Introduction to numpy:

Package for scientific computing with Python

Numerical Python, or "Numpy" for short, is a foundational package on which many of the most common data science packages are built. Numpy provides us with high performance multi-dimensional arrays which we can use as vectors or matrices.

The key features of numpy are:

- ndarrays: n-dimensional arrays of the same data type which are fast and space-efficient. There are a number of built-in methods for ndarrays which allow for rapid processing of data without using loops (e.g., compute the mean).
- Broadcasting: a useful tool which defines implicit behavior between multi-dimensional arrays of different sizes.
- Vectorization: enables numeric operations on ndarrays.
- Input/Output: simplifies reading and writing of data from/to file.

Additional Recommended Resources:
Numpy Documentation (https://docs.scipy.org/doc/numpy/reference/)
Python for Data Analysis by Wes McKinney
Python Data science Handbook by Jake VanderPlas

Getting started with ndarray

ndarrays are time and space-efficient multidimensional arrays at the core of numpy. Like the data structures in Week 2, let's get started by creating ndarrays using the numpy package.

How to create Rank 1 numpy arrays:

In [1]:

```
import numpy as np

an_array = np.array([3, 33, 333]) # create a rank 1 array
print(type(an_array)) # The type of an ndarray is "<class 'numpy.ndarray'>"
```

```
<class 'numpy.ndarray'>
```

In [2]:

```
# test the shape of the array we just created
print(an_array.shape)
```

```
(3,)
```

In [3]:

```
# because this is a 1-rank array, we need only one index to access each element
print(an_array[0], an_array[1], an_array[2])
```

```
3 33 333
```

In [4]:

```
an_array[0] = 888 #change an element of the array
print(an_array)
```

```
[888  33 333]
```

In [5]:

```
another = np.array([[11,12,13], [21,22,23]]) # Create a rank 2 array

print(another) # print the array

print('The shape is 2 rows, 3 columns: ', another.shape)  # rows x columns

print('Accessing elements [0,0], [0,1], and [1,0] of the narray', another[0,0],
',', another[0,1], ',', another[1,0])
```

```
[[11 12 13]
 [21 22 23]]
The shape is 2 rows, 3 columns:  (2, 3)
Accessing elements [0,0], [0,1], and [1,0] of the narray 11 , 12 , 2
1
```

In [6]:

```
import numpy as np

#create a 2x2 array of zeros
ex1 = np.zeros((2,2))
print(ex1)
```

```
[[0. 0.]
 [0. 0.]]
```

In [7]:

```
#create a 2x2 array filled with 9.0
ex2 = np.full((2,2),9.0)
print(ex2)
```

```
[[9. 9.]
 [9. 9.]]
```

In [8]:

```
# create a 2 x2 matrix with diagonal 1s and the others 0
ex3 = np.eye(2,2)
print(ex3)
```

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

In [9]:

```
# create an array of ones
ex4 = np.ones((1,2))
print(ex4)
```

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

In [10]:

```
#notice that the above ndarray (ex4) is actually rank 2, it is a 2x1 array
print(ex4.shape)

#which means we need to use two indexes to access an element
print()
print(ex4[0,1])
```

```
(1, 2)

1.0
```

In [11]:

```
# create an array of random floats between 0 and 1
ex5 = np.random.random((2,2))
print(ex5)
```

```
[[0.02494572 0.12015575]
 [0.57899657 0.07059495]]
```

Array Indexing

Slice indexing:

Similar to the use of slice indexing with lists and strings, we can use slice indexing to pull out sub-regions of ndarrays.

In [12]:

```
import numpy as np

# Rank 2 array of shape (3,4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

Use array slicing to get a subarray consisting of the first 2 rows x 2 columns.

In [13]:

```
a_slice = an_array[:2, 1:3] #印出 an_array[0,1], [0,2], [1,1], and [1,2]的值
print(a_slice)
```

```
[[12 13]
 [22 23]]
```

When you modify a slice, you actually modify the underlying array.

In [14]:

```
print('Before: ', an_array[0,1]) #inspect the element at 0,1
a_slice[0,0] = 1000 #a_slice[0,0] is the same piece of data as an_array[0,1]
print('After: ', an_array[0,1])
```

```
Before:  12
After:  1000
```

Use both integer indexing & slice indexing

We can use combinations of integer indexing and slice indexing to create different shaped matrices.

In [15]:

