# 1. Introduction to Numpy

## 1.1 Importing Numpy package

Numpy는 Numerical Python의 약자로 대수학 계산에 사용되는 Python 패키지이다.

```
import numpy as np
```

## 1.2 Array

### Create Array

Numpy의 핵심 데이터 구조는 ndarray로, 파이썬의 list와 비슷한 구조를 가지고 있다. array는 np.array()함수를 통해 생성할 수 있다.

```
import numpy as np
x = np.array([10, 20, 30])
length = len(x)
```

### Indexing Array

파이썬의 리스트처럼 Array는 인덱싱과 할당이 가능하다.

```
x = np.array([10, 20, 30])
x0 = x[0]
x2 = x[2]
x[1] = 42
```

### Slicing Array

```
x = np.array([9, 1, 5, 6, 2, 0, 4, 3, 8, 7])
first_half = x[:5]
last_8 = x[2:]
middle = x[1:-1]
```

### Slicing Array with step

Array를 Slicing 할때 추가적으로 Step을 지정하면 step만큼 건너뛴 인덱스를 Slicing한다.

```
x = np.array([9, 1, 5, 6, 2, 0, 4, 3, 8, 7])

even = x[0:len(x):2]
odd = x[1:len(x):2]
mul_3 = x[0:len(x):3]
every_2 = x[3:9:2]
```

## 1.3 Result of slicing

Array Slicing 한 결과는 새로운 Array가 아니라 기존에 존재한 Array의 View로써 작동하게 된다. 즉, Sliced array가 변경되면 기존의 Array의 값도 변경되게 된다. 새로운 Array를 생성하려면 copy() 메소드를 사용해서 복사해야 한다.

```
x = np.array([1, 0, 0, 0, 1])

y = x[1:-1]
z = y.copy()

z[0] = 9
print(x, y, z)

y[1] = 7
print(x, y, z)
```

## 1.4 Reverse Slicing

Python의 reverse index와 같이 array에도 뒤의 원소부터 -1, -2, ... 의 인덱스를 사용할 수 있다. 마찬가지로 Step Slicing에서도 음수를 사용하면 줄어든 인덱스의 원소를 Slicing 할 수 있다.

```
x = np.array([9, 1, 5, 6, 2, 0, 4, 3, 8, 7])
second_to_last = x[-2]
reversed_x = x[::-1]
first_5_reversed = x[4::-1]
last_5_reversed = x[-1:4:-1]
```

## 1.5 n-dimensional array

ndarray는 n-dimensional array로 다차원의 배열을 사용할 수 있다. 이전에 사용했던 배열은 1-dimensional array이고, 2-dimensional array는 list of list 형태로 구성되어 있다. 2darray를 인덱싱할 때 array2d[row_index, col_index]를 사용하면 된다. 2-dimensional array의 경우에도 1-dimensional array와 동일한 인덱싱 방법을 적용 가능하다. array2d[row_start:row_end:row_step, col_start:col_end:col_step]. 또한 원하는 행과 열의 인덱스의 값을 리스트에 저장해서 직접 지정할 수도 있다.

```
my_2d_array = np.array([[1, 4, 7],
                        [2, 5, 8],
                        [3, 6, 9]])
my_2d_array[1, 1] = 42
print(my_2d_array)

array2d = np.array([
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15]
])

x = array2d[:, 3:]
y = array2d[::2,:]
z = array2d[::2,::2]

# using list index
people_data = np.array([
    [27, 67, 1.65],
    [35, 81, 1.84],
    [29, 55, 1.60],
    [41, 73, 1.79]
])

anna_row = people_data[2,:]
bob_age_height = people_data[3,[0,2]]
ages_col = people_data[:,0]
weight_dexter_bob = people_data[[1,3],1]
```

