

기초 인공지능 프로그래밍

9장 Numpy 라이브러리 (데이터 분석)

외부 라이브러리(패키지) 설치

- PIP를 통한 패키지 설치
 - PIP(Pip Installs Packages) : 파이썬 패키지 관리 프로그램
 - 명령어 pip list 로 설치되어 있는 패키지를 확인할 수 있음
 - 미설치 패키지는 다음과 같이 명령 프롬프트에서 설치할 수 있음

```
C:\W>python -m pip install --upgrade pip # pip 프로그램을 최신 버전으로 업그레이드
```

```
C:\W>pip list # 현재 파이썬에 설치된 패키지 확인
```

Package	Version
---------	---------

pip	23.1.2
setuptools	49.2.1

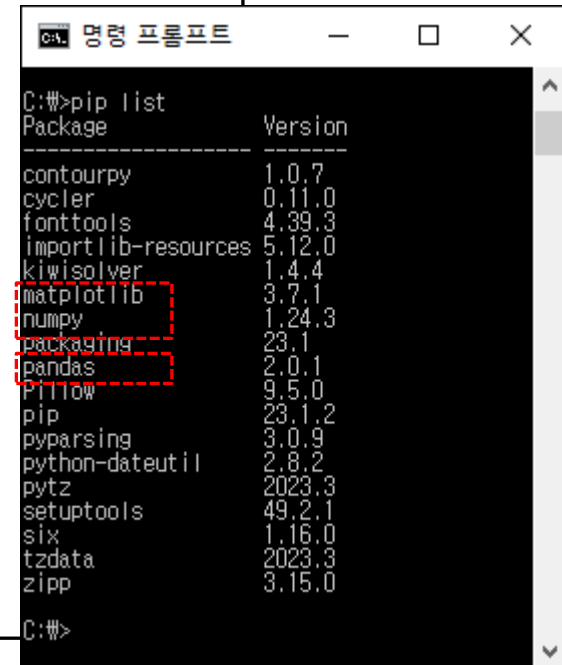
```
C:\W>pip install numpy # 패키지 설치 또는 삭제(install 대신 uninstall)
```

```
C:\W>pip install pandas
```

```
C:\W>pip list # 현재 파이썬에 설치된 패키지 확인
```

Package	Version
---------	---------

numpy	1.16.4
pandas	0.25.0
pip	19.1.1
python-dateutil	2.8.0
pytz	2019.1
setuptools	39.0.1
six	1.12.0



```
C:\W>pip list
Package            Version
-----
contourpy          1.0.7
cycler              0.11.0
fonttools          4.39.3
importlib-resources 5.12.0
kiwisolver          1.4.4
matplotlib          3.7.1
numpy               1.24.3
packaging           23.1
pandas              2.0.1
Pillow              9.5.0
pip                 23.1.2
pyparsing           3.0.9
python-dateutil     2.8.2
pytz                2023.3
setuptools          49.2.1
six                 1.16.0
tzdata              2023.3
zipp                 3.15.0
C:\W>
```

numpy (numerical python)

- 파이썬에서 대규모 다차원 배열을 다룰 수 있게 도와주는 라이브러리
 - 대규모 데이터를 리스트 자료형에 저장/관리하면 속도가 느리고 메모리도 많이 차지하는 단점이 있음
 - 같은 자료형(대부분 숫자)의 데이터들을 저장하는 배열(array)을 사용하면 효율적인 메모리 사용으로 데이터를 빠르게 처리할 수 있음
- 파이썬은 기본 자료형으로 배열을 지원하지 않기 때문에 배열 라이브러리 numpy를 설치한 후, import 해서 사용해야 함

numpy (numerical python)

- numpy(넘파이, 넘피)
 - 다차원의 배열 자료구조 클래스인 ndarray 객체를 지원
 - ndarray 객체는 numpy를 통해 생성되는 n 차원 배열 객체
 - 벡터화 연산(vectorized operation)을 지원
 - : 배열의 각 원소에 대한 반복 연산을 하나의 명령어로 처리
 - 랜덤 시뮬레이션, 통계학 연산, 선형대수학 등 다양한 수학적 연산이 가능
 - 배열 인덱싱(array indexing)을 사용한 질의(Query) 기능으로 복잡한 수식을 계산 가능하게 함

numpy (numerical python)

