# NumPy 배열의 기초

파이썬에서 데이터 처리는 Numpy 배열 처리와 거의 비슷하다. 더 최신 도구인 Pandas 도 Numpy 배열을 기반으로 만들어졌다. 이번 절에서는 데이터와 하위 배열에 접근하고 배열을 분할, 재구성, 결합하기 위해 Numpy 의 배열 조작을 사용하는 예제를 여러 개 소개한다. 여기서 소개하는 연산 유형이 다소 무미건조하고 현학적인 것처럼 보일 수 있다.

여기서는 기본 배열 조작의 일부 범주를 다룰 것이다.

- 배열 속성 지정: 배열의 크기, 모양, 메모리 소비량, 데이터 타입을 결정한다.
- *배열의 인덱싱*: 개별 배열 요소 값을 가져오고 설정한다.
- *배열 슬라이싱* : 큰 배열 내에 있는 작은 하위 배열을 가져오고 설정한다.
- *배열 재구조화*: 해당 배열의 형상을 변경한다.
- *배열 결합 및 분할* : 여러 배열을 하나로 결합하고 하나의 배열을 여러개로 분할한다.

# NumPy 배열 속성 지정

우선 몇 가지 유용한 배열 속성을 알아보자. 1 차원, 2 차원, 3 차원 난수 배열을 정의해 보자 Numpy 난수 생성기를 사용할 텐데, 이 코드가 실행될 때마다 똑같은 난수 배열이 생성되도록 시드 값을 설정할 것이다.

```
In [1]:import numpy as np
        np.random.seed(0)  # 재현 가능성을 위한 시드 값 랜덤값 만들기 위한
       x1 = np.random.randint(10, size=6)  # 1 차원 배열
       x2 = np.random.randint(10, size=(3, 4))  # 2 차원 배열
       x3 = np.random.randint(10, size=(3, 4, 5))  # 3 차원 배열
```

각 배열은 속성으로 ndim(차원의 개수), shape(각 차원의 크기), size(전체 배열 크기)를 가지고 있다:

```
In [2]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

다른 유용한 속성으로 배열의 데이터 타입인 dtype 이 있다.

```
In [3]: print("dtype:", x3.dtype)
dtype: int64
```

그 밖의 속성으로는 각 배열요소의 크기를 바이트 단위로 보여주는 itemsize 와 배열의 전체크기를 바이트 단위로 보여주는 nbyte 가 있다.

```
In [4]: print("itemsize:", x3.itemsize, "bytes")
        print("nbytes:", x3.nbytes, "bytes")
itemsize: 8 bytes
nbytes: 480 bytes
```

일반적으로 nbyte 는 itemsize 를 size 로 곱한 값과 같다고 볼 수 있다.

# 배열의 인덱싱 : 단일 요소에 접근하기

파이썬의 표준 리스트 인덱싱에 익숙한 독자라면 Numpy 의 인덱싱도 꽤 친숙하게 느껴질 것이다. 1 차원 배열에서 i 번째(0 부터 시작)값에 접근하려면 파이썬 리스트에서와 마찬가지로 대괄호 안에 원하는 인덱스를 지정하면 된다.

```
In [5]:x1
Out[5]:array([5, 0, 3, 3, 7, 9])
In [6]:x1[0]
Out[6]:5
In [7]:x1[4]
Out[7]:7
```

배열의 끝에서부터 인덱싱하려면 음수 인덱스를 사용하면 된다.

```
In [8]:x1[-1]
Out[8]:9
In [9]:x1[-2]
Out[9]:7
```

다차원 배열에서는 콤마로 구분된 인덱스 튜플을 이용해 배열 항목에 접근할 수 있다.

```
In [10]:x2
Out[10]:array([[3, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])
In [11]:x2[0, 0]
Out[11]:3
In [12]:x2[2, 0]
Out[12]:1
In [13]:x2[2, -1]
Out[13]:7
```

위 인덱스 표기법을 사용해 값을 수정할 수도 있다.

