Load in NumPy

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

Initializing Different Types of Arrays ¶

In [2]:
```
# All 0s matrix
np.zeros((2,3))
```

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

In [3]:
```
# All 1s matrix
np.ones((4,2,2), dtype='int32')
```

Out[3]:
```
array([[[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]],

       [[1, 1],
        [1, 1]]])
```

In [4]:
```
# Any other number
np.full((2,2), 99)
```

Out[4]:
```
array([[99, 99],
       [99, 99]])
```

In [7]:
```
# Any other number (full_like)
a = np.array([[1,2,3,4,5,6,7],[8,9,10,11,12,13,14]])
np.full_like(a, 4)
```

Out[7]:
```
array([[4, 4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4, 4]])
```

In [8]:
```
# Random decimal numbers
np.random.rand(4,2)
```

Out[8]:
```
array([[0.07414044, 0.86980098],
       [0.47974903, 0.05263716],
       [0.47460294, 0.34549138],
       [0.52060104, 0.3443335 ]])
```

In [9]:
```
# Random Integer values
np.random.randint(-4,8, size=(3,3))
```

Out[9]:
```
array([[ 6,  6, -3],
       [ 7,  7, -1],
       [-2,  5,  2]])
```

In [10]:
```
# The identity matrix
np.identity(5)
```

Out[10]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [11]:
```
# Repeat an array
arr = np.array([[1,2,3]])
r1 = np.repeat(arr,3, axis=0)
print(r1)
```

```
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

In [6]:
```
output = np.ones((5,5))
print(output)

z = np.zeros((3,3))
z[1,1] = 9
print(z)

output[1:-1,1:-1] = z
print(output)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
[[0. 0. 0.]
 [0. 9. 0.]
 [0. 0. 0.]]
[[1. 1. 1. 1. 1.]
 [1. 0. 0. 0. 1.]
 [1. 0. 9. 0. 1.]
 [1. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1.]]
```

Mathematics

In [12]:
```
a = np.array([1,2,3,4])
print(a)
```

```
[1 2 3 4]
```

In [13]:
```
a + 2
```

Out[13]:
```
array([3, 4, 5, 6])
```

In [14]:
```
a - 2
```

Out[14]:
```
array([-1,  0,  1,  2])
```

In [15]:
```
a * 2
```

Out[15]:
```
array([2, 4, 6, 8])
```

In [16]:
```
a / 2
```

Out[16]:
```
array([0.5, 1. , 1.5, 2. ])
```

In [17]:
```
b = np.array([1,0,1,0])
a + b
```

Out[17]:
```
array([2, 2, 4, 4])
```

In [18]:
```
a ** 2
```

Out[18]:
```
array([ 1,  4,  9, 16], dtype=int32)
```

In [19]:
```
# Take the sin
np.cos(a)
```

Out[19]:
```
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
```

In [20]:
```
# For a lot more (https://docs.scipy.org/doc/numpy/reference/routines.math.html)
```

Linear Algebra

In [21]:
```
a = np.ones((2,3))
print(a)

b = np.full((3,2), 2)
print(b)

np.matmul(a,b)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
[[2 2]
 [2 2]
 [2 2]]
```

Out[21]:
```
array([[6., 6.],
       [6., 6.]])
```

In [22]:
```
# Find the determinant
c = np.identity(3)
np.linalg.det(c)
```

Out[22]:
```
1.0
```

In [23]:
```
## Reference docs (https://docs.scipy.org/doc/numpy/reference/routines.linalg.html)

# Determinant
# Trace
# Singular Vector Decomposition
# Eigenvalues
# Matrix Norm
# Inverse
# Etc...
```

Statistics

In [24]:
```
stats = np.array([[1,2,3],[4,5,6]])
stats
```

Out[24]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [25]:
```
np.min(stats)
```

Out[25]:
```
1
```

In [26]:
```
np.max(stats, axis=1)
```

Out[26]:
```
array([3, 6])
```

In [27]:
```
np.sum(stats, axis=0)
```

Out[27]:
```
array([5, 7, 9])
```

Reorganizing Arrays

In [33]:
```
before = np.array([[1,2,3,4],[5,6,7,8]])
print(before)

after = before.reshape((2,4))
print(after)
```

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

In [34]:
```
# Vertically stacking vectors
v1 = np.array([1,2,3,4])
v2 = np.array([5,6,7,8])

