

Out[1]:

[Click here to toggle on/off the raw code.](#)

▣ One Point Tutorial I - NumPY

Python을 활용한 데이터 사이언스

December, 2019 | All rights reserved by Wooseok Song

1. About NumPy

파이썬은 원래 계산에 특화된 언어가 아니다.

하지만 과학계산 라이브러리 NumPy를 통해 데이터과학, 수치해석, 머신러닝 등 다양한 분야에 적용이 가능해졌다.

!NumPy는

1. Numerical Python의 약자로 전신은 Numerical
2. C언어로 구현되어 빠른 처리속도 보유
3. 2005년 Travis Oliphant가 공개
4. 가장 주목할만한 기능은 ndarray(N-dimensional Array)를 활용한 다차원 벡터화 연산(array 내의 각 원소를 한 번에 처리)
5. 기본적으로 array(vector) 단위로 데이터를 관리/연산
6. Linked list 형태인 리스트 객체와 달리 np.array내 **원소의 자료형(dtype)은 하나로 고정**(R의 Matrix, Vector/Array와 비교)
7. 이는, 원소들이 연속적인 메모리 배치를 갖으며 발생하는 현상으로, 사용은 제한적이나 빠른 내부 처리속도 보유

2. NumPy의 자료형(dtype)

Data type	Description
bool	Boolean (True or False) stored as a byte
int	Platform integer (normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats
complex128	Complex number, represented by two 64-bit floats
string_	String type
object	Object type
unicode_	Unicode type

3. NumPy 객체의 주요 속성(attribute) - np.array(=np.ndarray)

`ndarray.shape`

객체의 **행, 열** 정보를 반환(m행, n열)

`ndarray.size`

객체의 **크기**(# of elements)를 반환($m \times n$)

`ndarray.dtype`

객체의 **data type**을 반환

`ndarray.ndim`

객체(배열)의 **차원**을 반환

4. NumPy 활용(np.array)

- 4.1. 객체 생성
- 4.2. Reshape
- 4.3. Indexing&Slicing을 통한 접근과 변경
- 4.4. 객체 Operation
- 4.5. 복수 객체 Operation
- 4.6. Broadcasting
- 4.7. 그 외 자주 사용하는 NumPy 함수들
- 4.8. 예외값 처리

4.1. 객체 생성

4.1.1. 객체 생성 기본

Out[14]:

```
[(4,), 4, dtype('int32'), 1]
```

Out[19]:

```
[(3, 4), 12, dtype('int64'), 2]
```

4.1.2. 함수를 이용해 객체 생성하는 법

Out[9]:

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

Out[10]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Out[11]:

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

Out[24]:

```
array([0, 2, 4, 6, 8])
```

Out[13]:

```
array([0.19638858, 0.39084464, 0.99046498, 0.87997089, 0.91948237,  
      0.5565053 , 0.40799965, 0.142654 , 0.57696869, 0.01586342])
```

Out[14]:

```
array([ 0.31319626, -0.66104884, -0.50766262, -1.04002042,  0.31546138,  
      -0.78157702, -1.04612217,  2.35724023, -1.04234794,  0.28995275])
```

Out[15]:

```
array([[1., 0., 0., 0.],  
      [0., 1., 0., 0.],  
      [0., 0., 1., 0.],  
      [0., 0., 0., 1.]])
```

Out[16]:

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

Out[50]:

```
array([[1., 0., 0., 0., 0.],  
      [0., 1., 0., 0., 0.],  
      [0., 0., 1., 0., 0.]])
```

Out[49]:

```
array([[6.23042070e-307, 4.67296746e-307, 1.69121096e-306,  
      4.22803078e-307, 6.23044787e-307],  
      [8.45593934e-307, 7.56593017e-307, 1.33511290e-306,  
      1.89146896e-307, 1.11261027e-306],  
      [1.11261502e-306, 1.42410839e-306, 7.56597770e-307,  
      6.23059726e-307, 1.78022342e-306],  
      [6.23058028e-307, 9.34608432e-307, 1.78020848e-306,  
      8.45593934e-307, 9.34611148e-307],  
      [1.24610994e-306, 2.22522596e-306, 0.00000000e+000,  
      8.34402697e-308, 1.95227531e-312]])
```

4.1.3. 함수에 기존 객체를 넣어서 생성하는 방법

Out[32]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[26]:

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

Out[27]:

```
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Out[38]:

