

Numpy 기본

August 30, 2020

아래 강의 노트는 [Python for data Analysis](#) 책 4장을 기반으로 번역 및 편집하여 페이지 구성함

무단 배포를 금지 합니다.

1 NumPy 기본: 배열과 벡터 연산

Numpy 에서 제공하는 것

- 효율적인 다차원 배열인 ndarray는 빠른 배열 계산과 유연한 브로드캐스팅 기능 제공
- 반복문을 작성할 필요 없이 전체 데이터 배열을 빠르게 계산 할 수 있는 표준 수학적 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 연결할 수 있는 C API

데이터 분석 에서 중요하게 생각하는 기능

- 벡터 배열 상에서 데이터 가공, 정제, 부분집합, 필터링 변형 그리고 다른 여러 종류의 연산을 빠르게 수행
- 정렬, 유일 원소 찾기, 집합 연산 같은 일방적인 배열 처리 알고리즘
- 통계의 효과적인 표현과 데이터를 수집 요약하기
- 다양한 종류의 데이터를 병합하고 역끼 위한 데이터 정렬과 데이터 간의 관계 조직
- 내부에서 if-elif-else를 사용하는 반복문 대신 사용할 수 있는 조건 표현을 허용하는 배열 처리
- 데이터 묶음 전체에 적용할 수 있는 수집, 변형, 함수 적용 같은 데이터 처리

```
[1]: import numpy as np
my_arr = np.arange(1000000)
my_list = list(range(1000000))
```

Numpy 배열과 파이썬의 리스트의 성능 차이 비교

```
[2]: %time for _ in range(10): my_arr2 = my_arr * 2
%time for _ in range(10): my_list2 = [x * 2 for x in my_list]
```

```
CPU times: user 39.8 ms, sys: 19.5 ms, total: 59.3 ms
Wall time: 24.9 ms
CPU times: user 543 ms, sys: 123 ms, total: 666 ms
Wall time: 658 ms
```

Numpy를 사용한 코드 순수한 파이썬으로 작성한 코드보다 > 속도 : 열배 ~ 백배 빠름
> 메모리 적게사용

1.1 The NumPy ndarray: 다차원 배열

ndarray라고 하는 N 배열 객체
대규모의 집합을 담을 수 있고 빠르고 유연한 자료구조

```
[3]: import numpy as np
      np.random.seed(10)
      # Generate some random data
      data = np.round(np.random.randn(2, 3))*10
      data
```

```
[3]: array([[ 10.,  10., -20.],
            [-0.,  10., -10.]])
```

산술연산

```
[4]: data * 10
```

```
[4]: array([[ 100.,  100., -200.],
            [-0.,  100., -100.]])
```

```
[5]: data + data
```

```
[5]: array([[ 20.,  20., -40.],
            [-0.,  20., -20.]])
```

- ndarray는 모두 같은 자료 구조형
- shape: 튜플과 배열에 저장된 각 차원의 크기
- dtype: 튜플과 배열에 저장된 자료형
- dim: 배열의 차원수

```
[6]: data.shape
```

```
[6]: (2, 3)
```

```
[7]: data.dtype
```

```
[7]: dtype('float64')
```

1.1.1 ndarrays 생성하기

배열을 생성하는 가장 쉬운 방법은 array 함수 이용
예: 파이썬의 리스트는 변환하기 좋은 예

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

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

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

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

```
[10]: arr2.ndim
```

```
[10]: 2
```

```
[11]: arr2.shape
```

```
[11]: (2, 4)
```

```
[12]: arr1.dtype
```

```
[12]: dtype('float64')
```

```
[13]: arr2.dtype
```

```
[13]: dtype('int64')
```

특수한 배열 만들기

* `zeros(): ()` 사이즈 만큼 0으로 채워진 배열 생성 * `ones(): ()` 사이즈 만큼 1으로 채워진 배열 생성

* `empty(): ()` 사이즈 만큼 빈 배열 생성

```
[14]: np.zeros(10)
```

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

```
[15]: np.zeros((3, 6))
```

