Reference guide: Arrays

As you've learned, NumPy is a powerful library capable of performing advanced numerical computing. One of its main features is its ability to work with vectors, arrays, and matrices. The NumPy library also has many useful mathematical functions. Data professionals use NumPy for many different tasks. One of its primary advantages is its speed. An operation applied to a vector executes much faster than the same operation applied to a list. Performance increases become apparent when working with large volumes of data. This reading is a reference guide for working with NumPy arrays.

Create an array

Remember that to use NumPy, you must first import it. Standard practice is to alias it as `np`.

`np.array()`

Creates an `ndarray` (n-dimensional array). There is no limit to how many dimensions a NumPy array can have, but arrays with many dimensions can be more difficult to work with.

1-D array:

```
import numpy as np
array_1d = np.array([1, 2, 3])
array_1d
```

Notice that a one-dimensional array is similar to a list.

2-D array:

```
array_2d = np.array([(1, 2, 3), (4, 5, 6)])
array_2d
```

Notice that a two-dimensional array is similar to a table.

3-D array:

```
array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

```
array_3d
```

Notice that a three-dimensional array is similar to two tables.

np.zeros()

- Creates an array of a designated shape that is pre-filled with zeros:

```
np.zeros((3, 2))
```

np.ones()

- Creates an array of a designated shape that is pre-filled with ones:

```
np.ones((2, 2))
```

np.full()

- Creates an array of a designated shape that is pre-filled with a specified value:

```
np.full((5, 3), 8)
```

These functions are useful for various situations, including:

- To initialize an array of a specific size and shape and then subsequently fill it with values derived from a calculation
- To allocate memory for later use
- To perform matrix operations

Array methods

NumPy arrays have many methods that allow you to manipulate and operate on them. For a full list of these, refer to the NumPy array documentation. Some of the most commonly used methods include:

ndarray.flatten()

- Returns a copy of the array collapsed into one dimension

```
array_2d = np.array([(1, 2, 3), (4, 5, 6)])
array_2d.flatten()
```

ndarray.reshape()

- Gives a new shape to an array without changing its data

```
array_2d = np.array([(1, 2, 3), (4, 5, 6)])
array_2d.reshape(3, 2)
```

Note that a value of -1 in the designated new shape indicates for NumPy to infer the value based on other given values. This is for user convenience.

```
array_2d = np.array([(1, 2, 3), (4, 5, 6)])
array_2d.reshape(3, -1)
```

ndarray.tolist()

- Converts an array to a list object. Multidimensional arrays are converted to nested lists.

```
array_2d = np.array([(1, 2, 3), (4, 5, 6)])
array_2d.tolist()
```

Mathematical functions

NumPy arrays also have many methods that are mathematical functions, such as:

- ndarray.max(): returns the maximum value in the array or along a specified axis
- ndarray.mean(): returns the mean of all the values in the array or along a specified axis
- ndarray.min(): returns the minimum value in the array or along a specified axis
- ndarray.std(): returns the standard deviation of all the values in the array or along a specified axis

```
a = np.array([(1, 2, 3), (4, 5, 6)])
print(a.max())
print(a.mean())
print(a.min())
print(a.std())
```

Array attributes

NumPy arrays have several attributes that allow you to access information about the array. Some of the most commonly used attributes include:

- ndarray.shape: returns a tuple of the array's dimensions
- ndarray.dtype: returns the data type of the array's contents
- ndarray.size: returns the total number of elements in the array
- ndarray.T: returns the array transposed (rows become columns, columns become rows)

```
array_2d = np.array([(1, 2, 3), (4, 5, 6)])
print(array_2d.shape)
print(array_2d.dtype)
print(array_2d.size)
print(array_2d.T)
```

Indexing and slicing

You can access individual elements of a NumPy array using indexing and slicing. Indexing in NumPy is similar to indexing in Python lists, except you can use multiple indices to access elements in multidimensional arrays.

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

You can also use slicing to access subarrays of a NumPy array:

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

Array operations

NumPy arrays support many operations, including mathematical functions and arithmetic operations. These include array addition and multiplication, which performs element-wise arithmetic on arrays:

```
a = np.array([(1, 2, 3), (4, 5, 6)])
b = np.array([[1, 2, 3], [1, 2, 3]])
print(a + b)
print(a * b)
```

There are also nearly 100 other useful mathematical functions that can be applied to individual and/or multiple arrays.

Mutability

NumPy arrays are mutable, but with certain limitations. For instance, you can change an existing element of an array:

```
a = np.array([(1, 2), (3, 4)])
a[1][1] = 100
a
```

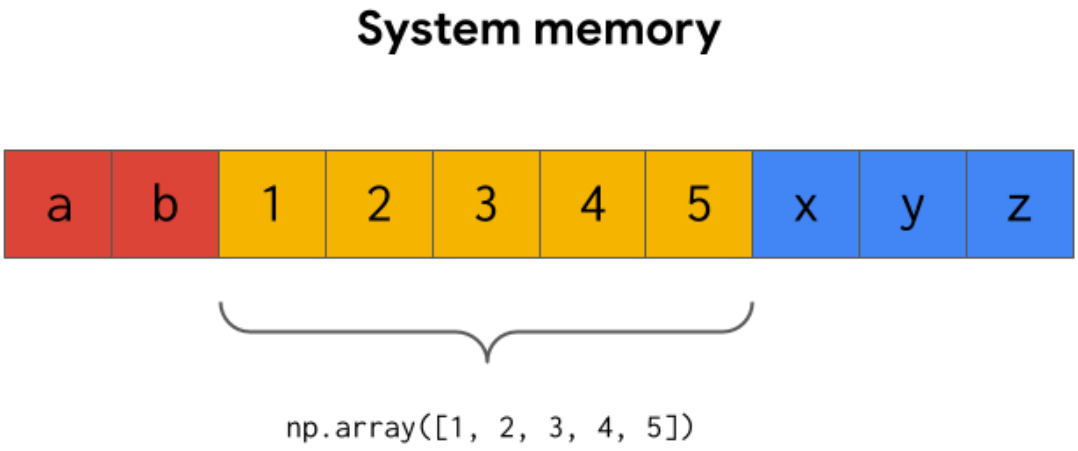
However, you cannot lengthen or shorten the array:

```
a = np.array([1, 2, 3])
a[3] = 100
a
```

How NumPy arrays store data in memory

NumPy arrays work by allocating a contiguous block of memory at the time of instantiation. Other structures in Python don't do this. Their data is scattered across the system's memory. This is what makes NumPy arrays so fast. All the data is stored together at a particular address in the system's memory. However, this is also what prevents you from being able to lengthen or shorten the array. The abutting memory is occupied by other information. There's no room for

more data at that memory address. However, you can replace existing elements of the array with new elements.



The only way to lengthen an array is to copy the existing array to a new memory address along with the new data.