```
# Create a Rank 2 array of shape (3,4)
an_array = np.array([[11,12,13,14], [21,22,23,24], [31,32,33,34]])
print(an_array)
```

```
[[11 12 13 14]
 [21 22 23 24]
 [31 32 33 34]]
```

In [16]:

```
# Using bother integer indexing & slicing generates an array of Lower Rank
row_rank1 = an_array[1, :] # Rank 1 view (row 1)

print(row_rank1, row_rank1.shape) # Notice only a single [] => get 1 row only
```

```
[21 22 23 24] (4,)
```

In [17]:

```
# Slicing alone: generates an array of the same rank as the an_array
row_rank2 = an_array[1:2, :] #Rank 2 view
print(row_rank2, row_rank2.shape) #Notice the [[]] 2D array => 1 row x 4 columns
```

```
[[21 22 23 24]] (1, 4)
```

In [18]:

```
# We can do the same thing for columns of an array:

print()
col_rank1 = an_array[:, 1]
col_rank2 = an_array[:, 1:2]

print(col_rank1, col_rank1.shape) # Rank 1 Only the column 1 value
print()
print(col_rank2, col_rank2.shape) # Rank 2 => 3 rows x 1 column
```

```
[12 22 32] (3,)

[[12]
 [22]
 [32]] (3, 1)
```

Array Indexing for changing elements:

Sometimes it's useful to use an array of indexes to access or change elements.

In [19]:

```
# Create a new array
an_array = np.array([[11,12,13], [21,22,23], [31,32,33], [41,42,43]])

print('Original Array:')
print(an_array)
```

```
Original Array:
[[11 12 13]
 [21 22 23]
 [31 32 33]
 [41 42 43]]
```

In [20]:

```
# Create an array of indices
col_indices = np.array([0, 1, 2, 0])
print('\nCol indices picked: ', col_indices)

row_indices = np.arange(4) # Use the np.arange fuction to create an ndarray
print('\nRows indices picked: ', row_indices)
```

```
Col indices picked:  [0 1 2 0]

Rows indices picked:  [0 1 2 3]
```

In [21]:

```
# Examine the pairings of row_indices and col_indices.
for row, col in zip(row_indices, col_indices):
    print(row, ', ', col)
```

```
0 ,  0
1 ,  1
2 ,  2
3 ,  0
```

zip()是Python的一个内建函数，它接受一系列可迭代的对象作为参数，将对象中对应的元素打包成一个个tuple（元组），然后返回由这些tuples组成的list（列表）

In [22]:

```
# Select one element from each row
print('Values in the array at those indices: ', an_array[row_indices, col_indice
s])
```

```
Values in the array at those indices:  [11 22 33 41]
```

In [23]:

```
# Change one element from each row using the indices selected
an_array[row_indices, col_indices] += 100000 # value加上100000

print('\nChanged Array: ')
print(an_array)
```

```
Changed Array:
[[100011     12     13]
 [    21 100022     23]
 [    31     32 100033]
 [100041     42     43]]
```

Quiz 1: If you try to access rows of a 3-by-3 numpy array called “arr” using the command: arr[:2,] How many rows will be returned?

In [24]:

```
arr = np.array([[11,12,13],[21,22,23],[31,32,33]])
print(arr)
```

```
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```

In [25]:

```
print(arr[:2,]) #印出 arr [0,0], [0,1], [0,3], [1,1],[1,2], and [1,3] 的數值
```

```
[[11 12 13]
 [21 22 23]]
```

Boolean Indexing

Array Indexing for changing elements:

In [26]:

```
# Create a 3x2 array
an_array = np.array([[11,12],[21,22],[31,32]])
print(an_array)
```

```
[[11 12]
 [21 22]
 [31 32]]
```

In [27]:

```
# create a filter which will be boolean values for whether each element meets th
is condition
filter = (an_array > 15)
filter
```

Out[27]:

```
array([[False, False],
       [ True,  True],
       [ True,  True]])
```

Notice that the filter is a same size ndarray as an_array which is filled with True for each element whose corresponding element in an_array which is greater than 15 and False for those elements whose value is less than 15.

