

NumPy

NumPy란?

- NumPy : Numerical Python
- 고성능 과학계산 컴퓨팅과 데이터 분석에 필요한 기본적인 패키지
- 간단히 : 우리가 사용하기 편하게 만들어진 “리스트” / “어레이”라고 생각하면 편함.
- 특징
 - 리스트에 비해서 빠르다
 - 반복문 없이, 전체 데이터에 일괄적인 연산 적용이 가능함 표준 수학 함수 제공
 - 상당히 유연하게 사용 가능
 - 기타

- 데이터 분석에서 중요한 이유는?
 - 벡터 상에서 계산할 수 있고, 이를 변형/가공 등이 용이함.
 - 정렬, 유일한 값, 집합연산 등 일반적인 배열 처리 알고리즘을 쉽게 사용가능
 - 기본적인 수학 및 통계기능을 활용하여 데이터 요약 및 확인

NumPy 사용준비

- 사용법 : `import numpy as np`
 - 참고) `import numpy` vs `import numpy as np`
 - 일반적으로 많은 사람들이 `np`, `pd` 등으로 사용함.
- 확인사항 : `numpy` 패키지가 설치 되어 있는지 확인 필요
(단, `anaconda` 로 설치하였을 경우에 이미 설치되어 있음!)

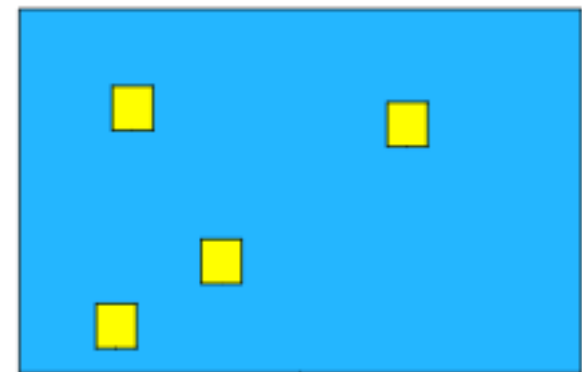
NumPy 구성요소는?

- 아무 것들이나 하나로 묶어 둔 것! 각기 구성 원소에 대한 타입 등에 관련된 부분은 상당히 유연함.
- 원소의 타입이 서로 달라도 상관없음.

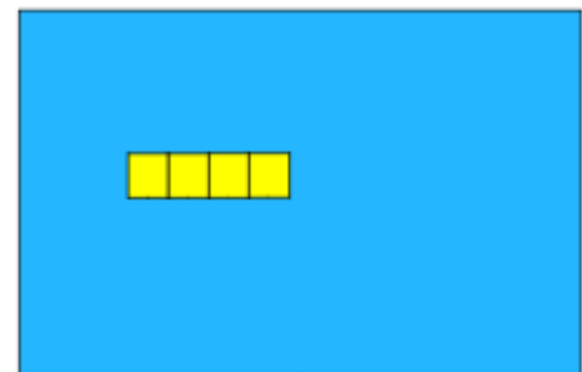
NumPy ndarray

ndarray 생성

- N차원 배열 = ndarray 라고 함.
- 생성방법 : `np.array(원소들...)`
- 참고) `np : import numpy as np`



List Data



NumPy Array Data

- 교재 : 117 참고
- 참고)리스트를 배열로 변환

```
# ndarray 생성  
data1 = [6, 7.5, 8, 0, 1]  
arr1 = np.array(data1)  
arr1  
  
array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

•

- 참고)직접 데이터 입력

```
data2 = [[1,2,3,4],[5,6,7,8,[9,10,11]]]  
arr2 = np.array(data2)  
arr2
```

```
array([[1, 2, 3, 4], [5, 6, 7, 8, [9, 10, 11]]], dtype=object)
```

-


```
] : # ndarray 생성  
data1 = [6, 7.5, 8, 0, 1]  
arr1 = np.array(data1)  
arr1
```

```
] : array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

```
] : data2 = [[1,2,3,4],[5,6,7,8,[9,10,11]]]  
arr2 = np.array(data2)  
arr2
```

```
] : array([[1, 2, 3, 4], [5, 6, 7, 8, [9, 10, 11]]], dtype=object)
```

```
] : data3 = [[1,2,3,4],[5,6,7,8]]  
arr3 = np.array(data3)  
arr3
```

