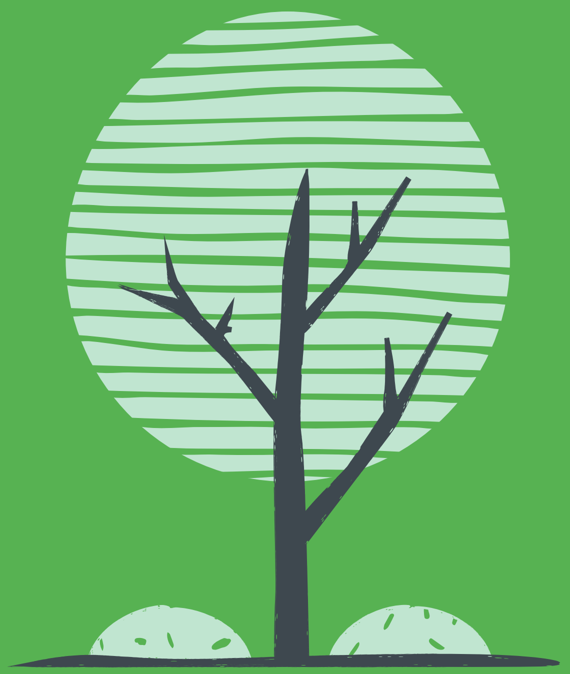
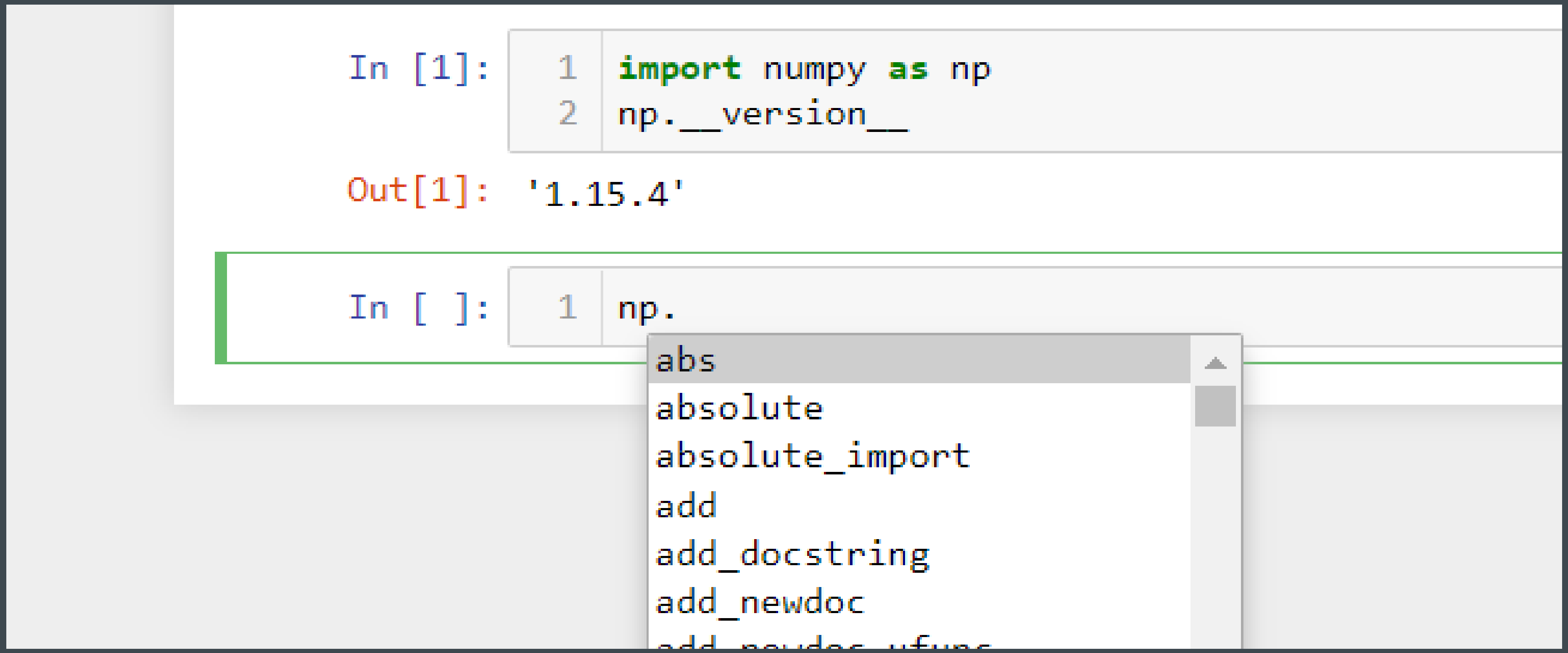