▪ numpy vs 리스트

- numpy 배열은 생성될 때 크기를 결정
- 데이터를 추가하면 새로운 numpy 배열이 생성
- 배열의 모든 원소들은 같은 자료형(대부분 숫자)이어야 함
- 배열은 수학에 관련된 많은 연산을 지원
- 리스트 자료형 연산보다 더 간단하게 해결할 수 있음

▪ a, b가 정수 n개를 원소로 하는 리스트 일 때

- a와 b의 같은 인덱스 위치의 원소들을 서로 합한 결과를 새로운 리스트에 저장하기 위해서는 반복문이 필요

```
c = []  
for i in range(len(a)) :  
    c.append(a[i] + b[i])
```

▪ a, b가 정수 n개를 원소로 하는 배열 일 때

- a와 b의 같은 인덱스 위치의 원소들을 서로 합한 결과를 새로운 배열에 저장하기 위해서는 간단한 연산으로 해결 가능

```
c = a + b
```

numpy

- numpy 라이브러리를 사용하기 위해서는 import 해야함
 - `import numpy as np` # 편리를 위해 별칭 np 지정 (관용적 사용)
- 1차원(axis) 배열
 - 수치 데이터 중심으로 구성, 데이터가 하나의 줄로 나열
 - 레이블(이름)이 없음
- 2차원(axis) 배열 : 행렬(matrix)
 - 1차원 배열을 여러 줄로 겹쳐 놓은 형태
 - 가로줄은 행(row), 세로줄은 열(column) 이라고 함
 - 행과 열에 레이블(이름)이 없음
- ndarray 객체 (배열)의 속성
 - ndim : 배열의 차원
 - size : 전체 원소의 개수
 - dtype : 원소의 자료형
 - itemsize : 원소의 크기(byte)
 - shape : 배열의 형상(튜플로 표시)

ndarray 객체는 배열 데이터, 메소드(함수), 속성 등을 저장하고 있으며 **객체이름.속성**, **객체이름.메소드** 형식으로 사용할 수 있음

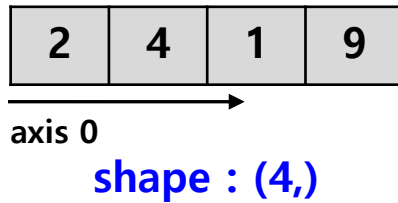
```
1 import numpy as np
2 a = np.array([1,2,3,4,5])
3 print(a, type(a))
4 print(a.ndim)
5 print(a.size)
6 print(a.dtype)
7 print(a.itemsize)
8 print(a.shape)
```

[1 2 3 4 5] <class 'numpy.ndarray'>
1
5
int64
8
(5,)

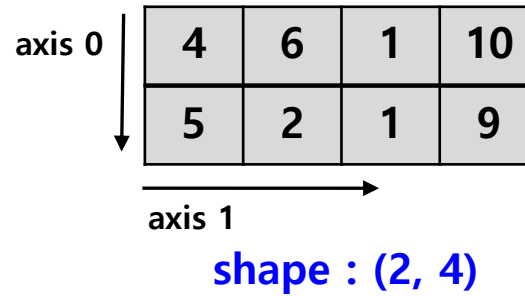
numpy 배열 생성

- numpy 배열

1D array (vector)



2D array (matrix)



numpy 배열 생성

- 1차원 배열 생성 방법 **numpy 라이브러리 함수 사용**

- ① array() 함수의 인수로 리스트 자료형을 지정하여 ndarray 객체(배열) 생성

```
1 import numpy as np
2 a = np.array([1,2,3,4,5])
3 print(a, type(a))
4 print(a.ndim)
5 print(a.size)
6 print(a.dtype)
7 print(a.shape)
```

```
[1 2 3 4 5] <class 'numpy.ndarray'>
1
5
int64
(5,)
```

array() 함수가 1차원 배열을 생성하여 반환.
변수 a는 ndarray 타입의 객체를 가리키게 됨.
a = np.array(range(1,6,1)) 와 동일.