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

```
] : data4 = [[1,2,3,4],[5,6,7,8,9]]  
arr4 = np.array(data4)  
arr4
```

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


- 기본 정보 확인 메소드 : ndim, shape, type 등

```
# 기본 정보 확인  
print arr1.ndim  
print arr1.shape  
print arr1.dtype
```

```
print arr2.ndim  
print arr2.shape  
print arr2.dtype
```

```
print arr3.ndim  
print arr3.shape  
print arr3.dtype
```

```
print arr4.ndim  
print arr4.shape  
print arr4.dtype
```

```
1  
(5,)  
float64  
1  
(2,)  
object  
2  
(2, 4)  
int64  
1  
(2,)  
object
```


- 이렇게 기존의 데이터로 생성하는 것 말고, 여러가지 일 반적이고 많이 사용하는 데이터를 생성하자(참고: 교재 119페이지 참고)
- 참고) array와 asarray로 생성할 때의 차이점! [asarray 는 입력 데이터가 ndarray인 경우에는 참조가 되어서 완벽하게 따로 동작하지 않고, 연동되어서 이름만 2개인 것이 된다]

• 징

```
# 기타 생성
#01)asarray --> asarray 는 기존의 ndarray를 입력으로 하면 완벽한 복사가 안 되어서 변화하는 값이 연동이 됨!
arr_etc1 = np.asarray(data1)
print arr_etc1
arr_etc2 = np.asarray(arr1)
print arr_etc2

arr1[0]=100
print arr1
print arr_etc2

data1[0] = 200
print arr_etc1

[ 6.  7.5  8.  0.  1.]
[100.   7.5   8.   0.   1.]
[100.   7.5   8.   0.   1.]
[100.   7.5   8.   0.   1.]
[ 6.  7.5  8.  0.  1.]
```


- Try) 기타 : 여러가지 많이 사용하는 것들을 쉽게 생성할 수 있도록 정의되어 있음(교재 : p.119참조)

```
: #02) arange, ones, zeros, empty, eye, identity : p.119|  
print np.arange(10)  
print np.ones(10)  
print np.zeros(10)  
print np.empty(10)  
print np.eye(10)  
print np.identity(10)
```


- 중요!!!) 파이썬은 상당히 유연하게 처리를 하고 있음. 그래서 C/C++과 같이 아주 하나하나 엄밀하게 검토하지 않아도 되어 속도가 빠르다. 다만, 이렇게 유연하다보니 정밀하고 속도적인 부분에 있어서는 단점이 있음.
- 하지만, 이렇게 유연하다고 아무거나 다 되는 것이 아님! 그리고 숫자에 중점이 맞추어져 있음.
- Try) 다음 예제들에서 data4를 각기 변경하면서 되는 경우와 안 되는 경우를 살펴보자.

```

: # 그냥 원하는 뭉치들을 다 해서 하면 array로 상당히 유연하게 만들어 준다.
# 단 그렇다고 모든 것을 다 이렇게 해주는 것이 아니라, 1줄로 하게 되면 여러가지 원소들을 한 번에 다 둘 수 있는데,
# 여러 중첩된 것을 사용할 때에는 그것들의 위상이 동일해야 한다.
# 아래의 예제를 통해서 되는 경우와 안 되는 경우를 이해할 것!!!

#data4 = ["aaa",111,-2.333443434343431232323232323232,[1,2,33,"aaa","q"]]
#data4 = ["aaa","111",123,344,-44444.3333333,[1,2,3,4]]
#data4 = ["aaa","bbb","ccc",["a","b","c","d"]]
#data4 = [1,2,3,[4,5,6]]
#data4 = [[1,2,3],[4,5,6],[7,8,9,[10,11,12,13]]]
#data4 = ["aa","bb",1111,-233.2223323232323232323232322323232323,"i am a boy"]
print data4

arr5 = np.array(data4)
arr5

['aa', 'bb', 1111, -233.22233232323234, 'i am a boy']
: array(['aa', 'bb', '1111', '-233.222332323', 'i am a boy'],
      dtype='<S14')

