

파이썬 리스트와 널파이 사용법

인덱싱, 슬라이싱, 벡터화 연산

강영민
동명대학교

박동규
창원대학교

1. 파이썬 리스트 사용하기
2. 리스트 인덱싱의 이해와 편리한 내장 함수, 메소드 활용
3. 슬라이스를 통해 리스트 일부 잘라 쓰기
4. 고급 슬라이싱 - 스텝 지정, 음수 스텝, 음수 인덱스 슬라이싱
5. 널파이는 무엇인가?
6. 널파이는 리스트와 어떻게 다를까?
7. 널파이 배열의 다양한 속성
8. 널파이 배열의 인덱싱과 슬라이싱
9. 널파이 배열의 슬라이싱과 높은 인덱싱
10. 널파이는 왜 성공하나?
11. 벡터화 연산의 성능
12. 보조드래스팅과 성능 개선
13. 최적화

파이썬 리스트 자료형

이 문서는 벡터, 행렬, 텐서를 다루는 데에 강력한 기능을 제공하는 넘파이에 대한 소개이다. 그리고 독자들이 프로그래밍에 대한 기본적인 이해를 하고 있는 것으로 가정한다. 특히 파이썬 프로그래밍을 경험했다면 이문서의 내용을 구현하는 데에 큰 어려움이 없을 것이다. 프로그래밍 기법에 대한 설명은 따로 하지 않을 것이지만, 파이썬의 리스트 자료구조와 넘파이 패키지를 잘 활용할 수 있도록 도움을 주고자 한다.

선형대수에서 다루는 가장 기본적인 수는 **벡터**^{vector}라고 할 수 있다. 벡터에 대한 상세한 설명은 생략하고 간단히 그 특징만 설명하자면 여러 개의 수가 나열되어 있다는 점이다. 이렇게 데이터가 나열되어 있는 것을 다루는 데에 적합한 자료형이 바로 파이썬의 **리스트**^{list}이다.

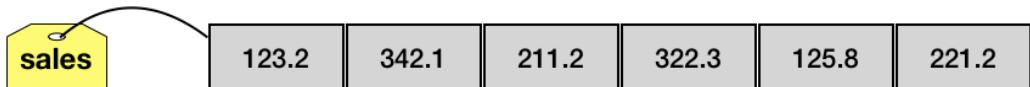
물론 넘파이가 리스트보다 더욱 강력한 수치 데이터 처리 능력을 갖고 있어, 실제 선형대수의 이론을 구현할 때에는 리스트보다 넘파이를 주로 사용할 것이다. 하지만, 넘파이가 데이터를 처리하는 기본적인 연산이 리스트가 사용하는 방법을 기본적으로 따르고 있기 때문에 리스트 자료형에 대해 살펴보겠다.

파이썬에서 리스트는 대괄호를 이용해서 만들 수 있다. 예를 들어, 지난 6 개월 동안의 매출을 리스트에 저장하려면 다음과 같이 한다. 파이썬의 리스트는 시작과 끝을 표시하기 위하여 대괄호를 사용한다. 또 내부의 항목을 분리하기 위하여 쉼표(,)를 사용한다.

```
▶ sales = [123.2, 342.1, 211.2, 322.3, 125.8, 221.2]
sales
```

```
▶ [123.2, 342.1, 211.2, 322.3, 125.8, 221.2]
```

이것은 아래 그림과 같이 **sales**라는 변수가 여섯 개의 데이터가 하나의 묶음으로 관리되는 데이터 뭉치를 가리키는 것과 같다. 벡터는 이 리스트와 같이 복수의 데이터를 담을 수 있는 자료구조가 필요하다.



여기서 사용된 `sales` 리스트 안에 저장된 각각의 데이터를 원소(element)라고 한다. 각 원소는 순서대로 나열되어 있고, 몇 번째인지 번호로 관리할 수 있다. 파이썬의 리스트는 시작과 끝을 표시하기 위하여 대괄호를 사용한다. 또 내부의 항목을 분리하기 위하여 쉼표(,)를 사용한다.

만일 초기에 아무런 원소 없이 빈 리스트를 만들고 싶을 때는 다음과 같은 방식도 가능하다. 이때는 빈 리스트가 만들어 진다.



```
loans = []
```

공백 리스트에는 `append()` 함수로 원소를 추가할 수 있다. 리스트에 몇 개의 원소가 들어갈 것인지를 미리 예측할 수 없을 때, 공백 리스트가 유용하다.



```
iteration = 3
for _ in range(iteration):
    loans.append(100)
loans
```



```
[100, 100, 100]
```

연속된 정수 리스트는 `range()` 함수를 사용하여 쉽게 생성할 수 있다. `range()`는 (시작값, 종료값, 스텝)을 인자로 가질 수 있으며 시작값과 스텝은 생략 가능하다. 생략되었을 경우 시작값은 0, 스텝은 1이 자동으로 사용된다.



```
list_a = list(range(5))          # 0에서 5사이의 정수열을 생성(5는 포함안됨)
list_b = list(range(0, 5))        # list(range(5))와 동일한 결과
list_c = list(range(0, 5, 1))    # list(range(0, 5))와 동일한 결과
list_d = list(range(0, 5, 2))    # 생성하는 값을 2씩 증가시킴 (step = 2)
list_e = list(range(2, 5))        # 2에서 5-1까지의 연속된 수 2, 3, 4를 생성
print(list_a, list_b, list_c, list_d, list_e)
```



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

리스트 인덱싱의 이해와, 편리한 내장 함수, 메소드 활용

파이썬 리스트에 저장된 데이터를 꺼내려면 어떻게 하면 될까? 리스트에서 데이터를 추출하려면 **인덱스 index**라는 정수를 사용한다. 리스트의 첫 번째 데이터는 인덱스 0을, 두 번째 원소는 인덱스 1을 가지게 된다. 나머지 원소들도 유사하게 인덱스가 할당된다. 인덱스란 리스트에서의 항목의 번호라고 생각하면 된다. 리스트의 인덱스는 0부터 시작한다. 따라서 첫 번째 항목을 얻으려면 다음과 같이 0을 인덱스로 사용하면 된다.

```
▶ sales = [123.2, 342.1, 211.2, 322.3, 125.8, 221.2]
▶ sales[0]
```

▶ 123.2

인덱스를 바꾸면 몇 번 째 원소든지 접근할 수 있다. 아래의 경우는 두 값을 쉼표로 연결하여 튜플의 형태로 출력되었다.

```
▶ sales[3], sales[5]
```

▶ (322.3, 221.2)



파이썬에서 특이한 점은 인덱스를 음수로 쓸 수 있다는 점이다. 음수는 리스트의 맨 끝에 있는 데이터를 얻고자 할 때 아주 유용하다. 예를 들어서 다음과 같이 마지막 원소에 접근할 수 있다. 가장 처음 원소는 -6의 인덱스를 사용하면 된다.

```
▶ sales[-1], sales[-6]
```

▶ (221.2, 123.2)

인덱스	0	1	2	3	4	5
음수 인덱스	-6	-5	-4	-3	-2	-1
sales	123.2	342.1	211.2	322.3	125.8	221.2
	sales[0]	sales[1]	sales[2]	sales[3]	sales[4]	sales[5]
	sales[-6]	sales[-5]	sales[-4]	sales[-3]	sales[-2]	sales[-1]

음수 인덱스가 없다면 우리는 리스트의 길이를 알아야 맨 끝의 데이터에 접근할 수 있다. 조금 귀찮은 작업이 된다. 하지만, 음수 인덱스를 이용하여 뒤에서 첫 번째, 뒤에서 세 번째 원소와 같은 개념으로 쉽게 접근할 수 있을 것이다.

파이썬이 제공하는 기본적인 **내장**^{built-in} **함수** `len()`, `min()`, `max()`를 이용하면, 리스트의 길이, 원소들 중 최소값, 혹은 최대값을 쉽게 구할 수 있다. 따라서 아래 코드를 실행하면 `sales` 리스트는 6 개의 원소를 가지고 있고, 그 중에서 가장 작은 값은 123.2, 가장 큰 값은 342.1이라는 것을 알 수 있다.



`len(sales), min(sales), max(sales)`



`(6, 123.2, 342.1)`

리스트는 클래스 객체로서 유용한 메소드를 가지는데, 앞에서 살펴본 `append()` 역시 리스트 클래스의 메소드이다. 리스트의 메소드와 하는 일을 정리하면 다음 표와 같다.

