tive 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.

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.

Motivating query() and eval(): Compound Expressions

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

```
In[1]: import numpy as np
       rng = np.random.RandomState(42)
       x = rng.rand(1E6)
       y = rng.rand(1E6)
       %timeit x + y
```

```
100 loops, best of 3: 3.39 ms per loop
```

As discussed in "Computation on NumPy Arrays: Universal Functions" on page 50, this is much faster than doing the addition via a Python loop or comprehension:

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

```
1 loop, best of 3: 266 ms per loop
```

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

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

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

```
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:

```
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.

pandas.eval() for Efficient Operations

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

```
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:

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

10 loops, best of 3: 87.1 ms per loop
```

We can compute the same result via `pd.eval` by constructing the expression as a string:

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

10 loops, best of 3: 42.2 ms per loop
```

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

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

Out[9]: True
```

Operations supported by pd.eval()

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

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

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