

벡터, 행렬, 배열

넘파이(NumPy)는 행렬이나 대규모 다차원 배열을 쉽게 처리할 수 있도록 지원하는 파이썬의 라이브러리이다. 넘파이는 데이터 구조 외에도 수치 계산을 위해 효율적으로 구현된 기능을 제공한다.

(1) 벡터 만들기

넘파이로 1차원 배열을 만들어 보겠습니다.

넘파이 라이브러리 임포트

```
[1] import numpy as np
```

행이 하나인 벡터 만들기(1, 2, 3)

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

열이 하나인 벡터 만들기(1, 2, 3)

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

(2) 행렬 만들기

넘파이로 2차원 배열을 만들어 보겠습니다.

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

위 그림처럼 넘파이의 2차원 배열로 행렬을 만들 수 있습니다. 이 행렬은 세개의 행과 두개의 열을 가집니다.

`empty` 함수는 임의의 값이 채워진 배열을 만듭니다. 초기값은 임의대로 정해집니다.

```
[8] empty_matrix = np.empty((3, 2))
empty_matrix
```

```
array([[4.662616e-310, 0.000000e+000],
       [0.000000e+000, 0.000000e+000],
       [0.000000e+000, 0.000000e+000]])
```

`zeros` 함수는 0으로 채운 배열을 만들고, `ones`은 1로 채운 배열을 만듭니다.

```
[9] zero_matrix = np.zeros((3, 2))
zero_matrix
```

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

```
[10] one_matrix = np.ones((3, 2))
one_matrix
```

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

0 행렬에 7을 더하면 모든 원소가 7인 행렬이 만들어집니다.

```
[11] seven_matrix = zero_matrix + 7
seven_matrix

array([[7., 7.],
       [7., 7.],
       [7., 7.]])
```

특정값으로 채운 배열을 만드려면 `full` 함수를 사용합니다. 첫번째 매개변수에는 배열의 크기, 두번째 매개변수에는 채울값을 입력합니다.

```
[12] seven_matrix2 = np.full((3, 2), 7)
seven_matrix2

array([[7, 7],
       [7, 7],
       [7, 7]])
```

`seven_matrix`과 `seven_matrix2`는 어느 것이 더 효율적으로 만들어졌을까요?

(3) 희소 행렬 만들기

희소 행렬은 데이터에 0이 아닌 값이 매우 적을 때 효율적으로 나타내는 방법입니다. 희소 행렬을 만드려면 일단 행렬을 만들고 `csc_matrix` 함수를 실행하면 됩니다.

```
④ from scipy import sparse
matrix2 = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])
matrix2_sparse = sparse.csc_matrix(matrix2)
matrix2_sparse
```

↳ <3x2 sparse matrix of type '<class 'numpy.longlong'>'
with 2 stored elements in Compressed Sparse Row format>

```
[14] print(matrix2_sparse)
```

(1, 1)	1
(2, 0)	3

위 그림에서 희소 행렬을 출력하면 0이 아닌 값만 출력하는 것을 확인할 수 있습니다. 0인 원소가 매우 많은 자료에서는 계산 비용이 크게 절감됩니다.

여기서 `matrix2`에 0을 좀더 많이 추가해보겠습니다.

```
[17] matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],  
                           [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],  
                           [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])  
  
# CSR 행렬 만들기  
matrix_large_sparse = sparse.csr_matrix(matrix_large)  
# 큰 희소 행렬 출력  
print("큰 희소 행렬:")  
print(matrix_large_sparse)  
# 비교를 위해 원래 희소 행렬 출력  
print("원래 희소 행렬:")  
print(matrix2_sparse)  
  
큰 희소 행렬:  
(1, 1) 1  
(2, 0) 3  
원래 희소 행렬:  
(1, 1) 1  
(2, 0) 3
```

일반적으로는 밀집 배열에서 희소 행렬을 만들기보다는 원소의 행과 열의 색인을 직접 지정하여 희소 행렬을 만듭니다. 즉, (`data`, (`row_index`, `col_index`))로 구성된 튜플을 전달하고 `shape` 매개변수에서 0을 포함한 행렬의 전체 크기를 지정합니다.

- ➊ `matrix_large_sparse2 = sparse.csr_matrix([(1, 3), ([1, 2], [1, 0])), shape=(3, 10)])`
`print(matrix_large_sparse2)`

↳ (1, 1) 1
(2, 0) 3

(4) 원소 선택

벡터나 행렬에서 원소를 하나 이상 선택하려면 넘파이 배열로 간단하게 해결할 수 있습니다.

```
[19] vector = np.array([1, 2, 3, 4, 5, 6])
print(vector[2])

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

3
5

위 그림에서는 벡터의 3번째 원소를 선택하고 매트릭스의 2번째 행과 2번째 열에 있는 원소를 선택한 모습을 나타냅니다.

```
[20] # 벡터에 있는 모든 원소 선택
print(vector[:])

# 세 번째 원소를 포함하여 그 전의 모든 원소 선택
print(vector[:3])

# 세 번째 이후의 모든 원소 선택
print(vector[3:])

# 마지막 원소 선택
print(vector[-1])
```

☞ [1 2 3 4 5 6]
[1 2 3]
[4 5 6]
6

```
[21] # 행렬에서 첫 번째 두 행과 모든 열 선택
print(matrix[:, :2])
```

```
# 모든 행과 두 번째 열 선택
print(matrix[:, 1:2])
```

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

```
[22] # 첫 번째 행과 세 번째 행 선택
matrix[[0, 2]]
```

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