메소드	하는 일
<code>index(x)</code>	원소 x를 이용하여 위치를 찾는 기능을 한다.
<code>append(x)</code>	원소 x를 리스트의 끝에 추가한다.
<code>count(x)</code>	리스트 내에서 x 원소의 개수를 반환한다.
<code>extend([x1, x2])</code>	[x1, x2] 리스트를 기존 리스트에 삽입한다.
<code>insert(index, x)</code>	원하는 index 위치에 x를 추가한다.
<code>remove(x)</code>	x 원소를 리스트에서 삭제한다.
<code>pop(index)</code>	index 위치의 원소를 삭제한 후 반환한다. index 생략시 마지막 원소 삭제.
<code>sort()</code>	값을 오름차순 순서대로 정렬한다. 키워드 인자 <code>reverse=True</code> 는 내림차순.
<code>reverse()</code>	리스트를 원래 원소들의 역순으로 만들어 준다.

슬라이싱을 통해 리스트 일부 잘라 쓰기

리스트에서 여러 원소를 선택해서 새로운 리스트를 만들고 싶다면, **슬라이싱**^{slicing}이라고 하는 기능을 사용할 수 있다. 리스트를 슬라이싱하려면 다음과 같이 콜론을 이용하여 범위를 지정한다. 6 개의 원소를 가진 리스트 sales에서 처음 3 개의 원소만 가져오고 싶다면 0에서 시작해서 3의 앞까지 가져온다.

```
▶ sales = [123.2, 342.1, 211.2, 322.3, 125.8, 221.2]
  first_three = sales[0:3]
  print(sales)
  print(first_three)

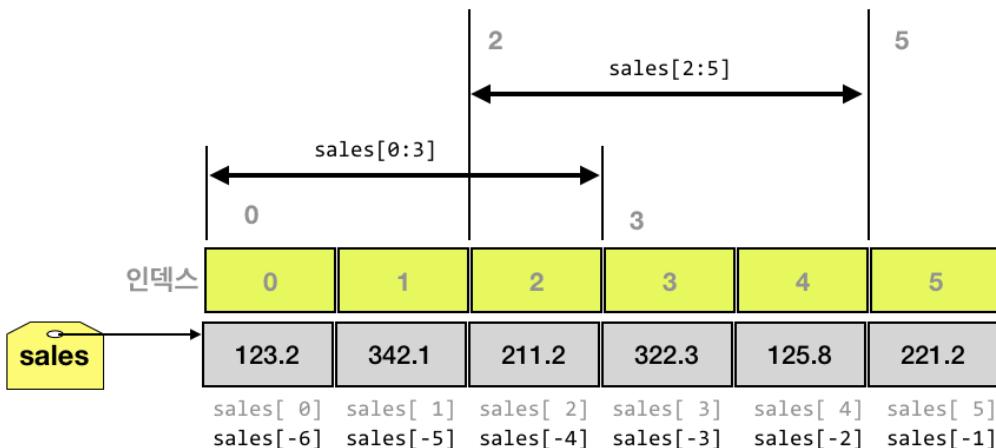
→ [123.2, 342.1, 211.2, 322.3, 125.8, 221.2]
   [123.2, 342.1, 211.2]
```

만약 2번 인덱스부터 3, 4 인덱스까지의 원소를 가지고 싶으면 어떻게 하면 될까? 아래와 같이 [2:5]라고 지정하여 슬라이싱을 하면 될 것이다.

```
▶ sales[2:5]

→ [211.2, 322.3, 125.8]
```

슬라이싱을 위한 인덱스를 지정하는 방법을 시각적으로 설명하면 아래 그림과 같다.



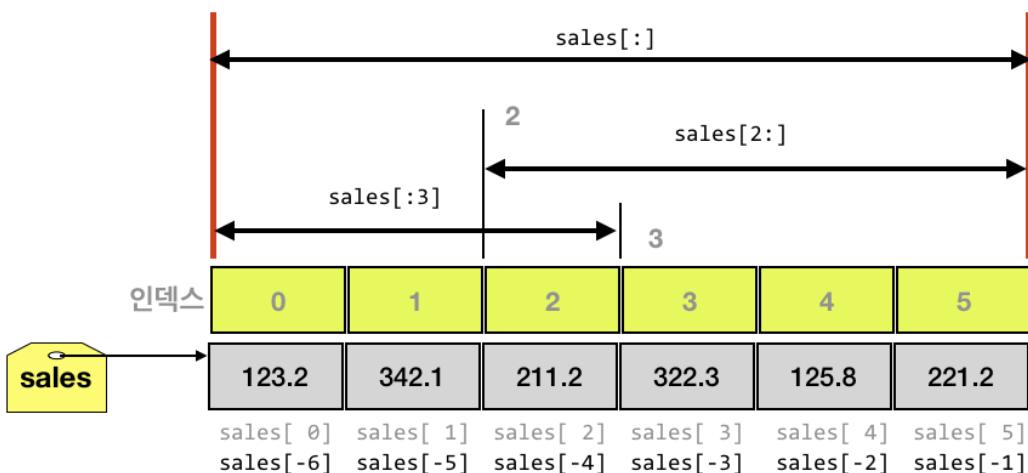
슬라이싱을 할 때 `sales[2:5]`와 같이 지정하면 인덱스 2부터 시작하여 원소들을 추출하다가 인덱스 5인 원소가 나오기 전에 중지한다. 따라서 추출되는 원소의 인덱스는 2,

3, 4가 되며 인덱스 5는 포함되지 않는다는 점에 주의하여야 한다. 슬라이싱에서 추출되는 항목의 개수는 (5-2)과 같이 두 번째 인덱스에서 첫 번째 인덱스를 빼서 계산할 수 있다.

리스트를 슬라이싱하면 원래의 리스트는 손상되지 않으며, 새로운 리스트가 생성되어서 반환된다. 즉 슬라이스는 원래의 리스트의 부분 복사본이다. 슬라이스는 다음과 같이 더 간략하게 표현할 수도 있다. 첫 번째 인덱스를 생략하면 무조건 리스트의 처음부터라고 가정하며, 두 번째 인덱스를 생략하면 리스트의 끝까지를 의미한다. 마지막 인덱스를 굳이 알 필요가 없다는 점에서 이것은 매우 편리한 기능이다. 물론만 있으면 리스트의 처음부터 끝까지라고 생각한다.

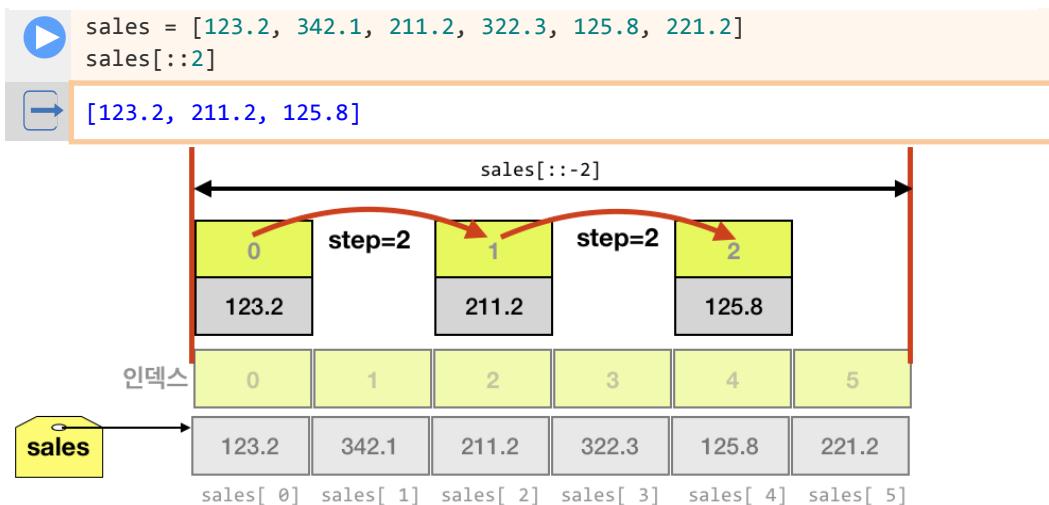
```
▶ print(sales[:3])
    print(sales[2:])
    print(sales[:])
```

➡ [123.2, 342.1, 211.2]
[211.2, 322.3, 125.8, 221.2]
[123.2, 342.1, 211.2, 322.3, 125.8, 221.2]