np.vstack([v1,v2,v1,v2])
```

Out[34]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8],
       [1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [35]:
```
# Horizontal  stack
h1 = np.ones((2,4))
h2 = np.zeros((2,2))

np.hstack((h1,h2))
```

Out[35]:
```
array([[1., 1., 1., 1., 0., 0.],
       [1., 1., 1., 1., 0., 0.]])
```

次の行列のコードを埋める

[[0 2 4 6 8] [10 12 14 16 18] [20 22 24 26 28] [30 32 34 36 38]]

次の部分を抽出するコード

[12 14] [22 24]

[[2 4 6] [12 14 16] [22 24 26] [32 34 36]]

In [ ]:
```
array2D =
print(array2D) # shape : (4,5)
'''output
[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]
 [30 32 34 36 38]]
'''
```

In [ ]:
```
sliced_array_1 =          # look we set 1:3 <- go 1 to 2 but not include 3
print(sliced_array_1)
'''output
[[12 14]
 [22 24]]
'''
```

In [ ]:
```
sliced_array_2 =          # The 'bare' slice [:] will asign to all values in an array
print(sliced_array_2)
'''output
[[ 2  4  6]
 [12 14 16]
 [22 24 26]
 [32 34 36]]
'''
```

In [36]:
```
'''
More practice. array2D:

[[ 0  2  4  6  8]
 [10 12 14 16 18]
 [20 22 24 26 28]
 [30 32 34 36 38]]

Let's get some specific portion.

1. [16 18],
   [26 28]

2. [20 22 24],
   [30 32 34]

3. [14 16],
   [24 26],
   [34 36]
'''
```

Out[36]:
```
"¥nMore practice. array2D:¥n¥n[[ 0  2  4  6  8]¥n [10 12 14 16 18]¥n [20 22 24 26 28]¥n [30 32 34
36 38]]¥n¥nLet's get some specific portion.¥n¥n1. [16 18],¥n   [26 28]¥n¥n2. [20 22 24],¥n   [30
32 34] ¥n   ¥n3. [14 16],¥n   [24 26],¥n   [34 36]¥n"
```

In [ ]:
```
print('----------------------------------------------')

sliced_array_4 =          # row: 1 to 2 ; column: 3 to all
print('1 ¥n', sliced_array_4)
'''output
1
 [[16 18]
 [26 28]]
'''


print('----------------------------------------------')

sliced_array_5 =            # row: 2 to all ; column: 0 to 2
print('2 ¥n', sliced_array_5)
'''output
2
 [[20 22 24]
 [30 32 34]]
'''


print('----------------------------------------------')

sliced_array_6 =            # row: 1 to all ; column: 2 to 3
print('3 ¥n', sliced_array_6)
'''output
3
 [[14 16]
 [24 26]
 [34 36]]
'''
```

In [ ]:
```
x = np.arange(0,4).reshape(2,2).astype('float64')
y = np.arange(5,9).reshape(2,2).astype('float64')
```

In [ ]:
```
# Elementwise sum; both produce the array

# ここにコードを追加する

'''output
[[ 5.  7.]
 [ 9. 11.]]
[[ 5.  7.]
 [ 9. 11.]]
'''

print('----------------------------')

# Elementwise difference; both produce the array
# ここにコードを追加する


[[-5. -5.]
 [-5. -5.]]
[[-5. -5.]
 [-5. -5.]]
'''



print('----------------------------')

# Elementwise product; both produce the array
# ここにコードを追加する

'''output
[[ 0.  6.]
 [14. 24.]]
[[ 0.  6.]
 [14. 24.]]
'''

print('----------------------------')

# Elementwise division; both produce the array
# ここにコードを追加する


'''output
[[0.         0.16666667]
 [0.28571429 0.375     ]]
[[0.         0.16666667]
 [0.28571429 0.375     ]]
'''

print('----------------------------')

# Elementwise square root; produces the array
# ここにコードを追加する


'''output
[[0.         1.        ]
 [1.41421356 1.73205081]]
'''
```

計算

In [37]:
```
'''
We use the dot function to compute inner products of vectors, to multiply a vector
by a matrix, and to multiply matrices. dot is available both as a function in the numpy
module and as an instance method of array objects. Note that, * is elementwise
multiplication, not matrix multiplication.
'''

x = np.arange(0,4).reshape(2,2).astype('float64')
y = np.arange(5,9).reshape(2,2).astype('float64')

v = np.array([5,1])
w = np.array([3, 4])

