asteval documentation

Release 0.9

Matthew Newville

April 09, 2012

CONTENTS

ASTEVAL is a safe(ish) evaluator of Python expressions and statements, using Python's ast module. The idea is to provide a simple, safe, and robust miniature mathematical language that can handle user-input in cases where one might be tempted to use Python's *eval*. The emphasis here is on mathematical expressions, so numpy functions are imported and used if available.

While much of Python's constructs are supported, there are important absences and differences, and this is by no means an attempt to reproduce Python with its own ast module. Important differences and absences are:

1. Variable and function symbol names are held in a simple symbol table (a single dictionary), giving a flat namespace.
2. creating classes is not supported.
3. importing modules is not supported.
4. function decorators, yield, lambda, and exec are not supported.

Many built-in python syntactical components (if-then-else, while loops, for loops, try-except blocks, list comprehension, slicing, subscripting), and built-in data structures (dictionaries, tuple, lists, numpy arrays, strings) are fully supported. In addition, many built-in functions are supported, including the standard builtin python functions, and all mathemetical functions from the math module. As mentioned above, if numpy is available, many of its functions will also be available. Users can define their own functions, but given the restrictions of not being able to define classes or import modules, the language is decidedly limited.

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Prerequisites

The asteval package requires Python 2.6 and higher. Extensive test with version compatibility has not been done yet. Initial tests work with Python 3.2. No testing has been done with 64-bit architectures, but as this package is pure Python, no significant troubles are expected.

1.2 Downloads

The latest stable version of asteval is available from PyPI or CARS (Univ of Chicago):

Download Option	Python Versions	Location
Source Kit	2.6, 2.7, 3.2	• asteval-0.9.tar.gz (PyPI) • asteval-0.9.tar.gz (CARS)
Win32 Installer	2.6	• asteval-0.9.win32-py2.6.exe (PyPI) • asteval-0.9.win32-py2.6.exe (CARS)
Win32 Installer	2.7	• asteval-0.9.win32-py2.7.exe (PyPI) • asteval-0.9.win32-py2.7.exe (CARS)
Win32 Installer	3.2	• asteval-0.9.win32-py3.2.exe (PyPI) • asteval-0.9.win32-py3.2.exe (CARS)
Development Version	all	use asteval github repository

if you have Python Setup Tools installed, you can download and install the asteval Package simply with:

```
easy_install -U asteval
```

1.3 Development Version

To get the latest development version, use:

```
git clone http://github.com/newville/asteval.git
```

1.4 Installation

Installation from source on any platform is:

```
python setup.py install
```

1.5 License

The ASTEVAL code is distribution under the following license:

CHAPTER

TWO

USING ASTEVAL

The asteval module is very easy to use. Import the module and create an Interpreter:

```
>>> from asteval import Interpreter
>>> aeval = Interpreter()
```

you have an embedded interpreter for a procedural, mathematical language that is very much like python in your application, all ready to use:

```
>>> aeval('x = sqrt(3)')
>>> aeval('print x')
1.73205080757
>>> aeval('''for i in range(10):
print i, sqrt(i), log(1+1)
''')
0 0.0 0.0
1 1.0 0.69314718056
2 1.41421356237 1.09861228867
3 1.73205080757 1.38629436112
4 2.0 1.60943791243
5 2.2360679775 1.79175946923
6 2.44948974278 1.94591014906
7 2.64575131106 2.07944154168
8 2.82842712475 2.19722457734
9 3.0 2.30258509299
```

2.1 accessing the symbol table

The symbol table (that is, the mapping between variable and function names and the underlying objects) is a simple dictionary held in the `symtable` attribute of the interpreter. Of course, this can be read or written to by the python program:

```
>>> aeval('x = sqrt(3)')
>>> aeval.symtable['x']
1.73205080757
>>> aeval.symtable['y'] = 100
>>> aeval('print y/8')
12.5
```

(Note the use of true division even though the operands are integers).

Certain names are reserved in Python, and cannot be used within the asteval interpreter. These reserved words are:

and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, True, False, None, eval, execfile, __import__, __package__

Valid symbol names must match the basic regular expression pattern:

```
valid_name = [a-zA-Z_][a-zA-Z0-9_]*
```

2.2 built-in functions

At startup, 130 symbols are loaded into the symbol table from Python's builtins and the **math** module. The builtins include several basic Python functions:

abs, all, any, bin, bool, bytearray, bytes, chr, complex, delattr, dict, dir, divmod, enumerate, filter, float, format, frozenset, getattr, hasattr, hash, hex, id, int, isinstance, len, list, map, max, min, oct, open, ord, pow, property, range, repr, reversed, round, set, setattr, slice, sorted, str, sum, tuple, type, zip

and a large number of named exceptions:

ArithmeticError, AssertionError, AttributeError, BaseException, BufferError, BytesWarning, DeprecationWarning, EOFError, EnvironmentError, Exception, False, FloatingPointError, GeneratorExit, IOError, ImportError, ImportWarning, IndentationError, IndexError, KeyError, KeyboardInterrupt, LookupError, MemoryError, NameError, None, NotImplemented, NotImplementedError, OSError, OverflowError, ReferenceError, RuntimeError, RuntimeWarning, StopIteration, SyntaxError, SyntaxWarning, SystemError, SystemExit, True, TypeError, UnboundLocalError, UnicodeDecodeError, UnicodeEncodeError, UnicodeError, UnicodeTranslateError, UnicodeWarning, ValueError, Warning, ZeroDivisionError

The symbols imported from Python's *math* module include:

acos, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign, cos, cosh, degrees, e, exp, fabs, factorial, floor, fmod, frexp, fsum, hypot, isinf, isnan, ldexp, log, log10, log1p, modf, pi, pow, radians, sin, sinh, sqrt, tan, tanh, trunc

If available, a very large number (~400) additional symbols are imported from numpy.

2.3 conditionals and loops

If-then-else blocks, for-loops (including the optional *else* block) and while loops (also including optional *else* block) are supported, and work exactly as they do in python. Thus:

```
>>> code = """
sum = 0
for i in range(10):
    sum += i*sqrt(*1.0)
    if i % 4 == 0:
        sum = sum + 1
print "sum = ", sum
"""
>>> aeval(code)
sum =  114.049534067
```

2.4 printing

For printing, asteval emulates Python's native `print()` function and `print` statement (for python 2). That is, the behavior mimics the version of python used.

2.5 writing functions

User-defined functions can be written and executed, as in python with a *def* block.

2.6 exceptions

CHAPTER

THREE

MOTIVATION FOR ASTEVAL

The asteval module provides a means to evaluate a large subset of Python in a python program, using a simple dictionary as a namespace and evaluating strings of statements. It is, in effect, a limited version of Python's built-in `eval()`. A completely fair question is: Why on earth would anyone do this? That is, why not simply use `eval()`, or just use Python itself.

The short answer is that sometimes you want to allow evaluation of user input, or expose a simple calculator inside a larger application. For this, `eval()` is pretty scary, as it exposes *all* of Python, which can make user input difficult to trust. Since asteval does not support the **import** statement, user code cannot access the 'os' and 'sys' modules or any functions or classes outside the provided symbol table.

Other missing features (modules, classes, lambda, yield, generators) are similarly motivated. The idea was to make a simple procedural, mathematically-oriented language that could be embedded into larger applications, not to write Python.

In fact, the asteval module grew out the the need for a simple expression evaluator for scientific applications. A first attempt using pyparsing, but was error-prone and difficult to maintain. It turned out that using the Python ast module is so easy that adding more complex programming constructs like conditionals, loops, exception handling, complex assignment and slicing, and even function definition and running was fairly simple implement. Importantly, because parsing is done by the ast module, a whole class of implementation errors disappears – valid python expression will be parsed correctly and converted into an Abstract Syntax Tree. Furthermore, the resulting AST is fairly simple to walk through, greatly simplifying evaluation over any other approach. What started as a desire for a simple expression evaluator grew into a quite useable procedural domain-specific language for mathematical applications.

There is no claim of speed in asteval. Clearly, evaluating the ast tree involves a lot of function calls, and will likely be slower than Python. In preliminary tests, it's about 4x slower than Python.