- ② arange(start, end, step) 함수로 특정한 범위의 숫자(정수, 실수)를 원소로 하는 배열 생성 (함수 사용법은 range() 함수와 유사함)

```
1 import numpy as np
2 a = np.arange(1, 21)    #a = np.array(range(1, 21))과 동일
3 print(a, type(a))
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20] <class 'numpy.ndarray'>
```


numpy 배열 생성

- 2차원 배열 생성
 - array() 함수의 인수로 두개의 리스트를 원소로 가지는 리스트를 지정하여 배열 생성
 - 행렬(matrix)라고도 함

```
1 import numpy as np
2 b = np.array([[1,2,3],[4,5,6]])
3 print(b)
4 print(b.ndim)
5 print(b.size)
6 print(b.dtype)
7 print(b.shape)
8 print(b[0], b[1])
```

[[1 2 3]
[4 5 6]]
2
6
int64
(2, 3) **b는 2행 3열 배열**
[1 2 3] [4 5 6]

b[0]는 배열의 첫번째 행
b[1]는 배열의 두번째 행

$$\begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \end{bmatrix}$$

b[1][1]
또는
b[1,1]

배열 속성 shape 값이 (5,)은 1차원 배열의 원소 수가 5개라는 의미이며,
shape 값이 (1, 5) 또는 (5, 1)은 2차원 배열을 의미함.

numpy 배열 생성

- 배열 생성시 원소 자료형 지정 방법
 - astype() 메소드로 float에서 int로 자료형 변환시 소수점 이하는 버림하며 -0.9와 0.4는 0으로 변환(-1보다 크고 1보다 작은 실수는 0으로 변환)

```
1 import numpy as np
2 a = np.array([1,2,3,4,5])
3 print(a)
4 b = np.array([1,2,3,4,5], dtype = "float")
5 print(b)
```

[1 2 3 4 5]
[1. 2. 3. 4. 5.]

생성할 때 원소의 자료형을 지정할 수 있음

```
1 import numpy as np
2 a = np.array([1.0, 2., 3, 4, 5])
3 print(a)
4 print(a.dtype)
5 a_int = a.astype(int)
6 print(a_int)
7 print(a_int.dtype)
```

[1. 2. 3. 4. 5.]
float64
[1 2 3 4 5]
int64

astype() 메소드로 int로 변환
변환하면 새로운 ndarray 객체가 생성

numpy 배열 생성

- numpy 라이브러리 함수로 여러 종류의 배열 생성
 - `zeros((n, m))`, `zeros(n)`
: 모든 원소 값이 0인 $n \times m$ 배열 생성
 - `ones((n, m))`, `ones(n)`
: 모든 원소 값이 1인 $n \times m$ 배열 생성
 - `full((n, m), value)`, `full(n, value)`
: 모든 원소 값이 value인 $n \times m$ 배열 생성
 - `eye(n)`
: 대각선의 원소 값은 1, 나머지는 0의 값을 갖는 행과 열이 같은 $n \times n$ 배열 생성
 - `linspace(x1, x2, n)`
: 구간 $[x1, x2]$ 내에서 균일한 간격의 데이터를 n 개 만들어 배열 생성 ($x1$ 과 $x2$ 포함), n 값이 주어지지 않으면 디폴트로 50개 생성
 - `a.reshape((n, m))`
: 기존 배열 a 를 $n \times m$ 배열 형태로 변형하는 함수
 - `a.flatten()`
: 기존 2차원 이상의 배열 a 를 1차원 배열 형태로 변형하는 함수

numpy 배열 생성

- numpy 라이브러리 함수로 배열 생성한 예시

```
1 import numpy as np
2 a_zero = np.zeros((2,3)); b_zero = np.zeros(5, dtype = int)
3 print(a_zero)
4 print(b_zero)
5 a_full = np.full((2,2), 10); b_full = np.full(6, 2.5)
6 print(a_full)
7 print(b_full)
8 c = np.arange(1, 10).reshape((3,3))
9 print(c)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
[0 0 0 0 0]
[[10 10]
 [10 10]]
[2.5 2.5 2.5 2.5 2.5]
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
1 import numpy as np
2 a = np.random.random((2,2))
3 print(a)
4 b = np.random.normal(0, 1, (2,2))
5 print(b)
6 c = np.random.randint(0, 10, (2,2))
7 print(c)
```

```
[[0.06494161 0.76755055]
 [0.77146756 0.32009013]]
[[-0.23810466 0.3563168 ]
 [-1.08030284 0.92160452]]
[[3 4]
 [6 9]]
```

np.random.random(size = k) : 0.0에서 1.0 미만의 난수 k개 원소로 하는 배열 생성. k값이 주어지지 않으면 1개 생성.

