02. NpmPy 기본: 배열과 벡터연산

1. NumPy ndarray: 다차원 배열 객체
2. 유니버설함수 : 배열의 각 원소를 빠르게 처리하는 함수
3. 배열을 이용한 배열 지향 프로그래밍

1. NumPy ndarray: 다차원 배열 객체

NumPy란?

- Numerical Python
  - 과학 계산을 위한 대부분의 패키지는 NumPy의 배열 객체를 데이터 교환을 위한 공통 언어 처럼 사용한다.

- NumPy에서 제공하는 것들
  - 빠른 배열 계산과 유연한 **브로드캐스팅** 기능
  - 반복문을 작성할 필요 없이 전체 데이터 배열을 빠르게 계산할 수 있는 표준 수학 함수
  - 배열 데이터를 디스크에 쓰거나 읽을 수 있는 도구와 메모리에 적재된 파일을 다루는 도구
  - 선형대수, 난수 생성기, 푸리에 변환 기능
  - C, C++, 포트란으로 작성한 코드를 연결하 수 있는 C API

NumPy란?

- 대용량 데이터 배열을 효율적으로 다룰 수 있도록 설계됨
  - NumPy는 데이터를 다른 내장 파이석 객체와 구분되는 연속된 메모리 블록에 저장
  - NumPy의 각종 알고리즘은 모두 C로 작성되어 타입 검사나 다른 오버헤드 없이 메모리를 직접 조작할 수 있음
  - NumPy 배열은 내장 파이선의 연속된 자료형들보다 훨씬 더 적은 메모리를 사용함
  - NumPy 연산은 파이선 반복문을 사용하지 않고 전체 배열에 대한 복잡한 계산을 수행함

NumPy 기본

- 기본설정
  - 아래 설정은 본 자료의 마지막까지 적용됨

```
import numpy as np
np.random.seed(12345)
import matplotlib.pyplot as plt
plt.rc('figure', figsize=(10, 6))
np.set_printoptions(precision=4, suppress=True)
```

  - 본 자료의 예제 작성 시 주의사항
    - jupyter notebook에서 하나의 파일에 작성하기
    - 각 코드 셀의 실행 결과가 다르게 나오는 경우
      - Cell>Run All을 실행하여 본 자료의 실행 순서와 일치시킬 것

NumPy 기본

- NumPy 배열과 Python 리스트 성능 비교

```
import numpy as np
my_arr = np.arange(1000000)      # numpy array
my_list = list(range(1000000))   # python list
```

```
%time for _ in range(10): my_arr2 = my_arr * 2
```

```
Wall time: 22.9 ms
```

```
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

```
Wall time: 868 ms
```

NumPy ndarray : 다차원 배열 객체

```
import numpy as np
# Generate some random data
data = np.random.randn(2, 3)
data
```

```
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

```
data * 10
```

```
array([[-2.0471,  4.7894, -5.1944],
       [-5.5573, 19.6578, 13.9341]])
```

```
data + data
```

```
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

NumPy ndarray : 다차원 배열 객체

- ndarray
  - 같은 종류의 데이터를 담을 수 있는 포괄적인 다차원 배열
  - 모든 원소는 같은 자료형이어야 함

```
data.shape
```

배열의 각 차원 크기를 알려 줌

```
(2, 3)
```

```
data.dtype
```

배열에 저장된 자료형을 알려줌

```
dtype('float64')
```

ndarray 생성하기

- array 함수
  - 배열 혹은 순차적인 객체를 전달 받아 새로운 NumPy 배열 생성

```
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
arr1
```

```
array([6. , 7.5, 8. , 0. , 1. ])
```

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
arr2
```

