Creating NumPy ndarrays

We strongly encourage you to type the commands that you have learned in this demo. However, the Notebook demonstrated in the video above, is available at the bottom of this page. To download the file (Creating and Saving NumPy ndarrays.ipynb) to your computer, right-click on the link, then choose "Save Link As...".

At the core of NumPy is the **ndarray**, where *nd* stands for n-dimensional. An ndarray is a multidimensional array of elements all of the same type. In other words, an ndarray is a grid that can take on many shapes and can hold either numbers or strings. In many Machine Learning problems you will often find yourself using ndarrays in many different ways. For instance, you might use an ndarray to hold the pixel values of an image that will be fed into a Neural Network for image classification.

But before we can dive in and start using NumPy to create ndarrays we need to import it into Python. We can import packages into Python using the import command and it has become a convention to import NumPy as np. Therefore, you can import NumPy by typing the following command in your Jupyter notebook:

```
import numpy as np
```

There are several ways to create ndarrays in NumPy. In the following lessons we will see two ways to create ndarrays:

1. Using regular Python lists
2. Using built-in NumPy functions

In this section, we will create ndarrays by providing Python lists to the NumPy np.array() function. This can create some confusion for beginners, but it is important to remember that np.array() is *NOT* a class, it is just a function that returns an ndarray. We should note that for the purposes of clarity, the examples throughout these lessons will use small and simple ndarrays. Let's start by creating 1-Dimensional (1D) ndarrays.

```
# We import NumPy into Python

import numpy as np


# We create a 1D ndarray that contains only integers

x = np.array([1, 2, 3, 4, 5])


# Let's print the ndarray we just created using the print() command

print('x = ', x)

x = [1 2 3 4 5]
```

Rank of an Array (numpy.ndarray.ndim)

Syntax:

```
ndarray.ndim
```

It returns the number of array dimensions.

Let's pause for a second to introduce some useful terminology. We refer to 1D arrays as *rank* 1 arrays. In general *N*-Dimensional arrays have rank *N*. Therefore, we refer to a 2D array as a rank 2 array.

```
# 1-D array

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

x.ndim


# 2-D array

Y = np.array([[1,2,3],[4,5,6],[7,8,9], [10,11,12]])

Y.ndim


# Here the`zeros()` is an inbuilt function that you'll study on the next page.

# The tuple (2, 3, 4( passed as an argument represents the shape of the ndarray

y = np.zeros((2, 3, 4))

y.ndim
```

```
1
2
3
```

numpy.ndarray.shape

Syntax:

```
ndarray.shape
```

It returns a tuple representing the array dimensions. Refer more details here.

Another important property of arrays is their *shape*. The shape of an array is the size along each of its dimensions. For example, the shape of a rank 2 array will correspond to the number of *rows* and *columns* of the array. As you will see, NumPy ndarrays have *attributes* that allow us to get information about them in a very intuitive way. For example, the shape of an ndarray can be obtained using the .shape attribute. The shape attribute returns a tuple of N positive integers that specify the sizes of each dimension.

numpy.dtype

The type tells us the data-type of the elements. Remember, a NumPy array is homogeneous, meaning all elements will have the same data-type. In the example below, we will create a rank 1 array and learn how to obtain its shape, its type, and the data-type (*dtype*) of its elements.

Example 1.a - Using a 1-D Array of Integers

```
# We create a 1D ndarray that contains only integers

x = np.array([1, 2, 3, 4, 5])


# We print information about x

print('x = ', x)

print('x has dimensions:', x.shape)

print('x is an object of type:', type(x))

print('The elements in x are of type:', x.dtype)
```

```
x = [1 2 3 4 5]

x has dimensions: (5,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: int64
```

We can see that the shape attribute returns the tuple (5,) telling us that x is of rank 1 (i.e. x only has 1 dimension) and it has 5 elements. The type() function tells us that x is indeed a NumPy ndarray. Finally, the .dtype attribute tells us that the elements of x are stored in memory as *signed 64-bit integers*. Another great advantage of NumPy is that it can handle more data-types than Python lists. You can check out all the different data types NumPy supports in the link below:

NumPy Data Types

As mentioned earlier, ndarrays can also hold strings. Let's see how we can create a rank 1 ndarray of strings in the same manner as before, by providing the np.array() function a Python list of strings.

Example 1.b - Using 1-D Array of Strings

```
# We create a rank 1 ndarray that only contains strings

x = np.array(['Hello', 'World'])


# We print information about x

print('x = ', x)

print('x has dimensions:', x.shape)
```

```
print('x is an object of type:', type(x))

print('The elements in x are of type:', x.dtype)

x = ['Hello' 'World']

x has dimensions: (2,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: U5
```

As we can see the shape attribute tells us that x now has only 2 elements, and even though x now holds strings, the type() function tells us that x is still an ndarray as before. In this case however, the .dtype attribute tells us that the elements in x are stored in memory as *Unicode strings of 5 characters*.