k 값이 튜플 형태(예를 들면 (2,2))이면 2차원 배열 생성

np.random.normal(n, m, size = k) : 평균 0, 표준편차 1의 표준 정규 분포의 난수 k개를 원소로 하는 배열 생성.

np.random.randint(n, m, size = k) : 정수 n부터 정수 m-1까지의 범위에서 난수 k개 원소로 하는 배열 생성

numpy 배열 인덱싱

- 1차원 배열 인덱싱
 - 리스트의 인덱싱과 동일하게 사용
 - 1차원 배열 a1의 2번째 원소 참조는 a1[1] 형태
- 2차원 배열 인덱싱
 - 리스트의 인덱싱과 동일하게 사용
 - 2차원 배열 a2의 3번째 행 참조는 a[2] 형태
 - 2차원 배열 a2의 2번째 행 3번째 원소 참조는 a[1, 2] 형태

```
1 import numpy as np
2 a1 = np.array([10, 20, 30])
3 a2 = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
4 a1[1] = 200
5 row = a2[2]
6 n = a2[1, 2]  n = a2[1][2] 와 동일
7 print(a1)
8 print(row)
9 print(n)
```

```
[ 10 200  30]
[ 7  8  9]
6
```

numpy 배열 슬라이싱

- 1차원 배열 슬라이싱
 - 1차원 배열 a에 대해 슬라이싱 형식
 - 인덱스 i, j, k 위치의 원소들로 부분 배열을 생성할 경우 : a[[i, j, k]]
 - 인덱스 i 부터 j까지 연속된 원소들로 생성할 경우 : a[i : j+1]

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4, 5])
3 n1 = a[[1]] # 배열 a의 a[1] 원소로 배열 생성
4 n2 = a[[1,3]] # 배열 a의 a[1], a[3] 원소로 배열 생성
5 n3 = a[1:4] # 배열 a의 인덱스 1부터 3까지의 원소로 배열 생성
6 print(n1, n2, n3)
```

[2] [2 4] [2 3 4]

**a[1]은 배열의 원소 참조하는 형식이며,
a[[1]]와 다름.**

numpy 배열 슬라이싱

- 2차원 배열 슬라이싱
 - 배열의 각 차원 별로 슬라이싱 범위를 지정

```
1 import numpy as np
2 a = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 a1 = a[0:2, 1:3]
4 print(a1)
5 a2 = a[1:] # a2 = a[1:, :]; a2 = a[1:, 0:] 와 동일
6 print(a2)
```

행 범위 0:2, 열 범위 1:3 즉, 첫번째와 두번째 행에서 두번째부터 세번째 열 범위

행 범위만 지정

```
[[2 3]
 [5 6]
 [4 5 6]
 [7 8 9]]
```

```
1 import numpy as np
2 a = np.array([[0,1,2,3],[10,11,12,13],[20,21,22,23]])
3 a1 = a[::2, ::2]
4 a2 = a[::2][::2]
5 print(a1)
6 print(a2)
7 print(a2.ndim)
8 print(a2.shape)
```

a1은 첫번째, 세번째 행에서 첫번째, 세번째 열 범위 슬라이싱

a2는 첫번째, 세번째 행 슬라이싱한 2차원 배열에서 다시 첫번째 행 슬라이싱

```
[[ 0  2]
 [20 22]]
[[0 1 2 3]]
2
(1, 4)
```

a2는 1 X 4 형태의 2차원 배열

numpy 배열 슬라이싱

- 2차원 배열 슬라이싱
 - 정수 인덱싱 (integer indexing) : Fancy indexing

```
1 import numpy as np
2 a = np.array([10, 20, 30, 40, 50])
3 idx = [1, 3]
4 a1 = a[idx]
5 print(a1)
6 idx1 = np.array([[1, 3], [2,1]])
7 a2 = a[idx1]
8 print(a2)
```

인덱스 정보를 2차원 배열 형태로 생성한 후,
1차원 배열로 2차원 배열 생성시 사용

```
[20 40]
[[20 40]
 [30 20]]
```

- 2차원 배열 a의 a[r1, c1], a[r2, c2],... 원소들로 부분 배열을 생성시 :
a[[r1, r2,...], [c1, c2,...]] 형식

```
1 import numpy as np
2 a = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 a1 = a[[0,2], [1, 2]]
4 print(a1)
```

a[0,1]와 a[2,2] 원소로 부분 배열을 생성

```
[2 9]
```


numpy 배열 데이터 추출

- 1차원 배열 배열에서 조건에 맞는 데이터 추출
 - 부울형(Boolean) 배열(원소 값이 True 또는 False인 배열)을 인덱스로 사용하여 특정 조건의 원소들만 필터링할 때 사용
 - 배열의 관계연산으로 부울형 배열 생성(추출하고자 하는 배열과 크기가 같아야 함)
 - 부울형 배열의 원소 값 True와 대응되는 원소만 추출

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4, 5])
3 b = a % 2 == 0
4 even = a[b]
5 print(b)
6 print(even)
```

[False True False True False]
[2 4]

부울형 배열 생성.
배열 a의 원소가 짝수이면 True,
아니면 False.

부울형 배열을 인덱스로 사용하여 a 원소 중
짝수 값만 추출하여 1차원 배열로 반환

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4, 5])
3 even = a[a % 2 == 0]
4 print(even)
```

[2 4]

numpy 배열 생성

- 1차원 배열에서 조건에 맞는 데이터 추출
 - 리스트 데이터로 ndarray 객체 생성

```
1 import numpy as np
2 scores = [23, 89, 45, 60, 99, 51]
3 scores_a = np.array(scores)
4 # 해당 원소가 조건을 만족하면 True, 아니면 False 값을 가지는 부울형 배열 생성
5 over_50 = scores_a > 50
6 print(over_50)
7 scores_over50 = scores_a[over_50] # 조건을 만족하는 원소만 추출한 배열 생성
8 print(scores_over50)
```

[False True False True True True]
[89 60 99 51]

부울형 배열을 인덱스로 사용하여 True 값에 해당하는 위치의 원소만 추출함

score_a 배열의 각 원소가 조건을 만족하면 True, 아니면 False 값이 저장되는 같은 크기의 부울형 배열(원소 값이 True 또는 False인 배열)이 생성됨

```
import numpy as np
scores_a = np.array([23, 89, 45, 60, 99, 51])
scores_over50 = scores_a[scores_a > 50]
print(scores_over50)
```

numpy 배열 슬라이싱

- 2차원 배열 배열에서 조건에 맞는 데이터 추출
 - $n \times m$ 배열의 각 원소의 선택 여부를 위해 같은 크기의 부울형 배열을 만듦
 - 만약 배열 a 가 2×3 의 배열이라면, 부울형 배열도 2×3 으로 만들어 선택할 배열 원소 위치에는 True, 그렇지 않으면 False 값을 줌

```
1 import numpy as np
2 a = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 b = a % 2 == 1
4 odd = a[b]
5 print(odd)
```

[1 3 5 7 9]

부울형 배열을 인덱스로 사용하여 a 원소
중 홀수 값만 추출하여 1차원 배열로 반환

```
1 import numpy as np
2 python = [[45,67], [77, 89], [90, 50]]
3 score = np.array(python)
4 print(score[score[:, 0] >= 60])
5 print(score[score[:, 1] >= 60])
```

[[77 89]
[90 50]]
[[45 67]
[77 89]]

3명의 중간, 기말 성적이 저장된 리스트

중간 점수가 60점 이상인 학생 정보

기말 점수가 60점 이상인 학생 정보

배열 원소 값 삽입/수정/삭제

- 삽입 : numpy 라이브러리의 insert() 함수 사용
 - insert() 함수의 첫번째 매개변수는 배열 객체, 두번째 매개 변수는 삽입할 위치, 세번째 매개변수는 삽입할 값, 네번째 매개변수는 axis(축)의 값을 지정
 - axis 값을 0으로 지정하면 행방향으로 삽입, 1로 지정하면 열방향으로 삽입, 지정하지 않으면 1차원 배열로 간주하여 삽입 후 반환

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4])
3 b = np.array([[10, 20, 30, 40],
4               [50, 60, 70, 80]])
5 a1 = np.insert(a, 1, 100)
6 print(a1)
7 b1 = np.insert(b, 1, 100)
8 b2 = np.insert(b, 1, 100, axis = 0)
9 b3 = np.insert(b, 1, 100, axis = 1)
10 print(b1)
11 print(b2)
12 print(b3)
```

