11. Numpy

Numpy

Numpy는 Numerical Python의 줄임말로, 파이썬으로 수치 해석이나 통계 관련 작업을 구현할 때 가장 기본이 되는 모듈이며 ndarray라 는 고성능 다차원 배열 객체와 이를 다루는 여러 함수를 제공합니다

Numpy는 상당 부분이 C언어와 포트란으로 작성되어 있기 때문에, 파이 썬 기본 자료 구조를 사용할 때보다 더 빠르게 수치 해석과 통계 관련 작 업을 처리할 수 있습니다. 특히 Numpy에서 제공하는 다차원 배열의 연산 기 능은 인공지능 관련 개발을 할 때 반드시 필요합니다

ndarry 객체의 중요 속성

속성	설명
ndarray.ndim	배열의 축 (차원) 수
ndarray.shape	배열의 차원 각 차원에서 배열의 크기를 나타내는 정수의 튜플 매트릭스와 *n 개의* 행과 *m 개의* 열 (n,m). shape따라서 튜플 의 길이는 축의 수
ndarray.size	배열의 총 요소 수입니다. 이것은의 요소의 곱과 같습니다
ndarray.dtype	배열의 요소 유형을 설명하는 개체 표준 Python 유형을 사용하여 dtype을 만들거나 지정 가능 예) umpy.int32, numpy.int16 및 numpy.float64
ndarray.itemsize	배열의 각 요소 크기 (바이트) 유형의 요소 배열 float64은 itemsize8 (= 64 / 8)이고 유형 omplex32는 itemsize4 (= 32 / 8) ndarray.dtype.itemsize
ndarray.data	배열의 실제 요소를 포함하는 버퍼

ndarry 객체 예제

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
```

```
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray‘>
```

배열 생성

array함수를 사용하여 일반 Python 목록 또는 튜플에서 배열 생성 가능

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

```
>>> a = np.array(1,2,3,4)    # 오류
Traceback (most recent call last):
  ...
TypeError: array() takes from 1 to 2 positional
arguments but 4 were given
>>> a = np.array([1,2,3,4])  # 정상
```

배열 생성

시퀀스 시퀀스를 2 차원 배열로,
시퀀스 시퀀스 시퀀스를 3 차원 배열로 변환

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

배열 유형은 생성시 명시 적으로 지정가능

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

배열 생성

zeros는 0으로 가득 찬 배열을 생성

ones는 1로 가득 찬 empty 배열을 생성

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )              # dtype can also be specified
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)
>>> np.empty( (2,3) )                               # uninitialized
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

배열 생성

arange : 일련의 숫자를 생성 배열

```
>>> 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])
```

linspace : 유한 부동 소수점을 이용한 배열 생성

```
>>> from numpy import pi
>>> 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*pi, 100 )        # useful to evaluate function at lots of points
>>> f = np.sin(x)
```

배열 출력

배열을 인쇄 할 때 NumPy는 중첩 된 목록과 비슷한 방식으로 배열을 표시하지만 다음 레이아웃을 사용합니다.

- 마지막 축은 왼쪽에서 오른쪽으로 인쇄.
- 마지막에서 두 번째는 위에서 아래로 인쇄.
- 나머지는 또한 위에서 아래로 인쇄되며 각 슬라이스는 빈 줄로 다음 슬라이스와 구분.

배열 출력

다음 1 차원 배열은 행으로, 2 차원은 행렬로, 3 차원은 행렬 목록으로 출력됨

```
>>> 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]]]
```

배열 출력

배열이 너무 커서 인쇄 할 수 없는 경우 NumPy는 배열의 중앙 부분을 자동으로 건너 뛰고 모서리 만 출력

```
>>> print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

전

```
>>> np.set_printoptions(threshold=sys.maxsize)   # sys module should be imported
```

기본 연산

배열의 산술 연산자는 요소별로 적용됩니다 . 새 배열이 생성되고 결과로 채워 짐

```
>>> 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])
```

기본 연산

product 연산자 *는 NumPy 배열에서 요소별로 작동.