```


NumPy 자료형

- NumPy에서 원소로 들어갈 수 있는 자료형 : 여러가지가 있고, 이것을 모두 외울 필요는 없음(교재 120~121)
- 아주 정교한 프로그램을 작성할 때는 고려해야 하지만, 일반적인 테스트와 파일럿을 하기 위해서는 크게 고려하지 않아도 됨.
- 다만, 숫자 데이터 : float, int, 기타 object 등

- Try) 임의로 numpy가 타입을 지정하게 하지 않고, 사용자가 직접 지정하기 위해서는 dtype=np.(type지정)으로 하면 됨.

```
# ndarray 자료형
arr1 = np.array([1,2,3], dtype=np.float64)
arr2 = np.array([1,2,3], dtype=np.int8)
arr3 = np.array([1,2,3], dtype=np.string_)
print arr1
print arr2
print arr3
```

```
[ 1.  2.  3.]
[1 2 3]
['1' '2' '3']
```


- Try)앞에서는 이미 초기에 데이터를 만들 때, 자료형을 지정하는 것이고, 이미 만들어져 있는 상태에서 변경하기 위해서는?[교재 121]
- Step1) 현재 자료형 확인 [.dtype]
- Step2) 변경할 자료형 확인[표 4-2참고: 교재 120]
- Step3) 변경 수행[.astype]

```
# 기존 타입 확인 / 변경할 타입 지정/ 변경  
print(arr3.dtype)  
arr3_1=arr3.astype(np.float64)  
print(arr3_1.dtype)  
print(arr3_1)
```

```
|S1  
float64  
[ 1.  2.  3.]
```


NumPy 배열/스칼라간 연산

- for 문 없이, 일괄처리가 가능한 기능을 제공하고 있음.
- matlab을 사용한 경험이 있다면, 이와 유사한 기능을 제공하고 있음을 이해할 수 있다.

- Try) 교재 123의 상단 예제

```
# 벡터/스칼라 연산
arr = np.array([[1,2,3],[4,5,6]])
print arr
print("\n")
print arr*arr
print("\n")
print arr-arr
print("\n")
print arr*10
print("\n")
print arr+20
```


NumPy 색인/슬라이싱

- 원하는 위치의 데이터를 추출하기 위해 사용함.
- 리스트와 유사하게 “정수 인덱스” 사용(numpy에서는 뒤에 index를 지정하는 법이 있고, 이를 활용할 수는 있지만 기본적으로 정수 인덱스 사용가능) : 리스트와 유사하게 데이터 접근이 가능할 듯…하지만,,
- 1차원 : 어디, 어디부터 어디까지
- 2차원 : 가로로 어디, 세로로 어디, 가로로 어디부터 어디까지, 세로로 어디부터 어디까지

- Try) 1차원 배열을 만들고, 정수인덱스를 활용해서 리스트와 비교해 보자

```
# 정수 인덱스로 리스트와 1차원 numpy 배열 접근 --> 참고) 원본은 변경되지 않음!! [복사가 남발하게 되면, 속도가 문제가 되어서임]
print list1[0]
print arr1[0]
print("\n")
print list1[1:3]
print arr1[1:3]
print("\n")
print list1[3:]
print arr1[3:]
print("\n")
print list1[:]
print arr1[:]

print("\n")
print list1
print arr1
```