해당 데이터로부터 형태를 추론하여 2차원 형태로 생성됨

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

ndarray 생성하기

```
arr2.ndim
```

차원

```
2
```

```
arr2.shape
```

```
(2, 4)
```

```
arr1.dtype
```

명시적으로 지정하지 않는 한 np.array는 생성될 때 적절한 자료형을 추론함

```
dtype('float64')
```

```
arr2.dtype
```

```
dtype('int32')
```

ndarray 생성하기

- zeros, ones 함수
  - 주어진 길이나 모양에 각각 0과 1이 들어 있는 배열을 생성

```
np.zeros(10)
```

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

```
np.zeros((3, 6))
```

원하는 형태의 튜플을 함수로 전달

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

ndarray 생성하기

- empty 함수
  - 초기화 되지 않은 배열 생성
  - 대부분의 경우 가비지로 채워진 배열을 반환

```
np.empty((2, 3, 2))
```

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

       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

ndarray 생성하기

- arrange 함수
  - 파이선 range 함수의 배열 버전

```
np.arange(15)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 1
0, 11, 12, 13, 14])
```

ndarray의 dtype

- dtype
  - ndrarray가 메모리에 있는 특정 데이터를 해석하기 위해 필요한 정보 (또는 메타데이터)를 담고 있는 특수한 객체

```
arr1 = np.array([1, 2, 3], dtype=np.float64)
arr2 = np.array([1, 2, 3], dtype=np.int32)
arr1.dtype
```

```
dtype('float64')
```

```
arr2.dtype
```

```
dtype('int32')
```

ndarray의 dtype

- NumPy 자료형

자료형	자료형 코드	설명
int8, uint8	i1, u1	부호가 있는 8비트(1바이트) 정수형과 부호가 없는 8비트 정수형
int16, uint16	i2, u2	부호가 있는 16비트 정수형과 부호가 없는 16비트 정수형
int32, uint32	i4, u4	부호가 있는 32비트 정수형과 부호가 없는 32비트 정수형
int64, uint64	i8, u8	부호가 있는 64비트 정수형과 부호가 없는 64비트 정수형
float16	f2	반정밀도 부동소수점
float32	f4 또는 f	단정밀도 부동소수점. C언어의 float형과 호환
float64	f8 또는 d	배정밀도 부동소수점. C언어의 double형과 파이썬의 float 객체와 호환
float128	f16 또는 g	확장정밀도 부동소수점
complex64, complex128, complex256	c8, c16, c32	각각 2개의 32, 64, 128비트 부동소수점형을 가지는 복소수
bool	?	True와 False 값을 저장하는 불리언형
object	O	파이썬 객체형
string_	S	고정 길이 아스키 문자열형(각 문자는 1바이트). 길이가 10인 문자열 dtype은 S10이 된다.
unicode_	U	고정 길이 유니코드형(플랫폼에 따라 문자별 바이트 수가 다르다). string_형과 같은 형식을 쓴다(예: U10).

ndarray의 dtype

- astype 메소드
  - 명시적 타입 변환 (캐스팅)
  - 정수형 → 부동 소수형

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

```
dtype('int32')
```

```
float_arr = arr.astype(np.float64)
float_arr.dtype
```

```
dtype('float64')
```

ndarray의 dtype

- 부동 소수형 → 정수형

```
arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
arr
```

```
array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])
```

```
arr.astype(np.int32)
```

```
array([ 3, -1, -2,  0, 12, 10])
```

소수점 아래 자리는 버려짐

ndarray의 dtype

- 숫자 문자열 → 숫자

```
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
numeric_strings.astype(float)
```

```
array([ 1.25, -9.6 , 42.  ])
```

※ 주의: NumPy에서 문자열 데이터는 고정 크기를 가지며 별다른 경고를 출력하지 않고 입력을 임의로 잘라낼 수 있음

- 다른 배열의 dtype 속성을 이용

```
int_array = np.arange(10)
calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
int_array.astype(calibers.dtype)
```

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

※ 주의: astype을 호출하면 새로운 dtype이 이전 dtype과 동일해도 항상 새로운 배열을 생성(데이터를 복사)한다.