한 개의 행 삽입

한 개의 열 삽입

axis 값을 지정하지 않아
1 차원 배열로 반환

```
[ 1 100  2  3  4]
[[ 10 100 20 30 40 50 60 70 80]]
[[ 10 20 30 40]
 [100 100 100 100]
 [ 50 60 70 80]]
[[ 10 100 20 30 40]
 [ 50 100 60 70 80]]
```

배열 원소 값 삽입/수정/삭제

- 수정
 - 배열의 인덱싱, 슬라이싱으로 원소 값 수정

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4])
3 b = np.array([[10, 20, 30, 40],
4               [50, 60, 70, 80]])
5 a[:2] = 100
6 print(a)
7 a[0] = 1
8 print(a)
9 b[:, 1:3] *= 10
10 print(b)
```



```
[100 100   3   4]
[   1 100   3   4]
[[ 10 200 300 40]
 [ 50 600 700 80]]
```

배열 원소 값 삽입/수정/삭제

- 삭제 : numpy 라이브러리의 delete() 함수 사용
 - delete() 함수의 첫번째 매개변수는 배열 객체, 두번째 매개 변수는 삭제할 위치, 세번째 매개변수는 axis의 값을 지정
 - axis 값을 0으로 지정하면 행방향으로 삭제, 1로 지정하면 열방향으로 삭제, 지정하지 않으면 1차원 배열로 간주하여 삭제 후 반환

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4])
3 b = np.array([[10, 20, 30, 40],
4               [50, 60, 70, 80]])
5 a1 = np.delete(a, 1)
6 print(a1)
7 b1 = np.delete(b, [1, 3, 5])
8 b2 = np.delete(b, 1, axis = 0)
9 b3 = np.delete(b, 1, axis = 1)
10 print(b1)
11 print(b2)
12 print(b3)
```

[1 3 4]
[10 30 50 70 80]
[[10 20 30 40]]
[[10 30 40]
[50 70 80]]

행 하나를 삭제한 1 X 4 형태의 2차원 배열

배열 모양 수정

- 이어 붙이기 : numpy 라이브러리의 concatenate() 함수 사용
 - concatenate() 함수의 첫번째 매개변수로 주어지는 두개의 배열 객체를 이어 붙임. 두번째 매개변수는 axis의 값 지정
 - axis 값을 0으로 지정하면 행방향, 1로 지정하면 열방향으로 이어붙임

```
1 import numpy as np
2 x1 = np.arange(4)
3 x2 = np.array([10, 11, 12, 13])
4 x = np.concatenate([x1, x2])
5 print(x)
6 x1 = x1.reshape((2,2))
7 x2 = x2.reshape((2,2))
8 x = np.concatenate([x1, x2], axis = 0)
9 print(x)
10 x = np.concatenate([x1, x2], axis = 1)
11 print(x)
```

[0 1 2 3 10 11 12 13]

[[0 1]
 [2 3]
 [10 11]
 [12 13]]

[[0 1 10 11]
 [2 3 12 13]]

← 행 방향으로 두개의 배열을 이어 붙임

배열 모양 수정

- 나누기 : numpy 라이브러리의 split() 함수 사용
 - split() 함수의 첫번째 매개변수로 주어지는 배열 객체를 두개의 배열로 나눔. 두번째 매개변수는 나누는 행 또는 열의 크기
 - axis 값을 0으로 지정하면 행방향, 1로 지정하면 열방향으로 나눔

```
1 import numpy as np
2 x = np.arange(15).reshape((3,5))
3 r1, r2 = np.split(x,[2], axis = 0)
4 print(r1)
5 print(r2)
6 c1, c2 = np.split(x,[2], axis = 1)
7 print(c1)
8 print(c2)
```

[[0 1 2 3 4]
[5 6 7 8 9]
[10 11 12 13 14]]

[[0 1]
[5 6]
[10 11]]

[[2 3 4]
[7 8 9]
[12 13 14]]

전체 3행 배열을 행 방향으로 두개의 배열로 나눔(첫번째 배열을 2행 크기로)

	[[0	1		2	3	4]
		[5	6		7	8	9]
행 방향 나누기		[10	11		12	13	14]]