In [28]:

```
# We can now select just those elements which meet that criteria
print(an_array[filter])
```

```
[21 22 31 32]
```

In [29]:

```
# For short, we could have just used the approach below without the need for the
 separate filter array.
an_array[(an_array > 20) & (an_array < 30)]
```

Out[29]:

```
array([21, 22])
```

In [30]:

```
an_array[(an_array % 2 == 0)]
```

Out[30]:

```
array([12, 22, 32])
```

In [31]:

```
an_array[an_array % 2 == 0] += 100
print(an_array)
```

```
[[ 11 112]
 [ 21 122]
 [ 31 132]]
```

Datatypes and Array Operations

Datatypes:

In [32]:

```
ex1 = np.array([11, 12]) # Python assigns the data type
print(ex1.dtype) # result is integer
```

```
int64
```

In [33]:

```
ex2 = np.array([11.0, 12.0]) # Python assigns the data type
print(ex2.dtype)
```

```
float64
```

In [34]:

```
ex3 = np.array([11, 12], dtype = np.int64) #You can also tell Python the  data t
ype
print(ex3.dtype)
```

```
int64
```

In [35]:

```
# you can use this to force floats into integers (using floor function)
ex4 = np.array([3.29, 11.29], dtype = np.int64)
print(ex4.dtype, end = '\n')
print(ex4)
```

```
int64
[ 3 11]
```

In [36]:

```
# you can use this to force integers into floats if you anticipate
# the values may change to floats later
ex5 = np.array([11, 21], dtype = np.float64)
print(ex5.dtype, end = '\n')
print(ex5)
```

```
float64
[11. 21.]
```

Arithmetic Array Operations:

In [37]:

```
x = np.array([[111,112], [121,122]], dtype = np.int)
y = np.array([[211.1, 212.1], [221.1, 222.1]], dtype = np.float)

print(x)
print()
print(y)
```

```
[[111 112]
 [121 122]]

[[211.1 212.1]
 [221.1 222.1]]
```

In [38]:

```
#add
print(x + y) # The plus sign works
print()
print(np.add(x, y)) # so does the numpy function 'add'
```

```
[[322.1 324.1]
 [342.1 344.1]]

[[322.1 324.1]
 [342.1 344.1]]
```

In [39]:

```
#subtract
print(x - y)
print()
print(np.subtract(x, y)) # so does the numpy function 'subtract'
```

```
[[-100.1 -100.1]
 [-100.1 -100.1]]

[[-100.1 -100.1]
 [-100.1 -100.1]]
```

In [40]:

```
# Multiply
print(x * y)
print()
print(np.multiply(x, y)) # so does the numpy function 'multiply'
```

```
[[23432.1 23755.2]
 [26753.1 27096.2]]

[[23432.1 23755.2]
 [26753.1 27096.2]]
```

In [41]:

```
# divide
print(x / y)
print()
print(np.divide(x, y)) # so does the numpy function 'divide'
```

```
[[0.52581715 0.52805281]
 [0.54726368 0.54930212]]

[[0.52581715 0.52805281]
 [0.54726368 0.54930212]]
```

In [42]:

```
# square root
print(np.sqrt(x))
```

```
[[10.53565375 10.58300524]
 [11.         11.04536102]]
```

In [43]:

```
# exponent (e ** x)
print(np.exp(x))
```

```
[[1.60948707e+48 4.37503945e+48]
 [3.54513118e+52 9.63666567e+52]]
```

Statistical Methods, Sorting, and Set Operations:

Basic Statistical Operations:

In [44]:

```
# Setup a random 2 x 5 matrix
arr = 10 * np.random.randn(2,5) #注意這邊是 randn 而不是 random
print(arr)
```

```
[[-4.29596702  8.85695623 -2.93709496 -5.74596835 12.94277054]
 [ 5.01788568 -5.96702379  8.35652711 10.57380255 -2.25933966]]
```

In [45]:

```
# compute the mean for All elements
print(arr.mean())
```

```
2.454254832366753
```

In [46]:

```
# compute the means by row
print(arr.mean(axis = 1))
```

```
[1.76413929 3.14437038]
```

In [47]:

```
# compute the means by column
print(arr.mean(axis = 0))
```

```
[0.36095933 1.44496622 2.70971607 2.4139171  5.34171544]
```

In [48]:

```
# sum all the elements
print(arr.sum())
```

```
24.542548323667532
```

In [49]:

```
# compute the medians by row
print(np.median(arr, axis = 1))
```

```
[-2.93709496  5.01788568]
```

