[0][0] returns the first element of that dimension, 1.

Here, c[0][0][0] returns 1 because c[0] returns the first dimension ([[1,2,3], [4,5,6]]), c[0][0] returns the second dimension ([1,2,3]) and c[0][0][0] returns the first element of that dimension, 1.

- **Array Slicing:**
- Slicing in Python is done like this: [start:end:step].
- If we don't pass "start" it's considered 0.
- If we don't pass "end" it's considered the length of the array in that dimension.
- If we don't pass "step" it's considered 1.
- E.g.

```
a = np.array([1,2,3,4,5,6,7])
print(a[::])
print(a[1:])
print(a[::2])
print(a[:6:])
```

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

```
b = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(b[::])
print()
print(b[1::])
print()
print(b[:1:])
print()
print(b[1::2])
print()
print(b[::2])
```

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

[[4 5 6]
 [7 8 9]]

[[1 2 3]]

[[4 5 6]]

[[1 2 3]
 [7 8 9]]
```

- **Math Operations:**

```
a = np.array([[1,2,3],[-3, 0, 1]])
b = np.array([[5,6,7], [1, 2, 1]])

# Element-wise addition
print(a + b)
print()

# Element-wise multiplication
print(a * b)
print()

# Element-wise division
print(a / b)
print()

# Element-wise exponential
print(a ** b)
print()

# Transposing B
print(b.T)
print()
# Matrix multiplication
print(a @ b.T)
print()
print(np.matmul(a, b.T))
```

```
[[ 6  8 10]
 [-2  2  2]]

[[ 5 12 21]
 [-3  0  1]]

[[ 0.2         0.33333333  0.42857143]
 [-3.          0.          1.        ]]

[[   1   64 2187]
 [  -3    0    1]]

[[5 1]
 [6 2]
 [7 1]]

[[38  8]
 [-8 -2]]

[[38  8]
 [-8 -2]]
```

Data Types:

- Some commonly used data types include:
  - np.uint8
  - np.int32
  - np.int64
  - np.float32
  - np.float64
  - np.bool
- When creating arrays, we can use the second parameter to specify the type of the elements in the array.
- Furthermore, to get the type of the elements of the array, you can use the **dtype** method.
- E.g.

```
x = np.array([1,2,3], dtype=np.int64)
```

```
print(x, x.dtype)
```

```
[1 2 3] int64
```

```
x = np.array([1,2,3], dtype=np.bool)
```

```
print(x, x.dtype)
```

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

NaN and Inf:

- **Nan** stands for “Not a number”. It usually happens when your function input is not part of the function domain. E.g. When you do log(-1) or divide by 0.
- E.g.

```
In [6]: print(np.log(-1))
```

```
nan
```

- **Inf** stands for infinity. **-Inf** stands for negative infinity.
- E.g.

```
print(np.exp(10000000000000000))
```

```
inf
```

```
print(np.log(0))
```

```
-inf
```