ndarray의 dtype

- dtype으로 사용할 수 있는 '축약 코드'

```
empty_uint32 = np.empty(8, dtype='u4')
empty_uint32
```

uint32

```
array([         0, 1075314688,          0, 1075707904,          0,
       1075838976,          0, 1072693248], dtype=uint32)
```

NumPy 배열의 산술 연산

- 배열의 중요한 특징 : 벡터화
  - for문을 작성하지 않고 데이터를 일괄 처리함
- 같은 크기의 배열 간의 산술 연산
  - 배열의 각 원소 단위로 적용됨

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

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

```
arr * arr
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

```
arr - arr
```

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

NumPy 배열의 산술 연산

- 스칼라 인자가 포함된 산술 연산
  - 배열 내 모든 원소에 스칼라 인자가 적용됨

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

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

```
1 / arr
```

```
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])
```

```
arr ** 0.5
```

```
array([[1.    , 1.4142, 1.7321],
       [2.    , 2.2361, 2.4495]])
```

NumPy 배열의 산술 연산

- 같은 크기의 배열 간의 비교 연산
  - 불리언 배열 반환

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

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

```
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
arr2
```

```
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])
```

```
arr2 > arr
```

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

※ 브로드캐스팅 : 크기가 다른 배열 간의 연산

색인과 슬라이싱 기초

- 1차원 배열

```
arr = np.arange(10)
arr
```

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

```
arr[5]
```

```
5
```

```
arr[5:8]
```

슬라이싱(slicing)

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

```
arr[5:8] = 12
arr
```

배열 조각에 스칼라값을 대입하면 12가 선택 영역 전체로 전파(브로드캐스팅)

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

색인과 슬라이싱 기초

- 배열 조각은 원본 배열의 '**뷰**'
  - 데이터는 복사되지 않고
  - 뷰에 대한 변경은 그대로 원본 배열에 반영됨

```
arr_slice = arr[5:8]
arr_slice
```

※ 주의: ndarray 슬라이스의 복사본 얻기
arr[5:8].copy()를 사용해야 함

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

```
arr_slice[1] = 12345
arr
```

```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

```
arr_slice[:] = 64
arr
```

배열의 모든 값

```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

색인과 슬라이싱 기초

- 2차원 배열
  - 각 색인에 해당하는 요소는 스칼라값이 아닌 1차원 배열

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2]
```

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

```
arr2d[0][2]
```

색인
(개별 요소는 재귀적으로 접근)

```
3
```

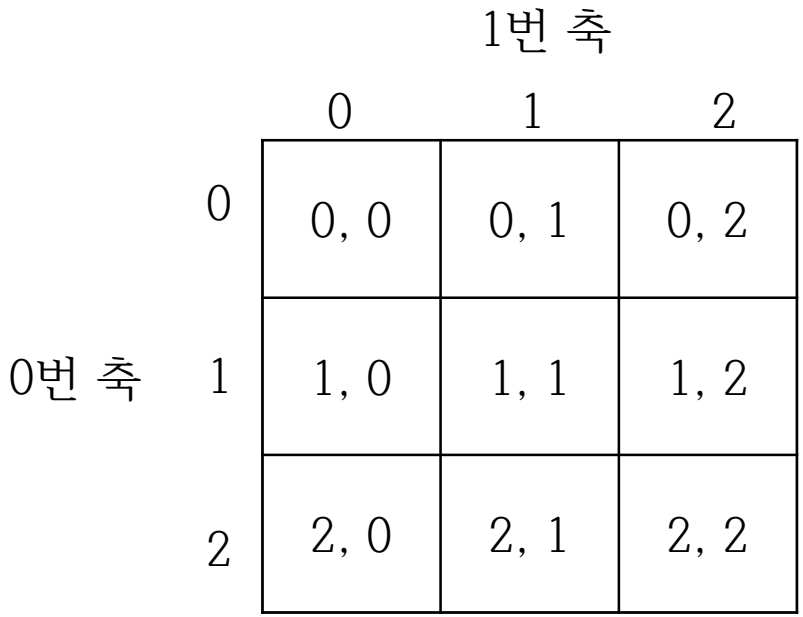
```
arr2d[0, 2]
```

,로 구분된 색인 리스트를 넘김

```
3
```

색인과 슬라이싱 기초

- 2차원 배열의 색인
  - 0번 축 : row
  - 1번 축 : column



색인과 슬라이싱 기초

- 3차원 배열
  - 2 × 2 × 3 크기 예

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3d
```

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

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

