넘파이(Numpy)로 수치 데이터를 처리해보자!

10.1 리스트보다 넘파이의 배열이 훨~씬~ 빠르다

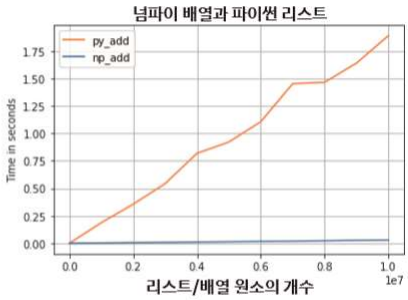
- 리스트는 여러 개의 값들을 저장할 수 있는 자료구조로서 강력하고 활용도가 높다. 리스트는 다양한 자료형의 데이터를 여러 개 저장할 수 있으며 데이터를 변경하거나 추가, 제거할 수 있다.

```
>>> scores = [10, 20, 30, 40, 50, 60]
```

- 하지만 데이터 과학에서는 파이썬의 기본 리스트로 충분하지 않다. 데이터를 처리할 때는 리스트와 리스트 간의 다양한 연산이 필요한데, 파이썬 기본 리스트는 이러한 기능이 부족하며 리스트를 다루는 일은 연산 속도도 빠르지 않다. 따라서 데이터 과학자들은 기본 리스트 대신에 넘파이Numpy를 선호한다.

10.1 리스트보다 넘파이의 배열이 훨~씬~ 빠르다

- 넘파이는 대용량의 배열과 행렬 연산을 빠르게 수행하며, 고차원적인 수학 연산자와 함수를 포함하고 있는 파이썬 라이브러리이다.
- 표에서 알 수 있듯이 넘파이의 배열은 주황색으로 표시된 파이썬의 리스트에 비하여 처리속도가 매우 빠름을 알 수 있다.



출처: Performance of Numpy Array vs Python List, Cory Gough

10.1 리스트보다 넘파이의 배열이 훨~씬~ 빠르다

- 넘파이의 핵심적인 객체는 다차원 배열이다. 예를 들어서 정수들의 2차원 배열(테이블)을 넘파이를 이용해서 생성할 수 있다. 배열의 각 요소는 인덱스index라고 불리는 정수들로 참조된다.
- 넘파이에서 차원은 축axis이라고도 한다.

10.2 리스트와 넘파이 배열은 무엇이 다른가

- 데이터 과학자들은 왜 넘파이를 많이 사용할까?
- 넘파이는 성능이 우수한 ndarray 객체를 제공한다.
- ndarray의 장점을 정리하면 아래와 같다.

  - ndarray 는 C 언어에 기반한 배열 구조이므로 메모리를 적게 차지하고 속도가 빠르다.
  - ndarray 를 사용하면 배열과 배열 간에 수학적인 연산을 적용할 수 있다.
  - ndarray 는 고급 연산자와 풍부한 함수들을 제공한다.

10.2 리스트와 넘파이 배열은 무엇이 다른가

- 넘파이 배열의 장점을 이해하기 위한 간단한 예제를 살펴보자. 다음과 같이 학생들의 중간고사와 기말고사 성적을 저장하고 있는 리스트가 있다고 하자.

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

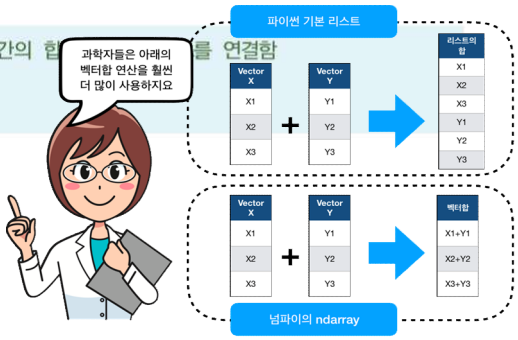
- Mid scores는 학생 3명의 중간고사 성적을 저장하고 final scores은 기말고사 성적을 저장하고 있는데, 다음 표와 같이 각 학생들의 중간고사 성적과 기말고사 성적을 합하여 오른쪽의 총점(total)이라는 리스트를 만들고 싶다.

