High-Performance Pandas: eval() and query()

高性能Pandas: eval() 和 query()

> As we've already seen in previous sections, the power of the PyData stack is built upon the ability of NumPy and Pandas to push basic operations into C via an intuitive syntax: examples are vectorized/broadcasted operations in NumPy, and grouping-type operations in Pandas. While these abstractions are efficient and effective for many common use cases, they often rely on the creation of temporary intermediate objects, which can cause undue overhead in computational time and memory use.

前面的章节中，我们已经了解了PyData的整个技术栈建立在NumPy和Pandas能将基础的向量化运算使用C底层的方式实现，语法却依然保持简单和直观：例子包括NumPy中的向量化和广播操作，及Pandas的分组类型的操作。虽然这些抽象在很多通用场合下是非常高效的，但是这些操作都涉及到创建临时对象，仍然会产生额外的计算时间和内存占用。

> As of version 0.13 (released January 2014), Pandas includes some experimental tools that allow you to directly access C-speed operations without costly allocation of intermediate arrays. These are the `eval()` and `query()` functions, which rely on the Numexpr package. In this notebook we will walk through their use and give some rules-of-thumb about when you might think about using them.

Pandas在0.13版本（2014年1月发布）加入了一些实验性的工具，能直接进行C底层的运算而不需要创建临时的数组。函数 `eval()` 和 `query()` 具有这个特性，底层是基于Numexpr包构建的。在本节中，我们会简单介绍它们的使用，然后给出何时适合使用它们的基础规则。

Motivating `query()` and `eval()`: Compound Expressions

使用 `query()` 和 `eval()`：复合表达式

> We've seen previously that NumPy and Pandas support fast vectorized operations; for example, when adding the elements of two arrays:

我们已经掌握了NumPy和Pandas能够支持快速向量化操作；例如，当将两个数组进行加法操作时：

In [1]:
```
import numpy as np
rng = np.random.RandomState(42)
x = rng.rand(1000000)
y = rng.rand(1000000)
%timeit x + y
```

```
2.04 ms ± 62.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

> As discussed in Computation on NumPy Arrays: Universal Functions, this is much faster than doing the addition via a Python loop or comprehension:

我们在使用Numpy计算：通用函数中已经讨论过，这种运算对比使用Python循环或列表解析的方法要高效的多：

In [2]:
```
%timeit np.fromiter((xi + yi for xi, yi in zip(x, y)), dtype=x.dtype, count=len(x))
```

```
186 ms ± 14.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

> But this abstraction can become less efficient when computing compound expressions. For example, consider the following expression:

但是当运算变得复杂的情况下，这种向量化运算就会变得没那么高效了。如下例：

In [3]:
```
mask = (x > 0.5) & (y < 0.5)
```

> Because NumPy evaluates each subexpression, this is roughly equivalent to the following:

因为NumPy会独立计算每一个子表达式，因此上面代码等同与下面：

In [4]:
```
tmp1 = (x > 0.5)
tmp2 = (y < 0.5)
mask = tmp1 & tmp2
```

> In other words, *every intermediate step is explicitly allocated in memory*. If the `x` and `y` arrays are very large, this can lead to significant memory and computational overhead. The Numexpr library gives you the ability to compute this type of compound expression element by element, without the need to allocate full intermediate arrays. The Numexpr documentation has more details, but for the time being it is sufficient to say that the library accepts a *string* giving the NumPy-style expression you'd like to compute:

换言之，*每个中间步骤都会显式分配内存*。如果 `x` 和 `y` 数组变得非常巨大，这会带来显著的内存和计算资源开销。Numexpr库提供了既能使用简单语法进行数组的逐元素运算的能力，又不需要为中间步骤数组分配全部内存的能力。Numexpr在线文档中有更加详细的说明，我们现在只需要将它理解为这个库能接受一个NumPy风格的表达式字符串，然后计算得到结果：

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

Out[5]: True

> The benefit here is that Numexpr evaluates the expression in a way that does not use full-sized temporary arrays, and thus can be much more efficient than NumPy, especially for large arrays. The Pandas `eval()` and `query()` tools that we will discuss here are conceptually similar, and depend on the Numexpr package.

这样做的优点是，Numexpr使用的临时数组不是完全分配空间的，并利用这少量数组即能完成计算，因此能比NumPy更加高效，特别是对大的数组来说。我们将会讨论到的Pandas的 `eval()` 和 `query` 工具，就是基于Numexpr包构建的。

`pandas.eval()` for Efficient Operations

`pandas.eval()` 更加高效的运算

> The `eval()` function in Pandas uses string expressions to efficiently compute operations using `DataFrame`s. For example, consider the following `DataFrame`s:

Pandas中的 `eval()` 函数可以使用字符串类型的表达式对 `DataFrame` 进行运算。例如，创建下面的 `DataFrame`：

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))
```

> To compute the sum of all four `DataFrame`s using the typical Pandas approach, we can just write the sum:

要计算所有四个 `DataFrame` 的总和，使用典型的Pandas方式，我们只需要将它们相加：

In [7]:
```
%timeit df1 + df2 + df3 + df4
```