행렬 곱은 @연산자 (python> = 3.5) 또는 dot함수 또는 방법을 사용하여 수행 가능

```
>>> A = np.array( [[1,1],
...               [0,1]] )
>>> B = np.array( [[2,0],
...               [3,4]] )
>>> A * B                     # elementwise 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]])
```

기본 연산

+=및 *= 연산자는 기존 배열을 수정 함.

```
>>> rg = np.random.default_rng(1)     # create instance of default random number
generator
>>> a = np.ones((2,3), dtype=int)
>>> b = rg.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b                          # b is not automatically converted to integer type
Traceback (most recent call last):
  ...
numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add' output
from dtype('float64') to dtype('int64') with casting rule 'same_kind'
```

기본연산

다른 자료형간의 연산은 업케스팅됨

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

기본연산

sum,min.max 메서드

```
>>> a = rg.random((2,3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
       [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.105710952998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.827702593820441 8
```

기본연산

axis 매개 변수 : 지정된 축을따라 실행

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                          # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                         # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                          # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Universal 함수

NumPy는 sin, cos 및 exp와 같은 수학 함수 제공

NumPy에서는이를 Universal 함수 라 함.

요소별로 연산함.

```
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([1.        , 2.71828183, 7.3890561 ])
>>> np.sqrt(B)
array([0.        , 1.        , 1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([2., 0., 6.])
```

Indexing, Slicing and Iterating

1 차원 배열은 목록 및 기타 Python 시퀀스와 Indexing, Slicing and Iterating 가능

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
```

Indexing, Slicing and Iterating

1 차원 배열은 목록 및 기타 Python 시퀀스와 Indexing, Slicing and Iterating 가능

```
# equivalent to a[0:6:2] = 1000;
# from start to position 6, exclusive, set every 2nd element to 1000
>>> a[:6:2] = 1000
>>> a
array([1000,    1, 1000,   27, 1000,  125,  216,  343,  512,  729])
>>> a[ : :-1]                                # reversed a
array([ 729,  512,  343,  216,  125, 1000,   27, 1000,    1, 1000])
```

```
>>> for i in a:
...     print(i**(1/3.))
...
9.999999999999998
1.0
9.999999999999998
3.0
.......
```

Indexing, Slicing and Iterating

다차원 배열은 축당 하나의 인덱스를 가짐

인덱스는 쉼표로 구분 된 튜플로 제공

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
```

```
>>> b[0:5, 1]                    # each row in the
second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                     # equivalent to the
previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                   # each column in the
second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Indexing, Slicing and Iterating

축 수보다 적은 인덱스가 제공되면 누락 된 인덱스는 완전한 걸로 간주됨

```
>>> b[-1]                                # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

괄호 안의 표현식 은 나머지 축을 나타내는 데 필요한만큼의 인스턴스가 뒤에 오는 것으로 b[i]처리 됩니다 . NumPy를 사용하면 점을 .i:b[i,...] 사용

x[1,2,...]는 x[1,2,:,:,:],

x[...,3]에 x[:,:,:,:,3]및

x[4,...,5,:]에 x[4,:,:,5,:].

Indexing, Slicing and Iterating

```
>>> c = np.array( [[[  0,  1,  2],          # a 3D array (two stacked 2D arrays)
...                [ 10, 12, 13]],
...               [[100,101,102],
...                [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                               # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                               # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

Indexing, Slicing and Iterating

다차원 배열에 대한 반복 은 첫 번째 축과 맞춰 실행

배열의 모든 요소에 flat대한 iterating 속성 사용 가능

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
....
42
43
```

배열 모양 변경

배열은 각 축을 따라있는 요소의 수로 주어진 모양을 갖음

```
>>> a = np.floor(10*rg.random((3,4)))
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.shape
(3, 4)
```

배열 모양 변경

세 명령은 모두 수정 된 배열을 반환하지만 원래 배열은 변경하지 않음

```
>>> a.ravel()  # returns the array, flattened
array([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
>>> a.reshape(6,2)  # returns the array with a
modified shape
array([[3., 7.],
       [3., 4.],
       [1., 4.],
       [2., 2.],
       [7., 2.],
       [4., 9.]])
>>> a.T  # returns the array, transposed
array([[3., 1., 7.],
       [7., 4., 2.],
       [3., 2., 4.],
       [4., 2., 9.]])
```

```
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

배열 모양 변경

reshape함수는 변경 된 모양으로 인수를 반환

ndarray.resize메서드는 배열 자체를 수정

```
>>> a
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
>>> a.resize((2,6))
>>> a
array([[3., 7., 3., 4., 1., 4.],
       [2., 2., 7., 2., 4., 9.]])
```

배열 모양 변경

모양 변경 작업에서 차원이 -1로 지정되면 다른 차원이 자동으로 계산됨.

```
>>> a.reshape(3,-1)
array([[3., 7., 3., 4.],
       [1., 4., 2., 2.],
       [7., 2., 4., 9.]])
```

다른 배열을 합하기

여러 배열을 서로 다른 축을 따라 함께 합할 수 있음

```
>>> a = np.floor(10*rg.random((2,2)))
>>> a
array([[9., 7.],
       [5., 2.]])
>>> b = np.floor(10*rg.random((2,2)))
>>> b
array([[1., 9.],
       [5., 1.]])
>>> np.vstack((a,b))
array([[9., 7.],
       [5., 2.],
       [1., 9.],
       [5., 1.]])
>>> np.hstack((a,b))
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
```

다른 배열을 합하기

column_stack 1D 배열을 열로 2D 배열로 합함. 단 hstack2D 배열에만 해당됨 .

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[4., 3.],
       [2., 8.]])
```

다른 배열을 합하기

column_stack 1D 배열을 열로 2D 배열로 합함. 단 hstack2D 배열에만 해당됨 .

```
>>> np.hstack((a,b))          # the result is different
array([4., 2., 3., 8.])
>>> a[:,newaxis]              # view `a` as a 2D column vector
array([[4.],
       [2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[4., 3.],
       [2., 8.]])
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[4., 3.],
       [2., 8.]])
```

다른 배열을 합하기

row_stack는 vstack 모든 입력 배열 과 동일

row_stack은 vstack 별칭.

```
>>> np.column_stack is np.hstack
False
>>> np.row_stack is np.vstack
True
```

일반적으로 차원이 3 개 이상인 배열의 경우 두 hstack번째 축을 vstack따라 스택하고 첫 번째 축을 따라 스택 concatenate 하며 연결이 발생해야하는 축의 수를 제공하는 선택적 인수를 허용합니다.

다른 배열을 합하기

복잡한 경우에, r_그리고 c_하나 개의 축을 따라 번호를 적층하여 배열을 만들 때 유용.

범위 리터럴 (":")을 사용할 수 있음

```
>>> np.r_[1:4,0,4]
array([1, 2, 3, 0, 4])
```

인수로 배열을 사용하는 경우 r_와 c_유사
vstack및 hstack기본 동작하지만, 연결하는 따라 축의 수를 지정하는 옵션 지정 가능

하나의 배열을 여러 개의 작은 배열로 나누기

hsplit 를 사용하면 반환 할 동일한 모양의 배열 수를 지정하거나 나눠지는 열을 지정하여 가로 축을 따라 배열을 분할 할 수 있음

```
>>> a = np.floor(10*rg.random((2,12)))
>>> a
array([[6., 7., 6., 9., 0., 5., 4., 0., 6., 8., 5., 2.],
       [8., 5., 5., 7., 1., 8., 6., 7., 1., 8., 1., 0.]])
# Split a into 3
>>> np.hsplit(a,3)
[array([[6., 7., 6., 9.],
       [8., 5., 5., 7.]]), array([[0., 5., 4., 0.],
       [1., 8., 6., 7.]]), array([[6., 8., 5., 2.],
       [1., 8., 1., 0.]])]
# Split a after the third and the fourth column
```

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

vsplit수직 축을 array_split 따라 분할하고 분할 할 축을 지정할 수 있음

복사 및 보기

단순 할당 : 데이터의 복사본을 만들지 않음

```
>>> a = np.array([[ 0,  1,  2,  3],
...               [ 4,  5,  6,  7],
...               [ 8,  9, 10, 11]])
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the
same ndarray object
True
```

함수 호출은 복사본을 만들지 않음

```
>>> def f(x):
...     print(id(x))
...
>>> id(a)                       # id is a unique identifier of an object
148293216  # may vary
>>> f(a)
148293216  # may vary
```

복사 및 보기

보기 또는 얕은 복사

- 다른 배열 객체는 동일한 데이터를 공유.
- view메서드는 동일한 데이터를 보는 새 배열 개체 생성.