# 2. Arithmetic with Numpy Arrays

## 2.1 Adding

서로 동일한 길이를 가진 ndarray는 각각의 인덱스의 원소끼리 더해 새로운 ndarray를 생성한다. Python의 경우 각각의 리스트에 대한 원소를 추출하여 더해야 하기 때문에 numpy 패키지가 대수학 연산면에서 훨씬 효율적임을 확인할 수 있다. numpy 객체가 Python list보다 훨씬 빠른 이유는 numpy 패키지가 low-level programming language인 C 언어로 작성되었기 때문이다. low-level programming language는 컴퓨터 전처리 작업에 낮은 추상화를 가진 대신 빠른 수행속도를 갖게 된다.

```
def add_list_values(list1, list2) :
    res = []
    for i in range(len(list1)) :
        res.append(list1[i] + list2[i])
    return res

list1 = [3, 4, 3, 2]
list2 = [1, 2, 4, 3]
list3 = add_list_values(list1, list2)
print(list3)

# Check time between list and numpy

import time
import random
random.seed(0)

# Generate test lists
list1 = [random.randint(0, 1000) for _ in range(100000)]
list2 = [random.randint(0, 1000) for _ in range(100000)]

# Measure the execution time of adding lists
start = time.time()
add_list_values(list1, list2)
end = time.time()
time_list = end - start

# Write your code below
import numpy as np
x1 = np.array(list1)
x2 = np.array(list2)
start = time.time()
x3 = x1+x2
end = time.time()
time_array = end - start
ratio = time_list/time_array
print(ratio)
```

## 2.2 Extensions of arithmetic to n-dimensional array

1-dimensional array에서 가능한 연산방식은 n-dimensional array로 확장될 수 있다. 이 연산들은 반드시 각각의 차원이 서로 일치해야하기 때문에 차원을 확인하는 작업은 매우 중요하다. ndarray의 row, column을 확인하는 attribute는 shape이다.

```
scores = np.array([
    [46, 74, 52, 81],
    [75, 45, 67, 53],
    [67, 80, 73, 63],
    [59, 94, 43, 78]
])

scores_day1 = scores[:, 0:2]
scores_day2 = scores[:, 2:4]
shape1 = scores_day1.shape
shape2 = scores_day2.shape
print(shape1, shape2)

total_scores = scores_day1 + scores_day2
print(total_scores)
```

## 2.3 Arithmetic methods

### max(), min()

numpy packages에서 max() 메소드를 사용하는 방법은 다음과 같다. 또한 max() 메소드 내부에 axis = 0/1 (0 : 행방향, 1 : 열방향) 매개변수를 입력함으로써 각 행을 추출하지 않고 벡터를 생성할 수 있다.

```
total_scores = np.array([
 [ 98, 155],
 [142,  98],
 [140, 143],
 [102, 172]
])

scores_game1 = total_scores[:, 0]
scores_game2 = total_scores[:, 1]
min_game1 = np.min(scores_game1)
max_game1 = scores_game1.max()
min_game2 = scores_game2.min()
max_game2 = np.max(scores_game2)

# set axis = 0/1
total_scores = np.array([
 [ 98, 155],
 [142,  98],
 [140, 143],
 [102, 172]
])

max_game_scores = total_scores.max(axis = 0)
min_game_scores = total_scores.min(axis = 0)
max_people_scores = total_scores.max(axis = 1)
min_people_scores = total_scores.min(axis = 1)
```

### sum()

sum() 메소드는 ndarray 내부의 원소를 더하는 연산을 수행한다. 마찬가지로 axis = 0/1을 통해 각 원소를 어느 방향으로 더할 것인지를 지정할 수 있다.

```
total_scores = np.array([
 [ 98, 155],
 [142,  98],
 [140, 143],
 [102, 172]
])

total_people_score = total_scores.sum(axis = 1)
max_score = total_people_score.max()
```