Sorting:

In [50]:

```
# create a 10 element array of randoms
unsorted = np.random.randn(10)

print(unsorted)
```

```
[ 0.6883593  -0.35184334 -1.23295929 -0.7871837   1.43636212 -0.5070
8798
 -1.48854685 -0.85241316 -1.82942734 -0.18040806]
```

In [51]:

```
# create copy and sort
sorted = np.array(unsorted) #copy unsorted, then name it as sorted
sorted.sort()

print(sorted)
print()
print(unsorted)
```

```
[-1.82942734 -1.48854685 -1.23295929 -0.85241316 -0.7871837  -0.5070
8798
 -0.35184334 -0.18040806  0.6883593   1.43636212]

[ 0.6883593  -0.35184334 -1.23295929 -0.7871837   1.43636212 -0.5070
8798
 -1.48854685 -0.85241316 -1.82942734 -0.18040806]
```

In [52]:

```
# inplace sorting
unsorted.sort()

print(unsorted)
```

```
[-1.82942734 -1.48854685 -1.23295929 -0.85241316 -0.7871837  -0.5070
8798
 -0.35184334 -0.18040806  0.6883593   1.43636212]
```

Finding Unique elements:

In [53]:

```
array = np.array([1, 2, 1, 4, 2, 1, 4, 2])
print(np.unique(array)) #不重複取出
```

```
[1 2 4]
```

Set Operations with np.array data type:

In [54]:

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

```
['desk' 'chair' 'bulb'] ['lamp' 'bulb' 'chair' 'table']
```

In [55]:

```
print(np.intersect1d(s1, s2)) #s1 & s2都有
```

```
['bulb' 'chair']
```

Notice that we're using intersect1d because intersect expects 1d arrays

In [56]:

```
print(np.union1d(s1, s2)) # The metod union will give us all of the unique eleme
nts across both arrays.
```

```
['bulb' 'chair' 'desk' 'lamp' 'table']
```

In [57]:

```
print(np.setdiff1d(s1,s2)) # S1有, S2沒有
```

```
['desk']
```

In [58]:

```
print(np.setdiff1d(s2,s1)) # S2有, S1没有
```

```
['lamp' 'table']
```

In [59]:

```
print(np.in1d(s1, s2)) # We can get back an array of Booleans, for whether each
 element(s1) is in the array (s2) or not
```

```
[False  True  True]
```

Broadcasting:

Introduction to broadcasting.
For more details, please see:
https://docs.scipy.org/doc/numpy-1.10.1/user/basics.broadcasting.html (https://docs.scipy.org/doc/numpy-1.10.1/user/basics.broadcasting.html)

In [60]:

```
# We create a 4x3 ndarray and we fill it with zeros
import numpy as np

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

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

In [61]:

```
# create a rank 1 ndarray with 3 values
```

In [62]:

```
add_rows = np.array([1, 0, 2])
print(add_rows)
```

```
[1 0 2]
```

In [63]:

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

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

In [64]:

```
# 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 #Transpose on it, denoted by T

print(add_cols)
```

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

In [65]:

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

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

In [66]:

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

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

Quiz 2 : True or False: Take a look at the following lines of code:

a = np.array([[0,0],[0,0]])
b1 = np.array([1,1])
b2 = 1
a+b1 and a+b2 result in the same matrix.

In [67]:

```
a = np.array([[0,0],[0,0]])
b1 = np.array([1,1])
b2 = 1

print(a+b1)
print()
print(a+b2)
```

```
[[1 1]
 [1 1]]

[[1 1]
 [1 1]]
```

Example from the slides:

In [68]:

```
# create our 3x4 matrix
arrA = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
print(arrA)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

In [69]:

```
# create our 4x1 array
arrB = [0,1,0,2]
print(arrB)
```

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

In [70]:

```
# add the two together using broadcasting
print(arrA + arrB)
```

```
[[ 1  3  3  6]
 [ 5  7  7 10]
 [ 9 11 11 14]]
```

Speedtest: ndarrays vs lists

First setup paramaters for the speed test. We'll be testing time to sum elements in an ndarray versus a list.