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

10.2 리스트와 넘파이 배열은 무엇이 다른가

- 그런데, 파이썬 리스트 더하기 연산자는 두 리스트를 연결하므로, mid_scores + final_scores을 적용하면 다음과 같이 2개의 리스트를 연결한 리스트가 만들어진다.

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



- 이것은 분명히 우리가 원하는 연산이 아니다.
- 하지만 넘파이 배열에 +연산을 하면, 대응되는 값끼리 합쳐진 결과를 얻을 수 있다.

10.3 넘파이의 별칭 만들기, 그리고 간단한 배열 연산하기

- 파이썬에서 넘파이를 사용하려면, 다음과 같이 넘파이 패키지를 불러와야 한다.
- import ~ as에서 as 뒤에 나타나는 이름은 as 앞의 이름을 대체하는 별칭이다. 보통 numpy의 별칭alias으로 np를 사용한다.
- 앞으로 사용하게 될 넘파이 사용 프로그램에서 이 코드의 호출은 필수이다.

```
import numpy as np
```

- 넘파이 배열을 만들려면 넘파이가 제공하는 array( ) 함수를 이용한다. array( ) 함수에 파이썬 리스트를 전달하면 넘파이 배열이 생성된다.

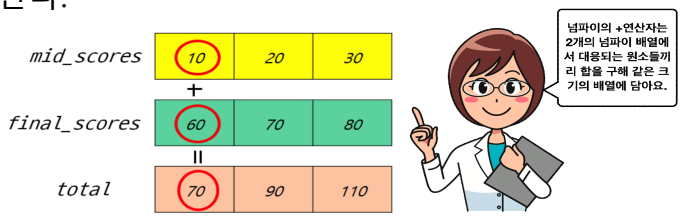
넘파이 배열 — 파이썬 리스트

```
mid_scores = np.array([10, 20, 30])
```

10.3 넘파이의 별칭 만들기, 그리고 간단한 배열 연산하기

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

- 2개의 배열을 합하여 학생들의 총점을 계산해보자. 이번에는 ndarray의 덧셈 연산을 수행한다.




```
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.]
```

10.3 넘파이의 별칭 만들기, 그리고 간단한 배열 연산하기

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
```

10.4 넘파이의 핵심 다차원배열을 알아보자

- 넘파이의 핵심이 되는 다차원배열ndarray은 다음과 같은 속성을 가지고 있다.
- 이러한 속성을 이용하여 프로그램의 오류를 찾거나 배열의 상세한 정보를 손쉽게 조회할 수 있다.

```
>>> 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
```

10.4 넘파이의 핵심 다차원배열을 알아보자

- 각각의 속성에 관련한 상세한 설명은 다음 표와 같다.

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

10.4 넘파이의 핵심 다차원배열을 알아보자

잠깐 – 네이처에 소개된 넘파이

2020년 9월 저명한 과학 저널 **네이처**Nature에 넘파이에 대한 리뷰 논문이 게재되었다. 기초과학 분야 논문을 주로 게재하는 네이처에 소프트웨어 모듈이 소개되는 것은 매우 이례적인 일이다. 논문은 **텐서**tensor라 불리는 다차원 배열을 효율적으로 다루는 넘파이가 과학 분야에 파이썬 생태계를 제공했고, 과학 각 분야의 수치 데이터 처리 기술들이 상호운용성을 가질 수 있도록 하는 역할을 했다고 밝히고 있다. 블랙홀의 모습을 최초로 찍어낸 **이벤트 호라이즌**Event Horizon 연구팀이 넘파이 배열을 이용해 연구의 각 단계별 데이터를 저장하고 다루었던 사례도 소개하였다.

영국 왕립 천문학회는 넘파이를 바탕으로 만들어진 Astropy에게 상을 수여하면서 "Astropy 프로젝트는 수백 명의 젊은 과학자들에게 버전 제어, 단위 검사, 코드 리뷰, 이슈 트래킹과 같은 전문가 수준의 소프트웨어 개발 경험을 제공했다. 현대의 연구자에게 이것들은 핵심적 기술들인데 물리학이나 천문학 분야의 정규 대학 교육에서는 종종 누락되고 있다."라고 밝혔다.

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
_________
```