```
72.2 ms ± 8.44 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

> The same result can be computed via `pd.eval` by constructing the expression as a string:

我们也可以使用 `pd.eval`，参数传入上述表达式的字符串形式，计算得到同样的结果：

In [8]:
```
%timeit pd.eval('df1 + df2 + df3 + df4')
```

```
35 ms ± 955 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

> The `eval()` version of this expression is about 50% faster (and uses much less memory), while giving the same result:

`eval()` 版本的计算比典型方法快了接近接近50%（而且使用了更少的内存），我们来使用 `np.allclose()` 函数验证一下结果是否相同：

译者注：50%是按照原文翻译的，在译者自己笔记本上 `eval()` 的运行时间是典型方式的不到一半，运算速度应该是提高了100%多。

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

Out[9]: True

Operations supported by `pd.eval()`

`pd.eval()` 支持的运算

> As of Pandas v0.16, `pd.eval()` supports a wide range of operations. To demonstrate these, we'll use the following integer `DataFrame`s:

到了Pandas 0.16版本，`pd.eval()` 支持很大范围的运算。我们使用下面的整数 `DataFrame` 来进行展示：

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

Arithmetic operators

算术运算

> `pd.eval()` supports all arithmetic operators. For example:

`pd.eval()` 支持所有的算术运算。例如：

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

Out[11]: True

Comparison operators

比较运算

> `pd.eval()` supports all comparison operators, including chained expressions:

`pd.eval()` 支持所有的比较运算，包括链式表达式：

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

Out[12]: True

Bitwise operators

位运算

> `pd.eval()` supports the `&` and `|` bitwise operators:

`pd.eval()` 支持与 `&` 以及或 `|` 位运算符：

译者注：还支持非 `~` 位运算符。

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

> In addition, it supports the use of the literal `and` and `or` in Boolean expressions:

而且，（译者注：对比NumPy）它还支持Python的在布尔表达式中使用逻辑运算 `and` 和 `or`：

译者注：还支持 `not` 逻辑运算。

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

Out[14]: True

Object attributes and indices

对象属性和索引

> `pd.eval()` supports access to object attributes via the `obj.attr` syntax, and indexes via the `obj[index]` syntax:

`pd.eval()` 支持使用 `obj.attr` 语法获取对象属性，也支持使用 `obj[index]` 语法进行索引：

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

Other operations

其他运算

> Other operations such as function calls, conditional statements, loops, and other more involved constructs are currently *not* implemented in `pd.eval()`. If you'd like to execute these more complicated types of expressions, you can use the Numexpr library itself.

其他运算例如函数调用、条件语句、循环以及其他混合结构目前都不被 `pd.eval()` 支持。如果你需要使用这种复杂的表达式，你可以使用Numexpr库本身。

`DataFrame.eval()` for Column-Wise Operations

`DataFrame.eval()` 操作列

> Just as Pandas has a top-level `pd.eval()` function, `DataFrame`s have an `eval()` method that works in similar ways. The benefit of the `eval()` method is that columns can be referred to *by name*. We'll use this labeled array as an example:

Pandas有着顶层的 `pd.eval()` 函数，`DataFrame` 也有自己的 `eval()` 方法，实现的功能类似。使用 `eval()` 方法的好处是可以使用*列名*指代列。我们使用下面的带列标签的数组作为例子说明：

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
3	0.264038	0.808055	0.347197
4	0.589161	0.252418	0.557789

> Using `pd.eval()` as above, we can compute expressions with the three columns like this:

使用上面的 `pd.eval()`，我们可以如下计算三个列的结果：

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

> The `DataFrame.eval()` method allows much more succinct evaluation of expressions with the columns:

使用 `DataFrame.eval()` 方法允许我们采用更加直接的方式操作列数据：

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

Out[18]: True

> Notice here that we treat *column names as variables* within the evaluated expression, and the result is what we would wish.

上面的代码中我们在表达式中将*列名作为变量*来使用，而且结果也是一致的。

Assignment in DataFrame.eval()

DataFrame.eval() 中的赋值

> In addition to the options just discussed, `DataFrame.eval()` also allows assignment to any column. Let's use the `DataFrame` from before, which has columns `'A'`, `'B'`, and `'C'`:

除了上面的操作外，`DataFrame.eval()` 也支持对任何列的赋值操作。还是使用上面的 `DataFrame`，有着 `A`、`B` 和 `C` 三个列：

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

> We can use `df.eval()` to create a new column `'D'` and assign to it a value computed from the other columns:

我们可以使用 `df.eval()` 方法类创建一个新的列 `'D'`，然后将它赋值为其他列运算结果：

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
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

> In the same way, any existing column can be modified:

同样的，已经存在的列可以被修改：

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

Local variables in DataFrame.eval()

