In [2]:

```
# Let's see another example of reshaping a NumPy array from lower to higher
# dimensions. The following script defines a NumPy array of shape (4,6). The original ar
# is then reshaped to a three-dimensional array of shape (3, 4, 2). Notice here
# again that the product of dimensions of the original array (4 x 6) and the
# reshaped array (3 x 4 x 2) is the same, i.e., 24.

import numpy as np
print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\nthree-dimensional array")
three_d_array = np.reshape(two_d_array,(3,4,2))
print(three_d_array)
```

```
two-dimensional array
[[16  4 16  8 12  9]
 [12  6  2  6 15 13]
 [15 10  5 14 12  5]
 [ 5 16  5  3  7 13]]

three-dimensional array
[[[16  4]
  [16  8]
  [12  9]
  [12  6]]

 [[ 2  6]
  [15 13]
  [15 10]
  [ 5 14]]

 [[12  5]
  [ 5 16]
  [ 5  3]
  [ 7 13]]]
```

In [5]:

```
# Let's try to reshape a NumPy array in a way that the product of dimensions
# does not match. In the script below, the shape of the original array is (4,6).
# Next, you try to reshape this array to the shape (1,4,2). In this case, since
# the product of dimensions of the original and the reshaped array don't match,
# you will see an error in the output.

import numpy as np
print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\nthree-dimensional array")
three_d_array = np.reshape(two_d_array,(1,4,2))
print(three_d_array)
```

```
two-dimensional array
[[15  7  2  6 13 11]
 [11  9 12 11 17 17]
 [17 11 14 19 17 12]
 [ 6  3 19  4  1  5]]

three-dimensional array


---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
<ipython-input-5-c1d48d27c366> in <module>
     10 print(two_d_array)
     11 print("\nthree-dimensional array")
---> 12 three_d_array = np.reshape(two_d_array,(1,4,2))
     13 print(three_d_array)

<__array_function__ internals> in reshape(*args, **kwargs)

c:\python\lib\site-packages\numpy\core\fromnumeric.py in reshape(a, newsha
pe, order)
    297            [5, 6]])
    298     """
--> 299     return _wrapfunc(a, 'reshape', newshape, order=order)
    300
    301

c:\python\lib\site-packages\numpy\core\fromnumeric.py in _wrapfunc(obj, me
thod, *args, **kwds)
     56
     57     try:
---> 58         return bound(*args, **kwds)
     59     except TypeError:
     60         # A TypeError occurs if the object does have such a method
in its

ValueError: cannot reshape array of size 24 into shape (1,4,2)
```

In [6]:

```
# Let's now see a few examples of reshaping NumPy arrays from higher to
# lower dimensions. In the script below, the original array is of shape (4,6)
# while the new array is of shape (24). The reshaping, in this case, will be successful
# since the product of dimensions for original and reshaped arrays is the same.

import numpy as np
print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\none-dimensional array")
one_d_array = two_d_array.reshape(24)
print(one_d_array)
```

```
two-dimensional array
[[12  5  3  6  1  4]
 [17  6 17  2  4 16]
 [ 5  3  2  3 11 13]
 [ 3 12  8  1 10 16]]

one-dimensional array
[12  5  3  6  1  4 17  6 17  2  4 16  5  3  2  3 11 13  3 12  8  1 10 16]
```

In [7]:

```
# Finally, to convert an array of any dimensions to a flat, one-dimensional
# array, you will need to pass –1 as the argument for the reshaped function, as
# shown in the script below, which converts a two-dimensional array to a one dimensional
# array.

import numpy as np
print("two-dimensional array")
two_d_array = np.random.randint(1,20, size = (4,6))
print(two_d_array)
print("\none-dimensional array")
one_d_array = two_d_array.reshape(-1)
print(one_d_array)
```

```
two-dimensional array
[[11 14  6  9  8  2]
 [ 8 16  6 14 15  6]
 [ 5 15  3  2 12  9]
 [17  1  9  6 16  8]]

one-dimensional array
[11 14  6  9  8  2  8 16  6 14 15  6  5 15  3  2 12  9 17  1  9  6 16  8]
```

In [8]:

```
# Similarly, the following script converts a three-dimensional array to a one dimensiona
# array.

import numpy as np
print("two-dimensional array")
three_d_array = np.random.randint(1,20, size = (4,2,6))
print(three_d_array)
print("\non-dimensional array")
one_d_array = three_d_array .reshape(-1)
print(one_d_array)
```

```
two-dimensional array
[[[ 8  6  8 16  5  9]
  [17  7 12 16 11  9]]

 [[ 8 16  4  4  1 13]
  [ 3  1 11  8  6 19]]

 [[ 5 13  1  6  6  4]
  [16 19  2 17 12 18]]

 [[ 3 17  2 19 11 19]
  [ 6  2  6 12  9 11]]]

on-dimensional array
[ 8  6  8 16  5  9 17  7 12 16 11  9  8 16  4  4  1 13  3  1 11  8  6 19
  5 13  1  6  6  4 16 19  2 17 12 18  3 17  2 19 11 19  6  2  6 12  9 11]
```

In [9]:

```
# NumPy arrays can be indexed and sliced. Slicing an array means dividing an
# array into multiple parts. NumPy arrays are indexed just like normal lists.
# Indexes in NumPy arrays start from 0, which means that the first item of a
# NumPy array is stored at the 0th index. The following script creates a simple
# NumPy array of the first 10 positive integers.

import numpy as np
s = np.arange(1,11)
print(s)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

In [10]:

```
# The item at index one can be accessed as follows:

import numpy as np
s = np.arange(1,11)
print(s)
print(s[1])
```

```
[ 1  2  3  4  5  6  7  8  9 10]
2
```

In [11]:

```
# To slice an array, you have to pass the lower index, followed by a colon and
# the upper index. The items from the lower index (inclusive) to the upper
# index (exclusive) will be filtered. The following script slices the array “s”
# from the 1st index to the 9th index. The elements from index 1 to 8 are
# printed in the output.

import numpy as np
s = np.arange(1,11)
print(s)
print(s[1:9])
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[2 3 4 5 6 7 8 9]
```

In [12]:

```
# If you specify only the upper bound, all the items from the first index to the
# upper bound are returned. Similarly, if you specify only the lower bound, all
# the items from the lower bound to the last item of the array are returned.

import numpy as np
s = np.arange(1,11)
print(s)
print(s[:5])
print(s[5:])
```

```
[ 1  2  3  4  5  6  7  8  9 10]
[1 2 3 4 5]
[ 6  7  8  9 10]
```

In [13]:

```
# Array slicing can also be applied on a two-dimensional array. To do so, you
# have to apply slicing on arrays and columns separately. A comma separates
# the rows and columns slicing. In the following script, the rows from the
# first and second indexes are returned, while all the columns are returned.
# You can see the first two complete rows in the output.

import numpy as np
row1 = [10,12,13]
row2 = [45,32,16]
row3 = [45,32,16]
nums_2d = np.array([row1, row2, row3])
print(nums_2d[:2,:])
```

```
[[10 12 13]
 [45 32 16]]
```