Numpy

0. Numpy

- 배열과 벡터 계산
- Numerical Python의 줄임말인 Numpy는 고성능의 과학계산 컴퓨팅과 데이터 분석에 필요한 기본 패키지다. Numpy에서 제공하는 기능은 다음과 같다.

- 빠르고 메모리를 효율적으로 사용하며 벡터 산술연산과 세련된 **브로드캐스팅 기능**을 제공하는 다차원 배열인 ndarray
- 반복문을 작성할 필요 없이 전체 데이터 배열에 대해 빠른 연산을 제공하는 표준 수학 함수
- 배열 데이터를 디스크에 쓰거나 읽을 수 있는 도구와 메모리에 올려진 파일을 사용하는 도구
- 선형대수, 난수 발생기, 푸리에 변환 기능
- C, C++, 포트란으로 쓰여진 코드를 통합하는 도구

1. NumPy ndarray: 다차원 배열 객체

- **Numpy 사용 형식**

```
import numpy as np
```

- NumPy의 핵심 기능 중 하나는 N차원의 배열 객체 또는 ndarray로 파이썬에서 사용할 수 있는 대규모 데이터 집합을 담을 수 있는 빠르고 유연한 자료 구조다.

```
In[2]: import numpy as np
In[3]: data = np.random.randn(2, 3)
In[4]: data
Out[4]:
array([[-0.56285369, -0.06812205, 0.75793005],
[ 0.71350143, 0.114842 , -0.78867801]])
```

1. NumPy ndarray: 다차원 배열 객체

- ndarray는 같은 종류의 데이터를 담을 수 있는 포괄적인 다차원 배열이며, ndarray의 모든 원소는 같은 자료 형이어야만 한다.

```
In[6]: data * 10
Out[6]:
array([[-5.62853693, -0.68122054, 7.5793005 ],
[ 7.13501431, 1.14842001, -7.88678008]])
In[7]: data + data
Out[7]:
array([[-1.12570739, -0.13624411, 1.5158601 ],
[ 1.42700286, 0.229684 , -1.57735602]])
```

1. NumPy ndarray: 다차원 배열 객체

1.1 ndarray 생성

배열을 생성하는 가장 쉬운 방법은 array함수를 이용하는 것이다. 순차적인 객체(다른 배열도 포함하여)를 받아 넘겨받은 데이터가 들어있는 새로운 NumPy 배열을 생성한다. 예를 들어 파이썬의 리스트는 변환하기 좋은 예다.

```
In[10]: data1 = [6, 7.5, 8, 0, 1]
In[11]: arr1 = np.array(data1)
In[12]: arr1
Out[12]: array([ 6. , 7.5, 8. , 0. , 1. ])
```

1. NumPy ndarray: 다차원 배열 객체

1.1 ndarray 생성

같은 길이의 리스트가 담겨있는 순차 데이터는 다차원 배열로 변환이 가능하다.

```
In[23]: data2 = [[1,2,3,4],[5,6,7,8]]
In[24]: arr2 = np.array(data2)
In[25]: arr2
Out[25]:
array([[1, 2, 3, 4],[5, 6, 7, 8]])
In[26]: arr2.ndim
Out[26]: 2
In[27]: arr2.shape
Out[27]: (2, 4)
```

1. NumPy ndarray: 다차원 배열 객체

1.1 ndarray 생성

명시적으로 지정하지 않는 한 np.array는 생성될 때 적절한 자료형을 추정한다. 그렇게 추정된 자료형은 dtype 객체에 저장되는데, 먼저 살펴본 예로 들어 보면,

```
In[28]: arr1.dtype
Out[28]: dtype('float64')
In[29]: arr2.dtype
Out[29]: dtype('int32')
```

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

1. NumPy ndarray: 다차원 배열 객체

1.1 ndarray 생성

또한 np.array는 새로운 배열을 생성하기 위한 여러 함수를 가지고 있는데,

예를 들면 zeros와 ones는 주어진 길이나 모양에 각각 0과 1이 들어있는 배열을 생성한다. empty함수는 초기화되지 않은 배열을 생성하는데, 이런 메서드를 사용해서 다차원 배열을 생성하려면 원하는 형태의 튜플을 넘기면 된다.

```
In[30]: np.zeros(10)
Out[30]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
In[31]: np.zeros((3, 6)) # 6개의 0으로 구성된 리스트 3개
Out[31]:
array([[ 0., 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0., 0.]])
In[32]: np.empty((2, 3, 2)) '''2개의 초기화되지 않은 값으로 채워진 배열을 가진
리스트 3개를 가진 리스트를 2개 생성 '''
Out[32]:
array([[[ 8.74496193e-322, 0.00000000e+000],
[ 0.00000000e+000, 0.00000000e+000],
[ 0.00000000e+000, 0.00000000e+000]],

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

1. NumPy ndarray: 다차원 배열 객체

1.1 ndarray 생성

arange는 파이썬의 range 함수와 유사하지만 리스트 대신 ndarray를 반환한다.

```
In[33]: np.arange(15)
Out[33]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

- 배열 생성 함수 목록(page 119 표 4-1 참조)

1. NumPy ndarray: 다차원 배열 객체

1.2 ndarray의 자료형

자료형, dtype은 ndarray가 특정 데이터를 메모리에서 해석하기 위해 필요한 정보를 담고 있는 특수한 객체다.

```
In[34]: arr1 = np.array([1, 2, 3], dtype=np.float64)
In[35]: arr2 = np.array([1, 2, 3], dtype=np.int32)
In[36]: arr1.dtype
Out[36]: dtype('float64')
In[37]: arr2.dtype
Out[37]: dtype('int32')
```

1. NumPy ndarray: 다차원 배열 객체

- dtype가 있기에 NumPy가 강력하면서도 유연한 도구가 될 수 있었는데, 대부분의 데이터는 디스크에서 데이터를 읽고 쓰기 편하도록 하위 레벨의 표현에 직접적으로 맞춰져 있으며, C나 포트란 같은 저수준 언어로 작성된 코드와 쉽게 연동이 가능하다. 산술 데이터의 dtype는 float, int 같은 자료형의 이름과 하나의 원소가 차지하는 비트 수로 이루어진다. 파이썬의 float 객체에서 사용되는 표준 배정밀도 부동소수점 값은 8바이트 혹은 64비트로 이루어지는데, 이 자료형은 NumPy에서 float64로 표현된다.

- NumPy 자료형 목록은 page 120 표 4-2 참조

- NOTE : NumPy의 모든 dtype을 외울 필요는 없다. 주로 사용하게 될 자료형의 일반적인 종류만 신경 쓰면 된다. 주로 대용량 데이터가 메모리나 디스크에 저장되는 방식을 제어해야 할 필요가 있을 때 알아 두면 좋다.

1. NumPy ndarray: 다차원 배열 객체

- **ndarray의 astype 메서드**

- dtype을 다른 형으로 명시적 변경이 가능하다.

```
In[41]: arr = np.array([1, 2, 3, 4, 5])
In[42]: arr.dtype
Out[42]: dtype('int32')
In[43]: float_arr = arr.astype(np.float64) #int32에서 float64로 자료형 변경
In[44]: float_arr.dtype # 변경됨을 확인, 정수형을 부동소수점으로 변환하였다.
Out[44]: dtype('float64')
```

- 만약 부동소수점 숫자를 정수형으로 변환하면 소수점 아랫자리는 버려질 것이다.

```
In[45]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
In[46]: arr
Out[46]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
In[47]: arr.astype(np.int32)
Out[47]: array([ 3, -1, -2, 0, 12, 10]) # 소숫점 아랫자리가 버려졌다.
```

1. NumPy ndarray: 다차원 배열 객체

- 숫자 형식의 문자열을 담고 있는 배열이 있다면 astype을 사용하여 숫자로 변환 할 수 있다.

```
In[50]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_) #문자열을 가지고 있는 배열
In[51]: numeric_strings.astype(float)
#astype명령을 사용해 문자열을 float으로 바꾸어줌, float이라고해도 알아서 바꿔줌
Out[51]: array([ 1.25, -9.6 , 42. ])
```

```
In[52]: int_array = np.arange(10)  # 0~9 순차적으로
In[53]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
#int_array배열의 자료형을 calibers 배열의 dtype인 float64형태로 변환
In[54]: int_array.astype(calibers.dtype)
Out[54]: [0. , 1., 2., 3., 4., 5., 6., 7., 8., 9.]
```

1. NumPy ndarray: 다차원 배열 객체

1.3 배열과 스칼라 간의 연산

- 배열은 for 반복문을 작성하지 않고 데이터를 일괄 처리할 수 있기 때문에 중요하다. 이를 벡터화라고 하는데, 같은 크기의 배열 간 산술연산은 배열의 각 요소 단위로 적용된다.

```
In[52]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])
In[53]: arr
Out[53]:
array([[ 1., 2., 3.], [ 4., 5., 6.]])
In[54]: arr* arr
Out[54]:
array([[ 1., 4., 9.], [ 16., 25., 36.]])
In[55]: arr- arr
Out[55]:
array([[ 0., 0., 0.], [ 0., 0., 0.]])
```

- 스칼라 값에 대한 산술 연산은 각 요소로 전달 된다.
- 크기가 다른 배열 간의 연산은 브로드캐스팅이라고 한다.

1. NumPy ndarray: 다차원 배열 객체

- [참고] 브로드캐스팅

```
In[56]: arr = np.arange(10)
In[57]: arr
Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[58]: arr[5] #5번째 수 불러오기
Out[58]: 5
In[59]: arr[5:8] # 6번째에서부터 8번째까지
Out[59]: array([5, 6, 7])
```

1. NumPy ndarray: 다차원 배열 객체

- **1.4 색인과 슬라이싱 기초**
- NumPy 배열 색인에 대해서는 다룰 주제가 많다. 데이터의 부분 집합이나 개별 요소를 선택 하기 위한 수많은 방법이 존재한다. 1차원 배열은 단순하며 표면적으로는 파이썬의 리스트와 유사하게 동작한다.

```
In[56]: arr = np.arange(10)
In[57]: arr
Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[58]: arr[5] #5번째 수 불러오기
Out[58]: 5
In[59]: arr[5:8] # 6번째에서부터 8번째까지
Out[59]: array([5, 6, 7])
```

```
In[60]: arr[5:8] = 12 # 6~8번째 수를 12로 치환
In[61]: arr
Out[61]: array([ 0, 1, 2, 3, 4, 12, 12, 12, 8, 9])
```

1. NumPy ndarray: 다차원 배열 객체

- 이처럼 배열 조각에 스칼라 값을 대입하면 12가 선택 영역 전체로 전파된다. (이후로는 브로드캐스팅이라고 한다.) 리스트와의 중요한 차이점은 배열 조각은 원본 배열의 뷰라는 점이다. 즉, 데이터는 복사되지 않고 뷰에 대한 변경은 그대로 원본 배열에 반영된다는 것이다

```
In[62]: arr_slice = arr[5:8]
In[63]: arr_slice[1]
Out[63]: 12
In[64]: arr_slice[1] = 12345
In[65]: arr
Out[65]: array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])

# 여기서 보면 arr_slice의 변수에서 슬라이싱하고 조작했던게 arr에도 반영된다.
연동되어있다.
```

- NumPy는 대용량 데이터 처리를 염두에 두고 설계되었기 때문에 만약 NumPy가 데이터 복사를 남발한다면 성능과 메모리 문제에 직면할 것이다.
- NOTE : 만약에 뷰 대신 ndrray 슬라이스의 복사본을 얻고 싶다면 arr[5:8].copy()를 사용해서 명시적으로 배열을 복사하면 된다.

1. NumPy ndarray: 다차원 배열 객체

- 다차원 배열을 다루려면 좀 더 많은 옵션이 필요하다. 2차원 배열에서 각 색인에 해당하는 요소는 스칼라 값이 아니라 1차원 배열이 된다.

```
In[66]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In[67]: arr2d[2]
Out[67]: array([7, 8, 9])
```

- 따라서 개별 요소는 재귀적으로 접근해야 한다. 다음의 두 표현은 같다. 하지만 그렇게 하기는 귀찮으니 구분된 색인 리스트를 넘기면 된다.

```
In[68]: arr2d[0][2]
Out[68]: 3
In[69]: arr2d[0, 2]
Out[69]: 3
```

1. NumPy ndarray: 다차원 배열 객체


```
In[73]: arr3d =
np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
In[74]: arr3d
Out[74]:
array([[[ 1, 2, 3],
[ 4, 5, 6]],

[[ 7, 8, 9],
[10, 11, 12]]])
In[75]: arr3d[0]
Out[75]:
array([[1, 2, 3],
[4, 5, 6]])
In[76]: old_values = arr3d[0].copy()
In[77]: arr3d[0] = 42
```

```
In[78]: arr3d
Out[78]:
array([[[42, 42, 42],
[42, 42, 42]],

[[ 7, 8, 9],
[10, 11, 12]]])
In[79]: arr3d[0] = old_values
In[80]: arr3d
Out[80]:
array([[[ 1, 2, 3],
[ 4, 5, 6]],

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

1. NumPy ndarray: 다차원 배열 객체

- **슬라이스 색인**

- 파이썬의 리스트 같은 1차원 객체처럼 ndarray는 익숙한 문법으로 슬라이싱 할 수 있다.

```
In[87]: arr[1:6]
Out[87]: array([ 1, 2, 3, 4, 64])
In[88]: arr2d
Out[88]:
array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
In[89]: arr2d[:2]
Out[89]:
array([[1, 2, 3],[4, 5, 6]])
In[90]: arr2d[:2, 1:]
Out[90]:
array([[2, 3],[5, 6]])
In[91]: arr2d[1, :2]
Out[91]: array([4, 5])
In[92]: arr2d[2, :1]
Out[92]: array([7])
```

- 물론 슬라이싱 구문에 값을 대입하면 선택 영역 전체에 값이 할당된다.

1. NumPy ndarray: 다차원 배열 객체

1.5 불리언 색인

- 중복된 이름이 포함된 배열이 있다고 하자. 그리고 numpy.random 모듈에 있는 randn 함수를 사용해서 임의의 표준정규분포 데이터를 생성하자.

```
In[93]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
In[94]: data = np.random.randn(7, 4) # 4개의 난수를 갖는 리스트를 7개 뱉어라
In[95]: names
Out[95]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='|S4')
In[96]: data
Out[96]:
array([[-0.5099588 , -0.26549335, 0.47190496, 1.28825708],
[-0.22645075, -0.33822525, 1.15948599, 0.62708074],
[-1.05545525, -0.29507054, 1.8157602 , -0.23001171],
[-0.2477215 , 0.16012906, 1.1783344 , 1.8553623 ],
[-0.82402224, 1.02722829, 0.54261483, -0.20464688],
[ 1.22017486, 1.25984082, 1.39068215, -0.61045326],
[ 1.40302308, -0.73592691, 1.28374355, 0.08466187]])
```

1. NumPy ndarray: 다차원 배열 객체

1.5 불리언 색인

- 중복된 이름이 포함된 배열이 있다고 하자. 그리고 numpy.random 모듈에 있는 randn 함수를 사용해서 임의의 표준정규분포 데이터를 생성하자.

```
In[93]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
In[94]: data = np.random.randn(7, 4) # 4개의 난수를 갖는 리스트를 7개 뱉어라
In[95]: names
Out[95]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='|S4')
In[96]: data
Out[96]:
array([[-0.5099588 , -0.26549335, 0.47190496, 1.28825708],
[-0.22645075, -0.33822525, 1.15948599, 0.62708074],
[-1.05545525, -0.29507054, 1.8157602 , -0.23001171],
[-0.2477215 , 0.16012906, 1.1783344 , 1.8553623 ],
[-0.82402224, 1.02722829, 0.54261483, -0.20464688],
[ 1.22017486, 1.25984082, 1.39068215, -0.61045326],
[ 1.40302308, -0.73592691, 1.28374355, 0.08466187]])
```

- 각각의 이름은 data 배열의 각 행에 대응한다고 가정하자. 만약에 전체 행에서 'Bob'과 같은 이름을 선택하려면 산술연산과 마찬가지로 배열에 대한 비교연산(== 같은) 도 벡터화 된다.

1. NumPy ndarray: 다차원 배열 객체

- 이 불리언 배열은 배열의 색인으로 사용할 수 있다.
- 이 불리언 배열은 반드시 색인하려는 축의 길이와 동일한 길이를 가져야 한다. 불리언 배열 색인도 슬라이스 또는 숫자 색인과 함께 혼용할 수 있다.

```
In[101]: names == 'Bob' # names에서 'Bob' 인 것들은 True 아니면 False
Out[101]: array([ True, False, False, True, False, False, False], dtype=bool)
In[102]: data[names == 'Bob'] # 0, 3에서 True
Out[102]:
array([[-0.5099588 , -0.26549335, 0.47190496, 1.28825708],
[-0.2477215 , 0.16012906, 1.1783344 , 1.8553623 ]])
In[103]: data[names == 'Bob', 2:] # 0,3 의 배열중에 3번째 부터 끝까지 뱉어라
Out[103]:
array([[ 0.47190496, 1.28825708],
[ 1.1783344 , 1.8553623 ]])
```

1. NumPy ndarray: 다차원 배열 객체

- Q) 0,3 번째를 참조하는 값 어떻게 될까?
- 'Bob'이 아닌 요소를 선택하려면 != 연산자를 사용하거나 ~를 사용해서 조건절을 부정하면 된다.

  name != 'Bob'

  data[~(names == 'Bob')]
- 세 가지 이름 중에서 두 가지 이름을 선택하려면 &(and) 와 |(or) 같은 논리 연산자를 사용한 여러 개의 불리언 조건을 조합하여 사용하면 된다.

```
In[107]: mask = (names == 'Bob') | (names == 'Will')
In[108]: mask #1,3,4,5만 True
Out[108]: array([ True, False, True, True, True, False, False],
dtype=bool)
In[109]: data[mask] # 1,3,4,5 값만 내놔라
Out[109]:
array([[-0.5099588 , -0.26549335, 0.47190496, 1.28825708],
[-1.05545525, -0.29507054, 1.8157602 , -0.23001171],
[-0.2477215 , 0.16012906, 1.1783344 , 1.8553623 ],
[-0.82402224, 1.02722829, 0.54261483, -0.20464688]])
```

*파이썬의 and와 or는 불리언 배열에서 사용할 수 없다.

1. NumPy ndarray: 다차원 배열 객체

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

- 불리언 배열에 값을 대입하는 것은 상식선에서 이루어지며, data에 저장된 모든 음수를 0으로 대입하려면 아래와 같이 하면 된다.

```
data[data <0] = 0

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

1. NumPy ndarray: 다차원 배열 객체

- 3차원 배열의 해석

```
print("3차원 배열(높이가 3인 2행 4열 배열 = 2행 4열 배열 3개)")
array3 = np.arange(24).reshape(3, 2, 4)
print(array3)
print("만약 배열의 크기가 너무 커서, 다 표시될 수 없으면 자동으로 배열의 중간 값들을 생략한다.")
print(np.arange(10000).reshape(100,100))
```

1. NumPy ndarray: 다차원 배열 객체

1.6 팬시 색인

- 팬시 색인은 정수 배열을 사용한 색인을 설명하기 위해 NumPy에서 차용한 단어다. 8X4 크기의 배열이 있다고 하자.

```
In[120]: arr = np.empty((8, 4)) #초기화 되지 않은 4개의 값을 같는 리스트 8개로 된 배열 만들기
# 0~7까지의 숫자를 i에 반복문, array 색인 마다 같은 번호 부여
In[121]: for i in range(8):
... arr[i] = i
In[123]: arr
Out[121]:
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.]])
```

1. NumPy ndarray: 다차원 배열 객체

1.6 팬시 색인

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

```
In[125]: arr[[4, 3, 0, 6]] # array중 5번째, 4번째, 1번째, 7번째 참조
Out[123]:
array([[ 4., 4., 4., 4.],
[ 3., 3., 3., 3.],
[ 0., 0., 0., 0.],
[ 6., 6., 6., 6.]])
```

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

1. NumPy ndarray: 다차원 배열 객체

1.6 팬시 색인

- 이 예제를 잠시 살펴보자 (1,0), (5,3), (7,1) (2,2)에 대응하는 요소가 선택 되었다. 이 예제에서 팬시 색인은 우리의 예상과는 조금 다르게 동작했다.

```
In[131]: arr = np.arange(32).reshape((8,4)) # 4개의 값을 가진 리스트 8개를 반환하는데 0~31의 값을 갖게 한다.
In[132]: arr
Out[130]:
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[133]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
'''2번째 배열에서 첫 번째 값, 6번째 배열에서 4번째 값, 8번째 베열에서 2번째 값, 3번째 배열에서 3번째 값'''

Out[131]: array([ 4, 23, 29, 10])
```

1. NumPy ndarray: 다차원 배열 객체

1.6 팬시 색인

- 행렬의 행과 열에 대응하는 사각형 모양의 값이 선택되기를 기대했는데 사실 그렇게 하려면 다음처럼 선택해야 한다.

```
In[135]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]] # 2, 6, 8, 3 행에서 1, 4, 2, 3순서로 담은 배열을 갖게 해라
Out[133]:
array([[ 4, 7, 5, 6],
[20, 23, 21, 22],
[28, 31, 29, 30],
[ 8, 11, 9, 10]])
```

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

```
In[125]: arr[[4, 3, 0, 6]] # array중 5번째, 4번째, 1번째, 7번째 참조
Out[123]:
array([[ 4., 4., 4., 4.],
[ 3., 3., 3., 3.],
[ 0., 0., 0., 0.],
[ 6., 6., 6., 6.]])
```

1. NumPy ndarray: 다차원 배열 객체

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

```
In[131]: arr = np.arange(32).reshape((8,4)) # 4개의 값을 가진 리스트 8개를 반환하는데 0~31의 값을 갖게 한다.
In[132]: arr
Out[130]:
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]])
```

1. NumPy ndarray: 다차원 배열 객체

- 이 예제를 잠시 살펴보자 (1,0), (5,3), (7,1) (2,2)에 대응하는 요소가 선택되었다. 이 예제에서 팬시 색인은 우리의 예상과는 조금 다르게 동작했다.

```
In[133]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
'''2번째 배열에서 첫 번째 값, 6번째 배열에서 4번째 값,
8번째 베열에서 2번째 값, 3번째 배열에서 3번째 값'''

Out[131]: array([ 4, 23, 29, 10])
```

1. NumPy ndarray: 다차원 배열 객체

- 행렬의 행과 열에 대응하는 사각형 모양의 값이 선택되기를 기대했는데 사실 그렇게 하려면 다음처럼 선택해야 한다.

```
# 2, 6, 8, 3 행에서 1, 4, 2, 3순서로 담은 배열을 갖게 해라
In[135]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[133]:
array([[ 4, 7, 5, 6],
[20, 23, 21, 22],
[28, 31, 29, 30],
[ 8, 11, 9, 10]])
```

```
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]])
```

1. NumPy ndarray: 다차원 배열 객체

- np.ix_ 함수를 사용하면 같은 결과를 얻을 수 있는데, 1차원 정수 배열 2개를 사각형 영역에서 사용할 색인으로 변환해준다.

```
In[138]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[135]:
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.

1. NumPy ndarray: 다차원 배열 객체

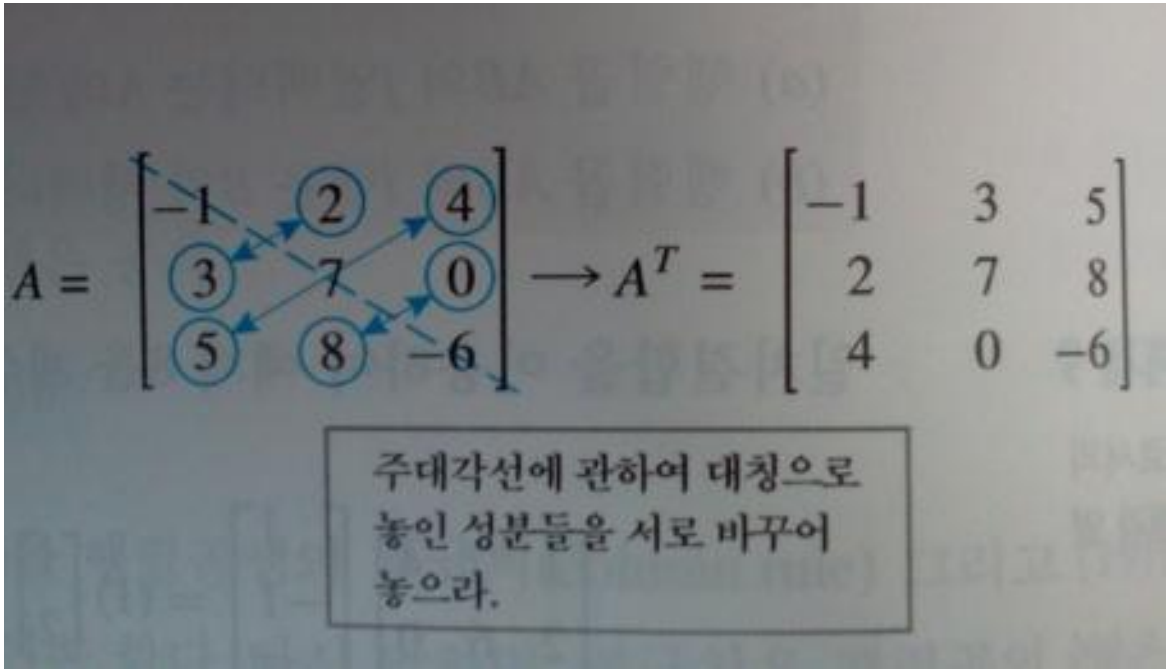
1.7 배열 전치와 축 바꾸기

배열 전차는 데이터를 복사하지 않고 데이터 모양이 바뀐 뷰를 반환하는 특별한 기능이다. ndarray는 transpose 메서드와 T라는 이름의 특수한 속성을 가지고 있다.

```
In[139]: arr = np.arange(15).reshape((3, 5))
In[140]: arr
Out[137]:
array([[ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]])
In[141]: arr.T   #전치행렬
Out[138]:
array([[ 0, 5, 10],
[ 1, 6, 11],
[ 2, 7, 12],
[ 3, 8, 13],
[ 4, 9, 14]])
```

1. NumPy ndarray: 다차원 배열 객체

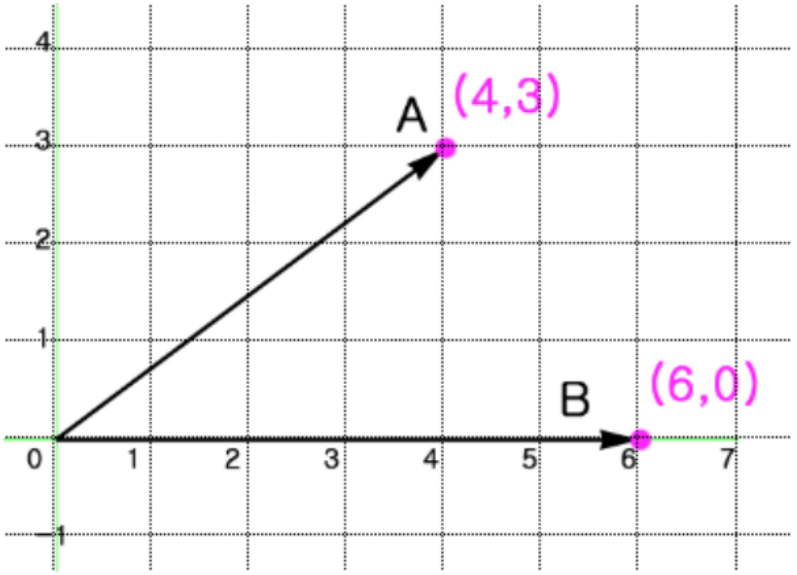
- 전치행렬



1. NumPy ndarray: 다차원 배열 객체

- **벡터의 내적**

1) 스칼라 값을 이용하는 방법



A·B (A와 B의 내적)

$= (A_x \times B_x) + (A_y \times B_y)$

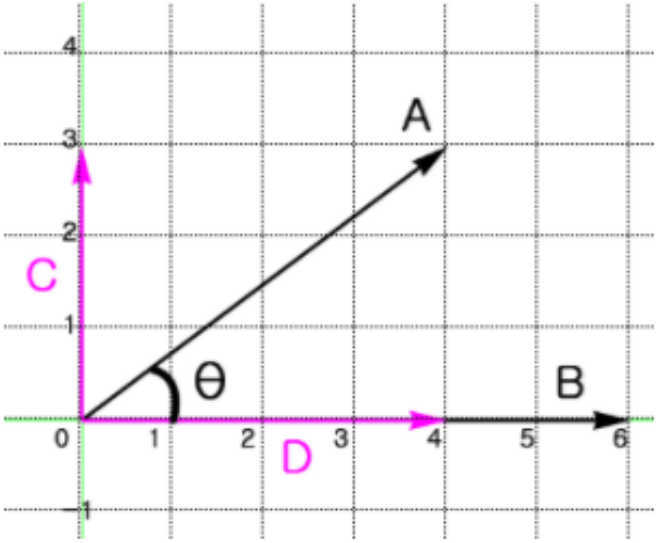
$= (4 \times 6) + (3 \times 0) = 24$

http://mrw0119.tistory.com/12

1. NumPy ndarray: 다차원 배열 객체

- **벡터의 내적**

2) 벡터의 크기를 곱하는 방법



A·B (A와 B의 내적)

= B 크기 × D의 크기

= | B | × | A | × cosθ

= 6 × 4 = 24

http://mrw0119.tistory.com/12

1. NumPy ndarray: 다차원 배열 객체

- **벡터의 내적**

3) 단위벡터(크기가1인 벡터)를 이용한 내적

$$A \cdot B = |B| \times |A| \times \cos\theta$$
$$= 1 \times 1 \times \cos\theta$$
$$= \cos\theta$$

이 공식을 통해 두 단위벡터의 내적은 cos@가 된다는 것을 알 수 있다.

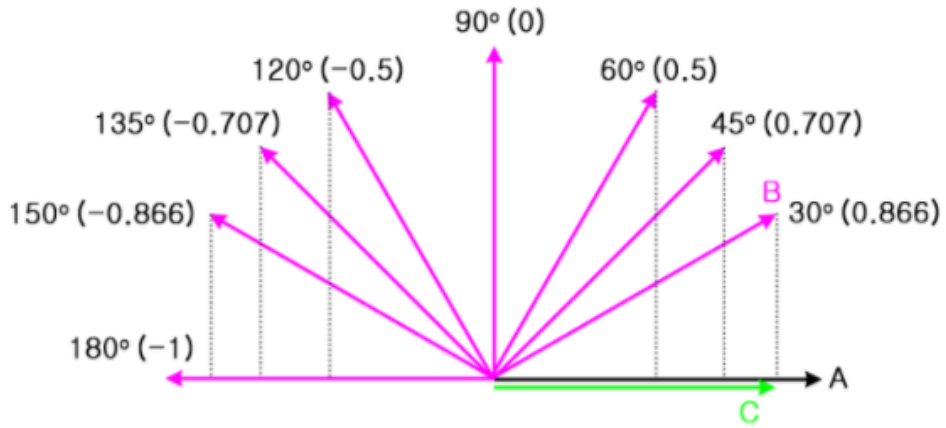
http://mrw0119.tistory.com/12

1. NumPy ndarray: 다차원 배열 객체

- **벡터의 내적**

3) 단위벡터(크기가1인 벡터)를 이용한 내적

- 단위벡터 A의 각에 따르는 내적 값



- B벡터를 30도 회전시켰을 때 A와 B의 내적값은 0.866이고, 60도 회전시켰을 때의 내적값은 0.5이다. 이는 벡터 C(B의 분해벡터)가 벡터 A에 대해 뻗어있는 정도(비율)을 말한다.

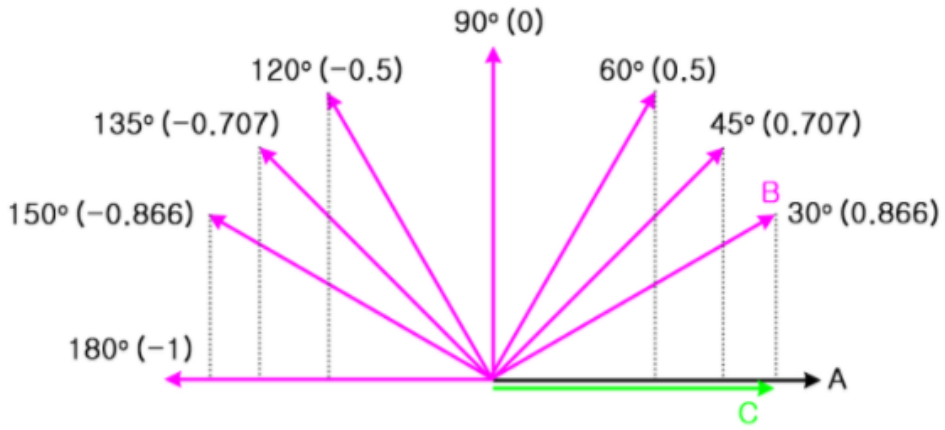
http://mrw0119.tistory.com/12

1. NumPy ndarray: 다차원 배열 객체

- **벡터의 내적**

3) 단위벡터(크기가1인 벡터)를 이용한 내적

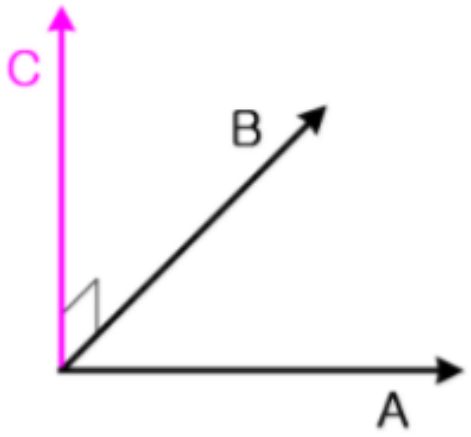
- 내적의 의미 : 두 벡터의 내적의 값이 양수이면 두 벡터의 사이각이 0~90도 또는 0~-90도 이고, 내적값이 음수이면 두 벡터의 사이각이 90~180도 또는 -90~-180도 이며, 내적값이 0이면 두 벡터는 수직관계이다.



http://mrw0119.tistory.com/12

1. NumPy ndarray: 다차원 배열 객체

- 벡터의 외적



A×B (벡터 A,B의 외적)
= 벡터 C (A,B의 수직벡터)

1. NumPy ndarray: 다차원 배열 객체

- **전치행렬**

선형대수학에서, 어떤 행렬의 전치행렬(轉置行列, transposed matrix)은 원래 행렬의 열은 행으로, 행은 열으로 바꾼 것이다. 정사각행렬의 경우에는 행렬의 왼쪽 위에서 오른쪽 아래를 가르는 주대각선을 기준으로 대칭되는 원소끼리 바꿔치기하여 전치행렬을 얻을 수 있다. A행렬의 전치행렬은 A^{T}, ${}^{\mathrm{t}}A$, A', A^{tr} 등으로 나타낸다.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

$n \times n$ $matrix$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$m \times n$ $matrix$

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \quad \begin{bmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{bmatrix}$$

1. NumPy ndarray: 다차원 배열 객체

1.7 배열 전치와 축 바꾸기

- 행렬 계산을 할 때 자주 사용하게 될 텐데, 예를 들면 **행렬의 내적** X^TX 는 np.dot을 이용해서 구할 수 있다.

```
In[145]: arr = np.arange(16).reshape((2, 2, 4))
In[146]: arr
Out[143]:
array([[[ 0, 1, 2, 3],
[ 4, 5, 6, 7]],
[[ 8, 9, 10, 11],
[12, 13, 14, 15]]])
In[147]: arr.transpose((1,0,2))   #축 번호(높이=1,행=0, 열=2)를 받아 치환
Out[144]:
array([[[ 0, 1, 2, 3],
[ 8, 9, 10, 11]],
[[ 4, 5, 6, 7],
[12, 13, 14, 15]]])
```

1. NumPy ndarray: 다차원 배열 객체

1.7 배열 전치와 축 바꾸기

- .T 속성을 이용하는 간단한 전치는 축을 뒤바꾸는 특별한 경우다. ndarray에는 swapaxes 메서드가 있는데 2개의 축 번호를 받아서 배열을 뒤바꾼다.

```
In[148]: arr
Out[145]:
array([[[ 0, 1, 2, 3],[ 4, 5, 6, 7]],

[[ 8, 9, 10, 11],
[12, 13, 14, 15]]])
In[149]: arr.swapaxes(1, 2)
Out[146]:
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.

2 유니버설 함수

- ufunc라고 불리는 유니버설 함수는 ndarray 안에 있는 데이터 원소별로 연산을 수행하는 함수다. 유니버설 함수는 하나 이상의 스칼라 값을 받아서 하나 이상의 스칼라 결과 값을 반환하는 간단한 함수를 고속으로 수행할 수 있는 벡터화된 래퍼 함수라고 생각하면 된다.
- sqrt나 exp같은 간단한 변형을 전체 원소에 적용할 수 있다. 이 경우는 단항 유니버설 함수라 하고, add나 maximum처럼 2개의 인자를 취해서 단일 배열을 반환하는 함수를 이항 유니버설 함수라고 한다. *배열 여러개를 반환*

```
In[150]: x = np.random.randn(8)
In[151]: y = np.random.randn(8)
In[152]: x
Out[149]:
array([ 0.57483741, 0.39598833, -0.35815866, -1.73518055, 1.09899127,
0.81740202, 0.82778791, -0.34151437])
In[153]: y
Out[150]:
array([ 0.92638336, 0.70822114, 0.66838903, 0.76546891, 0.22617098,
-1.29688062, 1.9109471 , 2.54965647])
In[154]: np.maximum(x, y)
Out[151]:
array([ 0.92638336, 0.70822114, 0.66838903, 0.76546891, 1.09899127,
0.81740202, 1.9109471 , 2.54965647])
```

3. XML Parsing

- https://docs.python.org/3/library/xml.etree.elementtree.html

3. “Automatic” Reshaping


```
>>> a = np.arange(30)
>>> a.shape = 2,-1,3  # -1 means "whatever is needed"
>>> a.shape
(2, 5, 3)
>>> a
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]]])
```

Pandas

0. Pandas 의 특징

- pandas는 고수준의 자료구조와 파이썬을 통한 빠르고 쉬운 데이터 분석 도구를 포함한다.
- pandas는 Numpy 기반에서 개발되어 Numpy를 사용하는 애플리케이션에서 쉽게 사용 가능하다.

① 자동적으로 혹은 명시적으로 축의 이름에 따라 데이터를 정렬할 수 있는 자료 구조, 잘못 정렬된 데이터에 의한 일반적인 오류를 예방하고 다양한 소스에서 가져온 다양한 방식으로 색인되어 있는 데이터를 다룰 수 있는 기능

② 통합된 시계열 기능

③ 시계열 데이터와 비시계열 데이터를 함께 다룰 수 있는 통합 자료 구조

④ 산술연산과 한 축의 모든 값을 더하는 등의 데이터 축약연산은 축의 이름 같은 메타데이터로 전달될 수 있어야 한다.

⑤ 누락된 데이터를 유연하게 처리할 수 있는 기능

⑥ SQL같은 일반 데이터베이스처럼 데이터를 합치고 관계연산을 수행하는 기능

0. Pandas 사용

- **from** pandas **import** Series, DataFrame
  import pandas **as** pd
  import numpy **as** np

- pd는 pandas를 지칭하며 Series와 DataFrame은 로컬 네임스페리스로 import 하는 것이 편해서 이렇게 사용함.

1. Pandas의 자료구조

- Series와 DataFrame

1-1. Series

- Series는 일련의 객체를 담을 수 있는 1차원 배열 같은 자료 구조다. (어떤 NumPy 자료형이라도 담을 수 있다.)
- 색인(index)이라고 하는 배열의 데이터에 연관된 이름을 가지고 있다. 가장 간단한 Series객체는 배열 데이터로부터 생성할 수 있다.

```
In[6]: obj = Series([4, 7, -5, 3])
In[7]: obj
Out[7]:
0 4
1 7
2 -5
3 3
dtype: int64
```

- Series 객체의 문자열 표현은 왼쪽에 색인(index)을 보여주고 오른쪽에 해당 색인의 값을 보여준다.
- **데이터 색인 디폴트 : 기본 색인인 정수 0에서 N-1까지의 숫자가 표시된다.**

1-1. Series

- Series의 배열과 색인 객체는 각각 values와 index 속성을 통해 얻을 수 있다.

```
In[8]: obj.values
Out[8]: array([ 4, 7, -5, 3], dtype=int64)
In[9]: obj.index
Out[9]:  Int64Index([0, 1, 2, 3], dtype='int64')
```

- 각각의 데이터에 색인을 지정해 생성해 다음처럼 생성한다.

```
In[10]: obj2 = Series([4, 7, -5, 3],
                      index=['d', 'b', 'a', 'c'])
In[11]: obj2
Out[11]:
d 4
b 7
a -5
c 3
dtype: int64
In[12]: obj2.index
Out[12]: Index([u'd', u'b', u'a', u'c'], dtype='object')
```

1-1. Series

- 배열에서 값을 선택하거나 대입할 때는 색인을 이용해서 접근한다.

```
In[13]: obj2['a']
Out[13]: -5
In[14]: obj2['d'] = 6
In[15]: obj2[['c', 'a', 'd']]
Out[15]:
c 3
a -5
d 6
dtype: int64
```

Pandas의 자료구조 : Series

- 불리언 배열을 사용해서 값을 걸러내거나 산술 곱셈을 수행하거나 또는 수학 함수를 적용하는 등 NumPy 배열연산을 수행해도 색인- 값 연결은 유지된다.

```
In[17]: obj2[obj2>0]
Out[17]:
d 6
b 7
c 3
dtype: int64
In[18]: obj2 * 2
Out[18]:
d 12
b 14
a -10
c 6
dtype: int64
#np.exp(x) = e^x (e는 자연상수)
In[19]: np.exp(obj2)
Out[19]:
d 403.428793
b 1096.633158
a 0.006738
c 20.085537
dtype: float64
```

1-1. Series

- Series를 이해하는 다른 방법은 고정 길이의 정렬된 사전형(dictionary)이라고 이해하는 것이다. Series는 색인 값에 데이터 값을 매핑하고 있으므로 파이썬의 사전형과 비슷하다. Series 객체는 파이썬의 사전형을 인자로 받아야 하는 많은 함수에서 사전형을 대체하여 사용할 수 있다.

```
In[20]: 'b' in obj2
Out[20]: True
In[21]: 'e' in obj2
Out[21]: False
```

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

```
In[22]: sdata = {'Ohio':35000, 'Texas': 71000,
'Oregon':16000, 'Utah':5000}
In[23]: obj3 = Series(sdata)
In[24]: obj3
Out[24]:
Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
dtype: int64
```

1-1. Series

- 사전 객체만 가지고 Series 객체를 생성하면 생성된 Series 객체의 색인은 사전의 키 값이 순서대로 들어간다.

```
In[28]: states = ['California', 'Ohio', 'Oregon', 'Texas']
In[29]: obj4 = Series(sdata, index=states)
In[30]: obj4
Out[30]:
California NaN
Ohio 35000
Oregon 16000
Texas 71000
dtype: float64
```

- 이 예제를 보면 sdata에 있는 값 중 3개만 확인할 수 있는데, 이는 'California'에 대한 값을 찾을 수 없기 때문이다. 이 값은 NaN (not a number)으로 표시되고 pandas에서는 누락된 값 혹은 NA값으로 취급된다.]

1-1. Series

- pandas의 isnull과 notnull 함수는 누락된 데이터를 찾을 때 사용된다.

```
In[31]: pd.isnull(obj4)
Out[31]:
California True
Ohio False
Oregon False
Texas False
dtype: bool
In[32]: pd.notnull(obj4)
Out[32]:
California False
Ohio True
Oregon True
Texas True
dtype: bool
```

1-1. Series

- isnull과 notnull 함수는 Series의 인스턴스 메서드이기도 하다.

```
In[33]: obj4.isnull()

Out[33]:
California True
Ohio False
Oregon False
Texas False
dtype: bool
```

1-1. Series

- 누락된 데이터를 처리하는 방법은 이 장의 끝부분에서 좀 더 자세히 살펴보기로 하자. 가장 주요한 Series의 기능은 다르게 색인된 데이터에 대한 산술 연산이다.

In[34]: **obj3 In[35]: obj4**
Out[34]: Out[35]:

Ohio 35000	California NaN
Oregon 16000	Ohio 35000
Texas 71000	Oregon 16000
Utah 5000	Texas 71000
dtype: int64	dtype: float64

In[36]: **obj3 + obj4**
Out[36]:
California NaN
Ohio 70000
Oregon 32000
Texas 142000
Utah NaN
dtype: float64

1-1. Series

- Series 객체와 Series의 색인은 모두 name 속성이 있는데, 이 속성은 pandas의 기능에서 중요한 부분을 차지하고 있다.

```
In[49]: obj4.name = 'population'
In[50]: obj4.index.name = 'state'
In[51]: obj4
Out[51]:
state
California NaN
Ohio 35000
Oregon 16000
Texas 71000
Name: population, dtype: float64
```

```
#Series의 색인은 대입을 통해 변경할 수 있다.
In[52]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
In[53]: obj
Out[53]:
Bob 4
Steve 7
Jeff -5
Ryan 3

dtype: int64
```

1-2. DataFrame

- DataFrame은 표 같은 스프레드시트 형식의 자료 구조로 여러 개의 칼럼이 있는데, 각 칼럼은 서로 다른 종류의 값(숫자, 문자열, 불리언 등)을 담을 수 있다.
- DataFrame은 로우(row)와 칼럼에 대한 색인(index)이 있는데, 이 DataFrame은 색인의 모양이 같은 Series 객체를 담고 있는 파이썬 사전으로 생각하면 편하다. R의 data.frame 같은 다른 DataFrame과 비슷한 자료 구조와 비교했을 때, DataFrame에서의 로우 연산과 칼럼 연산은 거의 대칭적으로 취급된다. 내부적으로 데이터는 리스트나 사전 또는 1차원 배열을 담고 있는 다른 컬렉션이 아니라 하나 이상의 2차원 배열에 저장된다.

1-2. DataFrame

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

```
In[57]: data = {'state' : ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
... 'year' : [2000, 2001, 2002, 2001, 2002],
... 'pop' : [1.5, 1.7, 3.6, 2.4, 2.9]}


...
In[60]: frame = DataFrame(data)
```

- 만들어진 DataFrame의 색인은 Series와 같은 방식으로 자동으로 대입되며 칼럼은 정렬되어 저장된다.

```
In[62]: frame
Out[60]:
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
```

Pandas의 자료구조 : DataFrame

- 원하는 순서대로 columns를 지정하면 원하는 순서를 가진 DataFrame 객체가 생성된다.

```
In[63]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[61]:
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
```

1-2. DataFrame

- Series와 마찬가지로 data에 없는 값을 넘기면 NA 값이 저장된다.

```
In[64]: frame2 =
DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                                  index=['one','two','three','four','five'])
In[65]: frame2
Out[63]:
          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
In[66]: frame2.columns
Out[64]: Index([u'year', u'state', u'pop', u'debt'], dtype='object')
```

1-2. DataFrame

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

```
In[67]: frame2['state']
Out[65]:
one      Ohio
two      Ohio
three    Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
In[68]: frame2.year
Out[66]:
one 2000
two 2001
three 2002
four 2001
five 2002
Name: year, dtype: int64
```

1-2. DataFrame

- 반환된 Series 객체가 DataFrame 같은 색인을 가지면 알맞은 값으로 name 속성이 채워진다.
- 로우는 위치나 ix 같은 몇 가지 메서드를 통해 접근할 수 있다.

```
In[69]: frame2.ix['three']
Out[67]:
year    2002
state   Ohio
pop     3.6
debt    NaN
Name:   three, dtype: object
```

1-2. DataFrame

- 칼럼은 대입이 가능하다. 예를 들면 현재 비어있는 'debt' 칼럼에 스칼라 값이나 배열의 값을 대입할 수 있다.

```
In[70]: frame2['debt'] = 16.5
In[71]: frame2
Out[69]:
          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
In[72]: frame2['debt'] = np.arange(5.)
In[73]: frame2
Out[71]:
year state pop debt
one 2000 Ohio 1.5 0
two 2001 Ohio 1.7 1
three 2002 Ohio 3.6 2
four 2001 Nevada 2.4 3
five 2002 Nevada 2.9 4
```

1-2. DataFrame

- 리스트나 배열을 칼럼에 대입할 때는 대입하려는 값의 길이가 DataFrame의 크기와 같아야 한다.
- Series를 대입하면 DataFrame의 색인에 따라 값이 대입되며 없는 색인에는 값이 대입되지 않는다.

```
In[74]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
In[75]: frame2['debt'] = val
In[76]: frame2
Out[74]:
       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
```

1-2. DataFrame

- 없는 칼럼을 대입하면 새로운 칼럼이 생성된다. 파이썬 사전형에서와 마찬가지로 del 예약어를 사용해서 칼럼을 삭제할 수 있다.

```
In[77]: frame2['eastern']= frame2.state == 'Ohio'
In[78]: frame2
Out[76]:
            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
In[79]: del frame2['eastern']
In[80]: frame2.columns
Out[78]: Index([u'year', u'state', u'pop', u'debt'], dtype='object')
```

Pandas의 자료구조 : DataFrame

- **DataFrame의 색인을 이용해서 생성된 칼럼은 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.
- 또한 중첩된 사전을 이용하여 데이터를 생성할 수 있는데, 다음과 같은 중첩한 사전이 있다면.

```
In[81]: pop = {'Nevada' : {2001: 2.4, 2002: 2.9},
'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

- 바깥에 있는 사전의 키 값이 칼럼이 되고 안에 있는 키는 로우가 된다.

```
In[81]: pop = {'Nevada' : {2001: 2.4, 2002: 2.9},
              'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
In[82]: frame3 = DataFrame(pop)
In[83]: frame3
Out[81]:
Nevada Ohio
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
```

1-2. DataFrame

- NumPy에서와 마찬가지로 결과 값의 순서를 뒤집을 수 있다.

```
In[84]: frame3.T
Out[82]:
2000 2001 2002
Nevada NaN 2.4 2.9
Ohio 1.5 1.7 3.6
```

- 중첩된 사전을 이용해서 DataFrame을 생성할 때 안쪽에 있는 사전 값은 키 값별로 조합되어 결과의 색인이 되지만 색인을 직접 지정한다면 지정된 색인으로 DataFrame을 생성한다.

```
In[85]:
DataFrame(pop, index=[2001, 2002, 2003])
Out[83]:
Nevada Ohio
2001 2.4 1.7
2002 2.9 3.6
2003 NaN NaN
```

1-2. DataFrame

- Series 객체를 담고 있는 사전 데이터도 같은 방식으로 취급된다.

```
In[86]: pdata = {'ohio' : frame3['Ohio'][:-1],
'Nevada': frame3['Nevada'][:2]}
In[87]: DataFrame(pdata)
Out[85]:
Nevada ohio
2000 NaN 1.5

2001 2.4 1.7
```

- DataFrame 생성자에 넘길 수 있는 자료형의 목록은 [표]을 참고하자.

```
In[88]: frame3.index.name = 'year';
frame3.columns.name = 'state'
In[89]: frame3
Out[87]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
```

1-2. DataFrame

- DataFrame 생성자에서 사용 가능한 입력데이터(1/2)

형	설명
2차원 ndarray	데이터를 담고 있는 행렬, 선택적으로 로우와 칼럼의 이름을 전달할 수 있다.
배열, 리스트, 튜플의 사전	사전의 모든 항목은 같은 길이를 가져야 하며, 각 항목의 내용이 DataFramew의 칼럼이 된다.
Numpy으 구조화 배열	배열의 사전과 같은 방식으로 취급된다.
Series 사전	Series의 각 값이 컬럼이 된다. 명시적으로 색인을 넘겨주지 않으면 각 Series의 색인 하나로 합쳐져서 로우의 색인이 된다.
사전의 사전	내부에 있는 사전이 컬럼이 된다. 키 값은 Series의 사전과 마찬가지로 합쳐져 로우의 색인이 된다.

1-2. DataFrame

- DataFrame 생성자에서 사용 가능한 입력데이터(2/2)

형	설명
사전이나 Series의 리스트	리스트의 각 항목이 DataFrame의 로우가 된다. 합쳐진 사전의 키 값이나 Series의 색인이 DataFrame 컬럼의 이름이 된다.
리스트나 튜플의 리스트	2차원 ndarray와 같은 방식으로 취급된다.
다른 DataFramew	색인이 따로 지정되지 않는다면 DataFramew의 색인이 그대로 사용된다.
Numpy MaskedArray	2차원 ndarray와같은 방식으로 취급되지만 마스크 값은 반환되는 DataFrame에서 NA 값이 된다.

1-2. DataFrame

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

```
In[90]: frame3.values
Out[88]:
array([[ nan, 1.5],
[ 2.4, 1.7],
[ 2.9, 3.6]])
```

- DataFrame의 칼럼에 서로 다른 dtype이 있다면 모든 칼럼을 수용하기 위해 그 칼럼 배열의 dtype이 선택된다.

```
In[91]: frame2.values
Out[89]:
array([[2000L, 'Ohio', 1.5, nan],
[2001L, 'Ohio', 1.7, -1.2],
[2002L, 'Ohio', 3.6, nan],
[2001L, 'Nevada', 2.4, -1.5],
[2002L, 'Nevada', 2.9, -1.7]], dtype=object)
```

1-3. 색인(index) 객체

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

```
In[92]: obj = Series(range(3), index=['a', 'b', 'c'])

In[93]: index = obj.index

In[94]: index

Out[92]: Index([u'a', u'b', u'c'], dtype='object')

In[95]: index[1:]

Out[93]: Index([u'b', u'c'], dtype='object')
```

1-3. 색인(index) 객체

- 색인 객체는 변경할 수 없다.

```
In[96]: index[1] = 'd'

Traceback (most recent call last):
  File "C:₩Users₩Jusung₩Anaconda2₩lib₩site-
packages₩IPython₩core₩interactiveshell.py", line 3066, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-94-676fdeb26a68>", line 1, in <module>
    index[1] = 'd'
  File "C:₩Users₩Jusung₩Anaconda2₩lib₩site-
packages₩pandas₩core₩index.py", line 1130, in __setitem__..(이하 생략..)
```

- 색인 객체는 변경될 수 없기에 자료 구조 사이에서 안전하게 공유될 수 있다.

```
In[101]: index = pd.Index(np.arange(3))
In[102]: index
Out[100]: Int64Index([0, 1, 2], dtype='int64')
In[103]: obj2= Series([1.5, -2.5, 0], index=index)
In[104]: obj2.index is index
Out[102]: True
```

1-3. 색인(index) 객체

- [표]에는 pandas에서 사용한 내장 색인 클래스가 정리되어 있다. 특수한 목적으로 축을 색인하는 기능을 개발하기 위해 Index 클래스의 서브클래스를 만들 수 있다.
- <pandas의 주요 Index객체>

클래스	설명
Index	가장 일반적인 Index 객체이며, 파이썬 객체의 Numpy 배열 형식으로 축의 이름을 표현한다.
Int64Index	정수값을 위한 특수한 Index
MultiIndex	단일 축에 여러 단계의 색인을 표현하는 계층적 색인 객체, 튜플의 배열과 유사하다고 볼 수 있다.
DatetimeIndex	나노초 타임스탬프를 저장한다.(Numpy의 datetime64 dtype으로 표현된다)
PeriodIndex	기간 데이터를 위한 특수한 indexs

1-3. 색인(index) 객체

- 또한 배열과 유사하게 Index 객체도 고정 크기로 동작한다.

```
In[107]: frame3
Out[105]:
state Nevada Ohio
year
2000 NaN 1.5
2001 2.4 1.7
2002 2.9 3.6
In[108]: 'Ohio' in frame3.columns
Out[106]: True
In[109]: 2003 in frame3.index
Out[107]: False
```

1-3. 색인(index) 객체

- 각각의 색인은 담고 있는 데이터에 대한 정보를 취급하는 여러 가지 메서드와 속성을 가지고 있다.

클래스	설명
appned	추가적인 Index 객체를 덧붙여 새로운 색인을 반환한다.
diff	색인의 차집합을 반환한다.
intersection	색인의 교집합을 반환한다.
union	색인의 합집합을 반환한다.
isin	넘겨받은 값이 해당 색인 위치에 존재하는지 알려주는 불리언 배열을 반환한다.
deletesd	I 위치의 색인이 삭제된 새로운 색인을 반환한다.
drop	넘겨받은 값이 삭제된 새로운 색인을 반환한다.
insert	I 위치에 값이 추가된 새로운 색인을 반환한다.

1-3. 색인(index) 객체

- 각각의 색인은 담고 있는 데이터에 대한 정보를 취급하는 여러 가지 메서드와 속성을 가지고 있다.
- [표] 색인 메서드와 속성

클래스	설명
is_monotonic	색인이 단조성을 가진다면 True을 반환한다.
is_unique	중복되는 색인이 없다면 True을 반환한다.
unique	색인에서 중복되는 요소를 제거하고 유일한 값만을 반환한다.

2. 핵심 기능

- Series나 DataFrame에 저장된 데이터를 다루는 기본 방법을 설명한다. 앞으로 pandas를 이용한 데이터 분석과 조작에 관한 좀 더 자세한 내용을 살펴볼 것이다. 이 책은 pandas 라이브러리에 대한 완전한 설명은 자제하고 중요한 기능에만 초점을 맞추고 있다.

2-1. 재색인(reindex)

- pandas객체의 기막힌 기능 중 하나인 reindex는 새로운 색인에 맞도록 객체를 새로 생성하는 기능이다.

```
In[110]: obj = Series([4.5, 7.2, -5.3, 3.6], index= ['d', 'b', 'a', 'c'])
In[111]: obj
Out[109]:
d 4.5
b 7.2
a -5.3
c 3.6
dtype: float64
```

2-1. 재색인(reindex)

- 이 Series 객체에 대해 reindex를 호충하면 데이터를 새로운 색인에 맞게 재배열하고, 없는 값이 있다면 비어있는 값을 새로 추가한다.

```
In[112]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
In[113]: obj2
Out[111]:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN
dtype: float64
In[114]: obj.reindex(['a','b','c','d','e'], fill_value=0.0)
Out[112]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0

dtype: float64
```

2-1. 재색인(reindex)

- 시계열 같은 순차적인 데이터를 재색인할 때 값을 보간하거나 채워 넣어야 할 경우가 있다. 이런 경우 method 옵션을 이용해서 해결할 수 있으며, ffill 메서드를 이용하면 앞의 값으로 누락된 값을 채워넣을 수 있다.

```
In[115]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
In[116]: obj3.reindex(range(6), method='ffill')
Out[114]:
0 blue
1 blue
2 purple
3 purple
4 yellow
5 yellow
dtype: object
```

- reindex 메소드(보간) 옵션

인자	설명
ffill 또는 pad	앞의 값으로 채워 넣는다.
bfill 또는 backfill	뒤의 값으로 채워 넣는다.

2-1. 재색인(reindex)

- DataFrame에 대한 reindex는 (로우) 색인, 칼럼 또는 둘 다 변경이 가능하다. 그냥 순서만 전달하면 로우가 재색인된다.

```
In[117]: frame =
DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'b', 'd'],
                        columns= ['Ohio', 'Texas', 'California'])
In[118]: frame
Out[116]:
Ohio Texas California
a 0 1 2
b 3 4 5
d 6 7 8
In[119]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
In[120]: frame2
Out[118]:
Ohio Texas California
a 0 1 2
b 3 4 5
c NaN NaN NaN
d 6 7 8
```

2-1. 재색인(reindex)

- 열은 columns 예약어를 사용해서 재색인할 수 있다.

```
In[121]: states = ['Texas', 'Utah', 'California']
In[122]: frame.reindex(columns=states)
Out[120]:
Texas Utah California
a 1 NaN 2
b 4 NaN 5
d 7 NaN 8
```

- 로우와 칼럼은 모두 한 번에 재색인할 수 있지만 보간은 로우에 대해서만 이루어진다 (axis 0).

```
In[123]:
frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill', columns=states)
Out[121]:
Texas Utah California
a 1 NaN 2
b 4 NaN 5
c 4 NaN 5
d 7 NaN 8
```

2-1. 재색인(reindex)

- **재색인은 ix를 이용해서** 라벨로 색인하면 좀 더 간결하게 할 수 있다.

```
In[124]: frame.ix[['a', 'b', 'c', 'd'], states]
Out[122]:
Texas Utah California
a 1 NaN 2
b 4 NaN 5
c NaN NaN NaN
d 7 NaN 8
```

2-1. 재색인(reindex)

- 재색인 함수 인자

인자	설명
index	색인으로 사용할 새로운 순서, Index 인스턴스나 다른 순차적인 자료 구조를 사용할 수 있다. 색인은 복사가 이루어지지 않고 그대로 사용된다.
method	보간 메서도(앞 표참조)
full_value	재색인 과정 중에 새롭게 나타나는 비어있는 데이터를 채우기 위한 값
limit	전/후 보간 시에 사용할 최대 갭 크기
level	MultiIndex 단계(level)에 단순 색인을 맞춘다. 그렇지 않으면 MultiIndex의 하위 부분집합에 맞춘다.
copy	True인 경우 새로운 색인이 이전 색인과 같더라도 데이터를 복사한다. False라면 두 색인 같을 경우 데이터를 복사하지 않는다.

2-2. 하나의 로우 또는 칼럼 삭제하기

- 색인 배열 또는 삭제하려는 로우나 칼럼이 제외된 리스트를 이미 가지고 있다면 로우나 칼럼을 쉽게 삭제할 수 있는데, 이 방법은 데이터의 모양을 변경하는 작업이 필요하다.
- drop 메서드를 사용하면 선택한 값이 삭제된 새로운 개체를 얻을 수 있다.

```
In[125]: obj =
Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
In[126]: new_obj = obj.drop('c')
In[127]: new_obj
Out[125]:
a 0
b 1
d 3
e 4
dtype: float64
```

```
In[128]: obj.drop(['d', 'c'])
Out[126]:
a 0
b 1
e 4
dtype: float64
```

2-2. 하나의 로우 또는 칼럼 삭제하기

- DataFrame에서는 로우와 칼럼 모두에서 값을 삭제할 수 있다.

```
In[130]: data.drop(['Colorado', 'Ohio'])
Out[128]:
one two three four
Utah 8 9 10 11
New York 12 13 14 15
In[131]: data.drop('two', axis=1)
Out[129]:
one three four
Ohio 0 2 3
Colorado 4 6 7
Utah 8 10 11
New York 12 14 15
In[132]: data.drop(['two', 'four'], axis=1)
Out[130]:
one three
Ohio 0 2
Colorado 4 6
Utah 8 10
New York 12 14
```

2-3. 색인하기, 선택하기, 거르기

- Series의 색인 (obj[...])은 NumPy 배열의 색인과 유사하게 동작하는데, Series의 색인은 정수가 아니어도 된다는 점이 다르다.
- 라벨 이름으로 슬라이싱하는 것은 시작점과 끝점을 포함한다는 점이 일반 파이선에서 슬라이싱과 다른 점이다.

```
In[136]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
In[137]: obj['b':'c']
Out[135]:
b 1
c 2
dtype: float64
```

- 슬라이싱 문법으로 선택된 영역에 값을 대입하는 것은 예상한 대로 동작한다.

```
In[138]: obj['b':'c'] = 5
In[139]: obj
Out[137]:
a 0
b 5
c 5
d 3
dtype: float64
```

2-3. 색인하기, 선택하기, 거르기

- 앞에서 확인한대로 색인으로 DataFrame에서 칼럼의 값을 하나 이상 가져올 수 있다.

```
In[140]: data = DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns = ['one', 'two', 'three', 'four'])
In[141]: data
Out[139]:
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
```

2-3. 색인하기, 선택하기, 거르기

- 색인으로 DataFrame에서 칼럼의 값을 하나 이상 가져올 수 있다.

```
In[142]: data['two']
Out[140]:
Ohio 1
Colorado 5
Utah 9
New York 13
Name: two, dtype: int32
In[143]: data[['three', 'one']]
Out[141]:
three one
Ohio 2 0
Colorado 6 4
Utah 10 8

New York 14 12
```

5-3. 색인하기, 선택하기, 거르기

- 슬라이싱으로 로우를 선택하거나 불리언 배열로 칼럼을 선택할 수 있다.

```
In[144]: data[:2]
Out[142]:
one two three four
Ohio 0 1 2 3
Colorado 4 5 6 7
In[145]: data[data['three'] > 5]
Out[143]:
one two three four
Colorado 4 5 6 7
Utah 8 9 10 11
New York 12 13 14 15
```

- 이 문법에 모순이 있다고 생각할 수 있지만, 실용성에 기인한 것일 뿐이다.

5-3. 색인하기, 선택하기, 거르기

- 또 다른 사례는 스칼라 비교를 통해 생성된 불리언 DataFrame을 사용해서 값을 선택하는 것이다.

```
In[146]: data < 5
Out[144]:
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[147]: data[data < 5] = 0
In[148]: data
Out[146]:
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
```

- 이 예제는 DataFrame을 ndarray와 문법적으로 비슷하게 보이도록 의도한 것이다.

2-3. 색인하기, 선택하기, 거르기

- DataFrame의 칼럼에 대해 라벨로 색인하는 방법으로, 특수한 색인 필드인 ix를 소개한다. ix는 NumPy와 비슷한 방식에 추가적으로 축의 라벨을 사용하여 DataFrame의 로우와 칼럼을 선택할 수 있도록 한다. 이 방법은 재색인을 좀 더 간단하게 할 수 있는 방법이다.

```
In[149]: data.ix['Colorado', ['two', 'three']]
Out[147]:
two 5
three 6
Name: Colorado, dtype: int32
In[150]: data.ix[['Colorado', 'Utah'], [3,0,1]]
Out[148]:
four one two
Colorado 7 0 5
Utah 11 8 9
In[151]: data.ix[2]
Out[149]:
one 8
two 9
three 10
four 11
Name: Utah, dtype: int32
```

```
In[152]: data.ix[:'Utah', 'two']
Out[150]:
Ohio 0
Colorado 5
Utah 9
Name: two, dtype: int32
In[153]: data.ix[data.three > 5, :3]
Out[151]:
one two three
Colorado 0 5 6
Utah 8 9 10
New York 12 13 14
```

2-3. 색인하기, 선택하기, 거르기

- pandas 객체에서 데이터를 선택하고 재배열하는 방법은 여러 가지가 있다.
- DataFrame의 값 선택하기

방식	설명
obj[val]	DataFrame에서 하나의 칼럼 또는 여러 칼럼을 선택한다. 편의를 위해 불리언 배열, 슬라이스, 불리언 DataFrame(어떤 기준에 근거해서 값을 대입해야 할 때)을 사용할 수 있다.
obj.ix[val]	DataFrame에서 로우의 부분집합을 선택한다.
obj.ix[: , val]	DataFrame에서 칼럼의 부분집합을 선택한다.
obj.ix[val1 , val2]	DataFrame에서 로우와 칼럼의 부분집합을 선택한다.
reindex()	하나 이상의 축을 새로운 색인으로 맞춘다.
icol(), irow()	각각 정수 색인으로 단일 로우나 칼럼을 Series 형식으로 선택한다.
get_value,set_value()	로우와 칼럼 이름으로 DataFrame의 값을 선택한다.

2-4. 산술 연산과 데이터 정렬

- pandas에서 중요한 기능은 색인이 다른 객체 간의 산술연산이다. 객체를 더할 때 짝이 맞지 않는 색인이 있다면 결과에 두 색인이 통합된다.

```
In[159]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd','e'])
In[160]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
In[161]: s1 + s2
Out[159]:
a 5.2
c 1.1
d NaN
e 0.0
f NaN
g NaN
dtype: float64
```

2-4. 산술 연산과 데이터 정렬

- 서로 겹치는 색인이 없다면 데이터는 NA 값이 된다. 산술연산 시 누락된 값은 전파되며, DataFrame에서는 로우와 칼럼 모두에 적용된다.

```
In[162]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
                         index=['Ohio', 'Texas', 'Colorado'])
In[163]: df2 = DataFrame(np.arange(12.).reshape((4,3)), columns=list('bde'),
                         index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In[164]: df1 + df2
Out[162]:
b c d e
Colorado NaN NaN NaN NaN
Ohio 3 NaN 6 NaN
Oregon NaN NaN NaN NaN
Texas 9 NaN 12 NaN
Utah NaN NaN NaN NaN
```

2-4. 산술 연산과 데이터 정렬

Big Data Analytics

- 산술연산 메서드에 채워 넣을 값 지정하기
- 서로 다른 색인을 가지는 객체 간의 산술연산에서 존재하지 않는 축의 값을 특수한 값(0 같은)으로 지정하고 싶을 때는 다음과 같이 할 수 있다.

```
In[168]: df1 = DataFrame(np.arange(12.).reshape((3,4)),
columns=list('abcd'))
df2 = DataFrame(np.arange(20.).reshape((4,5)),
columns=list('abcde'))
In[169]: df1
Out[167]:
a b c d
0 0 1 2 3
1 4 5 6 7
2 8 9 10 11
In[170]: df2
Out[168]:
a b c d e
0 0 1 2 3 4
1 5 6 7 8 9
2 10 11 12 13 14
3 15 16 17 18 19
```

```
In[171]: df1 + df2
Out[169]:
a b c d e
0 0 2 4 6 NaN
1 9 11 13 15 NaN
2 18 20 22 24 NaN
3 NaN NaN NaN NaN
NaN
```

- 이 둘을 더했을 때 겹치지 않는 부분의 값이 NA값이 된 것을 알 수 있다.

2-4. 산술 연산과 데이터 정렬

- df1의 add 메서드로 df2와 fill_value 값을 인자로 전달한다.

```
In[172]: df1.add(df2, fill_value=0)

Out[170]:
a b c d e
0 0 2 4 6 4
1 9 11 13 15 9
2 18 20 22 24 14
3 15 16 17 18 19
In[173]: df1.reindex(columns=df2.columns, fill_value=0)

Out[171]:
a b c d e
0 0 1 2 3 0
1 4 5 6 7 0
2 8 9 10 11 0
```

- Series나 DataFrame을 재색인할 때 역시 fill_value를 지정할 수 있다.

2-4. 산술 연산과 데이터 정렬

- DataFrame과 Series 간의 연산
- NumPy 배열의 연산처럼 DataFrame과 Series 간의 연산도 잘 정의되어 있다. 먼저 2차원 배열과 그 배열 중 한 칼럼의 차이에 대해서 생각할 수 있는 예제를 살펴보자.

```
In[175]: arr
Out[173]:
array([[ 0., 1., 2., 3.],
[ 4., 5., 6., 7.],
[ 8., 9., 10., 11.]])
In[176]: arr[0]
Out[174]: array([ 0., 1., 2., 3.])
In[177]: arr - arr[0]
Out[175]:
array([[ 0., 0., 0., 0.],
[ 4., 4., 4., 4.],
[ 8., 8., 8., 8.]])
```

2-4. 산술 연산과 데이터 정렬

- 브로드캐스팅은 다른 모양의 배열 간 산술연산을 어떻게 수행해야 하는지를 설명한다.

```
In[6]: frame = DataFrame(np.arange(12.).reshape((4, 3)),
columns=list('bde'),
index=['Utah', 'Ohio', 'Texas', 'Oregon'])
In[7]: series = frame.ix[0]
In[8]: frame
Out[8]:
b d e
Utah 0 1 2
Ohio 3 4 5
Texas 6 7 8
Oregon 9 10 11
In[9]: series
Out[9]:
b 0
d 1
e 2
Name: Utah, dtype: float64
```

브로드캐스팅 규칙

만일 이어지는 각 차원(시작부터 끝까지)에 대해 축의길이가 일치하거나 둘 중 하나의 길이가 1이라면 두 배열은 브로드캐스팅 호환이다. 브로드캐스팅은 누락된, 혹은 길이가 1인 차원에 대해 수행된다.

2-4. 산술 연산과 데이터 정렬

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

```
In[10]: frame - series
Out[10]:
b d e
Utah 0 0 0
Ohio 3 3 3
Texas 6 6 6
Oregon 9 9 9
```

- 만약 색인 값을 DataFrame의 칼럼이나 Series의 색인에서 찾을 수 없다면 그 객체는 형식을 맞추기 위해 재색인된다.

```
In[11]: series2 = Series(range(3), index = list('bef'))
In[12]: frame + series2
Out[12]:
b d e f
Utah 0 NaN 3 NaN
Ohio 3 NaN 6 NaN
Texas 6 NaN 9 NaN
Oregon 9 NaN 12 NaN
```

2-4. 산술 연산과 데이터 정렬

- 만약 각 로우에 대해 연산을 수행하고 싶다면 산술연산 메서드를 사용하면 된다.

```
In[17]: series3 = frame['d']

In[18]: frame

Out[18]:
b d e
Utah 0 1 2
Ohio 3 4 5
Texas 6 7 8
Oregon 9 10 11
```

```
In[19]: series3
Out[19]:
Utah 1
Ohio 4
Texas 7
Oregon 10
Name: d, dtype: float64
In[20]: frame.sub(series3, axis=0)

Out[20]:
b d e
Utah -1 0 1
Ohio -1 0 1
Texas -1 0 1
Oregon -1 0 1
```

5-5. 함수 적용과 매핑

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

```
In[21]: frame = DataFrame(np.random.randn(4,3), columns=list('bde'),
            index=['Utah', 'Ohio', 'Texas', 'Oregon'])
In[22]: frame
Out[22]:
b d e
Utah -1.077549 1.063706 1.300466
Ohio -0.028174 -0.805961 1.124684
Texas 0.884636 -0.271803 0.133528
Oregon -1.284909 0.136602 0.705397
In[23]: np.abs(frame) #절대값
Out[23]:
b d e
Utah 1.077549 1.063706 1.300466
Ohio 0.028174 0.805961 1.124684
Texas 0.884636 0.271803 0.133528
Oregon 1.284909 0.136602 0.705397
```

5-5. 함수 적용과 매핑

- 자주 사용되는 또 다른 연산은 각 로우나 칼럼의 1차원 배열에 함수를 적용하는 것이다.
- DataFrame의 apply 메서드를 통해 수행할 수 있다.

```
In[24]: f = lambda x: x.max() - x.min()
In[25]: frame.apply(f)
Out[25]:
b 2.169545
d 1.869667
e 1.166938
dtype: float64
In[26]: frame.apply(f, axis=1)
Out[26]:
Utah 2.378015
Ohio 1.930645
Texas 1.156440
Oregon 1.990306
dtype: float64
```

2-5. 함수 적용과 매핑

- 배열의 합계나 평균 같은 일반적인 통계는 DataFrame의 메서드로 있으므로 apply 메서드를 사용해야만 하는 것은 아니다.
- apply 메서드에 전달된 함수는 스칼라 값을 반환할 필요가 없으며, Series 또는 여러 값을 반환해도 된다.

```
In[27]: def f(x):
return Series([x.min(), x.max()], index=['min', 'max'])
In[28]: frame.apply(f)
Out[28]:
b d e
min -1.284909 -0.805961 0.133528
max 0.884636 1.063706 1.300466
```

2-5. 함수 적용과 매핑

- 배열의 각 원소에 적용되는 파이썬의 함수를 사용할 수도 있다. frame 객체에서 실수 값을 문자열 포맷으로 변환하고 싶다면 applymap을 이용해서 다음과 같이 해도 된다.

```
In[31]: format = lambda x: '%.2f' % x
In[32]: frame.applymap(format)
Out[32]:
b d e
Utah -1.08 1.06 1.30
Ohio -0.03 -0.81 1.12
Texas 0.88 -0.27 0.13
Oregon -1.28 0.14 0.71
```

2-5. 함수 적용과 매핑

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

```
In[35]: frame['e'].map(format)
Out[35]:
Utah 1.30
Ohio 1.12
Texas 0.13
Oregon 0.71
Name: e, dtype: object
```

5.6 정렬과 순위

- 어떤 기준에 근거해서 데이터를 정렬하는 것 역시 중요한 명령이다. 로우나 칼럼의 색인을 알파벳 순으로 정렬하려면 정렬된 새로운 객체를 반화하는 sort_index 메서드를 사용하면 된다.

```
In[35]: frame['e'].map(format)
Out[35]:
Utah      1.30
Ohio      1.12
Texas     0.13
Oregon    0.71
Name: e, dtype: object
```

- 어떤 기준에 근거해서 데이터를 정렬하는 것 역시 중요한 명령이다. 로우나 칼럼의 색인을 알파벳 순으로 정렬하려면 정렬된 새로운 객체를 반환하는 sort_index 메서드를 사용하면 된다.

```
In[36]: obj =
Series(range(4), index=['d', 'a', 'b', 'c'])
In[37]: obj.sort_index()
Out[37]:
a    1
b    2
c    3
d    0 dtype: int64
```

2.6 정렬과 순위

- DataFrame은 로우나 칼럼 중 하나의 축을 기준으로 정렬할 수 있다.

```
In[40]: frame = DataFrame(np.arange(8).reshape((2,4)),
                          index = ['three', 'one'], columns = ['d', 'a', 'b', 'c'])
In[41]: frame.sort_index()
Out[41]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
In[42]: frame.sort_index(axis=1)
Out[42]:
a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

2.6 정렬과 순위

- 데이터는 기본적으로 오름차순으로 정렬되지만 내림차순으로 정렬할 수도 있다.

```
In[43]: frame.sort_index(axis=1, ascending=False)
Out[43]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

2.7 중복색인

- 지금까지 살펴본 모든 예제는 모두 축의 이름(색인 값)이 유일했다.
- pandas의 많은 함수(reindex 같은) 에서 색인 값은 유일해야 하지만 강제 사항은 아니다. 이제 색인 값이 중복된 Series객체를 살펴보자.

```
In[42]: obj =
Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
In[43]: obj
Out[41]:
a 0
a 1
b 2
b 3
c 4
dtype: int64
```

- 색인의 is_unique 속성은 해당 값이 유일한지 아닌지 알려준다.

```
In[44]: obj.index.is_unique
Out[42]: False
```

2.7 중복색인

- 중복되는 색인 값이 있으면 색인을 이용한 데이터 선택은 다르게 동작하고 하나의 Series 객체를 반환한다. 하지만 중복되는 색인 값이 없으면 색인을 이용한 데이터 선택은 스칼라 값을 반환한다.

```
In[45]: obj['a']
Out[43]:
a 0
a 1
dtype: int64
In[46]: obj['c']
Out[44]: 4
```

5.7 중복색인

- DataFrame에서 로우를 선택하는 것도 동일하다.

```
In[47]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b','b'])
In[48]: df
Out[46]:
0 1 2
a 2.053621 0.432342 0.236329
a -0.843233 2.030160 0.603298
b -0.487331 0.949086 -0.163972
b -2.950345 -0.615662 -1.235880
```

```
In[49]: df.ix['b']

Out[47]:
0 1 2
b -0.487331 0.949086 -0.163972
b -2.950345 -0.615662 -1.235880
```

3. 기술통계 계산과 요약

- pandas 객체는 일반적인 수학 메서드와 통계 메서드를 가지고 있다. 이 메서드는 대부분 Series나 DataFrame 하나의 칼럼이나 로우에서 단일 값(합이나 평균 같은)을 구하는 축소 혹은 요약통계 범주에 속한다. 순수 NumPy 배열에서 제공하는 동일한 메서드와 비교하여 pandas의 메서드는 처음부터 누락된 데이터를 제외하도록 설계되었다. 다음과 같은 DataFrame을 생각해보자.

```
In[50]: df = 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[51]: df
Out[49]:
one two
a 1.40 NaN
b 7.10 -4.5
c NaN NaN
d 0.75 -1.
```

3. 기술통계 계산과 요약

- DataFrame의 sum 메서드를 호출하면 각 칼럼의 합을 담은 Series를 반환한다.

```
In[53]: df.sum()
Out[51]:
one 9.25
two -5.80
dtype: float64

In[54]: df.sum(axis=1) #axis=1 옵션을 넘기면 각 로우의 합을 반환한다.
Out[52]:
a 1.40
b 2.60
c 0.00
d -0.55
dtype: float64
```

3. 기술통계 계산과 요약

- 전체 로우나 칼럼의 값이 NA가 아니라면 계산 과정에서 NA 값은 제외시키고 계산된다. 이는 skipna 옵션을 통해 조정할 수 있다.

```
In[55]: df.mean(axis=1, skipna=False)
Out[53]:
a NaN
b 1.300
c NaN
d -0.275
dtype: float64
```

3. 기술통계 계산과 요약

- idxmin이나 idxmax 같은 메서드는 최소 혹은 최대 값을 가지고 있는 색인 값 같은 간접 통계를 반환한다.(1/2)

```
In[59]: df.idxmax()

Out[57]:
one b
two d
dtype: object
In[60]: df.cumsum() # 누산 메서드

Out[58]:
one two
a 1.40 NaN
b 8.50 -4.5
c NaN NaN
d 9.25 -5.8
```

3. 기술통계 계산과 요약

- idxmin이나 idxmax 같은 메서드는 최소 혹은 최대 값을 가지고 있는 색인 값 같은 간접 통계를 반환한다.(2/2)

```
Out[59]:
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
```

3-1. 상관관계와 공분산

- 상관관계와 공분산 같은 요약통계 계산은 인자가 두 벌 필요하다. 야후! 금융사이트에서 구한 주식가격과 시가 총액을 담고 있는 다음 DataFrame에 대해 생각해보자.

```
In[62]: import pandas.io.data as web

C:\Users\Jusung\Anaconda2\lib\site-packages\pandas\io\data.py:33: FutureWarning:
The pandas.io.data module is moved to a separate package (pandas-datareader) and will
be removed from pandas in a future version.
After installing the pandas-datareader package (https://github.com/pydata/pandas-datareader)
, you can change the import ``from pandas.io import data, wb`` to ``
```

```
from pandas_datareader import data, wb``. FutureWarning)
```

3-1. 상관관계와 공분산

- Future Warning이 뜨지만 (파이썬 2.7버전) 괜찮다.

```
In[63]: all_data = {}

In[64]: for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:

... all_data[ticker] = web.get_data_yahoo(ticker)


...
In[66]: price = DataFrame({tic: data['Adj
Close'] for tic, data in all_data.items()})
In[67]: volume = DataFrame({tic:
data['Volume'] for tic, data in all_data.items()})
```

3-1. 상관관계와 공분산

- 각 주식의 퍼센트 변화율을 계산해보겠다.

```
In[68]: returns = price.pct_change()

In[69]: returns.tail()
Out[66]:
AAPL GOOG IBM MSFT
Date
2016-03-31 -0.005203 -0.007435 0.020484 0.003270
2016-04-01 0.009175 0.006658 0.007065 0.006156
2016-04-04 0.010274 -0.006161 -0.002950 -0.002519
2016-04-05 -0.011789 -0.010050 -0.013612 -0.015695
2016-04-06 0.010473 0.010694 0.000133 0.010264
```

3-1. 상관관계와 공분산

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

```
In[73]: returns.MSFT.corr(returns.IBM)

Out[70]: 0.5004475899488285 4
In[74]: returns.MSFT.cov(returns.IBM)

Out[71]: 8.998439535124876 6e-05
```

3-1. 상관관계와 공분산

- 반면에 DataFrame에서 corr과 cov메서드는 DataFrame 행렬상에서 상관관계와 공분산을 계산한다.

```
n[75]: returns.corr()

Out[72]:
AAPL GOOG IBM MSFT
AAPL 1.000000 0.409961 0.393292 0.397760
GOOG 0.409961 1.000000 0.399543 0.455731
IBM 0.393292 0.399543 1.000000 0.500448
MSFT 0.397760 0.455731 0.500448 1.000000
In[76]: returns.cov()

Out[73]:
AAPL GOOG IBM MSFT
AAPL 0.000284 0.000112 0.000081 0.000099
GOOG 0.000112 0.000263 0.000079 0.000109
IBM 0.000081 0.000079 0.000148 0.000090
MSFT 0.000099 0.000109 0.000090 0.000219
```

3-1. 상관관계와 공분산

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

```
In[77]: returns.corrwith(returns.IBM)

Out[74]:
AAPL 0.393292
GOOG 0.399543
IBM 1.000000
MSFT 0.500448
dtype: float64
```

3-1. 상관관계와 공분산

- DataFrame을 넘기면 맞아 떨어지는 칼럼의 이름에 대한 상관관계를 계산한다. 여기서는 시가 총액의 퍼센트 변화율에 대한 상관관계를 계산해보았다.

```
In[78]: returns.corrwith(volume)

Out[75]:
AAPL -0.083129
GOOG -0.003435
IBM -0.200958
MSFT -0.084664
dtype: float64
```

- axis=1 옵션을 넘기면 각 칼럼에 대한 상관관계와 공분산을 계산한다. 모든 경우 데이터는 상관관계를 계산하기 전에 색인의 이름 순서대로 정렬된다.

3-2. 유일 값, 값, 세기, 멤버십

- 또 다른 종류의 메서드로는 1차원 Series에 담긴 값의 정보를 추출하는 메서드가 있다. 다음 예제를 살펴보자.

```
In[79]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
In[80]: uniques = obj.unique()
In[81]: uniques
Out[78]: array(['c', 'a', 'd', 'b'], dtype=object)
```

- 유일 값은 정렬된 순서로 반환되지 않지만 필요하다면 uniques.sort()를 이용해서 나중에 정렬 할 수 도 있다. 그리고 value_counts 는 Series에서 도수를 계산하여 반환한다.

```
In[82]: obj.value_counts()
Out[79]:
c 3
a 3
b 2
d 1
dtype: int64
```

- value_counts에서 반환하는 Series는 담고 있는 값으로 내림차순 정렬된다. 또한 value_counts 메서드는 pandas의 최상위 메서드로, 어떤 배열이나 순차 자료 구조에서도 사용할 수 있다.

3-2. 유일 값, 값, 세기, 멤버십

- 마지막으로 isin 메서드는 어떤 값이 Series에 있는지 나타내는 불리언 벡터를 반환하는데, Series나 DataFrame의 칼럼에서 값을 골라내고 싶을 때 유용하게 사용할 수 있다.

```
In[87]: mask = obj.isin(['b', 'c'])

In[88]: mask

Out[85]:
0 True
1 False
2 False
3 False
4 False
5 True
6 True
7 True
8 True
dtype: bool
```

```
In[89]: obj[mask]

Out[86]:
0 c
5 b
6 b
7 c
8 c
dtype: object
```

3-2. 유일 값, 값, 세기, 멤버십

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

```
In[91]: data = DataFrame({'Qu1' : [1, 3, 4, 3, 4],
'Qu2' : [2, 3, 1, 2, 3],
'Qu3' : [1, 5, 2, 4, 4]})
data

Out[88]:
Qu1 Qu2 Qu3
0 1 2 1
1 3 3 5
2 4 1 2
3 3 2 4
4 4 3 4
```

4. 누락된 데이터 처리하기

- 누락된 데이터를 처리하는 일은 데이터 분석 애플리케이션에서 흔히 있는 일이다. pandas의 설계 목표 중 하나는 누락 데이터를 가능한 한 쉽게 처리할 수 있도록 하는 것이다. 예를 들어 앞에서 살펴봤듯이 pandas 객체의 모든 기술통계는 누락된 데이터를 배제하고 처리한다.
- pandas는 누락된 데이터를 실수든 아니든 모두 NaN(Not a Number)으로 취급한다. 그래서 누락된 값을 쉽게 찾을 수 있게 하는 파수병 역할을 한다.

```
In[96]: string_data =
Series(['aardvark', 'artichoke', np.nan, 'avocado'])
In[97]: string_data
Out[94]:
0 aardvark
1 artichoke
2 NaN
3 avocado
dtype: object
In[98]: string_data.isnull()
Out[95]:
0 False
1 False
2 True
3 False
dtype: bool
```

4. 누락된 데이터 처리하기

- 파이썬의 내장 None 값 또한 NA 값으로 취급된다.

```
In[99]: string_data[0] = None

In[100]: string_data.isnull()

Out[97]:
0 True
1 False
2 True
3 False
dtype: bool
```

- pandas에서 NA 값을 표기하는 것이 최선이라는 주장을 하려는 것은 아니지만 pandas에서 사용하는 방법이 더 간단하고 일관적이다. 성능 면에서도 훌륭하며 NumPy 자료형에는 존재하지 않는 진짜 NA 자료형이나 비트 패턴 위에서 만든 간단한 API를 제공한다. NumPy는 계속 개발 중인 프로젝트이므로 앞으로 변경될 가능성이 있다

4.1 누락된 데이터 골라내기

- 누락된 데이터를 골라내는 방법에는 여러 가지가 있는데, 직접 손으로 제거하는 것도 한 방법이지만 dropna를 사용하는 것도 매우 유용한 방법이다 . Series에 대해 dropna 메서드를 적용하면 실제 데이터가 들어있는 색인 값과 데이터를 Series 값으로 반환한다.

```
In[101]: from numpy import nan as NA
In[102]: data = Series([1, NA, 3.5, NA, 7])
In[103]: data.dropna()
Out[100]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

4-1. 누락된 데이터 골라내기

- 불리언 색인을 이용해서 직접 계산하는 것도 물론 가능하다.

```
In[104]: data[data.notnull()]

Out[101]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

4-1. 누락된 데이터 골라내기

- DataFrame 객체의 경우는 조금 복잡한데, 모두 NA인 로우나 칼럼을 제외하든가 하나라도 NA인 값을 포함하고 있는 로우나 칼럼을 제외시킬 수도 있다. dropna는 기본적으로 NA 값이 하나라도 있는 로우는 제외시킨다.

```
In[106]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
[NA, NA, NA], [NA, 6.5, 3.]])

In[107]: Cleaned = data.dropna()
In[108]: data
Out[105]:
0 1 2
0 1 6.5 3
1 1 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3
In[109]: Cleaned
Out[106]:
0 1 2
0 1 6.5 3
```

4-1. 누락된 데이터 골라내기

- how='all' 옵션을 주면 모든 값이 NA인 로우만 제외시킨다.

```
In[110]: data.dropna(how='all')
Out[107]:
0 1 2
0 1 6.5 3
1 1 NaN NaN
3 NaN 6.5 3
```

4-1. 누락된 데이터 골라내기

- 칼럼을 제외시키는 방법은 옵션으로 axis=1을 주면 로우를 제외시키는 것과 동일한 방식으로 동작한다.

```
In[118]: data[4] = NA

In[119]: data

Out[116]:
0 1 2 4
0 1 6.5 3 NaN
1 1 NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN 6.5 3 NaN
In[120]: data.dropna(axis=1, how='all')

Out[117]:
0 1 2
0 1 6.5 3
1 1 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3
```

4-1. 누락된 데이터 골라내기

- DataFrame의 로우를 제외시키는 방법은 주로 시계열 데이터에 사용되는 경향이 있다. 몇 개 이상의 값이 들어있는 로우만 살펴보고 싶다면 thresh 인자에 원하는 값을 넘기면 된다.
- df = DataFrame(np.random.randn(7, 3))

```
In[123]: df

Out[120]:
0 1 2
0 0.674988 NaN NaN
1 0.206816 NaN NaN
2 1.890366 NaN NaN
3 1.207395 NaN 0.513191
4 -1.933233 NaN 1.315672
5 -0.743783 0.009746 0.478140
6 0.945863 0.781102 -1.704737
In[124]: df.dropna(thresh=3)

Out[121]:
0 1 2
5 -0.743783 0.009746 0.478140
6 0.945863 0.781102 -1.704737
```

4.2 누락된 값 채우기

- 누락된 값을 제외시키지 않고 (잠재적으로 다른 데이터도 함께 버려질 가능성이 있다.) 데이터 상의 '구멍'을 어떻게는 메우고 싶은 경우가 있는데, 이런 경우에는 fillna 메서드를 활용하면 된다. 즉, fillna 메서드에 채워 넣고 싶은 값을 넘겨주면 된다.

```
In[127]: df.fillna(0)

Out[124]:
0 1 2
0 0.674988 0.000000 0.000000
1 0.206816 0.000000 0.000000
2 1.890366 0.000000 0.000000
3 1.207395 0.000000 0.513191
4 -1.933233 0.000000 1.315672
5 -0.743783 0.009746 0.478140
6 0.945863 0.781102 -1.704737
```

4.2 누락된 값 채우기

- fillna에 사전 값을 넘겨서 각 칼럼마다 다른 값을 채워 넣을 수도 있다.

```
In[128]: df.fillna({1: 0.5, 3: -1})

Out[125]:
0 1 2
0 0.674988 0.500000 NaN
1 0.206816 0.500000 NaN
2 1.890366 0.500000 NaN
3 1.207395 0.500000 0.513191
4 -1.933233 0.500000 1.315672
5 -0.743783 0.009746 0.478140
6 0.945863 0.781102 -1.704737
```

4.2 누락된 값 채우기

- fillna는 새로운 객체를 반환하지만 다음처럼 기존 객체를 변경할 수도 있다.

```
In[131]: _ = df.fillna(0, inplace=True) # _는 이전에 다루던 객체를 의미

In[132]: df

Out[129]:
0 1 2
0 0.674988 0.000000 0.000000
1 0.206816 0.000000 0.000000
2 1.890366 0.000000 0.000000
3 1.207395 0.000000 0.513191
4 -1.933233 0.000000 1.315672
5 -0.743783 0.009746 0.478140
6 0.945863 0.781102 -1.704737
```

4.2 누락된 값 채우기

- 재색인에서 사용 가능한 보간 메서드는 fillna 메서드에서도 사용이 가능하다.

```
In[133]: df = DataFrame(np.random.randn(6,3))

In[134]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA

In[135]: df

Out[132]:
0 1 2
0 2.321233 0.243797 -0.049593
1 -0.768766 -0.639125 -0.522648
2 0.580564 NaN 0.334274
3 0.757897 NaN -0.482256
4 0.122206 NaN NaN
5 1.561499 NaN NaN
```

```
In[136]: df.fillna(method='ffill')

Out[133]:
0 1 2
0 2.321233 0.243797 -0.049593
1 -0.768766 -0.639125 -0.522648
2 0.580564 -0.639125 0.334274
3 0.757897 -0.639125 -0.482256
4 0.122206 -0.639125 -0.482256
5 1.561499 -0.639125 -0.482256
In[137]: df.fillna(method='ffill', limit = 2)
#두 개까지만 채운다. 남용금지?

Out[134]:
0 1 2
0 2.321233 0.243797 -0.049593
1 -0.768766 -0.639125 -0.522648
2 0.580564 -0.639125 0.334274
3 0.757897 -0.639125 -0.482256
4 0.122206 NaN -0.482256
5 1.561499 NaN -0.482256
```

4.2 누락된 값 채우기

- 조금만 창의적으로 생각하면 fillna를 이용해서 매우 다양한 일을 할 수 있는데, 예를 들면 Series의 평균 값이나 중간 값을 전달할 수도 있다.

```
In[138]: data = Series([1., NA, 3.5, NA, 7])

In[139]: data.fillna(data.mean())

Out[136]:
0 1.000000
1 3.833333
2 3.500000
3 3.833333
4 7.000000
dtype: float64
```

5. 계층적 색인

- 계층적 색인은 pandas의 중요한 기능으로, 축에 대해 다중 색인 단계를 지정할 수 있도록 해준다. 약간 추상적으로 말하면 ㅏ원이 높은 데이터를 낮은 차원의 형식으로 다룰 수 있게 해주는 기능이다. 간단한 예제를 하나 살펴보자. 우선 리스트를 담고 있는 리스트나 배열을 가진 Series하나를 생성하자.

```
In[140]: data = Series(np.random.randn(10),
index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd','d'],[1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])

In[141]: data

Out[138]:
a 1 0.750448
2 -0.823054
3 0.562675
b 1 -0.629406
2 -1.099839
3 1.614658
c 1 -1.297442
2 0.232058
d 2 -0.753372
3 -0.176555
dtype: float64
```

5. 계층적 색인

- 지금 생성한 객체는 MultiIndex를 색인으로 하는 Series로, 색인의 계층을 보여주고 있다. 바로 위 단계의 색인을 이용해서 하위 계층을 직접 접근할 수 있다.

```
In[142]: data.index

Out[139]:
MultiIndex(levels=[[u'a', u'b', u'c', u'd'], [1, 2, 3]],
labels=[[0, 0, 0, 1, 1, 1, 2, 2, 3, 3], [0, 1, 2, 0, 1, 2, 0, 1, 1, 2]])
```

5. 계층적 색인

- 계층적으로 색인된 객체는 데이터의 부분집합을 부분적 색인으로 접근 하는 것이 가능하다.

```
In[143]: data['b']

Out[140]:
1 -0.629406
2 -1.099839
3 1.614658
dtype: float64
In[144]: data['b':'c']

Out[141]:
b 1 -0.629406
2 -1.099839
3 1.614658
c 1 -1.297442
2 0.232058
dtype: float64
```

```
In[145]: data.ix[['b', 'd']]

Out[142]:
b 1 -0.629406
2 -1.099839
3 1.614658
d 2 -0.753372
3 -0.176555
dtype: float64
```

5. 계층적 색인

- 하위 계층의 객체를 선택하는 것도 가능하다.

```
In[146]: data[:, 2]

Out[143]:
a -0.823054
b -1.099839
c 0.232058
d -0.753372
dtype: float64
```

5. 계층적 색인

- 계층적인 색인은 데이터를 재형성하고 피벗 테이블 생성 같은 그룹 기반의 작업을 할 때 중요하게 사용된다. 예를 들어 위에서 만든 DataFrame 객체에 unstack 메서드를 사용해서 데이터를 새롭게 배열할 수 도 있다.

5. 계층적 색인

- unstack의 반대되는 작업은 stack메서드로 수행한다.
- stack과 unstack 메서드는 7장에서 더 자세히 알아보기로 하자.
- DataFrame에서는 두 축 모두 계층적 색인을 가질 수 있다.

```
In[149]: frame = DataFrame(np.arange(12).reshape((4,3)),
index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
columns=[['ohio', 'Ohio', 'Colorado',],
['Green', 'Red', 'Green']])

In[150]: frame

Out[147]:
ohio Ohio Colorado
Green Red Green
a 1 0 1 2
2 3 4 5
b 1 6 7 8
2 9 10 11
```

5. 계층적 색인

- 계층적 색인의 각 단계는 이름(문자열이나 어떤 파이썬 객체라도 가능하다)을 가질 수 있고, 만약 이름이 있다면 콘솔 출력 시에 함께 나타난다. (색인의 이름과 축의 라벨을 혼동하지 말자!)

```
In[151]: frame.index.names = ['key1', 'key2']

In[152]: frame.columns.names = ['state', 'color']

In[153]: frame

Out[150]:
state ohio Ohio Colorado
color Green Red Green
key1 key2
a 1 0 1 2
2 3 4 5
b 1 6 7 8
2 9 10 11
```

5. 계층적 색인

- 칼럼의 부분집합을 부분적 색인으로 접근하는 것도 로우에 대한 부분적 색인과 비슷하게 사용하면 된다.

```
In[154]: frame['Ohio']

Out[151]:
color Red
key1 key2
a 1 1
2 4
b 1 7
2 10
```

5. 계층적 색인

- MultiIndex는 따로 생성한 다음에 재사용이 가능하다. 위에서 살펴본 DataFrame의 칼럼 계층의 이름은 다음처럼 생성할 수 있다.

```
In[162]: pd.MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'],
['Green', 'Red', 'Green']],
names=['state', 'color'])

Out[159]:
MultiIndex(levels=[[u'Colorado', u'Ohio'], [u'Green', u'Red']],
labels=[[1, 1, 0], [0, 1, 0]],
names=[u'state', u'color'])
```

5.1 계층 순서 바꾸고 정렬하기

- 계층적 색인에서 계층 순서를 바꾸거나 지정된 계층에 따라 데이터를 정렬해야 하는 경우도 있다. swaplevel은 넘겨받은 2개의 계층 버호나 이름이 뒤바뀐 새로운 객체를 반환한다(하지만 데이터는 변경되지 않는다.)

```
In[164]: frame.swaplevel('key1', 'key2')

Out[161]:
state ohio Ohio Colorado
color Green Red Green
key2 key1
1 a 0 1 2
2 a 3 4 5
1 b 6 7 8
2 b 9 10 11
```

5.1 계층 순서 바꾸고 정렬하기

- 반면에 sortlevel 메서드는 단일 계층에 속한 데이터를 정렬한다. swaplevel을 사용해서 계층을 바꿀 때 대개는 sortlevel을 사용해서 결과도 사전식으로 정렬한다.

```
In[165]: frame.sortlevel(1)

Out[162]:
state ohio Ohio Colorado
color Green Red Green
key1 key2
a 1 0 1 2
b 1 6 7 8
a 2 3 4 5
b 2 9 10 11
```

```
In[166]: frame.swaplevel(0, 1).sortlevel(0)

Out[163]:
state ohio Ohio Colorado
color Green Red Green
key2 key1
1 a 0 1 2
b 6 7 8
2 a 3 4 5
b 9 10 11
```

5.2 단계별 요약통계

- DataFrame과 Series의 많은 기술통계와 요약통계는 level 옵션을 가지고 있는데, 이는 어떤 한 축에 대해 합을 구하고 싶은 단계를 지정할 수 있는 옵션이다. 앞에서 살펴본 DataFrame에서 로우나 칼럼을 아래처럼 단계별로 정렬하여 합을 구할 수 있다.

```
In[167]: frame.sum(level = 'key2')

Out[164]:
state ohio Ohio Colorado
color Green Red Green
key2
1 6 8 10
2 12 14 16
```

```
In[168]: frame.sum(level='color', axis=1)

Out[165]:
color Green Red
key1 key2
a 1 2 1
2 8 4
b 1 14 7
2 20 10
```

- 내부적으로는 pandas의 groupby기능을 이용해서 구현함

5.2 DataFrame의 칼럼 사용하기

- DataFrame에서 로우를 선택하기 위한 색인으로 하나 이상의 칼럼을 사용하는 것은 드물지 않은 일이다. 아니면 로우의 색인을 DataFrame의 칼럼으로 옮기고 싶을 것이다. 다음과 같은 DataFrame이 있다.

```
In[169]: frame = DataFrame({'a': range(7), 'b':range(7, 0, -1),
'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
'd': [0,1, 2, 0, 1, 2, 3]})

In[170]: frame

Out[167]:
a b c d
0 0 7 one 0
1 1 6 one 1
2 2 5 one 2
3 3 4 two 0
4 4 3 two 1
5 5 2 two 2
6 6 1 two 3
```

5.2 DataFrame의 칼럼 사용하기

- DataFrame의 set_index 함수는 하나 이상의 칼럼을 색인으로 하는 새로운 DataFrame을 생성한다.

```
In[171]: frame2 = frame.set_index(['c', 'd'])

In[172]: frame2

Out[169]:
a b
c d
one 0 0 7
1 1 6
2 2 5
two 0 3 4
1 4 3
2 5 2
3 6 1
```

5.2 DataFrame의 칼럼 사용하기

- 다음처럼 칼럼을 명시적으로 남겨두지 않으면 DataFrame에서 삭제된다.

```
In[173]: frame.set_index(['c', 'd'], drop=False)

Out[170]:
a b c d
c d
one 0 0 7 one 0
1 1 6 one 1
2 2 5 one 2
two 0 3 4 two 0
1 4 3 two 1
2 5 2 two 2
3 6 1 two 3
```

5.2 DataFrame의 칼럼 사용하기

- 반면에 reset_index 함수는 set_index와 반대되는 개념으로, 계층적 색인 단계가 칼럼으로 이동한다.

```
In[174]: frame2.reset_index()

Out[171]:
c d a b
0 one 0 0 7
1 one 1 1 6
2 one 2 2 5
3 two 0 3 4
4 two 1 4 3
5 two 2 5 2
6 two 3 6 1
```

6. Pandas의 기타기능

6.1 정수 색인

- pandas 객체를 정수로 색인해서 사용하는 일은 파이썬에서 리스트나 튜플 같은 기본 자료 구조에서 사용되는 색인의 의미와 약간 달라서 초보자들은 종종 실수를 한다.

```
In[175]: ser = Series(np.arange(3.))
```

```
ser[-1]  # 오류
```

- 이때 pandas는 정수 색인에 대한 '대비책'을 세울 수 있지만 알아내기 쉽지 않은 버그가 생기지 않을 만한 안전하고 일반적인 방법은 내가 알기론 없다. 여기 ser 객체는 0, 1, 2 색인을 가지고 있지만 사용자가 원하는 것이 위치 색인인지 이름 색인인지 알아 맞히는 것은 어려운 일이다.

```
In[177]: ser

Out[174]:
0 0
1 1
2 2
dtype: float64
```

5.2 DataFrame의 칼럼 사용하기

- 반면에 정수 색인이 아니라면 그리 어렵지는 않다.

```
In[178]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])

In[179]: ser2[-1]

Out[176]: 2.0
```

- 일관성을 유지하기 위해 색인 값을 가진 축 색인이 있을 경우 정수 데이터는 항상 이름을 지향한다. 이는 ix 슬라이스에도 마찬가지로 적용된다.

```
In[180]: ser.ix[:1]

Out[177]:
0 0
1 1
dtype: float64
```

- 만일 색인의 종류에 상관없이 위치 기반의 색인이 필요하다면 Series의 iget_value 메서드와 DataFrame의 irow, icol 메서드를 사용하면 된다.

6.2 Panel 데이터

- pandas에는 Panel 이라고 하는 자료 구조가 있는데, Panel은 DataFrame의 3차원 버전이라고 이해하면 된다.

- 형식의 데이터를 다루는 데 초점을 맞추고 있고 계층적 색인을 이용하면 대개의 경우 N차원 배열은 불필요하다. Panel은 DataFrame 객체를 담고 있는 사전이나 3차원 ndarray를 통해 생성할 수 있다.

```
import pandas.io.data as web

pdata = pd.Panel(dict((stk, web.get_data_yahoo(stk)) for stk in ['AAPL',
'GOOG', 'MSFT', 'DELL']))
```

- Panel의 각 항목(DataFrame에서 칼럼이라고 생각하면 된다)은 DataFrame이다.

Panda’s 시계열

6.2 Panel 데이터

Scipy

1. Scipy 라이브러리

- SciPy는 파이썬을 기반으로 하여 과학, 분석, 그리고 엔지니어링을 위한 과학(계산)적 컴퓨팅 영역의 여러 기본적인 작업을 위한 라이브러리(패키지 모음)입니다. Scipy는 기본적으로 Numpy, Matplotlib, pandas, Sympy등 과 함께 동작을 합니다. SciPy는 수치적분 루틴과 미분방정식 해석기, 방정식의 근을 구하는 알고리즘, 표준 연속/이산 확률분포와 다양한 통계관련 도구 등을 제공합니다. NumPy와 Scipy를 함께 사용하면 확장 애드온을 포함한 MATLAB을 완벽하게 대체합니다.

- [Reference] http://www.scipy.org/getting-started.html

엘리 브레설트,SciPy와 NumPy,한빛미디어(2013) Wes McKinney,Python for Data Analysis,O'Reilly Media(2012)

2. Scipy의 기능

- scipy.integrate: 수치적분 루틴과 미분방정식 해법기
- scipy.linalg: numpy.linalg에서 제공하는 것보다 더 확장된 선형대수 루틴과 매트릭스 분해
- scipy.optimize: 함수 최적화기와 방정식의 근을 구하는 알고리즘
- scipy.signal: 시그널 프로세싱 도구
- scipy.sparse: 희소 행렬과 희소 선형 시스템 풀이법
- scipy.special: 감마 함수처럼 흔히 사용되는 수학 함수를 구현한 포트란 라이브러리인 SPECFUN 확장
- scipy.stats: 표준 연속/이산 확률 분포(집적도 함수, 샘플러, 연속 분포 함수)와 다양한 통계 테스트, 그리고 좀 더 기술적인 통계 도구
- scipy.weave: 배열 계산을 빠르게 하기 위해 인라인 C++ 코드를 사용하는 도구

수고하셨습니다