```
arr3d[0]
```

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

2 × 3 크기의 배열

색인과 슬라이싱 기초

```
old_values = arr3d[0].copy()
arr3d[0] = 42          스칼라 대입
arr3d
```

```
array([[[42, 42, 42],
        [42, 42, 42]],

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

```
arr3d[0] = old_values   배열 대입
arr3d
```

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

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

```
arr3d[1, 0]          (1, 0)으로 색인되는 1차원 배열 반환
```

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

색인과 슬라이싱 기초

- arr3d[1, 0]은 다음 2번에 걸친 인덱싱과 같다.

```
x = arr3d[1]
x
```

```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

```
x[0]
```

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

슬라이스로 선택하기

- 1차원 배열

```
arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

```
arr[1:6]
```

```
array([ 1,  2,  3,  4, 12])
```

슬라이스로 선택하기

- 2차원 배열

```
arr2d
```

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

```
arr2d[:2]
```

0번 축을 기준으로 슬라이싱
(축을 따라 선택 영역 내의 요소를 선택)

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

2차원 배열

  - 다차원 슬라이싱
    - 항상 같은 차원의 배열에 대한 뷰를 얻음

```
arr2d[:2, 1:]
```

2차원 슬라이싱

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

2차원 배열

슬라이스로 선택하기

- 정수 색인과 슬라이스를 함께 사용
  - 한 차원 낮은 슬라이스를 얻음

```
arr2d[1, :2]
```

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

1차원 배열

```
arr2d[:2, 2]
```

```
array([3, 6])
```

1차원 배열

- 콜론(:)만 쓰면 축 전체를 선택한다는 의미
  - 원래 차원의 슬라이스를 얻음

```
arr2d[:, :1]
```

```
array([[1],
       [4],
       [7]])
```

2차원 배열

슬라이스로 선택하기

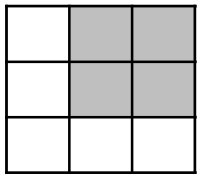
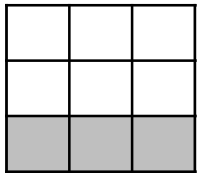
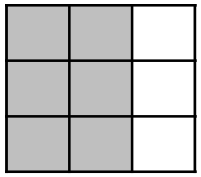
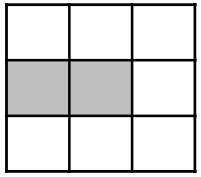
- 슬라이싱 구문에 값 대입
  - 선택 영역 전체에 값이 대입됨

```
arr2d[:2, 1:] = 0
arr2d
```

```
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

색인과 슬라이싱 기초

- 2차원 배열 슬라이싱

	코드	슬라이스의 형태
	arr[:2, 1:]	(2, 2)
	arr[2] arr[2, :] arr[2:, :]	(3,) (3,) (1, 3)
	arr[:, :2]	(3, 2)
	arr[1, :2] arr[1:2, :2]	(2,) (1, 2)

불리언 값으로 선택하기

중복된 이름이 포함된 배열

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
data = np.random.randn(7, 4)
names
```

임의의 표준 정규분포 데이터 생성 (7행 4열)

```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
data
```

```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

각 이름은 data 배열의 각 행에 대응한다고 가정하자

```
names == 'Bob'
```

배열 원소 별로 비교 연산 수행, 불리언 배열 반환

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

불리언 값으로 선택하기

