데이터분석 프로그래밍
수치 데이터 처리

임현기

리스트와 넘파이

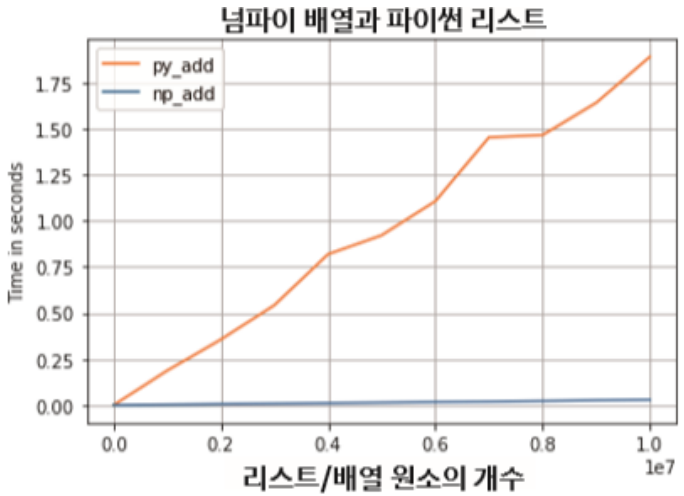
- 리스트는 여러 개의 값들을 저장할 수 있는 자료 구조로 강력하고 활용도가 높음
  - 다양한 자료형의 데이터 저장
  - 변경, 추가, 제거 가능

- 리스트의 한계
  - 리스트와 리스트 간의 연산 한계(기능 부족)
  - 연산 속도가 느림

- 데이터 분석할 때 리스트의 한계점들을 개선하고자 넘파이(Numpy)를 사용

리스트와 넘파이

- 넘파이
  - 파이썬 라이브러리
  - 대용량의 배열, 행렬 연산을 빠르게 수행
  - 고차원적인 수학 연산자와 함수를 포함



넘파이

- 넘파이의 다차원 배열
  - 2차원 이상의 배열을 생성
  - 배열의 각 요소는 인덱스로 참조 가능
  - 차원을 축(axis)라고 함

넘파이

- ndarray 객체
  - C언어 기반 배열 구조
  - 메모리를 적게 차지하고 속도가 빠름
  - 배열과 배열 간의 수학적인 연산 적용
  - 고급 연산자와 풍부한 함수들 제공

넘파이

- 중간고사 성적과 기말고사 성적의 합계

```
mid_scores = [10, 20, 30]    # 파이썬 리스트 mid_scores
final_scores = [70, 80, 90]  # 파이썬 리스트 final_scores
```

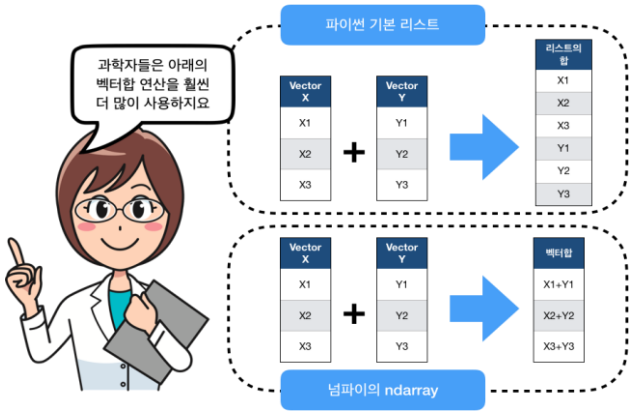
	중간고사 성적	기말고사 성적	총점
학생 #1	10	70	80
학생 #2	20	80	100
학생 #3	30	90	120

넘파이

- 리스트의 + 연산

```
>>> total = mid_scores + final_scores    # 원소간의 합이 아닌 리스트를 연결함
>>> total
[10, 20, 30, 70, 80, 90]
```