In [71]:

```
from numpy import arange
from timeit import Timer

size  = 1000000
timeits = 1000
```

In [72]:

```
# create the ndarray with values 0, 1, 2, ...., size - 1
nd_array = arange(size)
print(type(nd_array))
```

```
<class 'numpy.ndarray'>
```

In [73]:

```
# timer expects the operation as a parameter,
# here we pass nd_array.sum()
timer_numpy = Timer('nd_array.sum()', 'from __main__ import nd_array' )

print('Time taken by numpy ndarray: %f seconds' %
      (timer_numpy.timeit(timeits)/timeits))
```

```
Time taken by numpy ndarray: 0.000464 seconds
```

In [74]:

```
# create the list with values 0,1,2...,size-1
a_list = list(range(size))
print(type(a_list))
```

```
<class 'list'>
```

In [75]:

```
# timer expects the operation as a parameter, here we pass sum(a_list)
timer_list = Timer('sum(a_list)', 'from __main__ import a_list' )

print('Time taken by list: %f seconds' %
      (timer_list.timeit(timeits)/timeits))
```

```
Time taken by list: 0.003579 seconds
```

Read or Write to Disk:

Binary Format:

In [76]:

```
x = np.array([3.29, 11.29])
```

In [77]:

```
np.save('an_array', x)
```

In [78]:

```
np.load('an_array.npy')
```

Out[78]:

```
array([ 3.29, 11.29])
```

Text Format:

In [79]:

```
np.savetxt('array.txt', X=x, delimiter=',')
```

In [80]:

```
!cat array.txt
```

```
3.29000000000000036e+00
1.12899999999999915e+01
```

In [81]:

```
np.loadtxt('array.txt', delimiter=',')
```

Out[81]:

```
array([ 3.29, 11.29])
```

Additional Common ndarray Operations

Dot Product on Matrices and Inner Product on Vectors:

In [82]:

```
# determine the dot product of two matrices
x2d = np.array([[1,1],[1,1]])
y2d = np.array([[2,2],[2,2]])

print(x2d.dot(y2d))
print()
print(np.dot(x2d, y2d)) #或者這樣寫
```

```
[[4 4]
 [4 4]]

[[4 4]
 [4 4]]
```

In [83]:

```
# determine the inner product of two vectors
a1d = np.array([9 ,9 ])
b1d = np.array([10, 10])

print(a1d.dot(b1d))
print()
print(np.dot(a1d, b1d))
```

```
180

180
```

In [84]:

```
# dot produce on an array and vector
print(x2d.dot(a1d))
print()
print(np.dot(x2d, a1d))
```

```
[18 18]

[18 18]
```

Sum:

In [85]:

```
# sum elements in the array
ex1 = np.array([[11,12],[21,22]])

print(np.sum(ex1)) # add all members
```

```
66
```

In [86]:

```
print(np.sum(ex1, axis = 0)) # columnwise sum 11 + 21 = 32, 12 + 22 = 34
```

```
[32 34]
```

In [87]:

```
print(np.sum(ex1, axis = 1)) # rowwise sum 11 + 12 = 23, 21 + 22 = 43
```

```
[23 43]
```

Element-wise Functions:

For example, let's compare two arrays values to get the maximum of each.

In [88]:

```
# random array
x = np.random.randn(8)
x
```

Out[88]:

```
array([-0.98914009,  1.54842806,  1.3024303 ,  2.4891781 ,  0.218884
69,
       -0.09619948, -0.3736248 , -0.47017262])
```

In [89]:

```
# another random array
y = np.random.randn(8)
y
```

Out[89]:

```
array([ 0.48487665, -1.48107856, -0.88102   , -0.03890742, -0.752983
21,
       -0.40312076, -0.34005599,  0.96203137])
```

In [90]:

```
# returns element wise maximum between two arrays
np.maximum(x,y)
```

Out[90]:

```
array([ 0.48487665,  1.54842806,  1.3024303 ,  2.4891781 ,  0.218884
69,
       -0.09619948, -0.34005599,  0.96203137])
```

Reshaping array:

In [91]:

```
# grab values from 0 through 19 in an array
arr = np.arange(20) #不包含20
print(arr)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

In [92]:

```
# reshape to be a 4 x 5 matrix
arr.reshape(4,5)
```

Out[92]:

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Transpose:

In [93]:

```
# transpose
ex1 = np.array([[11,12],[21,22]])

ex1.T
```

Out[93]:

```
array([[11, 21],
       [12, 22]])
```

Indexing using where():

In [94]:

```
x_1 = np.array([1, 2, 3, 4, 5])
y_1 = np.array([11, 22, 33, 44, 55])

filter = np.array([True, False, True, False, True])
```

In [95]:

```
out = np.where(filter, x_1, y_1) # x_1中為True用 x_1的值, 反之y_1
print(out)
```

```
[ 1 22  3 44  5]
```

可參考: https://www.zhihu.com/question/62844162 (https://www.zhihu.com/question/62844162)

In [96]:

```
mat = np.random.rand(5,5)
mat
```

Out[96]:

```
array([[0.34180607, 0.90830567, 0.64669763, 0.64762527, 0.73685068],
       [0.83410405, 0.97054872, 0.77749058, 0.73295702, 0.40768179],
       [0.45726776, 0.56880766, 0.34389295, 0.42142743, 0.74086624],
       [0.5749876 , 0.69487209, 0.6335134 , 0.29589649, 0.60473162],
       [0.86313484, 0.42168064, 0.9590684 , 0.33773841, 0.9718863
1]])
```

In [97]:

```
np.where(mat > 0.5, 1000, -1) #大於0.5標示 1000, 反之-1
```

Out[97]:

```
array([[  -1, 1000, 1000, 1000, 1000],
       [1000, 1000, 1000, 1000,   -1],
       [  -1, 1000,   -1,   -1, 1000],
       [1000, 1000, 1000,   -1, 1000],
       [1000,   -1, 1000,   -1, 1000]])
```

"any" or "all" conditionals:

In [98]:

```
arr_bools = np.array([ True, True, False, True, False])
```

In [99]:

```
arr_bools.any() #任何一個有True就是True
```

Out[99]:

```
True
```

In [100]:

```
arr_bools.all() #全部都是True
```

Out[100]:

```
False
```

Random Number Generation:

In [101]:

```
Y = np.random.normal(size = (1,5))[0]
print(Y)
```

```
[ 0.31472104  0.22044968  1.50351937 -0.08042775  0.21288945]
```

In [102]:

```
Z = np.random.randint(low=2, high=50, size=4)
print(Z)
```

```
[24 25  9 14]
```

In [103]:

```
np.random.permutation(Z) #return a new ordering of elements in Z Z中的數值隨機排列
```

Out[103]:

```
array([25,  9, 14, 24])
```

In [104]:

```
np.random.uniform(size=4) #uniform distribution
```

Out[104]:

```
array([0.25273061, 0.46032597, 0.87630175, 0.86874634])
```

In [105]:

```
np.random.normal(size=4) #normal distribution
```

Out[105]:

```
array([-0.04285344, -0.39094056, -0.75571981,  0.85482712])
```

Merging data sets:

In [106]:

```
K = np.random.randint(low=2, high=50, size=(2,2))
print(K)

print()
M = np.random.randint(low=2, high=50, size=(2,2))
print(M)
```

```
[[22  9]
 [44 41]]

[[23 15]
 [ 6  3]]
```

In [107]:

```
np.vstack((K,M)) #垂直合併
```

Out[107]:

```
array([[22,  9],
       [44, 41],
       [23, 15],
       [ 6,  3]])
```

In [108]:

```
np.hstack((K,M)) #水平合併
```

Out[108]:

```
array([[22,  9, 23, 15],
       [44, 41,  6,  3]])
```

In [109]:

```
np.concatenate([K,M], axis = 0) #串連column
```

Out[109]:

```
array([[22,  9],
       [44, 41],
       [23, 15],
       [ 6,  3]])
```

In [110]:

```
np.concatenate([K,M], axis = 1) #串連row
```

Out[110]:

```
array([[22,  9, 23, 15],
       [44, 41,  6,  3]])
```

In [111]:

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

Out[111]:

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

In [112]:

```
slice = arr[:2, 1:3] #第一個取集合，第二度取數值
slice
```

Out[112]:

```
array([[2, 3],
       [5, 6]])
```

In [113]:

```
slice[0,0]
```

Out[113]:

```
2
```

In [114]:

```
arr
```

Out[114]:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [115]:

```
A = np.array([[1],[2]])
B = np.array([[1,2], [3,4]])
A+B
```

Out[115]:

```
array([[2, 3],
       [5, 6]])
```

In [116]:

```
B
```

Out[116]:

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

In [117]:

```
A
```

Out[117]:

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