Introduction to Numpy

- Python的資料型態
- NumPy陣列基礎
- Universal Functions
- 聚合操作:Min、Max及其他
- 在陣列上的計算:Broadcasting

Numerical Python

Numpy的陣列就像是Python內建的list，但是當陣列很大時，Numpy提供更有效率的儲存和工作



Python的資料型態

```
/* C code */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

While in Python the equivalent operation could be

```
# Python code
result = 0
for i in range(100):
    result += i
```

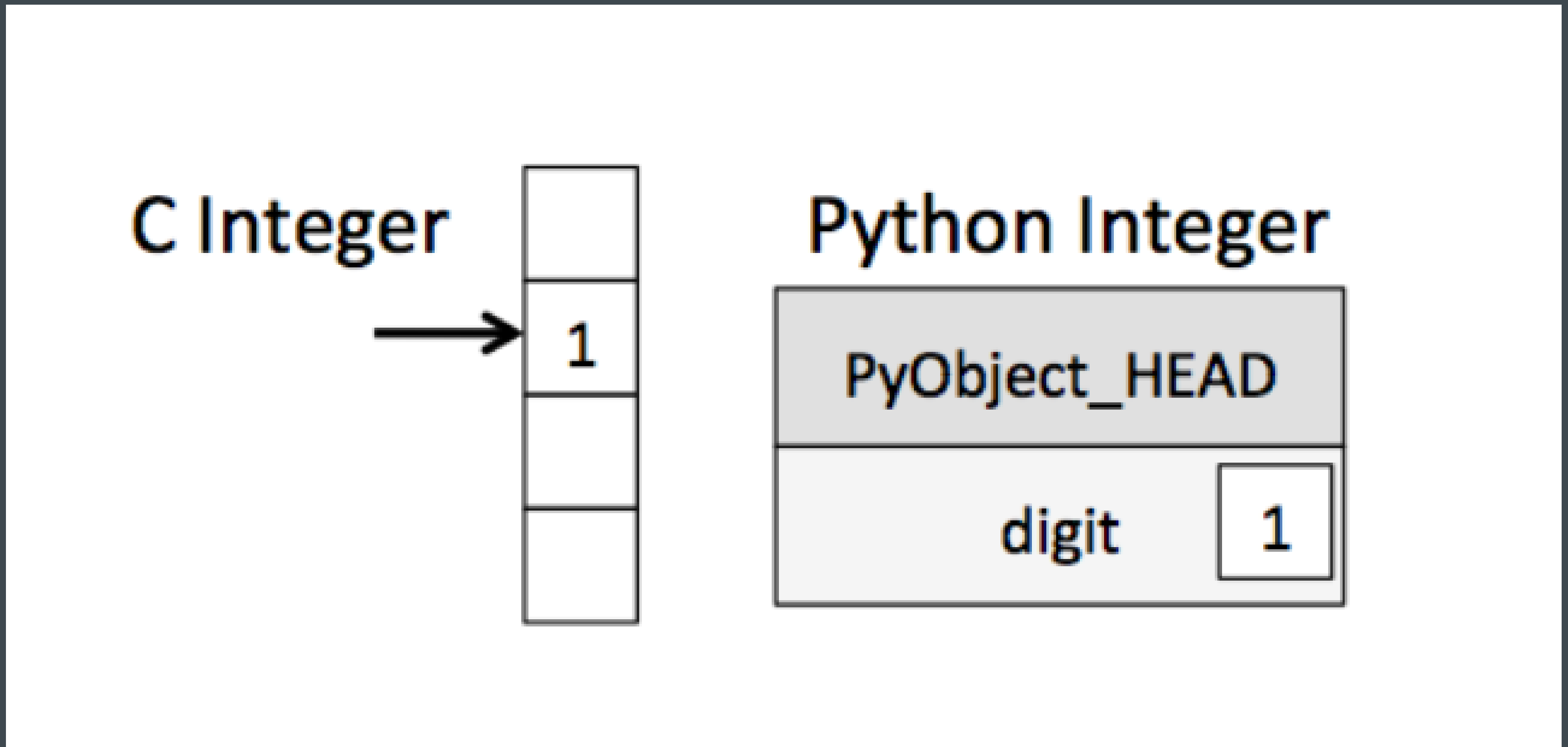
```
# Python code
x = 4
x = "four"
```

Here we've switched the contents of `x` from an in
thing in C would lead (depending on compiler sett
other unintented consequences:

```
/* C code */
int x = 4;
x = "four";  // FAILS
```

Python的型態是動態推導的，也就是Python的變數不會只儲存值，必須包含關於型別的額外資訊

Python的資料型態



Python是以C語言寫成，表示每一個Python物件都是一個精巧設計的C結構，因此Python相對來說在儲存整數時多了一些額外的負擔。這些額外的資訊讓Python可以自由及動態的編寫程式碼，然而也需要付出成本

Python的資料型態

```
In [5]:  L3 = [True, '2', 3.0, 4]
         [type(item) for item in L3]

Out[5]: [bool, str, float, int]
```

Python的list中每個項目都必須包含它的型態資訊、參考計數和其他的資訊，即每個項目都是完整的Python物件，然而當大部分的資料型態相同時，大部分的資訊都是多餘的

```
In [8]:  import array #python 3.3以後內建
         A = array.array('i',L)
         A
         #'i' means following content's type is interger

Out[8]: array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Python的資料型態

```
In [9]:
np.array([3.14,2,3])
#統一被轉成float

Out[9]: array([3.14, 2.  , 3.  ])
```

```
In [10]:
#numpy.ndarray
np.array([1,2,3,4],dtype='float32')

Out[10]: array([1., 2., 3., 4.], dtype=float32)
```

```
In [11]:
np.array([range(i,i+3) for i in [2,4,6]])

Out[11]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

```
In [3]:
np.array([1,4,2])

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

```
In [4]:
np.array([2,3,'hello'])

Out[4]: array(['2', '3', 'hello'], dtype='<U11')

In [7]:
a = np.array([2,3,'hello'])
a[0]+a[1]

Out[7]: '23'
```

NumPy限制所有陣列裡的內容需轉換成同樣的Type，如果不符合Numpy會自動將其轉換型態。也可以用dtype設定明確的資料型態

Python的資料型態

In [12]:
```python
# Create a length-10 integer array filled with zeros
np.zeros(10, dtype=int)
```

Out[12]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [13]:
```python
# Create a 3x5 floating-point array filled with ones
np.ones((3, 5), dtype=float)
```

Out[13]:
```
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

In [14]:
```python
# Create a 3x5 array filled with 3.14
np.full((3, 5), 3.14)
```

Out[14]:
```
array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

Python的資料型態

```
In [15]: # Create an array filled with a linear sequence
         # Starting at 0, ending at 20, stepping by 2
         # (this is similar to the built-in range() function)
         np.arange(0, 20, 2)

Out[15]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [16]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)

Out[16]: array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

In [17]: # Create a 3x3 array of uniformly distributed
         # random values between 0 and 1
         np.random.random((3, 3))

Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
                [ 0.08007488,  0.45429293,  0.20941444],
                [ 0.14360941,  0.96910973,  0.946117  ]])
```

Python的資料型態

```
In [17]: # Create a 3x3 array of uniformly distributed
         # random values between 0 and 1
         np.random.random((3, 3))

Out[17]: array([[ 0.99844933,  0.52183819,  0.22421193],
                [ 0.08007488,  0.45429293,  0.20941444],
                [ 0.14360941,  0.96910973,  0.946117  ]])

In [18]: # Create a 3x3 array of normally distributed random values
         # with mean 0 and standard deviation 1
         np.random.normal(0, 1, (3, 3))

Out[18]: array([[ 1.51772646,  0.39614948, -0.10634696],
                [ 0.25671348,  0.00732722,  0.37783601],
                [ 0.68446945,  0.15926039, -0.70744073]])

In [19]: # Create a 3x3 array of random integers in the interval [0, 10)
         np.random.randint(0, 10, (3, 3))

Out[19]: array([[2, 3, 4],
                [5, 7, 8],
                [0, 5, 0]])
```

Python的資料型態

```
In [20]: # Create a 3x3 identity matrix
         np.eye(3)

Out[20]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In [21]: # Create an uninitialized array of three
         # The values will be whatever happens to
         np.empty(3)

Out[21]: array([ 1.,  1.,  1.])
```

```
In [8]:  1 %memit np.ones(1000000)

peak memory: 70.19 MiB, increment: 6.57 MiB

In [9]:  1 %memit np.empty(1000000)

peak memory: 63.71 MiB, increment: 0.00 MiB

n [10]:  1 %memit np.zeros(1000000)

peak memory: 63.72 MiB, increment: 0.00 MiB
```

Python的資料型態

bool_	布林值(True/False)，以位元組來儲存
int16	整數 (-32768 ~ 32767)
int32	整數 (-2147483648 ~ 2147483647)
int32	整數 (-9223372036854775808 ~ 9223372036854775807)
float16	5位元指數，10位元尾數
float32	8位元指數，23位元尾數
float64	11位元指數，52位元尾數

