Broadcasting rules for elementwise operations (+,-,*,/)

"The term broadcasting describes how NumPy treats arrays with **different shapes** during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. Broadcasting provides a means of **vectorizing array operations so that looping occurs in C instead of Python**. It does this **without making needless copies of data** and usually leads to **efficient algorithm implementations**. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation." (Quote from documentation below, emphasis mine)

We will review the basics of broadcasting rules but please see the numpy broadcasting documentation below https://numpy.org/doc/stable/user/basics.broadcasting.html

In [1]:
```
import numpy as np
```

Scalar addition / multiplication is a simple case of broadcasting that is already obvious and used even in math notation.

In [2]:
```
a = np.arange(5)
print(a, 5 + a, 5 * a)
```

```
[0 1 2 3 4] [5 6 7 8 9] [ 0  5 10 15 20]
```

A more advanced case is subtracting or dividing a vector from every row or column of a matrix

In this example, for subtracting the column mean we need to make the vector a 2D array (a matrix with 1 column). This can be done by calling `a[:, None]` where `None` introduces a new dimension to the resulting array. In fact, you can add as many dimensions as you want using `None` . For example, if `a` has shape `(5,)` , then `b = a[None, :, None, None]` will have shape `(1,5,1,1)` .

You can also use `a.reshape(-1, 1)` to add a new axis. For the `reshape` call, one shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions, see https://numpy.org/doc/stable/reference/generated/numpy.ndarray.reshape.html for more details.

More details on why this reshaping is needed will be given after a few broadcasting examples.

In [3]:
```
X = np.outer(np.arange(3)+1, np.arange(5)+1)
row_mean = np.mean(X, axis=0)
col_mean = np.mean(X, axis=1)
print(X)
```

```
print('Row mean', row_mean)
print('Column mean', col_mean)
print('Remove mean from each row\n', X - row_mean)
print('Remove mean from each column\n', X - col_mean.reshape(-1, 1))
```

```
[[ 1  2  3  4  5]
 [ 2  4  6  8 10]
 [ 3  6  9 12 15]]
Row mean [ 2.  4.  6.  8. 10.]
Column mean [3. 6. 9.]
Remove mean from each row
 [[-1. -2. -3. -4. -5.]
 [ 0.  0.  0.  0.  0.]
 [ 1.  2.  3.  4.  5.]]
Remove mean from each column
 [[-2. -1.  0.  1.  2.]
 [-4. -2.  0.  2.  4.]
 [-6. -3.  0.  3.  6.]]
```

You can also divide each row and column by a vector (e.g., to normalize to have standard deviation of 1)

In [4]:

```
row_std = np.std(X, axis=0)
col_std = np.std(X, axis=1)
print('Row normalized X\n', (X - row_mean) / row_std)
print('Col normalized X\n', (X - col_mean[:, None]) / col_std[:, None])
```

```
Row normalized X
 [[-1.22474487 -1.22474487 -1.22474487 -1.22474487 -1.22474487]
 [ 0.          0.          0.          0.          0.        ]
 [ 1.22474487  1.22474487  1.22474487  1.22474487  1.22474487]]
Col normalized X
 [[-1.41421356 -0.70710678  0.          0.70710678  1.41421356]
 [-1.41421356 -0.70710678  0.          0.70710678  1.41421356]
 [-1.41421356 -0.70710678  0.          0.70710678  1.41421356]]
```

Two simple rules for broadcasting

(Copied and slightly modified from documentation)

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. **rightmost**) dimension and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1.

In addition, if the number of dimensions is different (e.g., comparing a 4D array to a 2D array), **missing dimensions will be assumed to have a size of 1.**

If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes.

In [5]:

```
def test_shapes(shape_1, shape_2):
  input_1 = np.ones(shape_1)
  input_2 = np.ones(shape_2)
  print(f'Input 1 shape: {str(input_1.shape):>20}')
```

```
  print(f'Input 2 shape: {str(input_2.shape):>20}')

  try:
    output = input_1 * input_2
  except ValueError as e:
    output = None
  if output is None:
    output_shape = 'Invalid input shapes'
  else:
    output_shape = output.shape
  print(f'Output shape:  {str(output_shape):>20}\n')

print('X and row_mean')
test_shapes(X.shape, row_mean.shape)
print('X and col_mean')
test_shapes(X.shape, col_mean.shape)
print('X and col_mean.reshape')
test_shapes(X.shape, col_mean[:, None].shape)
```

```
X and row_mean
Input 1 shape:               (3, 5)
Input 2 shape:                 (5,)
Output shape:                (3, 5)

X and col_mean
Input 1 shape:               (3, 5)
Input 2 shape:                 (3,)
Output shape:  Invalid input shapes

X and col_mean.reshape
Input 1 shape:               (3, 5)
Input 2 shape:               (3, 1)
Output shape:                (3, 5)
```

Consider scaling each channel of an RGB image example

In [6]:

```
scale = np.arange(3)
print('RGB image (matplotlib) and scale of each channel')
test_shapes((256, 256, 3), scale.shape)

print('RGB image (pytorch) and scale of each channel')
test_shapes((3, 256, 256), scale.shape)

print('RGB image (pytorch) and scale of each channel')
test_shapes((3, 256, 256), scale[:, None, None].shape)
```

```
RGB image (matplotlib) and scale of each channel
Input 1 shape:        (256, 256, 3)
Input 2 shape:                 (3,)
Output shape:         (256, 256, 3)

RGB image (pytorch) and scale of each channel
Input 1 shape:        (3, 256, 256)
Input 2 shape:                 (3,)
Output shape:  Invalid input shapes

RGB image (pytorch) and scale of each channel
Input 1 shape:        (3, 256, 256)
Input 2 shape:            (3, 1, 1)
```

```
Output shape:         (3, 256, 256)
```

Scaling each channel of each image in batch of 32 images

In [7]:

```
print('Scaling the channels of a batch of 32 pytorch RGB images')
test_shapes((32, 3, 256, 256), scale.shape)

print('Scaling the channels of a batch of 32 pytorch RGB images')
test_shapes((32, 3, 256, 256), scale[:, None, None].shape)
```

```
Scaling the channels of a batch of 32 pytorch RGB images
Input 1 shape:    (32, 3, 256, 256)
Input 2 shape:                (3,)
Output shape:  Invalid input shapes

Scaling the channels of a batch of 32 pytorch RGB images
Input 1 shape:    (32, 3, 256, 256)
Input 2 shape:            (3, 1, 1)
Output shape:     (32, 3, 256, 256)
```

Other examples from documentation

In [8]:

```
test_shapes((8,1,6,1), (5,1,3))
test_shapes((5,4), (1,))
test_shapes((5,4), (4,))
```

```
Input 1 shape:         (8, 1, 6, 1)
Input 2 shape:            (5, 1, 3)
Output shape:          (8, 5, 6, 3)

Input 1 shape:               (5, 4)
Input 2 shape:                 (1,)
Output shape:                (5, 4)

Input 1 shape:               (5, 4)
Input 2 shape:                 (4,)
Output shape:                (5, 4)
```

In [9]:

```
test_shapes((15,3,5), (15,1,5))
test_shapes((15,3,5), (3,5))
test_shapes((15,3,5), (3,1))
```

```
Input 1 shape:           (15, 3, 5)
Input 2 shape:           (15, 1, 5)
Output shape:            (15, 3, 5)

Input 1 shape:           (15, 3, 5)
Input 2 shape:              (3, 5)
Output shape:            (15, 3, 5)

Input 1 shape:           (15, 3, 5)
Input 2 shape:              (3, 1)
Output shape:            (15, 3, 5)
```

In [10]:

```
test_shapes((3,), (4,))
test_shapes((2,1), (8,4,3))
```

```
Input 1 shape:                 (3,)
Input 2 shape:                 (4,)
Output shape:  Invalid input shapes

Input 1 shape:               (2, 1)
Input 2 shape:            (8, 4, 3)
Output shape:  Invalid input shapes
```