# Inner product of vectors; both produce 19
print(v.dot(w))     # 19
print(np.dot(v, w)) # 19

print('----------------------------')

# Matrix / vector product; both produce the rank 1 array [1 13]
print(x.dot(v))
print(np.dot(x, v))
# [ 1.  13.]
# [ 1.  13.]

print('----------------------------')

# Matrix / matrix product; both produce the rank 2 array
print(x.dot(y))
print(np.dot(x, y))
# [[ 7.  8.]
# [31.  36.]]

# [[ 7.  8.]
# [31.  36.]]
```

```
19
19
----------------------------
[ 1.  13.]
[ 1.  13.]
----------------------------
[[ 7.  8.]
 [31.  36.]]
[[ 7.  8.]
 [31.  36.]]
```

In [38]:
```
'''
Numpy provides many useful functions for performing computations on arrays;
one of the most useful is sum
'''

import numpy as np

x = np.arange(5,9).reshape(2,2).astype('int64')

print(x)
print(np.sum(x))          # Compute sum of all elements; prints "26"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[12 14]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[11 15]"
```

```
[[5 6]
 [7 8]]
26
[12 14]
[11 15]
```

In [39]:
```
'''
Sometimes we need to manipulate the data in array. It can be done by reshaping or transpose
the array. Transposing is a special form of reshaping that similarly returns a view on the
underlying data without copying anything.

When doing matrix computation, we may do this very often.
'''


arr = np.arange(10).reshape(2,5)
print('At first ¥n', arr)    # At first
                             #  [[0 1 2 3 4]
                             #  [5 6 7 8 9]]
print()
print('After transpose ¥n', arr.T)
# After transpose
#  [[0 5]
#  [1 6]
#  [2 7]
#  [3 8]
#  [4 9]]

print('----------------------------')
transpose = np.arange(10).reshape(2,5)
print(np.dot(transpose.T, transpose)) # (5x2).(2,5) | matrix multiplication
# [[25 30 35 40 45]
#  [30 37 44 51 58]
#  [35 44 53 62 71]
#  [40 51 62 73 84]
#  [45 58 71 84 97]]
```

```
At first
 [[0 1 2 3 4]
 [5 6 7 8 9]]

After transpose
 [[0 5]
 [1 6]
 [2 7]
 [3 8]
 [4 9]]
----------------------------
[[25 30 35 40 45]
 [30 37 44 51 58]
 [35 44 53 62 71]
 [40 51 62 73 84]
 [45 58 71 84 97]]
```

In [40]:
```
'''
statistical functions and concern used function, such as

- mean
- min
- sum
- std
- median
- argmin, argmax
'''

ary = 10 * np.random.randn(2,5)
print('Mean   : ', np.mean(ary))      # 0.9414738037734729
print('STD    : ', np.std(ary))       # 5.897885490589387
print('Median : ', np.median(ary))    # 1.5337461352996276
print('Argmin : ', np.argmin(ary))    # 3
print('Argmax : ', np.argmax(ary))    # 2
print('Max    : ', np.max(ary))       # 10.399663734487659
print('Min    : ', np.min(ary))       # -9.849839643044087
print('Compute mean by column :', np.mean(ary, axis = 0)) # compute the means by column
# Compute mean by column : [-1.61785449  0.4643941   8.24396096 -6.32814416  3.94501261]
print('Compute median by row  : ', np.median(ary, axis = 1)) # compute the medians
# Compute median by row  :  [2.7236133  0.34387897]
```

```
Mean   :  -0.679657984378417
STD    :  8.033388378685286
Median :  -1.6923172384452863
Argmin :  3
Argmax :  9
Max    :  10.135282228062792
Min    :  -15.299614931730904
Compute mean by column : [-0.24995393  3.27950468 -0.88348233 -9.74916102  4.20480269]
Compute median by row  :  [-7.74235855  6.47579622]
```

In [41]:

```
ary = np.arange(5)

print('Find root of each elements-wise ¥n', np.sqrt(ary))
print()
print('Find exponential for each element-wise ¥n', np.exp(ary))

