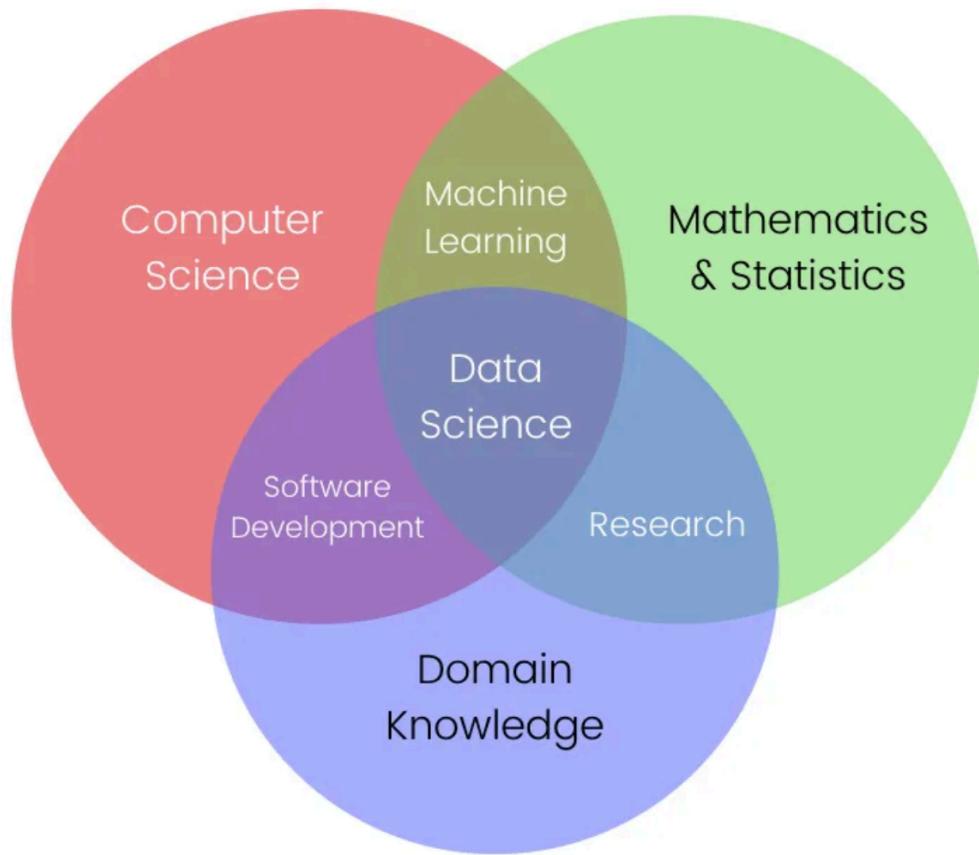


# Data Science (วิทยาการข้อมูล) คืออะไร

Data Science หรือในภาษาไทยแปลว่า “วิทยาการข้อมูล” คือ ศาสตร์ที่รวมเอาความรู้ด้านวิทยาการคอมพิวเตอร์(Computer Science) ด้านคณิตศาสตร์และสถิติ(Math & Statistics) ด้านความรู้เฉพาะทาง(Domain Knowledge) มาประยุกต์รวมกันเพื่อจัดเก็บ รวบรวม ตรวจสอบ วิเคราะห์ และนำเสนอข้อมูลที่ออกแบบมาในรูปแบบของข้อมูลเชิงลึก (Insight) เพื่อนำไปใช้ประโยชน์ในด้านต่าง ๆ เช่น เศรษฐศาสตร์ การเงิน โลจิสติกส์ วิศวกรรม การแพทย์ เป็นต้น



ขอบคุณรูปจาก [medium.com](https://medium.com)

## Data Science กับ Python

ภาษา Python เป็นหนึ่งในภาษาที่นิยมใช้กันในงานสาย Data Science

ข้อดีของการใช้งาน Python ทำงาน Data Science สามารถสรุปได้ดังนี้

1. เป็นมิตรกับมือใหม่ Python ใช้งานง่ายและมี syntax ที่เรียนง่าย ภาษาที่มีจึงเป็นเครื่องมือที่เหมาะสมกับมือใหม่

2. มีชุดเครื่องมือสำหรับคณิตศาสตร์และสถิติ Python มีฟังก์ชันในการคำนวณทางคณิตศาสตร์ ทำเรื่องสถิติ และสร้างโมเดลทางสถิติ มันจึงเป็นภาษาที่เหมาะสมกับการใช้งานทางด้านวิทยาศาสตร์ ข้อมูลมาก
3. เหมาะกับการทำ data visualization Python เหมาะกับการทำ data visualization ซึ่งจะช่วยให้เราเข้าใจข้อมูลได้ดี เช่น เข้าใจความสัมพันธ์ที่น่าจะเป็นไปได้ เทื่องความสัมพันธ์ที่ไม่ปรากฏเด่นชัด และเห็นเทอเรนด์ต่างๆ
4. มี open-source library จำนวนมากให้ใช้ Python เป็นภาษาที่มีห้องสมุด open-source จำนวนมากให้ใช้ และเป็นห้องสมุดที่มีมากกว่าส่วนของการคำนวณ สถิติ และ data visualization
5. มีประสิทธิภาพและปรับขนาดได้ Python เหมาะจะนำไปใช้ในงาน Data Science เพราะมันทั้งมีประสิทธิภาพ และใช้กับงานทั้งใหญ่และเล็กได้
6. มีชุมชนที่เข้มแข็ง Python มีชุมชนที่เข้มแข็ง และทำงานต่อเนื่องเพื่อพัฒนา libraries สำหรับงานวิทยาศาสตร์ ข้อมูลให้ดีขึ้น

## library พื้นฐานสำหรับงาน Data Science ของ Python

ภาษาสารพัดประโยชน์อย่าง Python ถ้าจะต้องจำทุกคำสั่ง ก็คงจะต้องใช้แรงไม่น้อย เ畧มีผู้พัฒนาหลายๆ คน พยายามที่จะนำคำสั่งต่างๆ ของ Python มาสร้างเป็นชุดคำสั่ง หรือเป็น Package เพื่อให้สามารถทำงานตามวัตถุประสงค์แต่ละด้านได้อย่างมีประสิทธิภาพมากขึ้น โดยที่เรียกสิ่งที่ว่านี้ว่า "Python Library"

### 1. NumPy

มีชื่อเต็มว่า "Numerical Python" โดยเด่นในด้านการคำนวณ และการทำงานกับตัวเลข ( NumPy ถือเป็น Scientific Computing Library ที่สำคัญมากของ Python)

นอกจากนี้ NumPy ยังมีความสามารถสำคัญในการสร้าง Array (โครงสร้างข้อมูล) และ Multidimensional Array ได้ ทำให้การคำนวณบน Python มีความรวดเร็วมากขึ้น ซึ่งแม้ Python พื้นฐานเอง จะมี Python list ที่มีความคล้ายคลึงกับ Array แต่ NumPy สามารถจัดการข้อมูลเหล่านี้ได้เร็วกว่าการใช้ Python list ธรรมชาติ

### 2. Pandas

สุดยอด Library แห่งการจัดการข้อมูล (Data Wrangling/ Data Cleaning) และการวิเคราะห์ข้อมูล (Data Analysis)

pandas สามารถเชื่อมต่อกับแหล่งข้อมูลได้หลากหลาย หลังจากนั้นก็สามารถจัดเตรียมข้อมูล ทำความสะอาด และจัดรูปแบบให้พร้อมกับการนำไปวิเคราะห์ ตลอดจนการแสดงผล

### 3. Matplotlib

เป็น Library อันดับหนึ่งในการสร้างกราฟ และทำ Data Visualization (คล้ายกับ MATLAB ซึ่งมาพร้อมกับ Python) โดยที่ Matplotlib สามารถสร้างกราฟได้หลายประเภทเพื่อตอบโจทย์การทำงานของผู้ใช้ให้ได้หลากหลาย เช่น กราฟเส้น แผนภูมิจุดแบบกระจัดกระจาย (Scatter Plot), กราฟแท่ง และชิลต์แกรม, แผนภูมิบ็อกซ์และวิสเกอร์ (Box Plot หรือ Whisker Plot) และอื่นๆ

## ทำไม Numpy Array ถึงน่าใช้กว่า List

แน่นอนว่าสิ่งที่เราสองัญ ณ ตอนนี้คือ แล้ว List กับ Numpy Array เนี่ย ตัวไหนมันทำงานได้ดีกว่ากันล่ะ หรือมันทำงานได้รวดเร็วเท่ากันกันแน่ แน่นอนว่ามีวิธีหาคำตอบนี้ครับ ซึ่งก็คือการลงมือเขียนโค้ดนั่นเองเทียบประสิทธิภาพให้ทุกคนเห็นกับตาันนั่นเองครับ!!! โดยจะยกตัวอย่างให้เห็นแบบลึกๆ ด้วยการเทียบประสิทธิภาพด้วยการให้ทั้ง 2 ตัวนี้มีสมาชิกทั้งหมด 1,000,000 ตัว (ใช้ครับ คาดว่า ประมาณนี้กำลังดีเลยในการยกตัวอย่างง่ายๆ เนื่องจากงาน Data science นั้นต้องจัดการกับข้อมูลที่มีจำนวนมากๆ ตั้งนั้น 1 ล้านตัวนั้นไม่น้อยเกินไปครับ) โดยเราจะนำสมาชิกทุกตัวใน List และ Numpy Array มาคูณเข้าด้วย 2 ทั้งหมด และมาดูกันครับว่าใครสามารถทำงานได้ดีกว่ากัน

```
In [1]: #import Numpy library to create Numpy Array and
# import time library for making computing performance metric
import numpy as np
import time as t
```

```
In [2]: # initialize number of elements and another subject
n = 1000000
ListA = []

# Create a List and Array that has 1M elements

# List
for i in range(n):
    ListA.append(i)

# Numpy Array
numpy_array = np.arange(n)
```

```
In [3]: # Perform multiply by 2, then observe time efficiency

# By List
start_time_list = t.time()
List = [i*2 for i in ListA]
list_perform_time = t.time() - start_time_list
print('Time used for computation by List is {} seconds'.format(list_perform_)

# By Numpy Array
start_time_array = t.time()
numpy_array = numpy_array*2
array_perform_time = t.time() - start_time_array
print('Time used for computation by Numpy is {} seconds'.format(array_perfor
```

Time used for computation by List is 0.01481318473815918 seconds  
Time used for computation by Numpy is 0.0008368492126464844 seconds

```
In [4]: # Time efficiency of List and Numpy Array on the same task

print(f'Difference between performing by List and Numpy Array are {list_perform}
print(f'Ratio between List and Numpy Array computation time is {list_perform

Difference between performing by List and Numpy Array are 0.0139763355255126
95
Ratio between List and Numpy Array computation time is 17.7011396011396
```

```
In [5]: # Perform multiply by 2, then observe time efficiency

# By List
start_time_list = t.time()
List = [i**2 for i in ListA]
list_perform_time = t.time() - start_time_list
print('Time used for computation by List is {} seconds'.format(list_perform

# By Numpy Array
start_time_array = t.time()
numpy_array = numpy_array**2
array_perform_time = t.time() - start_time_array
print('Time used for computation by Numpy is {} seconds'.format(array_perfor
```

Time used for computation by List is 0.02100515365600586 seconds  
 Time used for computation by Numpy is 0.0008189678192138672 seconds

```
In [6]: # Time efficiency of List and Numpy Array on the same task

print(f'Difference between performing by List and Numpy Array are {list_perform
print(f'Ratio between List and Numpy Array computation time is {list_perform

Difference between performing by List and Numpy Array are 0.0201861858367919
92
Ratio between List and Numpy Array computation time is 25.648326055312953
```

## บทสรุป

จะเห็นว่าตัวของ Numpy Array สามารถทำงานได้รวดเร็วกว่าตัวของ List อよ้วงหลายเท่าเลยทีเดียวครับ

เทียบจากการนำคูณด้วย 2 แล้ว การทำงานของ Numpy นั้นเร็วกว่า List ถึง 21 เท่าเลยทีเดียว!

เทียบจากการนำยกกำลัง 2 แล้ว การทำงานของ Numpy นั้นเร็วกว่า List ถึง 33 เท่าเลยทีเดียว!

ลองคิดดูนะครับว่าถ้าประมวลผลข้อมูลชุด A ใช้เวลา 1 ชั่วโมงโดยการใช้ Numpy Array และถ้าเราเลือกใช้ List ล่ะ (มันจะนานขนาดไหน) ซึ่ง ณ ตอนนี้ทุกคนก็น่าจะพอเห็นภาพแบบคร่าวๆแล้วนะครับว่าเจ้าตัว Numpy Array เนี่ยมันเร็วกว่าเยอะมากๆเมื่อต้องจัดการกับข้อมูลจำนวนขนาดใหญ่และมีความซับซ้อนสูงครับ

## List & Array (ndarray)

### ชนิดข้อมูล

- Array สามารถเป็น Array ต้องมีชนิดข้อมูลเหมือนกัน

- List สมาชิกมีชนิดข้อมูลต่างกันได้

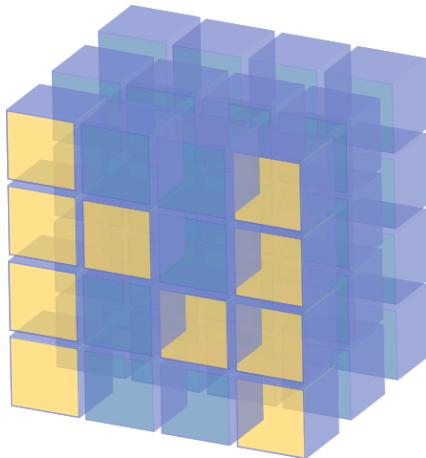
ขนาด

- Array ขนาดที่แน่นอนเปลี่ยนแปลงขนาดไม่ได้
- List มีขนาดที่ยืดหยุ่นกว่า

Array คือการนำข้อมูลมาอยู่ในกลุ่ม เดียวกัน โดยสมาชิกภายใน Array ต้องมีชนิดข้อมูลเหมือนกัน



In [ ]:



# NumPy

## Introduction to NumPy

- NumPy is the fundamental package for scientific computing in Python

```
In [1]: # เรียกใช้ numpy library โดยกำหนดชื่อย่อว่า np
import numpy as np
```

```
In [2]: # check version ของ numpy library
print(np.__version__)
```

1.26.0

- Use the following command to read the documentation:

```
In [ ]: # ข้อแนะนำคือ อย่า run!!!!!
help(np)
```

- This is a basic numpy array usage:

- Array คือการนำข้อมูลมาอยู่ในกลุ่มเดียวกัน โดยสามารถเข้าถึง Array ต้องมีชนิดข้อมูลเหมือนกัน

## List & Array (ndarray)

### ชนิดข้อมูล

↑  
Loading [MathJax]/extensions/Safe.js ใช้ใน Array ต้องมีชนิดข้อมูลเหมือนกัน

- List สมาชิกมีชนิดข้อมูลต่างกันได้

ขนาด

- Array ขนาดที่แน่นอนเปลี่ยนแปลงขนาดไม่ได้
- List มีขนาดที่ยืดหยุ่นกว่า

```
In [4]: # integer array: ข้อมูลคือจำนวนเต็มเหมือนกัน
np.array([1,4,2,5,3])
```

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

```
In [5]: # ถ้าข้อมูลใน array ไม่เหมือนกันจะเกิดอะไรขึ้น?????
np.array([3.2,4,6,5])
```

Out[5]: array([3.2, 4., 6., 5.])

```
In [6]: # ถ้าข้อมูลใน array ไม่เหมือนกันจะเกิดอะไรขึ้น?????
np.array([1,4,2,5,3,'11'])
```

Out[6]: array(['1', '4', '2', '5', '3', '11'], dtype='<U21')

```
In [7]: # ถ้าข้อมูลใน array ไม่เหมือนกันจะเกิดอะไรขึ้น?????
np.array([1,4,2,5,3,True])
```

Out[7]: array([1, 4, 2, 5, 3, 1])

```
In [8]: # ถ้าข้อมูลใน array ไม่เหมือนกันจะเกิดอะไรขึ้น?????
np.array([-1,4,-2,5,3+2j])
```

Out[8]: array([-1.+0.j, 4.+0.j, -2.+0.j, 5.+0.j, 3.+2.j])

## การสร้าง array 1 มิติ

```
In [9]: # สร้าง array โดยระบุสมาชิก(list) ลงไบเลย
a = np.array([1,2,3])
```

In [10]: a

Out[10]: array([1, 2, 3])

```
[1 2 3]
```

```
In [12]: # สร้าง array โดยระบุสมาชิก(tuple) ลงไปเลย
b = np.array((7,8,9))
```

```
In [13]: b
```

```
Out[13]: array([7, 8, 9])
```

```
In [14]: print(b)
[7 8 9]
```

```
In [15]: # check array ที่สร้างว่ามีกี่มิติ
a.ndim
```

```
Out[15]: 1
```

```
In [16]: np.array((17,18,19))
```

```
Out[16]: array([17, 18, 19])
```

```
In [17]: np.array((17,18,19)).ndim
```

```
Out[17]: 1
```

```
In [18]: # check array ที่สร้างว่ามีกี่มิติ
b.ndim
```

```
Out[18]: 1
```

```
In [19]: # สร้าง array โดยใช้ตัวแปร(list) เสริม
li = [2,4,6,8]
c = np.array(li)
```

```
In [20]: print(c)
[2 4 6 8]
```

```
In [21]: # สร้าง array โดยใช้ตัวแปร(tuple) เสริม
tu = (3,5,7,9)
d = np.array(tu)
```

```
In [22]: d
```

```
Out[22]: array([3, 5, 7, 9])
```

## การสร้าง array 2 มิติ

```
In [23]: # สร้าง array โดยระบุสมาชิก(list) ลงไบเดย
e = np.array([[1,2,3],[4,5,6]])
```

```
In [24]: e
```

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

```
In [25]: print(e)
```

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

```
In [26]: # check array ที่สร้างว่ามีกี่มิติ
e.ndim
```

```
Out[26]: 2
```

```
In [27]: # สร้าง array โดยใช้ตัวแปร(list) เสริม
li1 = [[1,2,3],[4,5,6],[7,8,9]]
f = np.array(li1)
```

```
In [28]: print(f)
```

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

```
In [29]: # nested lists result in multidimensional arrays
h = np.array([range(i,i+3) for i in [2,4,6]])
```

```
In [30]: h
```

```
Out[30]: array([[2, 3, 4],
 [4, 5, 6],
 [6, 7, 8]])
```

```
In [31]: h.ndim
```

```
Out[31]: 2
```

## การสร้าง array 3 มิติ

Loading [MathJax]/extensions/Safe.js

```
In [32]: # สร้าง array โดยระบุสมาชิก(list) ลงไบเลย
g = np.array([[[1,2,3],[4,5,6]])
```

```
In [33]: g
```

```
Out[33]: array([[1, 2, 3],
 [4, 5, 6]])
```

```
In [34]: g.ndim
```

```
Out[34]: 3
```

## Understanding Data Types in Python

```
In [35]: # เราสามารถสร้าง array โดยระบุ data type ได้
np.array([1,2,3,4], dtype="str")
```

```
Out[35]: array(['1', '2', '3', '4'], dtype='|<U1')
```

```
In [36]: # เราสามารถสร้าง array โดยระบุ data type ได้
np.array([3,6,2,3], dtype="float32")
```

```
Out[36]: array([3., 6., 2., 3.], dtype=float32)
```

## Creating Arrays from Scratch

### การสร้าง array ที่มีสมาชิกเป็น 0

```
In [37]: # Create a length-5 float array filled with zeros
np.zeros(5)
```

```
Out[37]: array([0., 0., 0., 0., 0.])
```

```
In [38]: # Create a length-10 integer array filled with zeros
np.zeros(10, dtype="int")
```

```
Out[38]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Loading [MathJax]/extensions/Safe.js

```
In [39]: np.zeros((5,6))
```

```
Out[39]: array([[0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.]])
```

```
In [40]: np.zeros([5,6])
```

```
Out[40]: array([[0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.]])
```

```
In [41]: np.zeros((5,6), dtype="float32")
```

```
Out[41]: array([[0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0.]], dtype=float32)
```

```
In [42]: np.zeros((5,6), dtype="int8")
```

```
Out[42]: array([[0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0]], dtype=int8)
```

## การสร้าง array ที่มีสมาชิกเป็น 1

```
In [43]: # Create a 10 floating-point array filled with 1s
```

```
np.ones(10)
```

```
Out[43]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
In [44]: np.ones(10).dtype
```

```
Out[44]: dtype('float64')
```

```
In [45]: np.ones((3,5))
```

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

```
In [46]: # Create a 3x5 integer array filled with 1s
```

Loading [MathJax]/extensions/Safe.js

```
np.ones((3,5), dtype="int")
```

```
Out[46]: array([[1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1]])
```

## การสร้าง array ที่มีสมาชิกเป็นค่าคงที่ไดๆ

```
In [47]: # Create a 10 array filled with 8
np.full(10, 8)
```

```
Out[47]: array([8, 8, 8, 8, 8, 8, 8, 8, 8])
```

```
In [48]: # Create a 3x5 array filled with 3.14
np.full((3,5), 3.14)
```

```
Out[48]: 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]])
```

```
In [49]: np.full((3,5,2), 9)
```

```
Out[49]: array([[[9, 9],
 [9, 9],
 [9, 9],
 [9, 9],
 [9, 9]],

 [[[9, 9],
 [9, 9],
 [9, 9],
 [9, 9],
 [9, 9]],

 [[[9, 9],
 [9, 9],
 [9, 9],
 [9, 9],
 [9, 9]]]])
```

## การสร้าง empty array (อาร์เรย์เปล่า)

จะทำการสุ่มค่าสมาชิกใน array (ใช้ในกรณีที่ต้องการสร้างแค่ขนาดของข้อมูลที่ต้องการโดยไม่สนใจสมาชิก)

