Basic Math

Note:

Python contains a math module providing functions which operate on built-in scalar data types (e.g. float and complex). This and subsequent chapters assume mathematical functions must operate on arrays, and so are imported from NumPy

Operators

These standard operators are available

Operator Meaning Example Algebraic

- |Addition x + y x+y

- |Subtraction x - y x−y
- |Multiplication x ∗ y xy

/ |Division (Left divide) x/y x

y //|Integer Division x//y b x y c

**| Exponentiation x∗∗y x y

When x and y are scalars, the behavior of these operators is obvious. When x and y are arrays, the behavior of mathematical operations is more complex.

Broadcasting

Under the normal rules of array mathematics, addition and subtraction are only defined for arrays with the same shape or between an array and a scalar. For example, there is no obvious method to add a 5-element vector and a 5 by 4 2-dimensional array. NumPy uses a technique called broadcasting to allow element-by-element mathematical operations on arrays which would not be compatible under the standard rules of array mathematics.

In [8]:

```
import pandas as pd
import numpy as np
x=np.array([[1,2,3.0]])
x
```

Out[8]:

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

In [9]:

```
y=np.array([[0],[0],[0.0]])
```

In [10]:

```
y
```

Out[10]:

```
array([[0.],
       [0.],
       [0.]])
```

In [11]:

```
 x + y # Adding 0 produces broadcast
```

Out[11]:

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

n the next example, x is 3 by 5, so y must be either scalar or a 5-element array or a 1 × 5 array to be broadcastable. When y is a 3-element array (and so matches the leading dimension), an error occurs.

In [13]:

```
x =np. reshape(np.arange(15),(3,5))
x
```

Out[13]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [14]:

```
y=5
```

In [15]:

```
x+y-x
```

Out[15]:

```
array([[5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5],
       [5, 5, 5, 5, 5]])
```

In [17]:

```
y=np.arange(5)
```

In [18]:

```
y
```

Out[18]:

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

In [19]:

```
x+y-x
```

Out[19]:

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

In [21]:

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

Out[21]:

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

In [22]:

```
x+y-x #error
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Input In [22], in <cell line: 1>()
----> 1 x+y-x

ValueError: operands could not be broadcast together with shapes (3,5)
(3,)
```

Addition (+) and Subtraction (-)

Subject to broadcasting restrictions, addition and subtraction operate element-by-element.

multiplication ($*$)

The standard multiplication operator, $*$, performs element-by-element multiplication and so inputs must be broadcastable.

Matrix Multiplication (@)

The matrix multiplication operator @ was introduced in Python 3.5. It can only be used to two arrays and cannot be used to multiply an array and a scalar. If x is N by M and y is K by L and both are non-scalar matrices, x @ y requires M = K. Similarly, y @ x requires L = N. When x and y are both arrays, z = x @ y produces an array with zi j = PM k=1 xikyk j . Notes: The rules for @ conform to the standard rules of matrix multiplication except that scalar multiplication is not allowed. Multiplying an array by a scalar requires using * or dot. x @ y is identical to x.dot(y) or np.dot(x, y).

In [24]:

```
x=np.array([[1.0,2],[3,2],[3,4]])
y=np.array([[9.0,8],[7,6]])
x@y
```

Out[24]:

```
array([[23., 20.],
       [41., 36.],
       [55., 48.]])
```

In [25]:

```
2 @ x # Error
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Input In [25], in <cell line: 1>()
----> 1 2 @ x

ValueError: matmul: Input operand 0 does not have enough dimensions (has
0, gufunc core with signature (n?,k),(k,m?)->(n?,m?) requires 1)
```

In [27]:

```
2 * x
```

Out[27]:

```
array([[2., 4.],
       [6., 4.],
       [6., 8.]])
```

@ supports broadcasting in the sense that multiplying a 1-d array and a 2-d array will promote the 1-d array to be a 2-d array using the rules of broadcasting so that the m element array is created as a 1 by m element array.

Array and Matrix Division (/)

Division is always element-by-element, and the rules of broadcasting are used.

Exponentiation (**)

Array exponentiation operates element-by-element.

```
# Parentheses
Parentheses can be used in the usual way to control the order in which mathematical
expressions are evaluated,
```

```
and can be nested to create complex expressions
```

Transpose

Matrix transpose is expressed using either .T or the transpose function. For instance, if x is an M by N array, transpose(x), x.transpose() and x.T are all its transpose with dimensions N by M. In practice, using the .T is the preferred method and will improve readability of code. Consider

In [31]:

```
x = randn(2,2)
xpx1 = x.T @ x
xpx2 = x.transpose() @ x
xpx3 = transpose(x) @ x
```

```
---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
Input In [31], in <cell line: 1>()
----> 1 x = randn(2,2)
      2 xpx1 = x.T @ x
      3 xpx2 = x.transpose() @ x

NameError: name 'randn' is not defined
```

Transpose has no effect on 1-dimensaional arrays. In 2-dimensions, transpose switches indices so that if z=x.T, z[j,i] is that same as x[i,j]. In higher dimensions, transpose reverses the order or the indices. For example, if x has 3 dimensions and z=x.T, then x[i,j,k] is the same as z[k,j,i]. Transpose takes an optional second argument to specify the axis to use when permuting the array.

Operator Precedence

Computer math, like standard math, has operator precedence which determined how mathematical expressions such as 2**3+3**2/7*13 are evaluated. Best practice is to always use parentheses to avoid ambiguity in the order or operations. The order of evaluation is:

Note:

```
Unary operators are + or - operations that apply to a single element. For exampl
e, consider the expression
```

(-4). This is an instance of a unary negation since there is only a single operation and so (-4)**2 produces 16. On the other hand, -4**2 produces -16 since ** has higher precedence than unary negation and so is interpreted as -(4**2). -4 * -4 produces 16 since it is interpreted as (-4) * (-4) since unary negation has higher precedence than multiplication.

Exercises

1. Using the arrays entered in exercise 1 of chapter 3, compute the values of u + v 0 , v + u 0 , vu, uv and xy (where the multiplication is as defined as linear algebra)

In [32]:

```
import numpy as np

# Define the vectors u and v
u = np.array([1, 1, 2, 3, 5, 8])
v = np.array([1, 1, 2, 3, 5, 8])

# Calculate u + v
addition_uv = u + v

# Calculate v + u
addition_vu = v + u

# Calculate the dot product vu
dot_product_vu = np.dot(v, u)

# Calculate the dot product uv
dot_product_uv = np.dot(u, v)

# Calculate the element-wise product xy
element_wise_product = u * v

print("u + v =", addition_uv)
print("v + u =", addition_vu)
print("vu (dot product) =", dot_product_vu)
print("uv (dot product) =", dot_product_uv)
print("xy (element-wise product) =", element_wise_product)
```

```
u + v = [ 2  2  4  6 10 16]
v + u = [ 2  2  4  6 10 16]
vu (dot product) = 104
uv (dot product) = 104
xy (element-wise product) = [ 1  1  4  9 25 64]
```

Is x/1 legal? If not, why not. What about 1/x?

The expression "s x/1" is not a standard mathematical expression, and its meaning is not clear without additional context or proper notation. It seems to be a combination of symbols, but it lacks mathematical structure. It's not clear what operation is intended with "s," and "x/1" by itself is simply equal to "x" since dividing any number by 1 results in the same number.

On the other hand, "1/x" is a valid mathematical expression. It represents the reciprocal or multiplicative inverse of the variable "x." In mathematical notation, "1/x" is used to denote "one divided by x." This expression is meaningful as long as "x" is not equal to zero because division by zero is undefined in mathematics.

So, "1/x" is a valid mathematical expression, while "s x/1" lacks clarity and is not standard mathematical notation.

4. Compute the values (x+y)**2 and x**2+x*y+y*x+y**2. Are they the same when x and y are arrays?

In [34]:

```
import numpy as np

# Define arrays x and y
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

# Calculate x**2 + 2*x*y + y**2
result3 = x**2 + 2*x*y + y**2

print("x**2 + 2*x*y + y**2 =", result3)
#x**2 + 2*x*y + y**2 = [25 64 121]
```

```
x**2 + 2*x*y + y**2 = [25 49 81]
```

5. Is x**2+2*x*y+y**2 the same as any of the above?

In [35]:

```
import numpy as np

# Define arrays x and y
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

# Calculate x**2 + 2*x*y + y**2
result3 = x**2 + 2*x*y + y**2

print("x**2 + 2*x*y + y**2 =", result3)
```

```
x**2 + 2*x*y + y**2 = [25 49 81]
```

In [37]:

```
 #6.For conformable arrays, is a*b+a*c the same as a*b+c? If so, show with an example. If
#second be changed so they are equal?

import numpy as np

# Define conformable arrays a, b, and c
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.array([7, 8, 9])

# Calculate a*b + a*c
result1 = a * b + a * c

# Calculate a*b + c
result2 = a * b + c

print("a*b + a*c =", result1)
print("a*b + c =", result2)
```

```
a*b + a*c = [11 26 45]
a*b + c = [11 18 27]
```

7. Suppose a command x**y*w+z was entered. What restrictions on the dimensions of w, x, y and z must be true for this to be a valid statement?

In [38]:

```
import numpy as np

# Define arrays x, y, w, and z with compatible shapes
x = np.array([[2, 3], [4, 5]])
y = np.array([[1, 2], [3, 4]])
w = np.array([[0.5, 0.5], [0.25, 0.25]])
z = np.array([[0, 1], [2, 3]])

# Calculate x**y**w + z
result = x**y**w + z

print("Result of x**y**w + z:")
print(result)
```

```
Result of x**y**w + z:
[[ 2.          5.72880439]
 [ 8.19948351 12.73851774]]
```

. What is the value of -2**4? What about (-2)**4? What about -2*-2*-2*-2?

Let's evaluate each of the expressions:

1. **-2**4**: In most programming languages and mathematical notation, the double asterisk (`**`) represents exponentiation. So, `-2**4` means raising -2 to the power of 4.

   -2**4 = -2^4 = -16

   So, the value of `-2**4` is -16.
2. **(-2)**4**: In this expression, the parentheses clarify the order of operations, and it means raising -2 to the power of 4.

   (-2)**4 = (-2)^4 = 16

   So, the value of `(-2)**4` is 16.
3. **-2*-2*-2*-2**: This expression involves the multiplication of -2 four times, and since multiplication is associative, it doesn't matter how you group the multiplications.

   -2*-2*-2*-2 = (-2 * -2) * (-2 * -2) = (4) * (4) = 16

   So, the value of `-2*-2*-2*-2` is 16.

In summary:

- **-2**4** = -16
- **(-2)**4** = 16
- **-2*-2*-2*-2** = 16

Thanku

In [ ]:

```

```