4. Numpy 기본 : 배열과 벡터 연산

Numpy는 Numerical Python의 줄임말로, 파이썬에서 산술 계산을 위한 가장 중요한 필수 패키지 중 하나이다. 과학 계산을 위한 대부분의 패키지는 Numpy의 배열 객체를 데이터 교환을 위한 공통 언어처럼 사용한다.

NumPy에서 제공하는 것들은 다음과 같다.

- 효율적인 다차원 배열인 ndarray는 빠른 배열 계산과 유연한 **브로드캐스팅** 기능을 제공한다.
- 반복분을 작성할 필요없이 전체 데이터 배열을 빠르게 계산할 수 있는 표준 수학 함수
- 배열 데이터를 디스크에 쓰거나 읽을 수 있는 도구와 메모리에 적재된 파일을 다루는 도구
- 선형 대수, 난수 생성기, 푸리에 변환 기능
- C,C++, 포트란으로 작성한 코드를 연결할 수 있는 C API

NumPy의 C API는 사용하기 쉬우므로 저수준 언어로 작성된 외부 라이브러리에 데이터를 전달하거나 반대로 외부 라이브러리에서 NumPy 배열 형태로 파이썬에 데이터를 전달하기 용이하다. 이 기능은 파이썬으로 레거시 C, C++, 포트란 코드를 감싸서 동적이며 쉽게 사용할 수 있는 인터페이스를 만들 수 있도록 해준다.

NumPy 자체는 모델링이나 과학 계산을 위한 기능은 제공하지 않으므로 먼저 NumPy 배열과 배열 기반 연산에 대한 이해를 한 다음 pandas 같은 배열 기반 도구를 사용하면 훨씬 더 효율적이다. NumPy 만으로도 방대한 주제이므로 브로드캐스팅 같은 NumPy의 고급 기능은 부록 A에서 따로 다루도록 하겠다.

대부분의 데이터 분석 어플에서 중요하게 생각하는 기능은 다음과 같다.

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

NumPy는 일반적인 산술 데이터 처리를 위한 기반 라이브러리를 제공하기 때문에 많은 독자가 통계나 분석, 특히 표 형식의 데이터를 처리하기 위해 pandas를 사용하기 원할 것이다. 또한 pandas는 NumPy에는 없는 시계열 처리 같은 다양한 도메인 특화 기능을 제공한다.

NOTE 파이썬에서 배열 기반 연산을 시도했던 기록은 짐 헝구닌이 Numeric 라이브러리를 작성했던 1995년 까지 거슬러 올라간다. 이후로 10년이 지나고 많은 과학 계산 커뮤니티는 배열 프로그래밍에 파이썬을 사용하기 시작했으나 라이브러리 생태계는 2000년대 초에 갈라지게 된다. 2005년 트라비스 올리펀트가 당시의 Numeric 라이브러리와 Numarray 프로젝트로부터 NumPy 프로젝트를 시작하여 커뮤니티에 단일 배열 컴퓨팅 프레임워크를 소개했다.

NumPy가 파이썬 산술 계산 영역에서 중요한 위치를 차지하는 이유 중 하나는 대용량 데이터 배열을 효율적으로 다룰 수 있도록 설계되었다는 점이다. 이에 대해 좀 더 알아보자.

- **NumPy는 내주적으로 데이터를 다른 내장 파이썬 객체와 구분된 연속된 메모리 블록에 저장한다. NumPy의 각종 알고리즘은 모두 C로 작성되어 타입 검사나 다른 오버헤드 없이 메모리를 직접 조작할 수 있다. NumPy 배열은 또한 내장 파이썬의 연속된 자료형들보다 훨씬 더 적은 메모리를 사용한다.**
- **NumPy 연산은 파이썬 반복문을 사용하지 않고 전체 배열에 대한 복잡한 계산을 수행할 수 있다.**

성능 차이를 확인 하기 위해 백만 개의 정수를 저장하는 NumPy 배열과 파이썬 리스트를 비교해 보자.

```
In [1]: import numpy as np

In [2]: my_arr = np.arange(1000000)

In [3]: my_list = list(range(1000000))

In [4]: my_arr
Out[4]: array([     0,      1,      2, ..., 999997, 999998, 999999])

In [5]: my_list
Out[5]:
[0,
 1,
 2,
 3,
 4,
 ...,
 999,
 ...]
```

이제 각각의 배열과 리스트 원소에 2를 곱해보자.

```
In [7]: %time for _ in range(10) : my_arr2 = my_arr *2
Wall time: 16.8 ms

In [8]: %time for _ in range(10) : my_list2 = [x*2 for x in my_list]
Wall time: 774 ms
```

NumPy를 사용한 코드가 순수 파이썬으로 작성한 코드보다 열 배에서 백 배 이상 빠르고 메모리도 더 적게 사용하는 것을 확인할 수 있다.

4.1 NumPy ndarray : 다차원 배열 객체

NumPy의 핵심 기능 중 하나는 ndarray라고 하는 N차원의 배열 객체인데 파이썬에서 사용 할 수 있는 대규모 데이터 집합을 담을 수 있는 빠르고 유연한 자료구조다. 배열은 스칼라 원소 간의 연산에 사용하는 문법과 비슷한 방식을 사용해서 전체 데이터 블록에 수학적인 연산을 수행할 수 있게 해준다.

파이썬 내장 객체의 스칼라값을 다루는 것과 유사한 방법으로 배치 계산을 처리하는 방법을 알아보기 위해 우선 NumPy 패키지를 임포트하고 임의의 값이 들어 있는 작은 배열을 만들어 보겠다.

```
In [10]: import numpy as np

In [11]: data = np.random.randn(2,3)

In [12]: data
Out[12]:
array([[-0.80355358, -0.28801767,  0.67386331],
       [ 0.96271257,  0.0016632 , -0.06666495]])
```

```
In [13]: data * 10
Out[13]:
array([[-8.03553578, -2.8801767 ,  6.73863315],
       [ 9.62712567,  0.01663201, -0.66664951]])

In [14]: data + data
Out[14]:
array([[-1.60710716, -0.57603534,  1.34772663],
       [ 1.92542513,  0.0033264 , -0.1333299 ]])
```

첫 번째 예제는 모든 원소의 값에 10을 곱했다. 두 번째 예제는 data 배열에서 같은 위치의 값끼리 서로 더했다.

NOTE 이 장과 책 전체에서 NumPy를 임포트할 경우 import numpy as np 컨벤션을 사용한다. 그냥 from numpy import * 라고 해서 np를 입력하지 않아도 되지만 이런 습관은 지양해라. numpy 네임스페이스는 방대하고 파이썬 내장 함수와 같은 이름을 사용하는 경우 (min과 max처럼)도 있기 때문이다.

ndarray는 같은 종류의 데이터를 담을 수 있는 포괄적인 다차원 배열이다. ndarray의 모든 원소는 같은 자료형이어야 한다. 모든 배열은 각 차원의 크기를 알려주는 shape라는 튜플과 배열에 저장된 **자료형**을 알려주는 dtype이라는 객체를 가지고 있다.

```
In [15]: data.shape
Out[15]: (2, 3)

In [16]: data.dtype
Out[16]: dtype('float64')
```

NOTE 배열, NumPy 배열, ndarray는 아주 극소수의 예외를 제외하면 모두 ndarray 객체를 이르는 말이다.

4.1.1 ndarray 생성하기

배열을 생성하는 가장 쉬운 방법은 array 함수를 이용하는 것이다. 순차적인 객체(다른 배열도 포함하여)를 넘겨받고, 넘겨받은 데이터가 들어 있는 새로운 NumPy 배열을 생성한다. 예를 들어 파이썬의 리스트는 변환하기 좋은 예다.

```
In [17]: data1 = [6,7,8,0,1]

In [18]: arr1 = np.array(data1)

In [19]: arr1
Out[19]: array([6, 7, 8, 0, 1])
```

같은 길이를 가지는 리스트를 내포하고 있는 순차 데이터는 다차원 배열로 변환 가능하다.

```
In [20]: data2 = [[1,2,3,4],[5,6,7,8]]

In [21]: arr2 = np.array(data2)

In [22]: arr2
Out[22]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

data2는 리스트를 담고 있는 리스트이므로 NumPy 배열인 arr2는 해당 데이터로부터 형태를 추론하여 2차원 형태로 생성된다. ndim과 shape 속성을 검사해서 이를 확인할 수 있다.

```
In [23]: arr2.ndim
Out[23]: 2

In [24]: arr2.shape
Out[24]: (2, 4)
```

명시적으로 지정하지 않는 한 np.array는 생성될 때 적절한 자료형을 추론한다. 그렇게 추론된 자료형은 dtype 객체에 저장되는데 앞서 살펴본 예제에서 확인해보면 다음과 같다.

```
In [25]: arr1.dtype
Out[25]: dtype('int32')

In [26]: arr2.dtype
Out[26]: dtype('int32')
```

또한 np.array는 새로운 배열을 생성하기 위해 여러 함수를 가지고 있는데, 예를 들어 zeros와 ones는 주어진 길이나 모양에 따라 각각 0과 1이 들어있는 배열을 생성한다. empty 함수는 초기화되지 않은 배열을 생성한다. 이런 메서드를 사용해서 다차원 배열을 생성하려면 원하는 형태의 튜플을넘기면 된다.

