고성능 Pandas: eval()과 query()

`query()` 와 `eval()`의 등장 배경: 복합 표현식

NumPy 와 Pandas 가 속도가 빠른 벡터화된 연산을 지원하는 것은 앞에서 봤다.

예를 들어, 두 배열의 요소를 더할 때의 코드는 다음과 같다.

```
In [1]: import numpy as np
        rng = np.random.RandomState(42)
        x = rng.rand(1000000)
        y = rng.rand(1000000)
        %timeit x + y
2.44 ms ± 37.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

NumPy 배열연산에서 유니버설 함수에서 논의했듯이 이렇게 계산하는 것이 다음과 같은 파이썬 루프나 컴프리헨션으로 하는 것보다 훨씬 빠르다.

```
In [2]: %timeit np.fromiter((xi + yi for xi, yi in zip(x, y)), dtype=x.dtype, count=len(x))
364 ms ± 3.71 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

그러나 이 추상화는 복합 표현식을 계산할 때는 효율성이 떨어질 수 있다.

```
In [3]: %timeit mask = (x > 0.5) & (y < 0.5)
1.16 ms ± 13.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

NumPy 는 각 하위 표현식을 평가하기 때문에 위 표현식은 다음 하위 표현식과 거의 동일하다.

```
In [4]: tmp1 = (x > 0.5)
        tmp2 = (y < 0.5)
        %timeit mask = tmp1 & tmp2
        mask = tmp1 & tmp2
80.8 µs ± 474 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

즉, 모든 중간 단계가 명시적으로 메모리에 할당한다. x 와 y 배열의 규모가 매우 크면 메모리와 계산능력에 심각한 오버헤드가 발생할 수 있다. Numexpr 라이브러리를 사용하면 이러한 중간 배열을 할당하지 않고도 요소별로 이러한 유형의 복합 표현식을 계산할 수 있다. Numexpr 문서에서 더 자세한 내용을 확인할 수 있다. 하지만 지금은 이 라이브러리가 계산하고자 하는 NumPy 스타일 표현식을 문자열로 받는다는 사실만 알아도 충분하다.:

```
In [5]: import numexpr
        mask_numexpr = numexpr.evaluate('(x > 0.5) & (y < 0.5)')
        np.allclose(mask, mask_numexpr)
Out[5]: True
```

이 방법의 이점은 Numexpr 가 전체 크기의 임시 배열을 사용하지 않고서 표현식을 평가한다는 점이다. 그래서 특히 큰 배열의 경우에는 이것이 NumPy 보다 훨씬 더 효율적이다. 이어서 다룰 Pandas eval()과 query()도구는 개념적으로 유사하며 Numexpr 패키지에 의존한다..

효율적인 연산을 위한 `pandas.eval()`

Pandas 의 `eval()` 함수는 `DataFrames` 을 사용하는 연산을 효율적으로 계산하기 위해 문자열 표현식을 사용한다.

예를 들어, 다음 `DataFrames` 을 생각해 볼 수 있다

```
In [6]: import pandas as pd
        nrows, ncols = 100000, 100
        rng = np.random.RandomState(42)
        df1, df2, df3, df4 = (pd.DataFrame(rng.rand(nrows, ncols))
                              for i in range(4))
```

전형적인 Pandas 접근 방식을 사용해 네개의 DataFrame 모두의 합을 계산하려면 그 합을 쓰기만 하면된다.

```
In [7]: %timeit df1 + df2 + df3 + df4
        10 loops, best of 3: 87.1 ms per loop
```

표현식을 문자열로 구성함으로써 pd.eval 을 통해 같은 계산 결과를 얻을 수 있다.

```
In [8]: %timeit pd.eval('df1 + df2 + df3 + df4')
        10 loops, best of 3: 42.2 ms per loop
```

이 표현식의 `eval()` 버전은 같은 결과를 주면서 50% 더 빠르고 메모리도 훨씬 적게 사용한다.:

```
In [9]: np.allclose(df1 + df2 + df3 + df4,
                    pd.eval('df1 + df2 + df3 + df4'))
Out[9]: True
```

`pd.eval()` 이 지원하는 연산

Pandas 0.16 버전 기준으로 pd.eval()은 다양한 연산을 지원한다. 이 연산을 보여주기 위해 다음의정수 DataFrame 을 사용할 것이다.

```
In [10]: df1, df2, df3, df4, df5 = (pd.DataFrame(rng.randint(0, 1000, (100, 3)))
                                    for i in range(5))
```

산술연산자: `pd.eval()` 은 모든산술 연산자를 지원한다.

```
In [11]: result1 = -df1 * df2 / (df3 + df4) - df5
         result2 = pd.eval('-df1 * df2 / (df3 + df4) - df5')
         np.allclose(result1, result2)
Out[11]: True
```

비교연산자 : `pd.eval()` 은 연쇄 표현식을 포함한 보든 비교 연산자를 지원한다.

```
In [12]: result1 = (df1 < df2) & (df2 <= df3) & (df3 != df4)
         result2 = pd.eval('df1 < df2 <= df3 != df4')
         np.allclose(result1, result2)
Out[12]: True
```

비트 단위 연산자 : `pd.eval()` 은 `&` 와 `|` 비트 단위 연산자를 지원한다.

```
In [13]: result1 = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)
         result2 = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')
         np.allclose(result1, result2)
Out[13]: True
```

그 밖에 부울 표현식에서 리터널 and 와 or 사용을 지원한다.

```
In [14]: result3 = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')
         np.allclose(result1, result3)
Out[14]: True
```

객체속성과 인덱스 : `pd.eval()` 은 `obj.attr` 구문을 통해 객체 속성에 접근하는 것을 지원하고 obj[index]구문을 통해 인덱스에 접근하는 것을 지원한다,:

```
In [15]: result1 = df2.T[0] + df3.iloc[1]
         result2 = pd.eval('df2.T[0] + df3.iloc[1]')
         np.allclose(result1, result2)
Out[15]: True
```

기타 연산 : 함수 호출, 조건문, 루프를 포함해 그 밖의 복잡한 생성과 같은 다른 연산은 현재 pd.eval()에 구현돼 있지 않다. 이처럼 더 복잡한 유형의 표현식을 실행하고 싶을 때는 Numexpr 라이브러리를 사용하면 된다. .

열 단위의 연산을 위한 `DataFrame.eval()`

Pandas 의 최상위 레벨에 `pd.eval()` 함수가 있듯이 `DataFrames` 에도 비슷한 방식으로 동작하는 `eval()` 메서드가 있다. eval()메서드의 이점은 열을 이름으로 부를 수 있다는 것이다.

레이블을 가진 예제이다.

```
In [16]: df = pd.DataFrame(rng.rand(1000, 3), columns=['A', 'B', 'C'])
         df.head()
Out[16]:
```

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324

	A	B	C
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

위와 같이 `pd.eval()` 을 사용하려면 다음과 같이 세 개의 열이 있는 표현식을 계산할 수 있다.

```
In [17]: result1 = (df['A'] + df['B']) / (df['C'] - 1)
         result2 = pd.eval("(df.A + df.B) / (df.C - 1)")
         np.allclose(result1, result2)
Out[17]: True
```

`DataFrame.eval()` 메서드는 사용하려면 열을 사용하는 표현식을 훨신 더 간결하게 평가할 수 있다.

```
In [18]: result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
Out[18]: True
```

평가된 표현식에서는 열이름 변수로 취급하며 바라는 것을 결과로 얻게 된다.

DataFrame.eval()에서의 할당

방금 이야기한 옵션과 더불어 `DataFrame.eval()` 을사용해 열을 할당할 수도 있다. 앞에서 사용한 'A','B','C'열을 갖는 `DataFrame` 을 사용한다.

```
In [19]: df.head()
Out[19]:
```

	A	B	C
0	0.375506	0.406939	0.069938
1	0.069087	0.235615	0.154374
2	0.677945	0.433839	0.652324
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

새로운 열`'D'`를 생성하고 거기에 다른 열로부터 계산된 값을 할당하는데 df.eval()을 사용할 수 있다.

```
In [20]: df.eval('D = (A + B) / C', inplace=True)
         df.head()
Out[20]:
```

	A	B	C	D
0	0.375506	0.406939	0.069938	11.187620
1	0.069087	0.235615	0.154374	1.973796

	A	B	C	D
2	0.677945	0.433839	0.652324	1.704344
3	0.264038	0.808055	0.347197	3.087857
4	0.589161	0.252418	0.557789	1.508776

같은 방식으로 어느 기존 열이든 수정할 수 있다.

```
In [21]: df.eval('D = (A - B) / C', inplace=True)
         df.head()
Out[21]:
```

	A	B	C	D
0	0.375506	0.406939	0.069938	-0.449425
1	0.069087	0.235615	0.154374	-1.078728
2	0.677945	0.433839	0.652324	0.374209
3	0.264038	0.808055	0.347197	-1.566886
4	0.589161	0.252418	0.557789	0.603708

ataFrame.eval()의 지역 변수

The `DataFrame.eval()` 메서드 지역 파이썬 변수와 함께 작업을 할 수 있도록 추가적인 구문을 지원한다.

```
In [22]: column_mean = df.mean(1)
         result1 = df['A'] + column_mean
         result2 = df.eval('A + @column_mean')
         np.allclose(result1, result2)
Out[22]: True
```

`@` 기호는 열 이름이 아닌 변수 이름을 표시해서 두 개의 '네임스페이스(namespace)', 즉 열의 네임스페이스와 파이썬 객체의 네임스페이스를 포함하는 표현식을 효율적으로 평가할 수 있게 해준다. 이 @ 기호는 pandas.eval()함수가 아닌 DataFrame.eval()메서드에서만 지원되는데, pandas.eval()함수는 하나의 (파이썬) 네임스페이스에만 접근할 수 있기 때문이다.

DataFrame.query() 메서드

`DataFrame` 에는 평가된 문자열을 기반으로 하는 다른 메서드로 query()메서드가 있다.

```
In [23]: result1 = df[(df.A < 0.5) & (df.B < 0.5)]
         result2 = pd.eval('df[(df.A < 0.5) & (df.B < 0.5)]')
         np.allclose(result1, result2)
```

```
Out[23]:True
```

`DataFrame.eval()`을 살펴볼 때 사용했던 예제에서와 마찬가지로 이것은 DataFrame의 열을 포함하는 표현식이다. 그렇지만 그것을 `DataFrame.eval()` 구문을 사용해 표현할 수 없다. 대신 이러한 유형의 필터링 연산에서는 query()메서드를 사용할 수 있다.

```
In [24]:result2 = df.query('A < 0.5 and B < 0.5')
        np.allclose(result1, result2)
Out[24]:True
```

마스킹 표현식에 비해 계산이 더 효율적인 것 이외에도 이 방법이 훨씬 더 읽고 이해하기 쉽다. Query()메서드도 지역 변수를 표시하기 위해 @플래그를 받는다.

```
In [25]:Cmean = df['C'].mean()
        result1 = df[(df.A < Cmean) & (df.B < Cmean)]
        result2 = df.query('A < @Cmean and B < @Cmean')
        np.allclose(result1, result2)
Out[25]:True
```

성능: 이 함수를 사용해야 하는 경우

이 함수의 사용 여부를 고려할 때는 계산 시간과 메모리 사용의 두 가지 사항을 고려해야 한다. 메모리 사용은 가장 예측하기 쉬운 부분이다. 이미 언급했듯이, NumPy 배열이나 Pandas DataFrame을 포함하는 모든 복합 표현식은 암묵적으로 임시 배열을 생성한다.

```
In [26]:x = df[(df.A < 0.5) & (df.B < 0.5)]
```

대략 다음과 같다.

```
In [27]: tmp1 = df.A < 0.5
         tmp2 = df.B < 0.5
         tmp3 = tmp1 & tmp2
         x = df[tmp3]
```

임시 `DataFrame`의크기가 사용 가능한 시스템 메모리(일번적으로 수 기가바이트)에 비해 상당히 크다면 eval()이나 query()표현식을 사용하는 것이 좋다.
다음 코드를 활용해 배열의 대략적인 크기를 바이트 단위로 확인할 수 있다.

```
In [28]:df.values.nbytes
Out[28]:32000
```

성능 측면에서 볼 때 시스템 메모리를 넘어지지 않는다면 eval()이 더 빠를 수 있다. 문제는 임시 DataFrame을 시스템의 1.1 이나 L2 CPU 캐시 규모와 비교하는 방법이 있다. 실제로 전형적인 메서드와 eval/query 메서드간의 계산시간의 차이는 일반적으로 중요하지 않다. 오히려 작은 배열에서는 전현적인 메서드가 더 빠르다. eval/query 의 이점은 주로 메모리를 절약하는 데 있으며, 때때로 구문이 더 깔끔하다는 것이다.