```
In [14]:x2[0, 0] = 12
      x2
```

```
Out[14]:array([[12,  5,  2,  4],
               [ 7,  6,  8,  8],
               [ 1,  6,  7,  7]])
```

파이썬 리스트와 달리 NumPy 배열운 고정 타입을 가지다는 점을 명심하라. 이 말은 가령 정수 배열에 부동 소수점 값을 삽입하려고 하면 아무 말 없이 그 값의 소수점 이하를 잘라버릴 거라는 뜻이다. 이러한 동작 방식에 주의하자.

```
In [15]:x1[0] = 3.14159 # 이 값의 소수점 이하는 잘릴 것이다.
     x1
Out[15]:array([3, 0, 3, 3, 7, 9])
```

# 배열 슬라이싱: 하위 배열 에 접근하기

꺽쇠괄호를 사용해 개별 요소에 접근할 수 있는 것처럼 콜론(:)기호로 표시되는 슬라이스(slice) 표기법으로 하위 배열에 접근할 수 있다. Numpy 슬라이싱 구문은 표준 파이썬 리스트의 구문을 따른다. 배열 x 의 슬라이스에 접근하려면 다음 구문을 사용하면 된다.

```
x[start:stop:step]
```

이 가운데 하아라도 지정되지 않으면 기본으로 start=0, stop=차원 크기, step=1 로 값이 살장된다. 이제 1 차원과 다차원에서 하위 배열에 접근하는 방법을 살펴본다.

## 1 차원 하위 배열

```
In [16]:x = np.arange(10)
     x
Out[16]:array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [17]:x[:5] # 첫 다섯개 요소
Out[17]:array([0, 1, 2, 3, 4])
In [18]:x[5:] # 인덱스 5 다음 요소
Out[18]:array([5, 6, 7, 8, 9])
In [19]:x[4:7] # 중간 하위 배열
Out[19]:array([4, 5, 6])
In [20]:x[::2] # 하나 걸러 하나씩의 요소로 구성된 배열
Out[20]:array([0, 2, 4, 6, 8])
In [21]:x[1::2] # 인덱스 1 에서 시작해 하나 걸러 하나씩 요소로 구성된 배열
Out[21]:array([1, 3, 5, 7, 9])
```

혼동을 줄 수 있는 경우는 step 값이 음수일 때다. 이 경우에는 start 와 stop 의 기본값이 서로 바뀐다. 이는 배열을 거꾸로 만드는 편리한 방법이 될 수 있다.

```
In [22]:x[::-1] # 모든 요소를 거꾸로 나열
Out[22]:array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
In [23]:x[5::-2] # 인덱스 5 부터 하나 걸러 하나씩
Out[23]:array([5, 3, 1])
```

## 다차원 하위 배열

다 차원 슬라이싱도 콤마로 구분된 다중 슬라이스를 사용해 똑같은 방식으로 동작한다. 예.

```
In [24]: x2
Out[24]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
In [25]: x2[:2, :3] # 두 개의 행, 세 개의 열
Out[25]: array([[12,  5,  2],
                [ 7,  6,  8]])
In [26]: x2[:3, ::2] # 모든 행, 한 열 걸러 하나씩
Out[26]:array([[12,  2],
               [ 7,  8],
               [ 1,  7]])
```

마지막으로 하위 배열 차원도 함께 역으로 반환할 수 있다.

```
In [27]:x2[::-1, ::-1]
Out[27]:array([[ 7,  7,  6,  1],
               [ 8,  8,  6,  7],
               [ 4,  2,  5, 12]])
```

### 배열의 행과 열에 접근하기

한 가지 공통으로 필요한 루틴은 배열의 단일 행이나 열에 접근하는 것이다. 이것은 단일 콜론으로 표시된 빈 슬라이스를 사용해 인덱싱과 슬라이싱을 결합함으로써 할 수 있다.