```
In [27]: np.zeros(10)
Out[27]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [28]: np.zeros((3,6))
Out[28]:
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])

In [29]: np.zeros((2,3,2))
Out[29]:
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

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

CAUTION_ np.empty는 0으로 초기화된 배열을 반환하지 않는다. 앞서 살펴본 바와 같이 대부분의 경우 np.empty는 초기화되지 않은 '가비지'값으로 채워진 배열을 반환한다.

```
In [30]: np.arange(15)
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

다음은 표준 배열 생성 함수의 목록이다. NumPy는 산술 연산에 초점이 맞춰져 있기에 만약 자료형을 명시하지 않으면 float64(부동소수점)가 될 것이다.

배열생성함수

함수	설명
array	입력 데이터(리스트, 튜플, 배열 또는 다른 순차형 데이터)를 ndarray로 변환하며 dtype을 명시하지 않은 경우 자료형을 추론하여 저장한다. 기본적으로 입력 데이터는 복사된다.
asarray	입력 데이터를 ndarray로 변환하지만 입력 데이터가 이미 ndarray일 경우 복사가 일어나지 않는다.
arange	내장 range 함수와 유사하지만 리스트대신 ndarray를 반환한다.
ones, ones_like	주어진 dtype과 모양을 가지는 배열을 생성하고 내용을 모두 1로 초기화한다. ones_like는 주어진 배열과 동일한 모양과 dtype을 가지는 배열을 새로 생성하여 내용을 모두 1로 초기화한다.
zeros, zeros_like	ones, ones_like와 동일하지만 내용을 모두 0으로 채운다.
empty, empty_like	메모리를 할당하여 새로운 배열을 생성하지만 ones나 zeros 처럼 값을 초기화하지 않는다.
full, full_like	인자로 받은 dtype과 배열의 모양을 가지는 배열을 생성하고 인자로 받은 값으로 배열을 채운다.
eye, identity	N x N 크기의 단위행렬을 생성한다. (최상단에서 우하단을 잇는 대각선은 1로 태워지고 나머지는 0으로 채워진다.)

4.1.2 ndarray의 dtype

dtype은 ndarray가 메모리에 있는 특정 데이터를 해석하기 위해 필요한 정보(또는 **메타데이터**)를 담고 있는 특수한 객체이다.

```
In [31]: arr1 = np.array([1,2,3], dtype=np.float64)

In [32]:  arr2 = np.array([1,2,3], dtype=np.int32)

In [33]: arr1.dtype
Out[33]: dtype('float64')

In [34]: arr2.dtype
Out[34]: dtype('int32')
```

dtype이 있기에 NumPy가 강력하면서도 유연한 도구가 될 수 있었는데, 대부분의 경우 데이터는 디스크에서 데이터를 읽고 쓰기 편하도록 하위 레벨의 표현에 직접적으로 맞춰져 있어서 C나 포트란 같은 저수준 언어로 작성된 코드와 쉽게 연동이 가능하다. 산술 데이터의 dtype은 float나 int 같은 자료형의 이름과 하나의 원소가 차지하는 비트 수로 이뤄진다. 파이썬의 float 객체에서 사용되는 표준 배정밀도 부동소

수점 값은 8바이트 혹은 64비트로 이뤄지는데 이 자료형은 NumPy에서 float64로 표현된다. 다음 표는 NumPy가 지원하는 모든 자료형의 목록이다.

NOTE NumPy의 모든 dtpye을 외울 필요는 없다. 주로 사용하게 될 자료형의 일반적인 종류(부동소수점, 복소수, 정수, 불리언, 문자열, 일반 파이썬 객체)만 신경 쓰긴 된다. 주로 대용량 데이터가 메모리나 디스크에 저장되는 방식을 제어해야 할 필요가 있을 때 알아두면 좋다.

ndarray의 astype 메서드를 사용해서 배열의 dtype을 다른 형으로 명시적으로 변환 (또는 **캐스팅**) 가능하다.

```
In [35]: arr = np.array([1,2,3,4,5])

In [36]: arr.dtype
Out[36]: dtype('int32')

In [37]: float_arr = arr.astype(np.float64)

In [38]: float_arr.dtype
Out[38]: dtype('float64')
```

위 예제에서는 정수형을 부동소수점으로 변환했다. 만약 보동소수점수를 정수형 dtype으로 변환하면 소수점 아래 자리는 바려진다.

```
In [39]: arr = np.array([3.7, -1.2,-2.6,0.5,12.9])

In [40]: arr
Out[40]: array([ 3.7, -1.2, -2.6,  0.5, 12.9])

In [41]: arr.astype(np.int32)
Out[41]: array([ 3, -1, -2,  0, 12])
```

CAUTION NumPy에서 문자열 데이터는 고정 크기를 가지며 별다른 경고를 출력하지 않고 입력을 임의로 잘라낼 수 있으므로 numpy.string_ 형을 이용할 때는 주의하는 것이 좋다. pandas는 숫자형식이 아닌 경우에 좀 더 직관적인 사용성을 제공한다.

만일 어떤 이유(문자열이 float64형으로 변환되지 않는 경우와 같은)로 형 변환이 실패하면 ValueError예외가 발생한다. 위 예에서 나는 귀찮아서 np.float64 대신 그냥 float라고 입력했는데 똑똑한 NumPy는 파이썬 자료형을 알맞는 dtype으로 맞춰준다.

다른 배열의 dtype 속성을 이용하는 것도 가능하다.

```
In [42]: int_array = np.arange(10)

In [43]: calibers = np.array([.22,.270,.357,.380,.44,.50], dtype=np.float64)

In [45]: int_array.astype(calibers.dtype)
Out[45]: array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

dtype으로 사용할 수 있는 축약 코드도 있다.

```
In [46]: empty_unit32 = np.empty(8,dtype='u4')

In [47]: empty_unit32
Out[47]:
array([         0,          0, 4100698352,      32762,          2,
                0, 4294967295, 4294967295], dtype=uint32)
```

NOTE astype을 호출하면 새로운 dtype 이 이전 dtype과 동일해도 항상 새로운 배열을 생성(데이터를 복사)한다.

4.1.3 NumPy 배열의 산술 연산

배열의 중요한 특징은 for문을 작성하지 않고 데이터를 일괄 처리할 수 있다는 것이다. 이를 **벡터화**라고 하는데, 같은 크기의 배열 간의 산술 연산은 산술 연산은 배열의 각 원소 단위로 적용된다.

```
In [48]: arr = np.array([[1.,2.,3.],[4.,5.,6.]])

In [49]: arr
Out[49]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [50]: arr * arr
Out[50]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [51]: arr - arr
Out[51]:
array([[0., 0., 0.],
       [0., 0., 0.]])
```

스칼라 인자가 포함된 산술 연산의 경우 배열 내의 모든 원소에 스칼라 인자가 적용된다.

```
In [52]: 1/arr
Out[52]:
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])

In [53]: arr ** 0.5
Out[53]:
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

같은 크기를 가지는 배열 간의 비교 연산은 불리언 배열을 반환한다.

```
In [54]: arr2 = np.array([[0.,4.,1.,],[7.,2.,12.]])

In [55]: arr2
Out[55]:
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])

In [56]: arr2 > arr
Out[56]:
array([[False,  True, False],
       [ True, False,  True]])
```

크기가 다른 배열 간의 연산은 **브로드캐스팅**이라고 하는데 12장에서 자세히 다루도록 하겠다.

4.1.4 색인과 슬라이싱 기초

NumPy 배열 색인에 대해서는 다룰 주제가 많다. 데이터의 부분집합이나 개별 요소를 선택하기 위한 수많은 방법이 존재한다. 1차원 배열은 단순한데, 표면적으로는 파이썬의 리스트와 유사하게 동작한다.

```
In [58]: arr = np.arange(10)

In [59]: arr
Out[59]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [60]: arr[5]
Out[60]: 5

In [61]: arr[5:8]
Out[61]: array([5, 6, 7])

In [62]: arr[5:8] = 12

In [63]: arr
Out[63]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

위에서 볼 수 있듯, arr[5:8] = 12 처럼 배열 조각에 스칼라값을 대입하면 12가 선택 영역 전체로 전파(또는 **브로드캐스팅**)된다. 리스트와의 중요한 차이점은 배열 조각은 원본 배열의 **뷰**라는 점이다. 즉, 데이터는 복사되지 않고 뷰에 대한 변경은 그대로 원본 배열에 반영된다.

이에 대한 예제로 먼저 arr배열의 슬라이스를 생성해보자.

```
In [64]: arr_slice = arr[5:8]

In [65]: arr_slice
Out[65]: array([12, 12, 12])
```

그리고 arr_slice의 값을 변경하면 원래 배열인 arr의 값도 바뀌어 있음을 확인할 수 있다.

```
In [66]: arr_slice[1]= 12345

In [67]: arr
Out[67]:
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

단순히 [:]로 슬라이스를 하면 배열의 모든 값을 할당한다.

```
In [68]: arr_slice[:]= 64

In [69]: arr
Out[69]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

NumPy를 처음 접한다면, 특히 데이터 복사가 자주 일어나는 다른 배열 프로그래밍 언어를 사용해본 적이 있다면 데이터가 복사되지 않는다는 점은 놀랄 만한 사실이다. NumPy는 대용량의 데이터 처리를 염두에 두고 설계되었기에 만약 NumPy가 데이터 복사를 남발한다면 성능과 메모리 문제에 마주치게 될 것이다.