LAB10-1 nadrray 객체를 생성하고 속성을 알아보자

넘파이의 다차원 배열을 이해하기 위해 다음과 같은 일들을 수행해 보자.

1. 0에서 9까지의 정수 값을 가지는 ndarray 객체 array_a를 넘파이를 이용하여 작성하여 내용을 출력해 보라.
2. range() 함수를 사용하여 0에서 9까지의 정수 값을 가지는 ndarray 객체 array_b를 만들고 아래의 결과와 같이 나타나도록 하여라.
3. 문제 2의 코드를 수정하여 0에서 9까지의 정수 값 중에서 다음과 같이 짝수를 가지는 ndarray 객체 array_c를 생성하여라.
4. array_c의 shape, ndim, dtype, size. itemsize를 출력해 보라.

원하는 결과

```
실습 1 : array_a = [0 1 2 3 4 5 6 7 8 9]
실습 2 : array_b = [0 1 2 3 4 5 6 7 8 9]
실습 3 : array_c = [0 2 4 6 8]
실습 4:
array_c의 shape : (5,)
array_c의 ndim : 1
array_c의 ctype : int64
array_c의 size : 5
array_c의 itemsize : 8
```

10.4 넘파이의 핵심 다차원배열을 알아보자

```
import numpy as np

# 실습 1
array_a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print('실습 1 : array_a =', array_a)


# 실습 2
array_b = np.array(range(10))
print('실습 2 : array_b =', array_b)

# 실습 3
array_c = np.array(range(0,10,2))
print('실습 3 : array_c =', array_c)

# 실습 4
print('실습 4: ')
print('array_c의 shape :', array_c.shape)
print('array_c의 ndim :', array_c.ndim)
print('array_c의 ctype :', array_c.dtype)
print('array_c의 size :', array_c.size)
print('array_c의 itemsize :',array_c.itemsize)
```

10.5 강력한 넘파이 배열의 연산을 알아보자

- 넘파이 배열에는 + 연산자나 * 연산자와 같은 수학적인 연산자를 얼마든지 적용할 수 있다. 예를 들어서 어떤 회사가 좋은 성과를 거두어 전 직원의 월급를 100만원씩 올려주기로 하였다. 어떻게 하면 될까? 현재 직원들의 월급이 [220, 250, 230]이라고 하자. 이것을 넘파이 배열에 저장한다.

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

- 넘파이 배열에 저장된 모든 값에 100을 더하려면 다음과 같이 배열에 스칼라 값 100을 더하면 된다.

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

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

10.5 강력한 넘파이 배열의 연산을 알아보자

- 위의 코드와 같이 넘파이 배열 salary에 100을 더하면 salary 배열의 모든 요소에 100이 더해진다. 만일 모든 직원들의 월급을 2배 올려주려면 어떻게 하면 될까? 다음과 같이 넘파이의 배열에 2를 곱하면 된다

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

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

- 물론 곱셈연산은 2.1과 같은 실수 값을 적용할 수도 있다

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

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

10.5 강력한 넘파이 배열의 연산을 알아보자

잠깐 – 넘파이의 계산은 왜 빠를까?

넘파이가 계산을 쉽고 빠르게 할 수 있는 데에는 이유가 있다. 넘파이는 각 배열마다 타입이 하나만 있다고 생각한다. 다시 말하면, **넘파이의 배열 안에는 동일한 타입의 데이터만 저장**할 수 있다. 즉 정수면 정수, 실수면 실수만을 저장할 수 있는 것이다. 파이썬의 리스트처럼 여러 가지 타입을 섞어서 저장할 수는 없다. 만약 여러분들이 여러 가지 타입을 섞어서 넘파이의 배열에 전달하면 넘파이는 이것을 전부 문자열로 변경한다. 예를 들어서 다음 배열은 문자열 배열이 된다.

```
>>> tangled = np.array([ 100, 'test', 3.0, False])
>>> print(tangled)
['100' 'test' '3.0' 'False']
```

이렇게 동일한 자료형으로만 데이터를 저장하면 각각의 데이터 항목에 필요한 저장공간이 일정하다. 따라서 몇 번째 위치에 있는 항목이든 그 순서만 안다면 바로 접근할 수 있기 때문에 빠르게 데이터를 다룰 수 있는 것이다. 이렇게 원하는 위치에 바로 접근하여 데이터를 읽고 쓰는 일을 **임의 접근**random access라고 한다. 우리가 주기억 장치로 많이 쓰는 기억장치가 **임의 접근 기억장치**random access memory이고 줄여서 RAM이라 한다. 임의 접근이 가능하기 때문에 기억장치가 회전하면서 원하는 위치의 데이터를 읽는 하드디스크보다 빠르다.

LAB10-2 여러 사람의 BMI를 빠르고 간편하게 계산하기

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]
```

LAB10-2 여러 사람의 BMI를 빠르고 간편하게 계산하기

```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)
```

10.6 인덱싱과 슬라이싱을 넘파이에서도 할 수 있다

- 지금부터는 다음과 같은 성적이 저장된 1차원 배열에서 요소들을 꺼내는 방법을 살펴보자.

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

- 넘파이 배열에서 특정한 요소를 추출하려면 인덱스를 사용한다.
- 파이썬 리스트와 마찬가지로 인덱스는 0부터 시작한다.
- 따라서 인덱스로 2를 지정하면 93이 출력된다.

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

- 마지막 요소에 접근하려면 리스트와 마찬가지로 인덱스로 -1을 주면 된다.

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

10.6 인덱싱과 슬라이싱을 넘파이에서도 할 수 있다

- 넘파이 배열에서는 다음과 같이 슬라이싱도 가능하다.
- 마지막 항목 인덱스가 4일때는 4-1인 scores[4]항목까지 슬라이싱 된다는 것에 주의하도록 하자.

```
>>> 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

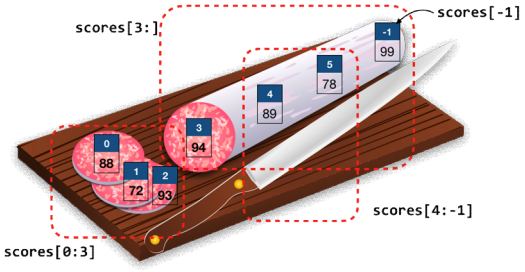
	0	1	2	3	4	5	6
scores	88	72	93	94	89	78	99
	-7	-6	-5	-4	-3	-2	-1

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

10.6 인덱싱과 슬라이싱을 넘파이에서도 할 수 있다

- 다음과 같이 시작 인덱스나 종료 인덱스는 생략이 가능하다.
- 이것은 파이썬의 리스트와 동일하다.
- 또한 scores[4:-1]과 같은 음수 인덱싱도 가능하다.

```
>>> 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])
```



10.7 논리적인 인덱싱을 통해 값을 추려내자

- 논리적인 인덱싱logical indexing이란 어떤 조건을 주어서 배열에서 원하는 값을 추려내는 것이다. 예를 들어서 사람들의 나이가 저장된 넘파이 배열 ages가 있다고 하자.

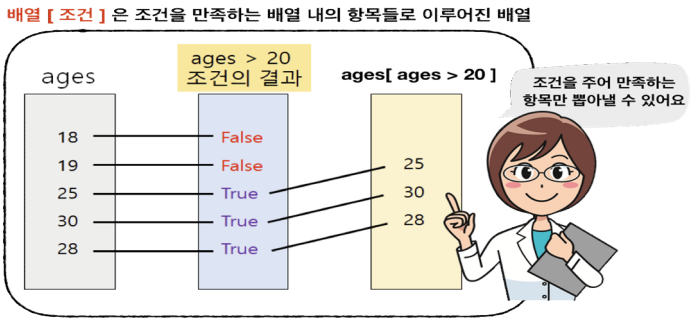
```
>>> ages = np.array([18, 19, 25, 30, 28])
```

- ages에서 20살 이상인 사람만 고르려고 하면 다음과 같은 조건식을 써준다.

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

10.7 논리적인 인덱싱을 통해 값을 추려내자

- 결과는 부울형의 넘파이 배열이 된다.
- ages 배열의 첫 번째와 두 번째 요소는 20보다 크지 않으므로 False가 되고 나머지 요소들은 모두 20보다 크므로 True가 되었다.
- 그런데 실제로는 배열 중에서 20살 이상인 사람들을 뽑아내는 연산이 많이 사용된다.
- 이때는 위의 부울형 배열을 인덱스로 하여 배열 ages에 보내면 된다.




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

10.7 논리적인 인덱싱을 통해 값을 추려내자

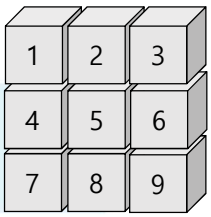
잠깐 – BMI가 25가 넘는 사람만 추출해 보자

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
```

10.8 2차원 배열 인덱싱도 해 보자

- 넘파이를 사용하면 2차원 배열도 쉽게 만들 수 있다.
- 파이썬의 2차원 리스트는 "리스트의 리스트"라고 할 수 있다.
- 수학에서의 행렬matrix과는 비슷하지만 리스트는 행렬 연산을 지원하지 않는다.



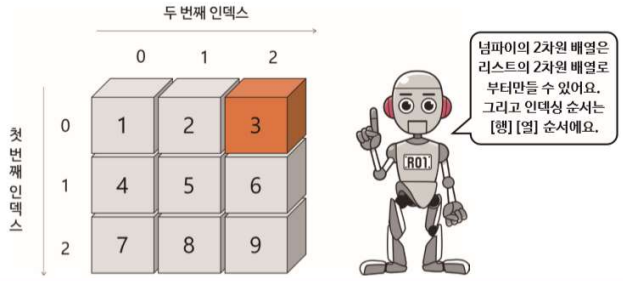

```
>>> import numpy as np
>>> y = [[1,2,3], [4,5,6], [7,8,9]]  # 2차원 배열(리스트 자료형)
>>> y
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- 넘파이 2차원 배열은 다음과 같이 np.array()를 호출하여 생성할 수 있다. 넘파이의 2차원 배열은 수학에서의 행렬과 같이 다룰 수 있다.
- 따라서 역행렬이나 행렬식을 구하는 등의 행렬 연산들이 넘파이 배열에 쉽게 적용될 수 있도록 구현되어 있다.

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

10.8 2차원 배열 인덱싱도 해 보자

- 2차원 배열에서 특정한 위치에 있는 요소는 어떻게 꺼낼까?
- 2차원 배열도 인덱스를 사용한다. 다만 2차원이기 때문에 인덱스가 2개 필요하다. 첫 번째 인덱스는 행의 번호이다.
- 두 번째 인덱스는 열의 번호이다. 예를 들어서 np_array[0][2]는 3이 된다.




```
>>> np_array[0][2]
3
```

10.9 넘파이는 넘파이 스타일로 인덱싱할 수 있다.

- 넘파이의 2차원 배열에서 np_array[0][2]와 같은 형태로도 특정한 요소를 꺼낼 수 있다.
- 하지만 넘파이에서는 많이 사용되는 표기법이 있다. 0번째 행과 2번째 열에 있는 요소에 접근할 때는 콤마를 사용하여 np_array[0, 1]로 써주어도 된다.
- 콤마 앞에 값은 행을 나타내며, 콤마 뒤에 값은 열을 나타낸다.

```
>>> np_array = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np_array[0, 2]
3
```

10.9 넘파이는 넘파이 스타일로 인덱싱할 수 있다.

- 2차원 넘파이 배열은 행렬이라는 점을 명심하자. 행렬에서는 행의 번호와 열의 번호만 있으면 특정한 요소를 꺼낼 수 있다.
- 따라서 넘파이 스타일로 [row, col] 인덱스를 사용하면 row 행을 가져온 뒤에 거기서 col 번째 항목을 찾는 것이 아니라 바로 특정 항목에 접근하게 된다.


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

10.9 넘파이는 넘파이 스타일로 인덱싱할 수 있다.

- 그리고 리스트와 유사하게 인덱스 표기법을 사용하여 배열의 요소를 변경할 수 도 있다.

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

- 파이썬 리스트와 달리, 넘파이 배열은 모든 항목이 동일한 자료형을 가진다는 것을 명심하여야 한다. 예를 들어 정수 배열에 부동 소수점 값을 삽입하려고 하면 소수점 이하값은 자동으로 사라진다.

```
>>> 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]])
```

10.10 넘파이 스타일의 2차원 배열 잘라내기

- 넘파이에서 슬라이싱은 큰 행렬에서 작은 행렬을 끄집어내는 것으로 이해하면 된다. 다음은 2차원 행렬의 일부를 추출하는 코드이다.

```
>>> 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]])
```

- 위의 표기법은 0에서 1까지의 행에서 2에서 3까지의 열로 이루어진 행렬을 지정한 것이다. 따라서 위와 같은 행렬이 출력된다.
- 넘파이의 2차원 행렬에서 하나의 행을 지정하는 방식은 다음과 같이 할 수 있다.

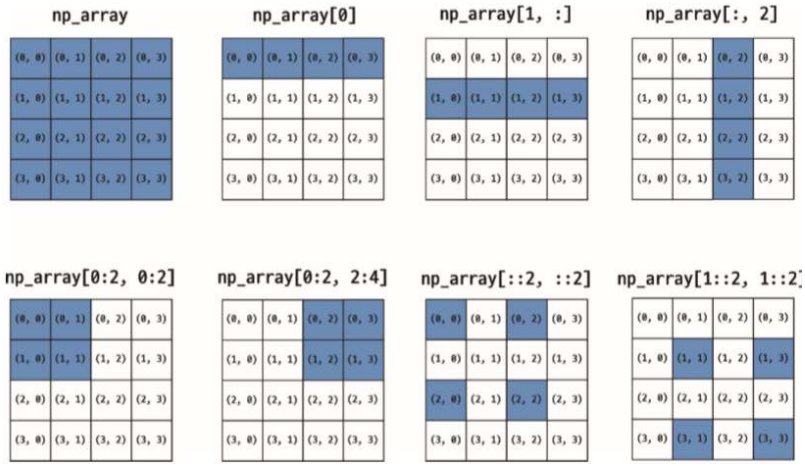
```
>>> np_array[0]
array([1, 2, 3, 4])
```

- 또한 다음과 같은 넘파이 스타일의 표기법도 가능하다.

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

10.10 넘파이 스타일의 2차원 배열 잘라내기

- 아래의 그림은 4x4 크기의 np_array라는 ndarray와 이 ndarray의 인덱싱 및 슬라이싱 결과를 보여준다



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]]
```

10.11 2차원 배열에서 논리적인 인덱싱을 해 보자

- 2차원 배열에서도 어떤 조건을 주어서 조건에 맞는 값들만 추려낼 수 있다.
- 이 코드는 np_array에 1에서 9까지의 값이 들어있는 2차원 배열에 대해서 np_array > 5 계산식의 결과를 보여준다.
- 이 계산의 결과 1, 2, 3, 4, 5까지의 값은 1>5, 2>5,..., 5>5 의 연산의 결과인 False 값이 나타남을 알 수 있다. 나머지 값인 6, 7, 8, 9는 모두 5보다 크기 때문에 True 값이 나타남을 알 수 있다.

```
>>> 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]])
```

- 2차원 배열에 대해서 비교연산자를 사용하면 위와 같이 True, False 로 이루어진 배열이 반환되는데, 이것을 이용하여 특정한 값들을 뽑아낼 수도 있다.
- 다음과 같이 코드를 살펴보자. 이 코드를 살펴보면 np_array의 큰 괄호 내부에서 np_array > 5 연산을 적용하였다.
- 이렇게 할 경우 np_array 배열내의 모든 원소들 중에서 5보다 큰 값만이 추출되어 [6, 7, 8, 9]와 같은 1차원 행렬이 출력되는 것을 볼 수 있다.

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

10.11 2차원 배열에서 논리적인 인덱싱을 해 보자

- 좀 더 나아가 아래와 같이 세번째 열을 추려내는 연산을 살펴보면 세 번째 열의 모든 원소 [3, 6, 9]가 나타남을 볼 수 있다.

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

- 이제 이 슬라이싱의 결과 값 [3, 6, 9]중에서 5를 넘는 값이 있는지를 부울형으로 반환한다. 이 경우 간단하게 다음과 같은 연산을 통해서 False, True, True를 얻을 수 있다.

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

10.11 2차원 배열에서 논리적인 인덱싱을 해 보자

- 이제 크기 비교 연산자를 약간 수정하여 다음과 같은 짝수 구하기 연산자를 적용해 보자.
- 위에서 살펴본 바와 같이 %2의 결과가 0인 경우가 짝수인 경우에 해당하므로 결과는 짝수 값이 있는 곳만 True가 나타나고 나머지는 모두 False가 나타남을 볼 수 있다.

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

- 이제 이 값이 True인 원소만을 추출하면 손쉽게 다음과 같은 1차원 배열을 얻을 수 있다. 이를 응용하면 특정수의 배수를 추출하는 등의 필터링 작업을 손쉽게 할 수 있다

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

LAB10-3 2차원 배열 연습하기

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]]
```

LAB10-3 2차원 배열 연습하기

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

잠깐 – 배열의 차원과 벡터의 차원

차원dimension은 어떤 수치 데이터를 다룰 때에 고려해야 하는 속성이 몇 개인지에 따라 결정된다. 예를 들어 3차원 공간의 위치는 x축 위의 위치, y축 위의 위치, 그리고 z축 위의 위치를 모두 고려해야만 정확한 한 점을 가리킬 수 있기 때문에 "3차원" 좌표로 표현한다.

차원이라는 용어를 많이 사용하는 데이터 구조로 배열이 있다. 가장 단순한 배열은 데이터가 하나의 줄로 나열되어 있는 것이다. 이것을 1차원 배열이라고 부른다. 수학과 과학 분야에서 이런 수치 데이터를 벡터vector라고 부른다. 벡터는 원소의 개수가 차원이 된다. 따라서 [1, 4], [3, 2, 1]은 모두 1차원 배열이지만 벡터로 간주하면 각각 1차원 데이터가 2개와 3개씩 있는 2차원 벡터, 3차원 벡터이다. **벡터의 차원은 몇 개의 항목이 있는지에 따라 결정되고, 배열의 차원은 몇 가지 방향으로 줄을 지어 있는지를 표현하는 것이다.**

배열의 차원을 한 단계 높이면 1차원 배열을 여러 줄로 겹쳐 놓은 형태가 된다. 이때 어떤 항목을 지목하는 인덱스를 표현하려면 두 가지 방향으로 각각 하나씩의 위치값이 필요하고, 이 방향을 각각 배열의 축이라 부른다. 배열의 차원은 축의 개수이므로, 이런 배열은 2차원 배열이라 부른다. 수학에서는 각각의 축을 행row과 열column이라고 부르며, 이런 모양의 데이터를 행렬라고 부른다.