# 3. Broadcasting Numpy Arrays

## 3.1 Array with same value

Array의 각 원소를 1만큼 변화를 주고싶을때 차원이 큰 Array일 경우 직접 생성하는 것은 한계가 있다. np.ones()는 동일한 shape를 가지면서 각 원소가 1인 Array를 생성한다.

```
import numpy as np
x = np.array([
 [7., 9., 2., 2.],
 [3., 2., 6., 4.],
 [5., 6., 5., 7.]
])

ones = np.ones(x.shape)
x = x - ones
```

## 3.2 Broadcasting

1이 아닌 값의 변화를 주고 싶을때 numpy 패키지는 ndarray와 상수간에 연산을 보장한다. **Broadcasting**은 ndarray와 상수간의 연산에서 ndarray의 shape과 동일한 상수 ndarray가 존재하고 있다고 간주하고 연산을 진행하는 것이다. 한쪽의 차수가 적은 값을 큰 차수로 확장시켜 연산을 해 연산이 진행되는 방식을 유의해야 한다.

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

z = x + y
print(z)
```

## 3.3 Change shape of array

Broadcasting과 다른 행렬 연산을 위해 array의 shape를 변경할 필요가 있다. reshape() 메소드는 내부에 입력된 tuple을 기준으로 array를 재배열한다.

```
dice1 = np.array([1, 2, 3, 4, 5, 6])
dice2 = dice1.reshape((6,1))
dice_sums = dice1 + dice2
print(dice_sums)
```

## 3.4 Order of reshape

reshape 메소드는 ndarray의 원소를 행을 단위로 원소를 추출하여 새롭게 저장한다. 열을 기준으로 원소를 추출하려면 order = 'F' 로 parameter를 설정해야 한다.

```
cell_numbers = np.array(range(1,37))
numbering_by_row = cell_numbers.reshape((6,6))
numbering_by_col = cell_numbers.reshape((6,6), order = 'F')
```

# 4. Datasets and Boolean Indexing

## 4.1 Load csv into ndarray

numpy.genfromtxt() 메소드는 텍스트파일 내부의 숫자 데이터를 ndarray에 저장한다.

```
import numpy as np
sars = np.genfromtxt('sars.csv', delimiter = ',')
first_five = sars[0:5, :]
print(first_five)

[[       nan       nan       nan       nan       nan       nan]
 [       nan 4.000e+00 2.000e+00 6.000e+00 0.000e+00 0.000e+00]
 [       nan 1.510e+02 1.000e+02 2.510e+02 4.300e+01 1.700e+01]
 [       nan 2.674e+03 2.607e+03 5.327e+03 3.490e+02 7.000e+00]
 [       nan 9.770e+02 7.780e+02 1.755e+03 2.990e+02 1.700e+01]]
```

## 4.2 Processing ndarray table

ndarray에 저장된 결과는 다음과 같다. nan은 not a number의 약자로 CSV 파일 내부의 문자 데이터를 의미한다. 또한 각각의 숫자 데이터는 scientific notation으로 표기가 되어있기 때문에 np.set_printoptions(suppress=True) 메소드를 통해 일반적인 숫자로 표기할 필요가 있다.

```
np.set_printoptions(suppress = True)
sars = sars[1:, :]
sars = sars[:, 1:]
print(sars[:5, :])

# result of sars
[[   4.    2.    6.    0.    0.]
 [ 151.  100.  251.   43.   17.]
 [2674. 2607. 5327.  349.    7.]
 [ 977.  778. 1755.  299.   17.]
 [   0.    1.    1.    0.    0.]]
```

## 4.3 Load csv file with names

Numpy는 숫자 데이터만 다루기 때문에 열의 이름을 입력받지 못하지만, np.genfromtxt() 메소드의 names parameter는 첫번째 행의 이름을 열의 이름으로 인식할 수 있게 해준다.

```
sars_with_names = np.genfromtxt('sars.csv', delimiter=',', names=True)
print(sars_with_names['Total'])
```

## 4.4 ndarray with boolean

Numpy는 비교연산자를 통해 생성된 boolean data를 ndarray에 저장할 수 있다. ndarray는 숫자 데이터만을 저장하기 때문에 True는 1로, False는 0으로 저장되어 연산도 가능하다. 비교 연산자도 broadcasting을 적용해 boolean 데이터를 생성할 수 있다.

