

Numpy

2019.04



1. Numpy 란?

■ 파이썬 과학 처리 패키지

- Numerical Python
- 파이썬의 고성능 과학 계산용 패키지
- Matrix와 Vector와 같은 Array 연산의 사실상의 표준

■ 특징

- 일반 List에 비해 빠르고, 메모리를 효율적으로 사용
- 반복문 없이 데이터 배열에 대한 처리를 지원함
- 선형대수와 관련된 다양한 기능을 제공함
- C, C++ 등의 언어와 통합 가능

■ References

- <https://docs.scipy.org/doc/numpy/user/quickstart.html>
- 데이터 사이언스 스쿨 (데이터 과학을 위한 파이썬 기초)
<https://datascienceschool.net/view-notebook/39569f0132044097a15943bd8f440ca5>
- Numpy 강좌 <https://www.youtube.com/playlist?list=PLBHVuYIKEkULZLnKLzRq1CnNBOBIBTkqp>

2. ndarray(Numpy Dimensional Array)

■ import

```
import numpy as np      # 표준화되어 있음
```

■ Array 생성

```
test_array = np.array([1, 4, 5, 8], float)
print(test_array)
type(test_array[3])
print(test_array.dtype)  # Array 전체의 데이터 타입을 반환함
print(test_array.shape)  # Array의 shape(차원 구성)을 반환함
```

- numpy는 np.array 함수를 활용하여 배열을 생성함 → ndarray
- numpy는 하나의 데이터 타입만 배열에 넣을 수 있음
- List와 가장 큰 차이점, Dynamic typing(예, [1, 2, "5", 4.2]) not supported
- C의 Array를 사용하여 배열을 생성함

3. Array shape

■ Vector (1차원)

```
test_array = np.array([1, 4, 5, 8], float)
```

➔ shape은 (4,) : 1차원에 4개의 element가 있는 벡터

■ Matrix (2차원)

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

```
np.array(matrix, int).shape
```

➔ shape은 (3, 4) : 행이 3개, 열이 4개인 매트릭스

■ Tensor (3차원)

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

```
np.array(tensor, int).shape
```

➔ shape은 (4, 3, 4) : 평면이 4개, 행이 3개, 열이 4개인 텐서

3. Array shape

■ ndim & size

```
np.array(tensor, int).ndim    # 3, number of dimension
np.array(tensor, int).size    # 48
```

■ dtype

```
np.array([[1, 2, 3], [4.5, "5", "6"]], dtype=np.float32)
array([[1. , 2. , 3. ],
       [4.5, 5. , 6. ]], dtype=float32)
```

- Single element가 가지는 데이터 타입
- C의 데이터 타입과 호환
- nbytes – ndarray object의 메모리 크기를 바이트 단위로 반환함

■ reshape

- Array의 shape을 변경함 (element의 개수는 동일)

3. Array shape

■ reshape

```
test_matrix = [[1,2,3,4], [5,6,7,8]]  
np.array(test_matrix).shape → (2, 4)  
np.array(test_matrix).reshape(8, ) → array([1,2,3,4,5,6,7,8])  
np.array(test_matrix).reshape(8, ).shape → (8, )
```

- Array의 shape을 변경함 (element의 개수는 동일)
- Array의 size만 같다면 다차원으로 자유로이 변형가능

```
np.array(test_matrix).reshape(2, 4).shape → (2, 4)  
np.array(test_matrix).reshape(-1, 2).shape → (4, 2)  
-1: size를 기반으로 row 개수 선정  
np.array(test_matrix).reshape(2, 2, 2).shape → (2, 2, 2)
```

■ flatten

```
test_matrix = [[[1,2,3,4], [5,6,7,8]], [[2,3,4,5], [6,7,8,9]]]  
np.array(test_matrix).flatten()  
→ array([1,2,3,4,5,6,7,8,2,3,4,5,6,7,8,9])
```

- 다차원 array를 1차원 array로 변환

4. Indexing & slicing

■ Indexing

```
a = np.array([[1,2,3], [4,5,6]], int)
print(a)
print(a[0,0])    # 2차원 배열 표기법 1
print(a[0][0])   # 2차원 배열 표기법 2
a[0, 0] = 1
```

- List와 달리 이차원 배열에서 [0, 0]과 같은 표기법을 제공함
- Matrix일 경우 앞은 행(row) 뒤는 열(column)을 의미함

■ Slicing

```
a = np.array([[1,2,3,4,5], [6,7,8,9,10]], int)
a[:, 2:]      # 전체 row의 2열 이상
a[1, 1:3]     # row 1의 1~2열
a[1:3]        # 1 row ~ 2 row 전체, column은 무시
a[:, ::2]     # step 가능
```

- List와 달리 행과 열 부분을 나눠서 slicing이 가능함
- Matrix의 부분 집합을 추출할 때 유용함

5. Creation function

■ arange

```
np.arange(10)    # arange – List의 range와 같은 효과  
→ array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])  
np.arange(0, 5, 0.5)    # floating point도 표시가능  
→ array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])  
np.arange(0, 5, 0.5).tolist()    # List로 만들 수 있음  
np.arange(30).reshape(5, 6)    # size가 같으면 가능
```

- List와 달리 행과 열 부분을 나눠서 slicing이 가능함
- Matrix의 부분 집합을 추출할 때 유용함

■ ones, zeros and empty

```
np.zeros(shape=(10,), dtype=np.int8)    # 원소가 10개인 벡터 생성  
→ array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int8)  
np.ones((2, 5))    # 2 x 5 – 값이 1인 matrix 생성  
→ array([[1., 1., 1., 1., 1.],  
          [1., 1., 1., 1., 1.]])  
np.empty((3, 5))    # 메모리가 초기화되어 있지 않음
```

- empty – shape만 주어지고 비어있는 ndarray 생성

5. Creation function

■ Something like

```
test_matrix = np.arange(30).reshape(5,6)
np.ones_like(test_matrix)
np.zeros_like(test_matrix)
```

- 기존 ndarray의 shape 크기 만큼 1, 0 또는 empty array를 반환

■ identity (단위 행렬 생성)

```
np.identity(n=3, dtype=np.int8)
→ array([1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]), dtype=int8)
```

n → number of rows

```
np.identity(5)
→ 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.]])
```

5. Creation function

■ eye (대각선이 1인 행렬)

```
np.eye(N=3, M=5, dtype=np.int8)
→ array([[1, 0, 0, 0, 0],
         [0, 1, 0, 0, 0],
         [0, 0, 1, 0, 0]], dtype=int8)

np.eye(5)
→ 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.]])

np.eye(3, 5, k=2)      # k → start index
→ array([[0., 0., 1., 0., 0.],
         [0., 0., 0., 1., 0.],
         [0., 0., 0., 0., 1.]])
```

■ diag (대각 행렬의 값을 추출)

```
matrix = np.arange(9).reshape(3,3)
np.diag(matrix)
np.diag(matrix, k=1)    # k → start index
```

5. Creation function

■ Random sampling(데이터 분포에 따른 sampling으로 array를 생성)

```
np.random.seed(seed=1000)          # 시드로 난수 생성 초기값 지정
```

```
np.random.uniform(0, 1, 10).reshape(2,5)      # 균등 분포
        최소 최대 개수
```

```
np.random.normal(0, 1, 10).reshape(2,5)      # 정규 분포
        평균 표준편차 개수
```

```
np.random.binomial(n, p, size)               # 이항 분포
np.random.poisson(lam, size)                  # 포아송 분포
np.random.standard_t(df, size)                # t-분포
np.random.f(dfnum, dfden, size)              # F-분포
```

```
import matplotlib.pyplot as plt
rand_norm = np.random.normal(0., 3., size=1000) # 평균, 표준편차
count, bins, ignored = plt.hist(rand_norm, normed=False)
```

```
rand_pois = np.random.poisson(lam=20, size=1000)
unique, counts = np.unique(rand_pois, return_counts=True)
np.asarray((unique, counts)).T
plt.bar(unique, counts, width=0.5, color="red", align='center')
```

6. Operation function

■ Sum

```
test_array = np.arange(1,11)
test_array.sum(dtype=np.float) → 55.0
```

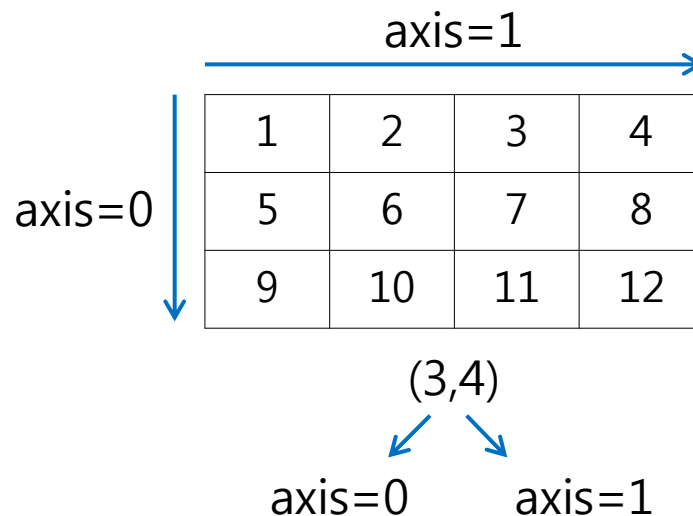
■ Axis

- 모든 operation function을 실행할 때, 기준이 되는 dimension 축

```
test_array = np.arange(1,13).reshape(3,4)
→ array([[1, 2, 3, 4],
        [5, 6, 7, 8],
        [9,10,11,12]])
```

```
test_array.sum(axis=1)
→ array([10, 26, 42])
```

```
test_array.sum(axis=0)
→ array([15, 18, 21, 24])
```



6. Operation function

■ mean & std

```
test_array = np.arange(1,13).reshape(3,4)
```

```
test_array.mean() → 6.5          # 평균(Mean)
```

```
test_array.mean(axis=0)  
→ array([5., 6., 7., 8.])
```

```
test_array.std()                # 표준 편차(Standard Deviation)  
test_array.std(axis=0)
```

■ Mathematical functions

지수 함수: exp, expm1, exp2, log, log10, loglp, log2, power, sqrt

삼각 함수: sin, cos, tan, arcsin, arccos, arctan

Hyperbolic: sinh, cosh, tanh, arcsinh, arccosh, arctanh

```
np.exp(test_array)
```

```
np.sqrt(test_array)
```

6. Operation function

■ Concatenate (Numpy array를 합치는 함수)

```
a = np.array([[1, 2], [3, 4]])
```

```
b = np.array([[5, 6]])
```

```
np.vstack((a,b))
```

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

```
np.concatenate((a,b), axis=0)    # 위의 결과와 동일
```

```
a = np.array([[1], [2], [3]])
```

```
b = np.array([[2], [3], [4]])
```

```
np.hstack((a,b))
```

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

```
a = np.array([[1, 2], [3, 4]])
```

```
b = np.array([[5, 6]])
```

```
np.concatenate((a,b.T), axis=1)
```

T - Transpose

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

7. Array operation

■ Operations btw arrays (기본적인 사칙 연산 지원)

```
test_a = np.array([[1,2,3], [4,5,6]], float)
test_a + test_a
→ array([2., 4., 6.],
        [8., 10., 12.]])
test_a - test_a
test_a * test_a          # Matrix element들 간의 곱, shape이 같을 때
                        # Element-wise product
```

■ Dot product

```
test_a = np.arange(1,7).reshape(2,3)    # Matrix 곱셈
test_b = np.arange(7,13).reshape(3,2)    # (l,m) x (m,n) → (l,n)
test_a.dot(test_b)
→ array([ 58,  64],
        [139, 154]])
```

■ Transpose

```
test_a = np.arange(1,7).reshape(2,3)
test_a.transpose()
test_a.T
```

7. Array operation

■ Broadcasting (Shape이 다른 배열간 연산 지원)

```
test_matrix = np.array([[1,2,3], [4,5,6]], float)
scalar = 3
test_matrix + scalar          # Matrix - Scalar 덧셈
→ array([4., 5., 6.],
        [7., 8., 9.]])
```

```
test_matrix - scalar
test_matrix * scalar
test_matrix / scalar      # 나누기
test_matrix // scalar     # 몫
test_matrix ** 2          # 제곱
```

```
# Matrix와 Vector간의 연산도 가능
test_matrix = np.arange(1,13).reshape(4,3)
test_vector = np.arange(10,40,10)
test_matrix + test_vector
→ array([11, 22, 33],
        [14, 25, 36],
        [17, 28, 39],
        [20, 31, 42]])
```


8. Comparison

■ All & Any

```
a = np.arange(10)
```

```
➔ array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.any(a>5) ➔ True    # any – 하나라도 조건에 만족하면 True
```

```
np.any(a<0) ➔ False
```

```
np.all(a>5) ➔ False  # all – 모두가 조건을 만족해야 True
```

```
np.all(a<10) ➔ True
```

```
a > 5
```

```
➔ array([False, False, False, False, False, False, True, True, True, True],  
        dtype=bool)
```

```
test_a = np.array([1, 3, 0], float)
```

```
test_b = np.array([5, 2, 1], float)
```

```
test_a > test_b          # 배열의 크기가 동일할 때 원소간 비교 가능
```

```
➔ array([False, True, False], dtype=bool)
```

```
test_a == test_b
```

```
(test_a > test_b).any()
```

8. Comparison

■ Logical operation

```
a = np.array([1, 3, 0], float)
b = np.logical_and(a>0, a<3)    # and 조건
→ array([True, False, False], dtype=bool)
```

```
c = np.logical_not(b)
→ array([False, True, True], dtype=bool)
np.logical_or(b,c)
→ array([True, True, False], dtype=bool)
```

```
np.where(a>0, 3, 2)    # where(condition, True, False)
→ array([3., 3., 2.],
```

```
a = np.arange(10, 20)
np.where(a>15)         # index 값 반환
→ (array([6, 7, 8, 9], dtype=int64),)
```

```
a = np.array([1, np.NaN, np.Inf], float)
np.isnan(a)           # is Not a Number?
np.isfinite(a)        # is finite number?
```

8. Comparison

■ `argmax` & `argmin` (array내 최대값 또는 최소값의 index를 리턴)

```
a = np.array([1,2,4,5,8,78,23,3])  
np.argmax(a), np.argmin(a)  
➔ (5, 0)
```

```
a = np.array([[1,2,4,7],[9,88,6,45],[8,78,23,3]])  
np.argmax(a, axis=1)  
➔ array([3, 1, 1])  
np.argmax(a, axis=0)  
➔ array([1, 1, 2, 1])
```

1	2	4	7
9	88	6	45
8	78	23	3

```
np.argmin(a, axis=1)  
➔ array([0, 2, 3])  
np.argmin(a, axis=0)  
➔ array([0, 0, 0, 2])
```

9. Boolean & fancy index

■ Boolean index

```
test_array = np.array([1,4,0,2,3,8,9,7], float)
```

```
test_array > 3
```

```
→ array([False,  True,  False,  False,  False,  True,  True,  True],  
        dtype=bool)
```

```
test_array[test_array > 3]    # 조건이 True인 index의 element만 추출
```

```
→ array([4., 8., 9., 7.])
```

```
condition = test_array < 3
```

```
test_array[condition]
```

```
→ array([1., 0., 2.])
```