- Try) Start/End/Step

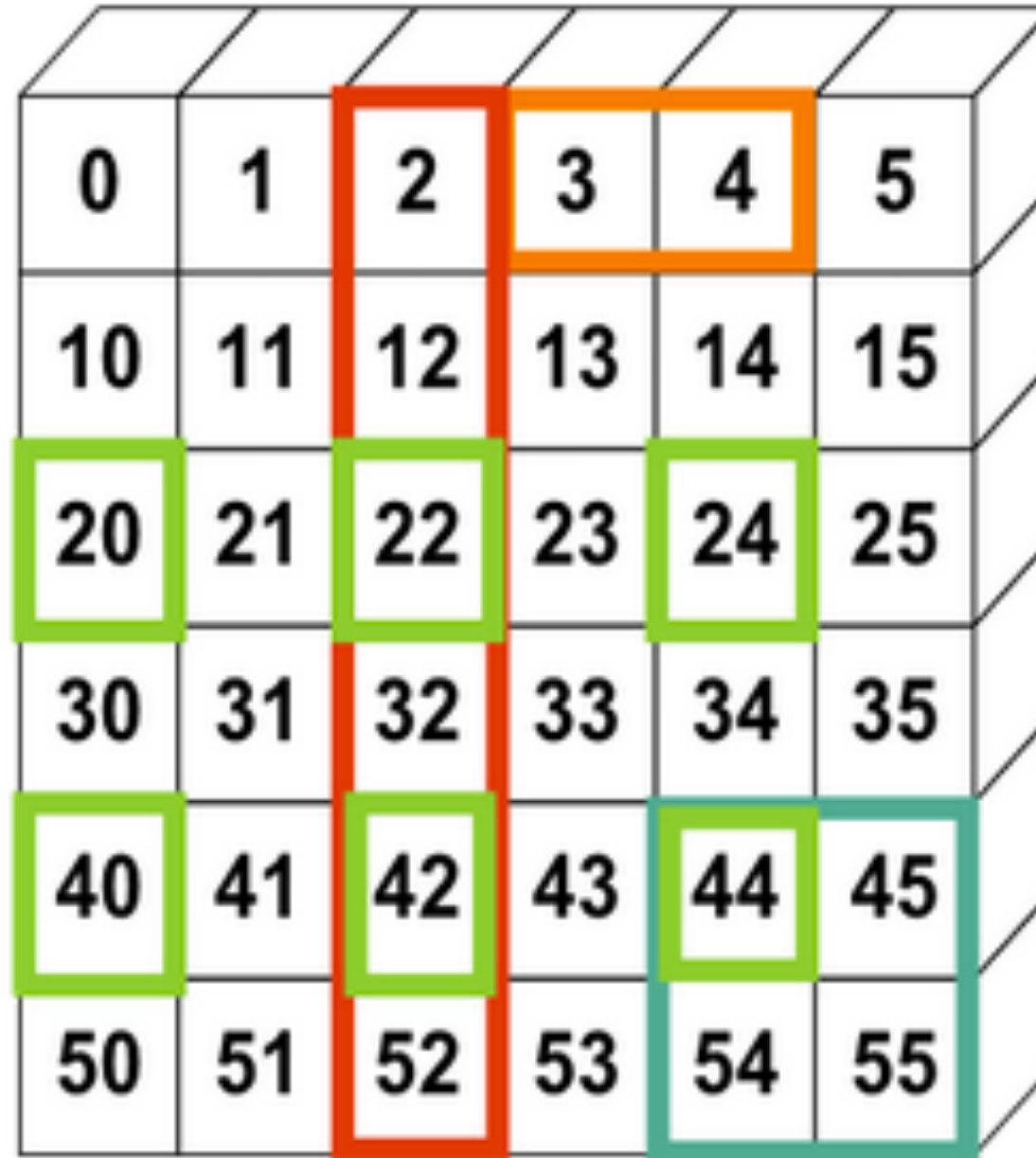
```
# start/end/step  
arr1 = np.arange(10)  
print arr1  
print arr1[2]  
print arr1[9]  
print arr1[2:9:3]  
print arr1[::2]
```

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


- 2차원 배열에 대한 이해 : matrix 라고 이해하면 편함.

		axis 1		
		0	1	2
axis 0	0	0, 0	0, 1	0, 2
	1	1, 0	1, 1	1, 2
	2	2, 0	2, 1	2, 2

- Try) 2차원 배열을 하나 샘플로 만들어보자



A 3D visualization of a 6x6 2D array. The array is represented as a grid of 36 cells, each containing a number from 0 to 55. The cells are arranged in 6 rows and 6 columns. The numbers are as follows:

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Highlighted paths are shown with colored borders:

- Red path:** A vertical line connecting cells (2,0), (2,1), (2,2), (2,3), and (2,4).
- Orange path:** A horizontal line connecting cells (0,2), (0,3), and (0,4).
- Green path:** A zig-zag path connecting cells (2,0), (2,2), (2,4), (4,0), (4,2), and (4,4).
- Teal path:** A horizontal line connecting cells (4,4), (4,5), and (5,5).

