Using Built-in Functions to Create ndarrays

We strongly encourage you to type the commands that you have learned in this demo. However, the notebook file demonstrated in the video above, Using Built-in Functions to Create ndarrays.ipynb, is available at the bottom of this page.

One great time-saving feature of NumPy is its ability to create ndarrays using built-in functions. These functions allow us to create certain kinds of ndarrays with just one line of code. Below we will see a few of the most useful built-in functions for creating ndarrays that you will come across when doing AI programming.

Let's start by creating an ndarray with a specified shape that is full of zeros. We can do this by using the np.zeros() function. The function np.zeros(shape) creates an ndarray full of zeros with the given shape. So, for example, if you wanted to create a rank 2 array with 3 rows and 4 columns, you will pass the shape to the function in the form of (rows, columns), as in the example below:

Example 1. Create a Numpy array of zeros with a desired shape

```
# We create a 3 x 4 ndarray full of zeros.

X = np.zeros((3,4))


# We print X

print()

print('X = \n', X)

print()


# We print information about 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 =
[[ 0. 0. 0. 0.]
[ 0. 0. 0. 0.]
[ 0. 0. 0. 0.]]

X has dimensions: (3, 4)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: float64
```

As we can see, the np.zeros() function creates by default an array with dtype float64. If desired, the data type can be changed by using the keyword dtype.

Similarly, we can create an ndarray with a specified shape that is full of *ones*. We can do this by using the np.ones() function. Just like the np.zeros() function, the np.ones() function takes as an argument the shape of the ndarray you want to make. Let's see an example:

Example 2. Create a Numpy array of ones

```
# We create a 3 x 2 ndarray full of ones.

X = np.ones((3,2))


# We print X

print()

print('X = \n', X)

print()


# We print information about 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. 1.]
[ 1. 1.]
[ 1. 1.]]

X has dimensions: (3, 2)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: float64
```

As we can see, thenp.ones() function also creates by default an array with dtype float64. If desired, the data type can be changed by using the keyword dtype.

We can also create an ndarray with a specified shape that is full of any number we want. We can do this by using the np.full() function. The np.full(shape, constant value) function takes two arguments. The first argument is the shape of the ndarray you want to make and the second is the constant value you want to populate the array with. Let's see an example:

Example 3. Create a Numpy array of constants

```
# We create a 2 x 3 ndarray full of fives.
```

```
X = np.full((2,3), 5)


# We print X

print()

print('X = \n', X)

print()


# We print information about 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 =
[[5 5 5]
[5 5 5]]

X has dimensions: (2, 3)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: int64
```

The np.full() function creates by default an array with the same data type as the constant value used to fill in the array. If desired, the data type can be changed by using the keyword dtype.

As you will learn later, a fundamental array in Linear Algebra is the Identity Matrix. An Identity matrix is a square matrix that has only 1s in its main diagonal and zeros everywhere else. The function np.eye(N) creates a square N x N ndarray corresponding to the Identity matrix. Since all Identity Matrices are square, the np.eye() function only takes a single integer as an argument. Let's see an example:

Example 4 a. Create a Numpy array of an Identity matrix

```
# We create a 5 x 5 Identity matrix.

X = np.eye(5)


# We print X

print()

print('X = \n', X)

print()
```

```
# We print information about 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. 0. 0. 0. 0.]
[ 0. 1. 0. 0. 0.]
[ 0. 0. 1. 0. 0.]
[ 0. 0. 0. 1. 0.]
[ 0. 0. 0. 0. 1.]]

X has dimensions: (5, 5)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: float64
```

As we can see, the np.eye() function also creates by default an array with dtype float64. If desired, the data type can be changed by using the keyword dtype. You will learn all about Identity Matrices and their use in the Linear Algebra section of this course. We can also create diagonal matrices by using the np.diag() function. A diagonal matrix is a square matrix that only has values in its main diagonal. The np.diag() function creates an ndarray corresponding to a diagonal matrix , as shown in the example below:

Example 4 b. Create a Numpy array of constants

```
# Create a 4 x 4 diagonal matrix that contains the numbers 10,20,30, and 50

# on its main diagonal

X = np.diag([10,20,30,50])


# We print X

print()

print('X = \n', X)

print()
```

```
X =
[[10 0 0 0]
[ 0 20 0 0]
[ 0 0 30 0]
[ 0 0 0 50]]
```

numpy.arange

Syntax:

```
numpy.arange([start, ]stop, [step, ]dtype=None)
```

It returns evenly spaced values within a given interval. Details about the optional arguments are available here.

