PYTHON

NumPy Arange Example | Np.Arange() Function In Python

By **Krunal** Last updated **Mar 29, 2020**

NumPy arange() is an inbuilt numpy function that returns a **ndarray** object containing evenly spaced values within the given range. Python np.arange() function returns an evenly spaced values within a given interval. For integer arguments, the method is equivalent to a Python built-in range function but returns the ndarray rather than a list.

NumPy arange()

NumPy is the fundamental Python library for numerical computing.

Its most important type is an array type called ndarray.

NumPy offers a lot of array creation routines for different circumstances. The arange() is one such function based on numerical ranges.

It’s often referred to as np.arange() because np is a widely used abbreviation for NumPy.

Creating NumPy arrays is important when you're working with other Python libraries that rely on them, like SciPy, Pandas, Matplotlib, scikit-learn, and more. NumPy is suitable for creating and working with arrays because it offers useful routines, enables performance boosts, and allows you to write concise code.

#Syntax

The syntax of **numpy.arange()** function is the following.

```
numpy.arange(start, stop, step, dtype)
```

The parameters are the following.

start: number, optional. Start of an interval. The interval includes this value. The default start value is 0.

stop: number. End of the interval. The interval does not include stop value, except in some cases where a *step* is not an integer and floating-point round-off affects the length of *out*.

step: number, optional. step can't be zero. Otherwise, you'll get a ZeroDivisionError. You can't move away anywhere from the start if the increment or decrement is 0.
dtype: The type of an output array. If the dtype is not given, infer the data type from the other input arguments. If dtype is omitted, arange() will try to deduce the type of the array elements from the types of start, stop, and step.

You can find more information on the parameters and the return value of arange() in the official documentation.

Let's see the NumPy arange function example in Jupyter Notebook.

When working with NumPy routines, you have to import NumPy first.

Write the following code inside the first cell.

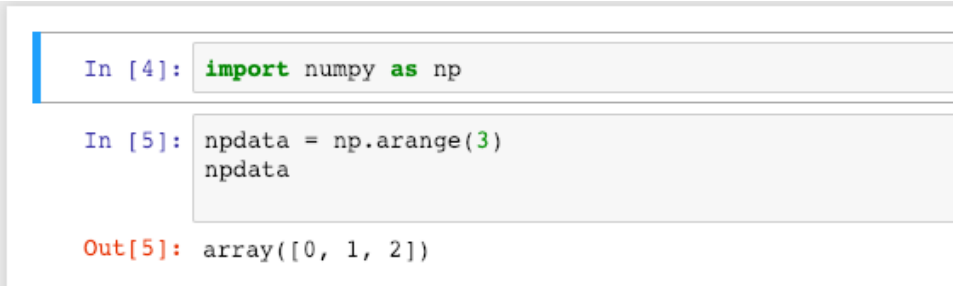
```
import numpy as np
```

Now, you have NumPy imported, and you're ready to apply **arange()**.

Run that cell using Ctrl + Enter and then write the following code in the next cell.

```
npdata = np.arange(3)
npdata
```

See the output below.

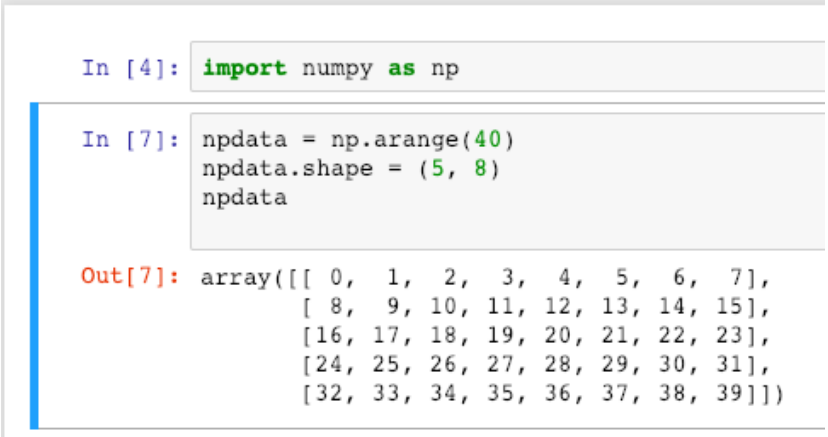


It has created a numpy array from 0 to 2 elements with a length of 3.

Let's see another example. Write the following Python code in the cell.

```
npdata = np.arange(40)
npdata.shape = (5, 8)
npdata
```

In the above code, we have defined an array with the items of 40, and then we have numpy array's shape attribute to shape that array into 5 rows and 8 columns. See the output.