# Find root of each elements-wise
#  [0.         1.         1.41421356 1.73205081 2.        ]

# Find exponential for each element-wise
#  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]


'''
np.maximum

This computed the element-wise maximum of the elements in two array and returned a single
array as a result.
'''
print()
print('Max values between two array ¥n', np.maximum(np.sqrt(ary), np.exp(ary)))
print('----------------------------')
print()
# Max values between two array
#  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]


'''
np.modf

Another unfunc but can return multiple arrays. It returns the fractional and integral parts
of a floating point array.
'''
rem, num = np.modf(np.exp(ary))
print('Floating Number ', np.exp(ary))
print()
print('Remainder       ', rem)
print('Number          ', num)
print('----------------------------')
print()

# Floating Number  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]

# Remainder        [0.         0.71828183 0.3890561  0.08553692 0.59815003]
# Number           [ 1.  2.  7. 20. 54.]



'''
np.ceil

Return the ceiling of the input, element-wise.
'''
ceil_num = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
print(ceil_num)               # [-1.7 -1.5 -0.2  0.2  1.5  1.7  2. ]
print(np.ceil(ceil_num))      # [-1. -1. -0.  1.  2.  2.  2.]


''' not ufunc

np.around

Evenly round to the given number of decimals.
'''
print(np.around(ceil_num))    # [-2. -2. -0.  0.  2.  2.  2.]
print('----------------------------')
print()

'''
```

```
np.absolute | np.abs | np.fabs

Calculate the absolute value element-wise.
'''
absl = np.array([-1, 2])
print('Absolute of Real Values              ', np.abs(absl))
print('Absolute of Real Values with Float   ', np.fabs(absl))
print('Absolute of Complex Values           ', np.abs(1.2 + 1j))

# Absolute of Real Values               [1 2]
# Absolute of Real Values with Float    [1. 2.]
# Absolute of Complex Values           1.5620499351813308
```

```
Find root of each elements-wise
 [0.         1.         1.41421356 1.73205081 2.        ]

Find exponential for each element-wise
 [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]

Max values between two array
 [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
----------------------------

Floating Number  [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]

Remainder        [0.         0.71828183 0.3890561  0.08553692 0.59815003]
Number           [ 1.  2.  7. 20. 54.]
----------------------------

[-1.7 -1.5 -0.2  0.2  1.5  1.7  2. ]
[-1. -1. -0.  1.  2.  2.  2.]
[-2. -2. -0.  0.  2.  2.  2.]
----------------------------

Absolute of Real Values              [1 2]
Absolute of Real Values with Float   [1. 2.]
Absolute of Complex Values          1.5620499351813308
```

NumPy Random

- np.random.rand
- np.random.randn
- np.random.random
- np.random.random_sample
- np.random.randint
- np.random.normal
- np.ranodm.uniform
- np.random.seed
- np.random.shuffle
- np.random.choice

In [42]:

```
'''
np.random.rand()

Create an array of the given shape and populate it with
random samples from a uniform distribution
over ``[0, 1)``
'''
ary = np.random.rand(5,2) # shape: 5 row, 2 column
print('np.random.rand() ¥n', ary)
print('----------------------------')
print()

'''
np.random.randn

Return a sample (or samples) from the "standard normal" distribution.
'''
ary = np.random.randn(6)
print('1D array: np.random.randn() ¥n', ary)
ary = np.random.randn(3,3)
print('2D array: np.random.randn() ¥n', ary)
print('----------------------------')
print()

'''
np.random.random.
numpy.random.random() is actually an alias for numpy.random.random_sample()

Return a sample (or samples) from the "standard normal" distribution.
'''
ary = np.random.random((3,3))
print('np.random.randn() ¥n', ary)
ary = np.random.random_sample((3,3))
print('np.random.random_sample() ¥n', ary)
print('----------------------------')
print()

'''
np.random.randint.
Return random integers from low (inclusive) to high (exclusive)

Return random integers from the "discrete uniform" distribution of the specified
dtype in the "half-open" interval [low, high). If high is None (the default),
then results are from [0, low).