열 방향
나누기

배열 연산

- 배열의 각 원소에 대한 반복 연산을 하나의 명령어로 처리하는 **벡터화 연산(vectorized operation)**을 지원
 - 반복적인 계산을 반복문을 사용한 것보다 더 효율적으로 빠르게 수행
 - 산술, 비교, 논리 연산 등 모든 수학 연산에 적용
- 배열간 기본 연산 지원
 - $+$, $-$, $*$, $/$ 연산자 사용하여 배열의 각 원소에 대한 반복 연산을 하나의 명령어로 처리해줌
 - `add()`, `subtract()`, `multiply()`, `divide()` 연산 함수도 지원

배열 연산

- 브로드캐스팅(broadcasting)
 - 차원(shape)이 다른 배열간 연산에서 같은 차원으로 확장하여 연산하는 것

```
1 import numpy as np
2 a = np.array([1, 2, 3, 4])
3 a_10 = a * 10
4 print(a)
5 print(a_10)
6
7 b = np.array([[10, 20, 30, 40],
8               [50, 60, 70, 80]])
9 c = a + b
10 print(c)
```

출력:

```
[1 2 3 4]
[10 20 30 40]
[[11 22 33 44]
 [51 62 73 84]]
```

배열 a를 `[1, 2, 3, 4]`, `[1, 2, 3, 4]` 형태의 2차원 배열로 변환한 후에 배열 b와 + 연산 실행

```
1 import numpy as np
2 x = np.arange(4).reshape((4,1)) + np.arange(5)
3 print(x)
```

출력:

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

배열 연산

배열 연산 예제

```
1 import numpy as np
2 a = np.array([1, 2, 3])
3 b = np.array([10, 20, 30])
4 c = a + b # c = np.add(a, b)와 동일
5 d = a - b # d = np.subtract(a, b)와 동일
6 e = a * b # e = np.multiply(a, b)와 동일
7 f = a / b # f = np.divide(a, b)와 동일
8 g = 2*a - b
9 h = b > 10
10 i = (a == 2) & (b > 10)
11 print(c, d, e, f)
12 print(g)
13 print(h, i)
```

```
[11 22 33] [-9 -18 -27] [10 40 90] [0.1 0.1 0.1]
[-8 -16 -24]
[False True True] [False True False]
```

배열의 각 원소에 대해 비교, 관계 연산 실행해서
True, False 값 산출

배열의 각 원소에 대한 반복 연산을
하나의 명령어로 처리하여 계산 속도
가 반복문을 사용할 때보다 훨씬 빠름

리스트 연산에 for문을 사용
data = [1, 2, 3]

```
a = [] # 리스트
for i in data:
    a.append(2 * i)
print(a) # [2, 4, 6]
```

```
# 1차원 배열로 만들어 연산하기
a = np.array(data)
a = 2 * a
print(a) # [2 4 6]
```

배열 연산

- 배열 연산 예제



```
1 import numpy as np
2 a = np.array([[1,2,3],[4,5,6],[7,8,9]])
3 b = (a % 2 == 0) # a의 원소 중, 2로 나누어 떨어지면 True, 아니면 False
4 c = (a[:,2] > 5) # a의 모든 행, 3번째 열의 원소가 5보다 크면 True, 아니면 False
5 print(b)
6 print(c, c.shape)
```

```
[[False  True False]
 [ True False  True]
 [False  True False]]
[False  True  True] (3,)
```

c 배열은 1 차원 배열

배열 연산

- 통계 함수 (np 라이브러리 함수/ ndarray 객체 함수 형태로 지원)
 - mean() 함수 : 평균 반환
 - median() 함수 : 중앙값 반환 (np 라이브러리 함수로만 지원)
 - std() 함수 : 표준편차 반환
 - max(), min(), sum() 함수 : 최대값, 최소값, 합계 반환

배열 통계 함수

- `sum(배열, axis = n)` **필요하면 지정**
 - 각 배열 원소들을 더하는 함수
 - axis 인자가 0이면 열끼리 더함
 - axis 인자가 1이면 행끼리 더함

```
1 import numpy as np
2 x = np.arange(10).reshape((2,5))
3 print(x)
4 s = np.sum(x)
5 print(s)
6 s1 = np.sum(x, axis = 0)
7 print(s1)
8 s2 = np.sum(x, axis = 1)
9 print(s2)
10 min_x = np.min(x)
11 print(min_x)
12 max_x = np.max(x)
13 print(max_x)
14 mean_x = np.mean(x)
15 print(mean_x)
```

```
[[0 1 2 3 4]
 [5 6 7 8 9]]
45
[ 5  7  9 11 13]
[10 35]
0
9
4.5
```