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

```
[16]: np.empty((2, 3, 2))
```

```
[16]: array([[[1.72723371e-077, 1.72723371e-077],
           [3.95252517e-323, 0.00000000e+000],
           [0.00000000e+000, 0.00000000e+000]],
```

```
[[0.00000000e+000, 0.00000000e+000],
 [0.00000000e+000, 0.00000000e+000],
 [0.00000000e+000, 0.00000000e+000]]])
```

```
[17]: np.arange(15)
```

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

1.1.2 ndarrays의 데이터 타입

dtype은 ndarray가 메모리에 있는 특정 데이터를 해저하기 위해 필요한 정보를 담고 있는 특수한 객체 astype을 이용하여 배열의 dtype을 다른 형식으로 변환 가능

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

```
[18]: dtype('float64')
```

```
[19]: arr2.dtype
```

```
[19]: dtype('int32')
```

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

```
[20]: dtype('int64')
```

정수형을 부동소수점으로 변환

```
[21]: float_arr = arr.astype(np.float64)
      float_arr.dtype
```

```
[21]: dtype('float64')
```

부동소수점수를 정수형 dtype으로 변환하면 소수점 아래 자리는 버려짐

```
[22]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
      arr
      arr.astype(np.int32)
```

```
[22]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

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

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

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

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

```
[25]: empty_uint32 = np.empty(8, dtype='u4')
empty_uint32
```

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

astype을 호출 하면 새로운 dtype과 동일해도 항상 새로운 배열을 생성 (데이터복사)

1.1.3 NumPy 산술연산

벡터화 : 배열의 중요한 특징은 for을 사용하지 않고도 데이터를 일괄 처리 가능

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

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

```
[27]: arr * arr
```

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

```
[28]: arr - arr
```

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

```
[29]: 1 / arr
```

```
[29]: array([[1.         , 0.5         , 0.33333333],
          [0.25        , 0.2         , 0.16666667]])
```

```
[30]: arr ** 0.5
```

```
[30]: array([[1.         , 1.41421356, 1.73205081],
          [2.         , 2.23606798, 2.44948974]])
```

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

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

1.1.4 색인과 슬라이싱

인덱스는 파이썬의 리스트 인덱싱과 유사하게 동작

```
[32]: arr = np.arange(10)
      print(arr)
      print(arr[5])
      print(arr[5:8])
      arr[5:8] = 12
      print(arr)
```

```
[0  1  2  3  4  5  6  7  8  9]
5
[5  6  7]
[ 0  1  2  3  4 12 12 12  8  9]
```

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

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

```
[34]: arr
```

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

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

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

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

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

		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

Figure 4-1. Indexing elements in a NumPy array

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

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

```
[38]: arr2d[0][2]
```

```
[38]: 3
```

```
[39]: arr2d[0, 2]
```

```
[39]: 3
```

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

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

```
[41]: arr3d[0]
```

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

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

```
arr3d
arr3d[0] = old_values
arr3d
```

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

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

```
[43]: arr3d[1, 0]
```

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

```
[44]: x = arr3d[1]
x
x[0]
```

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

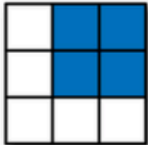

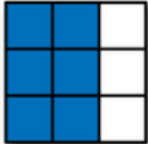

	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code>	<code>(3,)</code>
	<code>arr[2, :]</code>	<code>(3,)</code>
	<code>arr[2:, :]</code>	<code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code>	<code>(2,)</code>
	<code>arr[1:2, :2]</code>	<code>(1, 2)</code>

Figure 4-2. Two-dimensional array slicing

슬라이스로 선택하기


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

```
[45]: array([ 1,  2,  3,  4, 64])
```

```
[46]: arr2d
      arr2d[:2]
```

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

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

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

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

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

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

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

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

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

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

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

1.1.5 불린 인덱스 (Boolean Indexing)

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

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