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a        # c is a view of the data owned by a
True
>>> c.flags.owndata
False
>>>
>>> c = c.reshape((2, 6))    # a's shape doesn't change
>>> a.shape
(3, 4)
```

```
>>> c[0, 4] = 1234      # a's data changes
>>> a
array([[   0,    1,    2,    3],
       [1234,    5,    6,    7],
       [   8,    9,   10,   11]])
```

복사 및 보기

배열을 분할하면 보기가 반환

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:, 1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s = 10 and s[:] =
10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

깊은 복사

copy메서드 : 배열과 데이터의 완전한 복사본

```
>>> d = a.copy()                          # a new array object with new data is created
>>> d is a
False
>>> d.base is a                           # d doesn't share anything with a
False
>>> d[0,0] = 9999
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

copy원래 배열이 더 이상 필요하지 않은 경우 슬라이스 후 때때로 호출해야합니다. 예를 들어 a큰 중간 결과가 있고 최종 결과 b에의 작은 부분 만 포함되어 있다고 가정 a하면 b슬라이싱으로 구성 할 때 깊은 복사본을 만들어야합니다 .

깊은 복사

대신 사용되는 경우에서 참조되며 실행 되더라도 메모리에 유지됩니다 .

b = a[:100]

del a

```
>>> a = np.arange(int(1e8))
>>> b = a[:100].copy()
>>> del a  # the memory of ``a`` can be released.
```

함수 및 방법 개요

배열생성(**Array Creation**)

arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r_, zeros, zeros_like

전환(**Conversions**)

ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat

조작(**Manipulations**)

array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack

질문(**Questions**)

all, any, nonzero, where

함수 및 방법 개요

주문(Ordering)

argmax, argmin, argsort, max, min, ptp, searchsorted, sort

운영(Operations)

choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum

기본 통계(Basic Statistics)

cov, mean, std, var

기본 선형 대수(Basic Linear Algebra)

cross, dot, outer, linalg.svd, vdot

고급 인덱싱 및 인덱스 트릭

NumPy는 일반 Python 시퀀스보다 더 많은 인덱싱 기능을 제공합니다. 정수와 슬라이스로 인덱싱하는 것 외에도, 앞에서 본 것처럼 배열은 정수 배열과 부울 배열로 인덱싱 할 수 있습니다

인덱스 배열로 인덱싱

덱스 배열 a가 다차원 인 경우 인덱스의 단일 배열 a 은 a의 첫 번째 차원을 참조

```
>>> a = np.arange(12)**2                  # the first 12 square numbers
>>> i = np.array([1, 1, 3, 8, 5])         # an array of indices
>>> a[i]                                  # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
>>>
>>> j = np.array([[3, 4], [9, 7]])      # a bidimensional array of indices
>>> a[j]                                  # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

인덱스 배열로 인덱싱

팔레트를 사용하여 레이블 이미지를 컬러 이미지로 변환하여이 동작을 보여줍니다.

```
>>> palette = np.array([[0, 0, 0],         # black
...                     [255, 0, 0],       # red
...                     [0, 255, 0],       # green
...                     [0, 0, 255],       # blue
...                     [255, 255, 255]])  # white
>>> image = np.array([[0, 1, 2, 0],       # each value corresponds to a color in the
palette
...                   [0, 3, 4, 0]])
```

```
>>> palette[image]          # the (2, 4, 3) color image
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],

       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

인덱스 배열로 인덱싱

2차원 이상의 인덱스를 제공 단 각 차원에 대한 인덱스 배열의 모양은 동일해야함

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> i = np.array([[0, 1],                        # indices for the first dim of a
...               [1, 2]])
>>> j = np.array([[2, 1],                        # indices for the second dim
...               [3, 3]])
>>>
```

인덱스 배열로 인덱싱

```
>>> a[i, j]                    # i and j must have equal shape
array([[ 2,  5],
       [ 7, 11]])
>>>
>>> a[i, 2]
array([[ 2,  6],
       [ 6, 10]])
>>>
>>> a[:, j]                     # i.e., a[ : , j]
array([[[ 2,  1],
        [ 3,  3]],

       [[ 6,  5],
        [ 7,  7]],

       [[10,  9],
        [11, 11]]])
```

인덱스 배열로 인덱싱

파이썬에서 arr[i, j] 정확히 arr[(i, j)]와 동일하다

```
>>> l = (i, j)
# equivalent to a[i, j]
>>> a[l]
array([[ 2,  5],
       [ 7, 11]])
```

인덱스 배열로 인덱싱

```
>>> s = np.array([i, j])

# not what we want
>>> a[s]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3

# same as a[i, j]
>>> a[tuple(s)]
array([[ 2,  5],
       [ 7, 11]])
```

인덱스 배열로 인덱싱

배열을 사용한 인덱싱의 또 다른 일반적인 용도는 시간 종속 계열의 최대 값을 검색하는 것

```
>>> time = np.linspace(20, 145, 5)                 # time scale
>>> data = np.sin(np.arange(20)).reshape(5,4)      # 4 time-dependent series
>>> time
array([ 20.  ,  51.25,  82.5 , 113.75, 145.  ])
>>> data
array([[ 0.        ,  0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ,  0.6569866 ],
       [ 0.98935825,  0.41211849, -0.54402111, -0.99999021],
       [-0.53657292,  0.42016704,  0.99060736,  0.65028784],
       [-0.28790332, -0.96139749, -0.75098725,  0.14987721]])

# index of the maxima for each series
>>> ind = data.argmax(axis=0)
>>> ind
array([2, 0, 3, 1])
```

인덱스 배열로 인덱싱

배열을 사용한 인덱싱의 또 다른 일반적인 용도는 시간 종속 계열의 최대 값을 검색하는 것

```
# times corresponding to the maxima
>>> time_max = time[ind]
>>>
>>> data_max = data[ind, range(data.shape[1])] # => data[ind[0],0], data[ind[1],1]...

>>> time_max
array([ 82.5 ,  20.  , 113.75,  51.25])
>>> data_max
array([0.98935825, 0.84147098, 0.99060736, 0.6569866 ])

>>> np.all(data_max == data.max(axis=0))
True
```

인덱스 배열로 인덱싱

배열의 인덱스를 이용하여 값을 할당 할 수 있음

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

인덱스가 반복된 경우 마지막 값이 할당됨.

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

인덱스 배열로 인덱싱

Python의 +=구조

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

인덱스 목록에서 0이 두 번 발생하더라도 0번째 요소는 한 번만 증가합니다

부울 배열로 인덱싱

(정수) 인덱스 배열로 배열을 인덱싱 할 때 선택할 인덱스 목록을 제공합니다. 부울 인덱스를 사용하면 접근 방식이 다릅니다. 배열에서 원하는 항목과 원하지 않는 항목을 명시 적으로 선택합니다.

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                                  # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]])
>>> a[b]                               # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

부울 배열로 인덱싱

```
>>> a[b] = 0                    # All elements of 'a' higher than 4 become 0
>>> a
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```

부울 배열로 인덱싱

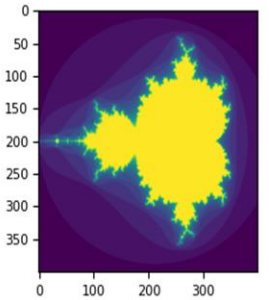
부울 인덱싱을 사용하여 Mandelbrot 집합 이미지 생성

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def mandelbrot( h,w, maxit=20 ):
    """Returns an image of the Mandelbrot fractal of size (h,w)."""
    y,x = np.ogrid[ -1.4:1.4:h*1j, -2:0.8:w*1j ]
    c = x+y*1j
    z = c
    divtime = maxit + np.zeros(z.shape, dtype=int)

    for i in range(maxit):
        z = z**2 + c
        diverge = z*np.conj(z) > 2**2          # who is diverging
        div_now = diverge & (divtime==maxit)  # who is diverging now
        divtime[div_now] = i                  # note when
        z[diverge] = 2                        # avoid diverging too much

    return divtime
```

```
>>> plt.imshow(mandelbrot(400,400))
```



부울 배열로 인덱싱

부울을 사용하여 인덱싱하는 두 번째 방법은 정수 인덱싱과 더 유사합니다. 배열의 각 차원에 대해 원하는 슬라이스를 선택하는 1D 부울 배열을 제공합니다

```
>>> a = np.arange(12).reshape(3,4)
>>> b1 = np.array([False,True,True])          # first dim selection
>>> b2 = np.array([True,False,True,False])     # second dim selection
>>>
>>> a[b1,:]                   # selecting rows
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> a[b1]                     # same thing
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
>>>
>>> a[:,b2]                   # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                  # a weird thing to do
array([ 4, 10])
```

ix_ () 함수

이 ix_함수는 각 n-uplet에 대한 결과를 얻기 위해 다른 벡터를 결합하는 데 사용할 수 있다.

예를 들어, 각 벡터 a, b 및 c에서 가져온 모든 트리플렛에 대해 모든 a + b * c를 계산하는 경우

```
>>> a = np.array([2,3,4,5])
>>> b = np.array([8,5,4])
>>> c = np.array([5,4,6,8,3])
>>> ax,bx,cx = np.ix_(a,b,c)
>>> ax
array([[[2]],

       [[3]],

       [[4]],

       [[5]]])
```

```
>>> bx
array([[[8],
        [5],
        [4]]])
>>> cx
array([[[5, 4, 6, 8, 3]]])
>>> ax.shape, bx.shape, cx.shape
((4, 1, 1), (1, 3, 1), (1, 1, 5))
```

ix_ () 함수

계속

```
>>> result = ax+bx*cx
>>> result
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],

       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],

       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],

       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
```

```
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

ix_ () 함수

축소하여 구현하기

```
>>> def ufunc_reduce(ufct, *vectors):
...     vs = np.ix_(*vectors)
...     r = ufct.identity
...     for v in vs:
...         r = ufct(r,v)
...     return r
```

```
>>> ufunc_reduce(np.add,a,b,c)
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],

       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],

       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],

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

선형 대수

간단한 배열 작업

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[1. 2.]
 [3. 4.]]

>>> a.transpose()
array([[1., 3.],
       [2., 4.]])

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

선형 대수

간단한 배열 작업

```
>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[1., 0.],
       [0., 1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> j @ j        # matrix product
array([[-1.,  0.],
       [ 0., -1.]])

>>> np.trace(u)  # trace
2.0

>>> y = np.array([[5.], [7.]])
>>> np.linalg.solve(a, y)
array([[-3.],
       [ 4.]])
```

선형 대수

계속

```
>>> np.linalg.eig(j)
(array([0.+1.j, 0.-1.j]), array([[0.70710678+0.j        , 0.70710678-0.j        ],
       [0.        -0.70710678j, 0.        +0.70710678j]]))
```

```
Parameters:
    square matrix
Returns
    The eigenvalues, each repeated according to its multiplicity.
    The normalized (unit "length") eigenvectors, such that the
    column ``v[:,i]`` is the eigenvector corresponding to the
    eigenvalue ``w[i]`` .
```

트릭과 팁

"자동"모양 변경 : 배열의 크기를 변경하려면 자동으로 추론되는 크기 중 하나를 생략 할 수 있다

```
>>> a = np.arange(30)
>>> b = a.reshape((2, -1, 3))  # -1 means "whatever is needed"
>>> b.shape
(2, 5, 3)
>>> b
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],
        [24, 25, 26],
        [27, 28, 29]]])
```

벡터 쌓기

동일한 크기의 행 벡터 목록에서 2D 배열 구성 방법

- MATLAB을 이용
- x및 y길이가 같은 두 개의 벡터 인 경우 m=[x;y]. NumPy기능을 통해 작동
- column_stack, dstack, hstack및 vstack치수에 따라하는 스태킹 할 것

```
>>> x = np.arange(0,10,2)
>>> y = np.arange(5)
>>> m = np.vstack([x,y])
>>> m
array([[0, 2, 4, 6, 8],
       [0, 1, 2, 3, 4]])
>>> xy = np.hstack([x,y])
>>> xy
array([0, 2, 4, 6, 8, 0, 1, 2, 3, 4])
```

히스토그램

histogram배열에 적용된 NumPy 함수는 벡터 쌍, 즉 배열의 히스토그램과 빈 가장자리의 벡터를 반환.

주의 : matplotlib또한 histNumPy의 것과 다른 히스토그램 (Matlab에서와 같이 라고 함)을 작성하는 기능이 있다.

주요 차이점은 pylab.hist히스토그램을 자동으로 플로팅 numpy.histogram하고 데이터 만 생성한다는 것.

히스토그램

```
>>>import numpy as np
>>> rg = np.random.default_rng(1)
>>> import matplotlib.pyplot as plt
>>> # Build a vector of 10000 normal deviates >>> with variance 0.5^2 and mean 2
>>> mu, sigma = 2, 0.5
>>> v = rg.normal(mu,sigma,10000)
>>> # Plot a normalized histogram with 50 bins
>>> plt.hist(v, bins=50, density=1)       # matplotlib version (plot)
>>> # Compute the histogram with numpy and then plot it
>>>(n, bins) = np.histogram(v, bins=50, density=True)  # NumPy version (no plot)
>>> plt.plot(.5*(bins[1:]+bins[:-1]), n)
```