

# 넘파이 (Numpy)

```
def parse_url(url, css_selector):  
    r = requests.get(url)  
    soup = BeautifulSoup(r.content, 'lxml')  
    s = soup.select_one(css_selector)  
    with open('article.txt', 'w+') as f:  
        f.write(s.text.strip())  
    return f.name
```

소프트웨어융합대학원  
진혜진



# 넴파이 배열 연산



## 03 넘파이 배열 연산

### 1. 연산 함수

- 연산 함수(operation function) : 배열 내부 연산을 지원하는 함수
- 축(axis): 배열의 랭크가 증가할 때마다 새로운 축이 추가되어 차원 증가
- sum 함수 : 각 요소의 합을 반환

In [1]:	<pre>import numpy as np test_array = np.arange(1, 11) test_array.sum()</pre>
Out [1]:	55

## 03 넘파이 배열 연산

- sum 함수를 랭크가 2 이상인 배열에 적용할 때 축으로 연산의 방향을 설정

In [2]:	test_array = np.arange(1,13).reshape(3,4) test_array
Out [2]:	array([[ 1,  2,  3,  4], [ 5,  6,  7,  8], [ 9, 10, 11, 12]])
In [3]:	test_array.sum(axis=0)
Out [3]:	array([15, 18, 21, 24])
In [4]:	test_array.sum(axis=1)
Out [4]:	array([10, 26, 42])

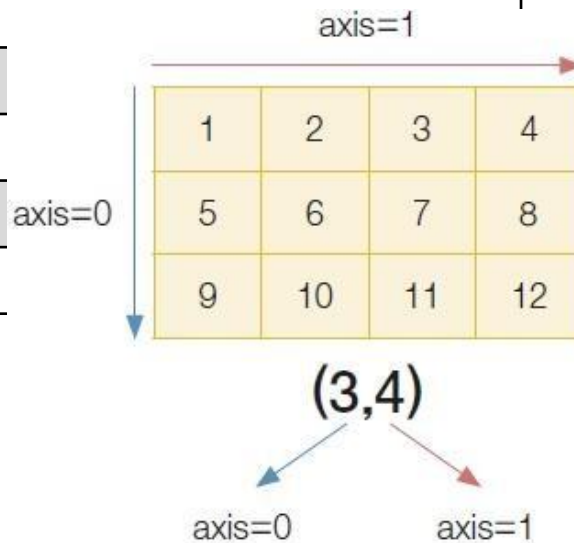
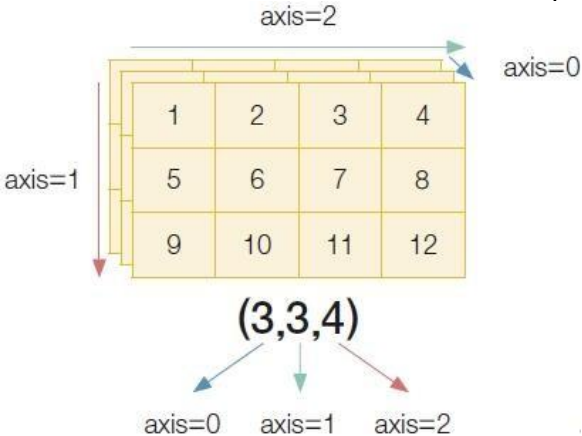


그림 3-12 축에 따른 연산

## 03 넘파이 배열 연산

In [5]:	<pre>test_array = np.arange(1, 13).reshape(3, 4) third_order_tensor = np.array([test_array, test_array, test_array]) third_order_tensor</pre>
Out [5]:	<pre>array([[[ 1,  2,  3,  4],         [ 5,  6,  7,  8],         [ 9, 10, 11, 12]],        [[ 1,  2,  3,  4],         [ 5,  6,  7,  8],         [ 9, 10, 11, 12]],        [[ 1,  2,  3,  4],         [ 5,  6,  7,  8],         [ 9, 10, 11, 12]]])</pre> 

## 03 넘파이 배열 연산

In [6]:	<code>third_order_tensor.sum(axis=0)</code>
Out [6]:	<code>array([[ 3,  6,  9, 12],        [15, 18, 21, 24],        [27, 30, 33, 36]])</code>
In [7]:	<code>third_order_tensor.sum(axis=1)</code>
Out [7]:	<code>array([[15, 18, 21, 24],        [15, 18, 21, 24],        [15, 18, 21, 24]])</code>
In [8]:	<code>third_order_tensor.sum(axis=2)</code>
Out [8]:	<code>array([[10, 26, 42],        [10, 26, 42],        [10, 26, 42]])</code>

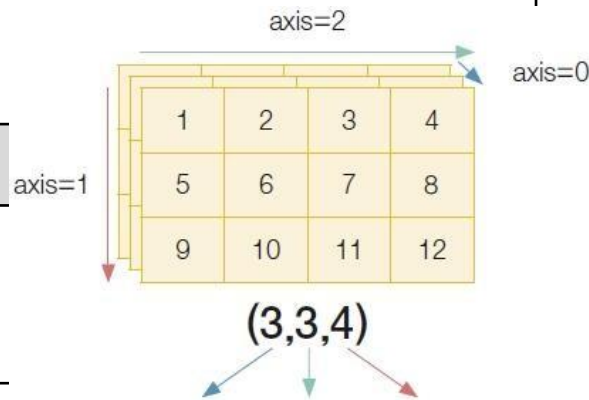


그림 3-13 3차원 텐서에서의 연산

## 03 넘파이 배열 연산

In [9]:	test_array = np.arange(1, 13).reshape(3, 4) test_array
Out [9]:	array([[ 1,  2,  3,  4], [ 5,  6,  7,  8], [ 9, 10, 11, 12]])
In [10]:	test_array.mean(axis=1) # axis=1 축을 기준으로 평균 연산
Out [10]:	array([ 2.5,  6.5, 10.5])
In [11]:	test_array.std() # 전체 값에 대한 표준편차 연산
Out [11]:	3.452052529534663



## 03 넘파이 배열 연산

In [12]:	test_array.std(axis=0) # axis=0 축을 기준으로 표준편차 연산
Out [12]:	array([3.26598632, 3.26598632, 3.26598632, 3.26598632])
In [13]:	np.sqrt(test_array) # 각 요소에 제곱근 연산 수행
Out [13]:	array([[1. , 1.41421356, 1.73205081, 2. ], [2.23606798, 2.44948974, 2.64575131, 2.82842712], [3. , 3.16227766, 3.31662479, 3.46410162]])

## 03 넘파이 배열 연산

### 2. 연결 함수

- 연결 함수(concatenation functions) : 두 객체 간의 결합을 지원하는 함수
- vstack 함수 : 배열을 수직으로 붙여 하나의 행렬을 생성
- hstack 함수 : 배열을 수평으로 붙여 하나의 행렬을 생성

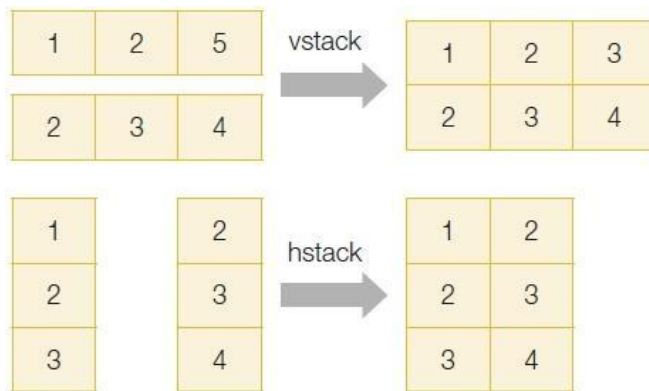


그림 3-14 연결을 통한 행렬의 결합

## 03 넘파이 배열 연산

- 넘파이는 열 벡터를 표현할 수 없어 2차원 행렬 형태로 표현

In [14]:	<pre>v1 = np.array([1, 2, 3]) v2 = np.array([4, 5, 6]) np.vstack((v1, v2))</pre>
Out [14]:	<pre>array([[1, 2, 3],        [4, 5, 6]])</pre>
In [15]:	<pre>np.hstack((v1,v2))</pre>
Out [15]:	<pre>array([1, 2, 3, 4, 5, 6])</pre>

- 벡터 형태 그대로 hstack을 붙일 경우 그대로 벡터 형태의 배열 생성

## 03 넘파이 배열 연산

In [16]:	<pre>v1 = v1.reshape(-1, 1) v2 = v2.reshape(-1, 1) v1</pre>
Out [16]:	<pre>array([[1],        [2],        [3]])</pre>
In [17]:	<pre>V2</pre>
Out [17]:	<pre>array([[4],        [5],        [6]])</pre>
In [18]:	<pre>np.hstack((v1,v2))</pre>
Out [18]:	<pre>array([[1, 4],        [2, 5],        [3, 6]])</pre>

- 2차원 행렬 형태로 표현한 열 벡터를 hstack으로 연결

## 03 넘파이 배열 연산

- concatenate 함수 : 축을 고려하여 두 개의 배열을 결합
  - 스택(stack) 계열의 함수와 달리 생성될 배열과 소스가 되는 배열의 차원이 같아야 함
  - 두 벡터를 결합하고 싶다면, 해당 벡터를 일단 2차원 배열 꼴로 변환 후 행렬로 나타내야 함

In [19]:	<pre>v1 = np.array([[1, 2, 3]]) v2 = np.array([[4, 5, 6]]) np.concatenate((v1,v2), axis=0)</pre>
Out [19]:	<pre>array([[1, 2, 3],        [4, 5, 6]])</pre>

- v1과 v2 모두 사실상 행렬이지만 벡터의 형태
- 매개변수 axis=0로 행을 기준으로 연결

## 03 넘파이 배열 연산

In [20]:	<pre>v1 = np.array([1, 2, 3, 4]).reshape(2,2) v2 = np.array([[5,6]]).T v1</pre>
Out [20]:	<pre>array([[1, 2],        [3, 4]])</pre>
In [21]:	<pre>v2</pre>
Out [21]:	<pre>array([[5],        [6]])</pre>
In [22]:	<pre>np.concatenate((v1,v2), axis=1)</pre>
Out [22]:	<pre>array([[1, 2, 5],        [3, 4, 6]])</pre>

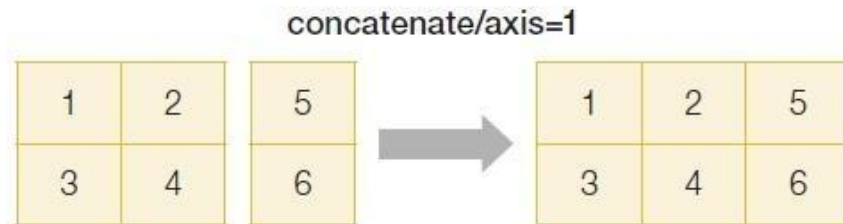


그림 3-15 배열의 연결

## 03 넘파이 배열 연산

### 3. 사칙연산 함수

- 넘파이는 파이썬과 동일하게 배열 간 사칙연산 지원
  - 행렬과 행렬, 벡터와 벡터 간 사칙연산이 가능
- 같은 배열의 구조일 때 요소별 연산(element-wise operation)
  - 요소별 연산 : 두 배열의 구조가 동일할 경우 같은 인덱스 요소들끼리 연산

1	2	3	4
5	6	7	8
9	10	11	12

 × 

1	2	3	4
5	6	7	8
9	10	11	12

 = 

1	4	9	16
25	36	49	64
81	100	121	144

그림 3-16 배열 간 요소별 연산의 예시

## 03 넘파이 배열 연산

In [23]:	<code>x = np.arange(1, 7).reshape(2,3)</code> <code>x</code>
Out [23]:	<code>array([[1, 2, 3],        [4, 5, 6]])</code>
In [24]:	<code>x + x</code>
Out [24]:	<code>array([[ 2,  4,  6],        [ 8, 10, 12]])</code>
In [25]:	<code>x - x</code>
Out [25]:	<code>array([[0, 0, 0],        [0, 0, 0]])</code>
In [26]:	<code>x / x</code>
Out [26]:	<code>array([[1., 1., 1.],        [1., 1., 1.]])</code>
In [27]:	<code>x ** x</code>
Out [27]:	<code>array([[ 1,  4, 27],        [256, 3125, 46656]], dtype=int32)</code>



## 03 넘파이 배열 연산

- 배열 간의 곱셈에서는 요소별 연산과 벡터의 내적(dot product) 연산 가능
  - 벡터의 내적 : 두 배열 간의 곱셈
  - 두 개의 행렬에서 첫 번째 행렬의 열 크기와 두 번째 행렬의 행 크기가 동일해야 함
  - $m \times n$  행렬과  $n \times l$  행렬, 벡터의 내적 연산하면  $m \times l$ 의 행렬 생성

$$(m \times n) \cdot (n \times k) = (m \times k)$$

곱셈 연산이 정의됨

그림 3-17 벡터의 내적(dot product) 연산

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1 \cdot 5) + (2 \cdot 7) & (1 \cdot 6) + (2 \cdot 8) \\ (3 \cdot 5) + (4 \cdot 7) & (3 \cdot 6) + (4 \cdot 8) \end{bmatrix}$$
$$= \begin{bmatrix} 5 + 14 & 6 + 16 \\ 15 + 28 & 18 + 32 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

그림 3-18 벡터의 내적(dot product) 연산의 예시

## 03 넘파이 배열 연산

- dot 함수 : 벡터의 내적 연산

In [28]:	<pre>x_1 = np.arange(1, 7).reshape(2,3) x_2 = np.arange(1, 7).reshape(3,2) x_1</pre>
Out [28]:	<pre>array([[1, 2, 3],        [4, 5, 6]])</pre>
In [29]:	<pre>x_2</pre>
Out [29]:	<pre>array([[1, 2],        [3, 4],        [5, 6]])</pre>
In [30]:	<pre>x_1.dot(x_2)</pre>
Out [30]:	<pre>array([[22, 28],        [49, 64]])</pre>

- 2×3 행렬과 3×2 행렬의 연산 결과는 2×2 행렬

## 03 넘파이 배열 연산

- 브로드캐스팅 연산(broadcasting operations) :  
하나의 행렬과 스칼라 값들 간의 연산이나 행렬과 벡터 간의 연산
  - 방송국의 전파가 퍼지듯 뒤에 있는 스칼라 값이 모든 요소에 퍼지듯이 연산



그림 3-19 브로드캐스팅 연산

## 03 넘파이 배열 연산

In [31]:	<code>x = np.arange(1, 10).reshape(3,3)</code> <code>x</code>
Out [31]:	<code>array([[1, 2, 3],        [4, 5, 6],        [7, 8, 9]])</code>
In [32]:	<code>x + 10</code>
Out [32]:	<code>array([[11, 12, 13],        [14, 15, 16],        [17, 18, 19]])</code>
In [33]:	<code>x - 2</code>
Out [33]:	<code>array([[ -1,  0,  1],        [ 2,  3,  4],        [ 5,  6,  7]])</code>

## 03 넘파이 배열 연산

In [34]:	x // 3
Out [34]:	array([[0, 0, 1], [1, 1, 2], [2, 2, 3]], dtype=int32)
In [35]:	x ** 2
Out [35]:	array([[ 1,  4,  9], [16, 25, 36], [49, 64, 81]], dtype=int32)