NumPy also allows you to create ndarrays that have evenly spaced values within a given interval. NumPy's np.arange() function is very versatile and can be used with either one, two, or three arguments. Below we will see examples of each case and how they are used to create different kinds of ndarrays.

Let's start by using np.arange() with only one argument. When used with only one argument, np.arange(N) will create a rank 1 ndarray with consecutive integers between 0 and N - 1. Therefore, notice that if I want an array to have integers between 0 and 9, I have to use N = 10, *NOT* N = 9, as in the example below:

Example 5. Create a Numpy array of evenly spaced values in a given range, using arange(stop_val)

```
# We create a rank 1 ndarray that has sequential integers from 0 to 9

x = np.arange(10)


# We print the ndarray

print()

print('x = ', x)

print()


# We print information about the ndarray

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 = [0 1 2 3 4 5 6 7 8 9]

x has dimensions: (10,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: int64
```

When used with two arguments, np.arange(start,stop) will create a rank 1 ndarray with evenly spaced values within the half-open interval [start, stop). This means the evenly spaced numbers will include start but *exclude* stop. Let's see an example

Example 6. Create a Numpy array using arange(start_val, stop_val)

```
# We create a rank 1 ndarray that has sequential integers from 4 to 9.

x = np.arange(4,10)


# We print the ndarray

print()

print('x = ', x)

print()


# We print information about the ndarray

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 = [4 5 6 7 8 9]

x has dimensions: (6,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: int64
```

As we can see, the function np.arange(4,10) generates a sequence of integers with 4 inclusive and 10 exclusive.

Finally, when used with three arguments, np.arange(start,stop,step) will create a rank 1 ndarray with evenly spaced values within the half-open interval [start, stop) with step being the distance between two adjacent values. Let's see an example:

Example 7. Create a Numpy array using arange(start_val, stop_val, step_size)

```
# We create a rank 1 ndarray that has evenly spaced integers from 1 to 13 in steps of 3.

x = np.arange(1,14,3)


# We print the ndarray

print()
```

```
print('x = ', x)

print()


# We print information about the ndarray

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 4 7 10 13]

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 x has sequential integers between 1 and 13 but the difference between all adjacent values is 3.

numpy.linspace

Syntax:

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
```

It returns num evenly spaced values calculated over the interval [start, stop]. Details about the optional arguments are available here.

Even though the np.arange() function allows for non-integer steps, such as 0.3, the output is usually inconsistent, due to the finite floating point precision. For this reason, in the cases where non-integer steps are required, it is usually better to use the function np.linspace(). The np.linspace(start, stop, N) function returns N evenly spaced numbers over the *closed* interval [start, stop]. This means that both the start and thestop values are included. We should also note the np.linspace() function needs to be called with at least two arguments in the form np.linspace(start,stop). In this case, the default number of elements in the specified interval will be *N= 50*. The reason np.linspace() works better than the np.arange() function, is that np.linspace() uses the number of elements we want in a particular interval, instead of the step between values. Let's see some examples:

Example 8. Create a Numpy array using linspace(start, stop, n), with stop inclusive.

```
# We create a rank 1 ndarray that has 10 integers evenly spaced between 0 and 25.

x = np.linspace(0,25,10)


# We print the ndarray

print()
```

```
print('x = \n', x)

print()


# We print information about the ndarray

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 = [ 0. 2.77777778 5.55555556 8.33333333 11.11111111
13.88888889 16.66666667 19.44444444 22.22222222 25. ]

x has dimensions: (10,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: float64
```

As we can see from the above example, the function np.linspace(0,25,10) returns an ndarray with 10 evenly spaced numbers in the closed interval [0, 25]. We can also see that both the start and end points, 0 and 25 in this case, are included. However, you can let the endpoint of the interval be excluded (just like in the np.arange() function) by setting the keyword endpoint = False in the np.linspace() function. Let's create the same x ndarray we created above but now with the endpoint excluded:

Example 9. Create a Numpy array using linspace(start, stop, n), with stop excluded.

```
# We create a rank 1 ndarray that has 10 integers evenly spaced between 0 and 25,

# with 25 excluded.

x = np.linspace(0,25,10, endpoint = False)


# We print the ndarray

print()

print('x = ', x)

print()


# We print information about the ndarray

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 = [0. 2.5 5. 7.5 10. 12.5 15. 17.5 20. 22.5]

x has dimensions: (10,)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: float64

As we can see, because we have excluded the endpoint, the spacing between values had to change in order to fit 10 evenly spaced numbers in the given interval.

numpy.reshape - This is a Function.

Syntax:

```
numpy.reshape(array, newshape, order='C')[source]
```

It gives a new shape to an array without changing its data. More details about the arguments are available here.

So far, we have only used the built-in functions np.arange() and np.linspace() to create rank 1 ndarrays. However, we can use these functions to create rank 2 ndarrays of any shape by combining them with the np.reshape() function. The np.reshape(ndarray, new_shape) function converts the given ndarray into the specified new_shape. It is important to note that the new_shape should be compatible with the number of elements in the given ndarray. For example, you can convert a rank 1 ndarray with 6 elements, into a 3 x 2 rank 2 ndarray, or a 2 x 3 rank 2 ndarray, since both of these rank 2 arrays will have a total of 6 elements. However, you can't reshape the rank 1 ndarray with 6 elements into a 3 x 3 rank 2 ndarray, since this rank 2 array will have 9 elements, which is greater than the number of elements in the original ndarray. Let's see some examples:

Example 10. Create a Numpy array by feeding the output of arange() function as an argument to the reshape() function.

```
# We create a rank 1 ndarray with sequential integers from 0 to 19

x = np.arange(20)


# We print x

print()

print('Original x = ', x)

print()


# We reshape x into a 4 x 5 ndarray

x = np.reshape(x, (4,5))
```

```
# We print the reshaped x

print()

print('Reshaped x = \n', x)

print()


# We print information about the reshaped 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)
```

```
Original x = [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]

Reshaped x =
[[ 0 1 2 3 4]
[ 5 6 7 8 9]
[10 11 12 13 14]
[15 16 17 18 19]]

x has dimensions: (4, 5)
x is an object of type: class 'numpy.ndarray'
The elements in x are of type: int64
```

numpy.ndarray.reshape - This one is a Method.

Syntax:

```
ndarray.reshape(shape, order='C')
```

It returns an array containing the same data with a new shape. More details about the arguments are available here.

Need clarity between a Method versus Function? - Refer to this discussion on StackOverflow

One great feature about NumPy, is that some functions can also be applied as methods. This allows us to apply different functions in sequence in just one line of code. ndarray methods are similar to ndarray attributes in that they are both applied using *dot* notation (.). Let's see how we can accomplish the same result as in the above example, but in just one line of code:

Example 11. Create a Numpy array by calling the reshape() function from the output of arange() function.

```
# We create a a rank 1 ndarray with sequential integers from 0 to 19 and

# reshape it to a 4 x 5 array
```

```
Y = np.arange(20).reshape(4, 5)


# We print Y

print()

print('Y = \n', Y)

print()


# We print information about Y

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

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

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

```
Y =
[[ 0 1 2 3 4]
[ 5 6 7 8 9]
[10 11 12 13 14]
[15 16 17 18 19]]

Y has dimensions: (4, 5)
Y is an object of type: class 'numpy.ndarray'
The elements in Y are of type: int64
```

As we can see, we get the exact same result as before. Notice that when we use reshape() as a method, it's applied as ndarray.reshape(new_shape). This converts the ndarray into the specified shape new_shape. As before, it is important to note that the new_shape should be compatible with the number of elements in ndarray. In the example above, the function np.arange(20) creates an ndarray and serves as the ndarray to be reshaped by the reshape() method. Therefore, when using reshape() as a method, we don't need to pass the ndarray as an argument to the reshape() function, instead we only need to pass the new_shape argument.

In the same manner, we can also combine reshape() with np.linspace() to create rank 2 arrays, as shown in the next example.