DataFrame.eval()中的本地变量

> The `DataFrame.eval()` method supports an additional syntax that lets it work with local Python variables. Consider the following:

`DataFrame.eval()` 方法还支持使用脚本中的本地Python变量。见下例：

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

> The `@` character here marks a *variable name* rather than a *column name*, and lets you efficiently evaluate expressions involving the two "namespaces": the namespace of columns, and the namespace of Python objects. Notice that this `@` character is only supported by the `DataFrame.eval()` *method*, not by the `pandas.eval()` *function*, because the `pandas.eval()` function only has access to the one (Python) namespace.

上面的字符串表达式中的 `@` 符号表示的是一个*变量名称*而不是一个*列名*，这个表达式能高效的计算涉及列空间和Python对象空间的运算表达式。需要注意的是 `@` 符号只能在 `DataFrame.eval()` 方法中使用，不能在 `pandas.eval()` 函数中使用，因为 `pandas.eval()` 实际上只有一个命名空间。

DataFrame.query() Method

DataFrame.query() 方法

> The `DataFrame` has another method based on evaluated strings, called the `query()` method. Consider the following:

`DataFrame` 还有另外一个方法也是建立在字符串表达式运算的基础上的，就是 `query()`。看下面这个例子：

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

> As with the example used in our discussion of `DataFrame.eval()`, this is an expression involving columns of the `DataFrame`. It cannot be expressed using the `DataFrame.eval()` syntax, however! Instead, for this type of filtering operation, you can use the `query()` method:

根据前面的例子和讨论，这是一个涉及 `DataFrame` 列的表达式。但是它却不能使用 `DataFrame.eval()` 来实现。在这种情况下，你可以使用 `query()` 方法：

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

Out[24]: True

> In addition to being a more efficient computation, compared to the masking expression this is much easier to read and understand. Note that the `query()` method also accepts the `@` flag to mark local variables:

除了提供更加高效的计算外，这种语法比遮盖数组的方式更加容易读明白。而且 `query()` 方法也接受 `@` 符号来标记本地变量：

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

Performance: When to Use These Functions

性能：什么时候选择使用这些函数

> When considering whether to use these functions, there are two considerations: *computation time* and *memory use*. Memory use is the most predictable aspect. As already mentioned, every compound expression involving NumPy arrays or Pandas `DataFrame`s will result in implicit creation of temporary arrays: For example, this:

是否使用这些函数主要取决与两个考虑：*计算时间*和*内存占用*。其中最易预测的是内存使用。我们之前已经提到，每个基于NumPy数组的复合表达式都会在每个中间步骤产生一个临时数组，例如：

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

> Is roughly equivalent to this:

等同于：

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

> If the size of the temporary `DataFrame`s is significant compared to your available system memory (typically several gigabytes) then it's a good idea to use an `eval()` or `query()` expression. You can check the approximate size of your array in bytes using this:

如果产生的临时的 `DataFrame` 与你可用的系统内存容量在同一个量级（如数GB）的话，那么使用 `eval()` 或者 `query()` 表达式显然是个好主意。可以通过数组的nbytes属性查看大概的内存占用：

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

Out[28]: 32000

> On the performance side, `eval()` can be faster even when you are not maxing-out your system memory. The issue is how your temporary `DataFrame`s compare to the size of the L1 or L2 CPU cache on your system (typically a few megabytes in 2016); if they are much bigger, then `eval()` can avoid some potentially slow movement of values between the different memory caches. In practice, I find that the difference in computation time between the traditional methods and the `eval`/`query` method is usually not significant–if anything, the traditional method is faster for smaller arrays! The benefit of `eval`/`query` is mainly in the saved memory, and the sometimes cleaner syntax they offer.

至于计算时间考虑，`eval()` 即使在不考虑内存占用的情况下也可能会更快。造成这个差异的原因主要在于临时的 `DataFrame` 的大小与计算机CPU的L1和L2缓存大小（在2016年通常是几个MB）的比值；如果缓存相比而言足够大的话，那么 `eval()` 可以避免在内存和CPU缓存之间的数据复制开销。在实践中，作者发现使用传统方式和 `eval`/`query` 方法之间的计算时间差异通常很小，如果存在的话，传统方法在小尺寸数组的情况下甚至还更快。因此 `eval`/`query` 的优势主要在于节省内存和它们的语法会更加清晰易懂。

> We've covered most of the details of `eval()` and `query()` here; for more information on these, you can refer to the Pandas documentation. In particular, different parsers and engines can be specified for running these queries; for details on this, see the discussion within the "Enhancing Performance" section.

我们在本节讨论了 `eval()` 和 `query()` 的大部分内容；要获取更多相关资源，请参考Pandas的在线文档。特别的，其他不同的解析器和引擎也可以指定运行这些表达式和查询；有关内容参见性能增强章节中的说明。