- Try) 1차원 배열에 대한 접근방법을 활용하여서, 계층적으로 하나씩 접근해보자

```
# 계층적으로 접근하는 방법  
print a[0]  
print a[1]  
print a[3]  
print("\n")  
print a[0][0]  
print a[3][1]  
print a[3][3]
```

```
[0 1 2 3 4 5]  
[10 11 12 13 14 15]  
[30 31 32 33 34 35]
```

```
0  
31  
33
```


- Try)가로, 세로 각기 접근해보자

```
: # 가로, 세로로 접근  
print a[0][0]  
print a[3][1]  
print a[3][3]  
  
print("\n")  
print a[0,0]  
print a[3,1]  
print a[3,3]
```

```
0  
31  
33
```

```
0  
31  
33
```


가로, 세로 모두 1차원에서 범위 지정도 사용가능

```
print a[1, 0:3]
```

```
print a[2:4, 3]
```

```
print a[:, :1]
```

```
[10 11 12]
```

```
[23 33]
```

```
[[ 0]
```

```
 [10]
```

```
 [20]
```

```
 [30]
```

```
 [40]
```

```
 [50]]
```


- Try) 아래 그림과 같이 해당하는 색깔별로 원하는 정보를 추려보기

A 6x6 grid of numbers from 0 to 55, arranged in 6 rows and 6 columns. The grid is shown in a 3D perspective. The numbers are as follows:

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

The grid features several colored borders highlighting specific data patterns:

- Red border:** A vertical line through column 2 (indices 2, 12, 22, 32, 42, 52) and a horizontal line through row 0 (indices 2, 3, 4).
- Orange border:** A horizontal line through row 0 (indices 3, 4).
- Green border:** A horizontal line through row 2 (indices 20, 21, 22, 23, 24) and a horizontal line through row 4 (indices 40, 41, 42, 43, 44).
- Teal border:** A horizontal line through row 4 (indices 44, 45) and a horizontal line through row 5 (indices 54, 55).

그림에 해당하는 정보 추출

```
print a[0,3:5]
```

```
print a[4:,4:]
```

```
print a[:,2]
```

```
print a[2::2, ::2]
```

```
[3 4]
```

```
[[44 45]
```

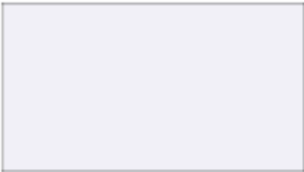

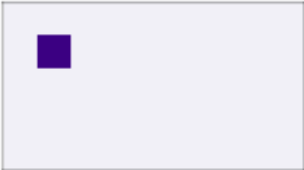
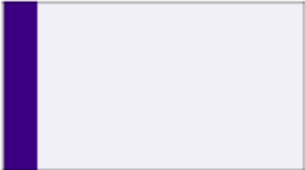







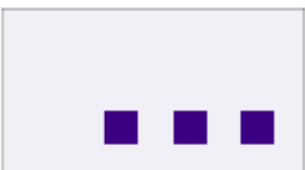
```
 [54 55]]
```

```
[ 2 12 22 32 42 52]
```

```
[[20 22 24]
```

```
 [40 42 44]]
```


- 참고) <http://www.labri.fr/perso/nrougier/teaching/numpy/numpy.html>

Code	Result	Code	Result
<code>z</code>		<code>z[...] = 1</code>	
<code>z[1,1] = 1</code>		<code>z[:,0] = 1</code>	
<code>z[0,:] = 1</code>		<code>z[2:,2:] = 1</code>	
<code>z[:,::2] = 1</code>		<code>z[:,2:] = 1</code>	
<code>z[:-2,:-2] = 1</code>		<code>z[2:4,2:4] = 1</code>	
<code>z[:,2::2] = 1</code>		<code>z[3::2,3::2] = 1</code>	

- Boolean Indexing : 원본 배열과 동일한 길이를 가지고 있으며, 해당하는 조건에 의한 T/F의 값을 가지고 있음. 이를 활용하여 이용하면 원하는 값만 추출할 수 있음.

```
# Boolean indexing  
print arr1  
print arr1 < 5  
print arr1[arr1 < 5]
```

