```
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, `DataFrames` 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()
```

```
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:

```
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:

```
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.

Assignment in DataFrame.eval()

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'`:

```
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:

```
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:

```
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()

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

```
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.

DataFrame.query() Method

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

```
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:

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

```
Out[24]: True
```