Zen of NumPy

Sungjoo Ha

TensorFlow-KR 첫 오프라인 모임

June 18th, 2016

Zen of Python[1]

아름다움이 추함보다 좋다.
명시가 암시보다 좋다.
단순함이 복잡함보다 좋다.
복잡함이 꼬인 것보다 좋다.
수평이 계층보다 좋다.
여유로운 것이 밀집한 것보다 좋다.
가독성은 중요하다.
특별한 경우라는 것은 규칙을 어겨야 할 정도로 특별한 것이 아니다.
허나 실용성은 순수성에 우선한다.
오류 앞에서 절대 침묵하지 말지어다.
명시적으로 오류를 감추려는 의도가 아니라면.
모호함을 앞에 두고, 이를 유추하겠다는 유혹을 버려라.
어떤 일에든 명확한 - 바람직하며 유일한 - 방법이 존재한다.
비록 그대가 우둔하여 그 방법이 처음에는 명확해 보이지 않을지라도.
지금 하는게 아예 안하는 것보다 낫다.
아예 안하는 것이 지금 *당장*보다 나을 때도 있지만.
구현 결과를 설명하기 어렵다면, 그 아이디어는 나쁘다.
구현 결과를 설명하기 쉽다면, 그 아이디어는 좋은 아이디어일 수 있다.
네임스페이스는 대박 좋은 아이디어다 -- 마구 남용해라!

---

[1] PEP #20 https://bitbucket.org/sk8erchoi/peps-korean

Zen of NumPy[2]

등간격이 흩어진 것보다 낫다.
Strided is better than scattered
연속된 것이 등간격보다 낫다.
Contiguous is better than strided
원하는 것을 설명하는 편이 명령을 내리는 것보다 낫다 (데이터 타입을 사용하자).
Descriptive is better than imperative (use data-types)
배열 지향이 대체로 객체 지향보다 낫다.
Array-oriented is often better than object-oriented
브로드캐스팅은 좋은 아이디어다 – 가능하면 사용하자.
Broadcasting is a great idea – use where possible
벡터화한 것이 명시적인 루프보다 좋다.
Vectorized is better than an explicit loop
하지만 복잡하다면 numexpr, weave나 Cython을 사용해라.
Unless it's complicated — then use numexpr, weave, or Cython
고차원에서 생각하라.
Think in higher dimensions

[2] https://github.com/numpy/numpy/issues/2389

NumPy?

- 배열을 활용한 **효율적** 계산을 위한 라이브러리
- 많은 과학 계산 라이브러리가 NumPy를 기반으로 둠
  - scipy, matplotlib, pandas, scikit-learn, statsmodels, etc.
  - 라이브러리 간의 공통 인터페이스

NumPy and TensorFlow

- TensorFlow는 NumPy의 `ndarray` 기능에 약간의 추가적인 기능과 자동 미분[3] + GPU 지원을 추가한 라이브러리로 생각할 수도 있음
  - NumPy의 동작을 이해하면 TensorFlow의 방식 이해에 도움됨
  - autograd[4] 등의 라이브러리를 사용하면 NumPy 코드만으로 자동 미분도 가능

[3] `https://en.wikipedia.org/wiki/Automatic_differentiation`

[4] `https://github.com/HIPS/autograd`

효율성의 기반

NumPy의 효율성은 . . .

- 데이터 저장 방식
- 데이터 접근 방식
- 벡터화된 연산

의 결과물이다.

파이썬 리스트 구현

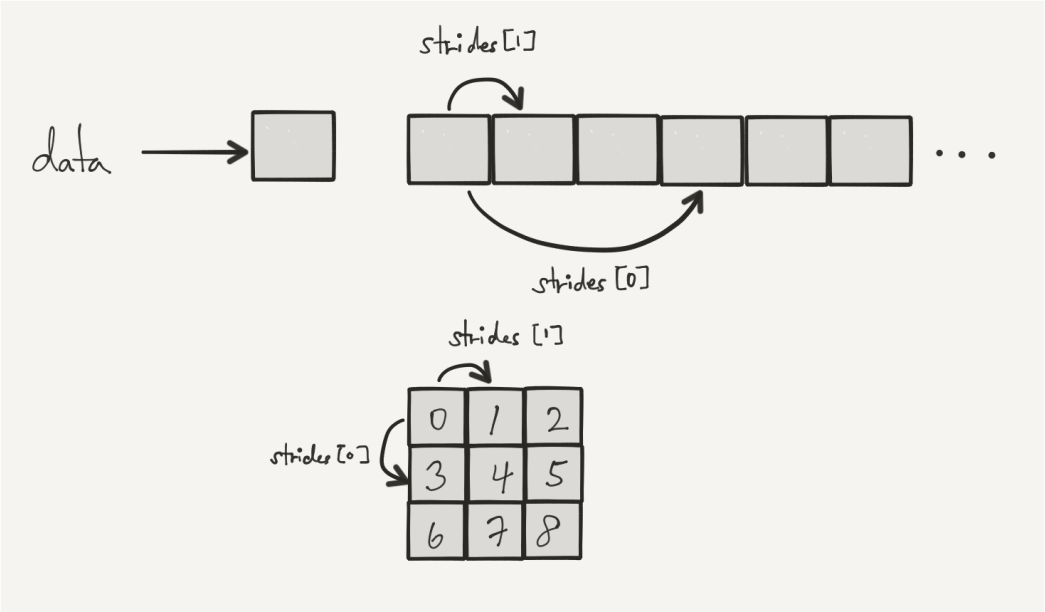
```
typedef struct {
    PyObject_HEAD
    Py_ssize_t ob_size;

    /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
    PyObject **ob_item;

    /* ob_item contains space for 'allocated' elements.  The number
     * currently in use is ob_size.
     * Invariants:
     *     0 <= ob_size <= allocated
     *     len(list) == ob_size
     *     ob_item == NULL implies ob_size == allocated == 0
     */
    Py_ssize_t allocated;
} PyListObject;
```

파이썬 리스트가 느린 이유

- 파이썬 리스트는 결국 포인터의 배열
- 경우에 따라서 각각 객체가 메모리 여기저기 흩어져 있음
- 그러므로 캐시 활용이 어려움

NumPy ndarray 구현

```
data : 4297514880
shape : (3, 3)
strides : (6, 2)
dtype : uint16
```

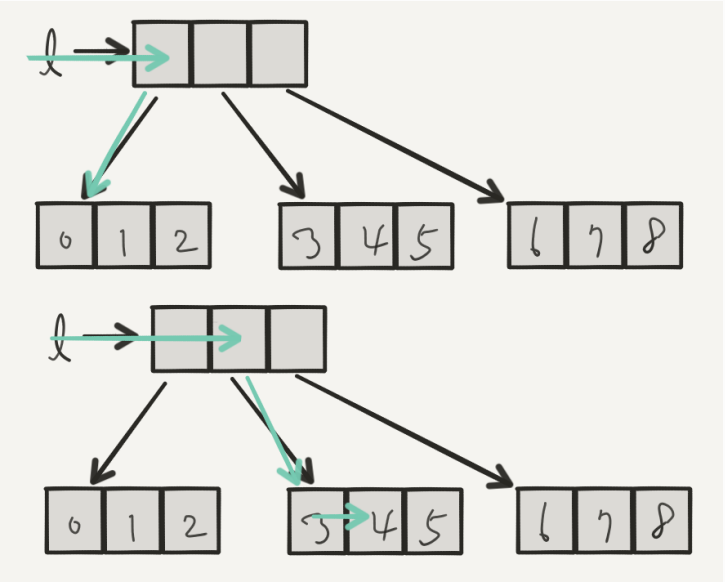


NumPy ndarray가 빠른 이유

- `ndarray`는 타입을 명시하여 원소의 배열로 데이터를 유지
- 다차원 데이터도 연속된 메모리 공간이 할당됨
- 많은 연산이 `strides`를 잘 활용하면 효율적으로 가능
  - 가령 transpose는 `strides`를 바꾸는 것으로 거의 공짜
- `ndarray` **구현 방식을 떠올리면 어떻게 성능을 낼 수 있는지 상상 가능**

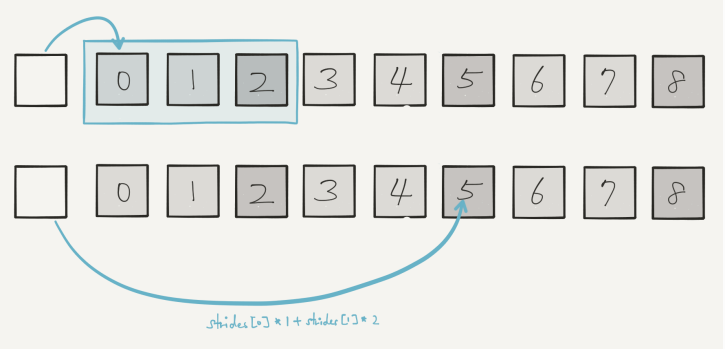
리스트 인덱싱

```
>>> l = [[0, 1, 2], [3, 4, 5],
... [6, 7, 8]]
>>> l
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> l[0]
[0, 1, 2]
>>> l[1][2]
5
```



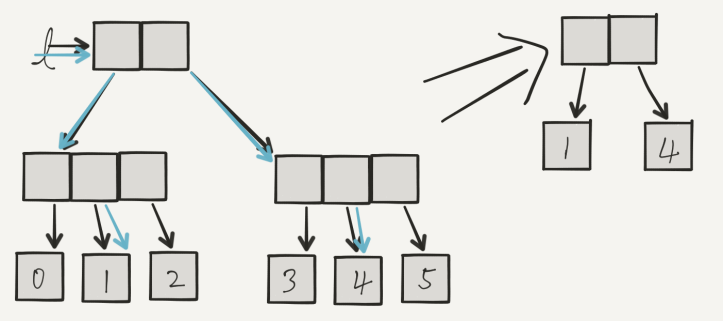
NumPy 인덱싱

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



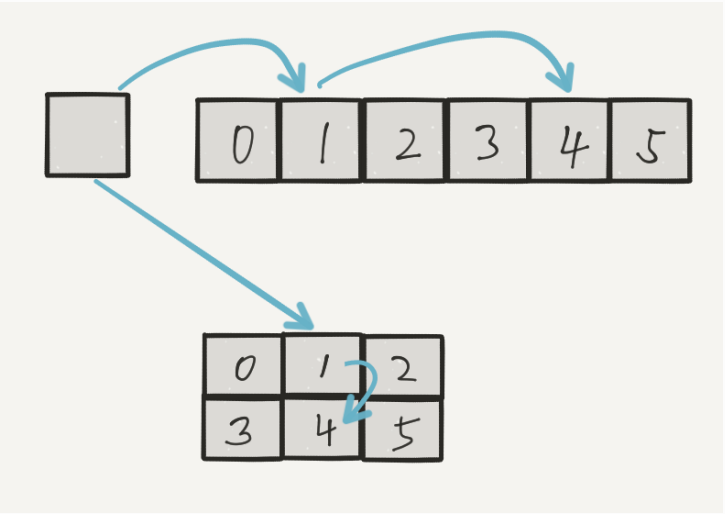
파이썬 슬라이싱

```
>>> l = [[0, 1, 2], [3, 4, 5]]
>>> [l[i][1] for i, j in enumerate(l)]
[1, 4]
```



NumPy 슬라이싱

```
>>> a = np.arange(6).reshape(2, 3)
>>> a[:, 1]
array([1, 4])
```



연속된 접근과 등간격 접근

- 인덱싱 혹은 슬라이싱 시 데이터의 복사 없음
  - 임의의 타입을 다뤘다면 불가능
  - 원하는 것을 설명하는 편이 명령을 내리는 것보다 낫다 (데이터 타입을 사용하자)
  - 배열 지향이 대체로 객체 지향보다 낫다
- 등간격 접근과 연속된 데이터 접근 모두 캐시를 활용
- 하지만 연속된 데이터를 가져오는 것이 더 효율적
  - 연속된 것이 등간격보다 낫다

인덱싱 그리고 슬라이싱

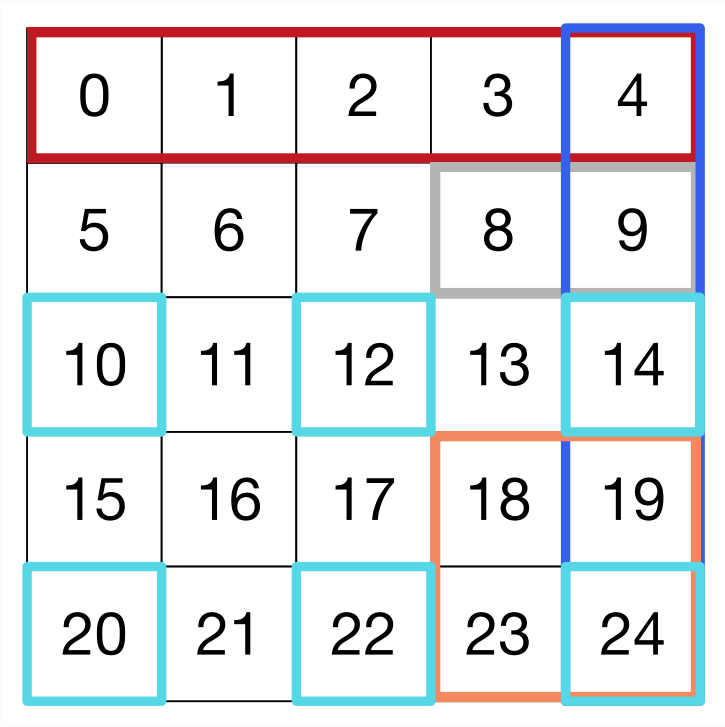
```
>>> a[0]
array([0, 1, 2, 3, 4])
```

```
>>> a[1, 3:5]
array([8, 9])
```

```
>>> a[:, 4]
array([ 4,  9, 14, 19, 24])
```

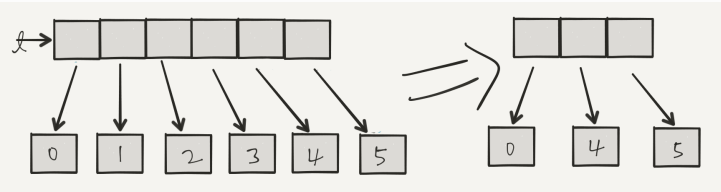
```
>>> a[-2:, -2:]
array([[18, 19],
       [23, 24]])
```

```
>>> a[2::2, ::2]
array([[10, 12, 14],
       [20, 22, 24]])
```



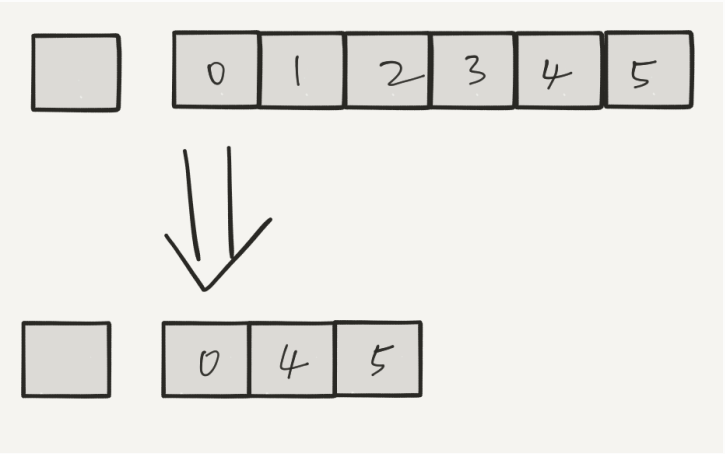
파이썬 마스킹

```
>>> l = list(range(6))
>>> mask = [True, False, False,
...          False, True, True]
>>> [i for i, j in zip(l, mask) if j]
[0, 4, 5]
```



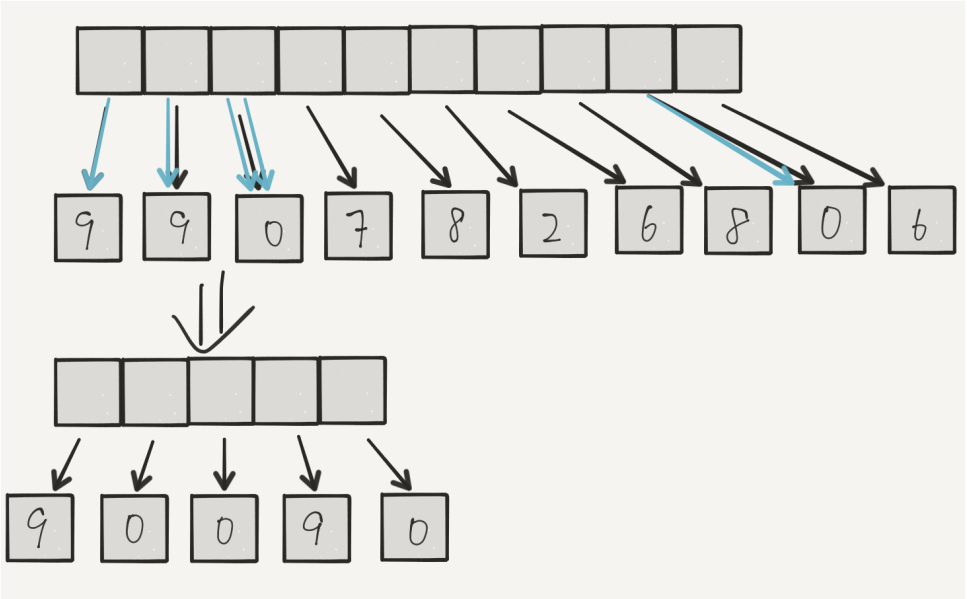
NumPy Fancy Indexing

```
>>> a = np.arange(6)
>>> a[np.array(mask)]
array([0, 4, 5])
>>> mask = np.array([1, 0, 0, 0, 1, 1],
...                 dtype=np.bool)
>>> a[mask]
array([0, 4, 5])
```



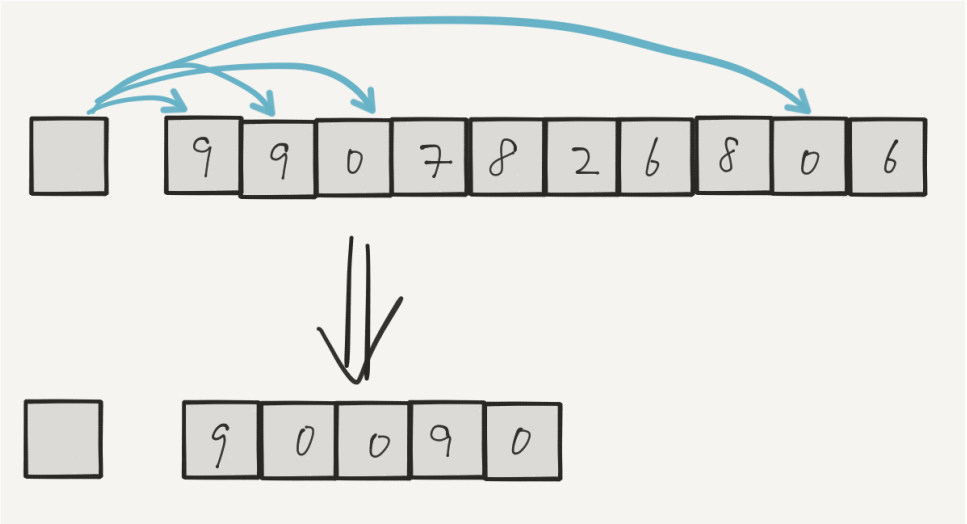
파이썬 인덱스 배열 사용

```
>>> l = [random.randint(0, 10)
...      for i in range(10)]
>>> l
[9, 9, 0, 7, 8, 2, 6, 8, 0, 6]
>>> mask = [0, 2, 2, 1, 8]
>>> [l[i] for i in mask]
[9, 0, 0, 9, 0]
```



NumPy Fancy Indexing

```
>>> from numpy.random\
... import random_integers
>>> a = random_integers(0, 10, 10)
>>> a
array([9, 9, 0, 7, 8, 2, 6, 8, 0, 6]
>>> a[mask]
array([9, 0, 0, 9, 0])
```



흩어진 접근과 Fancy Indexing

- Fancy indexing 사용시 데이터의 복사가 필연적
  - 어떤 규칙으로 메모리를 뛰어야 하는지 모름
- 캐시의 활용도도 떨어짐
  - 등간격이 흩어진 것보다 낫다
- 하지만 표현 방식이 간결하며 자연스러움
  - 원하는 것을 설명하는 편이 명령을 내리는 것보다 낫다 (데이터 타입을 사용하자)

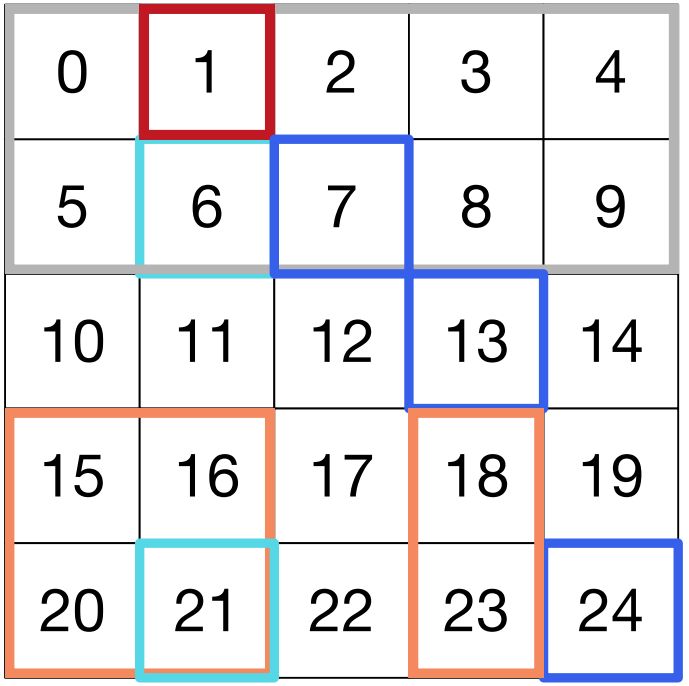
Fancy Indexing

```
>>> a[0, 1]
1
```

```
>>> a[[0, 1]]
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

```
>>> a[[1, 2, 4], [2, 3, 4]]
array([ 7, 13, 24])
```

```
>>> a[3:, [0, 1, 3]]
array([[15, 16, 18],
       [20, 21, 23]])
```




```
>>> mask = np.array([0, 1, 0, 0, 1], dtype=np.bool)
>>> a[mask, 1]
array([ 6, 21])
```

벡터화된 연산

- 파이썬은 컴파일된 언어에 비해 느린 언어
  - 타입 정보를 활용할 수 없음
- 특정 종류의 연산을 미리 컴파일된 언어로 구현하였다면 추가적인 성능 확보 가능
  - **벡터화된 연산을 사용**

루프와 벡터화된 연산 사용 비교

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 7, 3])
>>> [x + y for x, y in zip(a, b)]
[5, 9, 6]
```

```
>>> a = np.array([1, 2, 5])
>>> b = np.array([4, 7, 5])
>>> a + b
array([ 5,  9, 10])
```

루프와 벡터화된 연산 성능 비교

```
l = range(1000)
[i ** 2 for i in l]
```

$\sim$500 μs per loop

```
import numpy as np
a = np.arange(1000)
a ** 2
```

$\sim$2 μs per loop

벡터화된 연산

- NumPy는 이미 컴파일된 코드로 만들어진 다양한 벡터화된 연산을 제공
  - 이런 종류의 연산을 ufunc라고 함
  - 더 간결하고 효율적
  - 원하는 것을 설명하는 편이 명령을 내리는 것보다 낫다 (데이터 타입을 사용하자)
  - 이를 통해 컴파일된 언어의 성능을 끌어다 쓸 수 있음
  - 벡터화한 것이 명시적인 루프보다 좋다
- 모든 연산은 기본적으로 개별 원소마다 (elementwise) 적용
- 파이썬에서 기본적으로 제공하는 함수와 섞어 쓰지 않을 것

벡터화된 연산

```
>>> a = np.array([1, 2, 5])
>>> b = np.array([4, 7, 5])
>>> a + b
array([ 5,  9, 10])
>>> a == b
array([False, False,  True], dtype=bool)
>>> a * 3
array([ 3,  6, 15])
>>> np.log(a)
array([ 0.        ,  0.69314718,  1.60943791])
>>> np.cumsum(a)
array([1, 3, 8])
```

브로드캐스팅

```
>>> a = np.array([1, 2, 5])
>>> a + np.array([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

브로드캐스팅

- 기본 연산은 개별 원소마다 적용
  - 그러므로 모양이 다른 배열 간의 연산이 불가능
- 하지만 특정 조건이 만족되면 배열 변환이 자동으로 일어나서 연산 가능
  - 이를 브로드캐스팅 (broadcasting) 이라 함

브로드캐스팅 규칙

```
a : 4 x 3
b :     3
---------
a : 4 x 3
b : 1 x 3
---------
a : 4 x 3
b : 4 x 3
---------
r : 4 x 3
```

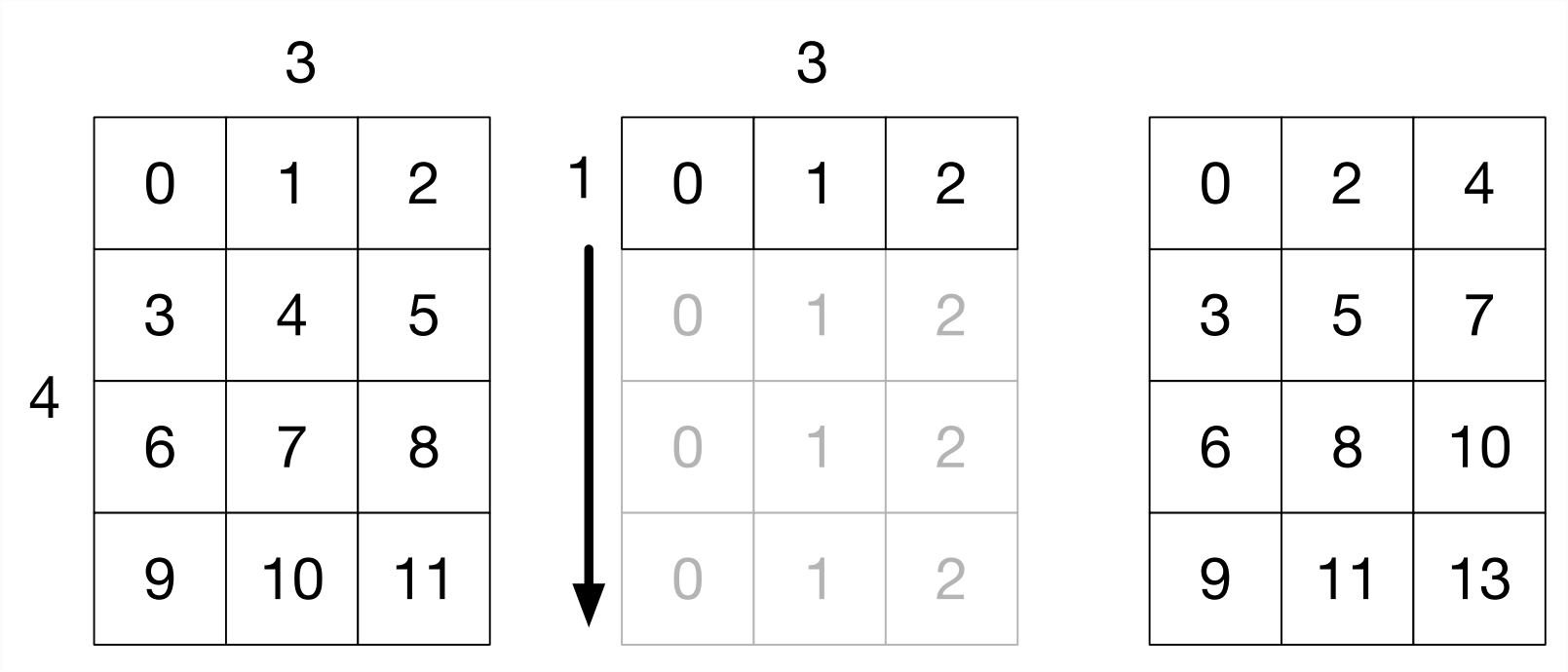
- 두 배열을 오른쪽 정렬
- 차원 개수가 작은 배열은 왼쪽 차원을 1로 채움
- 각각 짝이 되는 차원의 크기가 같으면 연산 가능
- 차원의 크기가 1이면 연산 가능
  - 잡아 당겨서 같은 같은 크기가 되도록 변환

브로드캐스팅 예제

```
a : 4 x 3
b :     3
---------
a : 4 x 3
b : 1 x 3
---------
a : 4 x 3
b : 4 x 3 (stretching)
---------
r : 4 x 3
```

```
>>> a = np.arange(12).reshape(4, 3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> b = np.arange(3)
>>> b
array([0, 1, 2])
>>> r = a + b
>>> r
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10],
       [ 9, 11, 13]])
```

브로드캐스팅



브로드캐스팅이 좋은 이유

- 잡아당기는 연산은 명시적인 메모리 복사가 일어나지 않음
  - 속도/메모리 이득
  - 브로드캐스팅은 좋은 아이디어다 – 가능하면 사용하자
- 브로드캐스팅과 꼴 (shape) 변환을 활용하면 루프를 피할 수 있음
  - 뒤의 최단거리 이웃 예제 참조
  - 고차원에서 생각하라

Reduction

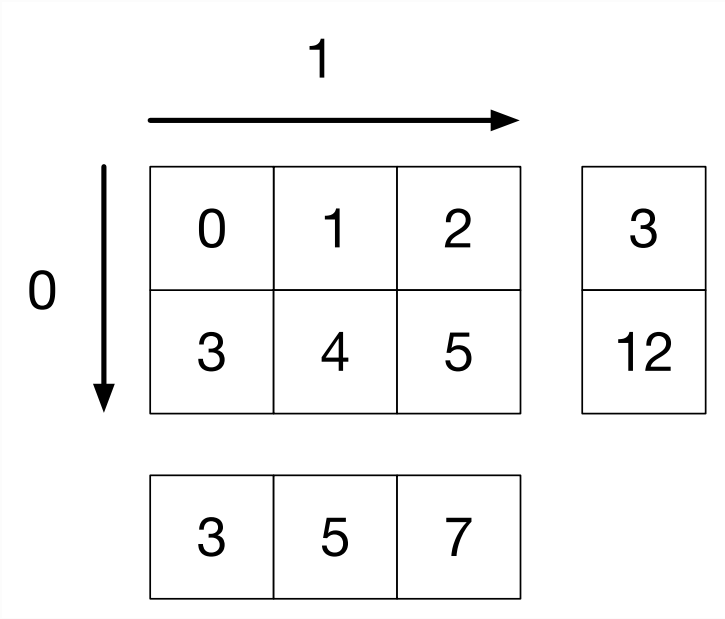
```
>>> l = [[0, 1, 2], [3, 4, 5]]
>>> l
[[0, 1, 2], [3, 4, 5]]
>>> sum([x + y for x, y in zip([0, 1, 2], [3, 4, 5])])
15
>>> [x + y for x, y in zip([0, 1, 2], [3, 4, 5])]
[3, 5, 7]
>>> [sum(k) for k in ([0, 1, 2], [3, 4, 5])]
[3, 12]
```

Reduction

- 같은 합연산이지만 매번 다른 방식으로 처리해야 함
- 파이썬 루프이므로 느림
- 이런 종류의 연산은 주어진 배열의 특정 축을 제거한다고 생각할 수 있음

Reduction

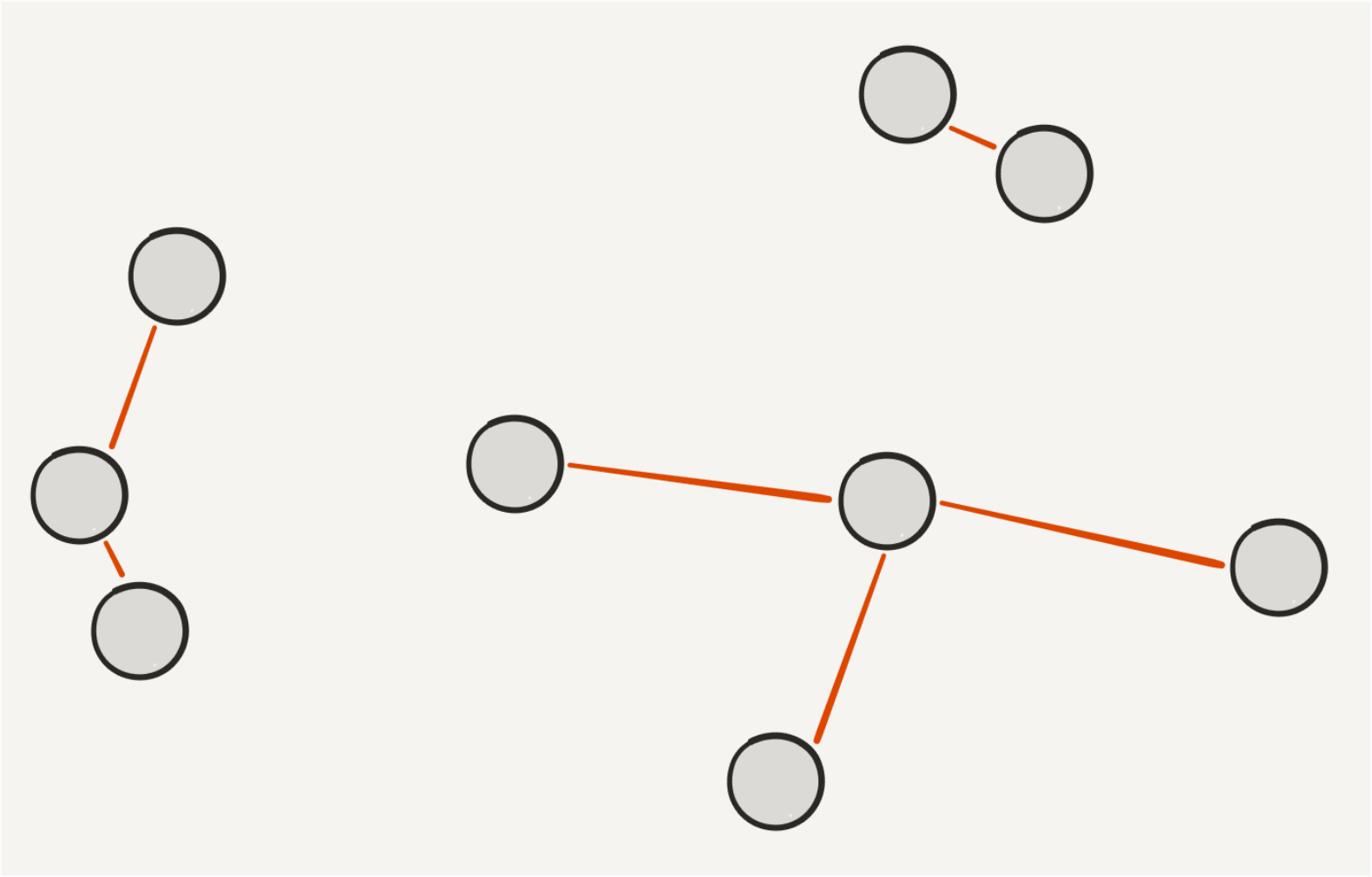
```
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.sum(a)
15
>>> np.sum(a, axis=0)
array([3, 5, 7])
>>> np.sum(a, axis=1)
array([ 3, 12])
```



Reduction

- 같은 인터페이스를 통해 원하는 연산을 적용
  - 원하는 것을 설명하는 편이 명령을 내리는 것보다 낫다 (데이터 타입을 사용하자)
- 벡터화된 연산이므로 빠름
  - 벡터화한 것이 명시적인 루프보다 좋다
- 여러 차원의 reduction은 tuple을 인자로

Nearest Neighbor



- 데이터의 이웃을 찾는 알고리즘
- 비슷한 연산이 k-means, vector quantization에서도 사용됨

Nearest Neighbor 구현

```
import numpy as np

n = 1000
dim = 3

data = np.random.random(size=dim*n).reshape(n, dim)
diff = data.reshape(n, 1, dim) - data
distance = np.sum(diff ** 2, axis=2)
oneself = np.arange(n)
distance[oneself, oneself] = np.inf
neighbors = np.argmin(distance, axis=1)
```

Nearest Neighbor

High dimensions, broadcasting, ufunc

```
diff = data.reshape(n, 1, dim) - data
```

- 1000 x 3 행렬을 1000 x 1 x 3의 고차원 ndarray로 변환
- 브로드캐스팅에 의해 1000 x 1000 x 3의 결과가 나옴
- 벡터화된 빼기 연산

Nearest Neighbor

Ufunc, reduction

```
distance = np.sum(diff ** 2, axis=2)
```

- 벡터화된 승수 연산
- Reduction 더하기 연산

Nearest Neighbor

Indexing

```
oneself = np.arange(n)
distance[oneself, oneself] = np.inf
```

- 인덱싱
- 자기 자신과의 거리를 무한대로 설정

Nearest Neighbor

Reduction

```
neighbors = np.argmin(distance, axis=1)
```

- 최소값의 위치를 찾는 reduction

Nearest Neighbor에서 사용된 아이디어

- 앞서 다룬 아이디어를 사용해서 루프 없이 계산함
  - 인덱싱
  - 고차원 변환
  - 벡터화된 연산 (ufunc)
  - Reduction
  - 브로드캐스팅
- 단 고차원으로 변환한 뒤 수행하는 브로드캐스팅은 메모리를 많이 사용할 수 있음
  - 문제에 따라 바깥 루프는 직접 도는 것이 효율적일 수 있음

Exploration

더 관심이 있다면 찾아볼 것들

- 작은 예제를 통해 NumPy를 익히고 싶다면 `http://www.labri.fr/perso/nrougier/teaching/numpy.100/`
- Scipy Lecture Notes에서 보다 자세한 설명을 찾을 수 있음 `http://www.scipy-lectures.org/`
- `np.cumsum`(scan, prefix sum)을 활용하면 의외로 복잡한 연산도 쉽게 벡터화 할 수 있음

Zen of NumPy

등간격이 흩어진 것보다 낫다.
연속된 것이 등간격보다 낫다.
원하는 것을 설명하는 편이 명령을 내리는 것보다 낫다 (데이터 타입을 사용하자).
배열 지향이 대체로 객체 지향보다 낫다.
브로드캐스팅은 좋은 아이디어다 – 가능하면 사용하자.
벡터화한 것이 명시적인 루프보다 좋다.
하지만 복잡하다면 numexpr, weave나 Cython을 사용해라.
고차원에서 생각하라.

감사합니다

- Email : shurain@gmail.com
- Twitter : @shurain
- Homepage : `http://shurain.net`