O'REILLY®

영화 평점, 이름 통계, 선거 데이터 등 실사례 사용

Python for Data Analysis

파이썬 라이브러리를 활용한 데이터 분석 2판

웨스 맥키니 지음
김영근 옮김

한빛미디어
Hanbit Media, Inc.

CHAPTER 3 내장 자료구조, 함수, 파일

3.1 자료구조와 순차 자료형 ····· **87**

3.1.1 튜플 ····· **87**

3.1.2 리스트 ····· **92**

3.1.3 내장 순차 자료형 함수 ····· **98**

3.1.4 사전 ····· **101**

3.1.5 집합 ····· **106**

3.1.6 리스트, 집합, 사전 표기법 ····· **109**

3.2 함수 ····· **112**

3.2.1 네임스페이스, 스코프, 지역 함수 ····· **113**

3.2.2 여러 값 반환하기 ····· **114**

3.2.3 함수도 객체다 ····· **115**

3.2.4 익명 함수 ····· **117**

3.2.5 커링: 일부 인자만 취하기 ····· **118**

3.2.6 제너레이터 ····· **119**

3.2.7 에러와 예외 처리 ····· **122**

3.3 파일과 운영체제 ····· **126**

3.3.1 바이트와 유니코드 ····· **130**

3.4 마치며 ····· **132**

CHAPTER 4 NumPy 기본: 배열과 벡터 연산

4.1 NumPy ndarray: 다차원 배열 객체 ····· **135**

4.1.1 ndarray 생성하기 ····· **137**

4.1.2 ndarray의 dtype ····· **139**

CONTENTS

4.1.3 NumPy 배열의 산술 연산 ····· 142
4.1.4 색인과 슬라이싱 기초 ····· 144
4.1.5 불리언값으로 선택하기 ····· 150
4.1.6 팬시 색인 ····· 153
4.1.7 배열 전치와 축 바꾸기 ····· 155
4.2 유니버설 함수 : 배열의 각 원소를 빠르게 처리하는 함수 ····· 158
4.3 배열을 이용한 배열지향 프로그래밍 ····· 161
4.3.1 배열 연산으로 조건절 표현하기 ····· 163
4.3.2 수학 메서드와 통계 메서드 ····· 165
4.3.3 불리언 배열을 위한 메서드 ····· 167
4.3.4 정렬 ····· 168
4.3.5 집합 관련 함수 ····· 169
4.4 배열 데이터의 파일 입출력 ····· 171
4.5 선형대수 ····· 172
4.6 난수 생성 ····· 174
4.7 계단 오르내리기 예제 ····· 176
4.7.1 한 번에 시뮬레이션하기 ····· 178
4.8 마치며 ····· 180

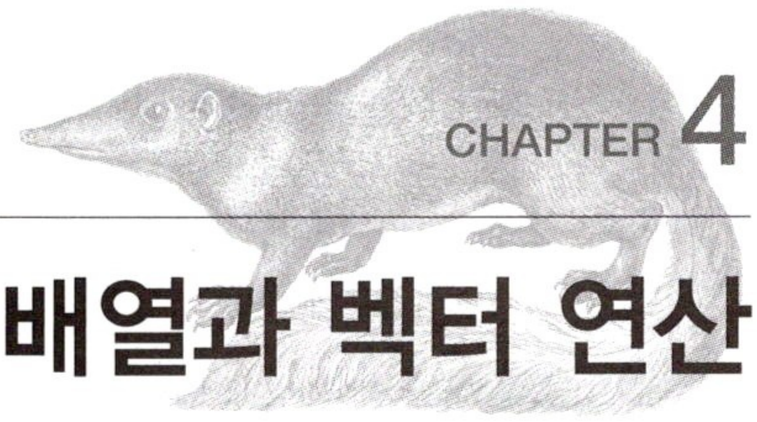
CHAPTER 5 pandas 시작하기

5.1 pandas 자료구조 소개 ····· 182
5.1.1 Series ····· 182
5.1.2 DataFrame ····· 187
5.1.3 색인 객체 ····· 195
5.2 핵심 기능 ····· 198
5.2.1 재색인 ····· 198

5.2.2 하나의 로우나 컬럼 삭제하기 ····· 201
5.2.3 색인하기, 선택하기, 거르기 ····· 203
5.2.4 정수 색인 ····· 209
5.2.5 산술 연산과 데이터 정렬 ····· 210
5.2.6 함수 적용과 매핑 ····· 217
5.2.7 정렬과 순위 ····· 220
5.2.8 중복 색인 ····· 224
5.3 기술 통계 계산과 요약 ····· 226
5.3.1 상관관계와 공분산 ····· 229
5.3.2 유일값, 값 세기, 멤버십 ····· 232
5.4 마치며 ····· 235

CHAPTER 6 데이터 로딩과 저장, 파일 형식

6.1 텍스트 파일에서 데이터를 읽고 쓰는 법 ····· 237
6.1.1 텍스트 파일 조금씩 읽어오기 ····· 244
6.1.2 데이터를 텍스트 형식으로 기록하기 ····· 246
6.1.3 구분자 형식 다루기 ····· 248
6.1.4 JSON 데이터 ····· 251
6.1.5 XML과 HTML: 웹 스크래핑 ····· 253
6.2 이진 데이터 형식 ····· 258
6.2.1 HDF5 형식 사용하기 ····· 259
6.2.2 마이크로소프트 엑셀 파일에서 데이터 읽어오기 ····· 261
6.3 웹 API와 함께 사용하기 ····· 263
6.4 데이터베이스와 함께 사용하기 ····· 264
6.5 마치며 ····· 267



CHAPTER 4

NumPy 기본: 배열과 벡터 연산

NumPy는 Numerical Python의 줄임말로, 파이썬에서 산술 계산을 위한 가장 중요한 필수 패키지 중 하나다. 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.
- 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 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 기능을 제공하지 않으므로 먼저 NumPy 배열과 배열 기반 연산에 대한 이해를 한 다음 pandas 같은 배열 기반 도구를 사용하면 훨씬 더 효율적이다. NumPy만으로도 방대한 주제이므로 브로드캐스팅 같은 NumPy의 고급 기능은 부록 A에서 따로 다루도록 하겠다.

대부분의 데이터 분석 애플리케이션에서 내가 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-else`를 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 헝구닌(Jim Hungunin)이 Numeric 라이브러리를 작성했던 1995년까지 거슬러 올라간다. 이후로 10년이 지나고 많은 과학 계산 커뮤니티는 배열 프로그래밍에 파이썬을 사용하기 시작했으나 라이브러리 생태계는 2000년대 초에 갈라지게 된다. 2005년 트라비스 올리펀트(Travis Oliphant)가 당시의 Numeric 라이브러리와 Numarray 프로젝트로부터 NumPy 프로젝트를 시작하여 커뮤니티에 단일 배열 컴퓨팅 프레임워크를 소개하게 되었다.

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 내부적으로 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 확인하기 위해 백만 개의 정수를 저장하는 NumPy 배열과 파이썬 리스트를 비교해 보자.

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

이제 각각의 배열과 리스트 원소에 2를 곱해보자.

```
In [10]: %time for _ in range(10): my_arr2 = my_arr * 2
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2 for x in my_list]
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

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 *N*차원의 배열 객체인데 파이썬에서 사용할 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 있도록 해준다.

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

```
In [12]: import numpy as np

# 임의의 값을 생성
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047, 0.4789, -0.5194],
       [-0.5557, 1.9658, 1.3934]])
```

그리고 그 값에 산술 연산을 해보자.

```
In [15]: data * 10
Out[15]:
array([[ -2.0471, 4.7894, -5.1944],
       [ -5.5573, 19.6578, 13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094, 0.9579, -1.0389],
       [-1.1115, 3.9316, 2.7868]])
```

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 지양하기 바란다. 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 [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

이 장에서는 NumPy의 배열을 사용하는 기초 방법을 소개하여 앞으로 책을 읽어나가는 데 충분한 도움을 줄 것이다. 대부분의 데이터 분석 애플리케이션을 작성하는 데 NumPy의 깊은 이해가 필수 사항은 아니며, 배열 위주의 프로그래밍과 생각하는 방법에 능숙해지는 것이 파이썬을 이용한 과학 계산의 고수가 되는 지름길이다.

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 [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,7.5 , 8. , 0. , 1. ])
```

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

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
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 [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

명시적으로 지정(자세한 내용은 잠시 후에 살펴보겠다)하지 않는 한 np.array는 생성될 때 적절한 자료형을 추론한다. 그렇게 추론된 자료형은 dtype 객체에 저장되는데 앞서 살펴본 예제에서 확인해보면 다음과 같다.

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

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

```
In [29]: np.zeros(10)
Out[29]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.]])

In [31]: np.zeros((2, 3, 2))
Out[31]:
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.

arange는 파이썬의 range 함수의 배열 버전이다.

```
In [32]: np.arange(15)
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

[표 4-1]은 표준 배열 생성 함수의 목록이다. NumPy는 산술 연산에 초점이 맞춰져 있기 때문에 만약 자료형을 명시하지 않으면 float64(부동소수점)가 될 것이다.

표 4-1 배열 생성 함수

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 0으로 채운다.
empty, empty_like	메모리를 할당하여 새로운 배열을 생성하지만 ones나 zeros처럼 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(좌상단에서 우하단을 잇는 대각선은 1로 채워지고 나머지는 0으로 채워진다).

4.1.2 ndarray의 dtype

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

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')
```

```
In [36]: arr2.dtype
Out[36]: 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 `float`나 `int` 같은 자료형의 이름과 하나의 원소가 차지하는 비트 수로 이루어진다. 파이썬의 `float` 객체에서 사용되는 표준 배정밀도 부동소수점[double-precision floating point] 값은 8바이트 혹은 64비트로 이루어지는데 이 자료형은 NumPy에서 `float64`로 표현된다. [표 4-2]는 NumPy가 지원하는 모든 자료형의 목록이다.

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

표 4-2 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 `float`형과 호환
`float64`	`f8` 또는 `d`	배정밀도 부동소수점. C언어의 `double`형과 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`	`?`	`True`와 `False` 값을 저장하는 불리언형
`object`	`O`	파이썬 객체형
`string_`	`S`	고정 길이 아스키 문자열형(각 문자는 1바이트). 길이가 10인 문자열 dtype은 `S10`이 된다.
`unicode_`	`U`	고정 길이 유니코드형(플랫폼에 따라 문자별 바이트 수가 다르다). `string_`형과 같은 형식을 쓴다(예: `U10`).

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

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

위 예제에서는 정수형을 부동소수점으로 변환했다. 만약 부동소수점수를 정수형 dtype으로 변환하면 소수점 아래 자리는 버려진다.

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [42]: arr
Out[42]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

숫자 형식의 문자열을 담고 있는 배열이 있다면 astype을 사용하여 숫자로 변환할 수 있다.

```
In [44]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [45]: numeric_strings.astype(float)
Out[45]: array([ 1.25, -9.6 , 42. ])
```

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 숫자 형식이 아닌 경우에 좀 더 직관적인 사용성을 제공한다.

만일 어떤 이유(문자열이 float64형으로 변환되지 않는 경우와 같은)로 형변환이 실패하면 ValueError 예외가 발생한다. 위 예에서 나는 귀찮아서 np.float64 대신 그냥 float라고 입력했는데 똑똑한 NumPy는 파이썬 자료형을 알맞은 dtype으로 맞춰준다.

다른 배열의 dtype 속성을 이용하는 것도 가능하다.

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

dtype으로 사용할 수 있는 축약 코드도 있다.

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:
array([         0, 1075314688,          0, 1075707904,          0,
       1075838976,          0, 1072693248], dtype=uint32)
```

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

4.1.3 NumPy 배열의 산술 연산

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

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

```
In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

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

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

같은 크기를 가지는 배열 간의 비교 연산은 불리언 배열을 반환한다.

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

In [58]: arr2
Out[58]:
array([[  0.,   4.,   1.],
       [  7.,   2.,  12.]])

In [59]: arr2 > arr
Out[59]:
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

크기가 다른 배열 간의 연산은 **브로드캐스팅**broadcasting이라고 하는데 12장에서 자세히 다루도록 하겠다. 이 책을 이해하기 위해 브로드캐스팅을 깊이 알고 있을 필요는 없다.

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 [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

위에서 볼 수 있듯이 arr[5:8] = 12처럼 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 arr 배열의 슬라이스를 생성해보자.

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

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

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
  9])
```

단순히 [:]로 슬라이스를 하면 배열의 모든 값을 할당한다.

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: 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 설계되었기 때문에 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 아니라 1차원 배열이다.

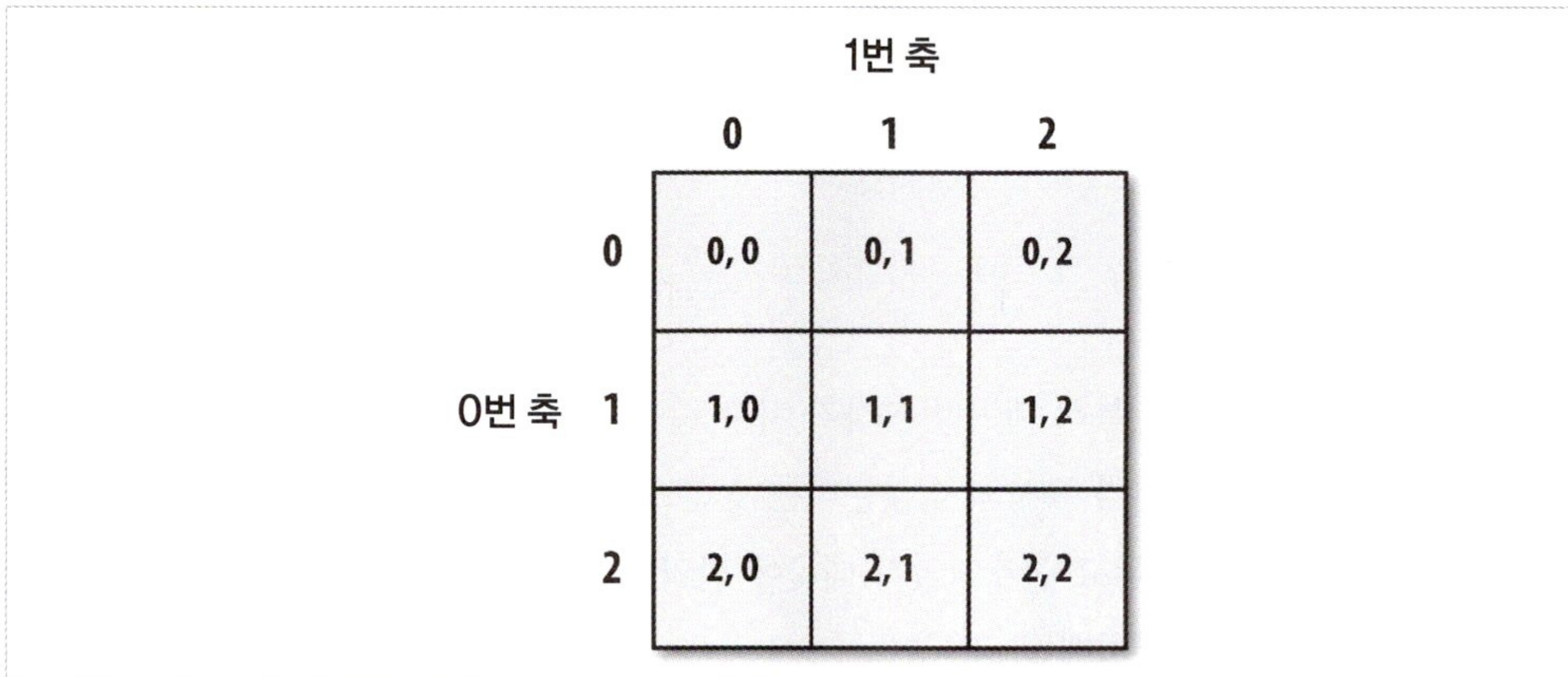
```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```

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

```
In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

[그림 4-1]에 2차원 배열에 대한 색인을 나타냈다. 0번 축을 '로우'로 생각하고 1번 축을 '컬럼'으로 생각하면 이해하기 쉽다.

그림 4-1 NumPy 배열에서 원소 색인하기



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

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

arr3d[0]은 2×3 크기의 배열이다.

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

arr3d[0]에는 스칼라값과 배열 모두 대입할 수 있다.

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42
```

```
In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values

In [83]: arr3d
Out[83]:
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 [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

이 표현은 다음처럼 두 번에 걸쳐 인덱싱한 결과와 동일하다.

```
In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]:
array([7, 8, 9])
```

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

슬라이스로 선택하기

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

```
In [88]: arr
Out[88]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

In [89]: arr[1:6]
Out[89]: array([ 1,  2,  3,  4, 64])
```

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

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
       [4, 5, 6]])
```

확인한 것처럼 첫 번째 축인 0번 축을 기준으로 슬라이싱되었다. 따라서 슬라이스는 축을 따라 선택 영역 내의 요소를 선택한다. arr2d[:2]는 'arr2d의 시작부터 두 번째 로우까지의 선택'이라고 이해하면 쉽다.

색인을 여러 개 넘겨서 다차원을 슬라이싱하는 것도 가능하다.

```
In [92]: arr2d[:2, 1:]
Out[92]:
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 [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

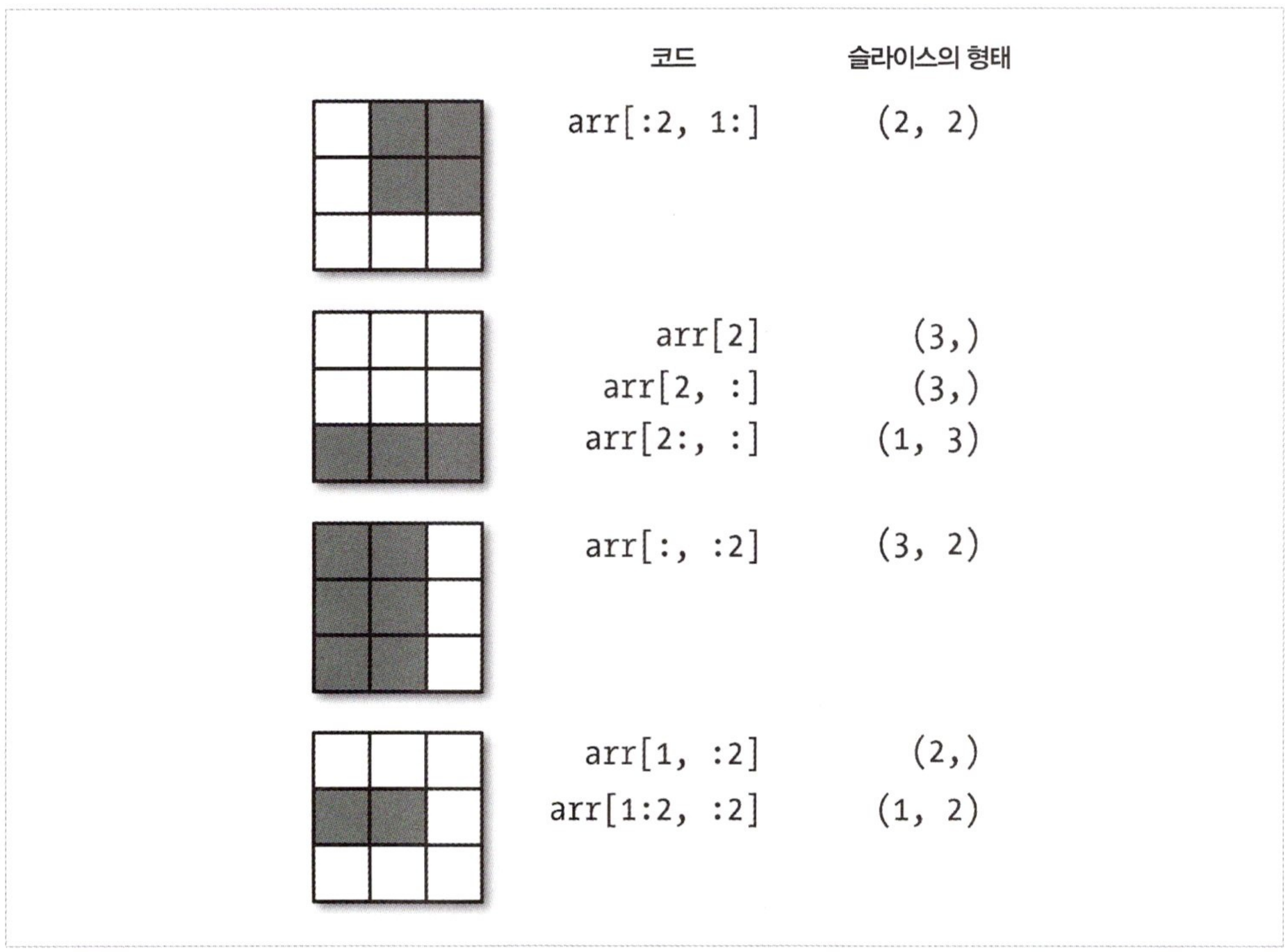
이와 유사하게 처음 두 로우에서 세 번째 컬럼만 선택하고 싶다면 아래처럼 하면 된다.

```
In [94]: arr2d[:2, 2]
Out[94]: array([3, 6])
```

[그림 4-2]를 보자. 그냥 콜론만 쓰면 전체 축을 선택한다는 의미이므로 이렇게 하면 하면 원래 차원의 슬라이스를 얻게 된다.

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```

그림 4-2 2차원 배열 슬라이싱



물론 슬라이싱 구문에 값을 대입하면 선택 영역 전체에 값이 대입된다.

```
In [96]: arr2d[:2, 1:] = 0

In [97]: arr2d
Out[97]:
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 정규 분포 데이터를 생성하자.

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [99]: data = np.random.randn(7, 4)

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

In [101]: data
Out[101]:
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]])
```

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 [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

이 불리언 배열은 배열의 색인으로 사용할 수 있다.

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

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

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

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

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

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

```
In [106]: names != 'Bob'
Out[106]: array([False,  True,  True, False,  True,  True,  True], dtype=bool)

In [107]: data[~(names == 'Bob')]
Out[107]:
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]])
```

~ 연산자는 일반적인 조건을 반대로 쓰고 싶을 때 유용하다.

```
In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
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]])
```

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

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, False, False], dtype=bool)

In [112]: data[mask]
Out[112]:
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]])
```

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 이루어진다. data에 저장된 모든 음수를 0으로 대입하려면 다음과 같이 하면 된다.

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
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.    ]])
```

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

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
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.    ]])
```

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

4.1.6 팬시 색인

팬시 색인fancy indexing은 정수 배열을 사용한 색인을 설명하기 위해 NumPy에서 차용한 단어다. 8×4 크기의 배열이 있다고 하자.

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
    .....:     arr[i] = i
```

```
In [119]: arr
Out[119]:
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 [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

다음 코드가 독자의 예상대로 실행되기 바란다! 색인으로 음수를 사용하면 끝에서부터 로우를 선택한다.

```
In [121]: arr[[-3, -5, -7]]
Out[121]:
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 [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
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 [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

reshape 메서드는 부록 A에서 더 자세히 살펴본다.

여기서 결과를 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(컬럼)에 대응하는 사각형 모양의 값이 선택되기를 기대했는데 그렇게 하려면 아래처럼 하면 된다.

```
In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
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. ndarray는 transpose 메서드와 T라는 이름의 특수한 속성을 가지고 있다.

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
```

```
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
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 [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
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]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622, 4.3437]])
```

다차원 배열의 경우 transpose 메서드는 튜플로 축 번호를 받아서 치환한다.

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
```

```
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
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 [135]: arr
Out[135]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
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.

4.2 유니버설 함수: 배열의 각 원소를 빠르게 처리하는 함수

ufunc라고 불리기도 하는 유니버설 함수는 ndarray 안에 있는 데이터 원소별로 연산을 수행하는 함수다. 유니버설 함수는 하나 이상의 스칼라값을 받아서 하나 이상의 스칼라 결괏값을 반환하는 간단한 함수를 고속으로 수행할 수 있는 벡터화된 래퍼 함수라고 생각하면 된다.

많은 ufunc는 sqrt나 exp 같은 간단한 변형을 전체 원소에 적용할 수 있다.

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
Out[140]:
array([    1.    ,     2.7183,     7.3891,    20.0855,    54.5982,
         148.4132,   403.4288,  1096.6332,  2980.958 ,  8103.0839])
```

위 함수들은 **단항** 유니버설 함수라고 한다. add나 maximum처럼 2개의 인자를 취해서 단일 배열을 반환하는 함수는 **이항** 유니버설 함수라고 한다.

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  0.0222,  0.7584,
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -0.9559, -0.0235,
       -2.3042])
```

```
In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  0.0222,  0.7584,
       -0.6605])
```

여기서 numpy.maximum은 x와 y의 원소별로 가장 큰 값을 계산한다.

흔하지는 않지만 여러 개의 배열을 반환하는 유니버설 함수도 있다. modf는 파이썬 내장 함수인 divmod의 벡터화 버전인데, 분수를 받아서 몫과 나머지를 함께 반환한다.

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45  ,  0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

유니버설 함수는 선택적으로 out 인자를 취해 계산 결과를 따로 저장할 수도 있다.

```
In [151]: arr
Out[151]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])

In [152]: np.sqrt(arr)
Out[152]: array([    nan,    nan,    nan,  2.318 ,  1.9022,  1.8574,  2.2378])

In [153]: np.sqrt(arr, arr)
Out[153]: array([    nan,    nan,    nan,  2.318 ,  1.9022,  1.8574,  2.2378])

In [154]: arr
Out[154]: array([    nan,    nan,    nan,  2.318 ,  1.9022,  1.8574,  2.2378])
```

[표 4-3]과 [표 4-4]에 사용 가능한 유니버설 함수를 나열해두었다.

표 4-3 단항 유니버설 함수

함수	설명
abs, fabs	각 원소(정수, 부동소수점수, 복소수)의 절댓값을 구한다. 복소수가 아닌 경우에는 빠른 연산을 위해서 fabs를 사용한다.
sqrt	각 원소의 제곱근을 계산한다. arr ** 0.5와 동일하다.
square	각 원소의 제곱을 계산한다. arr ** 2와 동일하다.
exp	각 원소에서 지수 e^x을 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)
ceil	각 원소의 소수자리를 올린다. 각 원소의 값보다 같거나 큰 정수 중 가장 작은 정수를 반환한다.
floor	각 원소의 소수자리를 내린다. 각 원소의 값보다 작거나 같은 정수 중 가장 작은 수를 반환한다.
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(non-inf, non-NaN) 무한한지 나타내는 불리언 배열을 반환한다.
cos, cosh, sin, sinh, tan, tanh	일반 삼각함수와 쌍곡삼각함수
arccos, arccosh, arcsin, arcsinh, arctan, arctanh	역삼각함수
logical_not	각 원소의 논리 부정(not) 값을 계산한다. ~arr과 동일하다.

표 4-4 이항 유니버설 함수

함수	설명
add	두 배열에서 같은 위치의 원소끼리 더한다.
subtract	첫 번째 배열의 원소에서 두 번째 배열의 원소를 뺀다.
multiply	배열의 원소끼리 곱한다.
divide, floor_divide	첫 번째 배열의 원소를 두 번째 배열의 원소로 나눈다. floor_divide는 몫만 취한다.
power	첫 번째 배열의 원소를 두 번째 배열의 원소만큼 제곱한다.
maximum, fmax	각 배열의 두 원소 중 큰 값을 반환한다. fmax는 NaN을 무시한다.
minimum, fmin	각 배열의 두 원소 중 작은 값을 반환한다. fmin은 NaN을 무시한다.

함수	설명
mod	첫 번째 배열의 원소를 두 번째 배열의 원소로 나눈 나머지를 구한다.
copysign	첫 번째 배열의 원소의 기호를 두 번째 배열의 원소의 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 사용해 많은 종류의 데이터 처리 작업을 할 수 있다. 배열 연산을 사용해서 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 계산하는 한다고 하자. np.meshgrid 함수는 두 개의 1차원 배열을 받아서 가능한 모든 (x, y) 짝을 만들 수 있는 2차원 배열 두 개를 반환한다.

```
In [155]: points = np.arange(-5, 5, 0.01) # -5부터 4.99까지 0.01씩 증가하는 값들의 배열

In [156]: xs, ys = np.meshgrid(points, points)

In [157]: ys
Out[157]:
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]])
```

이제 그리드 상의 두 포인트로 간단하게 계산을 적용할 수 있다.

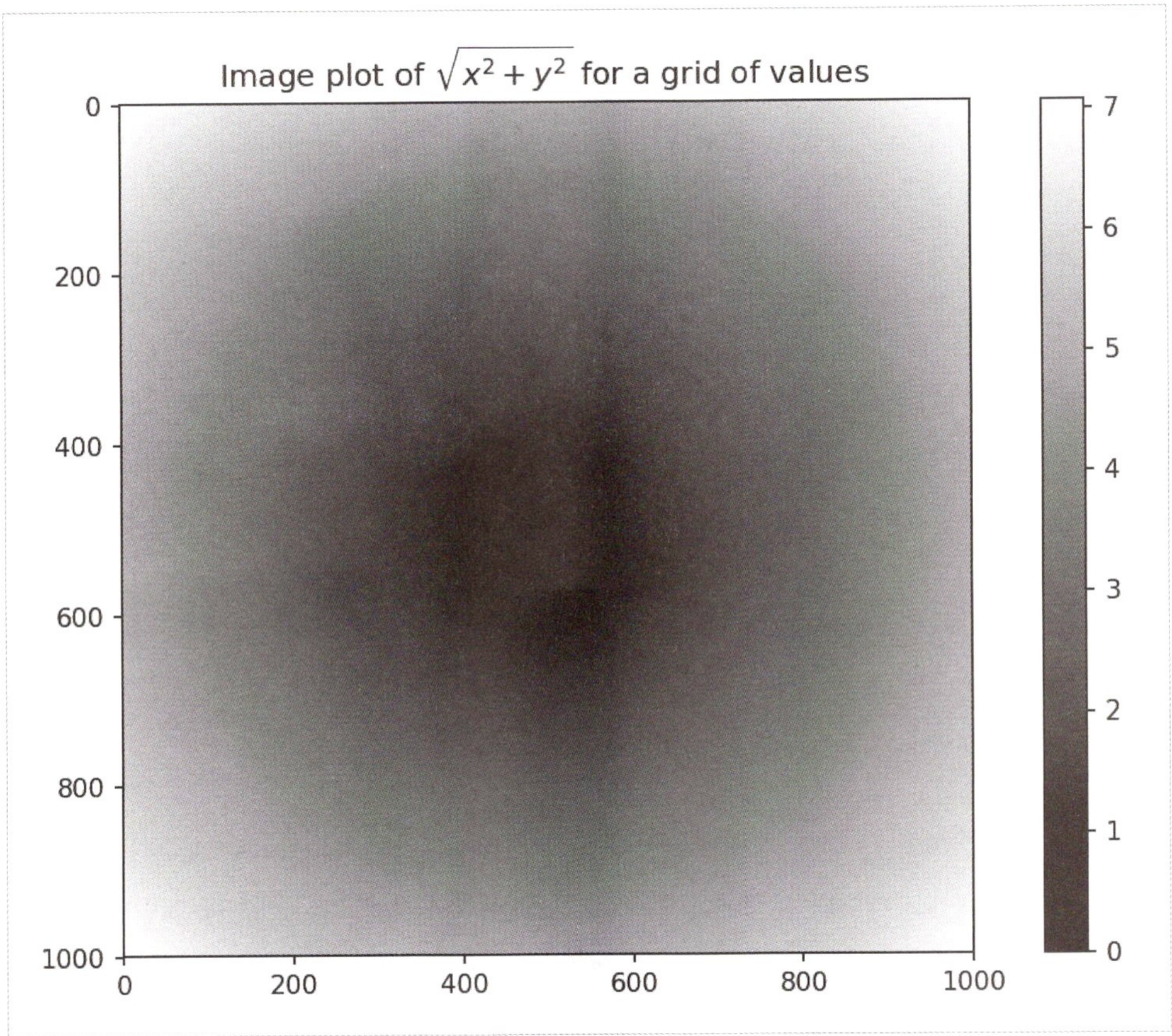
```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
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]])
```

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

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[161]: <matplotlib.colorbar.Colorbar at 0x7f715e3fa630>

In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```

[그림 4-3]을 보자. 계산된 값이 들어 있는 2차원 배열로부터 그래프 이미지를 생성하기 위해 matplotlib의 imshow 함수를 사용했다.

그림 4-3 그리드에 적용한 함수의 그래프



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

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, False])
```

cond의 값이 True일 때는 xarr의 값을 취하고 아니면 yarr의 값을 취하고 싶다면 리스트 표기법[list comprehension]을 이용해서 다음처럼 작성할 수 있다.

```
In [168]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, cond)]

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

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

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

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

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
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]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)
```

```
In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

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

```
In [176]: np.where(arr > 0, 2, arr) # 양수인 경우에만 2를 대입한다.
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  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 [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])
```

```
In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

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

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

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

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

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

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

[표 4-5]에서 지원하는 모든 함수를 확인할 수 있다. 이들 메서드에 대한 다양한 예제는 다음 장에서 살펴볼 것이다.

표 4-5 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)와 분산(var)을 구한다. 선택적으로 자유도를 줄 수 있으며 분모의 기본값은 n이다.
min, max	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 sum 메서드를 실행하면 불리언 배열에서 True인 원소의 개수를 셀 수 있다.

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # 양수인 원소의 개수
Out[191]: 42
```

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

```
In [192]: bools = np.array([False, False, True, False])

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: 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 [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938,  1.24  , -0.1357,  1.43  , -0.8469])

In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,  1.24  ,  1.43  ])
```

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

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])
```

```
In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

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 [203]: large_arr = np.random.randn(1000)

In [204]: large_arr.sort()

In [205]: large_arr[int(0.05 * len(large_arr))] # 5% 분위수
Out[205]: -1.5311513550102103
```

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 집합 관련 함수

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.unique일 것이다.

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [207]: np.unique(names)
Out[207]:
```

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

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

np.unique를 순수 파이썬만으로 구현하면 다음과 같다.

```
In [210]: sorted(set(names))
Out[210]: ['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 [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
Out[212]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

[표 4-6]에 NumPy에서 제공하는 집합 함수를 나열해두었다.

표 4-6 배열 집합 연산

메서드	설명
unique(x)	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 대칭차집합을 반환한다.

4.4 배열 데이터의 파일 입출력

NumPy는 디스크에서 텍스트나 바이너리 형식의 데이터를 불러오거나 저장할 수 있다. 여기서는 NumPy의 내장 이진 형식만 살펴본다. 많은 사람이 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 함수다. 배열은 기본적으로 압축되지 않은 원시(가공되지 않은) 바이너리 형식의 .npy 파일로 저장된다.

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
```

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

```
In [215]: np.load('some_array.npy')
Out[215]: 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 [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

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

```
In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

압축이 잘되는 형식의 데이터라면 대신 numpy.savez_compressed를 사용하자.

```
In [219]: np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

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 메서드이자 numpy 네임스페이스 안에 있는 dot 함수를 이용해서 계산한다.

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [226]: y
Out[226]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])

In [227]: x.dot(y)
Out[227]:
array([[ 28.,   64.],
       [ 67.,  181.]])
```

x.dot(y)는 np.dot(x, y)와 동일하다.

```
In [228]: np.dot(x, y)
Out[228]:
array([[ 28.,   64.],
       [ 67.,  181.]])
```

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

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([  6.,  15.])
```

파이썬 3.5부터 사용할 수 있는 @ 기호는 행렬 곱셈을 수행하는 연산자다.

```
In [230]: x @ np.ones(3)
Out[230]: array([  6.,  15.])
```

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

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)

In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[  933.1189,   871.8258, -1417.6902, -1460.4005,  1782.1391],
       [  871.8258,   815.3929, -1325.9965, -1365.9242,  1666.9347],
       [-1417.6902, -1325.9965,  2158.4424,  2222.0191, -2711.6822],
       [-1460.4005, -1365.9242,  2222.0191,  2289.0575, -2793.422 ],
       [ 1782.1391,  1666.9347, -2711.6822, -2793.422 ,  3409.5128]])

In [235]: mat.dot(inv(mat))
Out[235]:
array([[ 1.,  0., -0., -0., -0.],
       [-0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [-0.,  0.,  0.,  1., -0.],
       [-0.,  0.,  0.,  0.,  1.]])

In [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[-1.6914, 4.38 , 0.1757, 0.4075, -0.7838],
       [ 0.    , -2.6436,  0.1939, -3.072 , -1.0702],
       [ 0.    ,  0.    , -0.8138,  1.5414,  0.6155],
       [ 0.    ,  0.    ,  0.    , -2.6445, -2.1669],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.0002]])
```

X.T.dot(X)는 X.T.의 전치행렬과 X의 곱을 계산한다.

[표 4-7]에 자주 사용하는 선형대수 함수를 나열해두었다.

표 4-7 자주 사용하는 numpy.linalg 함수

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.
pinv	정사각 행렬의 무어-펜로즈 유사역원 역행렬을 구한다.
qr	QR 분해를 계산한다.
svd	특잇값 분해(SVD)를 계산한다.
solve	A가 정사각 행렬일 때 Ax = b를 만족하는 x를 구한다.
lstsq	Ax = b를 만족하는 최소제곱해를 구한다.

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 표준정규분포로부터 4×4 크기의 표본을 생성할 수 있다.

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

이에 비해 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 [240]: from random import normalvariate

In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _ in range(N)]
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

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

```
In [244]: 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 [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  0.8872,  0.8596,
       -0.6365,  0.0157, -2.2427])
```

[표 4-8]은 numpy.random에 포함된 일부 함수다. 다음 절에서 큰 표본을 생성하기 위해 이 함수들의 기능을 이용하는 예를 살펴보겠다.

표 4-8 일부 numpy.random 함수

함수	설명
seed	난수 생성기의 시드를 지정한다.
permutation	순서를 임의로 바꾸거나 임의의 순열을 반환한다.
shuffle	리스트나 배열의 순서를 뒤섞는다.
rand	균등분포에서 표본을 추출한다.

함수	설명
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 애플리케이션이다. 계단 중간에서 같은 확률로 한 계단 올라가거나 내려간다고 가정하자.

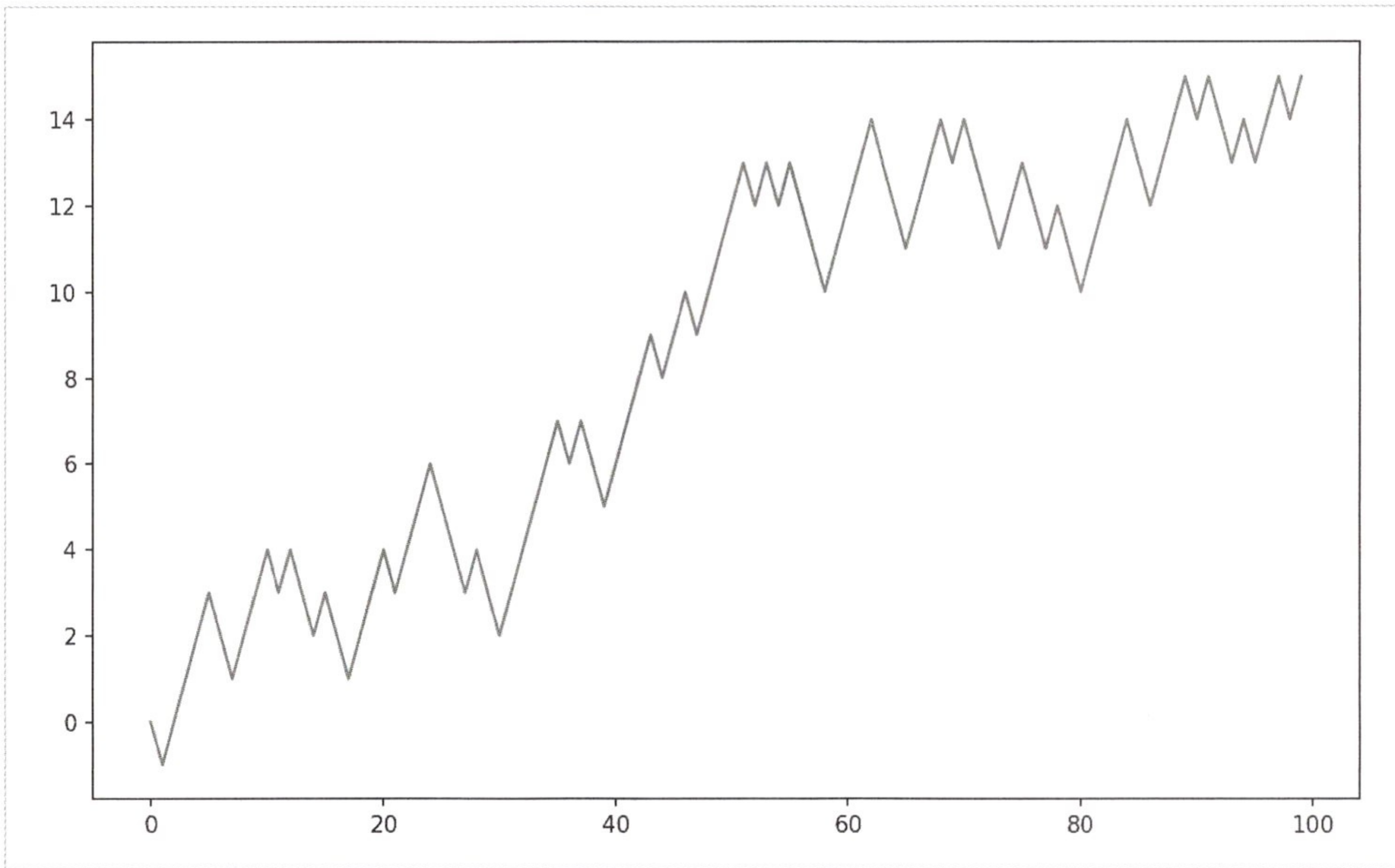
순수 파이썬으로 내장 random 모듈을 사용하여 계단 오르내리기를 1,000번 수행하는 코드는 다음처럼 작성할 수 있다.

```
In [247]: 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)
    .....:
```

[그림 4-4]는 처음 100회의 계단 오르내리기를 그래프로 나타낸 것이다.

```
In [249]: plt.plot(walk[:100])
```

그림 4-4 계단 오르내리기



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

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps)

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()
```

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

```
In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

계단에서 특정 위치에 도달하기까지의 시간 같은 좀 더 복잡한 통계를 구할 수 있는데, 계단의 처음 위치에서 최초로 10칸 떨어지기까지 얼마나 걸렸는지 확인해보자. `np.abs(walk) >= 10`으로 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 `argmax`를 사용하자(`True`가 최댓값이다).

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```

여기서 `argmax`를 사용했지만 `argmax`는 배열 전체를 모두 확인하기 때문에 효과적인 방법은 아니다. 또한 이 예제에서는 `True`가 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 방에 처리할 수 있다.

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 또는 1

In [261]: steps = np.where(draws > 0, 1, -1)

In [262]: walks = steps.cumsum(1)

In [263]: walks
Out[263]:
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]])
```

이제 모든 시뮬레이션에 대해 최댓값과 최솟값을 구해보자.

```
In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```

이 데이터에서 누적합이 30 혹은 −30이 되는 최소 시점을 계산해보자. 5,000회의 시뮬레이션 중 모든 경우가 30에 도달하지 않기 때문에 약간 까다로운데, any 메서드를 이용해서 해결할 수 있다.

```
In [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False,  True, False, ..., False,  True, False], dtype=bool)

In [268]: hits30.sum() # 누적합이 30 또는 -30이 되는 경우의 수
```

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

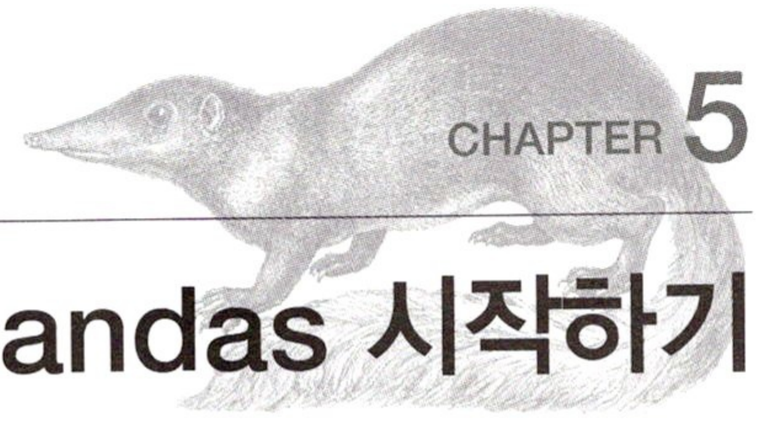
```
In [269]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [270]: crossing_times.mean()
Out[270]: 498.88973607038122
```

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

4.8 마치며

이 책에서는 데이터를 다루기 위해 주로 pandas를 사용하지만 배열 기반 방식도 쭉 살펴볼 것이다. 부록 A에서는 배열 계산 실력을 향상시킬 수 있는 NumPy의 자세한 기능을 더 살펴본다.



CHAPTER 5

pandas 시작하기

pandas는 앞으로 가장 자주 살펴볼 라이브러리다. 고수준의 자료구조와 파이썬에서 빠르고 쉽게 사용할 수 있는 데이터 분석 도구를 포함하고 있다. pandas는 다른 산술 계산 도구인 NumPy와 SciPy, 분석 라이브러리인 statsmodels와 scikit-learn, 시각화 도구인 matplotlib과 함께 사용하는 경우가 흔하다. pandas는 for 문을 사용하지 않고 데이터를 처리한다거나 배열 기반의 함수를 제공하는 등 NumPy의 배열 기반 계산 스타일을 많이 차용했다.

pandas가 NumPy의 스타일을 많이 차용했지만 가장 큰 차이점은 pandas는 표 형식의 데이터나 다양한 형태의 데이터를 다루는 데 초점을 맞춰 설계했다는 것이다. NumPy는 단일 산술 배열 데이터를 다루는 데 특화되어 있다.

2010년에 오픈소스로 공개한 이후 pandas는 다양한 현실의 요구 사항을 모두 수용할 수 있는 매우 큰 라이브러리가 되었다. 800명이 넘는 기여자가 자신이 매일 풀고 있는 데이터 문제를 더 편리하게 처리하기 위해 직접 pandas 프로젝트에 참여하고 있다.

이 책에서는 앞으로 pandas의 import 컨벤션을 다음과 같이 사용하겠다.

```
In [1]: import pandas as pd
```

앞으로 코드에서 pd.를 보면 pandas를 지칭하는 것으로 이해하자. Series와 DataFrame은 로컬 네임스페이스로 임포트하는 것이 훨씬 편하므로 그렇게 사용하도록 하겠다.

```
In [2]: from pandas import Series, DataFrame
```

5.1 pandas 자료구조 소개

pandas에 대해 알아보려면 Series와 DataFrame, 이 두 가지 자료구조에 익숙해질 필요가 있다. 이 두 가지 자료구조로 모든 문제를 해결할 순 없지만 대부분의 애플리케이션에서 사용하기 쉬우며 탄탄한 기반을 제공한다.

5.1.1 Series

Series는 일련의 객체를 담을 수 있는 1차원 배열 같은 자료구조다(어떤 NumPy 자료형이라도 담을 수 있다). 그리고 **색인**index이라고 하는 배열의 데이터와 연관된 이름을 가지고 있다. 가장 간단한 Series 객체는 배열 데이터로부터 생성할 수 있다.

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series 객체의 문자열 표현은 왼쪽에 색인을 보여주고 오른쪽에 해당 색인의 값을 보여준다. 위 예제에서는 데이터의 색인을 지정하지 않았으니 기본 색인인 정수 `0`에서 `N - 1`(`N`은 데이터의 길이)까지의 숫자가 표시된다. Series의 배열과 색인 객체는 각각 `values`와 `index` 속성을 통해 얻을 수 있다.

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index # range(4)와 같다.
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

각각의 데이터를 지칭하는 색인을 지정하여 Series 객체를 생성해야 할 때는 다음처럼 한다.

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d 4
b 7
a -5
c 3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

NumPy 배열과 비교하자면, 단일 값을 선택하거나 여러 값을 선택할 때 색인으로 라벨을 사용할 수 있다.

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

여기서 ['c', 'a', 'd']는 (정수가 아니라 문자열이 포함되어 있지만) 색인의 배열로 해석된다.

불리언 배열을 사용해서 값을 걸러 내거나 산술 곱셈을 수행하거나 또는 수학 함수를 적용하는 등 NumPy 배열 연산을 수행해도 색인-값 연결이 유지된다.

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64
```

```
In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
c     6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Series를 이해하는 다른 방법은 고정 길이의 정렬된 사전형이라고 생각하는 것이다. Series는 색인값에 데이터값을 매핑하고 있으므로 파이썬의 사전형과 비슷하다. Series 객체는 파이썬의 사전형을 인자로 받아야 하는 많은 함수에서 사전형을 대체하여 사용할 수 있다.

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

파이썬 사전형에 데이터를 저장해야 한다면 파이썬 사전 객체로부터 Series 객체를 생성할 수도 있다.

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

사전 객체만 가지고 Series 객체를 생성하면 생성된 Series 객체의 색인에는 사전의 키값이 순서대로 들어간다. 색인을 직접 지정하고 싶다면 원하는 순서대로 색인을 직접 넘겨줄 수도 있다.

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

이 예제를 보면 sdata에 있는 값 중 3개만 확인할 수 있는데, 'California'에 대한 값은 찾을 수 없기 때문이다. 이 값은 NaN[not a number]으로 표시되고 pandas에서는 누락된 값, 혹은 NA 값으로 취급된다. 'Utah'는 states에 포함되어 있지 않으므로 실행 결과에서 빠지게 된다.

나는 앞으로 '누락된' 또는 'NA'를 누락된 데이터를 지칭하는 데 사용하도록 하겠다. pandas의 isnull과 notnull 함수는 누락된 데이터를 찾을 때 사용된다.

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

이 메서드는 Series의 인스턴스 메서드로도 존재한다.

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

누락된 데이터를 처리하는 방법은 7장에서 좀 더 자세히 살펴보기로 하자.

Series의 유용한 기능은 산술 연산에서 색인과 라벨로 자동 정렬하는 것이다.

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [37]: obj3 + obj4
Out[37]:
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

데이터 정렬에 대한 내용은 나중에 좀 더 살펴보자. 데이터베이스를 사용해본 경험이 있다면 join 연산과 비슷하다고 여겨질 것이다.

Series 객체와 Series의 색인은 모두 name 속성이 있는데 이 속성은 pandas의 핵심 기능과 밀접한 관련이 있다.

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

Series의 색인은 대입하여 변경할 수 있다.

```
In [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

5.1.2 DataFrame

DataFrame은 표 같은 스프레드시트 형식의 자료구조이고 여러 개의 컬럼이 있는데 각 컬럼

은 서로 다른 종류의 값(숫자, 문자열, 불리언 등)을 담을 수 있다. DataFrame은 로우와 컬럼에 대한 색인을 가지고 있는데, 색인의 모양이 같은 Series 객체를 담고 있는 파이썬 사전으로 생각하면 편하다. 내부적으로 데이터는 리스트나 사전 또는 1차원 배열을 담고 있는 다른 컬렉션이 아니라 하나 이상의 2차원 배열에 저장된다. 구체적인 DataFrame의 내부 구조는 이 책에서 다루는 내용에서 한참 벗어나므로 생략하겠다.

NOTE_ 물리적으로 DataFrame은 2차원이지만 계층적 색인을 이용해서 좀 더 고차원의 데이터를 표현할 수 있으며 이를 포함하여 pandas에서 데이터를 다루는 고급 기법은 8장에서 더 설명하겠다.

DataFrame 객체는 다양한 방법으로 생성할 수 있지만 가장 흔하게 사용되는 방법은 같은 길이의 리스트에 담긴 사전을 이용하거나 NumPy 배열을 이용하는 것이다.

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

만들어진 DataFrame의 색인은 Series와 같은 방식으로 자동으로 대입되며 컬럼은 정렬되어 저장된다.

```
In [45]: frame
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

주피터 노트북을 사용한다면 DataFrame 객체는 브라우저에서 좀 더 보기 편하도록 HTML 표 형식으로 출력될 것이다.

큰 DataFrame을 다룰 때는 head 메서드를 이용해서 처음 5개의 로우만 출력할 수도 있다.

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

원하는 순서대로 columns를 지정하면 원하는 순서를 가진 DataFrame 객체가 생성된다.

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

Series와 마찬가지로 사전에 없는 값을 넘기면 결측치로 저장된다.

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])

In [49]: frame2
Out[49]:
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

DataFrame의 컬럼은 Series처럼 사전 형식의 표기법으로 접근하거나 속성 형식으로 접근할 수 있다.

```
In [51]: frame2['state']
Out[51]:
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

NOTE_ 편의를 위해 IPython에서는 `frame2.year`처럼 속성에 접근하듯 사용하거나 탭을 이용한 자동완성 기능을 제공한다.

`frame2[column]` 형태로 사용하는 것은 어떤 컬럼이든 가능하지만 `frame2.column` 형태로 사용하는 것은 파이썬에서 사용 가능한 변수 이름 형식일 때만 작동한다.

반환된 Series 객체가 DataFrame과 같은 색인을 가지면 알맞은 값으로 name 속성이 채워진다. 로우는 위치나 loc 속성을 이용해서 이름을 통해 접근할 수 있다.

```
In [53]: frame2.loc['three']
Out[53]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

컬럼은 대입이 가능하다. 예를 들어 현재 비어 있는 'debt' 컬럼에 스칼라값이나 배열의 값을 대입할 수 있다.

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

리스트나 배열을 컬럼에 대입할 때는 대입하려는 값의 길이가 DataFrame의 크기와 동일해야 한다. Series를 대입하면 DataFrame의 색인에 따라 값이 대입되며 존재하지 않는 색인에는 결측치가 대입된다.

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
```

```
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

존재하지 않는 컬럼을 대입하면 새로운 컬럼을 생성한다. 파이썬 사전형에서처럼 del 예약어를 사용해서 컬럼을 삭제할 수 있다.

del 예약어에 대한 예제로, state 컬럼의 값이 'Ohio'인지 아닌지에 대한 불리언값을 담고 있는 새로운 컬럼을 생성해보자.

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

CAUTION_ 이 새로운 컬럼은 frame2.eastern 형태의 문법으로는 생성되지 않는다.

del 예약어를 이용해서 이 컬럼을 삭제할 수 있다.

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

CAUTION_ DataFrame의 색인을 이용해서 얻은 컬럼은 내부 데이터에 대한 **뷰**(view)이며 복사가 이루어지지 않는다. 따라서 이렇게 얻은 Series 객체에 대한 변경은 실제 DataFrame에 반영된다. 복사본이 필요할 때는 Series의 copy 메서드를 이용하자.

중첩된 사전을 이용해서 데이터를 생성할 수 있다. 다음과 같은 중첩된 사전이 있다고 하자.

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

이 중첩된 사전을 DataFrame에 넘기면 바깥에 있는 사전의 키는 컬럼이 되고 안에 있는 키는 로우가 된다.

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

NumPy 배열과 유사한 문법으로 데이터를 전치(컬럼과 로우를 뒤집음)할 수 있다.

```
In [68]: frame3.T
Out[68]:
        2000  2001  2002
Nevada   NaN   2.4   2.9
Ohio     1.5   1.7   3.6
```

중첩된 사전을 이용해서 DataFrame을 생성할 때 안쪽에 있는 사전값은 키값별로 조합되어 결과의 색인이 되지만 색인을 직접 지정하면 지정된 색인으로 DataFrame을 생성한다.

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

Series 객체를 담고 있는 사전 데이터도 같은 방식으로 취급된다.

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
```

DataFrame 생성자에 넘길 수 있는 자료형의 목록은 [표 5-1]을 참조하자.

만일 데이터프레임의 색인(index)과 컬럼(columns)에 name 속성을 지정했다면 이 역시 함께 출력된다.

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

Series와 유사하게 values 속성은 DataFrame에 저장된 데이터를 2차원 배열로 반환한다.

```
In [74]: frame3.values
Out[74]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

DataFrame의 컬럼이 서로 다른 dtype을 가지고 있다면 모든 컬럼을 수용하기 위해 그 컬럼의 배열의 dtype이 선택된다.

```
In [75]: frame2.values
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
```

```
        [2002, 'Ohio', 3.6, nan],
        [2001, 'Nevada', 2.4, -1.5],
        [2002, 'Nevada', 2.9, -1.7],
        [2003, 'Nevada', 3.2, nan]], dtype=object)
```

표 5-1 DataFrame 생성을 위한 입력 데이터의 종류

형	설명
2차원 ndarray	데이터를 담고 있는 행렬. 선택적으로 행(로우)과 열(컬럼)의 이름을 전달할 수 있다.
배열, 리스트, 튜플의 사전	사전의 모든 항목은 같은 길이를 가져야 하며, 각 항목의 내용이 DataFrame의 컬럼이 된다.
NumPy의 구조화 배열	배열의 사전과 같은 방식으로 취급된다.
Series의 사전	Series의 각 값이 컬럼이 된다. 명시적으로 색인을 넘겨주지 않으면 각 Series의 색인이 하나로 합쳐져서 로우의 색인이 된다.
사전의 사전	내부에 있는 사전이 컬럼이 된다. 키값은 'Series의 사전'과 마찬가지로 합쳐져서 로우의 색인이 된다.
사전이나 Series의 리스트	리스트의 각 항목이 DataFrame의 로우가 된다. 합쳐진 사전의 키값이나 Series의 색인이 DataFrame의 컬럼 이름이 된다.
리스트나 튜플의 리스트	'2차원 ndarray'의 경우와 같은 방식으로 취급된다.
다른 DataFrame	색인을 따로 지정하지 않으면 DataFrame의 색인이 그대로 사용된다.
NumPy MaskedArray	'2차원 ndarray'의 경우와 같은 방식으로 취급되지만 마스크값은 반환되는 DataFrame에서 NA 값이 된다.

5.1.3 색인 객체

pandas의 색인 객체는 표 형식의 데이터에서 각 로우와 컬럼에 대한 이름과 다른 메타데이터(축의 이름 등)를 저장하는 객체다. Series나 DataFrame 객체를 생성할 때 사용되는 배열이나 다른 순차적인 이름은 내부적으로 색인으로 변환된다.

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: Index(['b', 'c'], dtype='object')
```

색인 객체는 변경이 불가능하다.

```
index[1] = 'd' # TypeError 발생
```

그러므로 자료구조 사이에서 안전하게 공유될 수 있다.

```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

CAUTION_ 몇몇 독자는 색인이 제공하는 기능을 유용하게 사용할 일이 별로 없을 수 있지만 일부 연산의 경우 색인을 반환하기도 하므로 어떻게 동작하는지 이해하는 것은 매우 중요하다.

또한 배열과 유사하게 Index 객체도 고정 크기로 동작한다.

```
In [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

```
In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

파이썬의 집합과는 달리 pandas의 인덱스는 중복되는 값을 허용한다.

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

중복되는 값으로 선택을 하면 해당 값을 가진 모든 항목이 선택된다.

각각의 색인은 자신이 담고 있는 데이터에 대한 정보를 취급하기 위한 여러 가지 메서드와 속성을 가지고 있는데 [표 5-2]를 참조하자.

표 5-2 색인 메서드와 속성

메서드	설명
append	추가적인 색인 객체를 덧붙여 새로운 색인을 반환한다.
difference	색인의 차집합을 반환한다.
intersection	색인의 교집합을 반환한다.
union	색인의 합집합을 반환한다.
isin	색인이 넘겨받은 색인에 존재하는지 알려주는 불리언 배열을 반환한다.
delete	i 위치의 색인이 삭제된 새로운 색인을 반환한다.
drop	넘겨받은 값이 삭제된 새로운 색인을 반환한다.
insert	i 위치에 색인이 추가된 새로운 색인을 반환한다.
is_monotonic	색인이 단조성을 가진다면 True를 반환한다.
is_unique	중복되는 색인이 없다면 True를 반환한다.
unique	색인에서 중복되는 요소를 제거하고 유일한 값만 반환한다.

5.2 핵심 기능

이 절에서는 Series나 DataFrame에 저장된 데이터를 다루는 기본적인 방법을 설명하겠다. 다음 몇몇 장에서는 pandas를 이용한 데이터 분석과 조작을 더 자세히 살펴볼 것이다. 이 책은 pandas 라이브러리의 완전한 설명을 포함하지 않고 중요한 기능에만 초점을 맞추고 있다. 잘 쓰이지 않는 내용에 대한 학습은 독자의 몫으로 남겨둔다.

5.2.1 재색인

pandas 객체의 중요한 기능 중 하나는 reindex인데, 새로운 색인에 **맞도록** 객체를 새로 생성한다. 아래 간단한 예제를 살펴보자.

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [92]: obj
Out[92]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

이 Series 객체에 대해 reindex를 호출하면 데이터를 새로운 색인에 맞게 재배열하고, 존재하지 않는 색인값이 있다면 NaN을 추가한다.

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

시계열 같은 순차적인 데이터를 재색인할 때 값을 보간하거나 채워 넣어야 할 경우가 있다. method 옵션을 이용해서 이를 해결할 수 있으며, ffill 같은 메서드를 이용해서 누락된 값을 직전의 값으로 채워 넣을 수 있다.

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [96]: obj3
Out[96]:
0      blue
2    purple
4    yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

DataFrame에 대한 reindex는 로우(색인), 컬럼 또는 둘 다 변경 가능하다. 그냥 순서만 전달하면 로우가 재색인된다.

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'])

In [99]: frame
Out[99]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8

In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [101]: frame2
```

```
Out[101]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

컬럼은 columns 예약어를 사용해서 재색인할 수 있다.

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

reindex의 인자는 [표 5-3]을 참조하자.

곧 다루겠지만 재색인은 loc를 이용해서 라벨로 색인하면 좀 더 간결하게 할 수 있으며 대부분의 사용자는 명시적으로 이 방식을 사용하는 것을 선호한다.

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
   Texas  Utah  California
a    1.0   NaN         2.0
b    NaN   NaN         NaN
c    4.0   NaN         5.0
d    7.0   NaN         8.0
```

표 5-3 재색인 함수 인자

인자	설명
index	색인으로 사용할 새로운 순서. Index 인스턴스나 다른 순차적인 자료구조가 사용 가능하다. Index는 복사가 이루어지지 않고 그대로 사용된다.
method	채움 메서드. ffill은 직전 값을 채워 넣고 bfill은 다음 값을 채워 넣는다.
fill_value	재색인 과정 중에 새롭게 나타나는 비어 있는 데이터를 채우기 위한 값
limit	전/후 보간 시에 사용할 최대 갭 크기(채워넣을 원소의 수)

인자	설명
tolerance	전/후 보간 시에 사용할 최대 갭 크기(값의 차이)
level	MultiIndex의 단계(level)에 단순 색인을 맞춘다. 그렇지 않으면 MultiIndex의 하위집합에 맞춘다.
copy	True인 경우 새로운 색인이 이전 색인과 동일하더라도 데이터를 복사한다. False인 경우 새로운 색인이 이전 색인과 동일할 경우 복사하지 않는다.

5.2.2 하나의 로우나 컬럼 삭제하기

색인 배열, 또는 삭제하려는 로우나 컬럼이 제외된 리스트를 이미 가지고 있다면 로우나 컬럼을 쉽게 삭제할 수 있는데 이 방법은 데이터의 모양을 변경하는 작업이 필요하다. drop 메서드를 사용하면 선택한 값들이 삭제된 새로운 객체를 얻을 수 있다.

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

DataFrame에서는 로우와 컬럼 모두에서 값을 삭제할 수 있다. 다음 예제를 살펴보자.

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

drop 함수에 인자로 로우 이름을 넘기면 해당 로우(axis 0)의 값을 모두 삭제한다.

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

컬럼의 값을 삭제할 때는 axis=1 또는 axis='columns'를 인자로 넘겨주면 된다.

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

drop 함수처럼 Series나 DataFrame의 크기 또는 형태를 변경하는 함수는 새로운 객체를 반환하는 대신 원본 객체를 변경한다.

```
In [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

inplace 옵션을 사용하는 경우 버려지는 값을 모두 삭제하므로 주의해서 사용하자.

5.2.3 색인하기, 선택하기, 거르기

Series의 색인(obj[...])은 NumPy 배열의 색인과 유사하게 동작하지만 정수가 아니어도 된다는 점이 다르다. 몇 가지 예제를 살펴보자.

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
```

```
c    2.0
d    3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

라벨 이름으로 슬라이싱하면 시작점과 끝점을 포함한다는 것이 일반 파이썬에서의 슬라이싱과 다른 점이다.

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

슬라이싱 문법으로 선택된 영역에 값을 대입하는 것은 생각하는 대로 동작한다.

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
a    0.0
b    5.0
```

```
c    5.0
d    3.0
dtype: float64
```

색인으로 DataFrame에서 하나 이상의 컬럼 값을 가져올 수 있다.

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [130]: data['two']
Out[130]:
Ohio        1
Colorado    5
Utah        9
New York   13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York 14 12
```

슬라이싱으로 로우를 선택하거나 불리언 배열로 로우를 선택할 수 있다.

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

```
In [133]: data[data['three'] > 5]
Out[133]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

data[:2] 형식의 문법으로 편리하게 로우를 선택할 수 있다. [] 연산자에 단일 값을 넘기거나 리스트를 넘겨서 여러 컬럼을 선택할 수 있다.

또 다른 방법으로는 스칼라 비교를 이용해 생성된 불리언 DataFrame을 사용해서 값을 선택하는 것이다.

```
In [134]: data < 5
Out[134]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

위 예제는 DataFrame을 2차원 ndarray와 문법적으로 비슷하게 보이도록 의도한 것이다.

loc과 iloc으로 선택하기

DataFrame의 로우에 대해 라벨로 색인하는 방법으로 특수한 색인 필드인 loc과 iloc을 소개한다. 이 방법을 이용하면 NumPy와 비슷한 방식에 추가적으로 축의 라벨을 사용하여 DataFrame의 로우와 컬럼을 선택할 수 있다. 축 이름을 선택할 때는 loc을, 정수 색인으로 선택할 때는 iloc을 사용한다.

앞선 예제에서 축의 라벨로 하나의 로우와 여러 컬럼을 선택해보자.

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two      5
three    6
Name: Colorado, dtype: int64
```

iloc을 이용하면 정수 색인으로도 위와 비슷하게 선택할 수 있다.

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [139]: data.iloc[2]
Out[139]:
one       8
two       9
three    10
four     11
Name: Utah, dtype: int64

In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

이 두 함수는 슬라이스도 지원할 뿐더러 단일 라벨이나 라벨 리스트도 지원한다.

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64
```

```
In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

지금까지 살펴봤듯이 pandas 객체에서 데이터를 선택하고 재배열하는 방법은 여러 가지가 있다. [표 5-4]에 다양한 방법을 요약해두었다. 나중에 살펴볼 계층적 색인을 이용하면 좀 더 다양한 방법을 사용할 수 있다.

NOTE_ pandas를 설계할 때 나는 컬럼을 선택하는 작업은 매우 빈번하게 일어나지만 컬럼을 선택하기 위해 매번 `frame[:, col]`이라고 입력해야 하는 것은 너무 과하다고 생각했다(입력할 내용이 많으면 에러가 발생할 확률도 높아진다). 그래서 라벨과 정수 색인을 이용해서 값을 선택할 수 있는 기능을 모두 `ix` 연산자에 밀어 넣기로 결정했다. 하지만 실사용성 측면에서 정수축 라벨을 사용할 경우 모호한 상황이 발생하는 빈도가 높아졌고 결국 pandas 개발팀은 라벨과 정수 색인을 명시적으로 구분해서 사용하게끔 `loc`와 `iloc`를 추가하기로 결정했다.

`ix` 색인 연산자는 아직 사용 가능하긴 하지만 곧 사라질 예정이니 더는 사용하지 않기를 추천한다.

표 5-4 DataFrame의 값 선택하기

방식	설명
`df[val]`	DataFrame에서 하나의 컬럼 또는 여러 컬럼을 선택한다. 편의를 위해 불리언 배열, 슬라이스, 불리언 DataFrame(어떤 기준에 근거해서 값을 대입해야 할 때)을 사용할 수 있다.
`df.loc[val]`	DataFrame에서 라벨값으로 로우의 부분집합을 선택한다.
`df.loc[:, val]`	DataFrame에서 라벨값으로 컬럼의 부분집합을 선택한다.
`df.loc[val1, val2]`	DataFrame에서 라벨값으로 로우와 컬럼의 부분집합을 선택한다.
`df.iloc[where]`	DataFrame에서 정수 색인으로 로우의 부분집합을 선택한다.
`df.iloc[:, where]`	DataFrame에서 정수 색인으로 컬럼의 부분집합을 선택한다.
`df.iloc[where_i, wher_j]`	DataFrame에서 정수 색인으로 로우와 컬럼의 부분집합을 선택한다.
`df.at[label_i, label_j]`	로우와 컬럼의 라벨로 단일 값을 선택한다.
`df.iat[i, j]`	로우와 컬럼의 정수 색인으로 단일 값을 선택한다.
`reindex` 메서드	하나 이상의 축을 새로운 색인으로 맞춘다.
`get_value, set_value` 메서드	로우와 컬럼 이름으로 DataFrame의 값을 선택한다.

5.2.4 정수 색인

정수 색인으로 pandas 객체를 다루다보면 리스트나 튜플 같은 파이썬 내장 자료구조에서 색인을 다루는 방법과의 차이점 때문에 실수하게 되는 경우가 있다. 예를 들어 다음 코드가 에러를 발생할 것이라고 생각하지 않을 것이다.

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

이 경우 pandas는 라벨 색인을 찾는 데 실패하므로 정수 색인으로 값을 찾는다. 하지만 이를 어떤 경우에도 버그 없이 잘 작동하도록 구현하기란 쉽지 않다. 라벨 색인이 0, 1, 2를 포함하는 경우 사용자가 라벨 색인으로 선택하려는 것인지 정수 색인으로 선택하려는 것인지 추측하기 쉽지 않다.

```
In [144]: ser
Out[144]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

반면 정수 기반의 색인을 사용하지 않는 경우 이런 모호함은 사라진다.

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 2.0
```

일관성을 유지하기 위해 정숫값을 담고 있는 축 색인이 있다면 우선적으로 라벨을 먼저 찾아보도록 구현되어 있다. 좀 더 세밀하게 사용하고 싶다면 라벨에 대해서는 loc을 사용하고 정수 색인에 대해서는 iloc을 사용하자.

```
In [147]: ser[:1]
Out[147]:
0    0.0
```

```
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0    0.0
1    1.0
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0    0.0
dtype: float64
```

5.2.5 산술 연산과 데이터 정렬

pandas에서 가장 중요한 기능 중 하나는 다른 색인을 가지고 있는 객체 간의 산술 연산이다. 객체를 더할 때 짝이 맞지 않는 색인이 있다면 결과에 두 색인이 통합된다. 데이터베이스를 사용해본 경험이 있다면 색인 라벨에 대한 외부 조인과 유사하게 동작한다고 생각할 수 있다. 예제를 보자.

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [153]: s2
Out[153]:
a   -2.1
c    3.6
e   -1.5
f    4.0
```

```
g    3.1
dtype: float64
```

이 두 객체를 더하면 다음과 같은 결과를 얻는다.

```
In [154]: s1 + s2
Out[154]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

서로 겹치는 색인이 없는 경우 데이터는 NA 값이 된다. 산술 연산 시 누락된 값은 전파된다.

DataFrame의 경우 정렬은 로우와 컬럼 모두에 적용된다.

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                    index=['Ohio', 'Texas', 'Colorado'])

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [157]: df1
Out[157]:
           b    c    d
Ohio     0.0  1.0  2.0
Texas    3.0  4.0  5.0
Colorado 6.0  7.0  8.0

In [158]: df2
Out[158]:
         b     d     e
Utah   0.0   1.0   2.0
Ohio   3.0   4.0   5.0
Texas  6.0   7.0   8.0
Oregon 9.0  10.0  11.0
```

이 두 DataFrame을 더하면 각 DataFrame에 있는 색인과 컬럼이 하나로 합쳐진다.

```
In [159]: df1 + df2
Out[159]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

'c'와 'e' 컬럼이 양쪽 DataFrame 객체에 존재하지 않으므로 결과에서는 모두 없는 값으로 나타난다. 로우 역시 마찬가지로 양쪽에 다 존재하지 않는 라벨에 대해서는 없는 값으로 나타난다.

공통되는 컬럼 라벨이나 로우 라벨이 없는 DataFrame을 더하면 결과에 아무것도 나타나지 않을 것이다.

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0  1
1  2

In [163]: df2
Out[163]:
   B
0  3
1  4

In [164]: df1 - df2
Out[164]:
    A   B
0 NaN NaN
1 NaN NaN
```

산술 연산 메서드에 채워 넣을 값 지정하기

서로 다른 색인을 가지는 객체 간의 산술 연산에서 존재하지 않는 축의 값을 특수한 값(0 같은)으로 지정하고 싶을 때는 다음과 같이 할 수 있다.

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [169]: df2
Out[169]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

위 둘을 더하면 겹치지 않는 부분은 NA 값이 된다.

```
In [170]: df1 + df2
Out[170]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN
```

df1에 add 메서드를 사용하고, df2와 fill_value 값을 인자로 전달한다.

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

[표 5-5]에 Series와 DataFrame의 산술 연산 메서드를 정리했다. 각각의 산술 연산 메서드는 r로 시작하는 계산 인자를 뒤집어 계산하는 짝궁 메서드를 가진다.

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

In [173]: df1.rdiv(1)
Out[173]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

Series나 DataFrame을 재색인할 때도 fill_value를 지정할 수 있다.

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

표 5-5 산술 연산 메서드

메서드	설명
add, radd	덧셈(+)을 위한 메서드
sub, rsub	뺄셈(-)을 위한 메서드

메서드	설명
div, rdiv	나눗셈(/)을 위한 메서드
floordiv, rfloordiv	소수점 내림(//) 연산을 위한 메서드
mul, rmul	곱셈(*)을 위한 메서드
pow, rpow	멱승(**)을 위한 메서드

DataFrame과 Series 간의 연산

다른 차원의 NumPy 배열과의 연산처럼 DataFrame과 Series 간의 연산도 잘 정의되어 있다. 먼저 2차원 배열과 그 배열의 한 로우의 차이에 대해 생각할 수 있는 예제를 살펴보자.

```
In [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])

In [178]: arr - arr[0]
Out[178]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

arr에서 arr[0]을 빼면 계산은 각 로우에 대해 한 번씩만 수행된다. 이를 **브로드캐스팅**이라고 하는데 더 자세한 내용은 부록 A에서 살펴볼 것이다. DataFrame과 Series 간의 연산은 이와 유사하다.

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [180]: series = frame.iloc[0]
```

```
In [181]: frame
Out[181]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [182]: series
Out[182]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

기본적으로 DataFrame과 Series 간의 산술 연산은 Series의 색인을 DataFrame의 컬럼에 맞추고 아래 로우로 전파한다.

```
In [183]: frame - series
Out[183]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

만약 색인값을 DataFrame의 컬럼이나 Series의 색인에서 찾을 수 없다면 그 객체는 형식을 맞추기 위해 재색인된다.

```
In [184]: series2 = pd.Series(rnge(3), index=['b', 'e', 'f'])

In [185]: frame + series2
Out[185]:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
Ohio    3.0 NaN   6.0 NaN
Texas   6.0 NaN   9.0 NaN
Oregon  9.0 NaN  12.0 NaN
```

만약 각 로우에 대해 연산을 수행하고 싶다면 산술 연산 메서드를 사용하면 된다. 예를 들어보자.

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [188]: series3
Out[188]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

인자로 넘기는 axis 값은 **연산을 적용할 축 번호**다. `axis='index'`나 `axis=0`은 DataFrame의 로우를 따라 연산을 수행하라는 의미다.

5.2.6 함수 적용과 매핑

pandas 객체에도 NumPy의 유니버설 함수(배열의 각 원소에 적용되는 메서드)를 적용할 수 있다.

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [191]: frame
```

```
Out[191]:
                b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [192]: np.abs(frame)
Out[192]:
                b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

자주 사용되는 또 다른 연산은 각 컬럼이나 로우의 1차원 배열에 함수를 적용하는 것이다. DataFrame의 apply 메서드를 이용해 수행할 수 있다.

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

여기서 함수 f는 Series의 최댓값과 최솟값의 차이를 계산하는 함수다. frame의 각 컬럼에 대해 한 번만 수행되며 결괏값은 계산을 적용한 컬럼을 색인으로 하는 Series를 반환한다.

apply 함수에 axis='columns' 인자를 넘기면 각 로우에 대해 한 번씩만 수행된다.

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

배열에 대한 일반적인 통계(sum이나 mean 같은)는 DataFrame의 메서드로 존재하므로 apply 메서드를 사용할 필요 없다.

apply 메서드에 전달된 함수는 스칼라값을 반환할 필요가 없다. 여러 값을 가진 Series를 반환해도 된다.

```
In [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
Out[197]:
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

배열의 각 원소에 적용되는 파이썬의 함수를 사용할 수도 있다. frame 객체에서 실숫값을 문자열 포맷으로 변환하고 싶다면 applymap을 이용해서 다음과 같이 할 수 있다.

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
            b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

이 메서드의 이름이 applymap인 이유는 Series는 각 원소에 적용할 함수를 지정하기 위한 map 메서드를 가지고 있기 때문이다.

```
In [200]: frame['e'].map(format)
Out[200]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

5.2.7 정렬과 순위

어떤 기준에 근거해서 데이터를 정렬하는 것 역시 중요한 명령이다. 로우나 컬럼의 색인을 알파벳순으로 정렬하려면 정렬된 새로운 객체를 반환하는 sort_index 메서드를 사용하면 된다.

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a    1
b    2
c    3
d    0
dtype: int64
```

DataFrame은 로우나 컬럼 중 하나의 축을 기준으로 정렬할 수 있다.

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
Out[204]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [205]: frame.sort_index(axis=1)
Out[205]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

데이터는 기본적으로 오름차순으로 정렬되고 내림차순으로 정렬할 수도 있다.

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

Series 객체를 값에 따라 정렬하고 싶다면 sort_values 메서드를 사용하면 된다.

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2   -3
3    2
0    4
1    7
dtype: int64
```

정렬할 때 비어 있는 값은 기본적으로 Series 객체에서 가장 마지막에 위치한다.

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

DataFrame에서 하나 이상의 컬럼에 있는 값으로 정렬을 하는 경우 sort_values 함수의 by 옵션에 하나 이상의 컬럼 이름을 넘기면 된다.

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [212]: frame
Out[212]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [213]: frame.sort_values(by='b')
```

```
Out[213]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

여러 개의 컬럼을 정렬하려면 컬럼 이름이 담긴 리스트를 전달하면 된다.

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

순위는 정렬과 거의 흡사한데, 1부터 배열의 유효한 데이터 개수까지 순서를 매긴다. 기본적으로 Series와 DataFrame의 rank 메서드는 동점인 항목에 대해서는 평균 순위를 매긴다.

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

데이터 상에서 나타나는 순서에 따라 순위를 매길 수도 있다.

```
In [217]: obj.rank(method='first')
Out[217]:
0    6.0
```

```
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

여기서 0번째와 2번째 항목에 대해 평균 순위인 6.5를 적용하는 대신 먼저 출현한 순서대로 6과 7을 적용했다.

내림차순으로 순위를 매길 수도 있다.

```
# 동률인 경우 그룹 내에서 높은 순위를 적용한다.
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

[표 5-6]에 사용 가능한 동률 처리 메서드를 나열해두었다.

DataFrame에서는 로우나 컬럼에 대해 순위를 정할 수 있다.

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})

In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5
```

```
In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

표 5-6 순위의 동률을 처리하는 메서드

메서드	설명
'average'	기본값. 같은 값을 가지는 항목들의 평균값을 순위로 삼는다.
'min'	같은 값을 가지는 그룹을 낮은 순위로 매긴다.
'max'	같은 값을 가지는 그룹을 높은 순위로 매긴다.
'first'	데이터 내의 위치에 따라 순위를 매긴다.
'dense'	method='min'과 같지만 같은 그룹 내에서 모두 같은 순위를 적용하지 않고 1씩 증가시킨다.

5.2.8 중복 색인

지금까지 살펴본 모든 예제는 축 이름(색인값)이 유일한 경우밖에 없었다. pandas의 많은 함수(reindex 같은)에서 색인값은 유일해야 하지만 의무적이지는 않다. 중복된 색인값을 가지는 Series 객체를 살펴보자.

```
In [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

색인의 is_unique 속성은 해당 값이 유일한지 아닌지 알려준다.

```
In [224]: obj.index.is_unique
Out[224]: False
```

중복되는 색인값이 있다면 색인을 이용해서 데이터에 접근했을 때 다르게 동작한다. 중복되는 색인값이 없을 때는 색인을 이용해서 데이터에 접근하면 스칼라값을 반환하지만 중복되는 색인값이 있을 때는 하나의 Series 객체를 반환한다.

```
In [225]: obj['a']
Out[225]:
a    0
a    1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

이는 라벨이 반복되는지 여부에 따라 색인을 이용해서 선택한 결과가 다를 수 있기 때문에 코드를 좀 더 복잡하게 만들 수 있다.

DataFrame에서 로우를 선택하는 것도 동일하다.

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [228]: df
Out[228]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [229]: df.loc['b']
Out[229]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

5.3 기술 통계 계산과 요약

pandas 객체는 일반적인 수학 메서드와 통계 메서드를 가지고 있다. 이 메서드의 대부분은 하나의 Series나 DataFrame의 로우나 컬럼에서 단일 값(합이나 평균 같은)을 구하는 **축소**reductions 혹은 **요약 통계**summary statistics 범주에 속한다. 순수 NumPy 배열에서 제공하는 동일한 메서드와 비교하여 pandas의 메서드는 처음부터 누락된 데이터를 제외하도록 설계되었다. 다음과 같은 DataFrame을 생각해보자.

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                    [np.nan, np.nan], [0.75, -1.3]],
   .....:                   index=['a', 'b', 'c', 'd'],
   .....:                   columns=['one', 'two'])

In [231]: df
Out[231]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

DataFrame의 sum 메서드를 호출하면 각 컬럼의 합을 담은 Series를 반환한다.

```
In [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

axis='columns' 또는 axis=1 옵션을 넘기면 각 컬럼의 합을 반환한다.

```
In [233]: df.sum(axis='columns')
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
dtype: float64
```

전체 로우나 컬럼의 값이 NA가 아니라면 NA 값은 제외되고 계산된다. 이는 skipna 옵션으로 조정할 수 있다.

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

[표 5-7]에서 공통적으로 사용되는 축소 메서드의 옵션을 확인할 수 있다.

표 5-7 축소 메서드의 옵션

옵션	설명
axis	연산을 수행할 축. DataFrame에서 0은 로우고 1은 컬럼이다.
skipna	누락된 값을 제외할 것인지 정하는 옵션. 기본값은 True
level	계산하려는 축이 계층적 색인(다중 색인)이라면 레벨에 따라 묶어서 계산한다.

idxmin이나 idxmax 같은 메서드는 최솟값 혹은 최댓값은 가지고 있는 색인값과 같은 간접 통계를 반환한다.

```
In [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

또 다른 메서드로 **누산**accumulation이 있다.

```
In [236]: df.cumsum()
Out[236]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

축소나 누산이 아닌 다른 종류의 메서드로 describe가 있는데, 이 메서드는 한 번에 여러 개의 통계 결과를 만들어낸다.

```
In [237]: df.describe()
Out[237]:
             one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

수치 데이터가 아닐 경우 describe는 다른 요약 통계를 생성한다.

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count     16
unique     3
top        a
freq       8
dtype: object
```

[표 5-8]에 요약 통계 관련 메서드의 전체 목록을 나열했다.

표 5-8 요약 통계 관련 메서드

메서드	설명
count	NA 값을 제외한 값의 수를 반환한다.
describe	Series나 DataFrame의 각 컬럼에 대한 요약 통계를 계산한다.
min, max	최솟값과 최댓값을 계산한다.
argmin, argmax	각각 최솟값과 최댓값을 담고 있는 색인의 위치(정수)를 반환한다.
idxmin, idxmax	각각 최솟값과 최댓값을 담고 있는 색인의 값을 반환한다.
quantile	0부터 1까지의 분위수를 계산한다.
sum	합을 계산한다.

메서드	설명
mean	평균을 계산한다.
median	중간값(50% 분위)을 반환한다.
mad	평균값에서 평균절대편차를 계산한다.
prod	모든 값의 곱
var	표본분산의 값을 계산한다.
std	표본표준편차의 값을 계산한다.
skew	표본비대칭도(3차 적률)의 값을 계산한다.
kurt	표본첨도(4차 적률)의 값을 계산한다.
cumsum	누적합을 계산한다.
cummin, cummax	각각 누적 최솟값과 누적 최댓값을 계산한다.
cumprod	누적곱을 계산한다.
diff	1차 산술차를 계산한다(시계열 데이터 처리 시 유용하다).
pct_change	퍼센트 변화율을 계산한다.

5.3.1 상관관계와 공분산

상관관계나 공분산 같은 요약 통계 계산은 두 쌍의 인자를 필요로 한다. pandas-datareader 패키지를 이용해서 야후! 금융 사이트에서 구한 주식가격과 시가총액을 담고 있는 다음 DataFrame을 생각해보자. 아직 설치하지 않았다면 conda나 pip를 통해 설치할 수 있다.

```
conda install pandas-datareader
```

pandas_datareader 모듈을 이용해서 주가 정보를 다운로드하자.

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                       for ticker, data in all_data.items()})
```

CAUTION_ 2017년에 버라이즌이 야후!를 인수했기 때문에 이 책을 읽고 있는 시점에 야후! 금융 서비스가 더 이상 작동하지 않을 가능성도 있다. 최신 기능은 온라인에서 pandas-datareader 문서로 확인할 수 있다.

이제 각 주식의 퍼센트 변화율을 계산해보자. 시계열을 다루는 법은 11장에서 자세히 설명하겠다.

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

corr 메서드는 NA가 아니며 정렬된 색인에서 연속하는 두 Series에 대해 상관관계를 계산하고 cov 메서드는 공분산을 계산한다.

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

MSFT는 파이썬 속성 이름 규칙에 어긋나지 않으므로 좀 더 편리한 문법으로 해당 컬럼을 선택할 수 있다.

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

반면에 DataFrame에서 corr과 cov 메서드는 DataFrame 행렬에서 상관관계와 공분산을 계산한다.

```
In [247]: returns.corr()
Out[247]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [248]: returns.cov()
Out[248]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

DataFrame의 corrwith 메서드를 사용하면 다른 Series나 DataFrame과의 상관관계를 계산한다. Series를 넘기면 각 컬럼에 대해 계산한 상관관계를 담고 있는 Series를 반환한다.

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

DataFrame을 넘기면 맞아떨어지는 컬럼 이름에 대한 상관관계를 계산한다. 여기서 나는 시가총액의 퍼센트 변화율에 대한 상관관계를 계산해보았다.

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

axis='columns' 옵션을 넘기면 각 컬럼에 대한 상관관계와 공분산을 계산한다. 모든 경우 데이터는 상관관계를 계산하기 전에 색인의 이름순으로 정렬된다.

5.3.2 유일값, 값 세기, 멤버십

또 다른 종류의 메서드로는 1차원 Series에 담긴 값의 정보를 추출하는 메서드가 있다. 이를 설명하기 위해 다음과 같은 예제가 있다고 하자.

```
In [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

unique 메서드는 중복되는 값을 제거하고 유일값만 담고 있는 Series를 반환한다.

```
In [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

유일값은 정렬된 순서대로 반환되지 않지만 필요하다면 uniques.sort( )를 이용해서 나중에 정렬할 수 있다. 그리고 value_counts 메서드는 Series에서 도수frequency를 계산하여 반환한다.

```
In [254]: obj.value_counts()
Out[254]:
c    3
a    3
b    2
d    1
dtype: int64
```

value_counts에서 반환하는 Series는 담고 있는 값을 내림차순으로 정렬한다. value_counts 메서드는 pandas의 최상위 메서드로, 어떤 배열이나 순차 자료구조에서도 사용할 수 있다.

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a    3
b    2
c    3
d    1
dtype: int64
```

isin 메서드는 어떤 값이 Series에 존재하는지 나타내는 불리언 벡터를 반환하는데, Series나 DataFrame의 컬럼에서 값을 골라내고 싶을 때 유용하게 사용할 수 있다.

```
In [256]: obj
Out[256]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
Out[258]:
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool

In [259]: obj[mask]
Out[259]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

isin과 관련이 있는 Index.get_indexer 메서드는 여러 값이 들어 있는 배열에서 유일한 값의 색인 배열을 구할 수 있다.

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

[표 5-9]에서 지금까지 살펴본 메서드를 참조하자.

표 5-9 유일값, 값 세기, 멤버십 메서드

메서드	설명
isin	Series의 각 원소가 넘겨받은 연속된 값에 속하는지 나타내는 불리언 배열을 반환한다.
match	각 값에 대해 유일한 값을 담고 있는 배열에서의 정수 색인을 계산한다. 데이터 정렬이나 조인 형태의 연산을 하는 경우에 유용하다.
unique	Series에서 중복되는 값을 제거하고 유일값만 포함하는 배열을 반환한다. 결과는 Series에서 발견된 순서대로 반환된다.
value_counts	Series에서 유일값에 대한 색인과 도수를 계산한다. 도수는 내림차순으로 정렬된다.

DataFrame의 여러 컬럼에 대해 히스토그램을 구해야 하는 경우가 있다. 다음 예제를 보자.

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})

In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

위 DataFrame의 apply 함수에 pandas.value_counts를 넘기면 다음과 같은 결과를 얻을 수 있다.

```
In [265]: result = data.apply(pd.value_counts).fillna(0)

In [266]: result
Out[266]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

여기서 결괏값의 로우 라벨은 전체 컬럼의 유일한 값들을 담고 있다. 각 값은 각 컬럼에서 해당 값이 몇 번 출현했는지 나타낸다.

5.4 마치며

다음 장에서는 pandas를 이용해서 데이터셋을 읽고(또는 로딩) 쓰는 도구를 다루도록 하겠다. 그다음에는 데이터를 정제하고 분석하고 시각화하는 도구를 더 깊이 살펴본다.