```
[53]: data
```

```
[53]: array([[ 0.26551159,  0.10854853,  0.00429143, -0.17460021],
           [ 0.43302619,  1.20303737, -0.96506567,  1.02827408],
           [ 0.22863013,  0.44513761, -1.13660221,  0.13513688],
           [ 1.484537   , -1.07980489, -1.97772828, -1.7433723  ],
           [ 0.26607016,  2.38496733,  1.12369125,  1.67262221],
           [ 0.09914922,  1.39799638, -0.27124799,  0.61320418],
           [-0.26731719, -0.54930901,  0.1327083  , -0.47614201]])
```

```
[54]: names == 'Bob'
```

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

```
[55]: data
```

```
[55]: array([[ 0.26551159,  0.10854853,  0.00429143, -0.17460021],
           [ 0.43302619,  1.20303737, -0.96506567,  1.02827408],
           [ 0.22863013,  0.44513761, -1.13660221,  0.13513688],
           [ 1.484537   , -1.07980489, -1.97772828, -1.7433723  ],
           [ 0.26607016,  2.38496733,  1.12369125,  1.67262221],
           [ 0.09914922,  1.39799638, -0.27124799,  0.61320418],
           [-0.26731719, -0.54930901,  0.1327083  , -0.47614201]])
```

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

```
[56]: array([[ 0.26551159,  0.10854853,  0.00429143, -0.17460021],
           [ 1.484537   , -1.07980489, -1.97772828, -1.7433723  ]])
```

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

```
[57]: array([[ 0.00429143, -0.17460021],
           [-1.97772828, -1.7433723  ]])
```

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

```
[58]: array([-0.17460021, -1.7433723  ])
```

```
[59]: names != 'Bob'
      data[~(names == 'Bob')]
```

```
[59]: array([[ 0.43302619,  1.20303737, -0.96506567,  1.02827408],
           [ 0.22863013,  0.44513761, -1.13660221,  0.13513688],
           [ 0.26607016,  2.38496733,  1.12369125,  1.67262221],
           [ 0.09914922,  1.39799638, -0.27124799,  0.61320418],
           [-0.26731719, -0.54930901,  0.1327083  , -0.47614201]])
```

```
[60]: cond = names == 'Bob'
      data[~cond]
```

```
[60]: array([[ 0.43302619,  1.20303737, -0.96506567,  1.02827408],
             [ 0.22863013,  0.44513761, -1.13660221,  0.13513688],
             [ 0.26607016,  2.38496733,  1.12369125,  1.67262221],
             [ 0.09914922,  1.39799638, -0.27124799,  0.61320418],
             [-0.26731719, -0.54930901,  0.1327083 , -0.47614201]])
```

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

```
[61]: array([[ 0.26551159,  0.10854853,  0.00429143, -0.17460021],
             [ 0.22863013,  0.44513761, -1.13660221,  0.13513688],
             [ 1.484537 , -1.07980489, -1.97772828, -1.7433723 ],
             [ 0.26607016,  2.38496733,  1.12369125,  1.67262221]])
```

```
[62]: data[data < 0] = 0
      data
```

```
[62]: array([[0.26551159, 0.10854853, 0.00429143, 0.          ],
             [0.43302619, 1.20303737, 0.          , 1.02827408],
             [0.22863013, 0.44513761, 0.          , 0.13513688],
             [1.484537 , 0.          , 0.          , 0.          ],
             [0.26607016, 2.38496733, 1.12369125, 1.67262221],
             [0.09914922, 1.39799638, 0.          , 0.61320418],
             [0.          , 0.          , 0.1327083 , 0.          ]])
```

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

```
[63]: array([[7.          , 7.          , 7.          , 7.          ],
             [0.43302619, 1.20303737, 0.          , 1.02827408],
             [7.          , 7.          , 7.          , 7.          ],
             [7.          , 7.          , 7.          , 7.          ],
             [7.          , 7.          , 7.          , 7.          ],
             [0.09914922, 1.39799638, 0.          , 0.61320418],
             [0.          , 0.          , 0.1327083 , 0.          ]])
```

1.1.6 Fancy Indexing

빈 배열 생성후 특정 행을 선택하여 정수를 담기

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