```
female = sars[:, 0]
male = sars[:, 1]
more_female_cases = female > male
equal_cases = female == male
num_more_female = more_female_cases.sum()
num_equal = equal_cases.sum()
num_more_male = len(sars) - num_more_female - num_equal
```

### sum()

np.sum() 메소드를 통해 boolean 데이터의 총 개수를 확인할 수 있다.

### any()

np.any() 메소드를 통해 boolean 데이터의 참이 하나라도 있는지 확인할 수 있다.

### all()

np.all() 메소드를 통해 boolean 데이터가 모두 참인지 확인할 수 있다.

## 4.5 Relational Operator of booleans

&, |, ~ 연산자는 비교 연산자를 통해 생성된 서로 다른 boolean array에 대해서 관계 연산을 수행한다.

```
deaths = sars[:,3]
fatality_ratio = sars[:,-1]

death_gt_100_ratio_lt_10 = (deaths >= 100) & (fatality_ratio <= 10)
count = death_gt_100_ratio_lt_10.sum()
```

## 4.6 Boolean indexing

Numpy의 유용한 점은 생성된 boolean array를 사용하여 ndarray를 필터링 할 수 있다. 이때 boolean array는 ndarray의 shape과 동일해야 한다.

```
mask_zeros = sars == 0
zeros = sars[mask_zeros]
num_zeros = len(zeros)
```

# 5. Numpy Datatypes

## 5.1 Check datatypes in ndarray

numpy ndarray는 동일한 데이터 타입만 저장한다. array의 데이터 타입을 확인하고자 할때는 ndarray.dtype 속성을 사용한다. array를 생성할때 데이터 타입을 지정하려면 np.array() 메소드에 dtype parameter에 원하는 데이터 타입을 작성하면 된다. 데이터 타입을 다른 데이터 타입으로 변경하려면 ndarray.astype() 메소드를 사용한다.

```
values = [1, 2, 3, 4]
x = np.array(values, dtype = np.float64)
x[0] = 5.5
print(x)
```

ndarray가 하나의 데이터 타입만 저장하지 못하는 이유는 numpy package가 C 언어를 기반으로 작성되었기 때문이다. C언어는 Python과 다르게 typed language이기 때문이다.

## 5.2 Datatype in ndarray

numpy는 ndarray의 데이터 타입을 모든 데이터를 포함할 수 있는 타입으로 정의한다. 예를들어 [1, 2, True, False] array의 경우 bool type은 2를 포함할 수 없기 때문에 해당 array의 데이터 타입은 integer가 된다. 이런 경우 때때로 array가 저장하고 있는 정보를 잃어버릴 가능성이 크다.

```
import numpy as np
values = [3.14, 6.42, 5.0, 0.5]
x = np.array(values, dtype = np.int64)
print(x)
```

## 5.3 Fixed-length bit representations

Python의 숫자 표현 방식은 fixed-length bit representation을 통해 음수와 양수를 표현할 수 있다. 따라서 어느정도 제한이 있는 수는 그에 해당하는 고정 길이 비트를 사용하면 공간상의 여유를 마련할 수 있다. array가 소모하고 있는 데이터의 크기를 평가하려면 ndarray.nbytes 속성을 사용하면된다. 하지만 애매한 고정 길이 방식 표현은 underflow와 overflow를 발생해 데이터를 손상시킬 가능성이 있다.

```
x = np.array([-127, -57, -6, 0, 9, 42, 125], dtype=np.int8)
print(x-2)
print(x+3)
```

## 5.4 Disk storage using datatype

numpy 64 bits의 데이터 타입을 사용할 경우, 80억개의 데이터의 크기는 아래와 같다.

$$\frac{(8 \times 10^9) \times 64}{8 \times 2^{30}} = 59GB$$