CAUTION 만약에 뷰 대신 ndarray 슬라이스의 복사본을 얻고 싶다면 arr[5:8].copy()를 사용해서 명시적으로 배열을 복사해야 한다.

다차원 배열을 다룰 때는 좀 더 많은 옵션이 있다. 2차원 배열에서 각 색인에 해당하는 요소는 스칼라값이 아닌 1차원 배열이다.

```
In [70]: arr2d = np.array([[1,2,3],[4,5,6],[7,8,9]])

In [71]: arr2d[2]
Out[71]: array([7, 8, 9])
```

따라서 개별 요소는 재귀적으로 접근해야 한다. 하지만 그렇게 하기는 귀찮기에 콤마로 구분된 색인 리스트로 넘기면 된다. 그러므로 다음 두 표현은 동일하다.

```
In [72]: arr2d[0][2]
Out[72]: 3

In [73]: arr2d[0,2]
Out[73]: 3
```

2차원 배열에 대한 색인을 나타냈다. 0번 축을 '로우'로 생각하고 1번 축'칼럼'으로 생각하면 이해가 쉽다.

다차원 배열에서 마지막 색인을 생략하면 반환되는 객체는 상위 차원의 데이터를 포함하고 있는 한 차원 낮은 ndarray가 된다. 2 x 3 x 3 크기의 배열 arr3d가 있다면

```
In [74]: arr3d = np.array ([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])

In [75]: arr3d
Out[75]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

arr3d[0]은 2x3 크기의 배열이다 .

```
In [76]: arr3d[0]
Out[76]:
array([[1, 2, 3],
       [4, 5, 6]])
```

arr3d[0] 에는 스칼라값과 배열 모두 대입할 수 있다.

```
In [77]: old_values = arr3d[0].copy()

In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])

In [79]: arr3d[0]=32

In [80]: arr3d
Out[80]:
array([[[32, 32, 32],
        [32, 32, 32]],

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

In [82]: arr3d[0] = old_values

In [83]: arr3d[0]
Out[83]:
array([[1, 2, 3],
       [4, 5, 6]])

In [84]: arr3d
Out[84]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

이런 식으로 arr3d[1,0]은 (1,0) 으로 색인되는 1차원 배열과 그 값을 반환한다.

```
In [86]: arr3d[1,0]
Out[86]: array([7, 8, 9])
```

이 표현은 다음처럼 두 번에 걸쳐 인덱싱한 결과와 동일하다.

```
In [86]: arr3d[1,0]
Out[86]: array([7, 8, 9])

In [87]: x = arr3d[1]

In [88]: x
Out[88]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [89]: x[0]
Out[89]: array([7, 8, 9])
```

여기서 살펴본 선택된 배열의 부분집합은 모두 배열의 뷰를 반환한다는 점을 기억하자.

슬라이스로 선택하기

파이썬의 리스트 같은 1차원 객체처럼 ndarray는 비슷한 문법으로 슬라이싱할 수 있다.

```
In [90]: arr
Out[90]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [91]: arr[1:6]
Out[91]: array([ 1,  2,  3,  4, 64])
```

위에서 살펴본 arr2d를 생각해보자. 이 배열을 슬라이싱하는 방법은 조금 다르다.

```
In [92]: arr2d
Out[92]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [93]: arr2d[:2]
Out[93]:
array([[1, 2, 3],
       [4, 5, 6]])
```

확인한 것 처럼 첫 번째 축인 0축을 기준으로 슬라이싱 되었다. 따라서 슬라이스는 축을 따라 선택영역 내의 요소를 선택한다. arr2d[:2]sms 'arr2d의 시작부터 두 번째 로우까지의 선택 ' 이라고 이해하자.

색인을 여러 개 넘겨서 다차원을 슬라이싱하는 것도 가능하다.

```
In [95]: arr2d[:2,1:]
Out[95]:
array([[2, 3],
       [5, 6]])
```

이렇게 슬라이싱하면 항상 같은 차원의 배열에 대한 뷰를 얻게 된다. 정수 색인과 슬라이스를 함께 사용해서 한 차원 낮은 슬라이스를 얻을 수 있다.

예를 들어 두 번째 로우에서 처음 두 컬럼만 선택하고 싶다면 아래처럼 하면 된다.

```
In [96]: arr2d[1,:2]
Out[96]: array([4, 5])
```

이와 유사하게 처음 두 로우에서 세 번째 컬럼만 선택하고 싶다면 아래처럼 하면 된다.

그냥 콜론만 쓰면 전체 축을 선택한다는 의미이므로 이렇게 하면 원래 차원의 슬라이스를 얻게 된다.

```
In [11]: arr2d[:2,2]
Out[11]: array([3, 6])

In [13]: arr2d[:,:1]
Out[13]:
array([[1],
       [4],
       [7]])

In [18]: arr2d[:2,1:]=0
```

```
In [19]: arr2d
Out[19]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

4.1.5 불리언값으로 선택하기

중복된 이름이 포함된 배열이 있다고 하자. 그리고 numpy.random 모듈에 있는 randn 함수를 사용해서 임의의 표준 정류 분포 데이터를 생성하자.

```
In [20]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [21]: data = np.random.randn(7,4)

In [22]: names
Out[22]: array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

In [23]: data
Out[23]:
array([[ 2.09897418e-01, -9.46161207e-01, -8.44692856e-02,
         6.12104708e-01],
       [ 1.62673048e+00, -4.84653788e-01,  5.71792416e-03,
        -1.72339130e+00],
       [ 4.42358647e-01,  1.66430758e-01, -3.62444097e-02,
         9.15115756e-01],
       [ 2.19695232e+00,  1.31769709e+00, -8.99851890e-01,
        -1.17519068e+00],
       [-1.05613335e+00, -1.91278752e+00, -5.04175567e-04,
        -1.43825357e-01],
       [-1.80384480e-01, -1.37326836e+00, -5.38694921e-01,
        -2.18885022e+00],
       [ 1.04141291e+00,  7.25124588e-01, -3.34925382e-01,
        -1.34265619e+00]])
```

각각의 이름은 data 배열의 각 로우에 대응한다고 가정하자. 만약에 전체 로우에서 'Bob'과 같은 이름을 선택하려면 산술 연산과 마찬가지로 배열에 대한 비교 연산(==같은)도 벡터화되므로 names를 'Bob' 문자열과 비교하면 불리언 배열을 반환한다.

```
In [24]: names == 'Bob'
Out[24]: array([ True, False, False,  True, False, False, False])
```

이 불리언 배열은 배열의 색인으로 사용할 수 있다.

```
In [25]: data[names=='Bob']
Out[25]:
array([[ 0.20989742, -0.94616121, -0.08446929,  0.61210471],
       [ 2.19695232,  1.31769709, -0.89985189, -1.17519068]])
```

불리언 배열은 반드시 색인하려는 축의 길이와 동일한 길이를 가져야 한다. 불리언 배열 색인도 슬라이스나 요소를 선택하는 데 맞출 수 있다.

CAUTION_ 불리언값으로 배열을 선택할 때는 불리언 배열의 크기가 다르더라도 실패하지 않는다. 따라서 이 기능을 사용할 때는 항상 주위하자.

다음 예제에서는 names=='Bob' 인 로우에서 2: 컬럼을 선택했다.

```
In [26]: data[names == 'Bob', 2:]
Out[26]:
array([[-0.08446929,  0.61210471],
       [-0.89985189, -1.17519068]])

In [32]: data[names == 'Bob',3]
Out[32]: array([ 0.61210471, -1.17519068])
```

'Bob'이 아닌 요소들을 선택하려면 != 연산자를 사용하거나 ~를 사용해서 조건절을 부인하면 된다.

```
In [33]: names != 'Bob'
Out[33]: array([False,  True,  True, False,  True,  True,  True])

In [34]: data[~(names=='Bob')]
Out[34]:
array([[ 1.62673048e+00, -4.84653788e-01,  5.71792416e-03,
        -1.72339130e+00],
       [ 4.42358647e-01,  1.66430758e-01, -3.62444097e-02,
         9.15115756e-01],
       [-1.05613335e+00, -1.91278752e+00, -5.04175567e-04,
        -1.43825357e-01],
       [-1.80384480e-01, -1.37326836e+00, -5.38694921e-01,
        -2.18885022e+00],
       [ 1.04141291e+00,  7.25124588e-01, -3.34925382e-01,
        -1.34265619e+00]])
```

~ 연산자는 일반적인 조건을 반대로 쓰고 싶을 때 유용하다.

```
In [36]: data[~cond]
Out[36]:
array([[ 1.62673048e+00, -4.84653788e-01,  5.71792416e-03,
        -1.72339130e+00],
       [ 4.42358647e-01,  1.66430758e-01, -3.62444097e-02,
         9.15115756e-01],
       [-1.05613335e+00, -1.91278752e+00, -5.04175567e-04,
        -1.43825357e-01],
       [-1.80384480e-01, -1.37326836e+00, -5.38694921e-01,
        -2.18885022e+00],
       [ 1.04141291e+00,  7.25124588e-01, -3.34925382e-01,
        -1.34265619e+00]])
```

세 가지 이름 중에서 두 가지 이름을 선택하려면 &(and) 나 |(or) 같은 논리 연산자를 사용한 여러 개의 불리언 조건을 사용하면 된다.

```
In [38]: mask
Out[38]: array([ True, False,  True,  True,  True, False, False])

In [39]: data[mask]
Out[39]:
array([[ 2.09897418e-01, -9.46161207e-01, -8.44692856e-02,
         6.12104708e-01],
       [ 4.42358647e-01,  1.66430758e-01, -3.62444097e-02,
         9.15115756e-01],
       [ 2.19695232e+00,  1.31769709e+00, -8.99851890e-01,
        -1.17519068e+00],
       [-1.05613335e+00, -1.91278752e+00, -5.04175567e-04,
        -1.43825357e-01]])
```

배열에 불리언 색인을 이용해서 데이터를 선택하면 반환되는 배열의 내용이 바뀌지 않더라도 항상 데이터 복사가 발생한다.

CAUTION_ 파이썬 예약어인 and와 or은 불리언 배열에서는 사용할 수 없다. 대신 &(and) 와 |(or)을 사용한다.

불리언 배열에 값을 대입하는 것은 상식적으로 이뤄진다. data에 저장된 모든 음수를 0으로 대입하려면 다음과 같이 하면 된다.

```
In [40]: data[data < 0] = 0

In [41]: data
Out[41]:
array([[0.20989742, 0.        , 0.        , 0.61210471],
       [1.62673048, 0.        , 0.00571792, 0.        ],
       [0.44235865, 0.16643076, 0.        , 0.91511576],
       [2.19695232, 1.31769709, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ],
       [1.04141291, 0.72512459, 0.        , 0.        ]])
```

1차원 불리언 배열을 사용해서 전체 로우나 컬럼을 선택하는 것은 쉽게 할 수 있다.

```
In [42]: data[names != 'Joe']=7

In [43]: data
Out[43]:
array([[7.00000000e+00, 7.00000000e+00, 7.00000000e+00, 7.00000000e+00],
       [1.62673048e+00, 0.00000000e+00, 5.71792416e-03, 0.00000000e+00],
       [7.00000000e+00, 7.00000000e+00, 7.00000000e+00, 7.00000000e+00],
       [7.00000000e+00, 7.00000000e+00, 7.00000000e+00, 7.00000000e+00],
       [7.00000000e+00, 7.00000000e+00, 7.00000000e+00, 7.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [1.04141291e+00, 7.25124588e-01, 0.00000000e+00, 0.00000000e+00]])
```

나중에 살펴보겠지만 2차원 데이터에 대한 이런 유형의 연산은 pandas를 이용해서 처리하는 것이 편리하다.

4.1.6 팬시 색인

팬시 색인(fancy indexing)은 정수 배열을 사용한 색인 설명하기 위해 NumPy에서 차용한 단어이다. 8 x 4 크기의 배열이 있다고 하자.

```
In [44]: arr = np.empty((8,4))

In [45]: for i in range(8):
    ...:     arr[i] = i
    ...:

In [46]: arr
Out[46]:
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.]])
```

특정한 순서로 로우를 선택하고 싶다면 그냥 원하는 순서가 명시된 정수가 담긴 ndarray나 리스트를 넘기면 된다.

```
In [47]: arr[[4,3,0,6]]
Out[47]:
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
```

색인으로 음수를 사용하면 끝에서부터 로우를 선택한다.

```
In [48]: arr[[-3,-5,-7]]
Out[48]:
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

다차원 색인 배열을 넘기는 것은 조금 다르게 동작하는데, 각각의 색인 튜플에 대응하는 1차원 배열이 선택된다.

```
In [49]: arr = np.arange(32).reshape((8,4))

In [50]: arr
Out[50]:
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]])

In [51]: arr[[1,5,7,2],[0,3,1,2]]
Out[51]: array([ 4, 23, 29, 10])
```

reshape 메서드는 부록 A에서 더 자세히 살펴본다.

여기서 결괄르 보면 (1,0), (5,3), (7,1), (2,2)에 대응하는 원소들이 선택되었다. 배열이 몇 차원이든지(여기서는 2차원) 팬시 색인의 결과는 항상 1차원이다.

이 예제에서 팬시 색인은 나를 포함한 사용자들이 기대하는 것과는 조금 다르게 동작했다. 행렬의 행(로우)과 열(칼럼)에 대응하는 사각형 모양의 값이 선택되기를 기대했는데 그렇게 하려면 아래처럼 하면된다.

```
In [52]: arr[[1,5,7,2]][:,[0,3,1,2]]
Out[52]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

팬시 색인은 슬라이싱과는 달리 선택된 데이터를 새로운 배열로 복사한다.

4.1.7 배열 전치와 축 바꾸기

배열 전치는 데이터를 복사하지 않고 데이터의 모양이 바뀐 뷰를 반환하는 특별한 기능이다. ndarray 는 transpose 메서드와 T라는 이름의 특수한 속성을 갖고 있다.

```
In [53]: arr = np.arange(15).reshape((3,5))

In [54]: arr
Out[54]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [55]: arr.T
Out[55]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

행렬 계산을 할 때 자주 사용하게 될 텐데, 예를 들어 행렬의 내적은 np.dot을 이용해서 구할 수 있다.

```
In [56]: arr = np.random.randn(6,3)

In [57]: arr
Out[57]:
array([[ 1.36116084e+00, -1.06664683e-01,  3.99957270e-01],
       [ 1.62025219e+00, -2.98827523e-01,  8.58972859e-02],
       [ 2.09790193e-01, -6.56001846e-01, -6.10943687e-01],
       [-6.28823053e-01,  1.05009335e-01,  4.37306289e-01],
       [ 1.50599951e+00, -9.20278131e-04, -4.71834955e-01],
       [ 1.41554731e-01, -6.31843730e-02,  1.15791249e+00]])

In [58]: np.dot(arr.T,arr)
Out[58]:
array([[ 7.20547863, -0.84334877, -0.26625205],
       [-0.84334877,  0.54603374,  0.30564389],
       [-0.26625205,  0.30564389,  2.29522269]])
```

다차원 배열의 경우 transpose 메서드는 튜플로 축 번호를 받아서 치환한다.

```
In [60]: arr = np.arange(16).reshape((2,2,4))

In [61]: arr
Out[61]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

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

In [62]: arr.transpose((1,0,2))
Out[62]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

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

이 예제에서 첫 번째와 두 번째의 축 순서가 뒤바뀌었고 마지막 축은 그대로 남았다.

T 속성을 이용하는 간단한 전치는 축을 뒤바꾸는 특별한 경우다. ndarray에는 swapaxes라는 메서드가 있는데 두 개의 축 번호를 받아서 배열을 뒤바꾼다.

```
In [64]: arr.swapaxes(1,2)
Out[64]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

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

swapaxes도 마찬가지로 데이터를 복사하지 않고 원래 데이터에 대한 뷰를 반환한다.

단항 유니버설 함수

함수	설명
abs, fabs	각 원소(정수, 부동소수점수, 복소수)의 절대값을 구한다. 복소수가 아닌 경우에는 빠른 연산을 위해서 fabs를 사용한다.
sqrt	각 원소의 제곱근을 계산한다. arr ** 0.5 와 동일하다.
square	각 원소의 제곱을 계산한다. arr ** 2 와 동일하다.
exp	각 원소에서 지수 e^x 을 계산한다.
log, log10, log2, log1p	각각 자연로그, 로그 10, 로그 2, 로그 (1+x)
sign	각 원소의 부호를 계산한다. 1(양수), 0(영), -1(음수)
rint	각 원소의 소수자리를 반올림한다. dtype은 유지된다.
modf	각 원소의 몫과 나머지를 각각의 배열로 반환한다.
isnan	각 원소가 숫자가 아닌지 (NaN, Not a Number) 를 나타내는 불리언 배열을 반환한다.
isfinite, isinf	각각 배열의 각 원소가 유한한지 무한한지 나타내는 불리언 배열을 반환한다.
logical_not	각 원소의 논리 부정(not) 값을 계산한다. ~arr과 동일하다.

이항 유니버설 함수

함수	설명
power	첫 번째 배열의 원소를 두 번째 배열의 원소만큼 제곱한다.
maximum, fmax	각 배열의 두 원소 중 큰 값을 반환한다. fmax는 NaN을 무시한다.
manimum, fmin	각 배열의 두 원소 중 작은 값을 반환한다. fmin는 NaN을 무시한다.
mod	첫 번째 배열의 원소를 두 번째 배열의 원소로 나눈 나머지를 구한다.
copysign	첫 번째 배열의 원소의 기호를 두 번째 배열의 원소 기호로 바꾼다.
greater, greater_equal, less, less_equal, equal, not_equal	각각 두 원소 간의 >,>=,<,<=,==,!= 비교 연산 결과를 불리언 배열로 반환한다.
logical_and, logical_or, logical_xor	각각 두 원소 간의 &,|,^ 논리 연산 결과를 반환한다.

4.3 배열을 이용한 배열지향 프로그래밍

NumPy 배열을 사용하면 반복문을 작성하지 않고 간결한 배열 연산을 사용하여 많은 종류의 데이터 처리 작업을 할 수 있다.

배열 연산을 상ㅇ해서 반복문을 명시적으로 제거하는 기법을 흔히 벡터화라고 부르는데, 일반적으로 벡터화된 배열에 대한 산술 연산은 순수 파이썬 연산에 비해 2~3배에서 많게는 수십, 수백 배까지 빠르다. 부록 A에서 다룰 **브로드캐스팅**은 아주 강력한 벡터 연산 방법이다.

간단한 예로 값이 놓여 있는 그리드에서 sqrt(x^2 + y^2)을 계산한다고 하자. np.meshgrid 함수는 두 개의 1차원 배열을 받아서 가능한 모든(x,y) 짝을 만들 수 있는 2차원 배열 두개를 반환한다.

```
In [65]: points = np.arange(-5,5,0.01)

In [66]: xs,ys = np.meshgrid(points, points)

In [67]: ys
Out[67]:
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]])

In [68]: xs
Out[68]:
array([[-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       ...,
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99],
       [-5.  , -4.99, -4.98, ...,  4.97,  4.98,  4.99]])
```

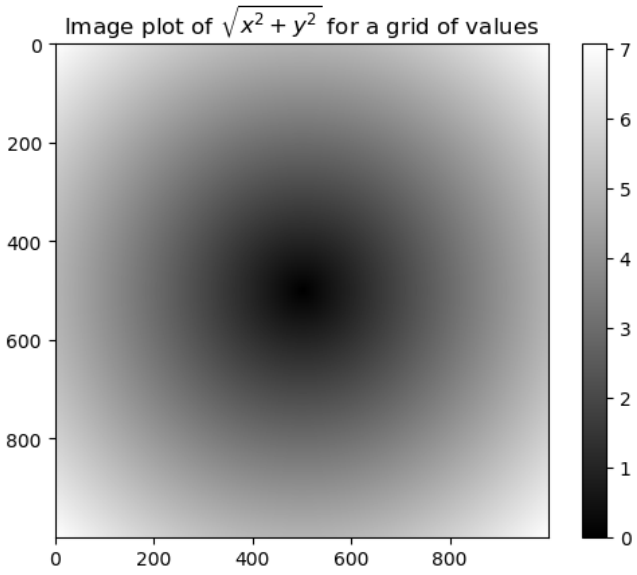
이제 그리드 상의 두 포인트로 간단하게 계산을 적용할 수 있다.

```
In [69]: z = np.sqrt(xs **2 + ys**2)

In [70]: z
Out[70]:
array([[7.07106781, 7.06400028, 7.05693985, ..., 7.04988652, 7.05693985,
        7.06400028],
       [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
        7.05692568],
       [7.05693985, 7.04985815, 7.04278354, ..., 7.03571603, 7.04278354,
        7.04985815],
       ...,
       [7.04988652, 7.04279774, 7.03571603, ..., 7.0286414 , 7.03571603,
        7.04279774],
       [7.05693985, 7.04985815, 7.04278354, ..., 7.03571603, 7.04278354,
        7.04985815],
       [7.06400028, 7.05692568, 7.04985815, ..., 7.04279774, 7.04985815,
        7.05692568]])
```

9장에서 살펴보겠지만 여기서 matplotlib을 이용해서 이 2차원 배열을 시각화 할 수 있다.

```
In [1]: import matplotlib.pyplot as plt

In [2]: import numpy as np
```

```
In [3]: x = np.arange(360,step = 20)

In [4]: y = np.sin(x*2*np.pi/360)

In [5]: plt.plot(x,y)
Out[5]: [<matplotlib.lines.Line2D at 0x229d8a35f08>]

In [6]: plt.show()

In [7]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-02025c5957eb> in <module>
----> 1 plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()

NameError: name 'z' is not defined

In [8]: z = np.sqrt(xs **2 + ys**2)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-92a6c92de1a5> in <module>
----> 1 z = np.sqrt(xs **2 + ys**2)

NameError: name 'xs' is not defined

In [9]: points = np.arange(-5,5,0.01)

In [10]: xs,ys = np.meshgrid(points, points)

In [11]: z = np.sqrt(xs **2 + ys**2)

In [12]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[12]: <matplotlib.colorbar.Colorbar at 0x229d7737d48>

In [13]: plt.title("Image plot of $\sqrt{x^2+y^2}$ for a grid of values")
Out[13]: Text(0.5, 1.0, 'Image plot of $\\sqrt{x^2+y^2}$ for a grid of values')
```



4.3.1 배열 연산으로 조건절 표현하기

numpy.where 함수는 x if 조건 else y 같은 삼항식의 벡터화된 버전이다. 다음과 같은 불리언 배열 하나와 값이 들어 있는 두 개의 배열이 있다고 하자.

```
In [3]: xarr = np.array([1.1,1.2,1.3,1.4,1.5])

In [5]: yarr = np.array([2.1,2.2,2.3,2.4,2.5])

In [6]: cond = np.array([True, False, True, True, False])
```

cond의 값이 True 일 때는 xarr의 값을 취하고 아니면 yarr의 값을 취하고 싶다면 리스트 표기법을 이용해서 다음처럼 작성할 수 있다.

```
In [8]: result = [(x if c else y)
   ...:              for x,y,c in zip(xarr,yarr,cond)]

In [9]: result
Out[9]: [1.1, 2.2, 1.3, 1.4, 2.5]
```

이 방법에는 몇 가지 문제가 있는데, 순수 파이썬으로 수행되기에 큰 배열을 빠르게 처리하지 못한다. 또한 다차원 배열에서는 사용할 수 없는 문제가 있다. np.where를 사용하면 아주 간결하게 작성할 수 있다.

```
In [10]: result = np.where(cond, xarr, yarr)

In [11]: result
Out[11]: array([1.1, 2.2, 1.3, 1.4, 2.5])
```

np.where의 두 번째와 세 번째 인자는 배열이 아니여도 상관없다. 둘 중 하나 혹은 둘 다 스칼라 값이어도 동작한다. 데이터 분석에서 일반적인 where의 사용은 다른 배열에 기반한 새로운 배열을 생성한다. 임의로 생성된 데이터가 들어있는 행렬이 있고 양수는 모두 2로, 음수는 모두 -2로 바꾸려면 np.where를 사용해서 쉽게 처리할 수 있다.

```
In [12]: arr = np.random.randn(4,4 )

In [13]: arr
Out[13]:
array([[-0.14796846,  0.71941459,  1.40472196,  1.58162662],
       [ 0.17369195,  0.19822631, -0.0581663 , -0.70889809],
       [ 1.88036314, -1.05458937, -1.28891143,  1.151842  ],
       [-0.15190048, -1.28445717, -1.67619482,  1.84952229]])

In [14]: arr > 0
Out[14]:
array([[False,  True,  True,  True],
       [ True,  True, False, False],
       [ True, False, False,  True],
       [False, False, False,  True]])

In [15]: np.where(arr>0,2,-2)
Out[15]:
array([[-2,  2,  2,  2],
       [ 2,  2, -2, -2],
       [ 2, -2, -2,  2],
       [-2, -2, -2,  2]])
```

np.where 를 사용할 때 스칼라값과 배열을 조합할 수 있다. 예를 들어 arr의 모든 양수를 2로 바꿀 수 있다.

```
In [16]: np.where(arr>0,2,arr)
Out[16]:
array([[-0.14796846,  2.        ,  2.        ,  2.        ],
       [ 2.        ,  2.        , -0.0581663 , -0.70889809],
       [ 2.        , -1.05458937, -1.28891143,  2.        ],
       [-0.15190048, -1.28445717, -1.67619482,  2.        ]])
```

np.where로 넘기는 배열은 그냥 크기만 같은 배열이거나 스칼라값이 될 수 있다.

4.3.2 수학 메서드와 통계 메서드

배열 전체 혹은 배열에서 한 축을 따르는 자료에 대한 통계를 계산하는 수학 함수는 배열 메서드로 사용할 수 있다. 전체의 합(sum) 이나 평균(mean), 표준편차(std)는 NumPy의 최상위 함수를 이용하거나 배열의 인스턴스 메서드를 사용해서 구할 수 있다.

임의의 정규 분포 데이터를 생성하고 집계해보자.

```
In [17]: arr = np.random.randn(5,4)

In [18]: arr
Out[18]:
array([[ 1.95775812,  1.15898139,  2.43694179, -1.10149932],
       [-0.77302676, -1.2754452 ,  0.19462044,  0.34263085],
       [ 1.0504324 ,  0.59230117,  1.71566561, -1.32981015],
       [-0.40014638, -0.39140109,  0.21741278,  0.87776667],
       [-0.12440938,  0.61968592, -0.20843264, -1.40740766]])

In [19]: arr.mean()
Out[19]: 0.20763092737636674

In [20]: np.mean(arr)
Out[20]: 0.20763092737636674

In [21]: arr.sum()
Out[21]: 4.152618547527335
```

mean이나 sum 같은 함수는 선택적으로 axis 인자를 받아서 해당 axis에 대한 통계를 계산하고 한 차수 낮은 배열을 반환한다.

```
In [23]: arr.mean(axis =1)
Out[23]: array([ 1.11304549, -0.37780517,  0.50714726,  0.07590799, -0.28014094])

In [24]: arr.sum(axis=0)
Out[24]: array([ 1.710608  ,  0.70412219,  4.35620797, -2.61831961])
```

여기서 arr.sum(0)은 로우의 합을 구하라는 의미이며, arr.mean(1)은 모든 컬럼에서 평균을 구하라는 의미이다.

cumsum과 cumprod 메서드는 중간 계산값을 담고 있는 배열을 반환한다.

```
In [25]: arr = np.array([0,1,2,3,4,5,6,7])

In [26]: arr.cumsum()
Out[26]: array([ 0,  1,  3,  6, 10, 15, 21, 28], dtype=int32)
```

다차원 배열에서 cumsum 같은 누산 함수는 같은 크기의 배열을 반환한다. 하지만 축을 지정하여 부분적으로 계산하면 낮은 차수의 슬라이스를 반환한다.

```
In [27]: arr = np.array([[0,1,2],[3,4,5],[6,7,8]])

In [28]: arr
Out[28]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [29]: arr.cumsum(axis=0)
Out[29]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]], dtype=int32)

In [30]: arr.cumprod(axis = 1)
Out[30]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]], dtype=int32)
```

다음 표에서 지원하는 모든 함수를 확인할 수 있다. 이들 메서드에 대한 다양한 예제는 다음 장에서 살펴볼 것이다.

기본 배열 통계 메서드

메서드	설명
sum	배열 전체 혹은 특정 축에 대한 모든 원소의 합을 계산한다. 크기가 0인 배열에 대한 sum 결과는 0이다.
mean	산술 평균을 구한다. 크기가 0인 배열에 대한 mean 결과는 NaN이다.
std, var	각각 표준편차(std), 분산(var)을 구한다. 선택적으로 자유도를 줄 수 있으며 분모의 기본 값은 n이다.
max,min	최솟값과 최댓값
argmin,argmax	최소 원소의 색인값과 최대 원소의 색인값
cumsum	각 원소의 누적합
cumprod	각 원소의 누적곱

4.3.3 불리언 배열을 위한 메서드

이전 메서드의 불리언값을 1(True) 또는 0(False)으로 강제할 수 있다. 따라서 sum메서드를 실행하면 불리언 배열에서 True인 원소의 개수를 셀 수 있다.

```
In [31]: arr= np.random.randn(100)

In [32]: arr
Out[32]:
array([-1.30674642e+00, -8.67988549e-01,  2.00722273e-01, -7.66226593e-01,
       -1.20873693e-01, -1.79642661e+00, -5.03325244e-01, -3.78642193e-01,
        2.00169017e-01,  1.79088206e+00,  2.26160501e+00,  3.30276211e-01,
        1.89644179e+00,  1.88006849e-01, -3.78275529e-01,  8.71050911e-01,
        7.75614235e-01,  6.16118317e-01,  8.76221454e-02,  1.06280664e+00,
       -9.33430941e-02,  6.04619859e-01, -1.58788310e+00, -1.04105094e-01,
       -1.68963790e+00, -1.60439244e-01, -1.15658058e+00, -1.20561247e+00,
       -1.44698313e-01,  8.76179833e-01,  1.25650034e+00, -2.39902923e-02,
        1.33532651e+00,  1.69271687e+00,  9.16017008e-01, -7.80972736e-01,
        2.02643413e+00,  1.58425372e+00,  1.67445940e+00,  1.64427869e+00,
       -4.59787281e-01, -7.83523447e-01, -2.20807259e-01, -1.71858593e+00,
       -1.14130587e+00, -5.78184892e-04,  8.09709507e-01,  2.32545035e-01,
        5.88654376e-01,  5.39845139e-01,  8.55568959e-01,  1.04164270e+00,
        1.80782818e+00, -8.94987362e-01, -1.16220836e-01, -9.88867703e-01,
       -1.01928373e+00,  9.33335141e-01, -5.87267514e-02, -3.27850339e-01,
       -1.80366281e+00,  1.40215289e+00, -5.34072364e-01,  6.02949383e-01,
       -6.96343185e-02,  1.06271091e-01, -3.79610020e-01, -8.32035666e-01,
       -5.18605260e-01, -1.36848235e-01,  1.21005420e+00,  1.59644894e+00,
       -1.63025483e+00,  7.01434358e-02,  4.73563022e-01, -1.24645191e+00,
        1.10659103e+00, -5.70711861e-01, -1.83232616e+00,  9.60989560e-01,
        5.68457299e-01, -5.30039885e-01,  2.56332279e-01, -6.47871117e-01,
       -6.80889973e-01, -3.01185536e-01, -3.28614534e-03,  1.18046412e+00,
       -1.61990615e-01, -1.22562051e+00, -2.19415220e+00,  1.23849497e+00,
       -2.50518384e-01,  2.27967098e-01,  6.51419396e-01, -2.18288526e-02,
       -2.32934609e-01, -1.52690857e+00,  6.30010269e-01,  4.74861326e-01])

In [33]: (arr>0).sum()
Out[33]: 47
```

any와 all 메서드는 불리언 배열에 특히 유용하다. any 메서드는 하나 이상의 값이 True인지 검사하고, all 메서드는 모든 원소가 True인지 검사한다.

```
In [34]: bools = np.array([False, False, True, False])

In [35]: bools.any
Out[35]: <function ndarray.any>

In [36]: bools.any()
Out[36]: True

In [37]: bools.all()
Out[37]: False
```

이들 메서드는 불리언 배열이 아니어도 동작하는데, 0이 아닌 원소는 모두 True로 간주한다.

4.3.4 정렬

파이썬의 내장 리스트형처럼 NumPy 배열 역시 sort 메서드를 이용해서 정렬할 수 있다.

```
In [44]: arr1 = np.random.randn(6)

In [45]: arr1.sort()

In [46]: arr1
Out[46]:
array([-1.16318382, -0.20101576,  0.46478562,  0.69451606,  1.04633644,
        1.52398169])
```

다차원 배열의 정렬은 sort 메서드에 넘긴 축의 값에 따라 1차원 부분을 정렬한다.

```
In [47]: arr = np.random.randn(5,3)

In [48]: arr
Out[48]:
array([[ 0.05353989, -1.71021358,  0.02956684],
       [-1.42768939, -0.55689324, -0.94447856],
       [-0.2774308 , -1.0419098 ,  2.0498365 ],
       [-3.14784389,  0.88566828,  0.55596468],
       [ 1.28422369, -0.3368726 , -0.19904185]])

In [49]: arr.sort(1)

In [50]: arr
Out[50]:
array([[-1.71021358,  0.02956684,  0.05353989],
       [-1.42768939, -0.94447856, -0.55689324],
       [-1.0419098 , -0.2774308 ,  2.0498365 ],
       [-3.14784389,  0.55596468,  0.88566828],
       [-0.3368726 , -0.19904185,  1.28422369]])
```

np.sort 메서드는 배열을 직접 변경하지 않고 정렬된 결과를 가지고 있는 복사본을 반환한다.

배열의 분위수를 구하는 쉽고 빠른 방법은 우선 배열을 정렬한 후 특정 분위의 값을 선택하는 것이다.

```
In [51]: large_arr = np.random.randn(100)

In [52]: large_arr.sort()

In [53]: large_arr[int(0.05*len(large_arr))]
Out[53]: -1.6795670361056048
```

NumPy의 정렬 메서드에 관한 자세한 내용과 간접 정렬 같은 고급 기법은 부록 A를 참조하자. 정렬과 관련된 다른 여러 가지 데이터 처리(표 형식의 데이터를 하나 이상의 열로 정렬하기 같은)내용은 pandas에서 다룬다.

4.3.5 집합관련함수

NumPy는 1차원 ndarray를 위한 몇 가지 기본적인 집합 연산을 제공한다. 아마도 가장 자주 사용하는 함수는 배열 내에서 중복된 원소를 제거하고 남은 원소를 정렬된 형태로 반환하는 np.unique 일 것이다.

```
In [1]: import numpy as np

In [2]: names = np.array(['Bob','Joe', 'Will','Bob', 'Will', 'Joe', 'Joe' ])
```

```
In [3]: np.unique(names)
Out[3]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [4]: ints = np.array([3,3,3,2,2,1,1,4,4])

In [5]: ints
Out[5]: array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [6]: np.unique(ints)
Out[6]: array([1, 2, 3, 4])
```

np.unique를 순수 파이썬만으로 구현하면 다음과 같다.

```
In [7]: sorted(set(names))
Out[7]: ['Bob', 'Joe', 'Will']
```

np.in1d 함수는 두 개의 배열을 인자로 받아서 첫 번째 배열의 원소가 두 번째 배열의 원소를 포함하는지 나타내는 불리언 배열을 반환한다.

```
In [8]: values = np.array([6,0,0,3,2,5,6])

In [9]: np.in1d(values,[2,3,6])
Out[9]: array([ True, False, False,  True,  True, False,  True])
```

다음표는 NumPy에서 제공하는 집합 함수를 정리한 것이다.

매서드	설명
unique()	배열 x에서 중복된 원소를 제거한 뒤 정렬하여 반환한다.
intersect1d(x,y)	배열 x와 y에 공통적으로 존재하는 원소를 정렬하여 반환한다.
union1d(x,y)	두 배열의 합집합을 반환한다.
in1d(x,y)	x의 원소가 y의 원소에 포함되는지 나타내는 불리언 배열을 반환한다.
setdiff1d(x,y)	x와 y의 차집합을 반환한다.
setxor1d(x,y)	한 배열에는 포함되지만 두 배열 모두에는 포함되지 않는 원소들의 집합인 대칭 차집합을 반환한다.

4.4 배열 데이터의 파일 입출력

NumPy는 디스크에서 텍스트나 바이너리 형식의 데이터를 불러오거나 저장할 수 있다. 여기서는 NumPy의 내장 이진 형식만 살펴본다. 많은 사람들이 텍스트나 표 형식의 데이터는 pandas나 다른 도구를 사용해서 처리하는 것을 선호하므로 6장에서 더 살펴보도록 하자.

np.save와 np.load는 배열 데이터를 효과적으로 디스크에 저장하고 불러오기 위한 함수이다.

배열은 기본적으로 압축되지 않은 원시 (가동되지 않은) 바이너리 형식의 .npy 파일로 저장된다.

```
In [10]: arr = np.arange(10)

In [11]: np.save('some_array',arr)
```

저장되는 파일 경로가 .npy로 끝나지 않으면 자동적으로 확장자가 추가된다. 이렇게 저장된 배열은 np.load를 이용해서 불러올 수 있다.

```
In [12]: np.load('some_array.npy')
Out[12]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

np.savez 함수를 이용하면 여러 개의 배열을 압축된 형식으로 저장할 수 있는데, 저장하려는 배열을 키워드 인자 형태로 전달한다.

```
In [13]: np.savez('array_archive.npz',a=arr,b=arr)
```

npz파일을 불러올 때는 각각의 배열을 필요할 때 불러올 수 있도록 사전 형식의 객체에 저장한다.

```
In [13]: np.savez('array_archive.npz',a=arr,b=arr)

In [14]: arch = np.load('array_archive.npz')

In [15]: arch['b']
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [16]: arch['a']
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

압축이 잘되는 형식의 데이터라면 대신 numpy.savez_compressed 를 사용하자.

```
In [18]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)

In [19]: acp = np.load('arrays_compressed.npz')

In [20]: acp[a]
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-271a87f48740> in <module>
----> 1 acp[a]

NameError: name 'a' is not defined

In [21]: acp['a']
Out[21]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

4.5 선형대수

행렬의 곱셈, 분할, 행렬식 그리고 정사각 행렬 수학 같은 선형대수는 배열을 다루는 라이브러리에서 중요한 부분이다. 매트랩 같은 언어와 다르게 두 개의 2차원 배열을 *연산자로 곱하면 행렬 곱셈이 아니라 대응하는 각각의 원소의 곱을 계산한다. 행렬 곱셈은 배열 메서드 이자 numpy 네임스페이스안에 있는 dot 함수를 이용해서 계산한다.

```
In [22]: x = np.array([[1.,2.,3.],[4.,5.,6.]])

In [23]: y = np.array([[6.,23],[-1.,7.],[8.,9.]])

In [24]: x
Out[24]:
array([[1., 2., 3.],
       [4., 5., 6.]])

In [25]: y
Out[25]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])

In [26]: x.dot(y)
Out[26]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

x.dot(y)는 np.dot(x,y)와 동일하다.

```
In [27]: np.dot(x,y)
Out[27]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

2차원 배열과 곱셈이 가능한 크기의 1차원 배열 간의 행렬 곱셈의 결과는 1차원 배열이다.

```


In [28]: np.dot(x,np.ones(3))
Out[28]: array([ 6., 15.])

#파이썬 3.5부터 사용할 수 있는 @기호는 행렬 곱셈을 수행하는 연산자이다.
In [29]: x @ np.ones(3)
Out[29]: array([ 6., 15.])
```

numpy.linalg는 행렬의 분할과 역행렬, 행렬식과 같은 것들을 포함하고 있다. 이는 매트랩, R 같은 언어에서 사용하는 포트란 라이프러리인 BLAS, LAPACK 또는 Intel MKL(NumPy 빌드에 따라 다르다)을 사용해서 구현되었다.

함수	설명
diag	정사각 행렬의 대각/비대각 원소를 1차원 배열로 반환하거나, 1차원 배열을 대각선 원소로 하고 나머지는 0으로 채운 단위행렬을 반환한다.
dot	행렬 곱셈
trace	행렬의 대각선 원소의 합을 계산한다.
det	행렬식을 계산한다.
eig	정사각 행렬의 고윳값과 고유벡터를 계산한다.
inv	정사각 행렬의 역행렬을 계산한다.
qr	QR분해를 계산한다.
pinv	정사각 해열ㄹ의 무어-펜로즈 유사역원 역행렬을 구한다.
svd	특잇값 분해(SVD)를 계산한다.
solve	A가 정사각 행렬일 때 Ax=b를 만족하는 x를 구한다.
listsq	Ax = b를 만족하는 최소제곱해를 구한다.

```
In [34]: from numpy.linalg import inv, qr

In [35]: X = np.random.randn(5,5)

In [36]: mat = X.T.dot(X)

In [43]: inv(mat)
Out[43]:
array([[ 6.15514339, -0.48418814, -3.4622115 , -4.35595172, 10.15173582],
       [-0.48418814,  0.36373976, -0.18614573,  0.09576768, -0.40554675],
       [-3.4622115 , -0.18614573,  3.82000399,  3.5137333 , -6.67910765],
       [-4.35595172,  0.09576768,  3.5137333 ,  3.84424165, -7.78906069],
       [10.15173582, -0.40554675, -6.67910765, -7.78906069, 18.19816797]])

In [44]: X
Out[44]:
array([[-0.03532845, -0.77586854,  0.90268773, -1.41252202, -0.32628393],
       [-0.79047343, -1.27634581, -0.62427085, -0.31085812, -0.03704607],
       [ 0.03911698, -0.59806512, -0.19366217, -0.35612913, -0.05317954],
       [-1.61919099,  0.02909907,  1.17123404, -1.50028291,  0.73211667],
       [ 0.69213934,  1.68090291,  0.62183286, -1.4566854 , -0.76343739]])

In [45]: mat.dot(inv(mat))
Out[45]:
array([[ 1.00000000e+00, -6.20442714e-18,  1.45616836e-16,
        -5.00914244e-16, -4.06056443e-16],
       [-7.90490144e-16,  1.00000000e+00, -1.27767524e-15,
        -4.83800425e-16, -8.62131132e-16],
       [ 1.14786332e-15, -1.08542718e-16,  1.00000000e+00,
        -2.61157815e-15,  6.27677443e-15],
       [ 1.23753848e-15,  1.40953599e-16, -3.40501792e-15,
         1.00000000e+00,  9.02085990e-16],
       [ 7.27561483e-16, -1.99010979e-16,  3.80444781e-16,
```

```
        -7.40411346e-16,  1.00000000e+00]])

In [46]: q,r = qr(mat)

In [48]: r
Out[48]:
array([[-5.01642839, -3.66388201,  2.09737793, -3.74180425,  1.90332224],
       [ 0.        , -4.85622265, -3.94879533,  5.09521062,  0.61998341],
       [ 0.        ,  0.        , -2.28566556,  4.35786446,  1.04599337],
       [ 0.        ,  0.        ,  0.        , -1.33150028, -0.5906968 ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.04304609]])


In [49]: q
Out[49]:
array([[-0.74325048,  0.12230633, -0.45054128, -0.19664691,  0.43699248],
       [-0.42445244, -0.79482563,  0.41861023,  0.11204048, -0.0174572 ],
       [ 0.20174775, -0.41817347, -0.40518037, -0.73320937, -0.28750944],
       [-0.33942713,  0.41804967,  0.57415906, -0.51763563, -0.33528857],
       [ 0.33392502, -0.06048807,  0.35769857, -0.37846307,  0.78335989]])
```

4.6 난수 생성

numpy.random 모듈은 파이썬 내장 random 함수를 보강하여 다양한 종류의 확률분포로부터 효과적으로 표본값을 생성하는 데 주로 사용된다. 예를 들어, normal을 사용하여 표준정규분포로 부터 4X4 크기의 표본을 생성할 수 있다.

```
In [52]: samples = np.random.normal(size=(4,4))

In [53]: samples
Out[53]:
array([[ 0.55622908, -0.05607987, -1.97894557, -1.8930354 ],
       [ 0.95688059, -1.05965315, -0.42172895,  0.42483578],
       [ 1.12144906,  1.55769297, -1.26571692, -0.21105079],
       [-0.45595811, -0.05739698, -2.11969903,  1.23615406]])
```

이에 대해 파이썬 내장 random 모듈은 한 번에 하나의 값만 생성할 수 있다. 다음 성능 비교에서 알 수 있듯이 numpy.random은 매우 큰 표본을 생성하는데 파이썬 내장 모듈보다 수십 배 이상 빠르다.