```
In [50]: np.empty(3)
```

Loading [MathJax]/extensions/Safe.js

```
Out[50]: array([8.4e-323, 8.9e-323, 9.4e-323])
```

```
In [51]: np.empty([15,3])
```

```
Out[51]: array([[0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.],
 [0., 0., 0.]])
```

```
In [52]: np.empty([2,5,7])
```

```
Out[52]: array([[[0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.]],
 
 [[[0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.],
 [0., 0., 0., 0., 0., 0., 0.]]])
```

## การสร้าง identity array

```
In [53]: # Create a 5x5 identity matrix
np.identity(5)
```

```
Out[53]: 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 [54]: # Create a 5x5 identity matrix
np.identity(5, dtype='int')
```

```
Out[54]: 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 [55]: # Create a 5x5 identity matrix
          np.eye(5)
```

```
Out[55]: 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 [56]: np.eye(5, dtype='int')
```

```
Out[56]: 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 [57]: np.eye(5, k=1)
```

```
Out[57]: array([[0., 1., 0., 0., 0.],
                 [0., 0., 1., 0., 0.],
                 [0., 0., 0., 1., 0.],
                 [0., 0., 0., 0., 1.],
                 [0., 0., 0., 0., 0.]])
```

```
In [58]: np.eye(5, k=-2)
```

```
Out[58]: array([[0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.],
                 [1., 0., 0., 0., 0.],
                 [0., 1., 0., 0., 0.],
                 [0., 0., 1., 0., 0.]])
```

```
In [59]: np.eye(3,4)
```

```
Out[59]: array([[1., 0., 0., 0.],
                 [0., 1., 0., 0.],
                 [0., 0., 1., 0.]])
```

```
In [60]: np.eye(3,4,k=1)
```

```
Out[60]: array([[0., 1., 0., 0.],
                 [0., 0., 1., 0.],
                 [0., 0., 0., 1.]])
```

```
In [61]: # Create an array of five values evenly spaced between 0 and 10
# np.linspace(start,stop,จำนวนที่ต้องการ)
np.linspace(0, 10, 20)
```

```
Out[61]: array([ 0.          ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
   2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
   5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
   7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])
```

```
In [62]: # np.linspace(start,stop)
# ถ้าไม่ใส่จำนวนที่ต้องการค่าdefault=50
np.linspace(0, 10)
```

```
Out[62]: array([ 0.          ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
   1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
   2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
   3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
   4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
   5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
   6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
   7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
   8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
   9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

```
In [63]: # np.linspace(start,stop,number)
# ถ้าไม่ใส่จำนวนที่ต้องการค่าdefault=50
# ถ้าไม่ต้องการค่า stop ให้ใส่ endpoint=False
np.linspace(0, 10, 10, endpoint=False)
```

```
Out[63]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

## การสร้าง array 1 มิติ โดยใช้คำสั่ง arange

```
In [64]: # Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# np.arange(start,stop,step)
# (this is similar to the built-in range() function)

np.arange(0, 10, 2)
```

```
Out[64]: array([0, 2, 4, 6, 8])
```

```
In [65]: np.arange(0, 10, 2, dtype='float')
```

```
Out[65]: array([0., 2., 4., 6., 8.])
```

```
In [66]: np.arange(0, 10, 2, dtype='complex')
```

Loading [MathJax]/extensions/Safe.js

```
Out[66]: array([0.+0.j, 2.+0.j, 4.+0.j, 6.+0.j, 8.+0.j])
```

```
In [67]: # np.arange(start,stop)
np.arange(-5, 10)
```

```
Out[67]: array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [68]: # np.arange(start,stop)
np.arange(-5, 10,dtype='float')
```

```
Out[68]: array([-5., -4., -3., -2., -1., 0., 1., 2., 3., 4., 5., 6., 7.,
8., 9.])
```

```
In [69]: # np.arange(stop)
np.arange(8)
```

```
Out[69]: array([0, 1, 2, 3, 4, 5, 6, 7])
```

## การสร้าง array โดยการสุ่มค่าตัวเลขใช้คำสั่ง random

```
In [70]: # Create a 3x3 array of uniformly distributed
# random values between 0 and 1

np.random.random((3,3))
```

```
Out[70]: array([[0.65808249, 0.73669105, 0.02741058],
[0.73585024, 0.30998174, 0.27841162],
[0.57849685, 0.34284511, 0.45238874]])
```

```
In [71]: # Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1

np.random.normal(0, 1, (3,3))
```

```
Out[71]: array([[-0.51209784, -0.64508255, 1.82499505],
[-0.99096173, 0.00288769, -0.79933835],
[ 0.14120029, 0.01602678, 0.01535607]])
```

```
In [72]: # Create a 3x3 array of random integers in the interval [0, 10)

np.random.randint(0, 10, (3,3))
```

```
Out[72]: array([[0, 0, 6],
[8, 0, 8],
[0, 2, 5]])
```

```
In [73]: np.random.randint(90, 100, 3)
```

Loading [MathJax]/extensions/Safe.js

```
Out[73]: array([94, 91, 92])
```

## NumPy Standard Data Types

NumPy รองรับรูปแบบชนิดของข้อมูลที่หลากหลายมากกว่า Python

ชนิดข้อมูลใน Python

- strings — ใช้แสดงข้อมูลที่เป็น text, ตัวอักษร โดยข้อมูลเหล่านี้จะต้องอยู่ข้างในเครื่องหมายคำพูด เช่น "ANN"
- integer — ใช้แสดงข้อมูลข้อมูลที่เป็นจำนวนเต็ม (เป็นได้ทั้งเต็มลบและเต็มบวก) เช่น 1, 2, -4, -10
- float — ใช้แสดงข้อมูลที่เป็นจำนวนจริง (จำนวนที่เป็นทศนิยม) เช่น 11.3, 2.4
- boolean — ใช้แสดงข้อมูลที่เป็น "จริง" หรือ "เท็จ"
- complex — ใช้แสดงจำนวนเชิงซ้อน เช่น 1.0 + 2.0j, 1.5 + 2.5j

ชนิดข้อมูลใน NumPy

- bool — ระบบจะใช้พื้นที่ในความจำไว้ 1 ในตัวสำหรับเก็บค่า boolean
- int\_ — ค่าเริ่มต้นของข้อมูลชนิด int
- intc — เมื่อกันกับ int ในภาษา C
- intp — integer ที่ใช้ในการ indexing เปรียบได้กับ ssize\_t
- int8 — byte (เริ่มที่ -128 ไปถึง 127)
- int16 — integer (เริ่มที่ -32768 ไปจนถึง 32767)
- int32 — integer (เริ่มที่ -2147483648 ไปจนถึง 2147483647)
- int64 — integer (เริ่มที่ -9223372036854775808 ไปจนถึง 922337203685477580)
- uint8 — unsigned integer (0 – 255)
- uint16 — unsigned integer (0 – 65535)
- uint32 — unsigned integer (0 – 4294967295)
- uint64 — unsigned integer (0 – 18446744073709551615)
- float\_ — เป็นการเขียน float64 แบบสั้น ตรงกับ float ใน builtin Python
- float16 — ครึ่งหนึ่งของ float; บิตเครื่องหมาย, เลขยกกำลังได้ 5 บิต, แมนทิสชาได้ 10 บิต

float128 — float หนึ่งตัว; บิตเครื่องหมาย, เลขยกกำลังได้ 8 บิต, แมนทิสชาได้ 23 บิต

- float64 — ตรงกับ float ใน builtin Python; บิตเครื่องหมาย, เลขยกกำลังได้ 11 บิต, แม่นทิลซ่าได้ 52 บิต
- complex\_ — เอียง complex128 แบบสั้น
- complex64 — เลขเชิงซ้อน แสดงด้วย float 32 บิต 2 ตัว
- complex128 — ตรงกับ complex ใน builtin Python แสดงด้วย float บิต

*Table 2-1. Standard NumPy data types*

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64
float16	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats

```
In [74]: #Return a new array of given shape and type, with random values
```

```
np.empty((3,3), dtype="int")
```

```
Out[74]: array([[-4620584249995491141, -4619386430262192074, 4610897866202437203],
 [-4616271027572021232, 4568804988641243445, -4617997017478485683],
 [4594255296740822083, 4580276627232348887, 4580005795403285181]])
```

```
In [75]: np.zeros(10, dtype="int16")
```

```
Loading [MathJax]/extensions/Safe.js 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int16)
```

```
In [76]: #or using the associated NumPy object:
```

```
np.zeros(10, dtype=np.int16)
```

```
Out[76]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int16)
```

```
In [77]: x = np.array([-128], dtype=np.int8)
```

```
print(x)
```

```
print(x-1)
```

```
[-128]
```

```
[127]
```

```
In [78]: np.iinfo(np.int8)
```

```
Out[78]: iinfo(min=-128, max=127, dtype=int8)
```

```
In [79]: np.iinfo(np.int_)
```

```
Out[79]: iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

## The Basics of NumPy Arrays

- We'll cover a few categories of basic array manipulations here:
- Attributes of arrays
  - Determining the size, shape, memory consumption, and data types of arrays
- Indexing of arrays
  - Getting and setting the value of individual array elements
- Slicing of arrays
  - Getting and setting smaller subarrays within a larger array
- Reshaping of arrays
  - Changing the shape of a given array
- Joining and splitting of arrays
  - Combining multiple arrays into one, and splitting one array into many

## NumPy Array Attributes

```
In [80]: #NumPy Array Attributes
```

```
#We'll use NumPy's random number generator, which we will seed with a set value
```

```
np.random.seed(0) # seed for reproducibility
```

```
x1 = np.random.randint(10, size=6) # One-dimensional array
```

Loading [MathJax]/extensions/Safe.js

x1

Out[80]: array([5, 0, 3, 3, 7, 9])

```
In [81]: ## Each array has attributes ndim (the number of dimensions), shape (the size
          of the array, and size (the total number of elements) and dtype (the data type of the
          elements). Other attributes include itemsize, which lists the size (in bytes) of each
          element and nbytes, which lists the total size (in bytes) of the array:
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) #it's same ((np.random.randint((0,10), size=6))
x2 = np.random.randint(10, size=(3,4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3,4,5)) # Three-dimensional array
```

In [82]: print(x1)

[5 0 3 3 7 9]

In [83]: print(x2)

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

In [84]: print(x3)

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

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

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

In [85]: print(x1)

```
print("x1 ndim: ",x1.ndim)
print("x1 shape: ",x1.shape)
print("x1 size: ",x1.size) #totally,6 elements

print("dtype: ",x1.dtype) #the data type of the array
# Other attributes include itemsize, which lists the size (in bytes) of each
# and nbytes, which lists the total size (in bytes) of the array:
print("itemsize:",x1.itemsize,"bytes")
print("nbytes:",x1.nbytes,"bytes")
```

```
[5 0 3 3 7 9]
x1 ndim: 1
x1 shape: (6,)
x1 size: 6
dtype: int64
itemsize: 8 bytes
nbytes: 48 bytes
```

```
In [86]: print(x2)
```

```
print("x2 ndim: ",x2.ndim)
print("x2 shape: ",x2.shape)
print("x2 size: ",x2.size) #totaly,12 elements

print("dtype: ",x2.dtype) #the data type of the array
print("itemsize:",x2.itemsize,"bytes")
print(" nbytes:",x2.nbytes,"bytes")
```

```
[[3 5 2 4]
 [7 6 8 8]
 [1 6 7 7]]
x2 ndim: 2
x2 shape: (3, 4)
x2 size: 12
dtype: int64
itemsize: 8 bytes
nbytes: 96 bytes
```

```
In [87]: print(x3)
```

```
print("x3 ndim: ",x3.ndim)
print("x3 shape: ",x3.shape)
print("x3 size: ",x3.size)#totaly,60 elements

print("dtype: ",x3.dtype) #the data type of the array
print("itemsize:",x3.itemsize,"bytes")
print(" nbytes:",x3.nbytes,"bytes")
```

```

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

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

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

x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size: 60
dtype: int64
itemsize: 8 bytes
nbytes: 480 bytes

```

## Array Indexing: Accessing Single Elements

- If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, you can access the  $i$ th value (counting from zero) by specifying the desired index in square brackets, just as with Python lists:

In [88]: `x1`

Out[88]: `array([5, 0, 3, 3, 7, 9])`

In [89]: `x1[0]`

Out[89]: `5`

In [90]: `x1[4]`

Out[90]: `7`

In [91]: *#To index from the end of the array, you can use negative indices:*

`x1[-1]`

Out[91]: `9`

In [92]: `x1[-2]`

Loading [MathJax]/extensions/Safe.js

- In a multidimensional array, you access items using a comma-separated tuple of indices:

In [93]: `x2`

```
Out[93]: array([[3, 5, 2, 4],
                 [7, 6, 8, 8],
                 [1, 6, 7, 7]])
```

In [94]: `x2[2,1]`

```
Out[94]: 6
```

In [95]: `x2[2,0]`

```
Out[95]: 1
```

In [96]: `x2[2,-4]`

```
Out[96]: 1
```

In [97]: `x2[-2,-3]`

```
Out[97]: 6
```

In [98]: `x2[-3,-2]`

```
Out[98]: 2
```

In [99]: *#You can also modify values using any of the above index notation:*

```
x2[0,0]=12
x2
```

```
Out[99]: array([[12, 5, 2, 4],
                 [ 7, 6, 8, 8],
                 [ 1, 6, 7, 7]])
```

In [100...]: `x1`

```
Out[100...]: array([5, 0, 3, 3, 7, 9])
```

In [101...]: `x1[0] = 3.14159 # this will be truncated!`  
`x1`

```
Out[101...]: array([3, 0, 3, 3, 7, 9])
```

## Array Slicing: Accessing Subarrays

Loading [MathJax]/extensions/Safe.js

- Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the slice notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array  $x$ , use this:
- $x[start:stop:step]$
- If any of these are unspecified, they default to the values  $start=0$ ,  $stop=size$  of dimension,  $step=1$ . We'll take a look at accessing subarrays in one dimension and in multiple dimensions.

## One-dimensional subarrays

```
In [102...]: x = np.arange(10)
x
```

```
Out[102...]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [103...]: x[:5] # first five elements
```

```
Out[103...]: array([0, 1, 2, 3, 4])
```

```
In [104...]: x[5:] # elements after index 5
```

```
Out[104...]: array([5, 6, 7, 8, 9])
```

```
In [105...]: x[4:7] # middle subarray
```

```
Out[105...]: array([4, 5, 6])
```

```
In [106...]: x[::2] # every other element
```

```
Out[106...]: array([0, 2, 4, 6, 8])
```

```
In [107...]: x[1::2] #every other element, starting at index 1
```

```
Out[107...]: array([1, 3, 5, 7, 9])
```

```
In [108...]: x[-7:-2:2]
```

```
Out[108...]: array([3, 5, 7])
```

```
In [109...]: x[-4:-2:1]
```

```
Out[109...]: array([6, 7])
```

```
In [110...]: # A potentially confusing case is when the step value is negative. In this case
# defaults for start and stop are swapped. This becomes a convenient way to
# slice arrays in reverse.
```

Loading [MathJax]/extensions/Safe.js

```
x[::-2] # all elements, reversed
```

Out[110... array([9, 7, 5, 3, 1])

```
In [111... x[5::-2]# reversed every other from index 5
```

Out[111... array([5, 3, 1])

```
In [112... x[5:1:-2]
```

Out[112... array([5, 3])

```
In [113... x[5:-8:-1]
```

Out[113... array([5, 4, 3])

```
In [114... x[7:-6:-1]
```

Out[114... array([7, 6, 5])

```
In [115... x[-7:-8:-1]
```

Out[115... array([3])

```
In [116... x[5:8:-1]
```

Out[116... array([], dtype=int64)

## Multidimensional subarrays

```
In [117... # Multidimensional slices work in the same way, with multiple slices separated by commas  
# For example:
```

```
x2
```

Out[117... array([[12, 5, 2, 4],  
[ 7, 6, 8, 8],  
[ 1, 6, 7, 7]])

```
In [118... # two rows, three columns
```

```
x2[:2, :3]
```

Out[118... array([[12, 5, 2],  
[ 7, 6, 8]])

```
In [119... # all rows, every other column
```

```
x2[:,::2]
```

Loading [MathJax]/extensions/Safe.js

```
Out[119... array([[12,  2],
                  [ 7,  8],
                  [ 1,  7]])
```

In [120... *#Finally, subarray dimensions can even be reversed together:*

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

```
Out[120... array([[ 7,  7,  6,  1],
                  [ 8,  8,  6,  7],
                  [ 4,  2,  5, 12]])
```

In [121... *# One commonly needed routine is accessing single*  
*# rows or columns of an array. You can do this by combining indexing and sli*  
*# using an empty slice marked by a single colon (:):*

```
print(x2[:, 0]) # first column of x2
```

```
[12  7  1]
```

In [122... *print(x2[0,:]) # first row of x2*

```
[12  5  2  4]
```

In [123... *#In the case of row access, the empty slice can be omitted for a more compact*

```
print(x2[0]) # equivalent to x2[0, :]
```

```
[12  5  2  4]
```

## Subarrays as no-copy views

- One important—and extremely useful—thing to know about array slices is that they return views rather than copies of the array data. This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies. Consider our two-dimensional array from before:

In [124... *print(x2)*

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

In [125... *#Let's extract a 2×2 subarray from this:*

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

```
[[12  5]
 [ 7  6]]
```

Tn [126 *#Now if we modify this subarray, we'll see that the original array is changed*

```
x2_sub[0,0] = 99
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

In [127... print(x2)

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

- This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

## Creating copies of arrays

- Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

In [128... x2\_sub\_copy = x2[:2,:2].copy()
print(x2\_sub\_copy)

```
[[99  5]
 [ 7  6]]
```

In [129... #If we now modify this subarray, the original array is not touched:

```
x2_sub_copy[0,0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

In [130... print(x2)

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

## Reshaping of Arrays

คือการจัดเรียงสมาชิกใน Array เลี้ยงใหม่ ให้มีขนาดและมิติตามที่กำหนด

In [131... # Another useful type of operation is reshaping of arrays. The most flexible
Loading [MathJax]/extensions/Safe.js is is with the reshape() method. For example, if you want to put t
# I through 9 in a 3x3 grid, you can do the following:

```
grid = np.arange(1,10,1).reshape(3,3)
print(grid)

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

In [132... # จาก code ด้านบน ถ้ามาลองทำให้ละขั้นตอนดังนี้  
# ขั้นที่ 1 สร้าง grid  
grid1 = np.arange(1,10,1)  
grid1

Out[132... array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [133... # ขั้นที่ 2 reshape grid  
grid1.reshape(3,3)

Out[133... array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])

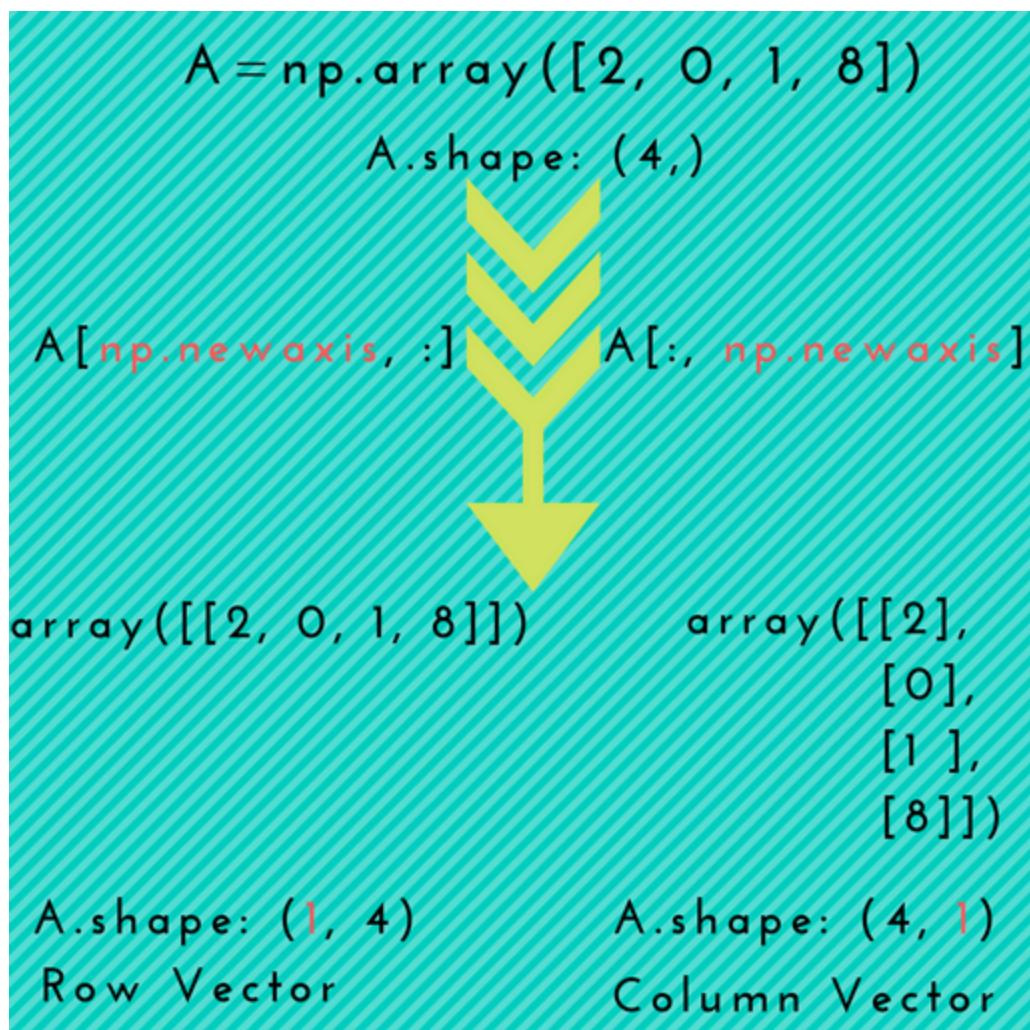
In [134... # คำสั่ง reshape จะเป็นการจัดเรียงสมาชิกให้มีขนาดตามที่ต้องการ  
# โดยเป็นการเปลี่ยนแปลงรูปร่างชั่วคราว  
# แต่ในส่วนของตัวแปร grid1 ข้อมูลใน array ที่ยังคงรูปร่างเดิม  
grid1

Out[134... array([1, 2, 3, 4, 5, 6, 7, 8, 9])

In [135... # ถ้าต้องการให้ ข้อมูลใน grid1 เปลี่ยนรูปร่างไปเลยเรา ก็ต้อง  
# สร้างตัวแปรมารับค่าดังนี้  
grid2 = grid1.reshape(3,3)  
grid2

Out[135... array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])

รูปแบบการ reshape ทั่วไปอีกแบบหนึ่งคือการแปลงอาร์เรย์มิติเดียวเป็นเมทริกซ์และหรือ colums สอง มิติ คุณสามารถทำได้โดยใช้วิธีการ reshape หรือทำได้ง่ายขึ้นโดยใช้คีย์เวิร์ด newaxis ภายในการ ดำเนินการสไลซ์(:) ดังรูป



In [136...]: `A = np.array([2, 0, 1, 8])  
A.shape # x is a vector (4,)`

Out[136...]: (4,)

In [137...]: `A`

Out[137...]: array([2, 0, 1, 8])

In [138...]: `# row vector via reshape  
A.reshape(1,4)`

Out[138...]: array([[2, 0, 1, 8]])

In [139...]: `# row vector via newaxis  
A[np.newaxis, :]`

Out[139...]: array([[2, 0, 1, 8]])

In [140...]: `A[np.newaxis, :].shape`  
Loading [MathJax]/extensions/Safe.js

```
Out[140... (1, 4)
```

```
In [141... # column vector via reshape
A.reshape(4, 1)
```

```
Out[141... array([[2],
[0],
[1],
[8]])
```

```
In [142... # column vector via newaxis
A[:, np.newaxis]
```

```
Out[142... array([[2],
[0],
[1],
[8]]))
```

```
In [143... A[:, np.newaxis].shape
```

```
Out[143... (4, 1)
```

## Array Concatenation and Splitting

### Concatenation of arrays

```
In [144... x = np.array([1,2,3])
y = np.array([3,2,1])
np.concatenate((x, y))
```

```
Out[144... array([1, 2, 3, 3, 2, 1])
```

- You can also concatenate more than two arrays at once:

```
In [145... z = np.array([9,99,999]) #z =[9,99,999]
np.concatenate((x,y,z))
```

```
Out[145... array([ 1,   2,   3,   3,   2,   1,   9,  99, 999])
```

- np.concatenate can also be used for two-dimensional arrays:

```
In [146... grid = np.array([[1,2,3],
[4,5,6]])
grid
```

Loading [MathJax]/extensions/Safe.js

```
Out[146... array([[1, 2, 3],
                  [4, 5, 6]])
```

```
In [147... # concatenate along the first axis

np.concatenate((grid,grid))
```

```
Out[147... array([[1, 2, 3],
                  [4, 5, 6],
                  [1, 2, 3],
                  [4, 5, 6]])
```

```
In [148... # concatenate along the first axis

np.concatenate((grid,grid), axis=0)
```

```
Out[148... array([[1, 2, 3],
                  [4, 5, 6],
                  [1, 2, 3],
                  [4, 5, 6]])
```

```
In [149... # concatenate along the second axis (zero-indexed)

np.concatenate((grid, grid), axis=1)
```

```
Out[149... array([[1, 2, 3, 1, 2, 3],
                  [4, 5, 6, 4, 5, 6]])
```

```
In [150... # For working with arrays of mixed dimensions,
#       it can be clearer to use the np.vstack (vertical stack)
#       and np.hstack (horizontal stack) functions:

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

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

```
Out[150... array([[1, 2, 3],
                  [9, 8, 7],
                  [6, 5, 4]])
```

```
In [151... np.vstack([grid,x])
```

```
Out[151... array([[9, 8, 7],
                  [6, 5, 4],
                  [1, 2, 3]])
```

```
In [152... #horizontally stack the arrays

y = np.array([[99],
              [99]])
np.hstack([grid,y])
```

Loading [MathJax]/extensions/Safe.js

```
Out[152... array([[ 9,  8,  7, 99],
   [ 6,  5,  4, 99]])
```

```
In [153... np.hstack([y,grid])
```

```
Out[153... array([[99,  9,  8,  7],
   [99,  6,  5,  4]])
```

- Similarly, np.dstack will stack arrays along the third axis.

## Splitting of arrays

- The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

```
In [154... x = [1,2,3,99,99,3,2,1]
x1, x2, x3 = np.split(x, [3,5])
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

```
In [155... x = np.array([1,2,3,99,99,3,2,1])
x1, x2, x3, x4 = np.split(x, [3,4,5])
print(x1, x2, x3,x4)
```

```
[1 2 3] [99] [99] [3 2 1]
```

- Notice that N split points lead to N + 1 subarrays. The related functions np.hsplit and np.vsplit are similar:

```
In [156... grid = np.arange(36, dtype=np.float64).reshape((6,6))
grid
```

```
Out[156... array([[ 0.,  1.,  2.,  3.,  4.,  5.],
   [ 6.,  7.,  8.,  9., 10., 11.],
   [12., 13., 14., 15., 16., 17.],
   [18., 19., 20., 21., 22., 23.],
   [24., 25., 26., 27., 28., 29.],
   [30., 31., 32., 33., 34., 35.]])
```

```
In [157... upper, lower = np.vsplit(grid, [2])
```

```
In [158... print(upper)
```

```
[[ 0.  1.  2.  3.  4.  5.]
 [ 6.  7.  8.  9. 10. 11.]]
```

```
In [159... print(lower)
```

Loading [MathJax]/extensions/Safe.js

```
[[12. 13. 14. 15. 16. 17.]  
 [18. 19. 20. 21. 22. 23.]  
 [24. 25. 26. 27. 28. 29.]  
 [30. 31. 32. 33. 34. 35.]]
```

```
In [160]: upper, middle, lower = np.vsplit(grid, [2,3])  
  
print("upper: ",upper)  
print("middle: ",middle)  
print("lower: ",lower)
```

```
upper: [[ 0.  1.  2.  3.  4.  5.]  
 [ 6.  7.  8.  9. 10. 11.]]  
middle: [[12. 13. 14. 15. 16. 17.]]  
lower: [[18. 19. 20. 21. 22. 23.]  
 [24. 25. 26. 27. 28. 29.]  
 [30. 31. 32. 33. 34. 35.]]
```

```
In [161]: left, right = np.hsplit(grid, [2])  
print("left: ",left)  
print("right: ",right)
```

```
left: [[ 0.  1.]  
 [ 6.  7.]  
 [12. 13.]  
 [18. 19.]  
 [24. 25.]  
 [30. 31.]]  
right: [[ 2.  3.  4.  5.]  
 [ 8.  9. 10. 11.]  
 [14. 15. 16. 17.]  
 [20. 21. 22. 23.]  
 [26. 27. 28. 29.]  
 [32. 33. 34. 35.]]
```

```
In [162]: left, right, g = np.hsplit(grid, 3)  
print("left: ",left)  
print("right: ",right)  
print("g: ",g)
```

```
left:  [[ 0.  1.]
 [ 6.  7.]
 [12. 13.]
 [18. 19.]
 [24. 25.]
 [30. 31.]]
right: [[ 2.  3.]
 [ 8.  9.]
 [14. 15.]
 [20. 21.]
 [26. 27.]
 [32. 33.]]
g:   [[ 4.  5.]
 [10. 11.]
 [16. 17.]
 [22. 23.]
 [28. 29.]
 [34. 35.]]
```

```
In [163]: np.hsplit(grid, 3)
```

```
Out[163]: [array([[ 0.,  1.],
 [ 6.,  7.],
 [12., 13.],
 [18., 19.],
 [24., 25.],
 [30., 31.]]),
 array([[ 2.,  3.],
 [ 8.,  9.],
 [14., 15.],
 [20., 21.],
 [26., 27.],
 [32., 33.]]),
 array([[ 4.,  5.],
 [10., 11.],
 [16., 17.],
 [22., 23.],
 [28., 29.],
 [34., 35.]])]
```

- Similarly, `np.dsplit` will split arrays along the third axis.

## Computation on NumPy Arrays: Universal Functions

Ufunc (Universal Function) ใน NumPy เป็นฟังก์ชันที่ทำงานบนอาร์เรย์แบบองค์ประกอบต่อองค์ประกอบ (element-wise) ซึ่งหมายความว่ามันจะประมวลผลแต่ละองค์ประกอบในอาร์เรย์แยกกัน และส่งผลลัพธ์กลับมาในรูปแบบของอาร์เรย์ที่มีขนาดเดียวกัน Ufunc เป็นฟังก์ชันที่สำคัญมากใน NumPy เพราะช่วยให้การคำนวณทางคณิตศาสตร์และโครงสร้างอาร์เรย์ทำได้อย่างมีประสิทธิภาพและรวดเร็ว

## Exploring NumPy's UFuncs

- Ufuncs exist in two flavors: unary ufuncs, which operate on a single input, and binary ufuncs, which operate on two inputs. We'll see examples of both these types of functions here.

### Array arithmetic

In [164...]

```
# NumPy's ufuncs feel very natural to use because
#   they make use of Python's native arithmetic operators.
#   The standard addition, subtraction, multiplication, and
#   division can all be used:
```

```
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]
```

In [165...]

```
#There is also a unary ufunc for negation,
#   a ** operator for exponentiation, and
#   a % operator for modulus:
print("x =", x)
print("-x =", -x)
print("x ** 2 =", x ** 2)
print("x % 2 =", x % 2)
```

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

Loading [MathJax]/extensions/Safe.js ion, these can be strung together however you wish,  
# and the standard order of operations is respected:

```

print("x =", x)
-(0.5*x+1) ** 2

x = [0 1 2 3]
Out[166... array([-1. , -2.25, -4. , -6.25])

```

```

In [167... # All of these arithmetic operations are simply
#     convenient wrappers around specific functions
#     built into NumPy; for example,
#     the + operator is a wrapper for the add function:

print(np.add(3,2))
print("x =", x)
print(np.add(x,2)) #Addition +
print(np.subtract(x,5)) #Subtraction -
print(np.negative(x)) #Unary negation -
print(np.multiply(x,3)) #Multiplication *
print(np.divide(x,2)) #Division /
print(np.floor_divide(x,2)) #Floor division //
print(np.power(x,2)) #Exponentiation **
print(np.mod(x,2)) #Modulus/remainder **

print(np.multiply(x, x))

5
x = [0 1 2 3]
[2 3 4 5]
[-5 -4 -3 -2]
[ 0 -1 -2 -3]
[0 3 6 9]
[0. 0.5 1. 1.5]
[0 0 1 1]
[0 1 4 9]
[0 1 0 1]
[0 1 4 9]

```

## Absolute value

```

In [168... # Just as NumPy understands Python's built-in
#     arithmetic operators, it also understands
#     Python's built-in absolute value function:

x = np.array([-2,-1,0,1,2])
abs(x)

```

```
Out[168... array([2, 1, 0, 1, 2])
```

```

In [169... # The corresponding NumPy ufunc is np.absolute,
#     which is also available under the alias np.abs:

print(np.absolute(x))
print(np.abs(x))

```

Loading [MathJax]/extensions/Safe.js

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

```
In [170...]: # This ufunc can also handle complex data,
#           in which the absolute value returns the magnitude:

x = np.array([7-24j, 4-3j, 2+0j, 1+3j])
np.abs(x)
```

```
Out[170...]: array([25.0, 5.0, 2.0, 3.16227766])
```

## Trigonometric functions

```
In [171...]: # NumPy provides a large number of useful ufuncs, and some of the most useful
# data scientist are the trigonometric functions. We'll start by defining angles:

theta = np.linspace(0,np.pi,3)

#Now we can compute some trigonometric fuctions on these values:
print("theta      =",theta)
print("sin(theta) =",np.sin(theta))
print("cos(theta) =",np.cos(theta))
print("tan(theta) =",np.tan(theta))

theta      = [0.          1.57079633 3.14159265]
sin(theta) = [0.000000e+00 1.000000e+00 1.2246468e-16]
cos(theta) = [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) = [ 0.0000000e+00  1.63312394e+16 -1.22464680e-16]
```

```
In [172...]: x = [-1, 0, 1]

print("x = ", x)
print("arcsin(x) = ", np.arcsin(x))
print("arccos(x) = ", np.arccos(x))
print("arctan(x) = ", np.arctan(x))

x = [-1, 0, 1]
arcsin(x) = [-1.57079633  0.          1.57079633]
arccos(x) = [3.14159265 1.57079633 0.          ]
arctan(x) = [-0.78539816  0.          0.78539816]
```

## Exponents and logarithms

- Another common type of operation available in a NumPy ufunc are the exponentials:

```
In [173...]: x = [1,2,3]
print("x      =",x)
print("      =",np.exp(x))
```

```

print("2^x    =" , np.exp2(x))
print("3^x    =" , np.power(3,x))

x      = [1, 2, 3]
e^x    = [ 2.71828183  7.3890561  20.08553692]
2^x    = [2. 4. 8.]
3^x    = [ 3  9 27]

```

In [174...]: # The inverse of the exponentials, the logarithms, are also available. The `log` function gives the natural logarithm; if you prefer to compute the base-2 logarithm or the base-10 logarithm, these are available as well:

```

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))

x      = [1, 2, 4, 10]
ln(x)  = [0.          0.69314718 1.38629436 2.30258509]
log2(x) = [0.          1.          2.          3.32192809]
log10(x) = [0.          0.30103    0.60205999 1.          ]

```

In [175...]: # There are also some specialized versions that are useful for maintaining precision when working with very small input:

```

x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =" , np.expm1(x))
print("log(1 + x) =" , np.log1p(x))

exp(x) - 1 = [0.          0.0010005  0.01005017 0.10517092]
log(1 + x) = [0.          0.0009995  0.00995033 0.09531018]

```

In [176...]: # Polynomial functions

```

x = np.linspace(0, 10, num=25)
y = x**2 - 3*x + 1

print(x)
print(y)

[ 0.          0.41666667  0.83333333  1.25          1.66666667  2.08333333
 2.5          2.91666667  3.33333333  3.75          4.16666667  4.58333333
 5.          5.41666667  5.83333333  6.25          6.66666667  7.08333333
 7.5          7.91666667  8.33333333  8.75          9.16666667  9.58333333
 10.          ]
[ 1.          -0.07638889 -0.80555556 -1.1875        -1.22222222 -0.90972222
 -0.25          0.75694444  2.11111111  3.8125        5.86111111  8.25694444
 11.          14.09027778 17.52777778 21.3125        25.44444444 29.92361111
 34.75         39.92361111 45.44444444 51.3125        57.52777778 64.09027778
 71.          ]

```

## Advanced Ufunc Features

Loading [MathJax]/extensions/Safe.js

## Aggregates

- For binary ufuncs, there are some interesting aggregates that can be computed directly from the object. For example, if we'd like to reduce an array with a particular operation, we can use the reduce method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains. For example, calling reduce on the add ufunc returns the sum of all elements in the array:

```
In [177]: x = np.arange(1,6)
print('x = ', x)
print(np.add.reduce(x)) # เป็นฟังก์ชันที่ใช้รวม(reduce) ข้อมูลในอาร์เรย์ x โดยการบวก(add)
print(np.subtract.reduce(x)) # เป็นฟังก์ชันที่ใช้รวม(reduce) ข้อมูลในอาร์เรย์ x โดยการลบ(subtract)
print(np.multiply.reduce(x)) # เป็นฟังก์ชันที่ใช้รวม(reduce) ข้อมูลในอาร์เรย์ x โดยการคูณ(multiply)
print(np.divide.reduce(x)) ## เป็นฟังก์ชันที่ใช้รวม(reduce) ข้อมูลในอาร์เรย์ x โดยการหาร(divide)

x = [1 2 3 4 5]
15
-13
120
0.00833333333333333
```

```
In [178]: #If we'd like to store all the intermediate results of
#   the computation, we can instead use accumulate:
x = np.arange(1,6)
print('x = ', x)
print(np.add.accumulate(x)) # เป็นฟังก์ชันที่ใช้คำนวณผลรวมสะสม(cumulative sum) ของข้อมูล
print(np.subtract.accumulate(x)) # เป็นฟังก์ชันที่ใช้คำนวณผลลบสะสม(cumulative subtraction)
print(np.multiply.accumulate(x)) # เป็นฟังก์ชันที่ใช้คำนวณผลคูณสะสม(cumulative multiplication)
print(np.divide.accumulate(x)) # เป็นฟังก์ชันที่ใช้คำนวณผลหารสะสม(cumulative division)
print(np.floor_divide.accumulate(x)) # เป็นฟังก์ชันที่ใช้คำนวณผลหารบัดเศษลงสะสม(cumulative floor division)
print(np.power.accumulate(x)) # เป็นฟังก์ชันที่ใช้คำนวณผลยกกำลังสะสม(cumulative power)

x = [1 2 3 4 5]
[ 1  3  6  10  15]
[ 1  -1  -4  -8  -13]
[ 1  2  6  24  120]
[1.          0.5          0.16666667  0.04166667  0.00833333]
[1 0 0 0 0]
[1 1 1 1 1]
```

## Outer products

- Finally, any ufunc can compute the output of all pairs of two different inputs using the outer method. This allows you, in one line, to do things like create a multiplication table:

Loading [MathJax]/extensions/Safe.js

```
In [179...]: x = np.arange(1,6)
print(np.multiply.outer(x, x))
print(np.add.outer(x, x))
```

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

## Aggregations: Min, Max, and Everything in Between

- NumPy has fast built-in aggregation functions for working on arrays; we'll discuss and demonstrate some of them here.

### Summing the Values in an Array

```
In [180...]: # As a quick example, consider computing the sum of all
#      values in an array. Python itself can do this using
#      the built-in sum function:

L = np.random.random(100)
sum(L)
```

Out[180...]: 52.12818058833704

```
In [181...]: #The syntax is quite similar to that of NumPy's sum function,
#      and the result is the same in the simplest case:

np.sum(L)
```

Out[181...]: 52.12818058833702

```
In [182...]: # However, because it executes the operation in compiled code, NumPy's versi
#      operation is computed much more quickly:

big_array = np.random.random(1000000)
```

Loading [MathJax]/extensions/Safe.js  
n(big\_array)

27.3 ms  $\pm$  315  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

In [184...]: %timeit np.sum(big\_array)

117  $\mu$ s  $\pm$  412 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)

## Minimum and Maximum

In [185...]: #Similarly, Python has built-in min and max functions,  
# used to find the minimum value  
# and maximum value of any given array:  
  
min(big\_array), max(big\_array)

Out[185...]: (1.4057692298008462e-06, 0.9999994392723005)

In [186...]: #NumPy's corresponding functions have similar syntax,  
# and again operate much more quickly:  
  
np.min(big\_array), np.max(big\_array)

Out[186...]: (1.4057692298008462e-06, 0.9999994392723005)

In [187...]: %timeit min(big\_array)  
%timeit np.min(big\_array)

19.2 ms  $\pm$  222  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)  
75.7  $\mu$ s  $\pm$  932 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)

In [188...]: # For min, max, sum, and several other NumPy aggregates,  
# a shorter syntax is to use methods of the  
# array object itself:  
  
print(big\_array.min(), big\_array.max(), big\_array.sum())

1.4057692298008462e-06 0.9999994392723005 500202.5348847683

In [189...]: # Whenever possible, make sure that you are using the NumPy  
# version of these aggregates when operating on NumPy  
# arrays!  
%timeit np.min(big\_array)  
%timeit big\_array.min()

78.1  $\mu$ s  $\pm$  448 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)  
76.1  $\mu$ s  $\pm$  868 ns per loop (mean  $\pm$  std. dev. of 7 runs, 10,000 loops each)

## Multidimensional aggregates

In [190...]: # One common type of aggregation operation is an aggregate along a row or column.  
# Say you have some data stored in a two-dimensional array:

Loading [MathJax]/extensions/Safe.js

```
M = np.random.randint(1,10,(3,4))
print(M)
```

```
[[1 8 2 8]
 [7 2 8 7]
 [8 5 7 2]]
```

In [191... M.sum()

Out[191... 65

In [192... M.sum(axis=0)

Out[192... array([16, 15, 17, 17])

In [193... M.sum(axis=1)

Out[193... array([19, 24, 22])

In [194... # Aggregation functions take an additional argument
# specifying the axis along which the aggregate
# is computed. For example, we can find the minimum value
# within each column by specifying axis=0:

```
M.min(axis=0)
```

Out[194... array([1, 2, 2, 2])

In [195... #or use that way

```
np.min(M, axis=0)
```

Out[195... array([1, 2, 2, 2])

In [196... # Similarly, we can find the maximum value within each row:

```
M.max(axis=1)
```

Out[196... array([8, 8, 8])

np.nan หมายถึง "Not a Number" ใน NumPy ซึ่งใช้แทนค่าที่ไม่สามารถคำนวณเป็นตัวเลขได้ หรือใช้เป็นสัญลักษณ์สำหรับข้อมูลที่ขาดหายไป (missing value) หรือไม่สามารถนิยามค่าได้ในเชิงคณิตศาสตร์ มีความสำคัญในงานวิเคราะห์ข้อมูล เนื่องจากช่วยให้สามารถจัดการกับข้อมูลที่ไม่สมบูรณ์ หรือขาดหายไปได้อย่างเหมาะสม

In [197... # Note that some of these NaN-safe functions were not added until
# NumPy 1.8, so they will not be available in older NumPy versions.

```
x = np.array([1,2,np.nan,4,5])
```

```
print('x = ', x)
```

Loading [MathJax]/extensions/Safe.js sum =", np.sum(x))

```
print("np.nansum      =" , np.nansum(x))

print("np.mean       =" , np.mean(x))
print("np.nanmean    =" , np.nanmean(x))

print("np.std        =" , np.std(x))
print("np.nanstd     =" , np.nanstd(x))

#Be careful that this is not a real index of minimum value.
print("np.argmin     =" , np.argmin(x))
#if there is a nan value in an array, it returns index of nan value.

print("np.nanargmin  =" , np.nanargmin(x))

#Be careful that this is not a real index of minimum value.
print("np.argmax     =" , np.argmax(x))
#if there is a nan value in an array, it returns index of nan value.

print("np.nanargmax  =" , np.nanargmax(x))

x = [ 1.  2.  nan  4.  5.]
np.sum      = nan
np.nansum   = 12.0
np.mean     = nan
np.nanmean  = 3.0
np.std      = nan
np.nanstd   = 1.5811388300841898
np.argmin   = 2
np.nanargmin= 0
np.argmax   = 2
np.nanargmax= 4
```

*Table 2-3. Aggregation functions available in NumPy*

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 median 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.argmax	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

In [ ]:

# Computation on Arrays: Broadcasting

- Broadcasting is simply a set of rules for applying binary ufuncs (addition, subtraction, multiplication, etc.) on arrays of different sizes.

## Introducing Broadcasting

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

```
Out[1]: array([5, 6, 7])
```

```
In [2]: a+np.array([5])
```

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

```
In [3]: a+np.array(5)
```

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

```
In [4]: a+5
```

```
Out[4]: array([5, 6, 7])
```

## Broadcasting

- เมื่อนำอารเรย์ 2 ตัวมาคำนวณแบบ element-wise แต่อารเรย์ทั้งสองมีขนาดไม่เท่ากัน
- ระบบจะ broadcast อารเรย์ตัวเล็ก (หรืออาจทำทั้งสองตัว) ให้มีขนาดเท่ากันก่อนทำงาน

$$\begin{bmatrix} 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 3 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

- แต่บางครั้งก็ broadcast ไม่ได้

$$\begin{bmatrix} 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \quad \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

### ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
u = np.array([2]) + x
```

$$\begin{bmatrix} 2 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$\begin{matrix} (1) \\ (3, 2) \end{matrix}$$

## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
u = np.array([2]) + x
```

$$\begin{bmatrix} 2 & 2 \\ 2 & 2 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

(3, 1)  (3, 2)  
 (3, 2)



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
u = np.array([2]) + x
```

$$\begin{bmatrix} 2+1 & 2+2 \\ 2+3 & 2+4 \\ 2+5 & 2+6 \end{bmatrix}$$

(3, 1)  (3, 2)  
 (3, 2)



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
w = np.array([10 20]) + x
```

$$\begin{bmatrix} 10 & 20 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

(2)  
(3, 2)



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
w = np.array([10 20]) + x
```

$$\begin{bmatrix} 10 & 20 \\ 10 & 20 \\ 10 & 20 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

(3, 2) (2) → (3, 2)



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
w = np.array([10 20]) + x
```

$$\begin{bmatrix} 10+1 & 20+2 \\ 10+3 & 20+4 \\ 10+5 & 20+6 \end{bmatrix}$$

 (2)  (3, 2)  
(3, 2)



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
v = np.array([[10],[20],[30]]) + x
```

$$\begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

(3, 1)  
(3, 2)



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
v = np.array([[10],[20],[30]]) + x
```

$$\begin{bmatrix} 10 & 10 \\ 20 & 20 \\ 30 & 30 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$(3, 1) \rightarrow (3, 2)$$

$$(3, 2) \rightarrow (3, 2)$$



## ตัวอย่าง: broadcast ตัวเล็ก ให้เท่าตัวใหญ่

```
x = np.array([[1,2],[3,4],[5,6]])
v = np.array([[10],[20],[30]]) + x
```

$$\begin{bmatrix} 10+1 & 10+2 \\ 20+3 & 20+4 \\ 30+5 & 30+6 \end{bmatrix}$$

$$(3, 1) \rightarrow (3, 2)$$

$$(3, 2) \rightarrow (3, 2)$$



## ตัวอย่าง: broadcast ทั้ง 2 ตัว

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + [4 \quad 5]$$

(3, 1)  
(2)



## ตัวอย่าง: broadcast ทั้ง 2 ตัว

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + [4 \quad 5] \rightarrow \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} + [4 \quad 5]$$

(3, 1)  
(2)

(3, 2)  
(2)



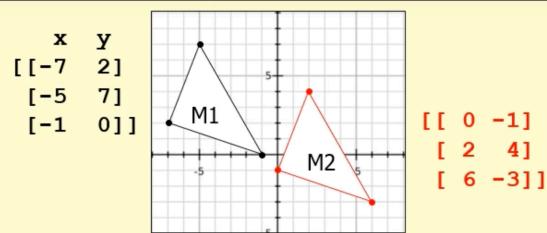
## ตัวอย่าง: broadcast ทั้ง 2 ตัว

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + [4 \quad 5] \rightarrow \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} + [4 \quad 5] \rightarrow \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} + \begin{bmatrix} 4 & 5 \\ 4 & 5 \\ 4 & 5 \end{bmatrix}$$

$(3, \frac{1}{2})$        $(3, \frac{2}{2})$        $(3, \frac{2}{2})$



## ตัวอย่าง: Matrix Translation



ต้องการย้ายทุกจุดไปทางขวา 7, ลงล่าง 3 คือบวกทุกด้วย  $[7, -3]$

M1 กับ T  
มีขนาดเท่ากัน

```
M1 = np.array([[-7, 2], [-5, 7], [-1, 0]])
T = np.array([[7, -3], [7, -3], [7, -3]])
M2 = M1 + T
```

เขียนแค่นี้พอ

```
M2 = M1 + np.array([7, -3])
```

$$\begin{bmatrix} -7 & 2 \\ -5 & 7 \\ -1 & 0 \end{bmatrix} + [7, -3] \rightarrow \begin{bmatrix} -7 & 2 \\ -5 & 7 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 7 & -3 \\ 7 & -3 \\ 7 & -3 \end{bmatrix}$$

## นอกเรื่อง : กฎการ Broadcasting

```
A = np.array([[1, 2, 3]]) # shape - (3, 1)
B = np.array([1, 2]) # shape - ( 2)
C = A + B # shape - (3, 2)
```

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + [1 \quad 2] = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 1 & 2 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1+1 & 1+2 \\ 2+1 & 2+2 \\ 3+1 & 3+2 \end{bmatrix}$$

shape ของสองอาร์เรย์ไม่เท่ากัน แต่เกิด broadcasting ได้ เมื่อเปรียบเทียบมิติจากขวาไปซ้ายแล้ว

- มีมิติเท่ากัน (เกิด broadcast ตรงที่ไม่มี)

A.shape (3, 2, 4)  
B.shape ( 2, 4) ได้ผลที่มี shape (3, 2, 4)

- มีมิติของอาร์เรย์หนึ่งเป็น 1 (เกิด broadcast ตรงที่เป็น 1)

A.shape (3, 1, 5)  
B.shape (1, 2, 1) ได้ผลที่มี shape (3, 2, 5)

- ด.ย. ที่ broadcast ไม่ได้

A.shape (2, 4)              A.shape (3, 4, 5)  
B.shape (3,)              B.shape ( 2, 5)

In [5]: *# We can similarly extend this to arrays of higher dimension. Observe the re  
# we add a one-dimensional array to a two-dimensional array:*

```
M = np.ones((3,3))
M
```

Out[5]: array([[1., 1., 1.],
 [1., 1., 1.],
 [1., 1., 1.]])

In [6]: a = np.array([0,1,2])
a

Out[6]: array([0, 1, 2])

In [7]: M+a
*# Here the one-dimensional array a is stretched, or broadcast, across the se  
# dimension in order to match the shape of M .*

Out[7]: array([[1., 2., 3.],
 [1., 2., 3.],
 [1., 2., 3.]])

In [8]: c = np.arange(3).reshape((3,1))
c

Out[8]: array([[0],
 [1],
 [2]])

Loading [MathJax]/extensions/Safe.js

```
In [9]: d = np.arange(3)
d
```

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

```
In [10]: c+d
```

```
Out[10]: array([[0, 1, 2],
                 [1, 2, 3],
                 [2, 3, 4]])
```

## Visualization of NumPy broadcasting

`np.arange(3)+5`

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 5 & 5 & 5 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 5 & 6 & 7 \\ \hline \end{array}$$

`np.ones((3, 3))+np.arange(3)`

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline \end{array}$$

`np.arange(3).reshape((3, 1))+np.arange(3)`

$$\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline \end{array} + \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline 0 & 1 & 2 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 1 & 2 & 3 \\ \hline 2 & 3 & 4 \\ \hline \end{array}$$

## Comparisons, Masks, and Boolean Logic

### Comparison Operators as ufuncs

- The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:
- for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

Loading [MathJax]/extensions/Safe.js

```
In [11]: x = np.array([1,2,3,4,5])

print(x<3) # less than
print(x>3) # greater than
print(x<=3) # less than or equal
print(x>=3) # greater than or equal
print(x!=3) # not equal
print(x==3) # equal

[ True  True False False False]
[False False False  True  True]
[ True  True  True False False]
[False False  True  True  True]
[ True  True False  True  True]
[False False  True False False]
```

```
In [12]: # It is also possible to do an element-by-element comparison of two arrays,
# include compound expressions:

(2*x) == (2**x)
```

```
Out[12]: array([ True,  True, False, False, False])
```

```
In [13]: # As in the case of arithmetic operators, the comparison operators are implemented
# ufuncs in NumPy; for example, when you write x < 3 , internally NumPy uses
# np.less(x, 3) . A summary of the comparison operators and their equivalent
# is shown here:
```

## Comparison operators and their equivalent

Operator	Equivalent ufunc
==	np.equal
!=	np.not_equal
<	np.less
<=	np.less_equal
>	np.greater
>=	np.greater_equal

```
In [14]: x = np.random.randint(10, size=(3,4))
print(x)

x<6
```

```
[[2 6 5 0]
 [7 9 4 0]]
```

Loading [MathJax]/extensions/Safe.js

```
Out[14]: array([[ True, False,  True,  True],
       [False, False,  True,  True],
       [ True, False,  True,  True]])
```

## Working with Boolean Arrays

```
In [15]: print(x)

# To count the number of True entries in a Boolean array,
#     np.count_nonzero is useful:
# np.count_nonzero เป็นฟังก์ชันใน NumPy ที่ใช้สำหรับนับจำนวนของสมาชิกใน array
#     ที่มีค่า "ไม่เท่ากับ 0" หรือในทางอื่นคือ นับจำนวนสมาชิกที่เป็น "True"
#     ในบริบทของการตีความเป็น Boolean(ซึ่ง 0 ถูกมองว่าเป็น False
#     และค่าที่ไม่ใช่ 0 ถูกมองว่าเป็น True)

# how many values less than 6?
print("A: ",np.count_nonzero(x<6))

# We see that there are eight array entries that are less than 6.
#     Another way to get at this information is to use np.sum ;
#     in this case, False is interpreted as 0 ,
#     and True is interpreted as 1 :

print("B: ",np.sum(x<6))

print("C: ",np.sum(x!=np.nan))
print("D: ",np.count_nonzero(x!=np.nan))

[[2 6 5 0]
 [7 9 4 0]
 [4 6 4 2]]
A: 8
B: 8
C: 12
D: 12
```

```
In [16]: # how many values less than 6 in each row?
print(np.sum(x < 6, axis=1))

# how many values less than 6 in each column?
print(np.sum(x < 6, axis=0))
```

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

```
In [17]: # If we're interested in quickly checking whether any or all
#     the values are true, we can use (you guessed it)
#     np.any() or np.all():

# are there any values greater than 8?
print(np.any(x>8))
```

Loading [MathJax]/extensions/Safe.js  
any values less than zero?  
print(np.any(x<0))

```
# are all values less than 10?
print(np.all(x<10))

# are all values equal to 6?
print(np.all(x==6))
```

True  
False  
True  
False

In [18]: # are all values in each row less than 8?  
print(np.all(x<8, axis=1))

```
# are all values in each column less than 3?
print(np.all(x<3, axis=0))
```

[ True False True]  
[False False False True]

In [19]: print(x)  
print(x<5)  
print(x[x<5])

```
[[2 6 5 0]
 [7 9 4 0]
 [4 6 4 2]]
 [[ True False False True]
 [False False True True]
 [ True False True True]]
 [2 0 4 0 4 4 2]
```

In [20]: # In Python, all nonzero integers will evaluate as True .
bool(42), bool(0), bool(-1)

Out[20]: (True, False, True)

In [21]: bool(42 and 0)

Out[21]: False

In [22]: bool(42 or 0)

Out[22]: True

In [23]: # When you have an array of Boolean values in NumPy, this can be thought of
# string of bits where 1 = True and 0 = False , and the result of & and | op
# similar manner as before:

```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
```

Out[23]: array([ True, True, True, False, True, True])

Loading [MathJax]/extensions/Safe.js

```
In [24]: x = np.arange(10)
(x > 4) & (x < 8)
```

```
Out[24]: array([False, False, False, False, False, True, True, True, False, False])
```

## Fancy Indexing

### Exploring Fancy Indexing

การเลือกสมาชิกด้วย array ของตัวชี้ใน array 1 มิติ

```
In [25]: import numpy as np

# สร้าง array 1 มิติ
a = np.random.randint(100, size=10)

# ใช้ fancy indexing โดยส่ง array ของตัวชี้ที่ต้องการดึงข้อมูล
indices = [1, 8, 5, 3] # หรืออาจใช้ ind = [1, 8, 5, 3]
result = a[indices] # หรืออาจใช้ ind

print("Array ต้นฉบับ:", a)
print("Indices ที่เลือก:", indices)
print("ผลลัพธ์จาก Fancy Indexing:", result)
```

```
Array ต้นฉบับ: [67 8 61 27 49 68 6 79 55 59]
```

```
Indices ที่เลือก: [1, 8, 5, 3]
```

```
ผลลัพธ์จาก Fancy Indexing: [ 8 55 68 27]
```

```
In [26]: indices = np.array([[3, 7],
                           [4, 5]])
a[indices]
```

```
Out[26]: array([[27, 79],
                 [49, 68]])
```

Fancy Indexing กับ array แบบหลายมิติ

```
In [27]: import numpy as np

# สร้าง array 2 มิติ (matrix)
X = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12],
              [13, 14, 15, 16]])
```

Loading [MathJax]/extensions/Safe.js  
ต้องการเลือกสมาชิกในตำแหน่งที่เฉพาะเจาะจง  
# เช่น เลือกสมาชิกที่อยู่ในตำแหน่ง (0,1), (1,2), (2,3), (3,0)

```

row_ind = np.array([0, 1, 2, 3])
col_ind = np.array([1, 2, 3, 0])

result = X[row_ind, col_ind]

print("X:")
print(X)
print("Row ind:", row_ind)
print("Column ind:", col_ind)
print("ผลลัพธ์ Fancy Indexing (เลือกสมาชิกตามตำแหน่ง):", result)

```

X:

```

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

```

Row ind: [0 1 2 3]

Column ind: [1 2 3 0]

ผลลัพธ์ Fancy Indexing (เลือกสมาชิกตามตำแหน่ง): [ 2 7 12 13]

In [28]: result2 = X[row\_ind.reshape((4,1)), col\_ind]

```

print("X:")
print(X)
print("Row indices2:")
print(row_ind.reshape((4,1)))
print("Column indices:", col_ind)
print("ผลลัพธ์ Fancy Indexing (เลือกสมาชิกตามตำแหน่ง):")
print(result2)

```

X:

```

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

```

Row indices2:

```

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

```

Column indices: [1 2 3 0]

ผลลัพธ์ Fancy Indexing (เลือกสมาชิกตามตำแหน่ง):

```

[[ 2  3  4  1]
 [ 6  7  8  5]
 [10 11 12  9]
 [14 15 16 13]]

```

## Combined Indexing

In [29]: print(X)

Loading [MathJax]/extensions/Safe.js

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

In [30]: `X[2,[2,0,1]]`

Out[30]: `array([11, 9, 10])`

In [31]: `X[1:, [2, 0, 1]]`

Out[31]: `array([[ 7, 5, 6],
 [11, 9, 10],
 [15, 13, 14]])`

## Modifying Values with Fancy Indexing

In [32]: `x = np.arange(10)
i = np.array([2,1,8,4])
x[i] = 99
print(x)`

`[ 0 99 99 3 99 5 6 7 99 9]`

In [33]: `x[i] -= 10
print(x)`

`[ 0 89 89 3 89 5 6 7 89 9]`

In [34]: `x = np.zeros(10)
x[[0, 2]] = [4, 6]
print(x)`

`[4. 0. 6. 0. 0. 0. 0. 0. 0.]`

In [35]: `x = np.zeros(10)
x[[0, 0]] = [4, 6]
print(x)`

`# Where did the 4 go?
# The result of this operation is to first assign x[0] = 4,
# followed by x[0] = 6 .
# The result, of course, is that x[0] contains the value 6.`

`[6. 0. 0. 0. 0. 0. 0. 0. 0.]`

การใช้ Boolean array สำหรับการเลือกข้อมูล (อีกรูปแบบของ Fancy Indexing)

In [36]: `import numpy as np`

`# สร้าง array 1 มิติ
a = np.random.randint(100, size=10)`

Loading [MathJax]/extensions/Safe.js  
lean array โดยใช้เงื่อนไข เช่น เลือกค่าที่มากกว่า 40

```

bool_index = a > 40
result = a[bool_index]

print("Array ต้นฉบับ:", a)
print("Boolean Index (a > 40):")
print(bool_index)
print("ผลลัพธ์จาก Fancy Indexing โดยใช้ Boolean array:")
print(result)

```

Array ต้นฉบับ: [73 14 79 57 75 28 57 27 37 73]  
 Boolean Index (a > 40):  
 [ True False True True True False True False False True]  
 ผลลัพธ์จาก Fancy Indexing โดยใช้ Boolean array:  
 [73 79 57 75 57 73]

ในตัวอย่างนี้ เราสร้าง Boolean array ที่มีค่า True สำหรับตำแหน่งที่สมาชิกใน a มีค่ามากกว่า 40 และใช้ Boolean array นี้ในการเข้าถึงสมาชิกใน a ผลลัพธ์จะได้ array ที่ประกอบด้วยสมาชิกที่ผ่านเงื่อนไข

## Sorting Arrays

### Fast Sorting in NumPy: np.sort and np.argsort

In [37]: `x = np.array([2,1,4,3,5])  
np.sort(x)`

Out[37]: `array([1, 2, 3, 4, 5])`

In [38]: `x.sort()  
print(x)`

`[1 2 3 4 5]`

In [39]: `#return indices  
x = np.array([2,1,4,3,5])  
i = np.argsort(x)  
print(i)`

`x[i]`

`[1 0 3 2 4]`

Out[39]: `array([1, 2, 3, 4, 5])`

In [40]: `y = np.random.randint(100, size=10)  
y`

Out[40]: `97, 80, 2, 15, 2, 24, 69, 97, 82])`  
 Loading [MathJax]/extensions/Safe.js

```
In [41]: np.sort(y)
```

```
Out[41]: array([ 2,  2, 15, 24, 69, 74, 80, 82, 97, 97])
```

```
In [42]: y.sort()
print(y)
```

```
[ 2  2 15 24 69 74 80 82 97 97]
```

## Sorting along rows or columns

```
In [43]: # A useful feature of NumPy's sorting algorithms is the
#       ability to sort along specific rows or columns of
#       a multidimensional array using the axis argument.
# For example:
```

```
X = np.random.randint(0,10,(4,6))
print(X)
```

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

```
In [44]: # sort each column of X

np.sort(X, axis=0)
```

```
Out[44]: array([[1, 0, 7, 0, 0, 4],
                 [4, 5, 7, 3, 4, 4],
                 [4, 6, 8, 8, 5, 5],
                 [9, 8, 9, 9, 8, 8]])
```

```
In [45]: # sort each row of X

np.sort(X, axis=1)
```

```
Out[45]: array([[4, 4, 5, 7, 8, 8],
                 [0, 3, 5, 7, 8, 9],
                 [0, 0, 1, 4, 6, 9],
                 [4, 4, 5, 8, 8, 9]])
```

## Partial Sorts: Partitioning

```
In [46]: # Note that the first four values in the resulting array
#       are the four smallest in the array, and the remaining
#       array positions contain the remaining values.
#       Within the two partitions, the elements have arbitrary order.
```

Loading [MathJax]/extensions/Safe.js

```
x = np.array([7, 2, 1, 3, 6, 5, 4])
np.partition(x, 4)
```

Out[46]: array([2, 3, 1, 4, 5, 6, 7])

```
In [47]: y = np.random.randint(100, size=10)
y
```

Out[47]: array([51, 87, 60, 14, 96, 59, 99, 41, 77, 22])

```
In [48]: np.partition(y, 4)
```

Out[48]: array([14, 22, 41, 51, 59, 60, 77, 87, 96, 99])

```
In [49]: print(X)
```

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

```
In [50]: # The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

np.partition(X, 2, axis=1)
```

Out[50]: array([[4, 4, 5, 8, 7, 8],
 [0, 3, 5, 9, 8, 7],
 [0, 0, 1, 9, 6, 4],
 [4, 4, 5, 9, 8, 8]])

```
In [51]: np.partition(X, 2, axis=0)
```

Out[51]: array([[1, 0, 7, 0, 0, 4],
 [4, 5, 7, 3, 4, 4],
 [4, 6, 8, 8, 5, 5],
 [9, 8, 9, 9, 8, 8]])

```
In [52]: np.argpartition(X, 2, axis=1)
```

Out[52]: array([[0, 5, 4, 3, 2, 1],
 [1, 3, 5, 0, 4, 2],
 [3, 4, 0, 2, 1, 5],
 [0, 4, 1, 3, 2, 5]])

```
In [53]: np.argpartition(X, 2, axis=0)
```

Out[53]: array([[2, 1, 0, 2, 2, 0],
 [0, 3, 1, 1, 3, 2],
 [3, 2, 3, 0, 0, 1],
 [1, 0, 2, 3, 1, 3]])

In [ ]: