Week 5. Python 데이터 분석의 기초 라이브러리 - NumPy

데이터는 수많은 숫자들로 이루어져 있다. 하나의 변수에 여러 개의 데이터를 넣는 방법으로 리스트를 사용할 수도 있지만 `리스트는 속도가 느리고 메모리를 많이 차지하는 단점`이 있다.

더 적은 메모리를 사용해서 빠르게 데이터를 처리하려면 `배열(array)`을 사용해야 한다.

배열은 같은 자료형의 데이터를 정해진 갯수만큼 모아놓은 것이다. 배열은 다음과 같은 점에서 리스트와 다르다.

- 모든 원소가 같은 자료형이어야 한다.
- 원소의 갯수를 바꿀 수 없다.

파이썬은 자체적으로 배열 자료형을 제공하지 않는다. 따라서 배열을 구현한 다른 패키지를 임포트해야 한다. 파이썬에서 배열을 사용하기 위한 표준 패키지는 `NumPy` 이다.

NumPy 기초

`NumPy`는 `np`라는 이름으로 임포트하는 것이 관례이다.

배열 객체 객체는 C언어의 배열처럼 연속적인 메모리 배치를 가지기 때문에 모든 원소가 같은 자료형이어야 한다.

- Numpy(Numerical Python의 줄임말)
- 빠르고 효율적인 다차원 배열 객체 `ndarray`
- 배열 원소를 다루거나 배열 간의 수학 계산을 수행하는 함수
  - 데이터 개조, 정제, 부분 집합, 필터링, 변형
  - 정렬, 유일 원소 찾기, 집합연산 같은 일반적인 배열 처리 알고리즘
- 통계의 효과적인 표현과 데이터의 수집/요약
- 데이터분석에서는 알고리즘에 사용할 데이터 컨테이너의 역할
- NumPy 배열은 파이썬 기본 자료 구조보다 훨씬 효율적인 방법으로 데이터를 저장하고 다룬다.

In [2]:
```
import numpy as np
```

In [3]:
```
a = np.array([0, 1, 2, 3])
print(type(a), a)

print(a.shape)
```

```
<class 'numpy.ndarray'> [0 1 2 3]
(4,)
```

기본 인덱싱

배열 객체로 구현한 다차원 배열의 원소 하나 하나는 다음과 같이 `콤마(comma ,)`를 사용하여 접근할 수 있다. 콤마로 구분된 차원을 `축(axis)`이라고도 한다.

In [26]:
```
a = np.array([[0, 1, 2],
              [3, 4, 5]])
print(a, a.shape)
print()
print(a[0,0]) # 첫번째 행의 첫번째 열
print(a[0,1]) # 첫번째 행의 두번째 열
print(a[-1, -1]) # 마지막 행의 마지막 열
```

```
[[0 1 2]
 [3 4 5]] (2, 3)

0
1
5
```

2차원 배열 - 행렬(matrix)

`ndarray` 는 N-dimensional Array의 약자이다. 이름 그대로 배열 객체는 단순 리스트와 유사한 1차원 배열 이외에도 2차원 배열, 3차원 배열 등의 다차원 배열 자료 구조를 지원

안쪽 리스트의 길이는 행렬의 열의 수 즉, 가로 크기가 되고 바깥쪽 리스트의 길이는 행렬의 행의 수, 즉 세로 크기가 된다. 예를 들어 2 x 3 배열은 다음과 같이 만든다.

In [3]:
```
b = np.array([[0, 1, 2], [3, 4, 5]])  # 2 x 3 array
print(' 행:',len(b), '\n', '열:',len(b[0]),'\n', b)
print('---------------------------')

#배열의 차원 및 크기는 `ndim` 속성과 `shape` 속성으로 알 수 있음
print('차원:', b.ndim)
print('구조:', b.shape)
```

```
 행: 2
 열: 3
 [[0 1 2]
 [3 4 5]]
---------------------------
차원: 2
구조: (2, 3)
```

슬라이싱

배열 객체로 구현한 다차원 배열의 원소 중 복수 개를 접근하려면 일반적인 파이썬 슬라이싱(slicing)과 `comma(,)` 를 함께 사용하면 된다.

> 배열 슬라이스는 값을 복사하는게 아니다. 그러므로 배열 슬라이스의 값을 바꿔도 원본에 반영된다.

뷰 대신에 슬라이스의 복사본을 얻고 싶다면 `copy` 메서드를 이용

In [4]:
```
a = np.array([[0, 1, 2], [3, 4, 5]])
print(a)
print(a[0, :])  # 첫번째 행 전체
print(a[:, 1])  # 두번째 열 전체
print(a[1, 1:])  # 두번째 행의 두번째 열부터 끝열까지

arr_slice_copy = a[1:3].copy() # 무조건 copy() 함수를 써야 복사가 됨
print(arr_slice_copy)
```

```
[[0 1 2]
 [3 4 5]]
[0 1 2]
[1 4]
[4 5]
[[3 4 5]]
```

벡터 연산

`ndarray` 배열 객체는 배열의 각 원소에 대한 연산을 한 번에 처리하는 벡터화 연산(vectorized operation)을 지원

In [6]:
```
# 리스트를 사용하는 경우
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print('리스트에 2를 곱하면?', a * 2)

b = []
for ai in a:
    b.append(ai * 2)
print('실제 리스트를 이용한 a * 2:',b)

# np array 를 사용하는 경우
x = np.array(a)
b = x * 2
print('x * 2:', b)
```

```
리스트에 2를 곱하면? [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
실제 리스트를 이용한 a * 2: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
x * 2: [ 0  2  4  6  8 10 12 14 16 18]
```

In [7]:
```
a = np.array([1, 2, 3])
b = np.array([10, 20, 30])
print(2 * a + b)

#exponential
print('----------------')
print(np.exp(a))

#log
print('------ --------')
print(np.log(a))

#sin, cos
print('--------------')
print(np.sin(a))
```

```
[12 24 36]
----------------
[  2.71828183   7.3890561   20.08553692]
------ --------
[ 0.          0.69314718  1.09861229]
--------------
[ 0.84147098  0.90929743  0.14112001]
```

팬시 인덱싱(fancy indexing)이라고도 부르는 배열 인덱싱(array indexing)

사실은 데이터베이스의 질의(Query) 기능을 수행한다.

불리언(Boolean) 방식 배열 인덱싱

- `True`인 원소만 선택
- 인덱스의 크기가 배열의 크기와 같아야 한다.

위치 지정 방식 배열 인덱싱

- 지정된 위치의 원소만 선택
- 인덱스의 크기가 배열의 크기와 달라도 된다.

In [5]:
```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
idx = np.array([True, False, True, False, True, False, True, False, True, False])
print('짝수를 가져와!:',a[idx])

bool_idx = (a % 2 == 0)
print('짝수를 가져와!:',a[bool_idx])

print('짝수 - 더 간단하게:',a[a % 3 == 0])

idx = np.array([0, 2, 4, 6, 8])
print('홀수번째 component!:',a[idx])
```

```
짝수를 가져와!: [0 2 4 6 8]
짝수를 가져와!: [0 2 4 6 8]
짝수 - 더 간단하게: [0 3 6 9]
홀수번째 component!: [0 2 4 6 8]
```

배열연산 표현하기 (+조건절)

In [41]:
```
arr = np.array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

print(arr)
print('------------------')

#조건절
print(np.where( arr > 3, 1, -1))
# 양수는 모두 2로, 음수는 모두 -1로 변경
print('------------------')

# Cumulative sum
print(arr.cumsum())
# column cumulative
print(arr.cumsum(0))
# row cumulative
print(arr.cumsum(1))

print('------------------')

# Cumulative product
print(arr.cumprod(1))
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
------------------
[[-1 -1 -1]
 [-1  1  1]
 [ 1  1  1]]
------------------
[ 0  1  3  6 10 15 21 28 36]
[[ 0  1  2]
 [ 3  5  7]
 [ 9 12 15]]
[[ 0  1  3]
 [ 3  7 12]
 [ 6 13 21]]
------------------
[[  0   0   0]
 [  3  12  60]
 [  6  42 336]]
```

NumPy 배열 초기화

배열 생성

앞에서 파이썬 리스트를 NumPy의 `ndarray` 객체로 변환하여 생성하려면 array 명령을 사용하였다. 그러나 보통은 이러한 기본 객체없이 다음과 같은 명령을 사용하여 바로 `ndarray` 객체를 생성한다.

- `zeros, ones` : 크기가 정해져 있고 모든 값이 0 (ones 인 경우 1) 인 배열
- `zeros_like, ones_like` : 특정한 배열 혹은 리스트와 같은 크기의 배열을 생성
- `empty` : 생성만 하고 초기화를 하지 않는 empty 명령
- `arange` : NumPy 버전의 range 명령. 특정한 규칙에 따라 증가하는 수열
- `linspace, logspace` : 선형 구간 혹은 로그 구간을 지정한 구간의 수만큼 분할
- `rand(uniform dist.), randn(gaussian dist.)` : 임의의 난수를 생성.시드(seed)값을 지정하려면 `seed`

In [57]:
```
# zeros, ones
print(np.zeros(5))
print(np.ones(5))
print(np.zeros((5, 2), dtype="f8"))
print('----------------------')
```

```
[ 0.  0.  0.  0.  0.]
[ 1.  1.  1.  1.  1.]
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
----------------------
```

In [60]:
```
# zeros_like, ones_like
e = range(10)
print(list(e))
print(np.ones_like(e, dtype="int"))
print('----------------------')
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1 1 1 1 1 1 1 1 1 1]
----------------------
```

In [61]:
```
# empty
print(np.empty((4,3)))
print('----------------------')
```

```
[[  1.50690911e-312   0.00000000e+000   8.76794447e+252]
 [  2.15895723e+227   1.14428494e+243   1.35617218e+248]
 [  2.78225511e+296   9.80058441e+252   1.23971686e+224]
 [  1.41066534e+232   1.16070543e-028   4.05919345e-317]]
----------------------
```

In [62]:
```
# arange
print(np.array(range(10)))
print(np.arange(10))   # 0 .. n-1
print(np.arange(3, 21, 2))  # 시작, 끝(포함하지 않음), 단계
print('----------------------')
```

```
[0 1 2 3 4 5 6 7 8 9]
[0 1 2 3 4 5 6 7 8 9]
[ 3  5  7  9 11 13 15 17 19]
----------------------
```

In [22]:
```
# linspace
print(np.linspace(0, 100, 5))  # 시작, 끝(포함), 갯수
print('----------------------')
```

```
[   0.   25.   50.   75.  100.]
----------------------
```

In [80]:
```
# rand, randn
np.random.seed(0)

print(np.random.rand(4))
print(np.random.random(4))
print(np.random.randn(4))
```

```
[ 0.5488135   0.71518937  0.60276338  0.54488318]
[ 0.4236548   0.64589411  0.43758721  0.891773  ]
[-0.10321885  0.4105985   0.14404357  1.45427351]
```

선형대수 연산

대각합 (trace)

- tr(kA) = k * tr(A)
- tr(A + B) = tr(A) + tr(B)
- tr(A') = tr(A)
- tr(AB) = tr(BA)

In [82]:
```
A = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])

print(np.trace(A))
```

```
15
```

행렬식 (deteminant)

- |A'| = |A|
- |AB| = |A||B|
- 대각행렬 A 이면, |A| = a11a22 ... amm 이다. (대각의 곱)
- 한 행이라도 모두 0 이면 |A| = 0
- 두 행 또는 두 열이 동등하면 |A| = 0

In [83]:
```
print(np.linalg.det(A))
```

```
6.66133814775e-16
```

계수 (rank) 와 비특이행렬 (nonsingular matrix)

- 만약 |S| = 0 이면,
- S 는 특이행렬(singular matrix) 이고,
- |S| != 0 이면 비특이행렬 (nonsingular matrix) 이라고 한다.

In [84]:
```
S = np.array([[1,2],[2,4]])

print (np.linalg.matrix_rank(S)) # rank 계수가 1 != 2
print(np.linalg.det(S)) # 행렬식 = 0 => 특이행렬이다.
```

```
1
0.0
```

선형종속 (linear dependence)

- n 개의 벡터{x1, x2, .. xn} 과 n 개의 스칼라 {k1, k2, .. kn}에서 k1x1 + k2x2 + ... + knxn = 0 을 충족하는 0이 아닌 스칼라 집합이 있다면 선형종속이다.
- 쉽게 말해, 어떤 벡터 xi 는 다른 벡터의 조합으로 나타낼 수 있을 경우. 그렇지 않은 경우를 선형독립 (linear independence) 라고 한다.

In [85]:
```
# determinant로 검증
# 선형종속 => det(X) = 0 => 특이행렬 singular matrix
round(np.linalg.det(X))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-85-709e139af061> in <module>()
      1 # determinant로 검증
      2 # 선형종속 => det(X) = 0 => 특이행렬 singular matrix
----> 3 round(np.linalg.det(X))

NameError: name 'X' is not defined
```

In [86]:
```
#### 역행렬 (inverse matrix)
A = np.array([[1,2],[3,4]])
print(A)

inv_A = np.linalg.inv(A)
print (inv_A)
print('-------------------------')

##선형등식체계 해
A = np.array([[3,5],
              [4,2]])
h = np.array([[13],[8]])

np.linalg.det(A)                # 역행렬 존재여부 확인
inv_A = np.linalg.inv(A)        # 역행렬
print(np.dot(inv_A, h))         # x = inv(A) * h
```

```
[[1 2]
 [3 4]]
[[-2.   1. ]
 [ 1.5 -0.5]]
-------------------------
[[ 1.]
 [ 2.]]
```

고유값과 고유행렬

A 라는 행렬을 일종의 변환행렬이라고 생각할때, 그 변환에 대해서 (크기만 변화) 방향이 변하지 않는 벡터이기 때문에 고유벡터라고 한다.

Ax = Lx

Ax - Lx = 0

(A - LI)x = 0 # 스칼라값 L 에 항등행렬 I 를 곱해 행렬뺄셈을 한다.

여기서 (A - LI) 행렬이 비특이행렬(역행렬 존재)하면, x = 0 이라는 자명해(trivial solution) 만 존재하고, 고유벡터는 없다.

(A - LI) 가 특이행렬(역행렬X) 이면, x = 0 이외의 비자명해(nontrivial solution) 을 가지게 된다.

(A - LI) 의 행렬식 deteminant 값이 0 이 되도록 하는 L 값을 계산해서 나오는 값이 고유값이다.

In [6]:
```
A = np.array([[1,2],[3,4]])

v, m = np.linalg.eig(A)
print(v)
print(m)
```

```
[-0.37228132  5.37228132]
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

벡터 및 행렬 norm

In [89]:
```
x = np.array([0, 1, 2, 3])
print(x)
print(np.linalg.norm(x))
```

```
[0 1 2 3]
3.74165738677
```

통계 - 최대최소

- 최대/최소: `min` , `max` , `argmin` , `argmax`
- 통계: `sum` , `mean` , `median` , `std` , `var`
- 불리언: `all` , `any`

In [10]:
```
x = np.array([1, 2, 3, 4])
y = np.array([[1, 1], [2, 2]])

# sum
print(np.sum(x), x.sum())
print('y 열합계:', y.sum(axis=0))  # 열 합계
print('y 행합계:', y.sum(axis=1))  # 행 합계
print('-----------------------')

#min, max
print(x.min(), x.max())
print('-----------------------')

# argmin
print(x.argmin(), x.argmax())
print('-----------------------')

# mean, median
print(x.mean(), np.median(x))
```

```
10 10
y 열합계: [3 3]
y 행합계: [2 4]
-----------------------
1 4
-----------------------
0 3
-----------------------
2.5 2.5
```

확률변수(random variables) 집합의 공분산행렬 (covariance matrix)

In [94]:
```
import numpy as np
# 두 개의 변수 , x_0 과 x_1이 완전히 상관관계가 있고, 방향은 반대라고 하자(부적상관)
x_0 = np.array([0, 1, 2])
x_1 = np.array([2, 1, 0])
X = np.vstack((x_0, x_1))

print(X)
print(np.cov(X))
print(np.cov(x_0, x_1))
```

```
[[0 1 2]
 [2 1 0]]
[[ 1. -1.]
 [-1.  1.]]
[[ 1. -1.]
 [-1.  1.]]
```

집합

In [25]:
```
# 중복 제거
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
print(names)

print(np.unique(names))
```

```
['Bob' 'Joe' 'Will' 'Bob' 'Will' 'Joe' 'Joe']
['Bob' 'Joe' 'Will']
```

In [28]:
```
# 2개의 배열을 인자로 받아 첫 번째 배열의 각 원소가 두 번째 배열의
# 원소를 포함하는지를 나타내는 불리언 배열을 반환
values = np.array([6, 0, 0, 3, 2, 5, 6])
print(values)

print(np.in1d(values, [2, 3, 6]))
```

```
[6 0 0 3 2 5 6]
[ True False False  True  True False  True]
```

배열 수정

배열 변형

- `reshape` : 형태만 바꾸기
- `transpose` : 전치
- `flatten` : 무조건 1차원 배열로 펼침
- `newaxis` : 차원 늘리기(1차원 증가)

In [54]:
```
# reshape
a = np.arange(12)
print(a)
print('--------------------')

b = a.reshape(3, 4)
print(b)
print('--------------------')

# transpose
print(b.transpose())
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
--------------------
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
--------------------
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

사용하는 원소의 갯수가 정해저 있기 때문에 reshape 명령의 형태 튜플의 원소 중 하나는 -1이라는 숫자로 대체할 수 있다. -1을 넣으면 해당 숫자는 다른 값에서 계산되어 사용된다.

In [42]:
```
print(a.reshape(2,2,-1)) # 2 * 2 * -1(3) 행렬

print('--------------------') # 2 * -1(3) * 2 행렬
print(a.reshape(2,-1,2))
```

```
[[[ 0  1  2]
  [ 3  4  5]]

 [[ 6  7  8]
  [ 9 10 11]]]
--------------------
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]
```

In [52]:
```
# flatten
c = b.flatten()
print(c)
print('--------------------')

# newaxis
x = np.arange(5)
print(x[:, np.newaxis])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
--------------------
[[0]
 [1]
 [2]
 [3]
 [4]]
```

배열 연결/분리

행의 수나 열의 수가 같은 두 개 이상의 배열을 연결(`concatenate`)하여 더 큰 배열을 만들 때는 다음과 같은 명령을 사용한다.

- `hstack` : 행의 수가 같은 두 개 이상의 배열을 옆으로 연결하여 열의 수가 더 많은 배열합친다.
- `vstack` : 열의 수가 같은 두 개 이상의 배열을 위아래로 연결하여 행의 수가 더 많은 배열을 만든다.
- `dstack` : 제3의 축 즉, 행이나 열이 아닌 깊이(depth) 방향으로 배열을 합친다.
- `stack` : 새로운 차원(축으로) 배열을 연결. axis 인수(디폴트 0)를 사용하여 연결후의 회전 방향을 정한다. 연결하고자 하는 배열들의 크기가 모두 같아야 한다.
- `r_`
- `c_`
- `tile` : 동일한 배열을 반복하여 연결
- `split` : 배열을 분리

In [3]:
```
a1 = np.ones((2, 3))
a2 = np.zeros((2, 3))

# hstack
print(np.hstack([a1, a2]))
print('----------------')

# vstack
print(np.vstack([a1, a2]))
print('----------------')
```

```
[[ 1.  1.  1.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.]]
----------------
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
----------------
```

In [4]:
```
# dstack
print(np.dstack([a1, a2]))
print('----------------')

# stack
print(np.stack([a1, a2], axis=1))
print('----------------')
```

```
[[[ 1.  0.]
  [ 1.  0.]
  [ 1.  0.]]

 [[ 1.  0.]
  [ 1.  0.]
  [ 1.  0.]]]
----------------
[[[ 1.  1.  1.]
  [ 0.  0.  0.]]

 [[ 1.  1.  1.]
  [ 0.  0.  0.]]]
----------------
```

In [8]:
```
# tile
b1 = np.array([1,2,3])
print(np.tile(b1, 3))
print(np.tile(b1, (2,3)))
print('----------------')

#split
arr = np.random.randn(5, 5)

first, second, third = np.split(arr, [1, 3], axis=1)
print(first)
print(second)
print(third)
```

```
[1 2 3 1 2 3 1 2 3]
[[1 2 3 1 2 3 1 2 3]
 [1 2 3 1 2 3 1 2 3]]
----------------
[[ 1.93076351]
 [ 0.81512242]
 [ 1.18248718]
 [-0.674037  ]
 [ 2.54432997]]
[[-0.79106612  0.46377816]
 [ 0.73086931 -1.58905706]
 [ 0.81096935 -0.12806666]
 [ 2.24701399 -0.70540449]
 [ 0.34714922 -1.06184133]]
[[-1.49795446 -2.70362831]
 [-0.68646859  0.05736132]
 [ 0.87660029 -0.36260088]
 [ 0.29207287  0.22563171]
 [-1.14705908 -0.14950814]]
```

그리드 생성

함수의 그래프를 그리거나 표를 작성하려면 많은 좌표를 한꺼번에 생성하여 각 좌표에 대한 함수 값을 계산

x, y 라는 두 변수를 가진 함수에서 x가 0부터 2까지, y가 0부터 4까지의 사각형 영역에서 변화하는 과정을 보고 싶다면 이 사각형 영역 안의 다음과 같은 (x,y) 쌍 값들에 대해 함수를 계산해야 한다.

```
(x,y)=(0,0),(0,1),(0,2),(0,3),(0,4),(1,0),⋯,(2,4)
```

이러한 과정을 자동으로 해주는 것이 NumPy의 `meshgrid` 명령이다. `meshgrid` 명령은 사각형 영역을 구성하는 가로축의 점들과 세로축의 점을 나타내는 두 벡터를 인수로 받아서 이 사각형 영역을 이루는 조합을 출력한다. 단 조합이 된 (x,y)쌍을 x값만을 표시하는 행렬과 y값만을 표시하는 행렬 두 개로 분리하여 출력한다.

In [ ]:
```
x = np.arange(3)
y = np.arange(5)

X, Y = np.meshgrid(x, y)
[zip(x, y) for x, y in zip(X, Y)]

import matplotlib.pylab as plt
plt.scatter(X,Y, linewidth= 10);
plt.show()
```

정렬

`sort` 명령이나 메서드를 사용하여 배열 안의 원소를 크기에 따라 정렬하여 새로운 배열을 만들 수도 있다.

2차원 이상인 경우에는 마찬가지로 `axis` 인수를 사용하여 방향을 결정한다.

In [69]:
```
a = np.array([[4, 3, 5], [1, 2, 1]])
print(a)
np.sort(a)
print(a)
```

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

계산!

NumPy는 코드를 간단하게 만들고 계산 속도를 빠르게 하기 위한 벡터화 연산(vectorized operation)을 지원한다. 벡터화 연산이란 반복문(`loop`)을 사용하지 않고 선형 대수의 벡터 혹은 행렬 연산과 유사한 코드를 사용하는 것을 말한다.

In [95]:
```
# 리스트로 연산
import numpy as np
x = np.arange(1, 100001)
y = np.arange(100001, 200001)
```

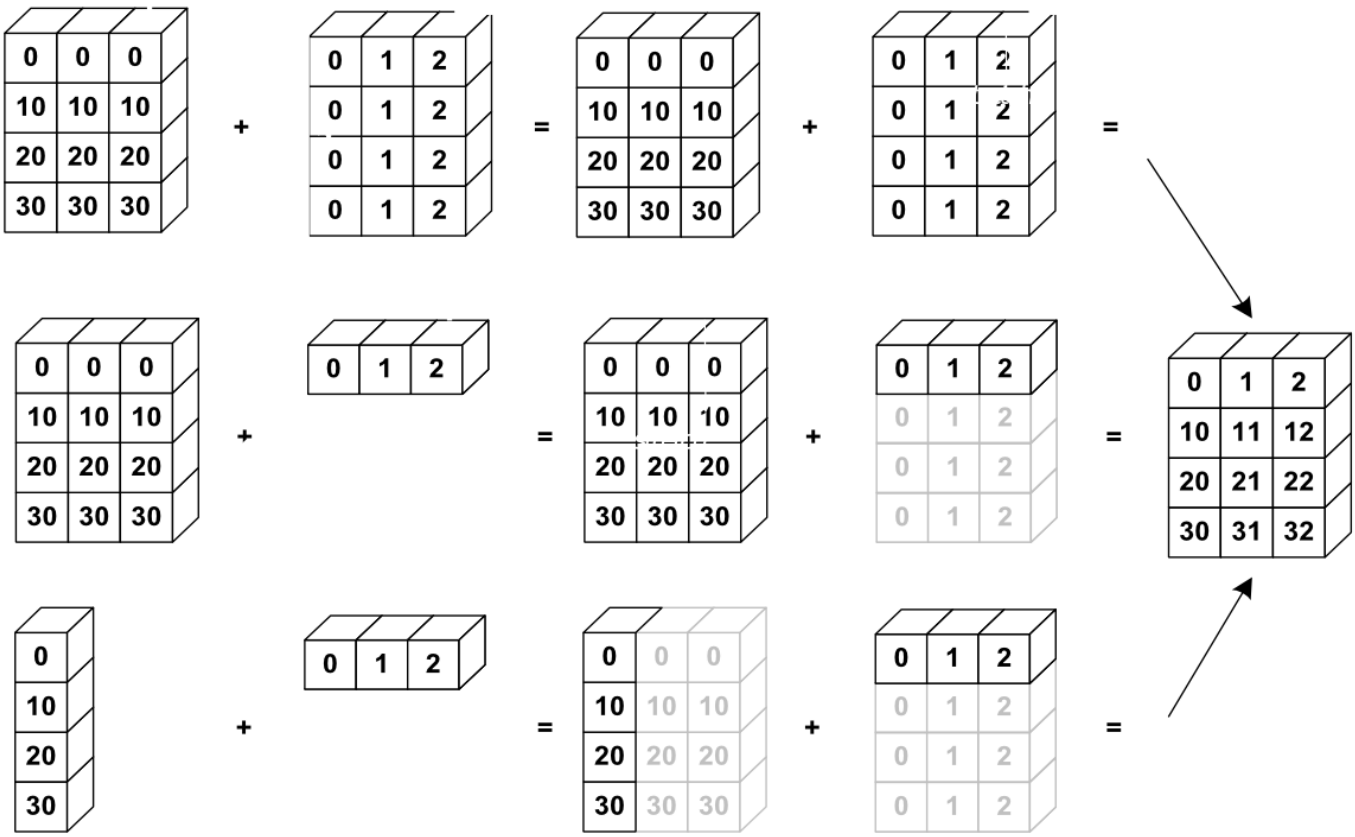
In [96]:
```
%%timeit
# timeit 의 경우 본 블록의 연산 속도를 측정 할 수 있음
a = np.zeros_like(x)
for i in range(1000):
    a[i] = x[i] + y[i]
```

```
1000 loops, best of 3: 398 µs per loop
```

In [97]:
```
%%timeit
a = x + y
```

```
The slowest run took 6.39 times longer than the fastest. This could mean that an i
ntermediate result is being cached.
10000 loops, best of 3: 65.6 µs per loop
```

브로드캐스팅



In [55]:
```
a = np.tile(np.arange(0, 40, 10), (3, 1)).transpose()
print(a)
print('---------------------')

b = np.array([0, 1, 2])
print(b)
print('---------------------')

# 방법 1
print('a+b:', a + b)
print('---------------------')

# 방법 2
a = np.arange(0, 40, 10)[:, np.newaxis]
print('a+b:', a + b)
```

```
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
---------------------
[0 1 2]
---------------------
a+b: [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
---------------------
a+b: [[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

nparray 읽기/쓰기

In [50]:
```
array = ([[ 0.580052,  0.18673 ,  1.04717 ,  1.13441 ],
       [ 0.194163, -0.636917, -0.938659,  0.124094],
       [-0.12641 ,  0.268607, -0.695724,  0.047428]])

# 쓰기
np.savetxt('data/array_save_ex.txt', array)

# 읽기
arr = np.loadtxt('data/array_save_ex.txt', delimiter=' ')
print(arr)
```

```
[[ 0.580052  0.18673   1.04717   1.13441 ]
 [ 0.194163 -0.636917 -0.938659  0.124094]
 [-0.12641   0.268607 -0.695724  0.047428]]
```