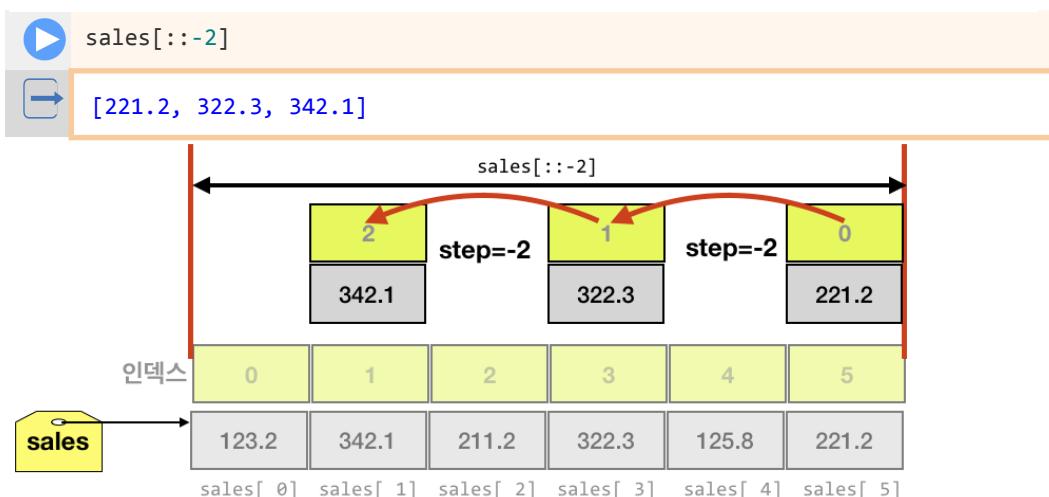


고급 슬라이싱 - 스텝 지정. 음수 스텝과 음수 인덱스 슬라이싱

슬라이싱을 할 때, 시작 인덱스와 마지막 인덱스 다음에 스텝^{step} 값을 세번째로 주어 지정한 스텝만큼 건너뛰며 원소를 추출할 수도 있다. 다음과 같이 `sales[::-2]`는 0번 인덱스에서 시작하여 리스트의 끝 까지 원소를 추출하는 데, 2 칸씩 뛰어가며 값을 가져온다.



음수 스텝 값을 지정할 경우에는 리스트의 원소를 뒤에서부터 읽으면서 추출해 온다.



음수 인덱스는 앞에서 살펴본 바와 같이 양수 인덱스와 대응되는 지점을 고려하면 특별한 어려움 없이 그 결과를 예상할 수 있다.

```
▶ print(sales[2:-2])
print(sales[: -3])
```

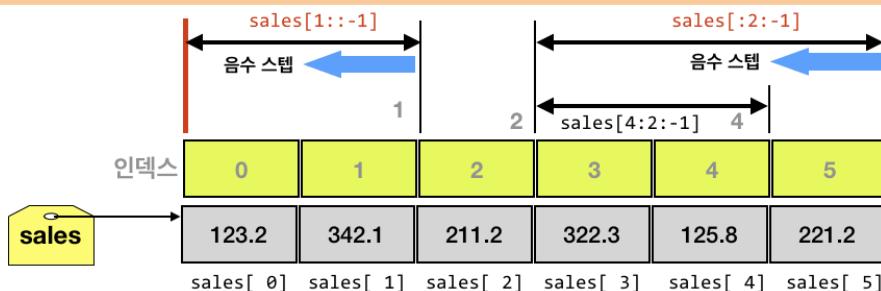
```
→ [211.2, 322.3]
[123.2, 342.1, 211.2]
```



만약 앞에 나타나는 인덱스가 뒤에 나타나는 인덱스보다 큰 값이면 어떻게 될까? 아래와 같이 아무런 원소를 가져오지 못 하기 때문에 빈 리스트가 된다. 그러나 똑같은 인덱스를 사용해도, 스텝을 음수로 주면 어떻게 될까? 아래와 같이 스텝을 -1로 지정하면 원소의 추출을 인덱스 4에서 거꾸로 인덱스 2 방향으로 가기 때문에 원소를 추출할 수 있게 된다. 첫 인덱스를 비우고 음수 스텝을 사용하면 마지막 원소부터 추출을 시작하고, 두 번째 인덱스를 비우고 음수 스텝을 사용하면 첫 원소까지 추출을 시작한다는 점에 유의하라.

```
▶ print(sales[4:2])
print(sales[4:2:-1])
print(sales[:2:-1])
print(sales[1::-1])
```

```
→ []
[125.8, 322.3]
[221.2, 125.8, 322.3]
[342.1, 123.2]
```



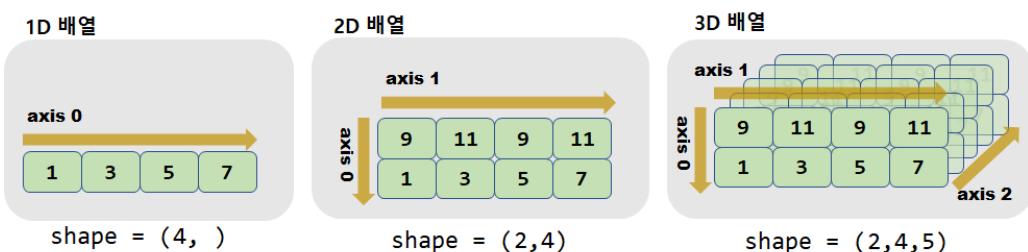
넘파이는 무엇인가?

넘파이([numpy](#))는 파이썬에 사용할 수 있는 패키지이다. 넘파이는 파이썬에서 수치 데이터를 다루는 가장 기본적이고 강력한 패키지라 할 수 있다. 데이터 분석이나 기계 학습 프로젝트를 수행한다면 넘파이에 대한 확실한 이해가 필수적이다. 왜냐하면, 데이터 분석을 위한 패키지인 [판다스 Pandas](#)나 머신러닝을 위한 [Scikit-learn](#), [Tensorflow](#) 등이 넘파이 위에서 작동하기 때문이다.

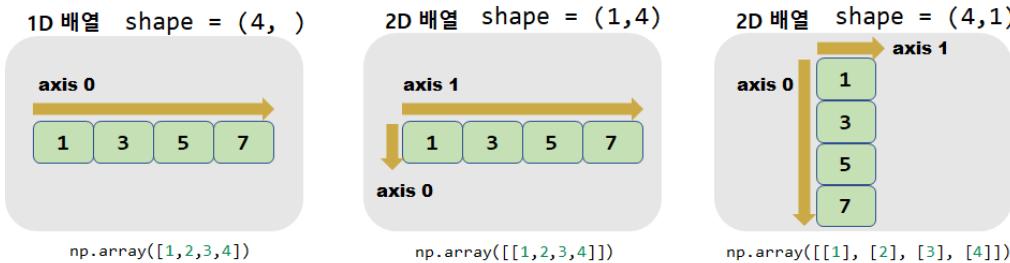
그런데, 이 넘파이와 선형대수는 어떤 관계를 가질까? 넘파이는 다차원 배열을 핵심적인 자료구조로 사용한다. 이 다차원 배열이 선형대수에서 다루는 벡터와 행렬, 텐서를 표현할 수 있다. 넘파이는 이러한 데이터를 잘 다룰뿐만 아니라, 현대적인 컴퓨터 구조를 활용하여 매우 빠른 속도로 이들에 대한 연산을 수행할 수 있다. 넘파이를 사용할 때는 [numpy](#) 모듈을 `import` 한다. 그리고 일반적으로 넘파이 모듈에 대해서는 다음과 같이 `as` 키워드 다음에 나오는 `np`라는 별칭을 지정하여 사용한다.

```
import numpy as np          # numpy의 별칭으로 np를 지정함
my_array = np.array( [ 1, 2, 3 ] )    # np는 numpy의 별칭
```

넘파이의 다차원 배열 각 원소는 [인덱스 index](#)라고 불리는 정수들로 참조된다. 이것은 파이썬의 기본 자료구조인 리스트와 유사하다. 그러나 넘파이 다차원 배열은 리스트와 달리 동일한 자료형을 가진 값으로 구성된다. 넘파이에서 차원은 [축 axis](#)라고도 불린다. 아래 그림은 넘파이의 1차원, 2차원, 3차원 배열의 축과 그 `shape`을 나타내고 있다. 배열의 축은 배열 내부의 원소를 인덱싱하는데 필요하며, 2차원 배열은 2개, 3차원 배열은 3개의 축을 가진다. 배열의 `shape`은 배열 내부의 각 축이 가지는 원소를 보이고 있다.



주의할 점은 `(4,)`와 `(4,1)`, 그리고 `(1,4)`의 차이이다. `(4,)`은 1차원 배열이고, `(4,1)`, `(1,4)`는 모두 2차원 배열이다. 다음 그림으로 그 차이를 잘 이해하자.



2020년 9월 저명한 과학 저널 [네이처](#)^{Nature}에 넘파이에 대한 리뷰 논문이 게재되었다¹. 논문은 [텐서](#)^{tensor}라 불리는 다차원 배열을 효율적으로 다루는 넘파이가 과학 분야에 어떤 기여를 했으며, 과학 각 분야의 수치 데이터 처리 기술들이 [상호운용성](#)^{interoperability}을 가질 수 있도록 하는 역할을 했다고 밝히고 있다.

넘파이가 계산을 쉽고 빠르게 할 수 있는 데에는 이유가 있다. 앞에서 이야기 한 바와 같이 넘파이는 각 배열마다 모든 원소에 대해 동일한 자료형을 사용한다. 즉 정수면 정수, 실수면 실수자료형만을 배열에 저장할 수 있는 것이다. 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 access}라고 한다. 따라서 원하는 데이터에 바로 접근할 수 있기에 빠르게 데이터를 처리할 수 있다. 넘파이의 높은 성능이 임의 접근 때문만은 아니지만, 이것은 넘파이가 빠르게 데이터를 다룰 수 있는 기본적인 원인이 된다.

넘파이가 속도가 빠른 것은 이러한 자료형 때문이기도 하지만, 또 다른 중요한 이유는 하나의 명령을 여러 데이터에 적용하여 병렬적으로 처리하기 때문인데, 이것을 [벡터화 연산](#)^{vectorized operation}이라고 하며, 뒤에 다시 설명할 것이다.

이 문서의 목적은 독자들이 선형대수의 이론을 실제로 구현하고 확인해 보기 위해 넘파이를 이용할 수 있도록 하는 것이다. 넘파이는 강력한 선형대수 서브모듈 `linalg`를 가지고 있는데, 이 `linalg` 서브모듈을 사용하지 않고 직접 구현한 뒤, 그 결과를 `linalg`가 가진 함수의 결과와 비교해 보면 좋을 것이다.

¹ Harris, C.R., Millman, K.J., van der Walt, S.J. et al. (2020) Array programming with NumPy. *Nature* 585, 357–362

넘파이는 리스트와 어떻게 다른가?

우리는 앞에서 파이썬의 리스트에 대하여 학습하였다. 리스트는 여러 개의 값들을 저장할 수 있는 자료구조로서 강력하고 활용도가 높다. 하지만 대규모 수치 데이터를 다루는 과학 분야에서는 파이썬의 기본 리스트가 성능 측면에서 만족스럽지 않다. 이러한 이유로 과학기술 분야에서 수치를 다룰 때는 리스트보다는 넘파이를 선호한다.

리스트와 넘파이의 가장 큰 차이는 계산 성능이다. 넘파이는 대용량의 배열과 행렬연산을 빠르게 수행하며, 고차원적인 수학 연산자와 함수를 포함하고 있다. 넘파이는 성능이 우수한 `ndarray` 객체를 제공한다. `ndarray`는 n차원 배열을 의미한다. 전통적으로 `배열`^{array}은 동일한 자료형을 가진 데이터를 연속으로 저장한다. `ndarray` 객체 역시 동일한 자료형의 항목들만 저장한다. 반면, 파이썬의 리스트는 동일하지 않은 자료형을 가진 항목들을 담을 수 있다. 이러한 차이로 넘파이가 원소에 접근하는 속도가 더 빠르다는 점은 이미 설명하였다.

또 다른 중요한 차이는, 리스트는 데이터를 벡터의 개념으로 해석하지 않고 단순히 여러 데이터가 나열된 것으로만 생각한다. 그러나 넘파이는 배열을 선형대수의 벡터 개념으로 다룬다. 이 차이는 리스트와 넘파이 배열에 대해 덧셈 연산을 적용해서 확인해 보자. 우리가 두 군데 매장을 소유하고 있다고 하고, 각 매장의 지난 3개월 매출이 다음과 같이 `store_a`와 `store_b`라는 리스트에 저장했다고 하자.



```
store_a = [20, 10, 30] # 매장 A의 매출
store_b = [70, 90, 70] # 매장 B의 매출
```

두 매장의 매출을 합하여 지난 3개월 매출을 월별로 확인하고 싶다. 두 리스트를 합하면 어떻게 될까? 아래와 같이 두 리스트는 월별로 합산되지 않고, 연결되어 버린다.



```
list_sum = store_a + store_b
list_sum
```



```
[20, 10, 30, 70, 90, 70]
```

이것은 리스트를 이루는 원소들의 인덱스가 특별한 의미를 갖지 않고 단순히 하나의 리스트 내에서의 순서만을 나타내는 것으로 간주되기 때문이다. 그러나 우리는 첫 번째 원소는 첫 달의 매출, 두 번째 원소는 두 번째 달의 매출이라는 것을 안다. 따라서 두 매장의 매출을

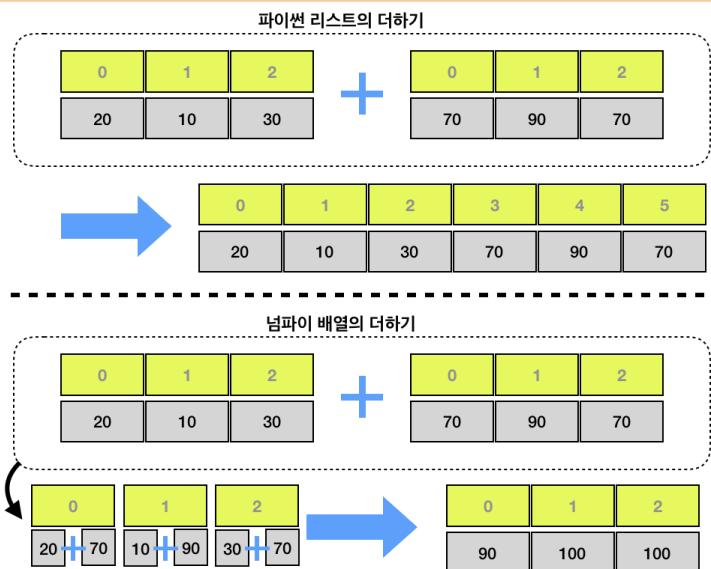
더할 때는 같은 인덱스를 가진 원소들끼리 더해야 의미있는 값을 얻을 수 있다. 넘파이는 대응되는 인덱스들이 더해지고 뺄 수 있는 연관된 데이터라고 가정한다. 선형대수의 언어로 표현하자면 데이터를 벡터로 간주하는 것이다.

실제로 덧셈을 수행해 보자. 넘파이의 `ndarray`는 파이썬의 리스트를 이용하여 쉽게 만들 수 있다. 넘파이가 가진 `array()` 함수에 파이썬 리스트를 넣으면 넘파이 배열을 돌려준다. 이를 이용하여 `store_a`와 `store_b`가 가진 값을 그대로 유지한 넘파이 배열 `np_store_a`와 `np_store_b`를 만들어 보자.

```
 import numpy as np # 넘파이를 임포트하고 np를 별칭으로 지정  
np_store_a = np.array(store_a) # store_a 리스트를 넘파이 배열로 변환  
np_store_b = np.array(store_b) # store_b 리스트를 넘파이 배열로 변환
```

그리고 이 두 배열을 더해서 결과를 확인해 보자. 매달 발생한 두 매장의 매출을 월별로 더한 의미있는 결과를 얻을 수 있을 것이다.

```
 array_sum = np_store_a + np_store_b  
array_sum  
 array([ 90, 100, 100])
```



넘파이 배열의 다양한 속성

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

속성	설명
<code>ndim</code>	배열 축 혹은 차원의 개수.
<code>shape</code>	배열의 차원으로 (m, n) 형식의 튜플. 이 때, m 과 n 은 각 차원의 원소의 크기.
<code>size</code>	배열 원소의 개수이다.
<code>dtype</code>	배열에 담긴 원소의 자료형
<code>itemsize</code>	배열에 담긴 원소의 크기를 바이트 단위로 나타냄
<code>data</code>	배열의 실제 원소들을 가지고 있는 메모리 버퍼
<code>stride</code>	배열 각 차원별로 다음 원소로 점프하는 데에 필요한 거리를 바이트로 표시

여기서 주의해야 할 용어가 차원이다. 우리가 3차원 좌표를 표현할 때 흔히 (x, y, z) 와 같이 세 개의 숫자로 표현한다. 그래서 (x, y, z) 는 3차원 벡터로 표현된 좌표라고 한다. 이것을 넘파이 배열로 표현하면 다음과 같다. x, y, z 변수에 값을 넣어 생성하였다.

```
▶ import numpy as np
x, y, z = 1, 2, 3
a_coord = np.array([x, y, z])
a_coord
[▶ array([1, 2, 3])]
```

이 배열의 `ndim`을 출력해 보자. 3차원으로 표시될까? 그러나 이 배열은 `ndim`이 1로 나온다. 1차원 배열이라는 것이다. 넘파이 배열의 `ndim`이 의미하는 차원은 데이터가 나열된 방향이 몇 개 존재하는가이다. 여기서는 데이터는 한 방향으로 나열되었기 때문에 축이 하나라는 것이다.

```
▶ a_coord.ndim
[▶ 1
```

1차원 배열 (`ndim = 1`)

데이터가 나열된 축
axis 0

그런데 `a_coord`에 담긴 값 (1, 2, 3)은 3차원 데이터이다. 이것은 어떻게 확인할 수 있을까? 다음과 같이 `shape`을 출력해 보자. 여기에는 각 축별로 원소의 개수가 나타난다. 하나의 축만 존재하므로 하나의 숫자만 나올 것이다. 3이다. 즉, 해당 축으로 3 개의 원소를 가진 데이터, 즉 3차원 벡터가 저장되어 있다는 것이다. 그런데 넘파이 배열은 축이 하나일 때에도 다차원 배열임을 표시하기 위해 하나의 숫자 다음에 쉼표가 나타난다.



`a_coord.shape`



(3,)

축^{axis}, 배열의 차원^{dimension} `ndim`, 배열의 형태 `shape`과 저장된 벡터의 차원은 혼동되기 쉬우므로 잘 구분할 수 있도록 하자.

2차원 배열을 만들고 싶으면, 리스트의 리스트를 이용하면 된다. 다음과 같은 모양의 2차원 배열을 만들어 보자.

2차원 배열 (`ndim = 2`)



이것은 축 0 방향으로 존재하는 두 개의 리스트 [1, 2, 3]과 [9, 8, 7]로 리스트를 만들어 넘파이 배열로 바꾸면 된다. 데이터가 나열된 축이 둘이므로 배열의 차원은 2가 되어 `ndim`이 2로 출력된다. 그리고 `shape`을 출력하면 각 축 방향으로 원소가 몇 개인지가 나타난다. 축 0 방향으로 2개씩, 축 1 방향으로 3개씩이므로 (2,3)이 나타난다. 배열의 내용을 출력해보면 우리가 입력한 값들이 그대로 들어가 있다.



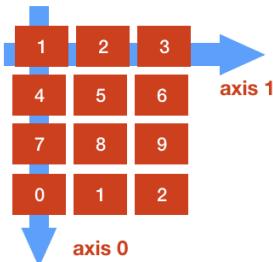
```
arr_2d = np.array([[1, 2, 3],
                   [9, 8, 7]])
print(arr_2d.ndim)
print(arr_2d.shape)
arr_2d.shape
```



2
(2, 3)
array([[1, 2, 3],
 [9, 8, 7]])

넘파이 배열의 인덱싱과 슬라이싱

2차원 배열 (ndim = 2)



리스트에서 사용했던 인덱싱과 슬라이싱은 넘파이 배열에도 거의 동일하게 적용된다. 1차원 배열의 인덱싱과 슬라이싱은 리스트와 큰 차이가 없으므로 2차원 배열을 이용하여 기본적인 인덱싱과 슬라이싱을 연습해 보자. 우선 왼쪽과 같은 2차원 배열을 만들어 보자. 이것은 아래와 같이 리스트의 리스트를 만들어 넘파이의 array() 함수를 이용하여 생성할 수 있다.



```
arr_2d = np.array( [[1, 2, 3], [4, 5, 6],
[7, 8, 9], [0, 1, 2]] )
```

인덱싱을 먼저 수행해 보자. 인덱싱은 두 축에 대해서 모두 가능하다. 첫 번째 축으로 0, 두 번째 축으로 0을 지정하면 가장 첫 원소이 1이 나타날 것이다. 두 번째 축에 대한 인덱싱을 하지 않으면, 첫 번째 축에서 지정된 위치의 모든 데이터가 나타난다.

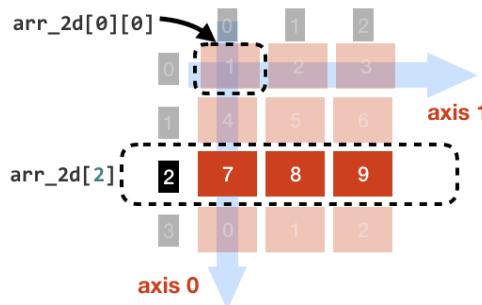


```
arr_2d[0][0]
arr_2d[2]
```



```
1
array([7, 8, 9])
```

2차원 배열 arr_2d



리스트의 슬라이싱과 같은 방법으로 슬라이싱을 할 수도 있다.

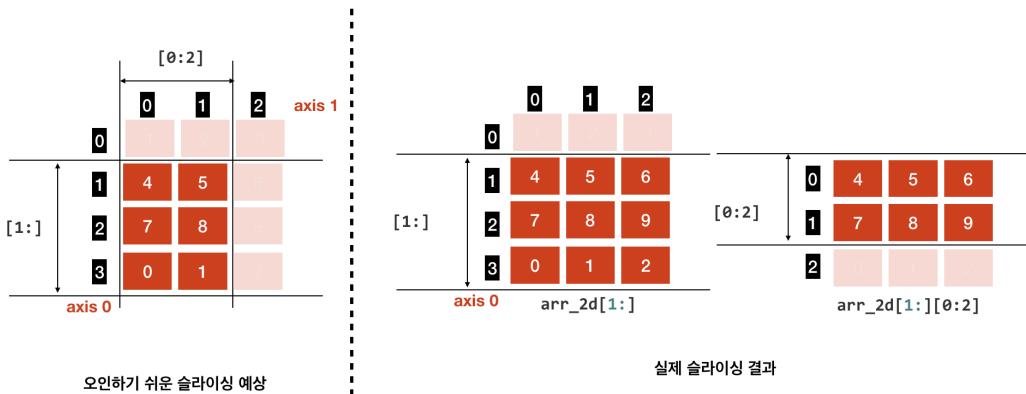


```
arr_2d[1:][0:2]
```



```
array([[4, 5, 6],
[7, 8, 9]])
```

그런데 이 결과가 왜 이렇게 나온 것인지는 조금 생각할 필요가 있다. 착각하기 쉬운 것은 위의 슬라이싱이 아래와 같이 축 0에 대해서는 `[1:0]`으로 슬라이싱하고, 축 1에 대해서는 `[0:2]`으로 슬라이싱해서 아래 그림의 왼쪽과 같이 짙은 색으로 표시된 원소들을 반환할 것으로 오해하는 것이다. 그러나 실제로는 첫번째 슬라이싱이 먼저 일어나서 `arr_2d[1:]`을 만들어내고, 여기에 대해 다시 슬라이싱 `[0:2]`가 적용되어 오른쪽 끝과 같은 결과가 나오는 것이다.



그런데, 사실 위의 그림에서 오인한 바와 같이 슬라이싱을 할 수 있다면 매우 유용하다. 특히 선형대수에서는 가로로 나열된 행과 세로로 나열된 열에 대해서 각각 슬라이싱을 해서 일부를 가져오는 일이 매우 빈번히 요구된다. 따라서 넘파이는 이러한 요구에 맞는 넘파인만의 인덱싱 방법이 있다. `[i][j]`와 같이 원소를 인덱싱하는 것이 아니라 `[i,j]`로 인덱싱하는 것이다. 각 축의 길이를 파악하여 `length0, length1`에 저장하고, 이 값의 범위 내에서 인덱스 `i, j`를 바꾸어 `[i][j]` 인덱싱과 `[i,j]` 인덱싱을 비교해 보자. 동일한 결과가 나온다. 그러면 이 넘파이 스타일의 인덱싱은 특별한 의미가 없어 보인다. 그러나 슬라이싱을 통해 그 차이를 알 수 있다. 다음 절에서 살펴 보자.

```

▶ length0, length1 = arr_2d.shape
for i in range(length0):
    for j in range(length1):
        print(arr_2d[i][j], arr_2d[i, j])
→
  1 1
  2 2
  ...
  0 0
  1 1
  2 2

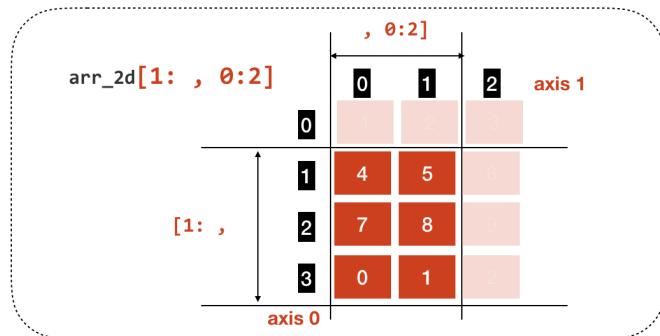
```

넘파이 스타일의 슬라이싱과 논리 인덱싱

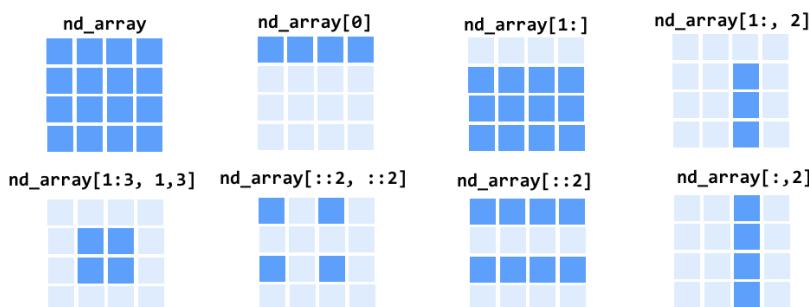
넘파이 스타일의 인덱싱을 이용하여 슬라이싱을 하면 각각의 슬라이싱 범위는 리스트의 슬라이싱처럼 순차적으로 적용되는 것이 아니라, 각각의 축에 별도로 적용된다. 따라서 다음과 같은 슬라이싱이 가능하다.

```
▶ arr_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9],
                     [0, 1, 2]])
    arr_2d[1: , 0:2]
▶ array([[4, 5],
         [7, 8],
         [0, 1]])
```

넘파이 스타일의 슬라이싱



이것은 행과 열로 이루어진 데이터에서 특정한 행들과 열만 추려서 뽑아내는 데에 매우 유용한 방식이며, 선형대수를 구현하는 데에 매우 중요한 기능이기도 하다. 넘파이 스타일의 슬라이싱은 큰 행렬에서 작은 행렬을 끄집어내는 것으로 이해하면 된다. 다음은 2차원 행렬의 일부를 추출하는 코드이다. 아래의 그림은 4x4 크기의 `np_array`라는 ndarray와 이 ndarray의 인덱싱 및 슬라이싱 결과를 보여준다.



넘파이는 데이터를 추려내는 데에 슬라이싱 뿐만 아니라 **논리 인덱싱**^{logical indexing}이란 효율적인 방법도 제공한다. 이것은 특정한 어떤 조건을 주어서 배열에서 원하는 값을 추려내는 것이다. 넘파이 배열에 조건을 주면 불리언 배열이 만들어진다. 2차원 배열을 이용하여 확인해 보자. 아래의 예는 16 개의 원소를 가진 2차원 넘파이 배열을 만들었다. 그리고, 동일한 크기의 배열인데, 각 원소가 “2의 배수”인 경우에 참이 되는 불리언 배열을 다음과 같이 만들 수 있다. 방법은 넘파이 배열을 조건식에 넣기만 하면 된다.



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



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

이렇게 얻어진 배열을 이용하여 인덱스로 사용하면 해당 조건에 부합하는 원소만 추출할 수 있다. 이렇게 할 경우 **np_array** 배열내의 모든 원소들 중에서 참인 위치의 값만 추출된다.

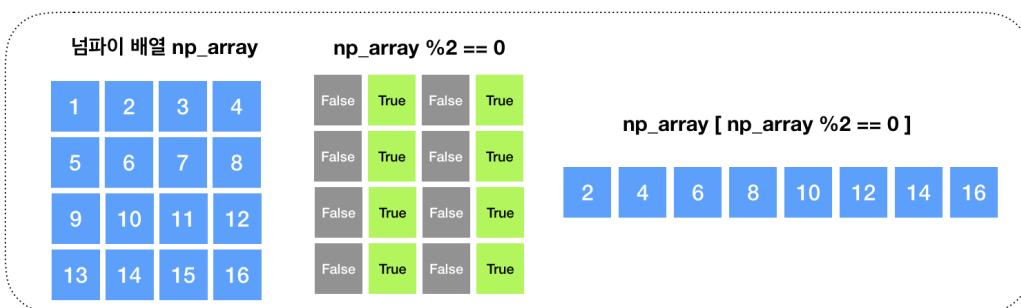


```
np_array[ np_array % 2 == 0 ]
```



```
array([ 2,  4,  6,  8, 10, 12, 14, 16])
```

이 동작을 시각적으로 표시하면 다음과 같은 그림으로 설명할 수 있다.



이상으로 파이썬의 리스트와 넘파이의 다차원 배열을 다루는 법을 살펴 보았다. 넘파이의 다차원 배열이 선형대수에서 다루는 벡터와 행렬을 다루기에 더 적합한 기능들을 가지고 있는 것을 확인할 수 있었다. 그런데, 이러한 장점 뿐만 아니라 넘파이는 다양한 수치 데이터를 고속으로 처리할 수 있는 데이터 병렬 처리 특성을 갖고 있다. 이에 대해서는 다음 장에서 살펴 보자.

넘파이는 왜 성공했나

넘파이가 앞 장에서 살펴본 바와 같이 다양한 수치 데이터를 다루고, 일부를 추출하는 데에 효율적인 기능을 제공하고 있지만, 이것만으로 넘파이의 성공 이유를 모두 설명할 수는 없다. 넘파이는 사실 강력한 **배열 프로그래밍**^{array programming} 기능 덕분에 오늘의 인기를 누리게 되었다. 배열 프로그래밍은 데이터에 연산을 실시할 때, 데이터를 구성하는 값 전체에 대해 한 번에 연산이 적용되도록 방식을 의미한다. 이러한 해법을 제공하지 못하는 언어를 **스칼라**^{scalar} 언어라고 하며, 전통적인 C, C++ 같은 스칼라 언어에 해당한다.

배열 프로그래밍은 데이터를 다루는 연산의 표현을 간결하게 만든다. 예를 들어 두 개의 배열 `arr_a`와 `arr_b`가 있고, 두 배열의 원소 개수는 `arr_len`이라고 가정해 보자. 스칼라 언어를 사용한다면 이 두 배열을 더하는 코드는 다음과 같은 의사코드로 표현될 것이다.

```
For i from 0 to arr_len-1
    arr_sum[i] = arr_a[i] + arr_b[j]
end For
```

그러나 배열 프로그래밍을 지원하는 언어는 다음과 같은 표현을 지원한다.

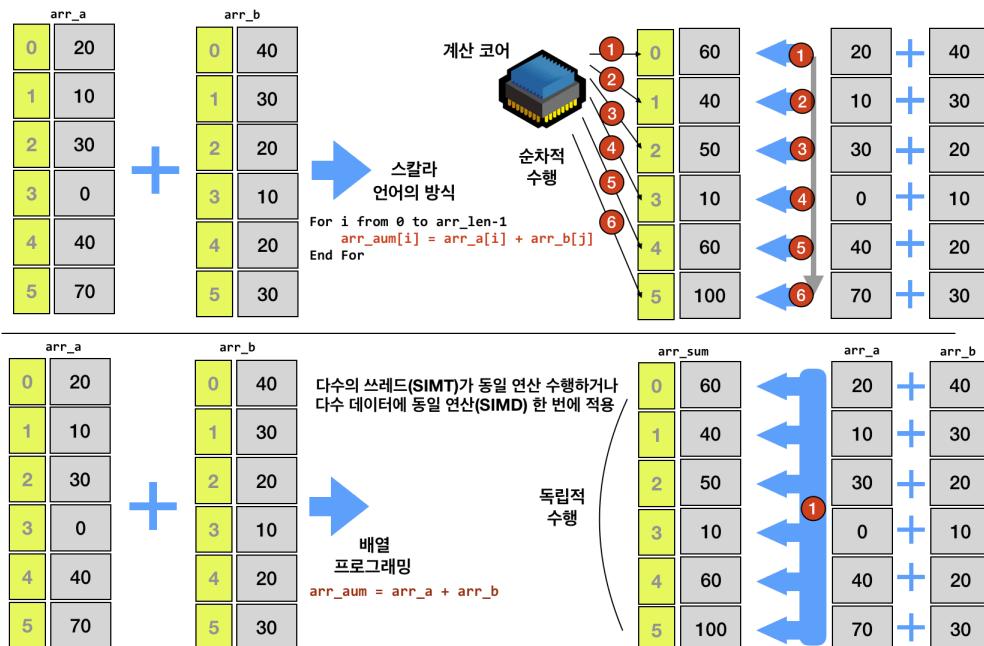
```
arr_sum = arr_a + arr_b
```

위와 같은 표현은 연산의 대상이 되는 **피연산자**^{operand}가 다수의 원소를 가진 배열, 수학적으로 여러 수로 이루어진 벡터 데이터라는 것을 가정하기 때문에 가능하다. 여기서 더하기 연산자는 배열과 배열, 수학적 표현으로는 벡터와 벡터의 더하기를 의미하는 것이다. 이러한 이유로 배열 프로그래밍에서 사용하는 연산을 **벡터화 연산**^{vectorized operation}이라고 한다.

물론 표현의 간결성만이 배열 프로그래밍의 장점이 아니다. 배열 프로그래밍이 사용하는 벡터화 연산은 강렬한 병렬 처리가 뒷받침될 때 완성된다. 즉 For 반복문을 사용한 스칼라 언어가 각각의 원소들을 순차적으로 처리해야 한다면, 벡터화 연산은 두 배열의 대응하는 원소를 서로 더해서 새로운 배열에 저장하는 일을 차례를 기다리지 않고 동시에 수행할 수 있다. 이렇게 데이터를 잘게 쪼개어 각각에 대해 동일한 연산을 독립적으로 수행하는 방식을 **데이터 병렬성**^{data parallelism}이라고 한다.

프로그래밍에서 문제를 해결할 때 사용하는 병렬성은 크게 두 가지 유형으로 구분된다. 하나는 전통적 병렬 프로그래밍의 주제인 **작업 병렬성 task parallelism**이다. 이것은 문제를 해결하는 방법을 이루는 작은 작업들 사이의 의존성을 분석한 뒤 동시에 처리할 수 있는 일은 여러 개의 계산 코어 core가 동시에 진행하는 방식이다. 또 다른 방식은 처리해야 하는 데이터를 잘게 쪼개어 각각에 대해 동일한 작업을 여러 개의 코어가 동시에 진행하는 방식이다. 이것을 데이터 병렬성이라고 부른다. 이 방식은 엄청나게 많은 핵심을 고려해야 하지만, 각각의 핵심을 그리는 데에 필요한 작업은 동일한 절차를 따르면 되는 컴퓨터 그래픽스와 같은 분야에 적합하다. 이러한 이유로 **그래픽 처리 장치 GPU**는 고성능 데이터 병렬 처리기로 발전을 하여 현재 슈퍼컴퓨팅 자원으로 적극 활용되고 있는 것이다.

GPU는 다수의 **쓰레드 thread**를 만들어 동일한 연산을 수행하게 하는데, 이러한 구조를 **단일 명령어 다중 쓰레드 single instruction multiple thread:SIMT** 방식이라고 한다. CPU는 GPU와 달리 쓰레드를 생성하지 않고, 하나의 연산이 다수의 데이터에 동시에 적용되도록 하는 **단일 명령어 다중 데이터 single instruction multiple data:SIMD** 방식을 사용한다. 전통적인 언어로도 이러한 데이터 병렬성을 활용할 수 있지만, 벡터화 연산을 그대로 표현하는 효율적인 코딩은 어렵다. 넘파이는 강력하고 압축적인 프로그래밍 구문을 제공하며, 프로그래머가 효율적으로 코드로 고성능의 데이터 병렬 처리가 동시에 가능하다. 넘파이의 성공 이유로 꼽아야 할 가장 중요한 요소가 바로 이것이다.



벡터화 연산의 성능

벡터화 연산은 하나의 명령이 여러 데이터에 동시에 적용된 방식으로 **배열 프로그래밍**^{array programming}의 핵심 요소이다. 큰 작업을 분할하여 다수의 **프로세서**^{processor}에서 나누어 처리하는 전통적 병렬 처리와 달리 하나의 프로세서 내에서 병렬적 데이터 처리가 **묵시적**^{implicit}으로 이루어진다. 이러한 방식을 지원하는 구조를 단일 명령으로 여러 데이터 처리를 한다는 의미로 **SIMD**^{single instruction multiple data} 방식이라고 하는데, 현재 대다수의 CPU가 SIMD 방식을 지원하고 있고, 넘파이는 따로 설정하지 않아도 이를 활용한다.

넘파이의 배열 프로그래밍 성능 측정을 위해 `time` 모듈도 함께 임포트한다. 그리고 100 개의 원소를 가진 두 개의 배열 `arr_a`와 `arr_b`를 만들고 난수를 채워보자. 그리고 이 두 배열을 연산하여 그 결과를 담을 배열 `result`를 만들고, 여기에는 100 개의 0을 채워 보자.

```
▶ import numpy as np
import time
arr_a = np.random.rand(100)
arr_b = np.random.rand(100)
result = np.zeros(100)
```

두 배열을 원소별로 곱하는 일은 전통적인 `for` 반복문을 이용하여 다음과 같이 할 수 있을 것이다. 이때 작업을 시작하기 전의 시간과 작업 종료 이후의 시간을 기록하여 차이를 구하면 계산에 소요된 시간을 알 수 있다.

```
▶ start = time.time()
for i in range(len(values)):
    result[i] = arr_a[i] * arr_b[i]
end = time.time()                                # 명시적인 작업 지시 : C 스타일
print('소요시간: ', end - start)

▶ 소요시간:  0.0012476444244384766
```

100 개의 원소를 가진 두 배열을 원소별로 곱하여 `result`에 담는 데에는 물론 오랜 시간이 걸리지 않고, 아무런 오류 없이 잘 동작하지만, 벡터화 연산을 사용하여 다시 써보자.

```
▶ start = time.time()
result = arr_a * arr_b                         # 벡터화 연산 *
end = time.time()
print('소요시간: ', end - start)

▶ 소요시간:  0.0000588893890380859
```

계산 시간은 대략 20 배 정도 단축되었다. 이때 연산자 *는 벡터화 연산자로 배열의 모든 원소에 대해 한꺼번에 적용된다. 이 연산자 뿐만 아니라 사칙 연산 모두와 나중에 다룰 행렬곱 연산 등이 모두 벡터화 연산으로 동작한다. 배열의 크기는 100,000개로 늘려 보자.

```
 n_elem = 100000      # 원소의 개수
arr_a = np.random.rand(n_elem)
arr_b = np.random.rand(n_elem)
result = np.zeros(n_elem)
```

아래와 같이 100,000 개의 원소를 가진 배열에 같은 연산을 하고 비교해 보자. 연산을 수행하는 데에 걸리는 시간은 60배 정도 차이가 나는 걸 알 수 있다.

```
 start = time.time()
for i in range(n_elem):
    result[i] = arr_a[i] * arr_b[i]
end = time.time()
print('원소별 접근 방식 소요시간: ', end - start)

start = time.time()
result = arr_a * arr_b
end = time.time()
print('벡터화 연산 수행 소요시간: ', end - start)
```

 원소별 접근 방식 소요시간: 0.05570673942565918
 벡터화 연산 수행 소요시간: 0.0009524822235107422

1차원 배열만 예를 들었지만, 2차원 이상의 고차원 배열에 대해서도 동일하게 원소별로 연산이 SIMD 방식으로 적용된다.

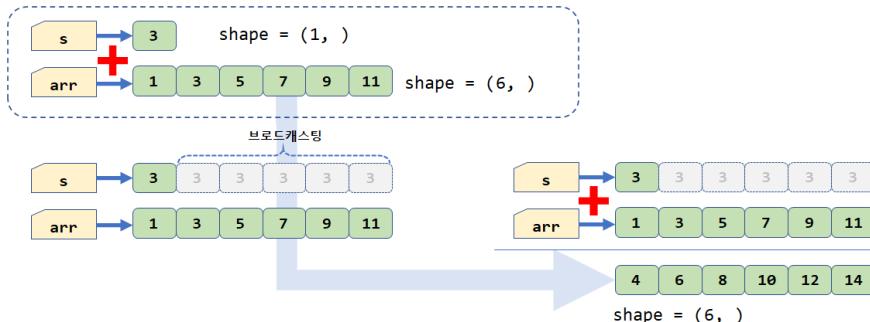
위와 같이 1차원 배열에서 원소별 접근을 했을 때, 벡터화 연산을 수행한 것에 비해 몇 배의 시간이 걸리는지를 성능값으로 제시하여 비교한 결과가 다음과 같다. K는 1,000개, M은 1,000,000개를 의미한다. 일정한 규모 이상에서 벡터화 연산을 사용한 것이 100 배에서 200 배까지 빠른 것을 확인할 수 있다.

N	10K	20K	40K	100K	200K	400K	1M	2M	3M	4M	5M
성능	46	148	126	143	150	179	187	195	177	182	206

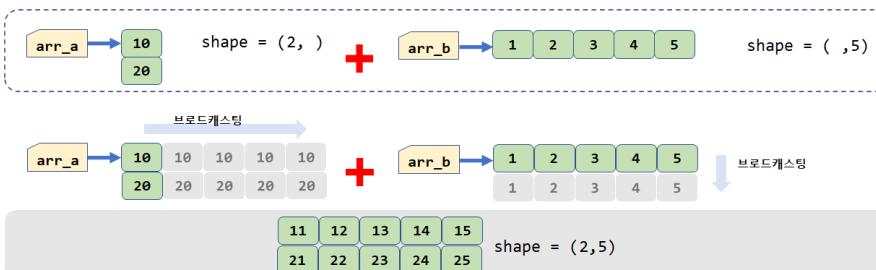
브로드캐스팅과 성능 개선

벡터화 연산은 두 피연산자 동일한 `shape`을 갖고 있을 때 대응되는 원소들에 대해 각각 연산이 적용된다. 그런데, **브로드캐스팅**^{broadcasting}은 이러한 벡터화 연산이 다른 `shape`을 가지고 있을 때에도 연산이 가능하게 만들어 준다.