```
[64]: 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.]])
```

2차원 배열 생성- 원하는 순서가 명시된 정수가 담긴 ndarray나 리스트 넘기기

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

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

마이너스 인덱스의 경우 역순으로 선택
 ex: a = [1,2,3,4,5]
 a[-1] == 5

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

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

```
[67]: arr = np.arange(32).reshape((8, 4))
arr
```

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

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

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

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

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

1.1.7 배열 전치와 축 바꾸기

배열 전체는 데이터를 복사하지 않고 데이터의 모양이 바뀐 뷰 반환
ndarray는 transpose 메서드와 T라는 이름의 특수한 속성을 갖고 있음

```
[70]: arr = np.arange(15).reshape((3, 5))
      arr
```

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

```
[71]: arr.transpose((1,0))
```

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

행렬의 내적은 np.dot를 이용하여 구할 수 있음

```
[72]: arr = np.random.randn(6, 3)
      arr
      np.dot(arr.T, arr)
```

```
[72]: array([[ 2.63820285, -0.10904676,  0.02035354],
            [-0.10904676,  2.48535187, -0.17196313],
            [ 0.02035354, -0.17196313,  2.02669571]])
```

```
[73]: arr = np.arange(16).reshape((2, 2, 4))
      arr
```

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

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

transpose 메서드는 튜플로 축 번호를 받아서 치환

이 예제에서는 첫번째와 두번째 축의 순서가 바뀌고 마지막 축은 그대로 남아 있음

```
[74]: arr.transpose((1, 0, 2))
```

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

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

```
[75]: arr.shape
```

```
[75]: (2, 2, 4)
```

```
[76]: arr
      arr.swapaxes(1, 2)
```

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

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

1.2 유니버설 함수: 배열의 각 원소를 빠르게 처리하는 함수

```
[77]: arr = np.arange(10)
      arr
      np.sqrt(arr)
      np.exp(arr)
```

```
[77]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
            5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
            2.98095799e+03, 8.10308393e+03])
```

```
[78]: x = np.random.randn(8)
      y = np.random.randn(8)
      x
      y
      np.maximum(x, y)
```

```
[78]: array([-0.23218226,  0.70816002,  1.12878515,  0.2035808 ,  2.39470366,
            0.91745894,  1.04618286, -0.36218045])
```

```
[79]: arr = np.random.randn(7) * 5
      arr
      remainder, whole_part = np.modf(arr)
      remainder
      whole_part
```

```
[79]: array([-1., -0.,  1.,  2., -1.,  4.,  1.])
```

```
[80]: arr
      np.sqrt(arr)
      np.sqrt(arr, arr)
      arr
```

/Users/Jaehee/Library/Python/3.7/lib/python/site-packages/ipykernel_launcher.py:2: RuntimeWarning: invalid value encountered in sqrt

/Users/Jaehee/Library/Python/3.7/lib/python/site-packages/ipykernel_launcher.py:3: RuntimeWarning: invalid value encountered in sqrt

This is separate from the ipykernel package so we can avoid doing imports until

```
[80]: array([          nan,          nan,  1.26363844,  1.51806275,          nan,
            2.22381705,  1.25449946])
```

1.3 배열을 이용한 배열 지향 프로그래밍

np.meshgrid함수는 두개의 1차원 배열을 받아서 가능한 모든(x,y) 짝을 만들 수 있음
2차원 배열 두개 반환