```
data[names == 'Bob']
```

불리언 배열을 배열의 색인으로 사용

```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

```
data[names == 'Bob', 2:]
```

```
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])
```

```
data[names == 'Bob', 3]
```

```
array([1.2464, 0.477 ])
```

※ 주의 : 불리언 배열을 이용한 색인
- 불리언 배열은 색인하려는 축의 길이와 동일한 길이를 가져야 함
  (불리언 배열의 크기가 다르더라도 실패하지 않음으로 주의)
- 반환되는 배열의 내용이 바뀌지 않더라도 항상 데이터 복사가 발생함
- Python의 and, or 연산자는 사용할 수 없음. 대신 &, | 사용

불리언 값으로 선택하기

```
names != 'Bob'
```

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

```
data[~(names == 'Bob')]
```

```
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

```
cond = names == 'Bob'
cond
```

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

```
data[~cond]
```

```
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

불리언 값으로 선택하기

```
mask = (names == 'Bob') | (names == 'Will')
mask
data[mask]
```

```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

```
data[data < 0] = 0
data
```

불리언 배열 색인을 사용하여 값 대입

```
array([[0.0929, 0.2817, 0.769 , 1.2464],
       [1.0072, 0.    , 0.275 , 0.2289],
       [1.3529, 0.8864, 0.    , 0.    ],
       [1.669 , 0.    , 0.    , 0.477 ],
       [3.2489, 0.    , 0.    , 0.1241],
       [0.3026, 0.5238, 0.0009, 1.3438],
       [0.    , 0.    , 0.    , 0.    ]])
```

```
data[names != 'Joe'] = 7
data
```

```
array([[7.    , 7.    , 7.    , 7.    ],
       [1.0072, 0.    , 0.275 , 0.2289],
       [7.    , 7.    , 7.    , 7.    ],
       [7.    , 7.    , 7.    , 7.    ],
       [7.    , 7.    , 7.    , 7.    ],
       [0.3026, 0.5238, 0.0009, 1.3438],
       [0.    , 0.    , 0.    , 0.    ]])
```

팬시 색인

- 팬시 색인
  - 정수 배열을 사용한 색인에 대해 NumPy에서 차용한 단어

```
arr = np.empty((8, 4))
for i in range(8):
    arr[i] = i
arr
```

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

```
arr[[4, 3, 0, 6]]
```

```
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

특정 순서로 row를 선택 → 원하는 순서의 정수가 담긴 ndarray나 리스트를 넘김

팬시 색인

```
arr[[-3, -5, -7]]
```

```
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

```
arr = np.arange(32).reshape((8, 4))
arr
```

```
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, 30, 31]])
```

```
arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
array([ 4, 23, 29, 10])
```

다차원 색인 배열 →
각각의 색인 튜플 ((1, 0), (5, 3), (7, 1), (2, 2))에 대응하는 1차원 배열이 선택됨

팬시 색인

```
arr
```

```
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, 30, 31]])
```

```
arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

행과 열에 대응하는
2차원 배열이 선택됨

주의: 팬시 색인은 슬라이싱과 달리
선택된 데이터를 새로운 배열로 복사한다.

배열 전치와 축 바꾸기

- 배열 전치
  - 데이터를 복사하지 않고 데이터의 모양이 바뀐 뷰를 반환
  - T, transpose(), swapaxes()

```
arr = np.arange(15).reshape((3, 5))
arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
arr.T
```

T 속성을 이용한 벡터 전치
(축을 뒤바꾸는 특별한 경우)

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

배열 전치와 축 바꾸기

```
arr = np.random.randn(6, 3)
arr
```

```
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])
```

```
np.dot(arr.T, arr)
```

벡터 내적

```
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

배열 전치와 축 바꾸기

```
arr = np.arange(16).reshape((2, 2, 4))
arr
```

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

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
arr.transpose((1, 0, 2))
```

축 번호 튜플을 받아 치환

첫번째, 두 번째 축 순서가 뒤바뀜

```
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

배열 전치와 축 바꾸기

```
arr
```

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

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
arr.swapaxes(1, 2)
```

2개의 축 번호를 받아 배열을 뒤 바꿈

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

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

2. 유니버설함수:
배열의 각 원소를 빠르게 처리하는 함수

유니버설 함수: 배열의 각 원소를 빠르게 처리하는 함수

- 유니버설 함수(ufunc)
  - ndarray 안에 있는 데이터 원소 별로 연산을 수행하는 함수
  - 간단한 함수를 고속으로 수행할 수 있게 벡터화시킨 래퍼 함수
- 단항 유니버설 함수

```
arr = np.arange(10)
arr
```

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

```
np.sqrt(arr)
```

```
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458,
       2.8284, 3.    ])
```

```
np.exp(arr)
```

```
array([   1.    ,    2.7183,    7.3891,   20.0855,   54.5982,  148.4132,
        403.4288, 1096.6332, 2980.958 , 8103.0839])
```

유니버설 함수

- 이항 유니버설 함수
  - 2개의 인자를 취해서 단일 배열을 반환하는 함수

```
x = np.random.randn(8)
y = np.random.randn(8)
x
```

```
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,
       -0.6605])
```

```
y
```

```
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,
       -2.3042])
```

```
np.maximum(x, y)                    x, y 원소 별로 가장 큰 값을 계산
```

```
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,
       -0.6605])
```

유니버설 함수

- modf
  - 여러 개의 배열을 반환하는 유니버설 함수
  - 파이선 내장함수인 divmod의 벡터화 버전
  - 분수를 받아서 몫과 나머지를 함께 반환

```
arr = np.random.randn(7) * 5
arr
```

```
array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])
```

나머지 몫

```
remainder, whole_part = np.modf(arr)
remainder
```

```
array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45  ,  0.0077])
```

```
whole_part
```

```
array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

유니버설 함수

```
arr
```

```
array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])
```

```
np.sqrt(arr)
```

```
C:₩Users₩gsjung₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: R
untimeWarning: invalid value encountered in sqrt
  """Entry point for launching an IPython kernel.
```

```
array([   nan,    nan,    nan, 2.318 , 1.9022, 1.8574, 2.2378])
```

?

```
np.sqrt(arr, arr)
```

```
C:₩Users₩gsjung₩Anaconda3₩lib₩site-packages₩ipykernel_launcher.py:1: R
untimeWarning: invalid value encountered in sqrt
  """Entry point for launching an IPython kernel.
```

```
array([   nan,    nan,    nan, 2.318 , 1.9022, 1.8574, 2.2378])
```

```
arr
```

```
array([   nan,    nan,    nan, 2.318 , 1.9022, 1.8574, 2.2378])
```

3. 배열을 이용한 배열 지향 프로그래밍

배열을 이용한 배열 지향 프로그래밍

- 벡터화
  - 배열 연산을 사용해서 반복문을 명시적으로 제거하는 기법
  - 벡터화된 배열에 대한 산술 연산은 순수 파이선 연산에 비해 2~3배에서 많게는 수십, 수백 배까지 빠름

- 사례
  - 값이 놓여 있는 그리드에서 sqrt(x^2 + y^2)을 계산
  - np.meshgrid 함수
    - 2개의 1차원 배열을 받아서 가능한 모든 (x, y) 짝을 만들 수 있는 2차원 배열 2개를 반환

배열을 이용한 배열 지향 프로그래밍