Numpy陣列基礎

In [1]:
```python
import numpy as np
np.random.seed(0)  # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

In [2]:
```python
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Numpy陣列基礎

```
In [5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])

In [6]: x1[0]
Out[6]: 5
```

```
In [9]: x1[-2]
Out[9]: 7
```

```
In [10]: x2
Out[10]: array([[3, 5, 2, 4],
                [7, 6, 8, 8],
                [1, 6, 7, 7]])

In [11]: x2[0, 0]
Out[11]: 3

In [12]: x2[2, 0]
Out[12]: 1

In [13]: x2[2, -1]
Out[13]: 7
```

Numpy陣列基礎

In [14]:
```
x2[0, 0] = 12
x2
```

Out[14]:
```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [15]:
```
x1[0] = 3.14159  # this will be truncated!
x1
```

Out[15]:
```
array([3, 0, 3, 3, 7, 9])
```

Numpy陣列基礎

A[a:b:c]

a:起始位置，默認0

b:結束位置，默認為size

c:每次步數，默認為1

```
In [16]: x = np.arange(10)
         x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [17]: x[:5]  # first five elements
Out[17]: array([0, 1, 2, 3, 4])

In [18]: x[5:]  # elements after index 5
Out[18]: array([5, 6, 7, 8, 9])

In [19]: x[4:7]  # middle sub-array
Out[19]: array([4, 5, 6])

In [20]: x[::2]  # every other element
Out[20]: array([0, 2, 4, 6, 8])

In [21]: x[1::2]  # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])
```

Question

input = 12345, type = int

output = 54321, type = int

Numpy陣列基礎

In [41]:
```python
ori = 3843158
ori = str(ori)
rev = ori[::-1]
rev = int(rev)
print(rev)
```

```
8513483
```

Numpy陣列基礎

In [16]:
```
print(x[::-1])
#當step是負值時start,step互換
print(x[5:2:-1])
print(x[2:5:-1])
print(x[5::-2])
```

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

Numpy陣列基礎

In [24]:
```
x2
```

Out[24]:
```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

In [25]:
```
x2[:2, :3]  # two rows, three columns
```

Out[25]:
```
array([[12,  5,  2],
       [ 7,  6,  8]])
```

In [26]:
```
x2[:3, ::2]  # all rows, every other column
```

Out[26]:
```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

Finally, subarray dimensions can even be reversed together:

In [27]:
```
x2[::-1, ::-1]
```

Out[27]:
```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

Numpy陣列基礎

```
In [24]:
print(x2)
x2_sub = x2[:2,:2]
print(x2_sub)
```

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

```
In [27]:
x2_sub[0,0] = 77
print(x2)
```

```
[[77  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

```
In [28]:
x2_sub_copy = x2[:2,:2].copy()
#With .copy(), the original value would not be changed
print(x2_sub_copy)
x2_sub_copy[0,0]=66
print(x2_sub_copy)
print(x2)
```

```
[[77  5]
 [ 7  6]]
[[66  5]
 [ 7  6]]
[[77  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

```
In [31]:
a = [2,3,5]
b= a[:2]
print(b)
b[0] = 6
print(a)
print(b)

#python內建的list會直接複製一
```

```
[2, 3]
[2, 3, 5]
[6, 3]
```

Numpy子陣列回傳的是視圖，不是複製一份出來，list則會直接複製子陣列的資料。

Numpy陣列基礎

```
In [38]: grid = np.arange(1, 10).reshape((3, 3))
         print(grid)

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

```
In [53]: x = np.array([1,2,3])
         x.shape

Out[53]: (3,)
```

```
In [41]: # column vector via reshape
         x.reshape((3, 1))