```
[81]: points = np.arange(-5, 5, 1) # 1000 equally spaced points
      xs, ys = np.meshgrid(points, points)
      ys
```

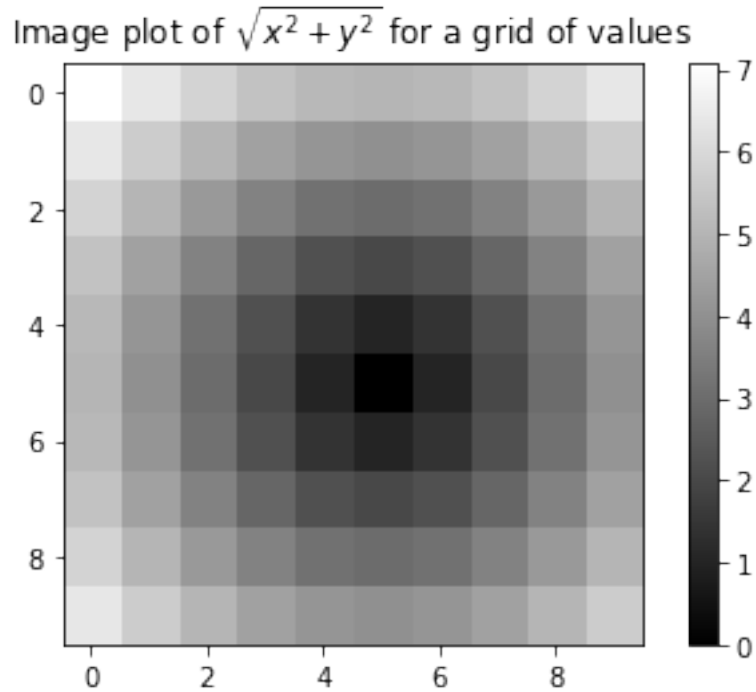
```
[81]: array([[ -5,  -5,  -5,  -5,  -5,  -5,  -5,  -5,  -5,  -5],
            [ -4,  -4,  -4,  -4,  -4,  -4,  -4,  -4,  -4,  -4],
            [ -3,  -3,  -3,  -3,  -3,  -3,  -3,  -3,  -3,  -3],
            [ -2,  -2,  -2,  -2,  -2,  -2,  -2,  -2,  -2,  -2],
            [ -1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,  -1,  -1],
            [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
            [  1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
            [  2,   2,   2,   2,   2,   2,   2,   2,   2,   2],
            [  3,   3,   3,   3,   3,   3,   3,   3,   3,   3],
            [  4,   4,   4,   4,   4,   4,   4,   4,   4,   4]])
```

```
[82]: z = np.sqrt(xs ** 2 + ys ** 2)
      z
```

```
[82]: array([[7.07106781, 6.40312424, 5.83095189, 5.38516481, 5.09901951,
           5.          , 5.09901951, 5.38516481, 5.83095189, 6.40312424],
          [6.40312424, 5.65685425, 5.          , 4.47213595, 4.12310563,
           4.          , 4.12310563, 4.47213595, 5.          , 5.65685425],
          [5.83095189, 5.          , 4.24264069, 3.60555128, 3.16227766,
           3.          , 3.16227766, 3.60555128, 4.24264069, 5.          ],
          [5.38516481, 4.47213595, 3.60555128, 2.82842712, 2.23606798,
           2.          , 2.23606798, 2.82842712, 3.60555128, 4.47213595],
          [5.09901951, 4.12310563, 3.16227766, 2.23606798, 1.41421356,
           1.          , 1.41421356, 2.23606798, 3.16227766, 4.12310563],
          [5.          , 4.          , 3.          , 2.          , 1.          ,
           0.          , 1.          , 2.          , 3.          , 4.          ],
          [5.09901951, 4.12310563, 3.16227766, 2.23606798, 1.41421356,
           1.          , 1.41421356, 2.23606798, 3.16227766, 4.12310563],
          [5.38516481, 4.47213595, 3.60555128, 2.82842712, 2.23606798,
           2.          , 2.23606798, 2.82842712, 3.60555128, 4.47213595],
          [5.83095189, 5.          , 4.24264069, 3.60555128, 3.16227766,
           3.          , 3.16227766, 3.60555128, 4.24264069, 5.          ],
          [6.40312424, 5.65685425, 5.          , 4.47213595, 4.12310563,
           4.          , 4.12310563, 4.47213595, 5.          , 5.65685425]])
```

```
[83]: 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")
```

```
[83]: Text(0.5, 1.0, 'Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values')
```

```
[84]: plt.draw()
```

<Figure size 432x288 with 0 Axes>

```
[85]: plt.close('all')
```

1.3.1 배열 연산으로 조건절 표현하기

np.where 함수는 x if 조건 else y 같은 삼항식의 벡터화된 버전

