Pandas 설치 및 사용

아주 기본적인 수준에서 보면 Pandas 객체는 행과 열이 단순 정수형 인덱스가 아닌 레이블로 식별되는 NumPy 의 구조화된 배열을 보강한 버전이라고 볼 수 있다. 앞으로 설명하겠지만 Pandas 는 이 기본 자료구조에 추가로 여러 가지 유용한 도구와 메서드, 기능을 제공하지만, 이에 대한 거의 모든 내용을 이해하려면 이 구조가 무엇인지에 대한 이해가 뒷받침돼야 한다. 따라서 추가로 더 설명하기전에 Pandas 의 세 가지 기본 자료구조인 series 와 DataFrame, Index 에 대해 알아보자.

먼저 표준 NumPy 와 Pandas 를 임포트하는 것으로 코드 세션을 시작해 보자.

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

Pandas Series 객체

Pandas Series 는 인덱싱된 데이터의 1 차원 배열이다. 그것은 다음과 같이 리스트나 배열로부터 만들 수 있다.

```
In [2]:data = pd.Series([0.25, 0.5, 0.75, 1.0])
       data
Out[2]:0    0.25
       1    0.50
       2    0.75
       3    1.00
       dtype: float64
```

보다시피 Series 는 일련의 값과 인덱스를 모두 감싸고 있으며, 각각 values 와 index 속성으로 접근할 수 있다.

```
In [3]:data.values
Out[3]:array([ 0.25,  0.5 ,  0.75,  1.  ])
```

Index 는 pd.Index 타입의 배열과 비슷한 객체이다..

```
In [4]:data.index
Out[4]:RangeIndex(start=0, stop=4, step=1)
```

NumPy 배열과 마찬가지로 데이터는 친숙한 파이썬 대괄호 표기법을 통해 연결된 인덱스로 접근할 수 있다.

```
In [5]:data[1]
Out[5]:0.5
In [6]:data[1:3]
Out[6]:1    0.50
```

```
2    0.75
dtype: float64
```

그러나 Pandas Series 가 1 차원 NumPy 배열보다 훨씬 더 일반적이고 유연하다는 것을 알게 될 것이다..

Series : 일반화된 NumPy 배열

지금까지 살펴본 내용으로 Series 객체가 기본적으로 1 차원 NumPy 배열과 호환될 것처럼 보일 수 있다. 근본적인 차이는 인덱스 존재 여부에 있다. NumPy 배열에는 값에 접근하는데 사용되는 암묵적으로 정의된 정수형 인덱스가 있고, Pandas Series 에는 값에 연결된 명시적으로 정의된 인덱스가 있다.

이 명시적인 인덱스 정의는 Series 객체에 추가적인 기능을 제공한다. 예를 들어 인덱스는 정수일 필요가 없고 어떤 타입의 값으로도 구성할 수 있다. :

```
In [7]:data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
        data
Out[7]:a    0.25
       b    0.50
       c    0.75
       d    1.00
       dtype: float64
```

그리고 예상한 대로 항목에 접근할 수 있다.

```
In [8]:data['b']
Out[8]:0.5
```

인접하지 않거나 연속적이지 않은 인덱스를 사용할 수도 있다.:

```
In [9]:data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=[2, 5, 3, 7])
        data
Out[9]:2    0.25
       5    0.50
       3    0.75
       7    1.00
       dtype: float64
In [10]:data[5]
Out[10]:0.5
```

Series : 특수한 딕셔러리

딕셔러리는 일련의 임의의 값에 임의의 키를 매핑하는 구조고 Series 는 타입이 지정된 키를 일련의 타입이 지정된 값에 매핑하는 구조라고 생각하면 Pandas Series 를 파이썬 딕셔러리의 특수한 버전 정도로

여길 수도 있다. 타입이 지정된다는 것이 중요한데, 특정 연산에서 NumPy 배열 뒤의 타입 특정 컴파일된 크기가 그것을 파이썬 리스트보다 더 효율적으로 만들어주는 것처럼 Pandas Series 의 타입 정보는 특정 연산에서 파이썬 딕셔러리보다 Pandas Series 를 훨씬 더 효율적으로 만든다.

파이썬 딕셔러리에서 직접 Series 객체를 구성함으로써 딕셔러리로서의 Series 의 의미를 더욱 분명하게 할 수 있다.

```
In [11]: population_dict = {'California': 38332521,
                            'Texas': 26448193,
                            'New York': 19651127,
                            'Florida': 19552860,
                            'Illinois': 12882135}
         population = pd.Series(population_dict)
         population
Out[11]: California    38332521
         Florida       19552860
         Illinois      12882135
         New York      19651127
         Texas         26448193
         dtype: int64
```

기본적으로 Series 는 인덱스가 정렬된 키에서 추출된 키에서 추출되는 경우에 생성된다. 거기서부터 전형적인 딕셔러리 스타일로 아이템에 접근할 수 있다.

```
In [12]: population['California']
Out[12]: 38332521
```

그러나 딕셔너리와 달리 Series 는 슬라이싱 같이 배열 스타일의 연산도 지원한다.

```
In [13]: population['California':'Illinois']
Out[13]: California    38332521
         Florida       19552860
         Illinois      12882135
         dtype: int64
```

.

Series 객체 구성하기

앞에서 이미 Pandas Series 객체를 처음부터 생성하는 몇 가지 방법을 살펴봤다. 그 방식들은 모두 다음과 같은 형태를 따른다.

```
>>> pd.Series(data, index=index)
```

여기서는 index 는 선택 인수고 data 는 많은 요소 중 하나일 수 있다.

예를 들어 data 는 리스트나 NumPy 배열일 수 있고, 그런 경우 index 는 정수가 기본이다.

```
In [14]:pd.Series([2, 4, 6])
Out[14]:0    2
        1    4
        2    6
       dtype: int64
```

data 는 지정된 인덱스를 채우기 위해 반복되는 스칼라값일 수 있다.

```
In [15]:pd.Series(5, index=[100, 200, 300])
Out[15]:100    5
        200    5
        300    5
        dtype: int64
```

data 는 딕셔너리일 수도 있는데, 그 경우 index 는 기본적으로 딕셔너리 키를 정렬해서 취한다.

```
In [16]:pd.Series({2:'a', 1:'b', 3:'c'})
Out[16]:1    b
        2    a
        3    c
       dtype: object
```

각각의 경우, 다른 결과를 얻고 싶으면 인덱스를 명시적으로 설정할 수 있다.

```
In [17]:pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
Out[17]:3    c
       2    a
       dtype: object
```

이 경우에는 Series 를 명시적으로 정의된 키로만 채울 수 있다.

Pandas DataFrame 객체

다음으로 다룰 Pandas 의 기본 구조체는 DataFrame 이다. Series 객체와 마찬가지로 DataFrame 또한 NumPy 배열의 일반화된 버전이나 파이썬 딕셔너리의 특수한 버전으로 생각할 수 있다.

DataFrame : 일반화된 NumPy 배열

Series 가 유연한 인덱스를 가지는 1 차원 배열이라면 DataFrame 은 유연한 행 인덱스와 유연한 열 이름을 가진 2 차원 배열이라고 볼 수 있다. 2 차원 배열을 정렬된 1 차원 배열의 연속으로 볼 수 있듯이 DataFrame 은 정렬된 Series 객체의 연속으로 볼 수 있다.

설명을 위해 우선 앞에서 사용한 미국 다섯 개 주의 면적을 열거한 새로운 Series 를 구성하자.

```
In [18]: area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                      'Florida': 170312, 'Illinois': 149995}
         area = pd.Series(area_dict)
         area
Out[18]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         dtype: int64
```

이제 앞에서 구성했던 population Series 와 함께 이 area Series 객체도 가지게 됐으니 딕셔너리를 사용해 이 정보를 포함하는 하나의 2 차원 객체를 구성할 수 있다.

```
In [19]: states = pd.DataFrame({'population': population,
                                'area': area})
         states
```

Out[19]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Series 객체가 마찬가지로 DataFrame 도 인덱스 레이블에 접근할 수 있는 index 속성을 가지고 있다.:

```
In [20]: states.index
Out[20]: Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'],
dtype='object')
```

DataFrame 은 열 레이블을 가지고 있는 Index 객체인 column 속성을 가지고 있다.

```
In [21]: states.columns
Out[21]: Index(['area', 'population'], dtype='object')
```

따라서 DataFrame 은 행과 열 모두 데이터 접근을 위한 일반화된 인덱스를 가지고 있는 2 차원 NumPy 배열의 일반화된 버전으로 볼 수 있다.

DataFrame : 특수한 딕셔너리

마찬가지로 DataFrame 을 딕셔너리의 특수 버전으로 볼 수도 있다. 딕셔너리는 키값에 매핑한다면 DataFrame 은 이름을 열 데이터로 이뤄진 Series 에 매핑한다. 예를 들면 'area'속성을 질의하면 앞에서 본 면적을 담고 있는 Series 객체를 반환한다.

```
In [22]:states['area']
Out[22]:California    423967
        Florida      170312
        Illinois     149995
        New York     141297
        Texas        695662
        Name: area, dtype: int64
```

여기가 바로 혼란스러울 수 있는 부분이다. 2 차원 NumPy 배열에서는 Data[0]이 첫번째 행을 반환한다. DataFrame 의 경우에는 Data['col0']이 첫번째 열을 반환한다. 이 때문에 DataFrame 을 일반화 된 배열보다 일반화된 딕셔너리로 보는 것이 더 적합할 수 있지만, 그 상황을 바라보는 두 가지 시각 모두 유용할 수 있다.

DataFrame 객체 구성하기

Pandas DataFrame 은 다양한 방법으로 구성할 수 있다.

단일 Series 객체에서 구성하기

DataFrame 은 Series 객체의 집합체로서 열 하나짜리 DataFrame 은 단일 Series 로부터 구성할 수 있다.

```
In [23]:pd.DataFrame(population, columns=['population'])
```

Out[23]:

	population
California	38332521
Florida	19552860
Illinois	12882135
New York	19651127
Texas	26448193

딕셔너리의 리스트에서 구성하기 딕서너리의 리스트는 DataFrame 으로 만들 수 있다. 여기서는 간단한 리스트 컴프리헨션 을 사용해 데이터를 만들 것이다.:

```
In [24]: data = [{'a': i, 'b': 2 * i}
                 for i in range(3)]
         pd.DataFrame(data)
```

Out[24]:

	a	b
0	0	0
1	1	2
2	2	4

딕셔너리의 일부 키가 누락되더라도 Pandas 는 누락된 자리를 NaN(숫자가 아님을 의미하는 'n ot a number')값으로 채운다.

:

```
In [25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[25]:

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

Series 객체의 딕셔러리에서 구성하기

DataFrame 은 Series 객체의 딕셔너리로 구성될 수 있다.

```
In [26]: pd.DataFrame({'population': population,
                       'area': area})
Out[26]:
```

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

	area	population

2 차원 NumPy 배열에서 구성하기

데이터의 2 차원 배열이 주어지면 지정된 열과 인덱스 이름을 가진 DataFram 을 생성할 수 있다. 만약 생략되면 각각에 대해 정수 인덱스가 사용된다.

```
In [27]: pd.DataFrame(np.random.rand(3, 2),
                      columns=['foo', 'bar'],
                      index=['a', 'b', 'c'])
Out[27]:
```

	foo	Bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

NumPy 구조화된 배열에서 구성하기

Pandas DataFrame 은 구조화된 배열처럼 동작하며 구조화된 배열로부터 직접 만들 수 있다

```
In [28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
         A
Out[28]: array([(0, 0.0), (0, 0.0), (0, 0.0)],
               dtype=[('A', '<i8'), ('B', '<f8')])
In [29]: pd.DataFrame(A)
Out[29]:
```

	A	B
0	0	0.0
1	0	0.0
2	0	0.0

The Pandas Index Object

Series 와 DataFrame 객체가 데이터를 참조하고 수정하게 해주는 명시적인 인덱스를 포함한다는 것을 알았다. Index 객체는 그 자체로 흥미로운 구조체이며 불변의 배열이나 정렬된 집합(Index 객체가

중복되는 값을 포함할 수 있으므로 기술적으로 중복집합)으로 볼 수 있다. 이 관점은 Index 객체에서 사용할 수 있는 연산에 몇가지 흥미로운 결과를 가져온다. 간단한 예로 정수 리스트로부터 Index 를 구성해 보자.

```
In [30]: ind = pd.Index([2, 3, 5, 7, 11])
         ind
Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index : 불변의 배열

Index 객체는여러 면에서 배열처럼 동작한다.예를들어 표준 파이썬 인덱싱 표기법을 사용해 값이나 슬라이스를 가져올 수 있다.

```
In [31]: ind[1]
Out[31]: 3
In [32]: ind[::2]
Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

`Index` 객체에는 NumPy 배열에서 익숙한 속성이 많이 있다.

```
In [33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)
         5 (5,) 1 int64
```

Index 객체와 Numpy 배열의 한 가지 차이점이라면 Index 객체는 일반적인 방법으로는 변경될 수 없는 불편의 값이라는 점이다.

```
In [34]: ind[1] = 0
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0

/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py      in
__setitem__(self, key, value)
   1243
   1244     def __setitem__(self, key, value):
-> 1245         raise TypeError("Index does not support mutable operations")
   1246
   1247     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

이 불변성 덕분에 예기치 않은 인덱스 변경으로 인한 부작용 없이 여러 DataFrame 과 배열 사이에서 인덱스를 더 안전하게 공유할 수 있다.

Index : 정렬된 집합

Pandas 객체는 집합 연산의 여러 측면에 의존하는 데이터세트 간의 조인과 같은 연산을 할 수 있게 하려고 고안됐다. Index 객체는 대체로 파이썬에 내장된 set 데이터 구조에서 사용하는 표기법을 따르기 때문에 합집합, 교집합, 차집합을 비롯해 그 밖의 조합들이 익숙한 방식으로 계산될 수 있다.

```
In [35]: indA = pd.Index([1, 3, 5, 7, 9])
         indB = pd.Index([2, 3, 5, 7, 11])
In [36]: indA & indB  # intersection
Out[36]: Int64Index([3, 5, 7], dtype='int64')
In [37]: indA | indB  # union
Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
In [38]: indA ^ indB  # symmetric difference
Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

이 연산들은 객체 메서드(예, `indA.intersection(indB)`)를 통해서도 접근할 수 있다.