It is important to remember that one big difference between Python lists and ndarrays, is that unlike Python lists, all the elements of an ndarray must be of the same type. So, while we can create Python lists with both integers and strings, we can't mix types in ndarrays. If you provide the np.array() function with a Python list that has both integers and strings, NumPy will interpret all elements as strings. We can see this in the next example:

Example 1.c - Using a 1-D Array of Mixed Datatype

```
# We create a rank 1 ndarray from a Python list that contains integers and strings

x = np.array([1, 2, 'World'])


# We print information about x

print('x = ', x)

print('x has dimensions:', x.shape)

print('x is an object of type:', type(x))

print('The elements in x are of type:', x.dtype)

x = ['1' '2' 'World']

x has dimensions: (3,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: U21
```

We can see that even though the Python list had mixed data types, the elements in x are all of the same type, namely, *Unicode strings of 21 characters*. We won't be using ndarrays with strings for the remaining of this introduction to NumPy, but it's important to remember that ndarrays can hold strings as well.

Using a 1-D Array to Demonstrate Upcasting in Numeric datatype

Up till now, we have only created ndarrays with integers and strings. We saw that when we create an ndarray with only integers, NumPy will automatically assign the dtype int64 to its elements. Let's see what happens when we create ndarrays with floats and integers.

Example 1.d - Using a 1-D Array of Int and Float

```
# We create a rank 1 ndarray that contains integers

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


# We create a rank 1 ndarray that contains floats

y = np.array([1.0,2.0,3.0])


# We create a rank 1 ndarray that contains integers and floats

z = np.array([1, 2.5, 4])


# We print the dtype of each ndarray

print('The elements in x are of type:', x.dtype)

print('The elements in y are of type:', y.dtype)

print('The elements in z are of type:', z.dtype)
```

```
The elements in x are of type: int64
The elements in y are of type: float64
The elements in z are of type: float64
```

We can see that when we create an ndarray with only floats, NumPy stores the elements in memory as *64-bit floating point numbers (float64)*. However, notice that when we create an ndarray with both floats and integers, as we did with the z ndarray above, NumPy assigns its elements a *float64* dtype as well. This is called *upcasting*. Since all the elements of an ndarray must be of the same type, in this case NumPy upcasts the integers in z to floats in order to avoid losing precision in numerical computations.

Using a 1-D Array of Float, and specifying the dtype of each element

Even though NumPy automatically selects the dtype of the ndarray, NumPy also allows you to specify the particular dtype you want to assign to the elements of the ndarray. You can specify the dtype when you create the ndarray using the keyword dtype in the np.array() function. Let's see an example:

Example 1.e - Using a 1-D Array of Float, and specifying the datatype of each element as int64

```
# We create a rank 1 ndarray of floats but set the dtype to int64

x = np.array([1.5, 2.2, 3.7, 4.0, 5.9], dtype = np.int64)


# We print the dtype x

print('x = ', x)

print('The elements in x are of type:', x.dtype)
```

x = [1 2 3 4 5]

The elements in x are of type: int64

We can see that even though we created the ndarray with floats, by specifying the dtype to be int64, NumPy converted the floating point numbers into integers by removing their decimals. Specifying the data type of the ndarray can be useful in cases when you don't want NumPy to accidentally choose the wrong data type, or when you only need certain amount of precision in your calculations and you want to save memory.

numpy.ndarray.size and Creating a 2-D array

Another useful attribute is NumPy.size, which returns the number of elements in the array. Let us now look at how we can create a rank 2 ndarray from a nested Python list.

Example 2 - Using a 2-D Array (Rank #2 Array)

```
# We create a rank 2 ndarray that only contains integers

Y = np.array([[1,2,3],[4,5,6],[7,8,9], [10,11,12]])


print('Y = \n', Y)


# We print information about Y

print('Y has dimensions:', Y.shape)

print('Y has a total of', Y.size, 'elements')

print('Y is an object of type:', type(Y))

print('The elements in Y are of type:', Y.dtype)
```

Y =
[[1 2 3]
 [4 5 6]

```
 [ 7 8 9]
 [10 11 12]]

Y has dimensions: (4, 3)
Y has a total of 12 elements
Y is an object of type: class 'numpy.ndarray'
The elements in Y are of type: int64
```

We can see that now the shape attribute returns the tuple (4,3) telling us that Y is of rank 2 and it has 4 rows and 3 columns. The .size attribute tells us that Y has a total of 12 elements.

Notice that when NumPy creates an ndarray it automatically assigns its *dtype* based on the type of the elements you used to create the ndarray.

Save the NumPy array to a File

Once you create an ndarray, you may want to save it to a file to be read later or to be used by another program. NumPy provides a way to save the arrays into files for later use - let's see how this is done.

Example 3 - Save the NumPy array to a File

```
# We create a rank 1 ndarray

x = np.array([1, 2, 3, 4, 5])


# We save x into the current directory as

np.save('my_array', x)
```

The above saves the x ndarray into a file named my_array.npy. You can *load* the saved ndarray into a variable by using the load() function.

```
# We load the saved array from our current directory into variable y

y = np.load('my_array.npy')


# We print y

print()

print('y = ', y)

print()


# We print information about the ndarray we loaded

print('y is an object of type:', type(y))
```

```
print('The elements in y are of type:', y.dtype)
```

```
y = [1 2 3 4 5]

y is an object of type: class 'numpy.ndarray'
The elements in y are of type: int64
```

When loading an array from a file, make sure you include the name of the file together with the extension .npy, otherwise you will get an error.

Additional Resource

- Refer more example at NumPy.org - How to create a basic array

Supporting Materials

- Creating and Saving NumPy ndarrays