Google

Reading: Reference guide: Arrays

As you've learned, NumPy is a powerful library capable of performing advanced numerical computing. One of its main benefits is the ability to work with arrays, as an operation applied to a vector executes much faster than the same operation applied to a list. Performance increases become further apparent when working with large volumes of data. This reading is a reference guide for working with NumPy arrays.

Save this course item

You may want to save a copy of this guide for future reference. Use it as a resource for additional practice or in your future professional projects. To access a downloadable version of this course item, click the link below and select "Use Template."

Reference guide: Arrays

OR

If you don't have a Google account, download the item directly from the attachment below.

Reference guide: Arrays

Create an array

As you've discovered, to use NumPy, you must first import it. Standard practice is to alias it as **np**.

np.array()

This 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
```

Output:

```
[1 2 3]
```

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
```

Output:

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

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
```

Output:

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

 [[5 6]
  [7 8]]]
```

np.zeros()

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

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

  Output:

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

np.ones()

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

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

Output:

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

np.full()

- And this creates an array of a designated shape that is pre-filled with a specified value:

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

Output:

```
[[ 8.  8.  8.]
 [ 8.  8.  8.]
 [ 8.  8.  8.]
 [ 8.  8.  8.]
 [ 8.  8.  8.]]
```

These functions are useful for various situations:

- To initialize an array of a specific size and shape, then 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, refer to the NumPy array documentation. Some of the most commonly used methods follow:

ndarray.flatten()

- This returns a copy of the array collapsed into one dimension.

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

Output:

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

[1 2 3 4 5 6]
```

ndarray.reshape()

- This gives a new shape to an array without changing its data.

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

Output:

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

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

Adding a value of -1 in the designated new shape makes the process more efficient, as it indicates for NumPy to automatically infer the value based on other given values.

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

Output:

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

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

ndarray.tolist()

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

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

Output:

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

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

Mathematical functions

NumPy arrays also have many methods that are mathematical functions:

- 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)
print()

print(a.max())
print(a.mean())
print(a.min())
print(a.std()
```

Output:

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

6
3.5
1
1.70782512766
```

Array attributes

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

- 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)
print()

print(array_2d.shape)
print(array_2d.dtype)
print(array_2d.size)
print(array_2d.T
```

Output:

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

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

Indexing and slicing

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

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

print(a[1])
print(a[0, 1])
```

```
print(a[1, 2]
```

Output:

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

[4 5 6]
2
6
```

Slicing may also be used to access subarrays of a NumPy array:

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

a[:, 1:]
```

Output:

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

[[2 3]
 [5 6]]
```

Array operations

NumPy arrays support many operations, including mathematical functions and arithmetic. 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:')
print(a)
print()
print('b:')
print(b)
print()
print('a + b:')
print(a + b)
```

```
print()
print('a * b:')
print(a * b)
```

Output:

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

b:
[[1 2 3]
 [1 2 3]]

a + b:
[[2 4 6]
 [5 7 9]]

a * b:
[[ 1  4  9]
 [ 4 10 18]]
```

In addition, there are nearly 100 other useful mathematical functions that can be applied to individual or multiple arrays.

Mutability

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

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

a[1][1] = 100
a
```

Output:

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

[[  1   2]
```

```
[  3 100]]
```

However, the array cannot be lengthened or shortened:

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

a[3] = 100
a
```

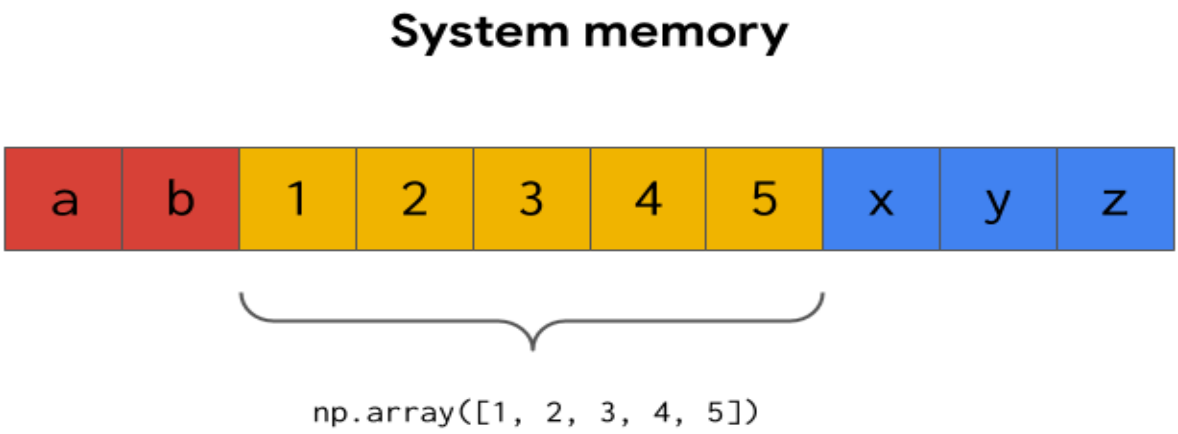
Output:

```
Error on line 5:
   a[3] = 100
IndexError: index 3 is out of bounds for axis 0 with size 3
```

How NumPy arrays store data in memory

NumPy arrays work by allocating a contiguous block of memory at the time of instantiation. Most 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.

Interestingly, this is also what prevents an array from being lengthened or shortened: The abutting memory is occupied by other information. There's no room for more data at that memory address. However, existing elements of the array can be replaced 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.