```
[86]: 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])
```

cond 값이 >True 일때는 xarr의 값을 취하고

>False 일때는 yarr의 값을 취하기

==> 문제점 다차원에서 사용불가, 큰 배열 사용 불가

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

```
[87]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

```
[88]: result = np.where(cond, xarr, yarr)
      result
```

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

```
[89]: arr = np.random.randn(4, 4)
      arr
      arr > 0
      np.where(arr > 0, 2, -2)
```

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

np.where를 사용하여 스칼라 값과 배열을 조합

아래 예는 arr 모든 양수를 2로 변경

```
[90]: np.where(arr > 0, 2, arr) # 양수인 경우에만 2를 대입
```

```
[90]: array([[ 2.          , -1.50832149,  2.          , -1.04513254],
             [-0.79800882,  2.          ,  2.          , -1.85618548],
             [-0.2227737 , -0.06584785, -2.13171211, -0.04883051],
             [ 2.          ,  2.          , -1.99439377,  2.          ]])
```

1.3.2 수학 메서드와 통계 메서드

```
[91]: arr = np.random.randn(5, 4)
      arr
      arr.mean()
      np.mean(arr)
      arr.sum()
```

```
[91]: -2.48396903263556
```

mean과 sum같은 함수는 선택적으로 axis 인자를 받아서 해당 axis에 대한 통계를 계산 하고 한차수 낮은 배열을 반환

```
[92]: arr.mean(axis=1)
      arr.sum(axis=0)
```

```
[92]: array([ 0.52626105,  0.13593418, -3.75916951,  0.61300525])
```

```
[93]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
      arr.cumsum()
```

```
[93]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

```
[94]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
      arr
      arr.cumsum(axis=0)
      arr.cumprod(axis=1)
```

```
[94]: array([[ 0,  0,  0],
             [ 3, 12, 60],
             [ 6, 42, 336]])
```

1.3.3 배열의 불리언을 위한 메서드

메서드의 불리언 값을 1 또는 0으로 강제 할 수 있어서 `sum` 메서드를 실행하는 불리언 배열의 True 갯수 셀 수 있음

```
[95]: arr = np.random.randn(100)
      (arr > 0).sum() # Number of positive values
```

```
[95]: 52
```

any 하나 이상의 값이 True 인지 확인 all 모든 원소값이 True 인지 확인

```
[96]: bools = np.array([False, False, True, False])
      bools.any()
      bools.all()
```

```
[96]: False
```

1.3.4 정렬

리스트의 `sort` 처럼 Numpy 배열 역시 `sort` 이용해 정렬가능

```
[97]: arr = np.random.randn(6)
      arr
      arr.sort()
      arr
```

```
[97]: array([-1.54730539, -0.67202344, -0.37276142,  0.10581208,  0.40246909,
             1.34217928])
```

```
[98]: arr = np.random.randn(5, 3)
      arr
      arr.sort(1)
      arr
```

```
[98]: array([[ 0.50318918,  1.1943506 ,  1.34480651],
             [-0.56350567,  0.68790473,  0.85482876],
             [-1.53306872,  0.1489607 ,  0.31511117],
             [-0.57491766, -0.36017113,  0.4361853 ]],
            dtype=float64)
```

```
[-1.00471635, -0.03791788, 0.59145309]])
```

```
[99]: large_arr = np.random.randn(1000)
      large_arr.sort()
      large_arr[int(0.05 * len(large_arr))] # 5% quantile
```

```
[99]: -1.5250342901043503
```

1.3.5 집합관련함수

np.unique 함수는 중복된 원소를 제거하고 남은 원소를 정렬된 형태로 반환

```
[100]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
      np.unique(names)
```

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

```
[101]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
      np.unique(ints)
```

```
[101]: array([1, 2, 3, 4])
```

파이썬의 set을 이용해서도 가능

```
[102]: sorted(set(names))
```

```
[102]: ['Bob', 'Joe', 'Will']
```

np.in1d 함수는 두개의 배열을 인자로 받아서 첫번째 배열의 원소가 두번째 배열의 원소를 포함하는지를 나타내는 불리언 배열로 반환

```
[103]: values = np.array([6, 0, 0, 3, 2, 5, 6])
      np.in1d(values, [2, 3, 6])
```

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

1.4 배열의 파일 입출력

numpy는 디스크나 텍스트나 바이너리 형식의 데이터를 불러오거나 저장 가능 포맷의 데이터는 pandas에서 처리하는 것을 선호하기 때문에 다음장에

np.save 와 np.load는 배열 데이터를 효과적으로 디스크에 저장하고 불러오기 위한 함수
확장자 npy로 끝나지 않으면 자동적으로 확장자 추가

```
[104]: arr = np.arange(10)
      np.save('some_array', arr)
```

```
[105]: np.load('some_array.npy')
```

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

np.savez함수를 이용하면 여러개의 배열을 압축된 형식으로 저장

```
[106]: np.savez('array_archive.npz', a=arr, b=arr)
```

```
[107]: arch = np.load('array_archive.npz')
arch['b']
```

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

압축이 잘되는 데이터라면 np.savez_compressed사용

```
[108]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

```
[109]: !rm some_array.npy
!rm array_archive.npz
!rm arrays_compressed.npz
```

1.5 선형대수

dot을 이용하여 행렬곱

x.dot(y)

코드는

np.dot(x,y)

와 같다

```
[110]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
x
y
x.dot(y)
```

```
[110]: array([[ 28.,  64.],
           [ 67., 181.]])
```

```
[111]: np.dot(x, y)
```

```
[111]: array([[ 28.,  64.],
           [ 67., 181.]])
```

```
[112]: np.dot(x, np.ones(3))
```

```
[112]: array([ 6., 15.])
```

Python3.5 이상부터 @ 기호는 행렬 곱셈

```
[113]: x @ np.ones(3)
```

```
[113]: array([ 6., 15.])
```

numpy.linalg는 행렬의 분할과 역행렬, 행렬식과 같은 것들을 포함

```
[114]: from numpy.linalg import inv, qr
X = np.random.randn(5, 5)
mat = X.T.dot(X)
inv(mat)
mat.dot(inv(mat))
q, r = qr(mat)
r
```

```
[114]: array([[ -6.34192354,  8.55475904, -1.96748645,  5.34369789,  0.40441882],
 [ 0.          , -3.3457706 ,  0.76648945, -7.54499654,  1.50486273],
 [ 0.          ,  0.          , -6.73722844, -0.21446272,  2.3005795 ],
 [ 0.          ,  0.          ,  0.          , -2.72492822, -0.27903852],
 [ 0.          ,  0.          ,  0.          ,  0.          ,  0.13249429]])
```

1.6 난수생성

numpy.random 모듈은 파이썬 내장 random 함수를 보강 하여 다양한 종류의 확률분포로부터 표본값을 생성 하는데 주로 사용

아래 코드는 표준정규분포로부터 4 x 4 크기의 표본 생성

```
[115]: samples = np.random.normal(size=(4, 4))
samples
```

```
[115]: array([[ 0.46609064,  0.34537575,  0.4089362 ,  0.3414533 ],
 [ 1.96051665,  1.04847626,  0.93608826,  0.70327107],
 [ 0.79620261, -1.42670884,  0.5128028 , -0.8404604 ],
 [ 1.82761021,  0.23279635,  0.2991754 ,  1.62109772]])
```

```
[116]: from random import normalvariate
N = 1000000
%timeit samples = [normalvariate(0, 1) for _ in range(N)]
%timeit np.random.normal(size=N)
```

774 ms ± 6.59 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

25.8 ms ± 377 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

난수 생성기의 시드값에 따라 정해진 난수를 알고리즘으로 생성

```
[117]: np.random.seed(1234)
```

Table 4-8. Partial list of `numpy.random` functions

Function	Description
<code>seed</code>	Seed the random number generator
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in-place
<code>rand</code>	Draw samples from a uniform distribution
<code>randint</code>	Draw random integers from a given low-to-high range
<code>randn</code>	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
<code>binomial</code>	Draw samples from a binomial distribution
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1) distribution

```
[118]: rng = np.random.RandomState(1234)
rng.randn(10)
```

```
[118]: array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,
              0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```

1.7 Example: 계단 오르기