```
In [28]:print(x2[:, 0]) # x2 의 첫 번째 열
[12  7  1]
In [29]:print(x2[0, :]) #x2 의 첫 번째 행
[12  5  2  4]
```

행에 접근하는 경우 더 간결한 구문을 위해 빈 슬라이스를 생략할 수 있다.

```
In [30]:print(x2[0]) # x2[0, :]와 동일
[12  5  2  4]
```

### 사본이 아닌 뷰로서의 하위 배열

배열 슬라이스에 대해 알아야 할 중요하고 매우 유용한 사실 하나는 배열 슬라이스가 배열 데이터의 사본(copy)이 아니라 뷰(view)를 반환한다는 점이다. 이는 Numpy 배열 슬라이싱이 파이썬 리스트 슬라이싱과 다른 점 중 하나다. 리스트에서 슬라이스는 사본이다.

```
In [31]: print(x2)
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

이 배열에서 2x2 하위 배열을 추출해 보자

```
In [32]: x2_sub = x2[:2, :2]
      print(x2_sub)
[[12  5]
 [ 7  6]]
```

이제 이 하위 배열을 수정하면 원래 배열이 변경되는 것을 보게 될 것이다.

```
In [33]:x2_sub[0, 0] = 99
      print(x2_sub)
[[99  5]
 [ 7  6]]
In [34]:print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

이 기본 행위는 실제로 매우 유용하다. 이것은 곧 데이터세트를 다룰 때 기반 데이터 버퍼를 복사하지 않아도 이 데이터의 일부에 접근하고 처리할 수 있다는 뜻이다.

## 배열의 사본 만들기

배열 뷰의 훌륭한 기능에도 불구하고 때로는 배열이나 하위 배열 내의 데이터를 명시적 으로 복사하는 것이 더 유용할 때가 있다. 그 작업은 copy()메서드로 가장 쉽게 할 수 있다.

```
In [35]: x2_sub_copy = x2[:2, :2].copy()
      print(x2_sub_copy)
[[99  5]
 [ 7  6]]
```

이제 이 하위 배열을 수정해도 원래 배열이 그대로 유지된다.

```
In [36]: x2_sub_copy[0, 0] = 42
      print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
In [37]:print(x2)
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

## 배열 재구조화

다른 유용한 조작 유형은 배열의 형상을 변경하는 것이다. 이를 유연하게 하는 방법은 reshape() 메서드를 사용하는 것이다. 가령 3x3 그리드에 숫자 1 부터 9 까지 넣고자 한다면 다음과 같이 하면된다.

```
In [38]:grid = np.arange(1, 10).reshape((3, 3))
        print(grid)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

이 코드가 동작하려면 초기 배열의 규모가 형상이 변경된 배열의 규모와 일치해야 한다. 가능하다면 reshape 메서드가 초기 배열의 사본이 아닌 뷰를 사용하겠지만, 연속되지 않은 메모리 버퍼일 경우에는 그렇지 않을 수도 있다.

또 다른 일 반적인 재구조화 패턴은 1 차원 배열을 2 차원 행이나 열 매트릭스로 전환하는 것이다. 이 작업은 reshap 메서드로 할 수 있으며, 그렇지 않으면 슬라이스 연산 내에 newaxis 키워드를 사용해 더 쉽게 할 수 있다.

