```
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:

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

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

```
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:

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

```
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. 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.

DataFrame.eval() for Column-Wise Operations

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:

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