같은 방식으로 배열의 차원을 높여 보자. 3차원 배열은 입체적인 모양이 될 것이다. 이런 구조가 표현하는 데이터는 벡터나 행렬이 아니라 텐서tensor라고 부른다. 텐서는 3차원 배열뿐만 아니라 모든 차원의 배열을 포괄하는 개념이므로 벡터는 1차원 텐서, 행렬은 2차원 텐서라 할 수 있다.

머릿속에 그림을 그리기는 어렵지만 코드로는 4차원, 5차원 배열을 쉽게 만들 수 있다. 3차원 배열을 나열한 배열, 그렇게 만든 4차원 배열을 또 나열한 배열이다. 그리고 이 모든 것이 텐서이다.

LAB10-4 넘파이 배열의 형태 알아내고 슬라이싱하여 연산

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]
```

LAB10-4 넘파이 배열의 형태 알아내고 슬라이싱하여 연산

```
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)
```

LAB10-5 2차원 배열에서 특정 조건을 만족하는 행만 추출하기

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]]
```

LAB10-5 2차원 배열에서 특정 조건을 만족하는 행만 추출하기

```
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 ])
```

10.12 arange() 함수와 range() 함수의 비교

- 넘파이도 많은 함수들을 가지고 있다.
- 제일 먼저 살펴볼 함수는 arange() 함수이다.


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

10.12 arange() 함수와 range() 함수의 비교

- 시작값을 지정하려면 다음과 같이 한다.

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

- 증가되는 값을 지정하려면 다음과 같이 한다.

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

- 이것은 우리가 for 반복문을 위해 많이 사용했던 range() 함수와 비슷하다. 그런데 range()를 통해 생성된 것은 반복 가능 객체이고 이것으로 리스트를 만들수 있다.

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

- 따라서 range()를 써서 arange()와 같이 넘파이 배열을 만들고 싶다면 다음과 같은 일을 수행하면 된다.

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

10.13 linspace() 함수와 logspace() 함수

- linspace()는 상당히 많이 사용되는 함수이다. linspace()는 시작값부터 끝값까지 균일한 간격으로 지정된 개수만큼의 배열을 생성한다.

- 비슷한 함수로 logspace() 함수가 있다. 이것은 로그 스케일로 수들을 생성한다.

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

10.13 linspace() 함수와 logspace() 함수

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

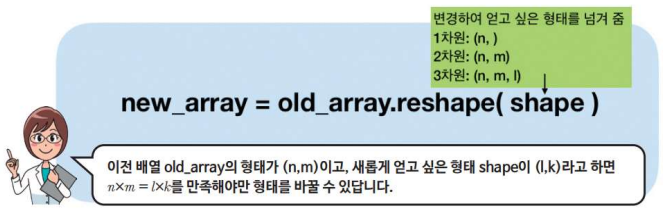
```
>>> np.linspace(0, 10, 100)
array([ 0.        , 0.1010101 , 0.2020202 , 0.3030303 , 0.4040404 ,
        ...
        8.08080808, 8.18181818, 8.28282828, 8.38383838, 8.48484848,
        8.58585859, 8.68686869, 8.78787879, 8.88888889, 8.98989899,
        9.09090909, 9.19191919, 9.29292929, 9.39393939,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

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

10.14 배열의 형태를 바꾸는 reshape() 함수와 flatten() 함수

- reshape() 함수는 상당히 많이 사용되는 함수이다. 데이터의 개수는 유지한 채로 배열의 차원과 형태를 변경한다. 이 함수의 인자인 shape을 튜플의 형태로 넘겨주는 것이 원칙이지만 reshape(x, y)라고 하면 reshape((x, y))와 동일하게 처리된다.




```
>>> 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]])
```

10.14 배열의 형태를 바꾸는 reshape() 함수와 flatten() 함수

```
>>> 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.