```
In [39]:x = np.array([1, 2, 3])
        # reshape 을 이용한 행 벡터
        x.reshape((1, 3))
Out[39]:array([[1, 2, 3]])
In [40]:# newaxis 를 이용한 행 벡터
        x[np.newaxis, :]
Out[40]:array([[1, 2, 3]])
In [41]:# reshape 을 이용한 열 벡터
        x.reshape((3, 1))
Out[41]:array([[1],
                       [2],
                       [3]])
In [42]:# newaxis 를 이용한  열 벡터
        x[:, np.newaxis]
```

```
Out[42]:array([[1],
                    [2],
                    [3]])
```

책의 나머지 부분에서도 종종 이러한 유형의 변형을 보게 될 것이다.

# 배열 연결 및 분할

지금까지 본 루틴은 모두  단일 배열에서 동작한다. 아울러 여러 배열을 하나로 결합하거나 그 반대로 하나의 배열을 여러 개의 배열로 분할하는 것도 가능하다. 여기서는 그러한 연산을 알아보겠다.

## 배열 연결

Numpy 에서는 주로 np.concatenate, np.vstack, np.hstack 루틴을 이용해 두 배열을 결합하거나 연결한다. 여기서 보다시피 np.concatenate 는 튜플이나 배열의 리스트를 첫 번째 인수로 취한다.

```
In [43]:x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])
Out[43]:array([1, 2, 3, 3, 2, 1])
```

또 한번에 두개 이상의 배열을 연결할 수도 있다.

```
In [44]:z = [99, 99, 99]
        print(np.concatenate([x, y, z]))
[ 1  2  3  3  2  1 99 99 99]
```

np.concatenate 는 2 차원 배열에서도 사용할 수 있다.

```
In [45]:grid = np.array([[1, 2, 3],
           [4, 5, 6]])
In [46]:# 첫번째 축을 따라 연결
        np.concatenate([grid, grid])
Out[46]:array([[1, 2, 3],
                    [4, 5, 6],
                    [1, 2, 3],
                    [4, 5, 6]])
In [47]:# 두 번째 축을 따라 연결(0 부터 시작하는 인덱스 방식)
        np.concatenate([grid, grid], axis=1)
Out[47]:array([[1, 2, 3, 1, 2, 3],
                   [4, 5, 6, 4, 5, 6]])
```

혼합된 차원의 배열로 작업할 때는 `np.vstack` (수직 스택,vertical stack) and `np.hstack` (수평 스택, horizontal stack) 함수를 사용하는 것이 더 명확하다.

- axis=None 은 기본값으로 모든 요소의 값을 합산하여 1 개의 스칼라값을 반환합니다.
- axis=0 은 x 축을 기준으로 여러 row 를 한 개로 합치는 과정입니다.
- axis=1 은 y 축을 기준으로 row 별로 존재하는 column 들의 값을 합쳐 1 개로 축소하는 과정입니다.
- axis=2 는 z 축을 기준으로 column 의 depth 가 가진 값을 축소하는 과정입니다.

```
In [48]:x = np.array([1, 2, 3])
        grid = np.array([[9, 8, 7],
            [6, 5, 4]])
        # 배열을 수직으로 쌓음
        np.vstack([x, grid])
Out[48]:array([[1, 2, 3],
                      [9, 8, 7],
                      [6, 5, 4]])
In [49]:# 배열을 수평으로 쌓음
        y = np.array([[99],
          [99]])
        np.hstack([grid, y])
Out[49]:array([[ 9,  8,  7, 99],
                      [ 6,  5,  4, 99]])
```

이와 마찬가지로 `np.dstack` 은 세번째 축을 따라 배열을 쌓을 것이다.

## 배열 분할하기

결합의 반대는 분할로, np.split, np.hsplit, np.vsplit 함수로 구현된다. 각 함수에 분할 지점을 알려주는 인덱스 목록을 전달할 수 있다.

```
In [50]: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]
```

N 개의 분할점은 N+1 개의 하위 배열을 만든다. 관련 함수인 np.hsplit 과 np.vsplit 은 서로 비슷하다.

```
In [51]:
grid = np.arange(16).reshape((4, 4))
grid
Out[51]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
In [52]: upper, lower = np.vsplit(grid, [2])
         print(upper)
         print(lower)
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
In [53]: left, right = np.hsplit(grid, [2])
         print(left)
         print(right)
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

비슷하게 np.dsplit 은 세번째 축을 따라 배열을 분할할 것이다.