Out[41]: array([[1],
                [2],
                [3]])
```

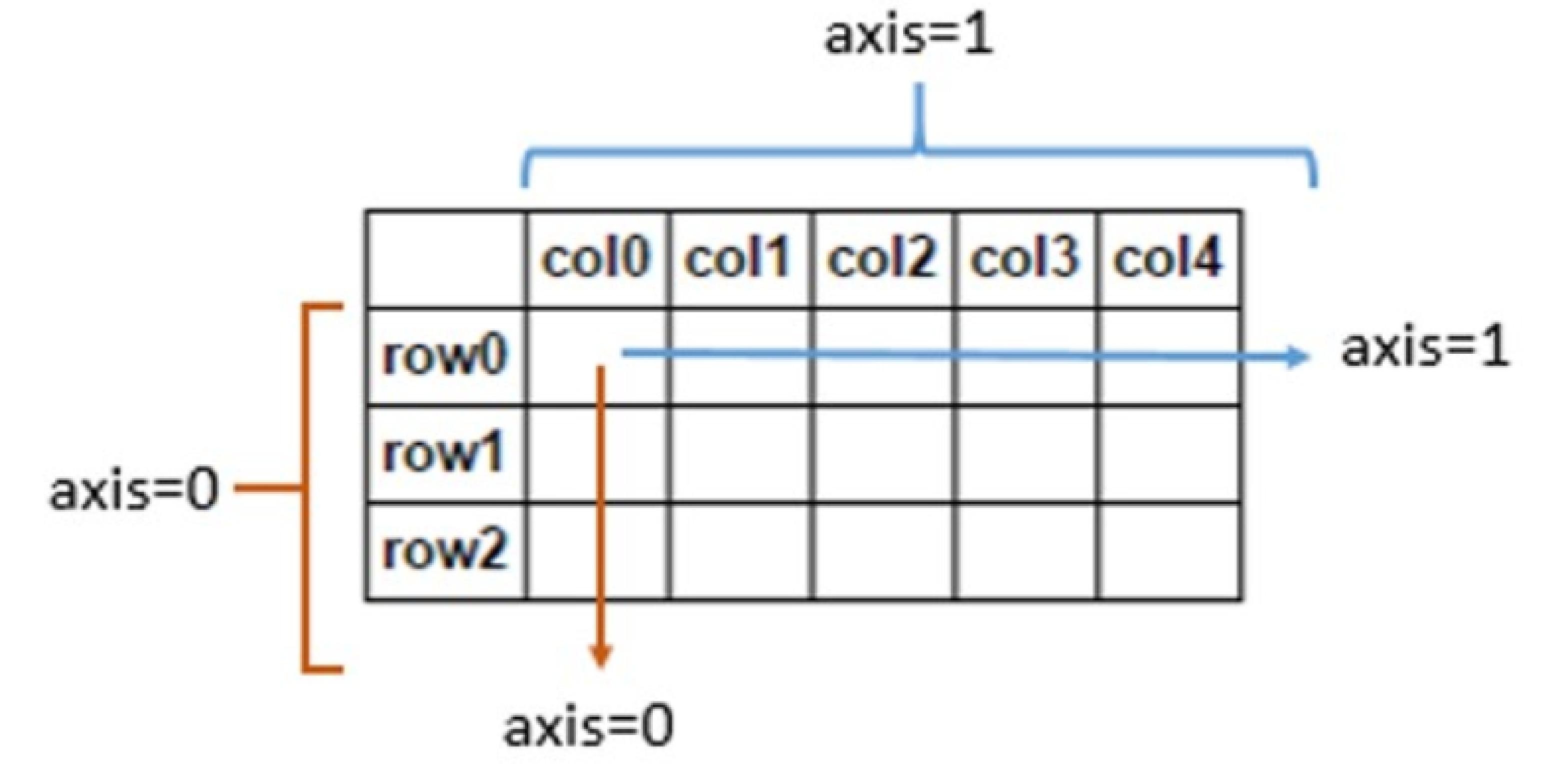
```
In [42]: # column vector via newaxis
         x[:, np.newaxis]

Out[42]: array([[1],
                [2],
                [3]])
```

Numpy陣列基礎

```
In [43]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])

Out[43]: array([1, 2, 3, 3, 2, 1])
```



from: Stack overflow debaonline4u

```
In [45]: grid = np.array([[1, 2, 3],
                          [4, 5, 6]])

In [46]: # concatenate along the first axis
         np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In [47]: # concatenate along the second axis (zer
         np.concatenate([grid, grid], axis=1)

Out[47]: array([[1, 2, 3, 1, 2, 3],
                [4, 5, 6, 4, 5, 6]])
```

Numpy陣列基礎

```
In [48]: x = np.array([1, 2, 3])
         grid = np.array([[9, 8, 7],
                          [6, 5, 4]])

         # vertically stack the arrays
         np.vstack([x, grid])

Out[48]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])

In [49]: # horizontally stack the arrays
         y = np.array([[99],
                       [99]])
         np.hstack([grid, y])

Out[49]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

不同shape的話可以用vstack(vertical)或hstack(horizontal)合併arrays

Numpy陣列基礎

In [51]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

Out[51]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [52]:
```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

In [53]:
```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

Universal Functions

NumPy在陣列中的計算可快可慢，要讓它的快速的關鍵在於使用向量化的操作，
通常都是透過NumPy的universal functions(ufuncs)

In [68]:
```
np.random.seed(0)
def compute_reciprocals(values):
    output = np.empty(len(values))
    for i in range(len(values)):
        output[i] = 1/values[i]
    return output
```

In [69]:
```
big_array = np.random.randint(1,100,size=100000)
%timeit compute_reciprocals(big_array)
```

```
35.9 ms ± 2.87 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [70]:
```
%timeit (1/big_array)
#向量化運算明顯快得多
```

```
283 µs ± 4.72 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

迴圈運算緩慢的原因不是來自於運算本身，而是每次迴圈都必須檢查該物件的type，才會呼叫適合這個type的函式

Note: (1)同一時間執行多次操作，通常是對不同的數據執行同樣的一個或一批指令
(2)記憶體儲存位置相近 (3)資料類型相同

Universal Functions

In [7]:
```python
x = np.arange(4)
print("x     =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2)  # floor division
```

```
x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [ 0.   0.5  1.   1.5]
x // 2 = [0 0 1 1]
```

There is also a unary ufunc for negation, and a `**` operator for

and a `%` operator for modulus:

In [8]:
```python
print("-x     = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2  = ", x % 2)
```

```
-x     =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2  =  [0 1 0 1]
```

`+`	`np.add`
`-`	`np.subtract`
`-`	`np.negative`
`*`	`np.multiply`
`/`	`np.divide`
`//`	`np.floor_divide`
`**`	`np.power`
`%`	`np.mod`

Universal Functions

```
In [11]: x = np.array([-2, -1, 0, 1, 2])
         abs(x)

Out[11]: array([2, 1, 0, 1, 2])
```

```
In [13]: np.abs(x)

Out[13]: array([2, 1, 0, 1, 2])
```

np.sin
np.cos
np.tan
np.arcsin
np.arccos
np.arctan

Universal Functions

```
In [18]: x = [1, 2, 3]
         print("x      =", x)
         print("e^x    =", np.exp(x))
         print("2^x    =", np.exp2(x))
         print("3^x    =", np.power(3, x))
```

```
In [19]: x = [1, 2, 4, 10]
         print("x        =", x)
         print("ln(x)    =", np.log(x))
         print("log2(x)  =", np.log2(x))
         print("log10(x) =", np.log10(x))
```

np.expm1
np.log1p

```
In [3]: x = [0,0.001,0.01,0.1]
        print('exp(x)-1 = ',np.expm1(x))
        print('exp(x)-1 = ',np.exp(x)-1)
        print('\n')
        print('log(1+x) = ',np.log(np.add(1,x)))
        print('log(1+x) = ',np.log1p(x))

exp(x)-1 =  [0.         0.0010005  0.01005017 0.10517092]
exp(x)-1 =  [0.         0.0010005  0.01005017 0.10517092]


log(1+x) =  [0.         0.0009995  0.00995033 0.09531018]
log(1+x) =  [0.         0.0009995  0.00995033 0.09531018]
```

Universal Functions

In [21]:
```
from scipy import special
```

In [22]:
```
# Gamma functions (generalized factorials) and r
x = [1, 5, 10]
print("gamma(x)     =", special.gamma(x))
print("ln|gamma(x)| =", special.gammaln(x))
print("beta(x, 2)   =", special.beta(x, 2))
```

```
gamma(x)     = [  1.00000000e+00   2.40000000e+0
ln|gamma(x)| = [  0.            3.17805383  12.80
beta(x, 2)   = [ 0.5          0.03333333  0.0090
```

Universal Functions

In [24]:
```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

```
[  0.  10.  20.  30.  40.]
```

This can even be used with array views. For example, we ca
computation to every other element of a specified array:

In [25]:
```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
```

```
[  1.   0.   2.   0.   4.   0.   8.   0.  16.   0.]
```

Universal Functions

In [26]:
```
x = np.arange(1, 6)
np.add.reduce(x)
```
Out[26]: 15

Similarly, calling `reduce` on the

elements:

In [27]:
```
np.multiply.reduce(x)
```
Out[27]: 120

In [28]:
```
np.add.accumulate(x)
```
Out[28]: `array([ 1,  3,  6, 10, 15])`

In [29]:
```
np.multiply.accumulate(x)
```
Out[29]: `array([  1,   2,   6,  24, 120])`

Universal Functions

In [30]:
```
x = np.arange(1, 6)
np.multiply.outer(x, x)
```

Out[30]:
```
array([[ 1,  2,  3,  4,  5],
       [ 2,  4,  6,  8, 10],
       [ 3,  6,  9, 12, 15],
       [ 4,  8, 12, 16, 20],
       [ 5, 10, 15, 20, 25]])
```

聚合操作:Min、Max及其他

In [4]:
```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
10 loops, best of 3: 104 ms per loop
1000 loops, best of 3: 442 µs per loop
```

In [7]:
```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
6.91 ms ± 238 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
29.7 µs ± 591 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [6]:
```
%timeit max(-big_array)
%timeit np.max(-big_array)
#np.max(-A)跟np.min的速度還是會有點小差別
```

```
7.07 ms ± 194 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
84.8 µs ± 1.6 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

聚合操作:Min、Max及其他

In [2]:

```python
M = np.random.random((3,4))
print(M)
print('min(M)=',M.min(axis=0))
print('Max(M)=',M.max(axis=1))

#Axis 0 will act on all the ROWS in each COLUMN, MO:每個column
#Axis 1 will act on all the COLUMNS in each ROW

#Axis用來指定陣列中要被收合起來的維度, 而不是要傳回來的那個
```

```
[[0.55210724 0.80814256 0.43532497 0.7307955 ]
 [0.25110438 0.16880296 0.95427731 0.53061535]
 [0.79423024 0.01138292 0.08227069 0.20342719]]
min(M)= [0.25110438 0.01138292 0.08227069 0.20342719]
Max(M)= [0.80814256 0.95427731 0.79423024]
```

聚合操作:Min、Max及其他

```
In [5]:  A = np.array([[[2,3,4],[4,5,6]],[[7,8,9],[10,11,12]]])

In [6]:  sum(A)

Out[6]: array([[ 9, 11, 13],
               [14, 16, 18]])

In [9]:  np.sum(A, axis=0)

Out[9]: array([[ 9, 11, 13],
               [14, 16, 18]])

In [8]:  np.sum(A)

Out[8]: 81
```

```
In [16]: print("25th percentile:   ", np.percentile(heights, 25))
         print("Median:            ", np.median(heights))
         print("75th percentile:   ", np.percentile(heights, 75))

25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

聚合操作:Min、Max及其他

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

聚合操作:Min、Max及其他

In [5]:
```
A = np.array([[[2,3,4],[4,5,6]],[[7,8,9],[10,11,12]]])
```

In [6]:
```
sum(A)
```

Out[6]:
```
array([[ 9, 11, 13],
       [14, 16, 18]])
```

In [9]:
```
np.sum(A, axis=0)
```

Out[9]:
```
array([[ 9, 11, 13],
       [14, 16, 18]])
```

In [8]:
```
np.sum(A)
```

Out[8]:
```
81
```

在陣列上的計算: Broadcasting

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

Out[2]: array([5, 6, 7])

In [3]:
a + 5

Out[3]: array([5, 6, 7])
```

把純量5拉伸成一維向量[5, 5, 5]

```
In [4]:
M = np.ones((3, 3))
M

Out[4]: array([[1., 1., 1.],
               [1., 1., 1.],
               [1., 1., 1.]])

In [5]:
M + a

Out[5]: array([[1., 2., 3.],
               [1., 2., 3.],
               [1., 2., 3.]])
```

把一維陣列拉為二維

在陣列上的計算: Broadcasting

In [6]:

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

print(a)
print(b)
```

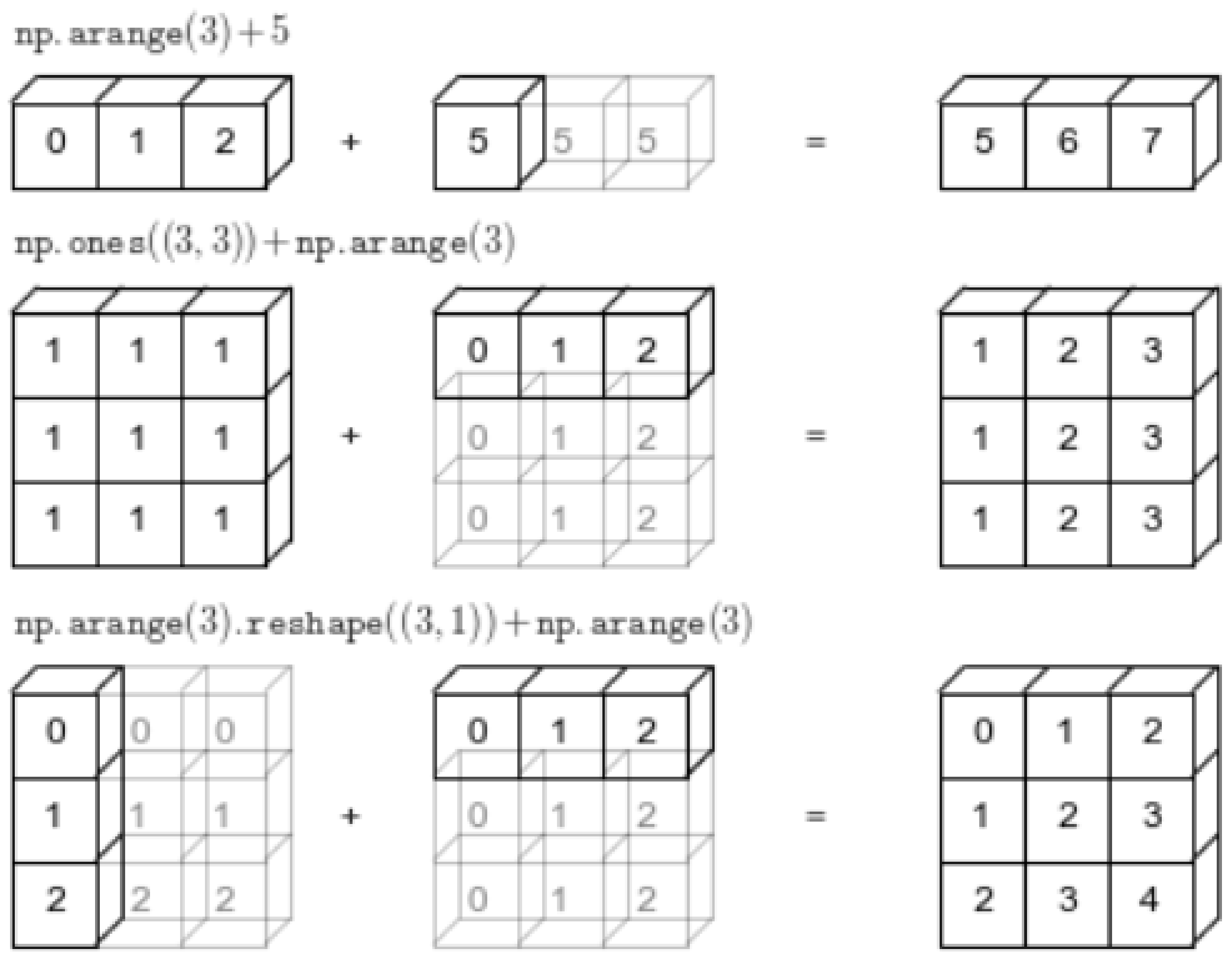
```
[0 1 2]
[[0]
 [1]
 [2]]
```

In [7]:

```python
a+b
```

Out[7]:

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



在陣列上的計算: Broadcasting

規則一:
2陣列維度不同，低維度的陣列從最左邊的元素墊充

規則二:
如果兩個陣列有任一個維度不符合，具有形狀是1的陣列該維度被拉長

規則三:
如果有維度不相同，且該維度沒有兩陣列值皆不等於1，則產生錯誤

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

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

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

在陣列上的計算: Broadcasting

```
In [8]:
M = np.ones((2, 3))
a = np.arange(3)
M + a

Out[8]: array([[1., 2., 3.],
               [1., 2., 3.]])
```

在陣列上的計算: Broadcasting

```
In [9]:  a = np.arange(3).reshape((3, 1))
          b = np.arange(3)
          a + b

Out[9]: array([[0, 1, 2],
               [1, 2, 3],
               [2, 3, 4]])
```

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

Rule 1 says we must pad the shape c

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

And rule 2 tells us that we upgrade e
corresponding size of the other array

- `a.shape -> (3, 3)`
- `b.shape -> (3, 3)`

在陣列上的計算: Broadcasting

```
In [10]:
M = np.ones((3, 2))
a = np.arange(3)
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-d4adfa68cd62> in <module>
      1 M = np.ones((3, 2))
      2 a = np.arange(3)
----> 3 M + a

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

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

Again, rule 1 tells us that we must pa

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

By rule 2, the first dimension of `a` is

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

在陣列上的計算: Broadcasting

```
In [11]: a[:, np.newaxis].shape
Out[11]: (3, 1)

In [12]: M + a[:, np.newaxis]
Out[12]: array([[1., 1.],
                [2., 2.],
                [3., 3.]])
```

在陣列上的計算: Broadcasting

```
In [16]: np.logaddexp(M, a[:, np.newaxis])

Out[16]: array([[ 1.31326169,  1.31326169],
                [ 1.69314718,  1.69314718],
                [ 2.31326169,  2.31326169]])
```

Broadcasting可以套用在任何Binary ufunc上