Computation on Arrays: Broadcasting

Broadcasting:

A **set of rules** by which NumPy lets you apply **binary operations** (e.g., addition, subtraction, multiplication, etc.) **between arrays of different sizes and shapes.**

Introducing Broadcasting

Recall that for arrays of the **same size,** binary operations are performed on an **element-by-element** basis:

In [80]:
```
import numpy as np
```

In [ ]:
```python
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

Out[ ]:
```
array([5, 6, 7])
```

Broadcasting allows these types of **binary operations** to be performed on arrays of **different sizes.**

For example, we can just as easily **add a scalar** (think of it as a zero-dimensional array) **to an array:**

In [ ]:
```python
a + 5
```

Out[ ]:
```
array([5, 6, 7])
```

We can think of this as an operation that **stretches or duplicates** the **value** `5` **into the array** `[5, 5, 5]`, and **adds the results.**

We can similarly **extend this idea** to **arrays of higher dimension.** Observe the result when we add a **one-dimensional array to a two-dimensional** array:

In [ ]:
```
M = np.ones((3, 3))
M
```

Out[ ]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

In [ ]:
```
M + a
```

Out[ ]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Here the one-dimensional array `a` is **stretched,** or **broadcasted, across the second dimension** in order to **match the shape** of `M`.

While these **examples are relatively easy** to understand, **more complicated cases** can involve **broadcasting of both arrays:**

In [67]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]

print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

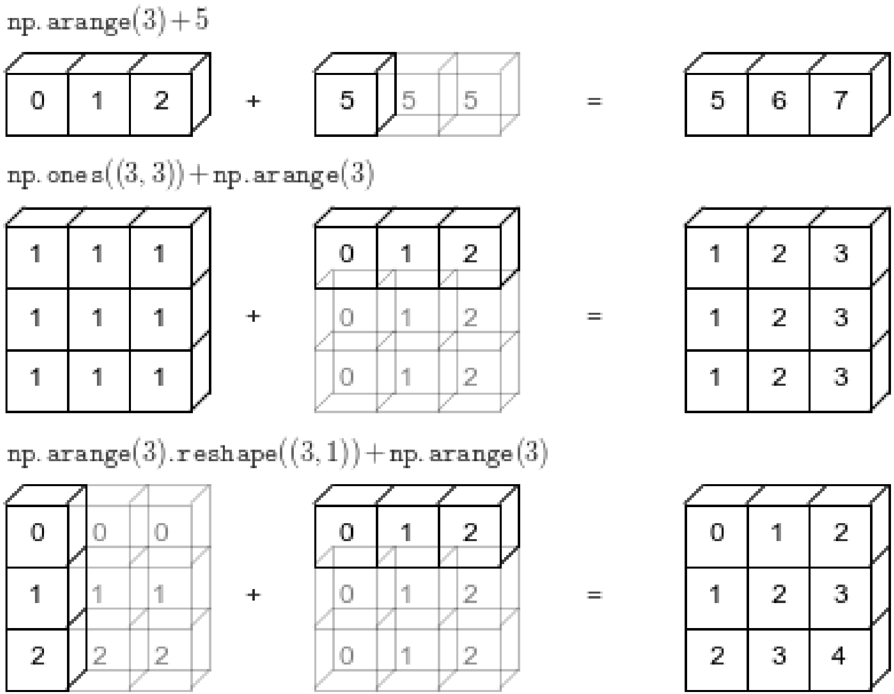
In [ ]:
```
a + b
```

Out[ ]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Just as before we **stretched or broadcasted** one value to **match the shape** of the other.

Here we've stretched **both** `a` and `b` to match **a** common shape, and the **result is a two-dimensional array!**

The **geometry** of these examples is visualized in the following figure:



The **light boxes** represent the **broadcasted values.**

This way of thinking about **broadcasting** may raise questions about its **efficiency in terms of memory use:**

NumPy broadcasting **does not actually copy the broadcasted values** in memory.

Still, this can be a **useful mental model** as we think about broadcasting.

Rules of Broadcasting

Broadcasting in NumPy follows a **strict set of rules** to determine the interaction between the two arrays:

- **Rule 1:** If the **two arrays differ in their number of dimensions,** the shape of the one with fewer dimensions is **padded** with **ones** on its **leading (left) side.**
- **Rule 2:** If the shape of the two arrays does **not match** in **any** dimension, the **array with shape equal to 1** in that dimension is **stretched** to match **the other shape.**
- **Rule 3:** If in **any** dimension the **sizes disagree** and **neither is equal to 1,** an **error** is raised.

To make these rules clear, let's consider **a few examples** in detail.

Broadcasting Example 1

Suppose we want to **add** a **two-dimensional** array to a **one-dimensional** array:

In [17]:
```
M = np.ones((2, 3))
a = np.arange(3)
```

In [ ]:
```
M
```

Out[ ]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [ ]:
```
a
```

Out[ ]:
```
array([0, 1, 2])
```

Let's consider an **operation on these two arrays,** which have the **following shapes:**

- `M.shape` is `(2, 3)`
- `a.shape` is `(3,)`

We see by **rule 1** that the array `a` has **fewer dimensions,** so we **pad** it **on the left with ones:**

- `M.shape` remains `(2, 3)`
- `a.shape` becomes `(1, 3)`

By **rule 2,** we now see that the **first dimension disagrees,** so we **stretch this dimension to match:**

- `M.shape` remains `(2, 3)`
- `a.shape` becomes `(2, 3)`

The **shapes now match,** and we see that the **final shape will be** `(2, 3)`:

In [ ]:
```
M + a
```

Out[ ]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Broadcasting Example 2

Now let's take a look at an **example** where **both arrays need to be broadcast:**

In [ ]:
```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