```
In [54]: from random import normalvariate

In [55]: N = 1000000

In [58]: %timeit samples = [normalvariate(0,1) for _ in range(N)]
919 ms ± 63.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [59]: %timeit np.random.normal(size=N)
29.3 ms ± 464 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

이를 엄밀하게 **유사난수** 라고 부르는데, 난수 생성기의 **시드값**에 따라 정해진 난수를 알고리즘으로 생성하기 때문이다. NumPy 난수 생성기의 시드값은 np.random.seed를 이용해서 변경할 수 있다.

```
In [60]: np.random.seed(1234)
```

numpy.random에서 제공하는 데이터를 생성할 수 있는 함수들은 전역 난수 시드값을 이용한다. numpy.random.RandomState를 이용해서 다른 난수 생성기로부터 격리된 난수 생성기를 만들 수 있다.

```
In [63]: rng =np.random.RandomState(1234)

In [64]: rng.randn(10)
Out[64]:
array([ 0.47143516, -1.19097569,  1.43270697, -0.3126519 , -0.72058873,
        0.88716294,  0.85958841, -0.6365235 ,  0.01569637, -2.24268495])
```

다음 표는 numpy.random에 포함된 일부 함수다. 다음 절에서 큰 표본을 생성하기 위해 이 함수들의 기능을 이용하는 예를 살펴보도록 하겠다.

함수	설명
seed	난수 생성기의 시드를 지정한다.
permutation	순서를 임의로 바꾸거나 임의의 순열을 반환한다.
shuffle	리스트나 배열의 순서를 뒤섞는다.
rand	균등분포에서 표본을 추출한다.
randint	주어진 최소/최대 범위 안에서 임의의 난수를 추출한다.
randn	표준편차가 1이고 평균값이 0인 정규분포(매트랩과 같은 방식)에서 표본을 추출한다.
binomial	이항분포에서 표본을 추출한다.
normal	정규분포(가우시안)에서 표본을 추출한다.
beta	베타분포에서 표본을 추출한다.
chisquare	카이제곱분포에서 표본을 추출한다.
gamma	감마분포에서 표본을 추출한다.
uniform	균등[0,1) 분포에서 표본을 추출한다.

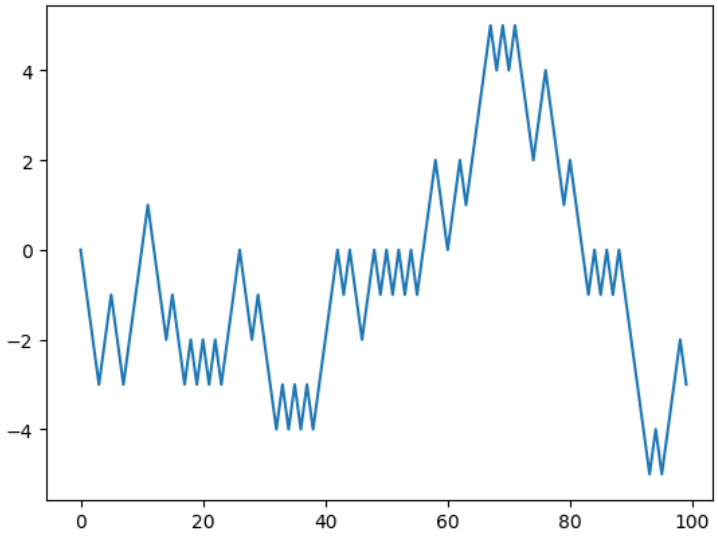
4.7 계단 오르내리기 예제

계단 오르내리기 예제는 배열 연산의 활용을 보여줄 수 있는 간단한 어플리케이션이다. 계단 중간에서 같은 확률로 한 계단 오르거나 내려간다고 가정하자.

순수 파이썬 내장 random 모듈을 사용해서 계단 오르내리기를 1,000번 수행하는 코드는 다음처럼 작성할 수 있다.

```
In [99]: 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)
    ...:

In [101]:  import matplotlib.pyplot as plt
     ...:
     ...:

In [102]: plt.plot(walk[:100])
Out[102]: [<matplotlib.lines.Line2D at 0x2154ce61708>]

In [103]: plt.show()
```



walk는 계단을 오르거나(+1) 내려간 (-1) 값의 누적합이라는 사실을 알 수 있으며 배열식으로 나타낼 수 있다. 이에 np.random 모듈을 사용해서 1,000번 수행한 결과 (1,-1)를 한 번에 저장하고 누적합을 계산한다.

```
In [107]: nsteps = 1000

In [108]: draws = np.random.randint(0,2, size = nsteps)

In [110]: steps = np.where(draws > 0,-1,1)

In [111]: walk = steps.cumsum()
```

이것으로 계단을 오르내린 위치의 최솟값과 최댓값 같은 간단한 통계를 구할 수 있다.

```
In [112]: walk.min()
Out[112]: -60

In [113]: walk.max()
Out[113]: 9
```

계단에서 특정 위치에 도달하기까지의 시간 같은 좀 더 복잡한 통계를 구할 수 있는데, 계단의 처음 위치에서 최초로 10칸 떨어지기까지 얼마나 걸렸는지 계산해보자. np.abs(walk) >= 10으로 처음 위치에서 10칸 이상 떨어진 시점을 알려주는 불리언 배열을 얻을 수 있다. 우리는 **최초의** 10 혹은 -10인 시점을 구해야 하므로 불리언 배열에서 최댓값의 처음 색인을 반환하는 argmax를 활용하자. (True가 최댓값이다)

```
In [114]: (np.abs(walk)>=10).argmax()
Out[114]: 297
```

여기서 argmax를 사용했지만 argmax는 배열 전체를 모두 확인하기 때문에 효과적인 방법은 아니다. 또한 이 예제에서는 True가 최댓값임을 이미 알고 있었다.

4.7.1 한 번에 시뮬레이션하기

계단 오르내리기를 많은 횟수(대략 5,000회 정도) 시뮬레이션하더라도 위 코드를 조금만 수정해서 해결할 수 있다. numpy.random 함수에 크기가 2인 튜플을 넘기면 2차원 배열이 생성되고 각 컬럼에서 누적합을 구해서 5,000회의 시뮬레이션을 한 번에 처리할 수 있다.

```
In [115]: nwalks = 5000

In [116]: nsteps = 1000

In [117]: draws = np.random.randint(0, 2, size = (nwalks, nsteps))
#draws에 (5000,1000) 사이즈 행렬에 0 또는 1을 쳐 넣어라
In [118]: steps = np.where(draws > 0,-1,1)
#draws가 0보다 큰 것을 -1로 바꾸고, 아닌 것은 1로 둬라
In [119]: walks = steps.cumsum(1)

In [120]: walks
Out[120]:
array([[ -1,  -2,  -3, ..., -46, -47, -46],
       [ -1,   0,  -1, ..., -40, -41, -42],
       [ -1,  -2,  -3, ...,  26,  27,  28],
       ...,
       [ -1,   0,  -1, ..., -64, -65, -66],
       [ -1,  -2,  -1, ...,  -2,  -1,   0],
       [  1,   2,   3, ..., -32, -33, -34]], dtype=int32)
```

이제 모든 시뮬레이션에 대해 최댓값과 최솟값을 구해보자.

```
In [122]: walks.max()
Out[122]: 128

In [123]: walks.min()
Out[123]: -122
```

이 데이터에서 누적합이 30혹은 -30이 되는 최소시점을 계산해보자. 5,000회의 시뮬레이션 중 모든 경우가 30에 도달하지 않기에 약간 까다로운데, any 메서드를 이용해서 해결할 수 있다.

```
In [124]: hits30 = (np.abs(walks)>=30).any(1)

In [125]: hits30
Out[125]: array([ True,  True,  True, ...,  True, False,  True])

In [127]: hits30.sum() # 누적합이 30 또는 -30인 경우
Out[127]: 3368
```

이 불리언 배열을 사용해서 walks에서 칼럼을 선택하고 절댓값이 30이 넘는 경우에 대해 축 1의 argmax 값을 구하면 처음 위치에서 30칸 이상 멀어지는 최소 횟수를 구할 수 있다.

```
In [132]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [133]: crossing_times
Out[133]: array([133, 395, 343, ..., 409, 297, 747], dtype=int64)

In [135]: crossing_times.mean()
Out[135]: 509.99762470308787
```

다른 분포를 사용해서 여러 가지 시도를 해보자. normal 함수에 표준편차와 평균값을 넣어서 정규분포에서 표본을 추출하는 것처럼 그냥 다른 난수 발생 함수를 사용하기만 하면 된다.

```
In [137]: steps = np.random.normal(loc=0,scale=0.25,size = (nwalks, nsteps))

In [138]: steps
Out[138]:
array([[-0.20472197, -0.03407671, -0.24781626, ...,  0.22195392,
         0.21250584,  0.43994277],
       [-0.09066595,  0.04741116,  0.09872352, ...,  0.10684413,
         0.2540548 , -0.22255203],
       [ 0.00942324,  0.03418277, -0.42305453, ..., -0.05711872,
        -0.11820873, -0.71117832],
       ...,
       [ 0.1230118 , -0.18155722,  0.12782267, ...,  0.0730254 ,
         0.06546779,  0.47920034],
       [-0.20550269,  0.16459572, -0.03138668, ...,  0.06915243,
         0.07831634,  0.27554056],
       [-0.23056172,  0.28924403,  0.08444807, ..., -0.27018412,
        -0.02140732,  0.06343304]])
```

4.8 마치며

이 책에서는 데이터를 다루기 위해 주로 pandas를 사용하지만 배열 기반 방식도 쭉 살펴볼 것이다.