## 03 넘파이 배열 연산

- 행렬과 스칼라 값 외에 행렬과 벡터, 벡터와 벡터 간에도 연산

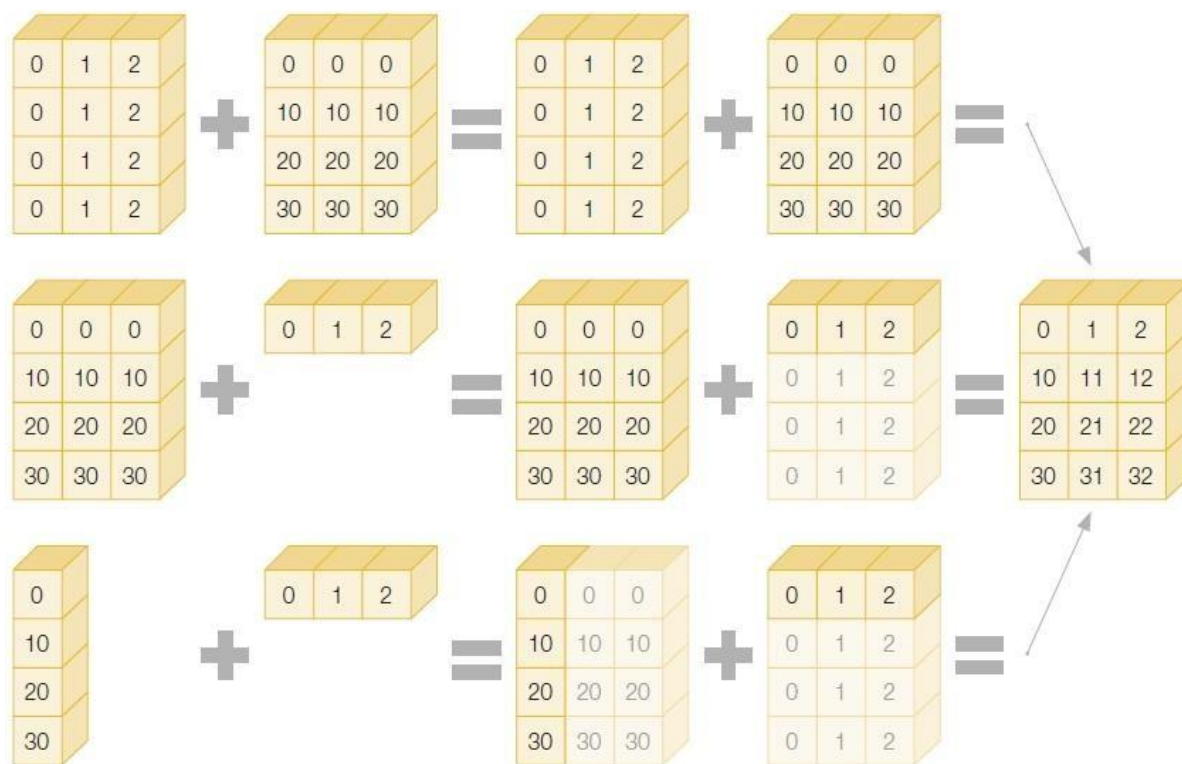


그림 3-20 확장된 브로드캐스팅 연산

## 03 넘파이 배열 연산

In [36]:	<pre>x = np.arange(1, 13).reshape(4,3) x</pre>
Out [36]:	<pre>array([[ 1, 2, 3],        [ 4, 5, 6],        [ 7, 8, 9],        [10, 11, 12]])</pre>
In [37]:	<pre>v = np.arange(10, 40, 10) v</pre>
Out [37]:	<pre>array([10, 20, 30])</pre>
In [38]:	<pre>v = np.arange(10, 40, 10) v</pre>
Out [38]:	<pre>array([10, 20, 30])</pre>
In [39]:	<pre>x + v</pre>
Out [39]:	<pre>array([[11, 22, 33],        [14, 25, 36],        [17, 28, 39],        [20, 31, 42]])</pre>

## 03 넘파이 배열 연산

1	2	3		10	20	30		11	22	33
4	5	6						14	25	36
7	8	9	+				=	17	28	39
10	11	12						20	31	42

그림 3-21 행렬과 벡터의 브로드캐스팅 연산

- 뒤에 있는 벡터가 앞에 있는 행렬과 크기를 맞추기 위해 4×3의 행렬처럼 복제
- 그 다음 요소별 연산처럼 연산



# 비교 연산과 데이터 추출



## 04 비교 연산과 데이터 추출

### 1. 비교 연산

- 연산 결과는 항상 불린형(boolean type)을 가진 배열로 추출

#### 1.1 브로드캐스팅 비교 연산

- 하나의 스칼라 값과 벡터 간의 비교 연산은 벡터 내 전체 요소에 적용

In [1]:	<pre>import numpy as np x = np.array([4, 3, 2, 6, 8, 5]) x &gt; 3</pre>
Out [1]:	<pre>array([ True, False, False,  True,  True,  True])</pre>

## 04 비교 연산과 데이터 추출

### 1.2 요소별 비교 연산

- 두 개의 배열 간 배열의 구조(shape)가 동일한 경우
- 같은 위치에 있는 요소들끼리 비교 연산
- $[1 > 2, 3 > 1, 0 > 7]$  과 같이 연산이 실시된 후 이를 반환

In [2]:	<pre>x = np.array([1, 3, 0]) y = np.array([2, 1, 7]) x &gt; y</pre>
Out [2]:	<pre>array([False,  True, False])</pre>

# 04 비교 연산과 데이터 추출

## 2. 비교 연산 함수

### 1. all과 any

- all 함수 : 배열 내부의 모든 값이 참일 때는 True,  
하나라도 참이 아닐 경우에는 False를 반환
  - and 조건을 전체 요소에 적용
- any 함수 : 배열 내부의 값 중 하나라도 참일 때는 True, 모두 거짓일 경우 False를 반환
  - or 조건을 전체 요소에 적용

## 04 비교 연산과 데이터 추출

In [3]:	<code>x = np.array([4, 6, 7, 3, 2])</code> <code>(x &gt; 3)</code>
Out [3]:	<code>array([ True,  True,  True, False, False])</code>

- `x > 3` 브로드캐스팅이 적용되어 불린형으로 이루어진 배열 반환

In [4]:	<code>(x &gt; 3).all()</code>
Out [4]:	<code>False</code>
In [5]:	<code>(x &gt; 3).any()</code>
Out [5]:	<code>True</code>

- `all` 함수를 적용하면 2개의 거짓이 있기 때문에 `False`를 반환
- `any` 함수를 적용하면 참이 있기 때문에 `True`를 반환