In [ ]:
```
a
```

Out[ ]:
```
array([[0],
       [1],
       [2]])
```

In [ ]:
```
b
```

Out[ ]:
```
array([0, 1, 2])
```

Again, we'll start by **determining the shapes** of the arrays:

- `a.shape` is `(3, 1)`
- `b.shape` is `(3,)`

Rule 1 says we must **pad the shape** of `b` with **ones:**

- `a.shape` remains `(3, 1)`
- `b.shape` becomes `(1, 3)`

And **rule 2** tells us that we must **upgrade each of these `1`s** to **match the corresponding size** of the other array:

- `a.shape` becomes `(3, 3)`

- `b.shape` becomes `(3, 3)`

Because the **results match,** these **shapes are compatible.** We can see this here:

In [ ]:
```
a + b
```

Out[ ]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Broadcasting Example 3

Next, let's take a look at an **example** in which the two **arrays are not compatible:**

In [39]:
```
M = np.ones((3, 2))
a = np.arange(3)
```

In [ ]:
```
M
```

Out[ ]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

In [ ]:
```
a
```

Out[ ]:
```
array([0, 1, 2])
```

This is just a **slightly different** situation than in the **first example:** the matrix `M` **is transposed.**

How does this affect the calculation? The **shapes** of the arrays are as follows:

- `M.shape` is `(3, 2)`
- `a.shape` is `(3,)`

Again, **rule 1** tells us that we must **pad the shape of `a` with ones:**

- `M.shape` remains `(3, 2)`
- `a.shape` becomes `(1, 3)`

By **rule 2,** the **first dimension of `a`** is then **stretched** to match that of `M`:

- `M.shape` remains `(3, 2)`
- `a.shape` becomes `(3, 3)`

Now we hit **rule 3** —the **final shapes** do **not match,** so these two arrays are **incompatible,** as we can observe by attempting this operation:

In [41]:
```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[41], line 1
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Note the **potential confusion** here:

You could imagine making `a` and `M` **compatible** by, say, padding `a` 's shape with **ones on the right rather than the left.**

But this is **not how the broadcasting rules work!**

That sort of **flexibility** might be **useful in some cases,** but it would lead to potential areas of **ambiguity.**

If **right-side padding is what you'd like,** you can do this **explicitly by reshaping the array** (we'll use the `np.newaxis` keyword introduced in **The Basics of NumPy Arrays** for this):

In [43]:
```
a
```

Out[43]:
```
array([0, 1, 2])
```

In [45]:
```
M
```

Out[45]:
```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

In [47]:
```
a.shape
```

Out[47]:
```
(3,)
```

In [49]:
```
M.shape
```

```
Out[49]: (3, 2)
```

```
In [61]: a[:, np.newaxis]
```

```
Out[61]: array([[0],
                [1],
                [2]])
```

```
In [59]: a[:, np.newaxis].shape
```

```
Out[59]: (3, 1)
```

```
In [57]: M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[57], line 1
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

In [55]:
```python
M + a[:, np.newaxis]
```

Out[55]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

Also notice that while we've been **focusing on the `+` operator** here, these broadcasting rules apply to **any binary ufunc.**

For **example,** here is the `logaddexp(a, b)` function, which computes `log(exp(a) + exp(b))` with **more precision** than the naive approach:

```
In [ ]: np.logaddexp(M, a[:, np.newaxis])
```

```
Out[ ]: array([[1.31326169, 1.31326169],
               [1.69314718, 1.69314718],
               [2.31326169, 2.31326169]])
```

For more information on the many available universal functions, refer to **Computation on NumPy Arrays: Universal Functions.**

Broadcasting in Practice

Broadcasting operations form the **core of many examples you'll see throughout this book.**

We'll now take a look at **some instances** of where they can be useful.

Centering an Array

In **Computation on NumPy Arrays: Universal Functions,** we saw that **ufuncs** allow a NumPy user to **remove the need to explicitly write slow** Python loops.

Broadcasting extends this ability.

One commonly seen **example** in data science is **subtracting the row-wise mean from an array of data.**

Imagine we have an **array of 10 observations, each of which consists of 3 values.**

Using the standard convention (**Data Representation in Scikit-Learn**), we'll **store this in a $\pmb{10 \times 3}$ array:**

In [ ]:
```
rng = np.random.default_rng(seed=1701)
X = rng.random((10, 3))
```

In [ ]:
```
rng
```

Out[ ]:
```
Generator(PCG64) at 0x7AF24F573760
```

In [ ]:
```
X
```

Out[ ]:
```
array([[0.4020733 , 0.30563311, 0.67668051],
       [0.15821208, 0.79247763, 0.09419469],
       [0.36753944, 0.06388928, 0.96431608],
       [0.35200998, 0.54550343, 0.88597945],
       [0.57016965, 0.26614394, 0.8170382 ],
       [0.55906652, 0.06387035, 0.84877751],
       [0.89414484, 0.18920785, 0.23660015],
       [0.16502896, 0.56583856, 0.29513111],
       [0.29078012, 0.90079544, 0.59992434],
       [0.09133896, 0.00578466, 0.97096222]])
```

We can compute the **mean of each column** using the `mean` **aggregate across the first dimension:**

In [ ]:
```
Xmean = X.mean(0)
Xmean
```

Out[ ]:
```
array([0.38503638, 0.36991443, 0.63896043])
```

And now we can **center the** `X` **array** by **subtracting the mean (this is a broadcasting operation):**

In [ ]:
```
X_centered = X - Xmean
```

In [ ]:
```
X_centered
```

Out[ ]:
```
array([[ 0.01703691, -0.06428131,  0.03772009],
       [-0.2268243 ,  0.4225632 , -0.54476574],
       [-0.01749695, -0.30602514,  0.32535566],
       [-0.0330264 ,  0.175589  ,  0.24701902],
       [ 0.18513326, -0.10377048,  0.17807777],
       [ 0.17403013, -0.30604408,  0.20981709],
       [ 0.50910846, -0.18070657, -0.40236028],
       [-0.22000743,  0.19592414, -0.34382932],
       [-0.09425626,  0.53088102, -0.03903608],
       [-0.29369742, -0.36412976,  0.33200179]])
```

To **double-check** that we've done this correctly, we can check that the **centered array has a mean near zero:**

In [ ]:
```
X_centered.mean(0)
```

Out[ ]:
```
array([ 4.99600361e-17, -4.44089210e-17,  0.00000000e+00])
```

To within *machine precision,* the **mean is now zero.**

Plotting a Two-Dimensional Function

One place that **broadcasting** often comes in **handy** is in **displaying images based on two-dimensional functions.**

If we want to define a function $z = f(x, y)$, **broadcasting** can be used to **compute the function across the grid**:

In [82]:
```
# x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]

z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

In [84]:
```
print(x.shape)
print(y.shape)
print(z.shape)
```

```
(50,)
(50, 1)
(50, 50)
```

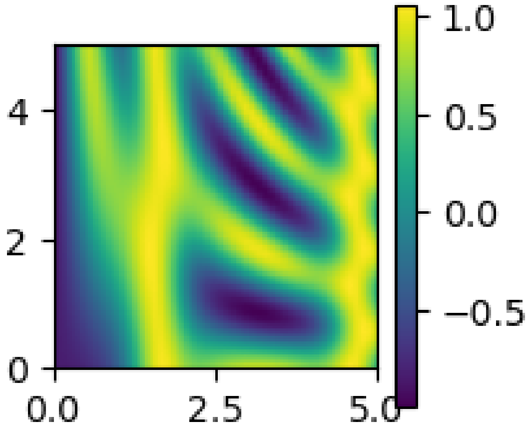
We'll use **Matplotlib** to **plot** this two-dimensional array, shown in the following figure (these tools will be discussed in full in **Density and Contour Plots**):

In [86]:
```
%matplotlib inline
```

```
import matplotlib.pyplot as plt
```

In [110…

```
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5])
plt.rcParams['figure.figsize'] = [.2, .2]
plt.colorbar();
```



The result is a **visualization of the two-dimensional function.**