Data Indexing and 선택

Numpy 배열의 값에 접근하고 그 값을 설정하고 수정하는 메서드와 도구에 대해 자세히 알아봤다. 여기서는 인덱싱(예, arr[0, [1, 5]])과 슬라이싱(예: arr[:, 1:5]), 마스킹(예: arr[arr > 0]), 팬시 인덱싱(예: arr[0, [1, 5]]), 그것들의 조합(예: arr[:[1,5]])이 포함된다. 이제 Pandas Series 와 DataFrame 객체의 값에 접근하고 그 값을 수정하는 도구를 살펴보겠다. NumPy 패턴을 사용해 본 적이 있다면 특이점이 몇 가지 있기는 하지만 Pandas 패턴도 아주 친숙하게 느낄 것이다.

Series 에서 데이터 선택

Series 객체는 여러면에서 1 차원 NumPy 배열과 표준 파이썬 딕셔너리처럼 동작한다. 이 둘의 유사점을 기억하고 있으면 배열에서 데이터를 인덱싱하고 선택하는 패턴을 이해하는 데 도움될 것이다.

Series : 딕셔너리

Series 객체는 딕셔너리와 마찬가지로 키의 집합을 값의 집합에 매핑한다.

```
In [1]:import pandas as pd
       data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
       data
Out[1]:a    0.25
       b    0.50
       c    0.75
       d    1.00
       dtype: float64
In [2]:data['b']
Out[2]:0.5
```

키/인텍스와 값을 조사하기 위해 딕셔너리와 유사한 파이썬 표현식과 메서드를 사용할 수도 있다.

```
In [3]:'a' in data
Out[3]:True
In [4]:data.keys()
Out[4]:Index(['a', 'b', 'c', 'd'], dtype='object')
In [5]:list(data.items())
Out[5]:[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

`Series` 객체는 딕셔너리와 유사한 구문을사용해 수정할 수도 있다. 새로운 키에 할당해 딕셔너리를 확정할 수 있는 것과 마찬가지로 새로운 인덱스 값에 할당함으로써 Series 를 확장할 수 있다.

```
In [6]:data['e'] = 1.25
       data
```

```
Out[6]:a    0.25
       b    0.50
       c    0.75
       d    1.00
       e    1.25
       dtype: float64
```

이렇게 객체의 변경이 쉽다는 것은 편리한 특징인데, 그 내부에서 Pandas 가 이 변경에 수반돼야 할 메모리 배치와 데이터 복사에 대한 결정을 수행하므로 일반적으로 사용자는 이러한 이슈에 대해 걱정할 필요가 없다.

Series : 1 차원 배열

Series 는 딕셔너리와 유사한 이터페이스를 기반으로 하며 슬라이스, 마스킹, 팬시 인덱싱 등 NumPy 배열과 똑같은 기본 매커니즘으로 배열 형태의 아이템을 선택할 수 있다.

```
In [7]:# 명시적인 인덱스로 슬라이싱하기
       data['a':'c']
Out[7]:a    0.25
       b    0.50
       c    0.75
       dtype: float64
In [8]:# 명시적 정수 인덱스로 슬라이싱하기
       data[0:2]
Out[8]:a    0.25
       b    0.50
       dtype: float64
In [9]:# 마스킹
       data[(data > 0.3) & (data < 0.8)]
Out[9]:b    0.50
       c    0.75
       dtype: float64
In [10]:# 팬시 인덱싱
       data[['a', 'e']]
Out[10]:a    0.25
       e    1.25
       dtype: float64
```

이 가운데 슬라이싱이 가장 많이 혼동을 일으킬 것이다. 명시적 인덱스(즉, data['a':'c'])로 슬라이싱 할 때는 최종 인덱스가 슬라이스에 포함되지만, 암묵적 인덱스(즉, data[0:2])로 슬라이싱하면 최종 인덱스가 그 슬라이스에서 제외된다는 점을 알아두자.

Indexers: loc, iloc, ix

이 슬라이싱과 인덱싱의 관계적 표기법은 혼동을 불러일으킨다. 가령 Series 가 명시적인 정수 인덱스를 가지고 있다면 `data[1]`과 같은 인덱싱 연산은 명시적인 인덱스를 사용하겠지만 `data[1:3]` 같은 슬라이싱 연산은 파이썬 스타일의 암묵적인덱스를 사용할 것이다.

```
In [11]: data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
         data
Out[11]: 1    a
         3    b
         5    c
         dtype: object
In [12]: # 인덱싱할 때 명시적인 인덱스 사용
         data[1]
Out[12]: 'a'
In [13]: # 슬라이싱할 때 암묵적 인덱스 사용
         data[1:3]
Out[13]: 3    b
         5    c
         dtype: object
```

정수 인덱스를 사용하는 경우 이런 혼선이 발생할 수 있기 때문에 Pandas 는 특정 인덱싱 방식을 명시적으로 드러내는 몇가지 특별한 인덱서(indexer) 속성을 제공한다. 이는 함수 메서드가 아니라 Series 의 데이터에 대한 특정 슬라이싱 인터페이스를 드러내는 속성이다.

먼저 loc 속성은 언제나 명시적인 인덱스를 참조하는 인덱싱과 슬라이싱을 가능하게 한다.

```
In [14]: data.loc[1]
Out[14]: 'a'
In [15]: data.loc[1:3]
Out[15]: 1    a
         3    b
         dtype: object
```

iloc 속성은 인덱싱과 슬라이싱에서 언제나 암묵적인 파이썬 스타일의 인덱스를 참조하게 해준다.

```
In [16]: data.iloc[1]
Out[16]: 'b'
In [17]: data.iloc[1:3]
Out[17]: 3    b
         5    c
         dtype: object
```

세 번째 인덱싱 속성인 ix 는 앞에서 설명한 두 속성의 하이브리드 형태로, Series 객체에 대해서는 표준 []기반의 인덱싱과 동일하다. ix 인덱서의 목적은 곧 논의할 DataFrame 객체에서 더 분명하게 알 수 있다.

파이썬 코드의 한 가지 원칙이라면 '명시적인 것이 암묵적인 것보다 낫다'는 것이다. loc 와 iloc 의 명시적 성격은 명확하고 가독성 있는 코드를 유지하는 데 매우 유용하다. 특히 정수형 인덱스인 경우, 이 두 속성을 사용하는 것이 코드를 읽고 이해하기 쉽게 만들며 뒤섞인 인덱싱/슬라이싱 관계가 초래하는 미묘한 버그를 방지할 수 있다.

DataFrame 에서 데이터 선택

`DataFrame` 은 여러 면에서 2 차원 배열이나 구조화된 배열과 비슷하고, 다른 면에서는 동일 인덱스를 공유하는 Series 구조체의 딕셔너리와 비슷하다. 이 유연성을 기억하고 있으면 이런 구조체에서 데이터를 선택하는 법을 살펴볼 때 도움이 된다.

DataFrame : dictionary

여기서 고려할 첫 번째 유사점은 DataFrame 이 관련 Series 객체의 딕셔너리라는 것이다.

예제 : 미국 주의 면적과 인구 예제

```
In [18]: area = pd.Series({'California': 423967, 'Texas': 695662,
                           'New York': 141297, 'Florida': 170312,
                            'Illinois': 149995})
         pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                           'New York': 19651127, 'Florida': 19552860,
                            'Illinois': 12882135})
         data = pd.DataFrame({'area':area, 'pop':pop})
         data
```

Out[18]:

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

DataFrame 의 열을 이루는 각 Series 는 열 이름으로 된 딕셔너리 스타일의 인덱싱을 통해 접근할 수 있다.

```
In [19]: data['area']
Out[19]: California    423967
         Florida       170312
         Illinois      149995
```

```
New York      141297
Texas         695662
Name: area, dtype: int64
```

마찬가지로 문자열인 열(column)이름을 이용해 속성 스타일로 접근할 수 있다.

```
In [20]:data.area
Out[20]:California    423967
        Florida       170312
        Illinois      149995
        New York      141297
        Texas         695662
        Name: area, dtype: int64
```

속성스타일로 열에 접근하면 사실상 딕셔너리 스타일로 접근하는 것과 똑같은 객체에 접근한다.

```
In [21]:data.area is data['area']
Out[21]:True
```

이 약식 표현이 유용하기는 하지만 모든 경우에 동작하지는 않는다. 예를 들어, 열 이름이 문자열이 아니거나 열이름이 DataFrame 의 메서드와 충돌할 때는 이 속성 스타일로 접근할 수 없다. 예를 들면 `DataFrame` 은 `pop()` 메서드를 가지고 있으므로 `data.pop` 은 `"pop"` 열이 아니라 그 메서드를 가리킬 것이다.:

```
In [22]:data.pop is data['pop']
Out[22]:False
```

특히 속성을 통해 열을 할당하려고 해서는 안된다.

```
In [23]:data['density'] = data['pop'] / data['area']
        data
```

Out[23]:

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

이것은 Series 객체 간에 요소단위로 산술 연산을 하는 간단한 구문이다.

DataFrame : 2 차원 배열

앞에서 언급한 것처럼 DataFrame 을 2 차원 배열의 보강된 버전으로 볼 수도 있다. Values 속성을 이용해 원시 기반 데이터 배열을 확인할 수 있다.:

```
In [24]:data.values
Out[24]:array([[  4.23967000e+05,   3.83325210e+07,   9.04139261e+01],
               [  1.70312000e+05,   1.95528600e+07,   1.14806121e+02],
               [  1.49995000e+05,   1.28821350e+07,   8.58837628e+01],
               [  1.41297000e+05,   1.96511270e+07,   1.39076746e+02],
               [  6.95662000e+05,   2.64481930e+07,   3.80187404e+01]])
```

이 예제를 염두해 두고 있으면 DataFram 자체에 대해 배열에서 익숙했던 많은 유사한 작업을 할 수 있다. 예를 들면, 전체 DataFrame 의 행과 열을 바꿀 수 있다.

```
In [25]:data.T
```

Out[25]:

	California	Florida	Illinois	New York	Texas
Area	4.239670e+05	1.703120e+05	1.499950e+05	1.412970e+05	6.956620e+05
Pop	3.833252e+07	1.955286e+07	1.288214e+07	1.965113e+07	2.644819e+07
Density	9.041393e+01	1.148061e+02	8.588376e+01	1.390767e+02	3.801874e+01

하지만 DataFrame 객체 인덱싱에서는 열을 딕셔너리 스타일로 인덱싱하면 그 객체를 단순히 NumPy 배열로 다룰 수 없게 된다는 것은 확실하다. 특히, 배열에 단일 인덱스를 전달하면 다음과 같이 행에 접근한다.

```
In [26]:data.values[0]
Out[26]:array([  4.23967000e+05,   3.83325210e+07,   9.04139261e+01])
```

그리고 DataFrame 에 단일 '인덱스'를 전달하면 열에 접근한다.:

```
In [27]:data['area']
Out[27]:California    423967
        Florida       170312
        Illinois      149995
        New York      141297
        Texas         695662
        Name: area, dtype: int64
```

따라서 배열 스타일 인덱싱의 경우 다른 표기법이 필요하다. 이때 Pandas 는 다시 언급한 loc, iloc, ix 인덱스를 사용한다. iloc 인덱서를 사용하면 DataFrame 객체가 단순 Numpy 배열인 것 처럼(암묵적 파이썬

스타일의 인덱스 사용)기반 배열을 인덱싱할 수 있지만, DataFrame 인덱스와 열 레이블은 결과에 그대로 유지된다.

```
In [28]: data.iloc[:3, :2]
```

Out[28]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

```
In [29]: data.loc[:'Illinois', :'pop']
```

Out[29]:

	area	pop
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

NumPy 스타일의 익숙한 데이터접근 패턴은 이 인덱서들에서도 사용할 수 있다. 예를 들어, loc 인덱서에서 다음 처럼 마스킹과 팬시 인덱싱을 결합할 수 있다.

```
In [31]: data.loc[data.density > 100, ['pop', 'density']]
```

Out[31]:

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

이 인덱싱 규칙은 값을 설정하거나 변경하는 데도 사용될 수 있다. 이는 NumPy 에서 작업하는데 익숙한 표준 방식으로 이뤄진다.

```
In [32]: data.iloc[0, 2] = 90
         data
```

Out[32]:

	area	pop	density
California	423967	38332521	90.000000
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746

Out[32]:

	area	pop	density
Texas	695662	26448193	38.018740

Pandas 에서 데이터 가동을 능숙하게 하려면 간단한 DataFrame 에 시간을 투자해서 다양한 인덱싱 기법이 제공하는 인덱싱, 슬라이싱, 마스킹, 팬시 인덱싱 유형을 알아보는 것이 좋다.

추가적인 인덱싱 규칙

앞의 내용과 전혀 다르게 보일지도 모르지만, 실무에서 매우 유용한 몇 가지 추가적인 인덱싱 규칙이 있다.

우선 인덱싱은 열을 참조하는 반면, 슬라이싱은 행을 참조한다.

```
In [33]: data['Florida':'Illinois']
```

Out[33]:

	Area	Pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

이 슬라이스는 인덱스 대신 숫자로 행을 참조할 수도 있다.

```
In [34]: data[1:3]
```

Out[34]:

	area	pop	density
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

이와 비슷하게 직접 마스킹 연산은 열 단위가 아닌 행단위로 해석된다.

```
In [35]: data[data.density > 100]
```

Out[35]:

	area	pop	density
Florida	170312	19552860	114.806121
New York	141297	19651127	139.076746

이 두 규칙은 구문적으로 NumPy 배열과 유사하며, Pandas 규칙의 틀에 딱 들어맞지는 않지만 실제로 꽤 유용하다.