가장 단순한 예인 1차원 배열 `arr`에 하나의 수 `s`를 더해 보자. 브로드캐스팅은 벡터화 연산으로 이 덧셈을 수행하기 위해 이 두 데이터의 차원과 축별 원소의 개수를 일치시킨다. 이 경우에는 `s`가 아래 그림과 같이 `arr`와 같은 차원의 배열처럼 다루어지며, 원래 가지고 있던 하나의 값이 원소들 모두에 적용되는 것이다. 이런 모양이라면 벡터화 연산이 적용되는 과정은 간단하다. 그런데 실제로 `s`를 배열로 만들어 데이터를 복사하는 일을 수행하지 않고, `s`가 가진 하나의 값이 `arr`의 각 원소에 적용되기 때문에 데이터 복사에 필요한 계산 시간을 단축시킬 수 있다.



이것은 물론 2차원 이상의 고차원 배열에 대해서도 적용이 가능하다. 세로로 한 개의 열을 가진 배열과, 한 개의 행을 가진 배열을 더하면 다음과 같은 결과를 얻는다.



두 배열을 더할 때, 벡터화 연산과 브로드캐스팅 없이 구해 보고, 다시 넘파이의 브로드캐스팅과 벡터화 연산을 적용하여 성능 개선을 확인해 보자. 아래와 같이 두 배열을 `arr_a`, `arr_b`로 만들었다. 이들은 각각 $(n, 1)$ 과 $(1, n)$ 의 크기로 생성되었다. 즉 `arr_a`는 세로로 긴 배열이고, `arr_b`는 가로로 긴 배열이다. 각각 1,000 개의 원소를 가진다.

```
 import numpy as np
import time

n = 1000
arr_a = np.random.rand(n, 1)
arr_b = np.random.rand(1, n)
print(arr_a.shape, arr_b.shape)
```

덧셈 연산에서 원소별 대응이 가능하도록 두 배열을 확장시킨 arr_a_ex와 arr_b_ex를 만들고, 이들의 합을 res_naive에 계산하는 방법은 다음과 같다. 시작 시간과 종료 시간을 측정해 소요시간을 출력해 보았다. 상당한 시간이 소요되는 것을 확인할 수 있다.

```
 # 브로드캐스팅과 벡터화 없이 더하기
start = time.time()                                # 연산 시작 시간
res_naive = np.empty((n, n))                      # 결과 저장 공간
arr_a_ex = np.empty((n, n))                        # 차원을 일치시킴
arr_b_ex = np.empty((n, n))                        # 차원을 일치시킴
for i in range(0, n):
    for j in range(0, n):
        arr_a_ex[j,i] = arr_a[j, 0]                # 확장된 공간 채움
        arr_b_ex[i,j] = arr_b[0, j]                # 확장된 공간 채움

for i in range(n):
    for j in range(n):
        # 두 배열의 원소별 합
        res_naive[i, j] = arr_a_ex[i,j] + arr_b_ex[i, j]
end = time.time()                                    # 연산 종료 시간
print(n, 'x', n, ":", end-start)
```

 1000 x 1000 : 1.60617995262146

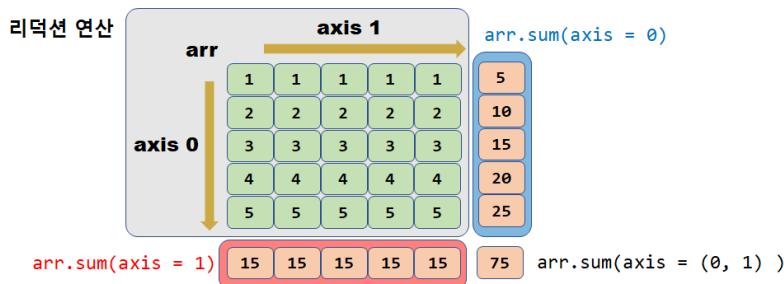
이제 브로드캐스팅과 벡터화 연산이 적용되도록 해 보자. 오히려 코드는 훨씬 더 단순해진다. 원래의 두 배열을 더해 새로운 배열에 넣기만 하면 된다. 배열을 채우고, 원소별 접근을 하지 않기 때문에 시간은 10^3 규모로 빨라졌다. 데이터의 수에 따라 더욱 큰 차이가 날 수도 있다. 고성능 고차원 배열에서 브로드캐스팅과 벡터화 연산은 필수이다.

```
 # 브로드캐스팅과 벡터화 연산으로 더하기
start = time.time()                                # 연산 시작 시간
res = arr_a + arr_b                                # 브로드캐스팅과 벡터화 연산 적용
end = time.time()                                    # 연산 종료 시간
print(n, 'x', n, ":", end-start)
```

 1000 x 1000 : 0.0021255016326904297

리덕션

넘파이 배열을 강력한 **리덕션**^{reduction} 기능을 제공한다. 리덕션이라는 것은 여러 개의 수로 이루어진 배열을 하나의 수로 **집계**^{aggregation}하는 것이다. 넘파이는 이 집계를 위한 함수로 **sum()**, **mean()**, **min()**, **max()** 같은 함수를 제공한다. 각각은 대상 배열의 원소에 대해 총합, 평균, 최소값, 최대값 등을 찾는 일을 한다. 물론 임의의 집계 함수를 만들 수도 있다. 이 리덕션 기능은 벡터화 연산 방식으로 적용되어 높은 성능을 보여 준다. 넘파이의 리덕션은 적용되는 축을 지정하여 이루어진다. 다음과 같이 2차원 배열이 있다면, 축은 0 번과 1 번의 두 개가 존재한다. **sum()**을 적용할 때 **axis=0** 혹은 **axis=1**을 적용하면 각각 해당 축 방향으로 원소의 개수를 유지하며 리덕션을 수행한다. 축을 두 개 모두 주면 전체 리덕션이 이루어질 것이다.



실제 코드를 통해 이 연산을 수행해 보자. 위의 그림과 같은 결과가 나온 것을 확인할 수 있다.

```
▶ arr = np.array([[1,2,3,4,5],
                  [1,2,3,4,5],
                  [1,2,3,4,5],
                  [1,2,3,4,5],
                  [1,2,3,4,5],
                  ])
print(np.sum(arr, axis = 0))
print(np.sum(arr, axis = 1))
np.sum(arr, axis=(0,1))
```

```
[ 5 10 15 20 25]
[15 15 15 15 15]
75
```

리덕션 연산은 벡터화 연산을 통해 이루어지므로 높은 성능을 보여준다. 큰 규모의 배열을 만들고 각 원소의 합을 원소별 접근을 통해서 계산하는 방법과 리덕션 기법을 이용한 방법으로 수행해 보고, 소요 시간을 비교해 보자.

```
 import numpy as np
import time

n = 1000
arr = np.random.rand(n, n)
```

이 배열은 총 1,000,000 개의 원소를 가진다. 배열 원소의 합은 모든 원소에 접근해 그 값을 변수 `sum_naive`에 누적하면 된다. 결과와 소요 시간을 출력해 보자. 구글 코랩에서 0.4초 정도가 걸렸다.

```
 sum_naive = 0
start = time.time()
for i in range(n):
    for j in range(n):
        sum_naive += arr[i,j]
end = time.time()
print(n, 'x', n, "배열의 원소 합", sum_naive, "계산 시간: ", end-start)

 1000 x 1000 배열의 원소 합 500048.24990126066
계산 시간: 0.39148378372192383
```

이것을 벡터화 연산이 적용되는 리덕션으로 구해보자. 아래와 같이 구현할 수 있다. 원소의 합은 동일한 값이 출력되고, 소요시간은 0.0013초로 300배 이상 빨라진 것을 확인할 수 있다.

```
 start = time.time()
sum_reduction = np.sum(arr, axis = (0,1))
end = time.time()
print(n, 'x', n, "배열의 원소 합", sum_reduction, "계산 시간: ",
end-start)

 1000 x 1000 배열의 원소 합 500048.2499012628
계산 시간: 0.00128173828125
```

지금까지 살펴본 바와 같이, 넘파이는 벡터화 연산, 브로드캐스팅, 그리고 리덕션을 통해 대규모 수치 배열에 대해 높은 성능의 계산을 제공한다. 이를 잘 활용하기 위해서는 스칼라 언어에서 사용하는 원소별 접근, 행이나 열 단위 접근을 최소한으로 사용해야 한다. 넘파이 배열을 다루면서 `for` 반복문을 사용한다면, 우선 이것은 성능을 저하시키지 않는지 의심하고 살펴볼 필요가 있다.