## 04 비교 연산과 데이터 추출

In [6]:	<code>(x &lt; 10).any()</code>
Out [6]:	True
In [7]:	<code>(x &lt; 10).all()</code>
Out [7]:	True
In [8]:	<code>(x &gt; 10).any()</code>
Out [8]:	False

- $x > 10$ 의 경우 모든 값이 10을 넘지 못하므로 모두 거짓인데, 여기에 `any` 함수를 적용하면 `False`를 반환

## 04 비교 연산과 데이터 추출

### 2.2 인덱스 반환 함수

- where 함수 : 배열이 불린형으로 이루어졌을 때  
참인 값들의 인덱스를 반환

In [9]:	<code>x = np.array([4, 6, 7, 3, 2])</code> <code>x &gt; 5</code>
Out [9]:	<code>array([False, True, True, False, False])</code>
In [10]:	<code>np.where(x&gt;5)</code>
Out [10]:	<code>(array([1, 2], dtype=int64),)</code>

- `x > 5`를 만족하는 값은 6과 7
- 6과 7의 인덱스 값인 `[1, 2]`를 반환

## 04 비교 연산과 데이터 추출

- True/False 대신 참/거짓인 경우의 값을 지정할 수 있음

In [11]:	<code>x = np.array([4, 6, 7, 3, 2])</code> <code>np.where(x&gt;5 , 10, 20)</code>
Out [11]:	<code>array([20, 10, 10, 20, 20])</code>

- 참일 경우에 10을, 거짓일 경우에 20을 반환



## 04 비교 연산과 데이터 추출

### 2.3 정렬된 값의 인덱스를 반환해주는 함수

- `argsort` : 배열 내 값들을 작은 순서대로 인덱스를 반환
- `argmax` : 배열 내 값들 중 가장 큰 값의 인덱스를 반환
- `argmin` : 배열 내 값들 중 가장 작은 값의 인덱스를 반환

In [12]:	<code>x = np.array([4, 6, 7, 3, 2])</code> <code>np.argsort(x)</code>
Out [12]:	<code>array([4, 3, 0, 1, 2], dtype=int64)</code>
In [13]:	<code>np.argmax(x)</code>
Out [13]:	<code>2</code>
In [14]:	<code>np.argmin(x)</code>
Out [14]:	<code>4</code>

## 04 비교 연산과 데이터 추출

### 3. 인덱스를 활용한 데이터 추출

#### 1. 불린 인덱스

- 불린 인덱스(boolean index) : 배열에 있는 값들을 반환할 특정 조건을 불린형의 배열에 넣어서 추출
  - 인덱스에 들어가는 배열은 불린형이어야 함
  - 불린형 배열과 추출 대상이 되는 배열의 구조가 같아야 함

## 04 비교 연산과 데이터 추출

In [15]:	<code>x = np.array([4, 6, 7, 3, 2])</code> <code>x &gt; 3</code>
Out [15]:	<code>array([ True,  True,  True, False, False])</code>
In [16]:	<code>cond = x &gt; 3</code> <code>x[cond]</code>
Out [16]:	<code>array([4, 6, 7])</code>
In [17]:	<code>x.shape</code>
Out [17]:	<code>(5,)</code>
In [18]:	<code>cond.shape</code>
Out [18]:	<code>(5,)</code>

## 04 비교 연산과 데이터 추출

### 3.2 팬시 인덱스

- 팬시 인덱스(fancy index) : 정수형 배열의 값을 사용하여 해당 정수의 인덱스에 위치한 값을 반환
  - 인덱스 항목에 넣을 배열은 정수로만 구성되어야 함
  - 정수 값의 범위는 대상이 되는 배열이 가지는 인덱스의 범위 내 대상이 되는 배열과 인덱스 배열의 구조(shape)가 같을 필요는 없음

In [19]:	<pre>x = np.array([4, 6, 7, 3, 2]) cond = np.array([1, 2, 0, 2, 2, 2], int) x[cond]</pre>
Out [19]:	<pre>array([6, 7, 4, 7, 7, 7])</pre>

## 04 비교 연산과 데이터 추출

In [20]:	x.take(cond)
Out [20]:	array([6, 7, 4, 7, 7, 7])
In [21]:	x = np.array([[1,4], [9,16]], int) a = np.array([0, 1, 1, 1, 0, 0], int) b = np.array([0, 0, 0, 1, 1, 1], int) x[a,b]
Out [21]:	array([ 1, 9, 9, 16, 4, 4])

	0	1
0	1	4
1	9	16

그림 3-23 팬시 인덱스