```
array([ 2,  7, 12])
```

4.2. Reshape

Out[52]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[53]:

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

Out[54]:

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

Out[55]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Out[57]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

4.3. Indexing&slicing을 통한 접근과 변경

4.3.1. Indexing

Out[58]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[59]:

7

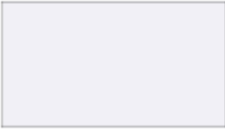

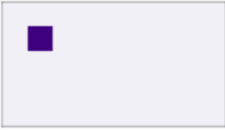
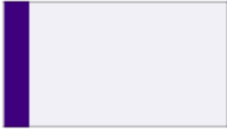
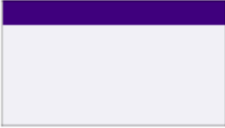



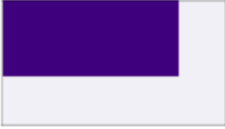
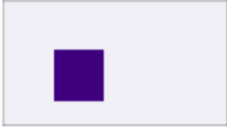

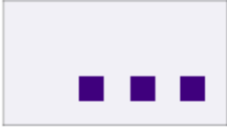
Out[60]:

7

Out[62]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6, 100,  8],
       [ 9, 10, 11, 12]])
```

4.3.2. Slicing (출처 (<https://github.com/rougier/numpy-tutorial>))

Code	Result		Code	Result
<code>Z</code>			<code>Z[...] = 1</code>	
<code>Z[1,1] = 1</code>			<code>Z[:,0] = 1</code>	
<code>Z[0,:] = 1</code>			<code>Z[2:,2:] = 1</code>	
<code>Z[:,::2] = 1</code>			<code>Z[:,2:] = 1</code>	
<code>Z[:-2,:-2] = 1</code>			<code>Z[2:4,2:4] = 1</code>	
<code>Z[:,::2,::2] = 1</code>			<code>Z[3::2,3::2] = 1</code>	

!!!한 번씩 슬라이싱 코드를 쳐보시기 바랍니다.

Out[75]:

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34, 35, 36],
       [37, 38, 39, 40, 41, 42, 43, 44, 45]])
```

4.3.3. Fancy indexing for 1-dim Boolean array

Out[80]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Out[82]:

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

Out[81]:

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

4.3.4. Fancy indexing for 2-dim Integer array

Out[83]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[85]:

```
array([ 5, 11,  8])
```

4.4. 객체 operation

4.4.1. 1-dim array

Out[86]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Out[88]:

78

Out[89]:

6.5

Out[43]:

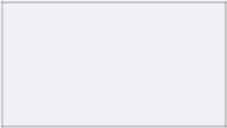

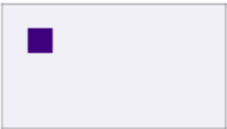
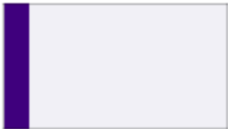








3.452052529534663

Out[90]:

12

4.4.2. 2-dim array

이제 우리의 연산이 어느 방향으로 진행되는 지 암시하는 axis라는 argument가 추가된다.

Code	Result		Code	Result
<code>z</code>			<code>z[...] = 1</code>	
<code>z[1,1] = 1</code>			<code>z[:,0] = 1</code>	
<code>z[0,:] = 1</code>			<code>z[2:,2:] = 1</code>	
<code>z[:,::2] = 1</code>			<code>z[:,::2, :] = 1</code>	
<code>z[:-2, :-2] = 1</code>			<code>z[2:4, 2:4] = 1</code>	
<code>z[:,::2, ::2] = 1</code>			<code>z[3::2, 3::2] = 1</code>	

(출처 (<https://stackoverflow.com/questions/17079279/how-is-axis-indexed-in-numpys-array>))

Out[91]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[92]:

```
array([15, 18, 21, 24])
```

Out[93]:

```
array([10, 26, 42])
```

4.5. 복수 객체 Operation

4.5.1. 1-dim array

Out[49]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

Out[50]:

```
array([13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24])
```

Out[95]:

```
array([14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36])
```

Out[96]:

```
array([-12, -12, -12, -12, -12, -12, -12, -12, -12, -12, -12, -12])
```

Out[98]:

```
array([0.07692308, 0.14285714, 0.2          , 0.25          , 0.29411765,
       0.33333333, 0.36842105, 0.4          , 0.42857143, 0.45454545,
       0.47826087, 0.5          ])
```

Out[99]:

```
array([ 13,  28,  45,  64,  85, 108, 133, 160, 189, 220, 253, 288])
```

Out[100]:

```
1586
```

Out[102]:

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24])
```

4.5.2. 2-dim array

Out[104]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[105]:

```
array([[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

Out[60]:

```
array([[14, 16, 18, 20],
       [22, 24, 26, 28],
       [30, 32, 34, 36]])
```

Out[61]:

```
array([[-12, -12, -12, -12],
       [-12, -12, -12, -12],
       [-12, -12, -12, -12]])
```

Out[62]:

```
array([[0.07692308, 0.14285714, 0.2          , 0.25          ],
       [0.29411765, 0.33333333, 0.36842105, 0.4          ],
       [0.42857143, 0.45454545, 0.47826087, 0.5          ]])
```

Out[63]:

```
array([[ 13,  28,  45,  64],
       [ 85, 108, 133, 160],
       [189, 220, 253, 288]])
```

Out[64]:

```
array([[13, 17, 21],
       [14, 18, 22],
       [15, 19, 23],
       [16, 20, 24]])
```

Out[65]:

```
array([[150, 190, 230],
       [382, 486, 590],
       [614, 782, 950]])
```

Out[108]:

```
array([[150, 190, 230],
       [382, 486, 590],
       [614, 782, 950]])
```

Out[66]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
```

Out[106]:

```
array([[ 1,  2,  3,  4, 13, 14, 15, 16],
       [ 5,  6,  7,  8, 17, 18, 19, 20],
       [ 9, 10, 11, 12, 21, 22, 23, 24]])
```

4.6. Broadcasting

원칙적으로 shape이 다른 array 간의 연산은 불가능하나, broadcasting을 통해 자동으로 형태를 맞춰 연산 가능

Out[109]:

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Out[113]:

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Out[114]:

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

Out[115]:

```
array([[ 3,  6,  9, 12],
       [15, 18, 21, 24],
       [27, 30, 33, 36]])
```

Out[116]:

```
array([[0.33333333, 0.66666667, 1.          , 1.33333333],
       [1.66666667, 2.          , 2.33333333, 2.66666667],
       [3.          , 3.33333333, 3.66666667, 4.          ]])
```

Out[124]:

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

Out[125]:

```
array([[ 2,  3,  4,  5],
       [ 7,  8,  9, 10],
       [12, 13, 14, 15]])
```

Out[129]:

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

4.7. 그 외 자주 사용하는 NumPy 함수들

!꼭 알고갔으면 하는 함수를 2개!

4.7.1 np.where

Out[130]:

```
(array([2, 3], dtype=int64),)
```

Out[132]:

```
array([ 0,  0, 10, 10])
```

Out[135]:

```
(array([1, 1, 2, 2, 2, 2], dtype=int64),
 array([2, 3, 0, 1, 2, 3], dtype=int64))
```

Out[136]:

```
array([[ 0,  0,  0,  0],
       [ 0,  0, 10, 10],
       [10, 10, 10, 10]])
```

4.7.2. np.argmax/min

Out[137]:

3

Out[138]:

11

Out[139]:

```
array([2, 2, 2, 2], dtype=int64)
```

Out[140]:

```
array([3, 3, 3], dtype=int64)
```

4.8. 예외값 처리

목표 : 예외값의 정의 및 분류에 대해 이해하고 이를 처리할 수 있도록 함

Out[92]:

nan

Out[88]:

-inf

```
-----
-
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-141-580a5669e7f6> in <module>
----> 1 0 / 0
```

ZeroDivisionError: division by zero

Out[90]:

nan

Out[91]:

nan

Out[93]:

nan

```
-----  
-  
ZeroDivisionError                                Traceback (most recent call las  
t)  
<ipython-input-94-bc757c3fda29> in <module>  
----> 1 1 / 0  
  
ZeroDivisionError: division by zero
```

```
-----  
-  
ZeroDivisionError                                Traceback (most recent call las  
t)  
<ipython-input-95-f86b194915f6> in <module>  
----> 1 -1 / 0  
  
ZeroDivisionError: division by zero
```

Out[96]:

8.218407461554972e+307

Out[97]:

inf

Out[98]:

0.0

Out[101]:

nan

Out[102]:

inf

Out[103]:

-inf

Out[106]:

(nan+nanj)

Out[108]:

True

Out[112]:

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

Out[149]:

```
array([1., 3.])
```

Out[117]:

```
nan
```

Out[119]:

```
4.0
```

Out[150]:

```
4.0
```

Out[121]:

```
nan
```

Out[151]:

```
3.0
```