- 넘파이 배열들의 + 연산은 대응되는 값끼리 + 연산이 수행됨



넘파이

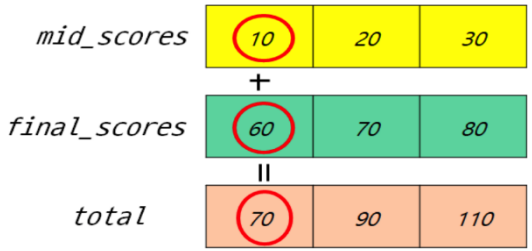
```
import numpy as np
```

- import ~ as ~
- numpy의 별칭으로 np를 주로 사용함

- 넘파이 배열을 만들려면 array() 함수를 이용

넘파이

```
mid_scores = np.array([10, 20, 30])
final_scores = np.array([60, 70, 80])
```




```
total = mid_scores + final_scores
print('시험성적의 합계 :', total)     # 각 요소별 합계가 나타난다
print('시험성적의 평균 :', total/2)   # 모든 요소를 2로 나눈다
```

```
시험성적의 합계 : [ 70  90 110]
시험성적의 평균 : [35. 45. 55.]
```

넘파이

도전문제 10.1

다음과 같은 연산의 결과를 출력해 보자.

```
>>> a = np.array(range(1, 11))
>>> b = np.array(range(10, 101, 10))
>>> a + b, a - b, a * b, a / b
```

다차원 배열

- 넘파이의 다차원 배열 ndarray는 몇가지 속성을 가짐

```
>>> a = np.array([1, 2, 3])        # 넘파이 ndarray 객체의 생성
>>> a.shape       # a 객체의 형태(shape)
(3,)
>>> a.ndim        # a 객체의 차원
1
>>> a.dtype       # a 객체 내부 자료형
dtype('int32')
>>> a.itemsize    # a 객체 내부 자료형이 차지하는 메모리 크기(byte)
4
>>> a.size        # a 객체의 전체 크기(항목의 수)
3
```

속성	설명
ndim	배열 축 혹은 차원의 개수
shape	배열의 차원으로 (m, n) 형식의 튜플 형이다. 이때, m과 n은 각 차원의 원소의 크기를 알려주는 정수
size	배열 원소의 개수이다. 이 개수는 shape내의 원소의 크기의 곱과 같다. 즉 (m, n) 형태 배열의 size는 $m \cdot n$이다.
dtype	배열내의 원소의 형을 기술하는 객체이다. 넘파이는 파이썬 표준 자료형을 사용할 수 있으나 넘파이 자체의 자료형인 bool_, character, int_, int8, int16, int32, int64, float, float8, float16_, float32, float64, complex_, complex64, object 형을 사용할 수 있다.
itemsize	배열내의 원소의 크기를 바이트 단위로 기술한다. 예를 들어 int32 자료형의 크기는 32/8 = 4 바이트가 된다.
data	배열의 실제 원소를 포함하고 있는 버퍼
stride	배열 각 차원별로 다음 요소로 점프하는 데에 필요한 거리를 바이트로 표시한 값을 모은 튜플

다차원 배열

도전문제 10.2

다음과 같은 연산의 결과를 출력해 보자.

```
>>> a = np.array(range(1, 11)) + np.array(range(10, 101, 10))
>>> a.shape
_______
>>> a.size
_______
```

배열의 연산

- 전직원 월급 100만원씩 인상

```
import numpy as np
salary = np.array([220, 250, 230])
```

```
salary = salary + 100
print(salary)
```

```
[320, 350, 330]
```

배열의 연산

- 전직원 월급 2배 인상

```
salary = np.array([220, 250, 230])
salary = salary * 2
print(salary)
```

```
[440, 500, 460]
```

```
salary = np.array([220, 250, 230])
salary = salary * 2.1
print(salary)
```

```
[462. 525. 483.]
```

넘파이를 이용하여 다수의 인원에 대해 BMI 계산을 효율적으로 적용해 보자.
수정 병원에서는 연구를 위하여 모집한 다수의 실험 대상자들의 키와 몸무게를 측정하였다. 하나의 리스트는 실험 대상자들의 키를 저장한 리스트로서 heights라고 하자. 또 하나의 리스트는 몸무게를 저장한 리스트로서 weights라고 하자.
수정 병원의 실험 대상자들의 BMI를 한 번에 계산할 수 있는 방법은 무엇일까?

원하는 결과

```
대상자들의 키: [1.83 1.76 1.69 1.86 1.77 1.73]
대상자들의 몸무게: [86 74 59 95 80 68]
대상자들의 BMI
[25.68007405 23.88946281 20.65754 27.45982194 25.53544639 22.72043837]
```

```python
import numpy as np

heights = [ 1.83, 1.76, 1.69, 1.86, 1.77, 1.73 ]
weights = [ 86, 74, 59, 95, 80, 68 ]

np_heights = np.array(heights)
np_weights = np.array(weights)

bmi = np_weights/(np_heights**2)
print('대상자들의 키:', np_heights)
print('대상자들의 몸무게:', np_weights)
print('대상자들의 BMI')
print(bmi)
```

넘파이 인덱싱/슬라이싱

```
>>> scores = np.array([88, 72, 93, 94, 89, 78, 99])
```

```
>>> scores[2]
93
```

```
>>> scores[-1]
99
```

넘파이 인덱싱/슬라이싱

```
>>> scores[1:4]      # 첫 번째, 두 번째, 세 번째, 네 번째 항목을 슬라이싱 함
array([72, 93, 94])
```

인덱싱은 특정한 요소를 얻는 방법

	0	1	2	3	4	5	6
scores	88	72	93	94	89	78	99
	-7	-6	-5	-4	-3	-2	-1

```
>>> scores[2]
93
```

슬라이싱은 요소 집합을 얻는 방법

	0	1	2	3	4	5	6
scores	88	72	93	94	89	78	99
	-7	-6	-5	-4	-3	-2	-1

```
>>> scores[1:4]
[72, 93, 94]
```

```
>>> scores[3:]       # 마지막 인덱스를 생략하면 디폴트 값은 -1임
array([94, 89, 78, 99])
>>> scores[4:-1]     # 마지막 인덱스로 -1을 사용할 경우 -1의 앞에 있는 78까지 슬라이싱함
array([89, 78])
```

논리 인덱싱

- 20살 이상인 조건식

```
>>> ages = np.array([18, 19, 25, 30, 28])
```

```
>>> y = ages > 20
>>> y
array([False, False, True, True, True])
```

- 20살 이상인 사람들을 뽑아내는 연산

```
>>> ages[ ages > 20 ]
array([25, 30, 28])
```

잠깐 – BMI가 25가 넘는 사람만 추출해 보자

앞서 실습을 통해 여러 사람의 BMI를 출력해 보았다. 이제 BMI가 25가 넘는 사람의 BMI만 출력하도록 해보자. 키와 몸무게 값을 담은 넘파이 배열 np_heights와 np_weights가 이미 만들어져 있다면 다음과 같이 구할 수 있다.

```
bmi = np_weights/(np_heights**2)
print(bmi[bmi > 25])       # BMI가 25 넘는 사람의 BMI만을 출력
```

2차원 배열 인덱싱

- 파이썬의 2차원 리스트는 '리스트의 리스트'
  - 행렬로 처리되지 못함 (행렬 연산 불가)

- 넘파이의 2차원 배열은 행렬 연산 가능
  - 역행렬, 행렬식 등

```
>>> np_array = np.array(y)  # 2차원 배열(넘파이 다차원 배열)
>>> np_array
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

2차원 배열 인덱싱

```
>>> np_array = np.array(y)  # 2차원 배열(넘파이 다차원 배열)
>>> np_array
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

- np_array[0][2]는 3
- np_array[0, 1]로도 사용 가능

```
>>> np_array[0, 0]
1
>>> np_array[2, -1]
9
```

2차원 배열 인덱싱

- 배열의 요소 변경

```
>>> np_array[0, 0] = 12   # ndarray의 첫 요소를 변경함
>>> np_array
array([[12, 2, 3],
       [ 4, 5, 6],
       [ 7, 8, 9]])
```

- 넘파이 배열은 모든 항목이 동일한 자료형
  - 정수 배열에 부동 소수점 값을 삽입하려고 하면

```
>>> np_array[2, 2] = 1.234  # 마지막 요소의 값을 실수로 변경하려고 하면 실패
>>> np_array
array([[12, 2, 3],
       [ 4, 5, 6],
       [ 7, 8, 1]])
```

2차원 배열 인덱싱

- 부분 행렬 인덱싱

```
>>> np_array = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
>>> np_array[0:2, 2:4]
array([[3, 4],
       [7, 8]])
```

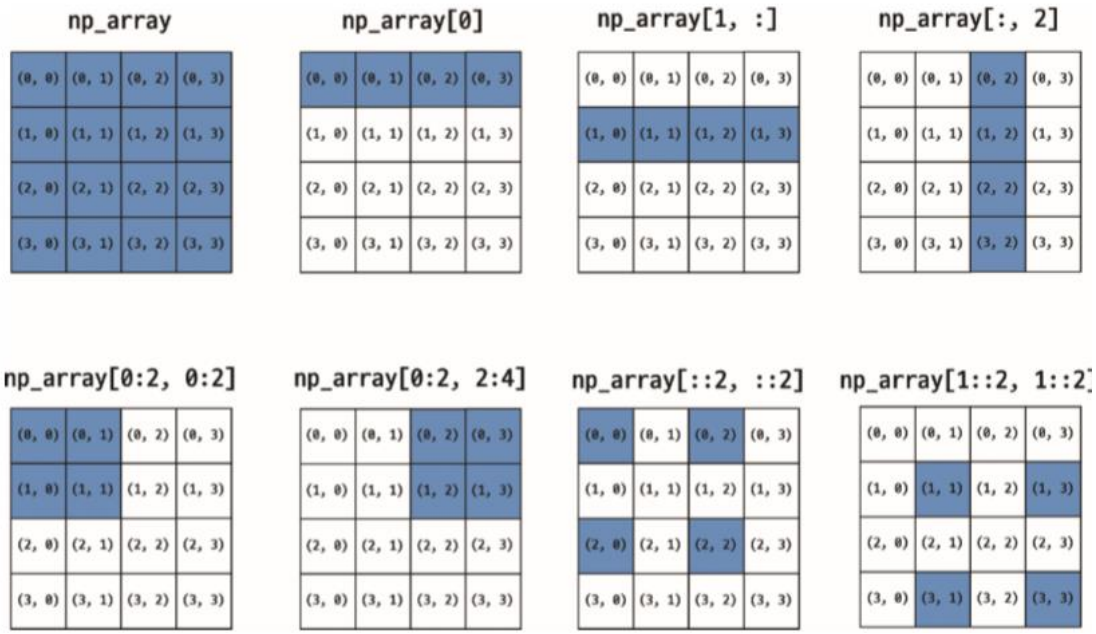
- 하나의 행을 지정

```
>>> np_array[0]
array([1, 2, 3, 4])
```

- 행렬에 넘파이 표기법 사용

```
>>> np_array[1, 1:3]
array([6, 7])
```

2차원 배열 인덱싱


np_array
np_array[0]
np_array[1, :]
np_array[:, 2]
np_array[0:2, 0:2]
np_array[0:2, 2:4]
np_array[::2, ::2]
np_array[1::2, 1::2]

잠깐 – 파이썬 리스트 슬라이싱과 넘파이 스타일 슬라이싱의 차이

다음과 같이 파이썬 리스트 슬라이싱과 넘파이 스타일의 슬라이싱을 적용했을 때, 슬라이싱에 사용된 범위와 간격은 동일하지만 전혀 다른 결과가 나온다. 이 이유를 잘 이해하는 것이 중요하다.

```
np_array = np.array([[ 1,  2,  3,  4],
                     [ 5,  6,  7,  8],
                     [ 9, 10, 11, 12],
                     [13, 14, 15, 16]])
print(np_array[::2][::2]) # 첫 슬라이싱: 0행, 2행 선택, 두 번째 슬라이싱: 그 중 0행 선택
print(np_array[::2,::2])  # 행 슬라이싱: 0행, 2행 선택, 열 슬라이싱: 0열 2열 선택
```

```
[[1 2 3 4]]
[[ 1  3]
 [ 9 11]]
```

2차원 배열 논리 인덱싱

```
>>> np_array = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np_array > 5
array([[False, False, False],
       [False, False, True],
       [ True, True, True]])
```

- 특정한 값들을 뽑는다면

```
>>> np_array[ np_array > 5 ]
array([6, 7, 8, 9])
```

2차원 배열 논리 인덱싱

```
>>> np_array = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np_array > 5
array([[False, False, False],
       [False, False, True],
       [ True, True, True]])
```

- 세번재 열의 값

```
>>> np_array[:, 2]
array([3, 6, 9])
```

  – 이 중에 5를 넘는 값

```
>>> np_array[:, 2] > 5
array([False, True, True])
```

2차원 배열 논리 인덱싱

- 짝수 찾기

```
>>> np_array[:] % 2 == 0
array([[False,  True, False],
       [ True, False,  True],
       [False,  True, False]])
```

```
>>> np_array[ np_array % 2 == 0 ]
array([2, 4, 6, 8])
```

문자 데이터를 저장하고 있는 어떤 2차원 넘파이 배열 x 에서 'c' 문자가 몇 개 있는지 알고 싶다. 'c'만을 추출하여 배열을 만들어 보라.

```
x = np.array( [['a', 'b', 'c', 'd'],
               ['c', 'c', 'g', 'h']])
```

그리고 다음과 같은 두 개의 2차원 배열에 정수를 담아 두 배열을 더해서 결과를 확인해 보라.

```
mat_a = np.array( [[10, 20, 30], [10, 20, 30]])
mat_b = np.array( [[2, 2, 2], [1, 2, 3]])
```

원하는 결과

```
['c' 'c' 'c']
[[ 8 18 28]
 [ 9 18 27]]
```

```
print(x [ x == 'c' ])
print(mat_a - mat_b)
```

검사 대상자들의 키와 몸무게가 다음과 같이 2차원 넘파이 배열에 저장되었다고 하자.

```
import numpy as np

x = np.array([[ 1.83, 1.76, 1.69, 1.86, 1.77, 1.73 ],
              [ 86.0, 74.0, 59.0, 95.0, 80.0, 68.0 ]])
y = x[0:2, 1:3]
z = x[0:2][1:3]
```

x와 y, 그리고 z의 형태가 어떠한지 확인해 보라.
그리고 이 정보를 바탕으로 각 대상자들의 BMI 값을 저장한 배열을 생성해 보라.

원하는 결과

```
x shape : (2, 6)
y shape : (2, 2)
z shape : (1, 6)
z values = : [[86. 74. 59. 95. 80. 68.]]
BMI data
[0.00024743 0.0003214  0.00048549 0.00020609 0.00027656 0.00037413]
```

```
import numpy as np

x = np.array([[ 1.83, 1.76, 1.69, 1.86, 1.77, 1.73 ],
[ 86.0, 74.0, 59.0, 95.0, 80.0, 68.0 ]])
y = x[0:2, 1:3]
z = x[0:2][1:3]

print('x shape :', x.shape)
print('y shape :', y.shape)
print('z shape :', z.shape)
print('z values = :', z)

bmi = x[0] / x[1]**2
print('BMI data')
print(bmi)
```

선수들의 키와 몸무게가 하나의 리스트를 구성하고 있으며, 또 이들의 리스트로 이루어진 데이터 player가 있다.

```
players = [[170, 76.4],
           [183, 86.2],
           [181, 78.5],
           [176, 80.1]]
```

이것을 바탕으로 넘파이 2차원 배열을 만들어 보고, 선수들 가운데 몸무게가 80을 넘는 선수들만 골라서 정보를 출력해 보자. 또 키가 180 이상인 선수들의 정보도 추출해 보자.

원하는 결과

```
몸무게가 80 이상인 선수 정보
[[183.    86.2]
 [176.    80.1]]
키가 180 이상인 선수 정보
[[183.    86.2]
 [181.    78.5]]
```

```python
import numpy as np

players = [[170, 76.4],
           [183, 86.2],
           [181, 78.5],
           [176, 80.1]]

np_players = np.array(players)

print('몸무게가 80 이상인 선수 정보')
print(np_players[ np_players[:, 1] >= 80.0 ])

print('키가 180 이상인 선수 정보')
print(np_players[ np_players[:, 0] >= 180.0 ])
```

arange() 함수

- arange()


```
>>> import numpy as np
>>> np.arange(5)
array([0, 1, 2, 3, 4])
```

arange() 함수

- 시작값 설정

```
>>> np.arange(1, 6)
array([1, 2, 3, 4, 5])
```

- 증가되는 값 설정

```
>>> np.arange(1, 10, 2)
array([1, 3, 5, 7, 9])
```

- range()를 써서 넘파이 배열로

```
>>> np.array(range(5))
array([0, 1, 2, 3, 4])
```

linspace(), logspace()

여기서는 10을 베이스로 잡았지만, base 키워드 매개변수에 설정한 인자에 따라 바꿀수도 있다.

linspace(), logspace()

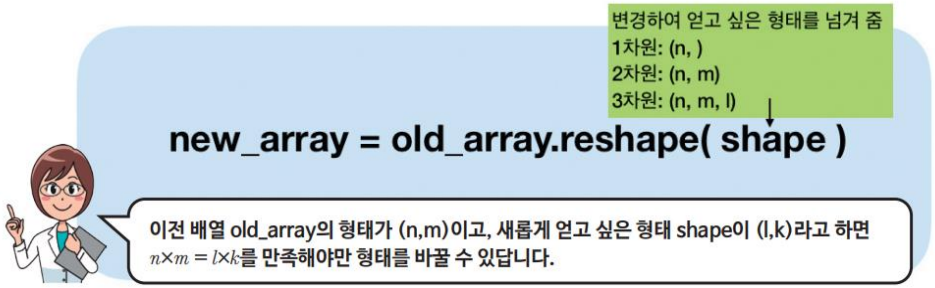
```
>>> np.linspace(0, 10, 100)
array([ 0.        , 0.1010101 , 0.2020202 , 0.3030303 , 0.4040404 ,
        ...
        8.08080808, 8.18181818, 8.28282828, 8.38383838, 8.48484848,
        8.58585859, 8.68686869, 8.78787879, 8.88888889, 8.98989899,
        9.09090909, 9.19191919, 9.29292929, 9.39393939, 9.49494949,
        9.5959596 , 9.6969697 , 9.7979798 , 9.8989899 , 10.        ])
```

linspace(0, 10, 100)이라고 호출하면 0에서 10까지 총 100개의 수들이 생성

```
>>> np.logspace(0, 5, 10)
array([1.00000000e+00, 3.59381366e+00, 1.29154967e+01, 4.64158883e+01,
       1.66810054e+02, 5.99484250e+02, 2.15443469e+03, 7.74263683e+03,
       2.78255940e+04, 1.00000000e+05])
```

logspace(x, y, n) : 생성되는 수의 시작은 10^x 부터 10^y 까지가 되며, n 개의 수가 생성

reshape(), flatten()




```
>>> y = np.arange(12)
>>> y
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

```
>>> y.reshape(3, 4)
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

reshape(), flatten()

```
>>> y.reshape(6, -1)
array([[ 0, 1],
       [ 2, 3],
       [ 4, 5],
       [ 6, 7],
       [ 8, 9],
       [10, 11]])
```

인수로 -1을 전달하면 데이터의 개수에 맞춰서 자동으로 배열의 형태가 결정

```
>>> y.reshape(7, 2)
...
    y.reshape(7, 2)
ValueError: cannot reshape array of size 12 into shape (7,2)
```

reshape()에 의해 생성될 배열의 형태가 호환되지 않을 경우 발생하는 오류

```
>>> y.flatten()    # 2차원 배열을 1차원 배열로 만들어 준다
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

flatten()은 평탄화 함수로 2차원 이상의 고차원 배열을 1차원 배열로 만들어 준다.

난수 생성

- seed 설정

```
>>> np.random.seed(100)
```

- 난수 생성

```
>>> np.random.rand(5)
array([0.54340494, 0.27836939, 0.42451759, 0.84477613, 0.00471886])
```

```
>>> np.random.rand(5, 3)
array([[0.12156912, 0.67074908, 0.82585276],
       [0.13670659, 0.57509333, 0.89132195],
       [0.20920212, 0.18532822, 0.10837689],
       [0.21969749, 0.97862378, 0.81168315],
       [0.17194101, 0.81622475, 0.27407375]])
```

난수로 이루어진 2차원 배열(5x3)

```
>>> a = 10
>>> b = 20
>>> (b - a) * np.random.rand(5) + a
array([14.31704184, 19.4002982 , 18.17649379, 13.3611195 , 11.75410454])
```

10에서 20사이에 있는 난수 5개 생성

```
>>> np.random.randint(1, 7, size=10)
array([4, 3, 4, 1, 1, 2, 6, 6, 2, 6])
```

```
>>> np.random.randint(1, 11, size=(4, 7))
array([[10, 2, 6, 9, 8, 5, 3],
       [ 7, 3, 2, 9, 5, 3, 2],
       [ 3, 1, 6, 2, 9, 8, 2],
       [ 7, 5, 2, 8, 3, 3, 6]])
```

1부터 (11-1)=10사이의 4행 7열 난수 생성

난수 생성

- 정규 분포 난수 생성

```
>>> np.random.randn(5)
array([ 0.78148842, -0.65438103, 0.04117247, -0.20191691, -0.87081315])
```

```
>>> np.random.randn(5, 4)
array([[ 0.22893207, -0.40803994, -0.10392514, 1.56717879],
       [ 0.49702472, 1.15587233, 1.83861168, 1.53572662],
       [ 0.25499773, -0.84415725, -0.98294346, -0.30609783],
       [ 0.83850061, -1.69084816, 1.15117366, -1.02933685],
       [-0.51099219, -2.36027053, 0.10359513, 1.73881773]])
```

난수로 이루어진 2차원 배열
(5행 4열의 난수)

```
>>> mu = 10
>>> sigma = 2
>>> randoms = mu + sigma * np.random.randn( 5, 4 )
>>> randoms
array([[ 9.82507212,  7.12282389,  5.88878504,  7.5865665 ],
       [ 6.05953536, 11.73521791, 10.90362868, 12.33878255],
       [10.2980491 ,  8.64563344,  9.09398278, 11.20863908],
       [ 9.30825873,  9.81230228,  9.71131179, 12.47776473],
       [10.00162592,  9.86745157,  8.51138086,  9.82922367]])
```

평균이 10이고 표준편차 값이
2인 정규분포를 가지는 난수

평균, 중앙값

- 10,000명의 키 난수 생성
  - 평균 175cm, 표준편차 10

```
>>> m = 175
>>> sigma = 10
>>> heights = m + sigma * np.random.randn(10000)
>>> heights
array([165.64438142, 172.90778856, 207.82109029, ..., 163.88596647,
       177.29401299, 180.7002071 ])
```

평균, 중앙값

```
>>> np.mean(heights)
175.14185004766918
```

```
>>> np.median(heights)
175.0251183448534
```

중앙값 계산: 리스트의 중앙에 있는 항목

```
>>> a = np.array([ 3, 7, 1, 2, 21])  # 21은 전체 데이터 중에서 비정상적으로 큰 값이다
>>> np.mean(a)
6.8
>>> np.median(a)   # [3, 7, 1, 2, 21]들 중 가운데 항목을 구한다
3.0
```

축구 선수들 100명의 데이터가 2차원 넘파이 배열에 저장되어 있다. 각 선수당 (키, 몸무게, 나이)를 저장한다. 넘파이 배열의 이름은 players이다. 정규 분포를 이용하여 자동으로 데이터를 생성해 보자. 100명의 선수가 각각 키, 몸무게, 나이를 가지므로 배열의 형태는 (100,3)이 된다. 생성할 때 키는 175, 몸무게는 70, 나이는 22세를 평균으로 하고, 표준편차는 모두 10이 되도록 하여 생성해서 생성된 데이터의 실제 평균과 중앙값을 구해 보자.

원하는 결과

```
신장 평균값:  175.05662687914526
신장 중앙값:  174.19575035548786
체중 평균값:  68.27591327992555
체중 중앙값:  68.93281750317813
나이 평균값:  20.68
나이 중앙값:  21.0
```

```python
import numpy as np

players = np.zeros( (100, 3) )
players[:, 0] = 10 * np.random.randn(100) + 175
players[:, 1] = 10 * np.random.randn(100) + 70
players[:, 2] = np.floor(10 * np.random.randn(100)) + 22

heights = players[:, 0]
print('신장 평균값:', np.mean(heights))
print('신장 중앙값:', np.median(heights))

weights = players[:, 1]
print('체중 평균값:', np.mean(weights))
print('체중 중앙값:', np.median(weights))

ages = players[:, 2]
print('나이 평균값:', np.mean(ages))
print('나이 중앙값:', np.median(ages))
```

상관관계 계산

- corrcoef(x, y)

```
x = [ 1, 2, 3, 4, …, 97, 98, 99 ]
y = [ 1, 4, 9, 16, …, 9409, 9604, 9801]
```

```
import numpy as np

x = [ i for i in range(100) ]       # 0에서 99까지의 값을 요소로 하는 리스트
y = [ i ** 2 for i in range(100) ]  # 0에서 99까지의 값의 제곱을 요소로 하는 리스트

result = np.corrcoef(x, y)
print(result)
```

```
[[1.         0.96764439]
 [0.96764439 1.        ]]
```

잠깐 – corrcoef() 함수가 계산하는 상관관계의 수학적 정의

넘파이의 corrcoef() 함수는 수학적으로 **피어슨**Pearson 상관 계수를 계산하는 함수이다. 피어슨 상관 계수는 통계학에서 사용하는 상관계수로 두 변량의 공분산을 각각의 표준편차를 서로 곱한 값으로 나눈 것이다. 상세한 것은 통계학 교과서를 참고하도록 하자.

상관관계 계산

```
x = [ i for i in range(100) ]
y = [ i ** 2 for i in range(100) ]
z = [ 100 * np.sin(3.14*i/100) for i in range(100) ]
```

```
result = np.corrcoef( [x, y, z] )
print(result)
```

```
[[ 1.          0.96764439  0.03763255]
 [ 0.96764439  1.         -0.21532645]
 [ 0.03763255 -0.21532645  1.        ]]
```