9. Boolean & fancy index

■ Fancy index

```
a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)    # 반드시 integer로 선언
a[b]                                     # b 배열의 값을 인덱스로 하여 a의 값들을 추출함
➔ array([2., 2., 4., 8., 6., 4.])        # bracket index
```

```
a.take(b)    # take 함수: bracket index와 같은 효과
```

```
a = np.array([[1,4], [9,16]], float)
b = np.array([0,0,1,1,1], int)
c = np.array([0,1,1,1,0], int)
a[b,c]        # b를 row index, c를 column index로 변환하여 표시
➔ array([1., 4., 16., 16., 9.])
```

10. Numpy data I/O

■ loadtxt & savetxt (Text type의 데이터를 읽고 저장하는 기능)

```
a = np.loadtxt(filename)          # 파일 호출
a[:10]

a_int = a.astype(int)
a_int[:3]

np.savetxt(filename, a_int, delimiter=',')    # csv 파일로 저장
```

■ numpy object – npy

- Numpy object(pickle) 형태로 데이터를 저장하고 불러옴
- Binary 파일 형태
np.save('npy_test', arr=a_int)

```
np_array = np.load(file='npy_test.npy')
```

11. 연습 문제

1. 길이가 10인 0-벡터를 만드세요.
2. 길이가 10이며 다섯번째 원소만 1이고 나머지 원소는 모두 0인 벡터를 만드세요.
3. 10 부터 49까지의 값을 가지는 벡터를 만드세요.
4. 위(3번) 벡터의 순서를 바꾸세요.
5. 0부터 8까지의 값을 가지는 3x3 행렬을 만드세요.
6. 벡터 $[1, 2, 0, 0, 4, 0]$ 에서 원소의 값이 0이 아닌 원소만 선택한 벡터를 만드세요.
7. 3x3 단위 행렬(identity matrix)을 만드세요
8. 난수 원소를 가지는 3x3 행렬을 만드세요
9. 위(8번)에서 만든 난수 행렬에서 최대값/최소값 원소를 찾으세요.
10. 위(8번)에서 만든 난수 행렬에서 행 평균, 열 평균을 계산하세요.

11. 연습 문제 (답)

1. 길이가 10인 0-벡터를 만드세요.

```
np.zeros(10)
```

2. 길이가 10이며 다섯번째 원소만 1이고 나머지 원소는 모두 0인 벡터를 만드세요.

```
np.where(np.arange(10)==4, 1, 0), np.eye(1, 10, k=4)
```

3. 10 부터 49까지의 값을 가지는 벡터를 만드세요.

```
np.arange(10, 50)
```

4. 위(3번) 벡터의 순서를 바꾸세요.

```
np.arange(49, 9, -1)
```

5. 0부터 8까지의 값을 가지는 3x3 행렬을 만드세요.

```
np.arange(9).reshape(3, 3)
```

6. 벡터 [1,2,0,0,4,0] 에서 원소의 값이 0이 아닌 원소만 선택한 벡터를 만드세요.

```
a = np.array([1,2,0,0,4,0], int)
a[a != 0]
```

7. 3x3 단위 행렬(identity matrix)을 만드세요

```
np.identity(3)
```

8. 난수 원소를 가지는 3x3 행렬을 만드세요

```
a = np.random.random(9).reshape(3, 3)
```

9. 위(8번)에서 만든 난수 행렬에서 최대값/최소값 원소를 찾으세요. (값)

```
a[np.argmax(a)//3, np.argmax(a)%3], np.argmin(a)
```

10. 위(8번)에서 만든 난수 행렬에서 행 평균, 열 평균을 계산하세요.

```
np.mean(a, axis = 1), np.mean(a, axis = 0)
```