```
points = np.arange(-5, 5, 0.01) # -5에서 4.99까지 0.01씩 증가하는 값들의 배열
                                # 1000 equally spaced points
xs, ys = np.meshgrid(points, points)
ys
```

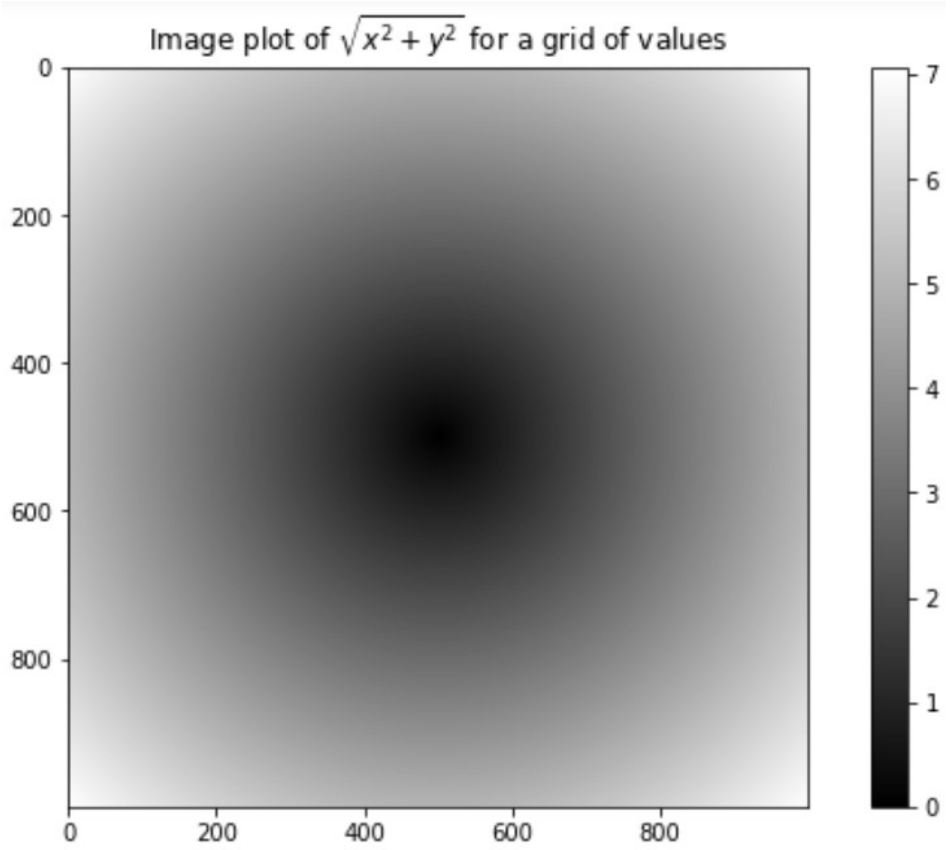
```
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

```
z = np.sqrt(xs ** 2 + ys ** 2)
z
```

```
array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       ...,
       [7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
       [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

배열을 이용한 배열 지향 프로그래밍

```
import matplotlib.pyplot as plt
plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
plt.title("Image plot of $₩sqrt{x^2 + y^2}$ for a grid of values")
```



배열 연산으로 조건절 표현하기

- numpy.where 함수
  - 'x if 조건 else y'와 같은 삼항식의 벡터화된 버전
  - 다른 배열에 기반한 새로운 배열 생성

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

```
result = [(x if c else y)
          for x, y, c in zip(xarr, yarr, cond)]
result
```

단점: 순수 파이선 수행이기에 큰 배열을 빨리 처리 못 함. 다차원 배열에 사용 불가

```
[1.1, 2.2, 1.3, 1.4, 2.5]
```

```
result = np.where(cond, xarr, yarr)
result
```

배열 아니어도 OK
둘 중 하나, 혹은 둘 다 스칼라이어도 동작

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

배열 연산으로 조건절 표현하기

```
arr = np.random.randn(4, 4)
arr
```

```
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])
```

```
arr > 0
```

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

```
np.where(arr > 0, 2, -2)
```

양수는 2로, 음수는 -2로 모두 변경

```
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

배열 연산으로 조건절 표현하기

```
np.where(arr > 0, 2, arr)        arr의 모든 양수를 2로 변경
```

```
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```