'''
ary = np.random.randint(low = 2, high = 6, size = (5,5))
print('np.random.randint() ¥n', ary)
ary = np.random.randint(low = 2, high = 6)
print('np.random.randint() :', ary)
```

```
np.random.rand()
 [[0.54648333 0.20709696]
 [0.97998772 0.46350266]
 [0.31343068 0.62973809]
 [0.52377181 0.36621881]
 [0.2905816  0.95856425]]
----------------------------

1D array: np.random.randn()
 [ 0.56704536  0.82921394  0.80237169 -1.00284039  0.31573988 -0.15876032]
2D array: np.random.randn()
 [[-0.45379241 -0.30652257 -0.2839688 ]
 [-1.71323422 -0.1346359   0.0352139 ]
 [-0.7267943   0.06126749 -0.49705862]]
----------------------------

np.random.randn()
 [[0.66051266 0.50338025 0.21100639]
 [0.86427576 0.18533315 0.34386328]
 [0.47476364 0.18992232 0.30236803]]
np.random.random_sample()
 [[0.55353606 0.01528067 0.45569839]
 [0.62188778 0.38079402 0.78642526]
 [0.91478793 0.28644167 0.10811371]]
----------------------------

np.random.randint()
 [[5 5 3 3 2]
 [5 4 2 2 2]
 [2 2 5 5 2]
 [5 5 5 2 2]
 [2 3 4 5 4]]
np.random.randint() : 3
```

In [43]:

```
'''
np.random.normal()

Draw random samples from a normal (Gaussian) distribution. This is Distribution is
also known as Bell Curve because of its characteristics shape.
'''
mu, sigma = 0, 0.1 # mean and standard deviation
print('np.random.normal() ¥n',np.random.normal(mu, sigma, 10)) # from doc
print('---------------------------')
print()

'''
np.random.uniform()

Draw samples from a uniform distribution
'''
print('np.random.uniform() ¥n', np.random.uniform(-1,0,10))
print('---------------------------')
print()

'''
np.random.seed()
'''
np.random.seed(3) # seed the result

'''
np.random.shuffle

Modify a sequence in-place by shuffling its contents
'''

ary = np.arange(9).reshape((3, 3))
print('Before Shuffling ¥n', ary)

np.random.shuffle(ary)
print('After Shuffling ¥n', ary)
print('---------------------------')
print()

'''
np.random.choice

Generates a random sample from a given 1-D array
'''
ary = np.random.choice(5, 3) # Generate a uniform random sample from np.arange(5) of size 3:
print('np.random.choice() ¥n', ary) # This is equivalent to np.random.randint(0,5,3)
```

```
np.random.normal()
 [-0.02468732  0.12023723 -0.03496799 -0.12221528  0.13646832  0.20079117
 -0.07322038 -0.16530129 -0.07127619  0.13399058]
----------------------------

np.random.uniform()
 [-0.59145195 -0.16163536 -0.68906276 -0.73108742 -0.83727574 -0.96405742
 -0.36250631 -0.93180379 -0.77081801 -0.01137883]
----------------------------

Before Shuffling
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
After Shuffling
 [[3 4 5]
 [0 1 2]
 [6 7 8]]
----------------------------

np.random.choice()
 [1 3 0]
```

Some used functions:

- sort()
- unique()
- vstack() and hstack()
- ravel()
- tile()
- concatenate()

In [44]:

```
'''
sort()
'''
# create a 10 element array of randoms
unsorted = np.random.randn(10)
print('Unsorted ¥n', unsorted)

# inplace sorting
unsorted.sort()
print('Sorted ¥n', unsorted)

print()
print('-----------------------------------')


'''
unique()
'''
ary = np.array([1,2,1,4,2,1,4,2])
print('Unique values : ', np.unique(ary))
print()
print('-----------------------------------')


'''
vstack and hstack
'''
arx = np.array([[1,2,3],[3,4,5]])
ary = np.array([[4,5,6],[7,8,9]])
print('Vertical Stack ¥n', np.vstack((arx,ary)))
print('Horizontal Stack ¥n', np.hstack((arx,ary)))
print('Concate along columns ¥n', np.concatenate([arx, ary], axis = 0)) # similar vstack
print('Concate along rows ¥n', np.concatenate([arx, ary], axis = 1))    # similar hstack
print()
print('-----------------------------------')


'''
ravel :  convert one numpy array into a single column
'''
ary = np.array([[1,2,3],[3,4,5]])
print('Ravel ¥n', ary.ravel())
print()
print('-----------------------------------')


'''
tile()
'''
ary = np.array([-1, 0, 1])
ary_tile = np.tile(ary, (4, 1)) # Stack 4 copies of v on top of each other
print('tile array ¥n', ary_tile)
```

```
Unsorted
 [ 0.1841282  -1.00595517 -0.34198034 -0.04472413  0.27844092 -0.58089402
 -0.15151488 -1.14743417 -0.61100002 -1.18951737]
Sorted
 [-1.18951737 -1.14743417 -1.00595517 -0.61100002 -0.58089402 -0.34198034
 -0.15151488 -0.04472413  0.1841282   0.27844092]

-----------------------------------
Unique values :  [1 2 4]

-----------------------------------
Vertical Stack
 [[1 2 3]
 [3 4 5]
 [4 5 6]
 [7 8 9]]
Horizontal Stack
 [[1 2 3 4 5 6]
 [3 4 5 7 8 9]]
Concate along columns
 [[1 2 3]
 [3 4 5]
 [4 5 6]
 [7 8 9]]
Concate along rows
 [[1 2 3 4 5 6]
 [3 4 5 7 8 9]]

-----------------------------------
Ravel
 [1 2 3 3 4 5]

-----------------------------------
tile array
 [[-1  0  1]
 [-1  0  1]
 [-1  0  1]
 [-1  0  1]]
```

In [45]:
```python
# Set Function

s1 = np.array(['desk','chair','bulb'])
s2 = np.array(['lamp','bulb','chair'])
print(s1, s2)

print( np.intersect1d(s1, s2) )
print( np.union1d(s1, s2) )
print( np.setdiff1d(s1, s2) )# elements in s1 that are not in s2
print( np.in1d(s1, s2) ) # which element of s1 is also in s2
```

```
['desk' 'chair' 'bulb'] ['lamp' 'bulb' 'chair']
['bulb' 'chair']
['bulb' 'chair' 'desk' 'lamp']
['desk']
[False  True  True]
```

Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array

In [46]:

```
start = np.zeros((4,3))
print(start)

'''output
[[0.  0.  0.]
 [0.  0.  0.]
 [0.  0.  0.]
 [0.  0.  0.]]
'''


# create a rank 1 ndarray with 3 values
add_rows = np.array([1,  0,  2])

y = start + add_rows  # add to each row of 'start' using broadcasting
print(y)

'''output
[[1.  0.  2.]
 [1.  0.  2.]
 [1.  0.  2.]
 [1.  0.  2.]]
'''


# create an ndarray which is 4 x 1 to broadcast across columns
add_cols = np.array([[0,1,2,3]])
add_cols = add_cols.T

print(add_cols)
'''output
[[0]
 [1]
 [2]
 [3]]
'''


# add to each column of 'start' using broadcasting
y = start + add_cols
print(y)

'''output
[[0.  0.  0.]
 [1.  1.  1.]
 [2.  2.  2.]
 [3.  3.  3.]]
'''


# this will just broadcast in both dimensions
add_scalar = np.array([1])
print(start + add_scalar)
'''output
[[1.  1.  1.]
 [1.  1.  1.]
 [1.  1.  1.]
 [1.  1.  1.]]
'''
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[1. 0. 2.]
 [1. 0. 2.]
 [1. 0. 2.]
 [1. 0. 2.]]
[[0]
 [1]
 [2]
 [3]]
[[0. 0. 0.]
 [1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Out[46]: 'output¥n[[1. 1. 1.]¥n [1. 1. 1.]¥n [1. 1. 1.]¥n [1. 1. 1.]]¥n'

2つの配列(Arrays)を一緒にブロードキャストすることは、次のルールに従います。

- 配列のランクが同じでない場合は、両方の形状の長さが同じになるまで、下位の配列の形状に1を付加します。
- 2つの配列は、次元のサイズが同じである場合、または配列の1つがその次元のサイズ1である場合、次元で互換性があると言われます。
- すべての次元で互換性がある場合、配列を一緒にブロードキャストできます。
- ブロードキャスト後、各配列は、2つの入力配列の形状の要素ごとの最大値に等しい形状を持っているかのように動作します。
- 一方の配列のサイズが1で、もう一方の配列のサイズが1より大きい任意の次元では、最初の配列はその次元に沿ってコピーされたかのように動作します

In [ ]: