IPA 주관 인공지능센터 기본(fundamental) 과정

- **GitHub link: here**
- **E-Mail: windkyle7@gmail.com**

NumPy

Vectorization 기법으로 구현된 행렬 및 벡터 연산 라이브러리

Numerical Python = NumPy

- 벡터, 행렬 연산을 위한 수치해석용 **python** 라이브러리
  - 빠른 수치 계산을 위한 **Structured Array** 및 **vectorized arithmetic operations (without having to write loops) and sophisticated broadcasting**을 통한 다차원 배열과 행렬 연산에 필요한 다양한 함수를 제공
  - **Linear algebra, random number generation, and Fourier transform capabilities**
  - 메모리 버퍼에 배열 데이터를 저장하고 처리
    - **list, array** 비교하면 **NumPy**의 **ndarray** 객체를 사용하면 더 많은 데이터를 더 빠르게 처리
    - **ndarray**는 타입을 명시하여 원소의 배열로 데이터를 유지
    - 다차원 데이터도 연속된 메모리 공간이 할당됨
    - 많은 연산이 **strides**를 잘 활용하면 효율적으로 가능
    - **transpose**는 **strides**를 바꾸는 것으로 거의 추가 구현이 필요치 않음
  - **C**로 구현 **(**파이썬용 **C**라이브러리**)**
  - **BLAS/LAPACK** 기반
- 많은 과학 계산 라이브러리가 **NumPy**를 기반으로 둠
  - **scipy, matplotlib, pandas, scikit-learn, statsmodels, etc.** 라이브러리 간의 공통 인터페이스
  - **Tools for integrating code written in C, C++, and Fortran**

Scientific Python = SciPy

- **NumPy** 기반 다양한 과학, 공학분야에 활용할 수 있는 함수 제공

Why NumPy ?

NumPy를 사용하는 이유는 다음과 같다**.**

- 속도가 빠르다**. (5**배 **~ 20**배 정도의 성능 차이**)**

- 쉽게 만들어져 사용하기 편리하다. **(Fancy Indexing 등)**
- 효율적인 자료구조 및 알고리즘으로 구성되어있다. **(ndarray)**

Vectorization

- **Arrays are important because they enable you to express batch operations on data without writing any for loops. This is usually called vectorization. Any arithmetic operations between equal-size arrays applies the operation elementwise.**
  - 벡터화하여 계산
- 실제 코딩의 양을 줄일뿐만아니라, 벡터 계산은 병렬 계산이 가능하기 때문에, **Multi core** 활용 가능
  - **CPU** 지원 **(vector processor)**
  - https://blogs.msdn.microsoft.com/nativeconcurrency/2012/04/12/what-is-vectorization/
- https://www.labri.fr/perso/nrougier/from-python-to-numpy/
- **NumPy**는 싱글 코어와 대형 배열에 최적화된 라이브러리라는 한계가 존재
  - 실제로 배열의 크기가 **100**개 이내인 경우 **NumPy**는 순수 파이썬 구현 보다도 오히려 낮은 성능을 보임

Python에서의 array

순수 파이썬에서도 호모지니어스 자료구조인 `array`가 있다.

In [1]:

```
import array
```

그러나 앞에서 언급했듯 성능 차이 때문에 **NumPy**를 사용한다.

NumPy 사용하기

In [2]:

```
import numpy as np
```

본 문서에서 사용하는 현재 `numpy` 버전은 `1.15.4`이다.

In [3]:

```
np.__version__
```

Out[3]:

```
'1.15.4'
```

In [4]:

```
a = np.array(0)
```

In [5]:

```
a
```

Out[5]:

```
array(0)
```

numpy로 만든 **array**의 타입은 `numpy.ndarray`이다.

In [6]:

```
type(a)
```

Out[6]:

```
numpy.ndarray
```

여러 요소를 포함하는 다차원 배열을 선언할 때, **관례적**으로 `list`로 묶어준다.

In [7]:

```
b = np.array([1, 2, 3, 4, 5])
```

In [8]:

```
b
```

Out[8]:

```
array([1, 2, 3, 4, 5])
```

다차원 배열

2차원 배열은 다음과 같이 리스트 안에 리스트를 포함하도록 선언한다.

In [9]:

```
c = np.array([[1, 2, 3], [4, 5, 6]])
```

In [10]:

```
c
```

Out[10]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

ndarray의 차수를 알기 위해서는 다음과 같이 `ndim`을 사용해서 알 수 있다.

In [11]:

```
c.ndim
```

Out[11]:

```
2
```

3차원 배열도 마찬가지로 다음과 같이 선언할 수 있다.

In [12]:

```
d = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]],
              [[10, 11, 12], [13, 14, 15], [16, 17, 18]]])
```

In [13]:

```
d
```

Out[13]:

```
array([[[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9]],

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
```

In [14]:

```
d.ndim
```

Out[14]:

```
3
```

원소의 개수를 알고싶을 때는 `size`를 사용한다.

In [15]:

```
d.size
```

Out[15]:

```
18
```

`shape`은 일반적으로 배열의 현재 모양을 가져 오는 데 사용되지만 배열 크기의 튜플을 할당하여 현재 위치에서 배열을 다시 모양을 바꾸는 데에도 사용할 수 있다. 마찬가지로 `numpy.reshape`처럼 `-1`로 할 경우, 이 값은 배열 크기와 나머지 치수에서 유추하여 크기를 맞춰준다.

`shape`과 `dtype`은 자주 사용하게 될 것이다.

In [16]:

```
d.shape
```

Out[16]:

```
(2, 3, 3)
```

In [17]:

```
e = d.reshape(3, 2, -1)
```

In [18]:

```
e
```

Out[18]:

```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]],

       [[13, 14, 15],
        [16, 17, 18]]])
```

In [19]:

```
e.shape
```

Out[19]:

```
(3, 2, 3)
```

배열의 데이터 타입을 알고싶을 때는 `dtype`을 사용한다.

In [20]:

```
d.dtype
```

Out[20]:

```
dtype('int64')
```

NumPy의 `dtype` 종류는 다음과 같다.

Generic types

NumPy Generic types	
number, inexact, floating	float
complexfloating	cfloat
integer, signedinteger	int_
unsignedinteger	uint
character	string
generic, flexible	void

Built-in Python types

NumPy Generic types	
int	int_
bool	bool_
float	float_
complex	cfloat
bytes	bytes_
str	bytes *(Python2) or unicode* (Python3)
unicode	unicode_
buffer	void
(all others)	object_

다른 `dtype`을 포함시켰을 경우, 한가지 타입으로 통일해준다. 이는 `ndarray`가 `호모지니어스`라는 특성을 가지기 때문이다.

In [21]:

```
f = np.array([1, 2, 3, '4', 5])
```

In [22]:

```
f
```

Out[22]:

```
array(['1', '2', '3', '4', '5'], dtype='<U21')
```

`astype`으로 **64**비트 정수형으로 변경해본다.

In [23]:

```
f = f.astype('int64')
```

In [24]:

```
f
```

Out[24]:

```
array([1, 2, 3, 4, 5])
```

In [25]:

```
f.strides
```

Out[25]:

```
(8,)
```

strides

배열을 순회할 때 각 차원에서 단계별로 이동할 수 있는 튜플이다.

In [26]:

```
g = np.array([[1, 2, 3], [4, 5, 6]])
```

In [27]:

```
g.strides
```

Out[27]:

```
(24, 8)
```

In [28]:

```
g[1, 2]
```

Out[28]:

```
6
```

In [29]:

```
g[1, 0]
```

Out[29]:

```
4
```

In [30]:

```
y = np.reshape(np.arange(2 * 3 * 4), (2, 3, 4))
```

In [31]:

```
y
```

Out[31]:

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

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

In [32]:

```
y.strides
```

Out[32]:

```
(96, 32, 8)
```

In [33]:

```
y[1, 1, 1]
```

Out[33]:

```
17
```

In [34]:

```
offset = sum(y.strides * np.array((1, 1, 1)))
```

In [35]:

```
offset // y.itemsize
```

Out[35]:

```
17
```

slicing

각 축마다 범위를 지정해서 해당 요소를 가져올 수 있다.

In [36]:

```
g
```

Out[36]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [37]:

```
g[:, 0]
```

Out[37]:

```
array([1, 4])
```

In [38]:

```
g[0:1, 1:2]
```

Out[38]:

```
array([[2]])
```

팬시 인덱싱 (Fancy Indexing)

In [39]:

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

In [40]:

```
h
```

Out[40]:

```
array([[[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]],

       [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]])
```

In [41]:

```
h[[0]]
```

Out[41]:

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

In [42]:

```
h[[0, 1]]
```

Out[42]:

```
array([[[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]],

       [[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]])
```

In [43]:

```
h[[0], [1]]
```

Out[43]:

```
array([[4, 5, 6]])
```

In [44]:

```
h[[0]]
```

Out[44]:

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

In [45]:

```
h[[0]].shape
```

Out[45]:

```
(1, 3, 3)
```

In [46]:

```
h[[0]][0, 0:1, 2:]
```

Out[46]:

```
array([[3]])
```

브로드캐스팅 (Broadcasting)

NumPy는 다음과 같이 `브로드캐스팅 연산` 도 지원한다.

In [47]:

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

In [48]:

```
b = np.array([6, 7, 8, 9, 10])
```

In [49]:

```
a + b
```

Out[49]:

```
array([ 7,  9, 11, 13, 15])
```

In [50]:

```
a * b
```

Out[50]:

```
array([ 6, 14, 24, 36, 50])
```

In [51]:

```
a + 3
```

Out[51]:

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

In [52]:

```
b - 1
```

Out[52]:

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

In [53]:

```
np.sum(a)
```

Out[53]:

```
15
```

In [54]:

```
a = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
```

In [55]:

```
a
```

Out[55]:

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

In [56]:

```
np.sum(a)
```

Out[56]:

```
55
```

In [57]:

```
np.sum(a, axis=0)
```

Out[57]:

```
array([ 7,  9, 11, 13, 15])
```

In [58]:

```
np.sum(a, axis=1)
```

Out[58]:

```
array([15, 40])
```

In [59]:

```
b.sum()
```

Out[59]:

```
40
```

In [60]:

```
a.dot(b)
```

Out[60]:

```
array([130, 330])
```

성능 비교

순수 파이썬 리스트

In [61]:

```
%%timeit
a = []
for i in range(1000):
    a.append(i)
sum(a)
```

```
39.8 µs ± 425 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

컴프리헨션

In [62]:

```
%%timeit
sum([x for x in range(1000)])
```

```
20.9 µs ± 266 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

NumPy

In [63]:

```
%%timeit
sum(np.array([x for x in range(1000)]))
```

```
101 µs ± 527 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [64]:

```
%%timeit
np.sum([x for x in range(1000)])
```

```
66.7 µs ± 10.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

In [65]:

```
%%timeit
np.sum(np.arange(1000))
```

```
3.42 µs ± 56.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [66]:

```
a = np.array([x for x in range(1000)])
```

In [67]:

```
%%timeit
np.sum(a)
```

```
2.23 µs ± 36.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

요소가 `100개 미만` 일 경우에는 기존 파이썬 자료구조보다 성능이 좋진 않다. 따라서 데이터가 많으면 많을수록, 즉 `빅데이터` 를 다루게 될 경우에는 `NumPy` 가 더 효율적이다.

In [68]:

```
%%timeit
sum([x for x in range(11)])
```

```
445 ns ± 4.85 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [69]:

```
%%timeit
sum(np.arange(11))
```

```
1.41 µs ± 24.8 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [70]:

```
%%timeit
np.sum(np.arange(11))
```

```
np.sum(np.arange(11))
```

```
2.22 µs ± 35.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```