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

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

```
[0 1 2 3 4]
```


- Try) 교재 128
- 상황 : 학생이름 데이터, 학생이름에 대한 국어/영어/수학/과학 성적 이라고 할 때
- 주의사항 : “성”을 누락하여, 동명이인이 그냥 이름으로 동일하게 나타나고 있음.

```
# p.128 : names - 학생이름, data - 국어/영어/수학/과학 성적이라고 생각하면 됨. (순서는 학생이름 순서) [ 이름을 잘 못 정리해서 성이 없이 이름이 같은 사람들이 존재!!!]  
names = np.array(["Bob","Joe","Will","Bob","Will","Joe","Joe"])  
data = np.random.randint(30,100,(7,4))  
print names  
print data
```

```
['Bob' 'Joe' 'Will' 'Bob' 'Will' 'Joe' 'Joe']  
[[43 73 54 89]  
 [48 58 32 98]  
 [85 37 73 48]  
 [39 86 50 80]  
 [45 41 71 70]  
 [73 30 88 95]  
 [53 55 87 97]]
```

•

- Boolean index : `names=="Bob"`
- 주의!) `names="Bob"`이 아님에 유의!!

```
: # 주의!) names="Bob" 아님!!!!  
names == "Bob"  
:  
: array([ True, False, False,  True, False, False, False], dtype=bool)
```

•

- Try) Bob의 이름을 가진 사람들의 성적을 보고 싶음

```
: # Bob의 성적만 보고 싶습니다.  
data[names=="Bob"]  
  
: array([[60, 80, 50, 37],  
        [64, 85, 91, 82]])
```

•

- Try) Bob이 2명이 있는데, 이 중에서 수학/과학의 성적만 보고 싶다.

```
# Bob이 2명이 있는데, 이 중에서 수학/과학의 성적만 보고 싶다.  
data[names=="Bob", 2:]
```

```
array([[50, 37],  
       [91, 82]])
```

•

- Try) Bob이 2명이 있는데, 처음 나온 Bob의 성적만 보고 싶다.

```
#Bob이 2명이 있는데, 처음 나온 Bob의 성적만 보고 싶다.  
data[names=="Bob"][0]
```

```
array([60, 80, 50, 37])
```

•

- Try) 이름이 Bob/ Will인 사람들의 성적을 보고 싶음.

```
# 이름이 bob/ Will인 사람들의 성적을 보고 싶음.  
data[(names=="Bob") | (names=="Will")]
```

```
array([[60, 80, 50, 37],  
       [34, 67, 77, 90],  
       [64, 85, 91, 82],  
       [35, 30, 80, 35]])
```

•

- Try) 성적에서 70점 이하는 과락이니 0으로 값을 변경, 70점 초과는 패스니 1로 데이터 변경을 통해서 누가 통과했는지 알아보자.
 - Step1) 먼저 0/1로 변경
 - Step2) 1인 것들 찾기(np.where : 교재 139]
 - Step3) 사람에 대한 index 추리기[np.unique(): 교재 145]
 - Step4) Boolean Index 를 이용해서 사람이름 찾기

•


```
: # 70점 이하는 과락이니 0으로, 70점 초과는 패스이니 1로  
print data  
data[data<=70] = 0  
data[data>70] = 1  
print data
```

```
[[60 80 50 37]  
 [53 30 59 65]  
 [34 67 77 90]  
 [64 85 91 82]  
 [35 30 80 35]  
 [56 91 51 91]  
 [65 90 49 96]]  
[[0 1 0 0]  
 [0 0 0 0]  
 [0 0 1 1]  
 [0 1 1 1]  
 [0 0 1 0]  
 [0 1 0 1]  
 [0 1 0 1]]
```



```
: # Step2)
# 참고) 원하는 값을 찾을 때는 np.where를 사용[p.139]
# 1로 패스한 사람들의 데이터만 찾을 --> 결과가 어떠한 것을 의미하는지 잘 확인할 것!!!
np.where(data>0)

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



```
# Step3/4)
```

```
# 이를 바탕으로 이름에 대한 유니크한 index를 찾아서
```

```
# 누가 패스했는지 찾아보자
```

```
print np.unique(np.where(data>0)[0])
```

```
print names[np.unique(np.where(data>0)[0])]
```

```
[0 2 3 4 5 6]
```

```
['Bob' 'Will' 'Bob' 'Will' 'Joe' 'Joe']
```


- NumPy Fancy Indexing : 정수 배열을 사용한 색인을 설명하기 위해서 차용한 용어. [p.131]
- 원하는 row를 보기 위해서는 보려고하는 row를 여러개 인 경우에는 “리스트”로 넘기면 된다. 여기서 “음수”를 사용하게 되면, 앞에서부터가 아니라 “뒤에서 부터” 위치를 찾게 됨.

-

- Try) 교재 p.131~132 예제