```
[23] # (0,1), (2, 0) 위치의 원소 선택
matrix[[0,2], [1,0]]
```

```
array([2, 7])
```

부울리언마스크(boolean mask) 배열로 원소를 선택할 수도 있습니다. 즉 matrix의 각 원소에 비교 연산자를 적용합니다:

```
[24] mask = matrix > 5
```

```
mask
```

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

그리고 부울리언마스크 배열로 원소를 선택합니다.

[25] matrix[mask]

```
array([6, 7, 8, 9])
```

(5) 배열 정보 확인

행렬의 크기, 원소의 갯수, 차원을 알고 싶을 때가 있습니다. 이럴 때는 `shape`와 `size`, `ndim`을 사용하면 가능합니다.

[26] matrix = np.array([[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]])

matrix.shape

```
  (3, 4)
```

[27] matrix.size

```
12
```

[28] matrix.ndim

```
2
```

위 그림에서는 매트릭스를 만들고, 행렬의 크기(`shape`), 원소의 갯수, 차원을 각각 구해보았습니다.

(6) 벡터화 연산 적용

배열의 여러 원소에 어떤 함수를 적용하고 싶을 때가 있습니다. 아래 그림에서는 간단한 함수를 만들고, `vectorize` 클래스를 함수에 적용한 예를 보여 줍니다. 결과적으로 매트릭스의 각 원소에 100을 더한 결과를 출력할 수 있습니다.

```
[29] def add_100(i):
    return i + 100
matrix = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
vectorized_add_100 = np.vectorize(add_100)
vectorized_add_100(matrix)

array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

(7) 최대값, 최소값

배열에서 최대값이나 최소값을 찾으려면 넘파이의 `max` 함수와 `min` 함수를 사용합니다.

```
[30] np.max(matrix)
```

9

```
[31] np.min(matrix)
```

1

위 그림은 조금전 `matrix`에서 최대값과 최소값을 구한 결과를 나타내고 있습니다.

```
[32] # 각 열에서 최대값 찾기
np.max(matrix, axis=0)
```

array([7, 8, 9])

```
[33] # 각 행에서 최대값 찾기
np.max(matrix, axis=1)
```

array([3, 6, 9])

위 그림에서는 `matrix`의 각 열 중 최대값이 들어있는 열과 행을 찾는 모습을 보여 줍니다.

```
[34] vector_column = np.max(matrix, axis=1, keepdims=True)
vector_column
```

```
array([3,
       6,
       9])
```

```
[35] matrix - vector_column
```

```
array([[-2, -1,  0],
       [-2, -1,  0],
       [-2, -1,  0]])
```

위 그림은 `keepdims=True`를 사용하여 원본 자료의 모습을 지키면서 원하는 연산을 수행하는 모습을 나타냅니다. 참고만 하세요.

(8) 평균, 분산, 표준편차

평균과 분산과 표준편차를 구하려면 넘파이의 `mean`, `var`, `std` 함수를 사용하면 됩니다.

```
[36] np.mean(matrix)
```

```
5.0
```

```
[37] np.var(matrix)
```

```
6.666666666666667
```

```
[39] np.std(matrix)
```

```
2.581988897471611
```

위 그림처럼 평균, 표준편차, 분산을 구할 수 있습니다. 여기서 한 축의 통계를 구하려면 다음과 같이 축 매개변수를 대입합니다. 예를 들어 각 열의 평균을 계산하고 싶다면 `mean` 함수의 매개 변수로 열을 지정합니다.

```
[40] np.mean(matrix, axis=0)
```

```
array([4., 5., 6.])
```

(9) 배열 크기 변경

원소의 값에는 변화가 없이, 배열의 크기(행과 열의 수)를 바꿀 수 있습니다. 이 때는 넘파이의 `reshape` 함수를 사용합니다.

▶ # 4X3 행렬 만들기
`matrix = np.array([[1, 2, 3],
[4, 5, 6],
[7, 8, 9],
[10, 11, 12]])`

2X6 행렬로 변경
`matrix.reshape(2, 6)`

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

[7, 8, 9, 10, 11, 12]])
```

`reshape` 함수는 데이터 변경 없이 배열의 구조를 바꾸어서 행과 열의 수를 다르게 만들 수 있습니다. 변경되는 행렬의 원소의 갯수는 원래 행렬의 원소의 갯수와 같아야 합니다.

[44] `matrix.reshape(1, 12)`

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

[42] `matrix.reshape(1, -1)`

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

[43] `matrix.reshape(12)`

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

[45] `matrix.reshape(-1)`

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

위 그림에서 `-1` 옵션은 “가능한한 많이”라는 의미를 내포합니다. `(1, -1)`로 실행시 원래 `matrix`의 원소의 갯수로 되돌려 준다는 뜻입니다. `(1, 12)`로 실행한 것과 같은 결과를 나타냅니다. 한편, `reshape`에 단일 변수로 `-1`을 입력하면 결과를 1차원으로 되돌려 줍니다.

(10) 벡터 또는 행렬의 전치
벡터나 행렬을 전치하려면 `T` 메소드를 사용하면 됩니다.

```
[46] vector = np.array([1, 2, 3, 4, 5, 6])
vector.T
array([1, 2, 3, 4, 5, 6])
```

```
[48] vector2 = np.array([[1, 2, 3, 4, 5, 6]])
vector2.T
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

```
[49] matrix = np.array([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]])
matrix.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

위 그림에서 보는 바와 같이 1차원 배열에서 전치한 결과는 동일합니다.

(11) 행렬 펼치기
행렬을 1차원 배열로 펼치려면 `flatten` 메소드를 사용하면 됩니다.

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

위 그림에서는 `flatten` 메소드를 실행한 결과를 보여줍니다. 물론 앞에서의 예제에서처럼 `reshape`를 사용할 수도 있습니다.