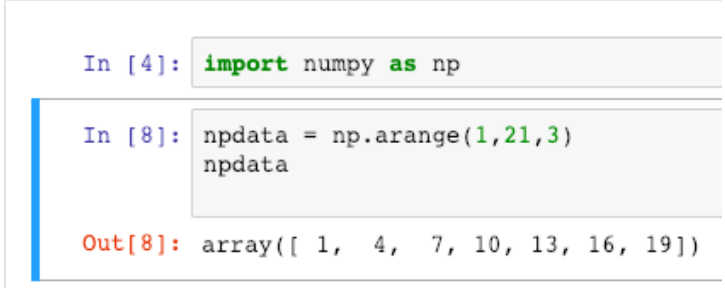


#Start and stop parameters set in numpy arange

Let's define the start and stop parameters in the numpy arange function. See the output.

```
npdata = np.arange(1,21,3)
npdata
```

In the above code, we have passed the first parameter as a starting point, then go to 21 and with step 3. See the output below.



The above code sample returns an array with the array starting from 1 and up to 21 with the step of 3. So 1, (1 +3 = 4), (4 + 3 = 7),... up to 21 as an endpoint.

Now, let's see another example.

```
#app.py

import numpy as np

print(np.arange(1, 21.1, 3))
```

See the output.

```
➜  pyt python3 app.py
[ 1.  4.  7. 10. 13. 16. 19.]
➜  pyt
```

In this case, you get the array with seven elements.

Notice that this example creates an array of floating-point numbers, unlike the previous one. That's because you haven't defined dtype and arange() deduced it for you.

#Providing negative arguments

If you provide negative values for **start** or both **start** and **stop**, and have a positive **step**, then arange() will work the same way as with all positive arguments:

```
# app.py

import numpy as np

print(np.arange(-8, -2, 2))
```

See the output.

```
➜  pyt python3 app.py
[-8 -6 -4]
➜  pyt
```

The counting begins with the value of **start**, repeatedly incrementing by **step**, and ending before a **stop** is reached.

Sometimes you'll want an array with the values decrementing from left to right. In such cases, you can use arange() with a negative value for step, and with a start greater than stop.

#Working with empty arrays

There are several edge cases where you can obtain empty NumPy arrays with arange(). These are regular instances of numpy.ndarray without any elements.

If you provide equal values for start and stop, then you'll get an empty array.

```
#app.py

import numpy as np

print(np.arange(11, 11))
```

See the output.

```
➜  pyt python3 app.py
[]
➜  pyt
```

#numpy arange reshape

numpy.reshape(array, shape, order = 'C')

It shapes an array without changing the data of array.

```
#app.py

import numpy as np

arr = np.arange(8)
print("Original Array : \n", arr)

# shape arr with 2 rows and 4 columns
arr = np.arange(8).reshape(2, 4)
print("\nArray reshaped with 2 rows and 4 columns : \n", arr)

# shape arr with 2 rows and 4 columns
arr = np.arange(8).reshape(4 ,2)
print("\nArray reshaped with 2 rows and 4 columns : \n", arr)

# Constructs 3D arr
arr = np.arange(8).reshape(2, 2, 2)
print("\nOriginal Array reshaped to 3D : \n", arr)
```

See the output.

```
➜  pyt python3 app.py
Original Array :
 [0 1 2 3 4 5 6 7]

Array reshaped with 2 rows and 4 columns :
 [[0 1 2 3]
 [4 5 6 7]]

Array reshaped with 2 rows and 4 columns :
 [[0 1]
 [2 3]
 [4 5]
 [6 7]]

Original Array reshaped to 3D :
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
➜  pyt
```

#numpy arange vs. python range

For large arrays, numpy should be the faster solution.

The range gives you a regular list (python 2) or a specialized “range object” (like a generator; python 3), np.arangegives you a numpy array. If you care about speed enough to use numpy, use numpy arrays.

NumPy’s arrays are more compact than Python lists: a list of lists as you describe, in Python, would take at least 20 MB or so, while a NumPy 3D array with single-precision floats in the cells would fit in 4 MB. Access to reading and writing items is also faster with NumPy.

NumPy is not just more efficient; it is also more convenient. You get a lot of vector and matrix operations for free, which sometimes allow one to avoid unnecessary work. And they are also efficiently implemented.

Finally, Numpy arange Example is over.

#See also

How To Create NumPy Arrays

Python Pandas DataFrame

Pandas DataFrame read_csv()

Python Pandas Series

Python property()

Python

Krunal - 835 Posts - 195 Comments

Krunal Lathiya is From India, and he is an Information Technology Engineer. By profession, he is the latest web and mobile technology adapter, freelance developer, Machine Learning, Artificial Intelligence enthusiast, and primary Author of this blog.

This site uses Akismet to reduce spam. Learn how your comment data is processed.

© 2017-2020 Sprint Chase Technologies. All rights reserved