```
# Fancy Indexing : 교재 p.131~132 예제
```

```
arr = np.empty((8,4))
```

```
print arr
```

```
for i in range(8):
```

```
    arr[i] = i
```

```
print arr
```

```
[[ 2.31584178e+077  2.31584178e+077  9.88131292e-323  1.27319747e-313]
 [ 1.27319747e-313  1.27319747e-313  2.96439388e-323  1.90979621e-313]
 [ 0.00000000e+000  5.92878775e-323  3.18299369e-313  0.00000000e+000]
 [ 4.03179200e-313  9.88131292e-323  4.88059032e-313  1.42142928e+160]
 [ 2.31584178e+077  6.15378779e-313  9.51498384e-053  7.00258611e-313]
 [ 2.25405648e+174  5.58048799e-091  8.31823323e-313  1.26121664e-076]
 [ 8.01597480e+165  9.57733613e-313  9.16651763e-072  1.03977794e-312]
 [ 1.20142249e-071  1.12465777e-312  2.31584178e+077  1.20953760e-312]]
```

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

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

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

```
 [ 3.  3.  3.  3.]
```

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

```
 [ 5.  5.  5.  5.]
```

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

```
 [ 7.  7.  7.  7.]]
```

```
arr[[4,3,0]]
```

```
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.]])
```

```
arr[[-3,-5,-7]]
```

```
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```


- Try) reshape : 전체 사이즈가 다르게 되면 에러;;;;

```
# NumPy Transform  
arr = np.arange(32).reshape((8,4))  
arr
```

```
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]])
```

```
arr = arr.reshape((4,8))  
arr
```

```
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]])
```

```
arr = arr.reshape((2,16))  
arr
```

```
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]])
```

```
arr = arr.reshape((3,9))  
arr
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-126-e3f55d389419> in <module>()
```


- Try) .T : 전치행렬

```
: # Transpose
arr = arr.reshape((4,8))
print arr
print arr.T

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

```
np.dot(arr, arr.T)

array([[ 140, 364, 588, 812],
       [ 364, 1100, 1836, 2572],
       [ 588, 1836, 3084, 4332],
       [ 812, 2572, 4332, 6092]])
```


- Try) transpose : 순서는 전체적인 순서에서 변경임

```
# Transpose  
arr = np.arange(16).reshape((2,2,4))  
print arr
```

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

```
arr[1]
```

```
array([[ 8,  9, 10, 11],  
       [12, 13, 14, 15]])
```

```
arr[0]
```

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

```
# 제일 작은 단위에서 이동임;;;|  
arr.transpose((1,0,2))
```

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


- Try) 그냥 참고로 알아두면 됨.

```
# swapaxes  
print arr.reshape(4,4)  
print("\n")  
print arr.swapaxes(2,2)  
print("\n")  
print arr.swapaxes(1,2)  
print("\n")  
print arr.swapaxes(0,2)
```

•

NumPy ufunc(유니버설 함수)/ math/stat

- ufunc : ndarray에 있는 원소별로 연산을 수행하는 함수
- 입력 : 하나 이상의 스칼라 값
- 출력 : 하나 이상의 스칼라 결과로 반환
- 간단한 함수들을 고속으로 처리할 수 있도록 래핑해서 만든 함수!
- 즉, 간단한 수학적인 함수들이 이미 정의되어 있어서, 쉽게 사용할 수 있음(p.136, p.137 : unfc, p.142:math/stat)

- Try)

```
# ufunc / math
arr1 = np.arange(10)
print arr1
print np.sqrt(arr1)

arr2 = np.arange(30,40,1)
print arr2

print np.maximum(arr1,arr2)

print arr1.max()
print arr2.max()

print arr1.mean()
print arr2.mean()
```

[0 1 2 3 4 5 6 7 8 9]
[0. 1. 1.41421356 1.73205081 2. 2.23606798
 2.44948974 2.64575131 2.82842712 3.]
[30 31 32 33 34 35 36 37 38 39]
[30 31 32 33 34 35 36 37 38 39]
9
39
4.5
34.5

NumPy where

- `[xv if c else yv for (c,xv,yv) in zip(condition,x,y)]`

- Try) p.139
- cond=True : xarr에서 선택, cond=False : yarr에서 선택