Example 12. Create a rank 2 Numpy array by using the reshape() function.

```
# We create a rank 1 ndarray with 10 integers evenly spaced between 0 and 50,

# with 50 excluded. We then reshape it to a 5 x 2 ndarray

X = np.linspace(0,50,10, endpoint=False).reshape(5,2)
```

```
# We print X

print()

print('X = \n', X)

print()


# We print information about 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 =
[[ 0. 5.]
[ 10. 15.]
[ 20. 25.]
[ 30. 35.]
[ 40. 45.]]

X has dimensions: (5, 2)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: float64
```

The last type of ndarrays we are going to create are *random* ndarrays. Random ndarrays are arrays that contain random numbers. Often in Machine Learning, you need to create random matrices, for example, when initializing the weights of a Neural Network. NumPy offers a variety of random functions to help us create random ndarrays of any shape.

Let's start by using the np.random.random(shape) function to create an ndarray of the given shape with random floats in the half-open interval [0.0, 1.0).

Example 13. Create a Numpy array using the numpy.random.random() function.

```
# We create a 3 x 3 ndarray with random floats in the half-open interval [0.0, 1.0).

X = np.random.random((3,3))


# We print X

print()

print('X = \n', X)

print()
```

```
# We print information about 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 =
[[ 0.12379926 0.52943854 0.3443525 ]
[ 0.11169547 0.82123909 0.52864397]
[ 0.58244133 0.21980803 0.69026858]]

X has dimensions: (3, 3)
X is an object of type: class 'numpy.ndarray'
The elements in x are of type: float64
```

NumPy also allows us to create ndarrays with random integers within a particular interval. The function np.random.randint(start, stop, size = shape) creates an ndarray of the given shape with random integers in the half-open interval [start, stop). Let's see an example:

Example 14. Create a Numpy array using the numpy.random.randint() function.

```
# We create a 3 x 2 ndarray with random integers in the half-open interval [4, 15).

X = np.random.randint(4,15,size=(3,2))


# We print X

print()

print('X = \n', X)

print()


# We print information about 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 =
[[ 7 11]
[ 9 11]
[ 6 7]]
```

X has dimensions: (3, 2)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: int64

In some cases, you may need to create ndarrays with random numbers that satisfy certain statistical properties. For example, you may want the random numbers in the ndarray to have an average of 0. NumPy allows you create random ndarrays with numbers drawn from various probability distributions. The function np.random.normal(mean, standard deviation, size=shape), for example, creates an ndarray with the given shape that contains random numbers picked from a normal (Gaussian) distribution with the given mean and standard deviation. Let's create a 1,000 x 1,000 ndarray of random floating point numbers drawn from a normal distribution with a mean (average) of zero and a standard deviation of 0.1.

Example 15. Create a Numpy array of "Normal" distributed random numbers, using the numpy.random.normal() function.

```
# We create a 1000 x 1000 ndarray of random floats drawn from normal (Gaussian) distribution

# with a mean of zero and a standard deviation of 0.1.

X = np.random.normal(0, 0.1, size=(1000,1000))


# We print X

print()

print('X = \n', X)

print()


# We print information about 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)

print('The elements in X have a mean of:', X.mean())

print('The maximum value in X is:', X.max())

print('The minimum value in X is:', X.min())

print('X has', (X < 0).sum(), 'negative numbers')

print('X has', (X > 0).sum(), 'positive numbers')
```

```
X =
[[ 0.04218614 0.03247225 -0.02936003 ..., 0.01586796 -0.05599115 -0.03630946]
```

```
[ 0.13879995 -0.01583122 -0.16599967 ..., 0.01859617 -0.08241612 0.09684025]
[ 0.14422252 -0.11635985 -0.04550231 ..., -0.09748604 -0.09350044 0.02514799]
...,
[-0.10472516 -0.04643974 0.08856722 ..., -0.02096011 -0.02946155 0.12930844]
[-0.26596955 0.0829783 0.11032549 ..., -0.14492074 -0.00113646 -0.03566034]
[-0.12044482 0.20355356 0.13637195 ..., 0.06047196 -0.04170031 -0.04957684]]

X has dimensions: (1000, 1000)
X is an object of type: class 'numpy.ndarray'
The elements in X are of type: float64
The elements in X have a mean of: -0.000121576684405
The maximum value in X is: 0.476673923106
The minimum value in X is: -0.499114224706
X has 500562 negative numbers
X has 499438 positive numbers
```

As we can see, the average of the random numbers in the ndarray is close to zero, both the maximum and minimum values in X are symmetric about zero (the average), and we have about the same amount of positive and negative numbers.

Additional Resource

- Refer to more examples available at NumPy.org - Array creation

Supporting Materials

- Using Built-in Functions to Create ndarrays