
NumPy

학습목표

학습목표

1. NumPy NDArray의 구조를 이해한다.
2. NumPy의 NDArray와 Python의 List의 차이를 이해한다.
3. NumPy의 기본 사용법을 이해한다.


NumPy

NumPy

- 배열과 행렬에 특화된 Python 라이브러리
- Python의 List나 Tuple은 너무 느림
- 효율적인 메모리 구조로 행렬 연산을 빠르게 수행

https://numpy.org/

Data Types: Integers

NumPy	C	Byte (64-bit Windows)
`bool`	`bool`	Boolean (True or False) in a byte
`byte`	`signed char`	Platform-defined (1)
`ubyte`	`unsigned char`	Platform-defined (1)
`short`	`short`	Platform-defined (2)
`ushort`	`unsigned short`	Platform-defined (2)
`intc`	`int`	Platform-defined (4)
`uintc`	`unsigned int`	Platform-defined (4)
`int_`	`long`	Platform-defined (4)
`uint`	`unsigned long`	Platform-defined (4)
`longlong`	`long long`	Platform-defined (8)
`ulonglong`	`unsigned long long`	Platform-defined (8)

Data Types: Floats

NumPy	C	Byte (64-bit Windows)
`single`	`float`	Platform-defined (4)
`double`	`double`	Platform-defined (8)
`longdouble`	`long double`	Platform-defined (8)
`csingle`	`float complex`	Platform-defined (8)
`cdouble`	`double complex`	Platform-defined (16)
`clongdouble`	`long double complex`	Platform-defined (16)

Data Types: Integers (fixed-size)

NumPy	C	Description	Range
`int8`	`int8_t`	Byte	-128 to 127
`int16`	`int16_t`	Integer	-32768 to 32767
`int32`	`int32_t`	Integer	-2147483648 to 2147483647
`int64`	`int64_t`	Integer	-9223372036854775808 to 9223372036854775807
`uint8`	`uint8_t`	Unsigned Integer	0 to 255
`uint16`	`uint16_t`	Unsigned Integer	0 to 65535
`uint32`	`uint32_t`	Unsigned Integer	0 to 4294967295
`uint64`	`uint64_t`	Unsigned Integer	0 to 18446744073709551615
`intp`	`intptr_t`	Integer Pointer	large enough
`uintp`	`uintptr_t`	Unsigned Integer Pointer	large enough

Data Types: Floats (fixed-size)

NumPy	C	Description
`float32`	`float`	
`float64`	`double`	
`complex64`	`float complex`	
`complex128`	`double complex`	

ndarray 생성하기

```
>>> import numpy as np

# Create an array from a list or a tuple
>>> a = np.array([2, 3, 4])
>>> a
array([2, 3, 4])
>>> type(a)        # <class 'numpy.ndarray'>

>>> a.dtype        # deduced type
dtype('int32')

>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')

>>> c = np.array(1,2,3,4)    # WRONG
>>> c = np.array([1,2,3,4])  # RIGHT

>>> a.fill(5)    # Fill with a value
>>> a
array([5, 5, 5])
```

ndarray 생성하기

```
>>> import numpy as np

# Create a 2D array
>>> a = np.array([(1,2,3), (4,5,6)])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.dtype
dtype('int32')

# Explicit type of the array
>>> b = np.array([[1,2], [3,4]], dtype=float)
>>> b
array([[1., 2.],
       [3., 4.]])
>>> b.dtype
dtype('float64')
```

https://numpy.org/doc/stable/user/quickstart.html

ndarray 생성하기

```
>>> import numpy as np

# The function zeros creates an array full of zeros
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])

# The functions ones creates an array full of ones
>>> np.ones((2,3,4), dtype=np.int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)

# The function empty creates a random array
>>> np.empty((2,3))
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

ndarray 생성하기

```
>>> import numpy as np

# Identity matrix (square matrix)
>>> np.identity(2)
array([[1., 0.],
       [0., 1.]])

# Identity matrix (any size)
>>> np.eye(2, 3)
array([[1., 0., 0.],
       [0., 1., 0.]])

# Diagonal matrix
>>> np.diag([1,2,3])
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

ndarray 생성하기

```
>>> import numpy as np

# Range from stop (not inclusive)
>>> np.arange(5)
array([0, 1, 2, 3, 4])

# Range from start, stop, step
>>> np.arange(10, 30, 5)
array([10, 15, 20, 25])

>>> np.arange(0, 2, 0.3)      # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])

# Uniformly spaced sequences using the number of elements
>>> np.linspace(0, 2, 9)         # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])

>>> x = np.linspace(0, 2*np.pi, 100) # useful to evaluate func
>>> f = np.sin(x)

>>> np.logspace(2, 3, 5, base = 2)
array([4.        , 4.75682846, 5.65685425, 6.72717132, 8.        ])
```

`ndarray` 출력하기

- 마지막 차원은 가로(왼쪽→오른쪽)로 출력
- 마지막에서 두번째 차원은 세로(위→아래)로 출력
- 나머지는 세로(위→아래)로 슬라이스해서 출력

- 예: 1-D array는 행벡터로 출력
- 예: 2-D array는 행렬로 출력
- 예: 3-D array는 행렬 슬라이스를 아래로 출력

ndarray 출력하기

```
>>> import numpy as np
>>> a = np.arange(6)                         # 1d array
>>> print(a)
[0 1 2 3 4 5]

>>> b = np.arange(12).reshape(4,3)           # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]

>>> c = np.arange(24).reshape(2,3,4)         # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

ndarray 모양바꾸기

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.shape
(2, 3)

>>> a.reshape(3, 2) # returns a new array with a modified shape
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> a.T             # returns a new array, transposed
array([[1, 4],
       [2, 5],
       [3, 6]])

>>> a.shape          # the original array is untouched
(2, 3)
```

ndarray 모양바꾸기

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])

>>> a.reshape(3, -1)
array([[1, 2],
       [3, 4],
       [5, 6]])

>>> a.resize(3, 2) # modifies the array itself (in-place)
>>> a.shape
(3, 2)

>>> np.resize(a, (3, 3)) # change the memory size
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3]])
```

ndarray 차원 확장

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

```
>>> a_row = a[np.newaxis]
>>> a_row
array([[0, 1, 2, 3]])
>>> a_row.ndim
2
>>> a_row.shape
(1, 4)
```

```
>>> a_col = a[:, np.newaxis]
>>> a_col
array([[0],
       [1],
       [2],
       [3]])
>>> a_col.ndim
2
>>> a_col.shape
(4, 1)
```

ndarray 차원 축소

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a.shape
(2, 3)

>>> a.flatten()        # returns the array, flattened
array([1, 2, 3, 4, 5, 6])

>>> a.ravel()         # returns the array, flattened
array([1, 2, 3, 4, 5, 6])
```

ndarray 병합

```
>>> import numpy as np
>>> a = np.arange(6).reshape(2, 3)
>>> b = np.arange(6, 12).reshape(2, 3)
```

```
>>> np.stack([a, b], axis=0)
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])

>>> np.stack([a, b], axis=1)
array([[[ 0,  1,  2],
        [ 6,  7,  8]],

       [[ 3,  4,  5],
        [ 9, 10, 11]]])
```

```
# axis = 2
>>> np.stack([a, b], axis=-1)
array([[[ 0,  6],
        [ 1,  7],
        [ 2,  8]],

       [[ 3,  9],
        [ 4, 10],
        [ 5, 11]]])
```

ndarray 분리

```
>>> import numpy as np
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])

>>> np.split(a, 2, axis=0)
[array([[0, 1, 2]]), array([[3, 4, 5]])]

>>> np.split(a, 3, axis=1)
[array([[0],   array([[1],   array([[2],
        [3]]),        [4]]),        [5]])]
```

Element-wise Operation

```
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])

>>> c = a - b
>>> c
array([20, 29, 38, 47])

>>> b**2
array([0, 1, 4, 9])

>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])

>>> a<35
array([ True,  True, False, False])
```

Element-wise vs. Matrix Product

```
>>> import numpy as np
>>> A = np.array([[1,1],
...               [0,1]])
>>> B = np.array([[2,0],
...               [3,4]])


>>> A * B         # element-wise product
array([[2, 0],
       [0, 4]])

>>> A @ B         # matrix product
array([[5, 4],
       [3, 4]])

>>> A.dot(B)      # another matrix product
array([[5, 4],
       [3, 4]])
```

Unary Operations

```
>>> import numpy as np
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.sum()
15
>>> a.min()
0
>>> a.max()
5

>>> a.sum(axis=0)      # sum of each column
array([3, 5, 7])

>>> a.min(axis=1)      # min of each row
array([0, 3])

>>> a.cumsum(axis=1)   # cumulative sum along each row
array([[ 0,  1,  3],
       [ 3,  7, 12]], dtype=int32)
```

Universal Functions

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

>>> np.exp(A)
array([1.        , 2.71828183, 7.3890561 ])

>>> np.sqrt(A)
array([0.        , 1.        , 1.41421356])

>>> np.sin(A)
array([0.        , 0.84147098, 0.90929743])
```

Broadcasting and Type Casting

```
>>> import numpy as np
>>> a =
np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> b = np.array([1.5, 2.5,
3.5])
>>> b
array([1.5, 2.5, 3.5])

>>> a + b
array([[1.5, 3.5, 5.5],
       [4.5, 6.5, 8.5]])
```

```
>>> import numpy as np
>>> a =
np.arange(4).reshape(1, 4)
>>> a
array([[0, 1, 2, 3]])
>>> b =
np.arange(3).reshape(3, 1)
>>> b
array([[0],
       [1],
       [2]])

>>> a + b
array([[0, 1, 2, 3],
       [1, 2, 3, 4],
       [2, 3, 4, 5]])
```

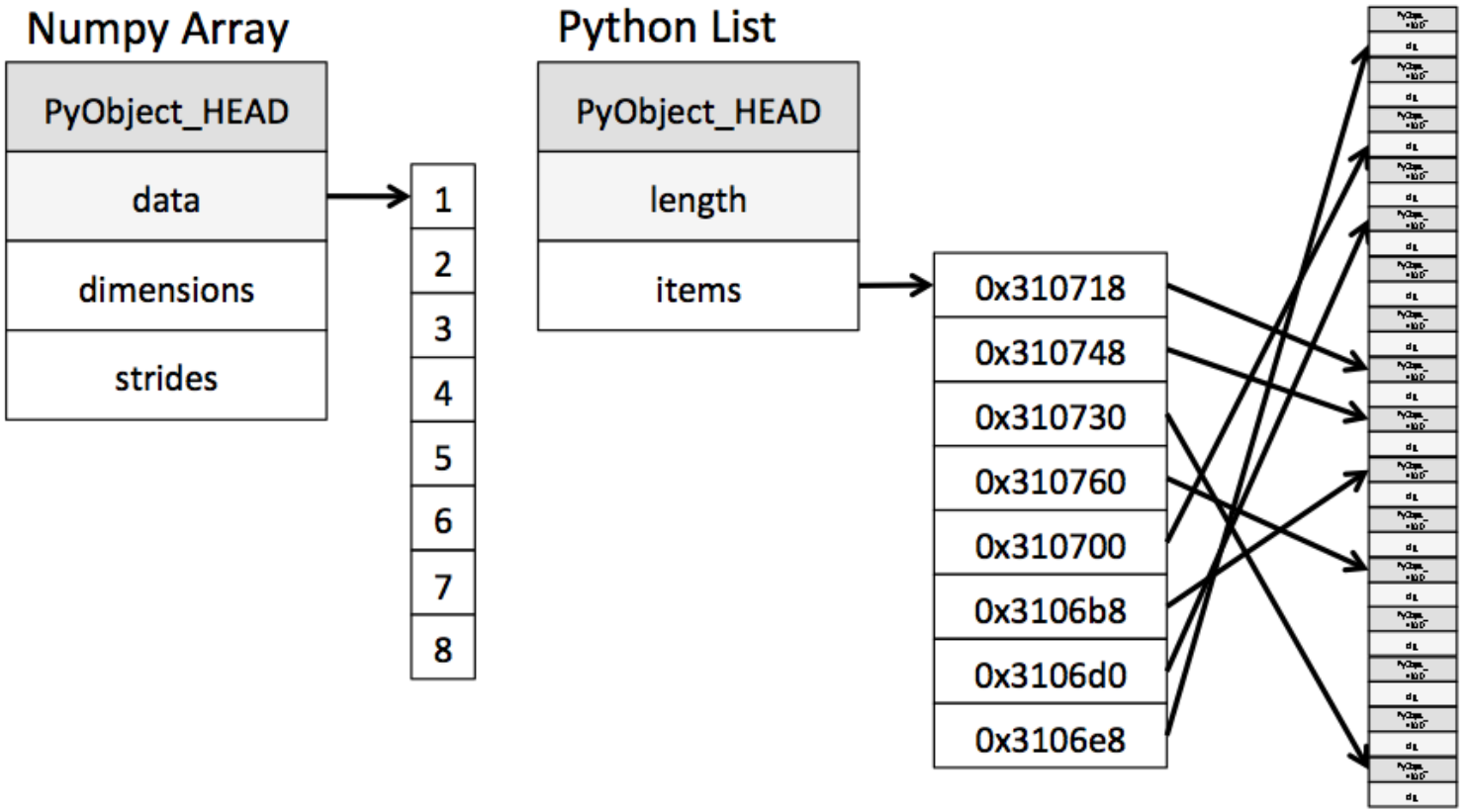
Broadcasting Rules

1. 두 차원의 차원(`ndim`)이 같지 않다면 차원이 더 낮은 배열이 차원이 더 높은 배열과 같은 차원의 배열로 인식된다. 예를 들어 (1, 2) 배열과 (1, 4, 2)의 배열을 연산한다면 (1, 2) 배열은 (1, 1, 2) 배열로 간주된다.
2. 반환된 배열은 연산을 수행한 배열 중 차원의 수(`ndim`)가 가장 큰 배열이 된다.
3. 연산에 사용된 배열과 반환된 배열의 차원의 크기(`shape`)가 같거나 1일 경우 브로드캐스팅이 가능하다.
4. 브로드캐스팅이 적용된 배열의 차원 크기(`shape`)는 연산에 사용된 배열들의 차원의 크기에 대한 최소 공배수 값으로 사용한다. 예를 들어 (6, 2, 1), (2, 3)의 배열을 브로드캐스팅한다면 (2, 3)은 (1, 2, 3)으로 반환되고 각 요소의 최소 공배수 값을 반환해서 (6, 2, 3)이 된다.

윤대희, "C#과 파이썬을 활용한 OpenCV4 프로그래밍", 2019.

Memory

NumPy NDArray vs. Python List



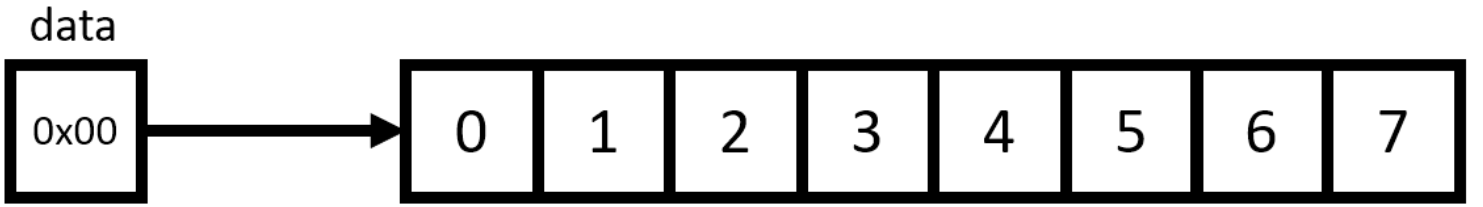
https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html

NumPy NDArray 속성

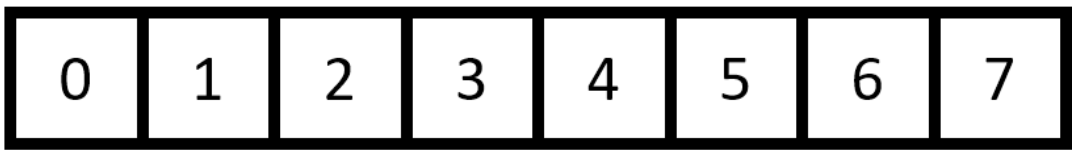
속성	설명
`ndim`	차원(dimension)의 개수
`shape`	각 차원의 크기
`stride`	각 차원에서 다음 요소까지의 바이트 크기
`size`	전체 요소(element)의 개수 = `shape`의 각 항목의 곱
`dtype`	요소(element)의 데이터 타입
`itemsize`	각 요소의 바이트 크기 = `dtype`의 바이트 크기
`nbytes`	총 바이트 크기 = `size` × `itemsize`
`data`	데이타 메모리 포인터

https://numpy.org/doc/stable/reference/arrays.ndarray.html

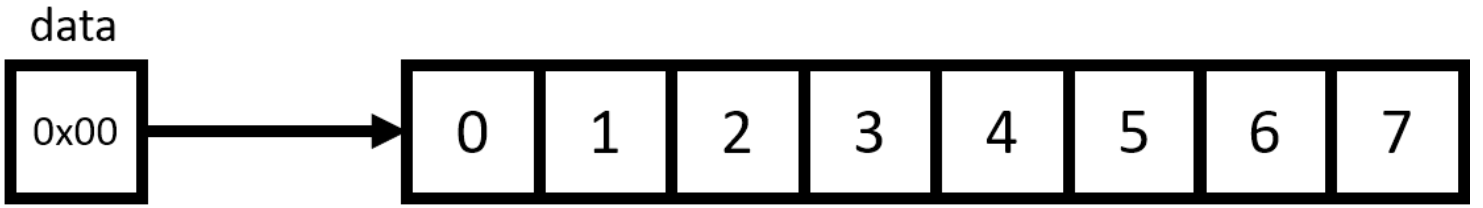
NumPy NDArray의 Memory 구조



ndims = 1
shape = (8,)
strides = (4,)
dtype = int32



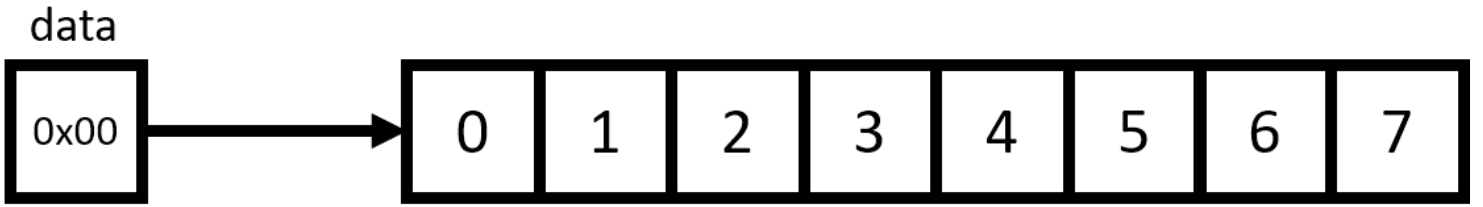
NumPy NDArray의 Memory 구조



ndims = 2
shape = (2, 4)
strides = (16, 4)
dtype = int32

0	1	2	3
4	5	6	7

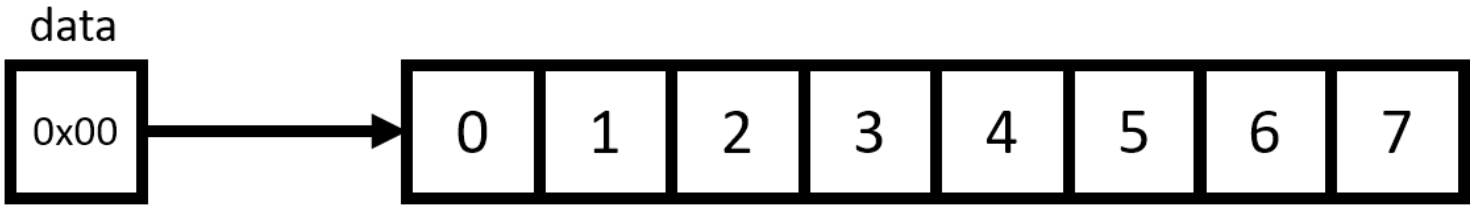
NumPy NDArray의 Memory 구조



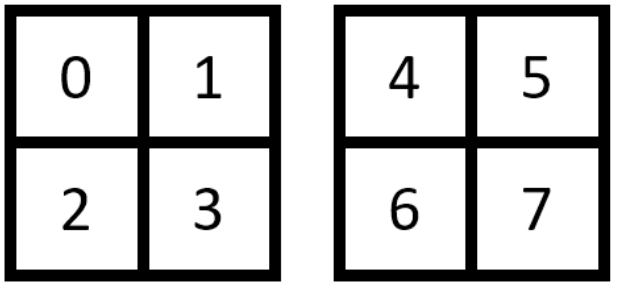
ndims = 2
shape = (4, 2)
strides = (8, 4)
dtype = int32

0	1
2	3
4	5
6	7

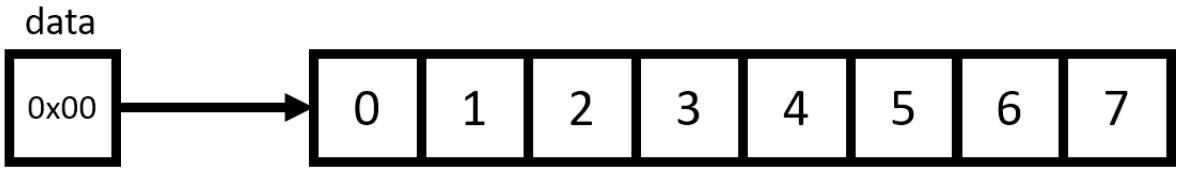
NumPy NDArray의 Memory 구조



ndims = 3
shape = (2, 2, 2)
strides = (8, 4, 16)
dtype = int32



Transpose? 전치행렬?



ndims = 2
shape = (2, 4)
strides = (16, 4)
dtype = int32

0	1	2	3
4	5	6	7

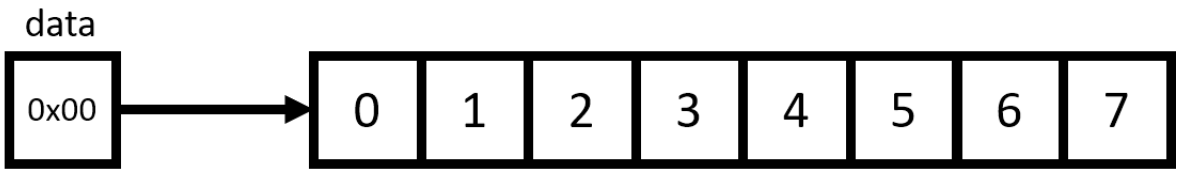
ndims = 2
shape = (4, 2)
strides = (8, 4)
dtype = int32

0	1
2	3
4	5
6	7

ndims = 2
shape = (4, 2)
strides = (?, ?)
dtype = int32

0	4
1	5
2	6
3	7

Transpose Matrix? Transpose Stride!



ndims = 2
shape = (2, 4)
strides = (16, 4)
dtype = int32

0	1	2	3
4	5	6	7

ndims = 2
shape = (4, 2)
strides = (8, 4)
dtype = int32

0	1
2	3
4	5
6	7

ndims = 2
shape = (4, 2)
strides = (4, 16)
dtype = int32

0	4
1	5
2	6
3	7

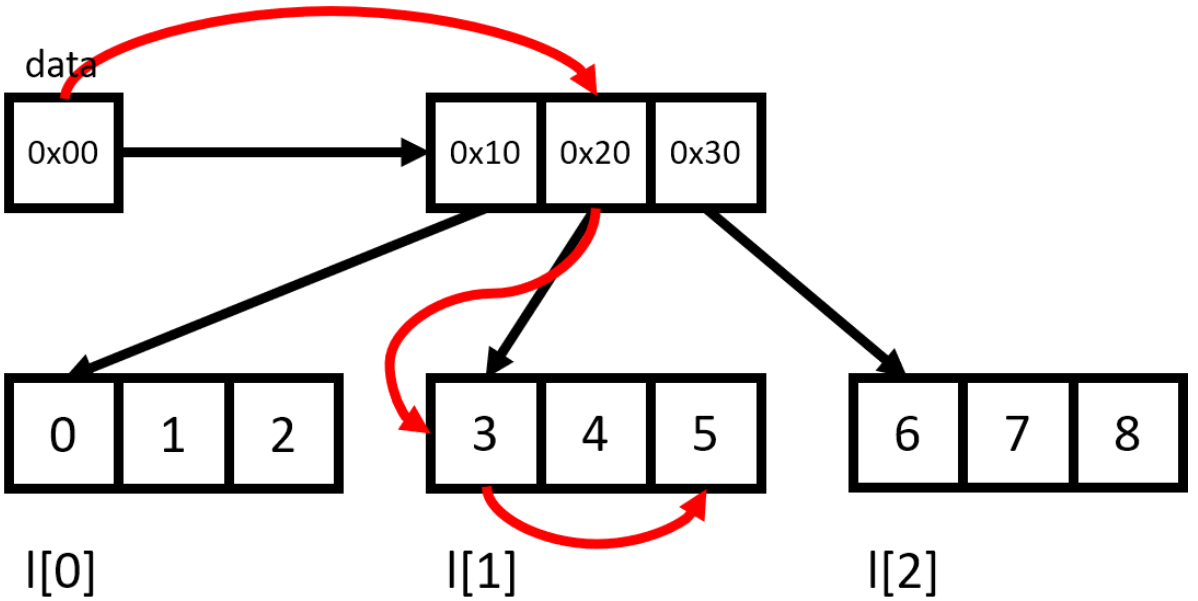
NumPy NDArray 장점 vs. 단점

- 구현
  - 원소의 타입을 명시하여 1차원 배열로 메모리를 할당한다.
- 장점 (speed)
  - 메모리가 연속적(continguous)이다.
  - Cache memory를 활용할 수 있다.
  - 해당 element로 바로 접근(direct access)이 가능하다.
  - 행렬을 다양한 모양(shape, view)으로 표현할 수 있다.
  - strides를 잘 활용하면 다양한 연산을 효율적으로 실행가능하다.
  - CPU의 Parallel Processing을 이용할 수 있다.
  - BLAS나 LAPACK과 같은 최저화된 선형대수 라이브러리와 링크
- 단점
  - 한가지 타입의 데이터만 저장할 수 있다.

Indexing

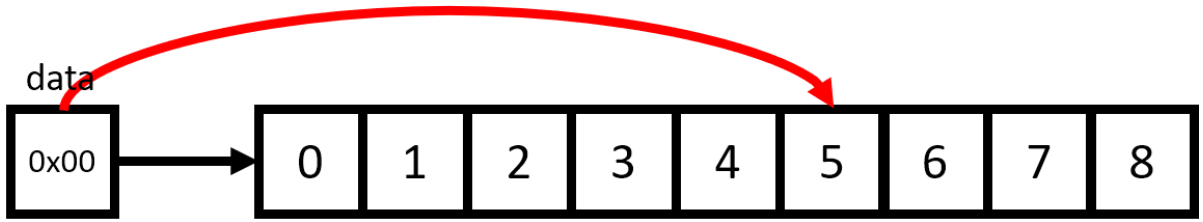
Python List Indexing

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



NumPy NDArray Indexing

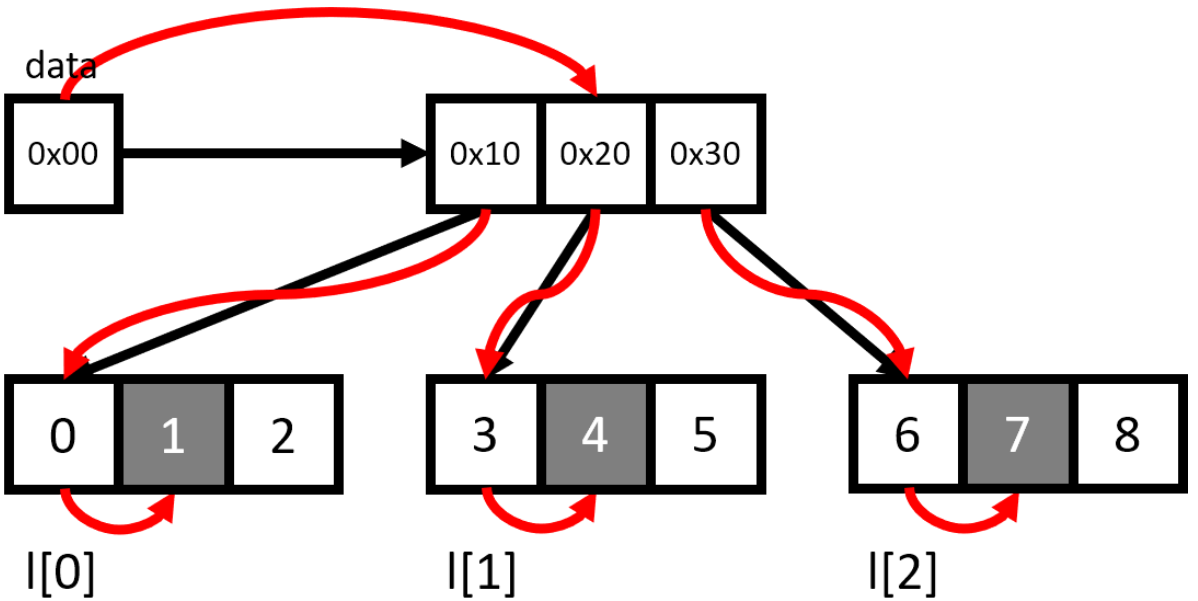
```
>>> import numpy as np
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> a[1]
array([3, 4, 5])

>>> a[1][2]
5
>>>
```



Slicing

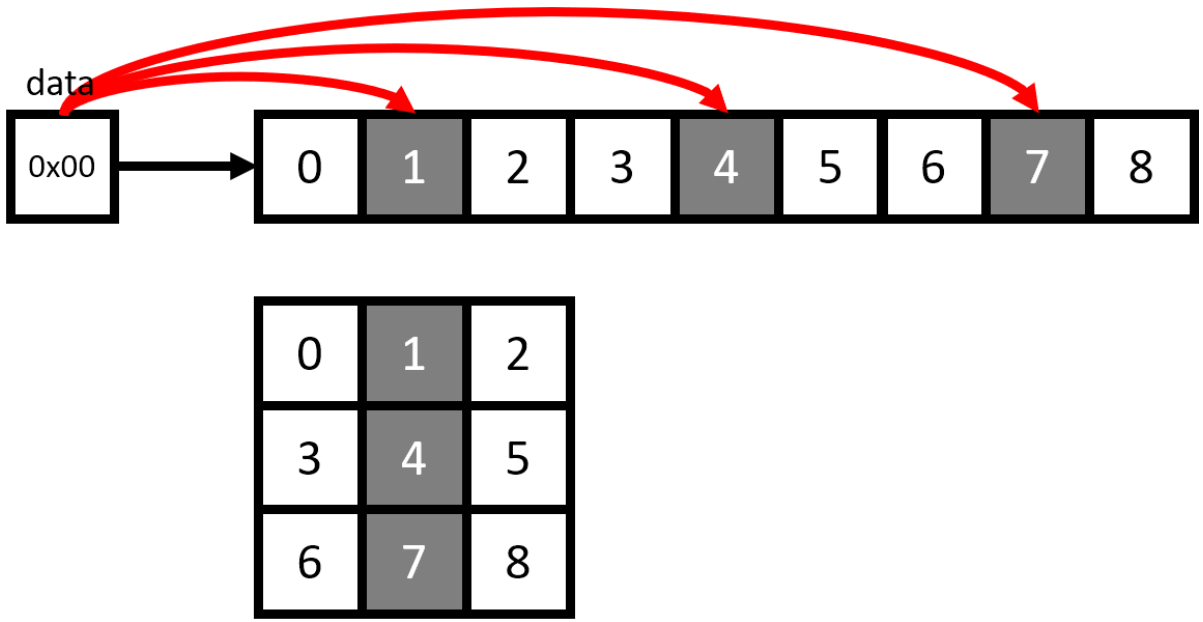
Python List Slicing

```
>>> l = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
>>> [l[i][1] for i, j in enumerate(l)]
[1, 4, 7]
```

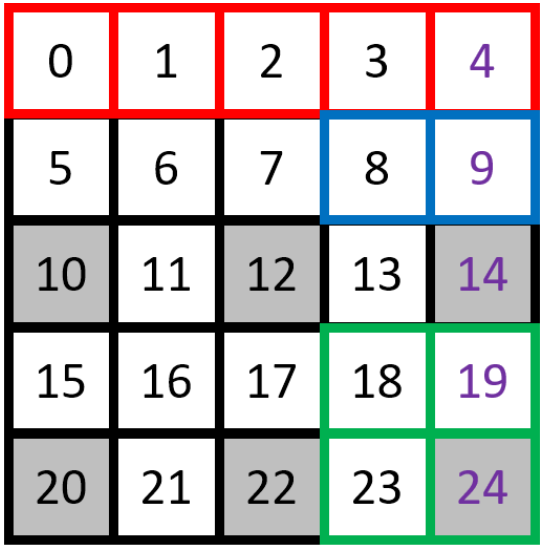


NumPy NDArray Slicing

```
>>> import numpy as np
>>> a = np.arange(9).reshape(3,3)
>>> a[:, 1]
array([1, 4, 7])
```



NumPy NDArray Indexing and Slicing




```
>>> a = np.arange(25).reshape(5,5)

>>> 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]])
```

Masking

Python Masking vs. NumPy Masking

```
>>> l = list(range(6))
>>> l
[0, 1, 2, 3, 4, 5]
>>> mask = [True, False, False, False, True, True]

>>> [i for i, j in zip(l, mask) if j]
[0, 4, 5]
```

```
>>> import numpy as np
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> mask = [True, False, False, False, True, True]

>>> a[mask]
array([0, 4, 5])
```

속도의 차이는 없지만 문법이 간결하다.

Python Indexing vs. NumPy Fancy Indexing

```
>>> import random
>>> l = [random.randint(0, 9) for i in range(10)]
>>> l
[3, 7, 3, 9, 0, 4, 3, 3, 0, 6]
>>> index = [0, 2, 2, 1, 8]

>>> [l[i] for i in index]
[3, 3, 3, 7, 0]
```

```
>>> from numpy.random import randint
>>> a = randint(0, 9, 10)
>>> a
array([0, 0, 3, 5, 5, 4, 1, 5, 0, 2])
>>> index = [0, 2, 2, 1, 8]

>>> a[index]
array([0, 3, 3, 0, 0])
```

NumPy Fancy Indexing

```
>>> import numpy as np
>>> a = np.random.randint(0, 9, 10)
>>> a
array([8, 1, 3, 5, 6, 6, 3, 5, 0, 6])

>>> index = [3, 7, 4]
>>> a[index]
array([5, 5, 6])

# When using fancy indexing, the shape of the result
# reflects the shape of the index arrays rather than
# the shape of the array being indexed
>>> index = np.array([[3, 7], [4, 5]])
>>> a[index]
array([[5, 5],
       [6, 6]])
```

NumPy Fancy Indexing

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

# Like with standard indexing, the first index refers to
# the row, and the second to the column
>>> row = np.array([0, 1, 2])
>>> col = np.array([2, 1, 3])
>>> a[row, col]
array([ 2,  5, 11])    # a[0, 2], a[1, 1], a[2, 3]
```

NumPy Masking and Fancy Indexing

- 장점 (readability)
  - 표현 방식이 간결하고 자연스럽다.
- 단점
  - 데이터의 복사는 일어날 수 밖에 없다.
  - 어떤 규칙으로 메모리를 점프해야하는지 모르니까.

Push Code to GitHub

References

References

- Scipy Lecture Notes - NumPy
- Zen of NumPy, 하성주
- NumPy for Matlab Users