```
# where  
#00)  
xarr = np.array([1.1,1.2,1.3,1.4,1.5])  
yarr = np.array([2.1,2.2,2.3,2.4,2.5])  
cond = np.array([True, False, True, True, False])  
#참고)  
zip(xarr, yarr, cond)
```

```
[(1.1000000000000001, 2.1000000000000001, True),  
 (1.2, 2.2000000000000002, False),  
 (1.3, 2.2999999999999998, True),  
 (1.3999999999999999, 2.3999999999999999, True),  
 (1.5, 2.5, False)]
```

```
#01)  
# cond=True : xarr 선택, cond=False : yarr선택  
print [(x if c else y) for x,y,c in zip(xarr, yarr, cond)]
```

```
[1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```



```
#02)np.where
```

```
np.where(cond, xarr, yarr)
```

```
array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```


- Try) np.where을 이용해서 데이터 변형

```
# 03) np.where  
arr = np.random.randn(4,4)  
print arr  
print np.where(arr>0,2,-2)  
print np.where(arr>0, 2, arr)
```

```
[[ 0.80708975 -1.53230537 -1.64563735  0.9790641 ]  
 [ 2.2038493  -0.55115384  0.37337344 -1.04913782]  
 [ 0.12417176 -1.68770873  0.42837772 -0.56147658]  
 [ 1.13397201  1.2100076  -0.13379619 -0.33057365]]  
[[ 2 -2 -2  2]  
 [ 2 -2  2 -2]  
 [ 2 -2  2 -2]  
 [ 2  2 -2 -2]]  
[[ 2.         -1.53230537 -1.64563735  2.         ]  
 [ 2.         -0.55115384  2.         -1.04913782]  
 [ 2.         -1.68770873  2.         -0.56147658]  
 [ 2.          2.         -0.13379619 -0.33057365]]
```


- Try) np.where에서 2가지 조건을 사용하기 위해서는 어떻게??? 1가지 조건만 사용할 수는 없는데;;;
- 예) p.140예제에서 $0 < \text{arr} < 0.1$ 인 값을 2로, 나머지를 -2로 변경하기 위해서는 어떻게?

```
# np.where with multiple condition  
np.where(arr>0 & arr<0.1, 2,-2)
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-172-ce2c2f2e38d0> in <module>()  
      1 # np.where with multiple condition  
----> 2 np.where(arr>0 & arr<0.1, 2,-2)
```

```
TypeError: ufunc 'bitwise_and' not supported for the input types, and the inputs could not be safely coerced to any supported types according to the casting rule "safe"
```



```
# sol1)
```

```
np.where((arr>0) & (arr<0.1), 2,-2)
```

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

```
# sol2)
```

```
np.where(np.logical_and(arr>0, arr<0.1), 2, -2)
```

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


NumPy 정렬

- sort 메소드가 제공하여, 따로 정렬에 대한 알고리즘을 구현할 필요가 없음.

```
# sort
# arr.sort()는 arr에 바로 적용을 하는 것이라서 직접 보이지 않음, 보려면 arr을 사용
arr = np.random.randn(8)
print arr
print arr.sort()
print arr
```

```
[-0.04673017 -0.1846534 -2.34594872  0.85986575 -0.1264739  0.28213828
 -0.44132056 -0.78428275]
```

```
None
```

```
[-2.34594872 -0.78428275 -0.44132056 -0.1846534 -0.1264739 -0.04673017
 0.28213828  0.85986575]
```


NumPy 집합함수

- 기본적인 집합연산을 제공[p.145 표 4-6]
- 중복제거 : unique
- 공통 : intersect1d
- 합집합 : union1d
- etc

```
# Set
print names
print np.unique(names)

['Bob' 'Joe' 'Will' 'Bob' 'Will' 'Joe' 'Joe']
['Bob' 'Joe' 'Will']
```

•

NumPy 응용

- ref) <http://www.scipy-lectures.org/index.html>