```
[119]: import random
position = 0
walk = [position]
steps = 1000
for i in range(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
```

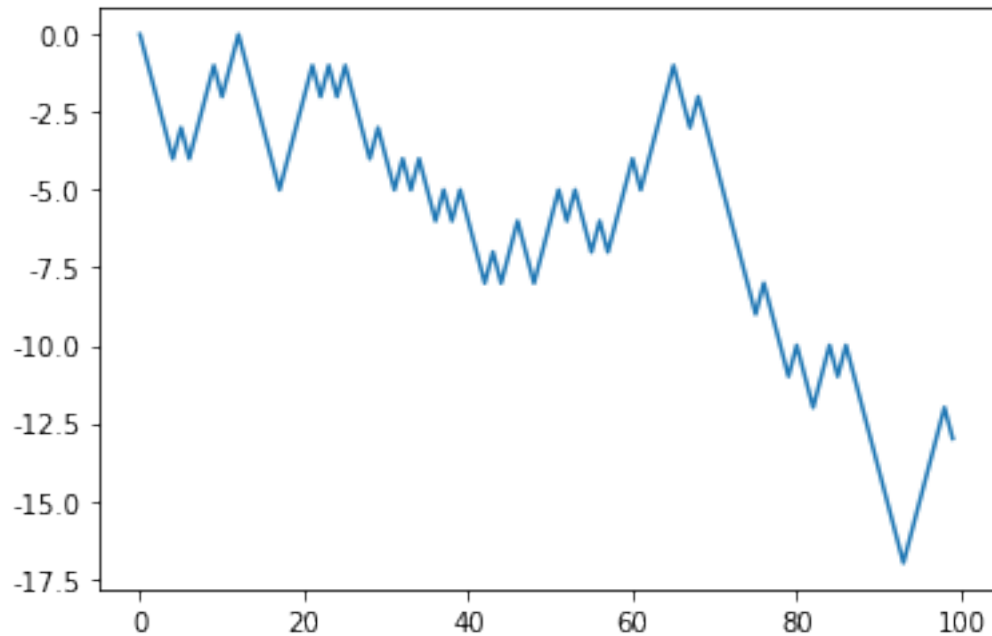
```
[120]: plt.figure()
```

```
[120]: <Figure size 432x288 with 0 Axes>
```

```
<Figure size 432x288 with 0 Axes>
```

```
[121]: plt.plot(walk[:100])
```

```
[121]: [<matplotlib.lines.Line2D at 0x12230ca20>]
```



```
[122]: np.random.seed(12345)
```

```
[123]: nsteps = 1000
draws = np.random.randint(0, 2, size=nsteps)
steps = np.where(draws > 0, 1, -1)
walk = steps.cumsum()
```

```
[124]: walk.min()
walk.max()
```

```
[124]: 31
```

```
[125]: (np.abs(walk) >= 10).argmax()
```

```
[125]: 37
```

1.7.1 Simulating Many Random Walks at Once

```
[126]: nwalks = 5000
nsteps = 1000
draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1
steps = np.where(draws > 0, 1, -1)
walks = steps.cumsum(1)
walks
```



```
[126]: array([[ 1,  0,  1, ...,  8,  7,  8],
              [ 1,  0, -1, ..., 34, 33, 32],
              [ 1,  0, -1, ...,  4,  5,  4],
              ...,
              [ 1,  2,  1, ..., 24, 25, 26],
              [ 1,  2,  3, ..., 14, 13, 14],
              [-1, -2, -3, ..., -24, -23, -22]])
```

```
[127]: walks.max()
       walks.min()
```

```
[127]: -133
```

```
[128]: hits30 = (np.abs(walks) >= 30).any(1)
       hits30
       hits30.sum() # Number that hit 30 or -30
```

```
[128]: 3410
```

```
[129]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
       crossing_times.mean()
```

```
[129]: 498.8897360703812
```

```
[130]: steps = np.random.normal(loc=0, scale=0.25,
                                size=(nwalks, nsteps))
```

1.8 Conclusion