asteval documentation

Release 0.9.5

Matthew Newville

June 07, 2015

CONTENTS

1 Downloading and Installation **3**

1.1 Requirements . . . 3
1.2 Downloads . . . 3
1.3 Development Version . . . 3
1.4 Installation . . . 3
1.5 License . . . 4

2 Motivation for asteval **5**

2.1 How Safe is asteval? . . . 5

3 Using asteval **7**

3.1 accessing the symbol table . . . 7
3.2 built-in functions . . . 8
3.3 conditionals and loops . . . 8
3.4 printing . . . 9
3.5 writing functions . . . 9
3.6 exceptions . . . 9

4 asteval reference **11**

Python Module Index **13**

Index **15**

ASTEVAL is a safe(ish) evaluator of Python expressions and statements, using Python's ast module. The idea is to provide a simple, robust miniature mathematical language that is more complete than `ast.literal_eval()` and can handle user-input more safely than `eval()`. The emphasis here is on mathematical calculations, so mathematical functions from Python's `math` module are available, and a large number of functions from numpy will be available if it is installed on your system.

Many parts of the Python language are supported, including if-then-else conditionals, while loops, for loops, try-except blocks, list comprehension, slicing, subscripting, and writing user-defined functions. All objects are true python objects, and many built-in data structures (strings, dictionaries, tuple, lists, numpy arrays), are supported. Still, there are important absences and differences, and asteval is by no means an attempt to reproduce Python with its own ast module. Some of the differences and absences include:

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 allowed.
3. importing modules is not allowed.
4. function decorators, generators, yield, and lambda are not supported.
5. several builtins (`eval()`, `execfile()`, `getattr()`, `hasattr()`, `setattr()`, and `delattr()`) are not allowed.
6. Accessing several private object attributes that can provide access to the python interpreter are not allowed.

The result of this makes asteval a decidedly restricted and limited language that is focused on mathematical calculations.

CHAPTER

ONE

DOWNLOADING AND INSTALLATION

1.1 Requirements

The asteval package requires Python 2.6 and higher. Most testing has been done with Python 2.7 and 3.2 through 3.4. As this package is pure Python, and depends on only packages from the standard library and numpy, no significant troubles are expected.

1.2 Downloads

The latest stable version of asteval is 0.9.5 and is available at PyPI:

Download Option	Python Versions	Location
Source Kit	2.6 and higher	asteval-0.9.5.tar.gz
Win32 Installer	2.7	asteval-0.9.5.win32-py2.7.exe
Win32 Installer	3.4	asteval-0.9.5.win32-py3.4.exe
Wheel Installer	2.7	asteval-0.9.5-py2-none-any.whl
Wheel Installer	3.4	asteval-0.9.5-py3-none-any.whl
Development Version	all	github repository

If you have pip, you can install asteval with:

```
pip install asteval
```

If you have Python Setup Tools installed, you can use:

```
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:

> Copyright (c) 2014 Matthew Newville, The University of Chicago
>
> Permission to use and redistribute the source code or binary forms of this software and its documentation, with or without modification is hereby granted provided that the above notice of copyright, these terms of use, and the disclaimer of warranty below appear in the source code and documentation, and that none of the names of The University of Chicago or the authors appear in advertising or endorsement of works derived from this software without specific prior written permission from all parties.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THIS SOFTWARE.

CHAPTER

TWO

MOTIVATION FOR ASTEVAL

The asteval module provides a means to evaluate a large subset of the Python language from within a python program, without using `eval()`. It is, in effect, a limited version of Python's built-in `eval()` that is more restricted, forbidding several actions, and using using a simple dictionary as a flat namespace. 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 (or many other constructs), 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 for asteval is to make a simple procedural, mathematically-oriented language that can be embedded safely into larger applications.

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 easy 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.

Asteval makes no claims about speed. Obviously, 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.

2.1 How Safe is asteval?

I'll be completely honest: I don't know.

If you're looking for guarantees that malicious code cannot ever cause damage, you're definitely looking in the wrong place. I don't suggest that asteval is completely safe, only that it is safer than the builtin `eval()`, and that you might find it useful.

For why `eval()` is dangerous, see, for example Eval is really dangerous and the comments and links therein. Clearly, making `eval()` perfectly safe from malicious user input is a difficult prospect. Basically, if one can cause Python to seg-fault, safety cannot be guaranteed.

Asteval is meant to be safer than the builtin `eval()`, and does try to avoid any known exploits. Many actions are not allowed from the asteval interpreter, including:

- importing modules. Neither 'import' nor '__import__' is supported.
- create classes or modules.

- access to Python's `eval()`, `execfile()`, `getattr()`, `hasattr()`, `setattr()`, and `delattr()`.

In addition (and following the discussion in the link above), the following attributes are blacklisted for all objects, and cannot be accessed:

> __subclasses__, __bases__, __globals__, __code__, __closure__, __func__, __self__, __module__, __dict__, __class__, __call__, __get__, __getattribute__, __subclasshook__, __new__, __init__, func_globals, func_code, func_closure, im_class, im_func, im_self, gi_code, gi_frame

Of course, this approach of making a blacklist cannot be guaranteed to be complete, but it does eliminate classes of attacks to seg-fault the Python interpreter. Of course, asteval will typically expose numpy ufuncs from the numpy module, and several of these can seg-fault Python without too much trouble. If you're paranoid about safe user input that can never cause a segmentation fault, you'll want to disable the use of numpy.

There are important categories of safety that asteval does not even attempt to address. The most important of these is resource hogging. There is no timeout on any calculation, and so a reasonable looking calculation such as:

```
>>> from asteval import Interpreter
>>> aeval = Interpreter()
>>> txt = """nmax = 1e8
... a = sqrt(arange(nmax)
... """
>>> aeval.eval(txt)
```

can take a noticeable amount of CPU time. It it not hard to come up with short program that can run for hundreds of years, which probably exceeds your threshold for an acceptable run-time.

In summary, there are many ways that asteval could be considered part of an un-safe programming environment. Recommendations for how to improve this situation would be greatly appreciated.

CHAPTER

THREE

USING ASTEVAL

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

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

and now you have an embedded interpreter for a procedural, mathematical language that es very much like python, ready for 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
```

3.1 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_]*
```

3.2 built-in functions

At startup, many 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, dict, dir, divmod, enumerate, filter, float, format, frozenset, hash, hex, id, int, isinstance, len, list, map, max, min, oct, ord, pow, range, repr, reversed, round, set, 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.

3.3 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
```

3.4 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.

You can change where output is sent with the `writer` argument when creating the interpreter. By default, outputs are sent to `sys.stdout`.

3.5 writing functions

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

```
>>> from asteval import Interpreter
>>> aeval = Interpreter()
>>> code = """def func(a, b, norm=1.0):
... return (a + b)/norm
... """
>>> aeval(code)
>>> aeval("func(1, 3, norm=10.0)")
0.4
```

3.6 exceptions

asteval monitors and caches exceptions in the evaluated code. Brief error messages are printed (with Python's print statement or function, and so using standard output by default), and the full set of exceptions is kept in the `error` attribute of the `Interpreter` instance. This `error` attribute is a list of instances of the asteval `ExceptionHolder` class, which is accessed through the `get_error()` method. The `error` attribute is reset to an empty list at the beginning of each `eval()`, so that errors are from only the most recent `eval()`.

Thus, to handle and re-raise exceptions from your Python code in a simple REPL loop, you'd want to do something similar to

```
>>> from asteval import Interpreter
>>> aeval = Interpreter()
>>> while True:
>>>     inp_string = raw_input('dsl:>')
>>>     result = aeval(inp_string)
>>>     if len(aeval.error)>0:
>>>         for err in aeval.error:
>>>             print(err.get_error())
>>>     else:
>>>         print(result)
```

CHAPTER

FOUR

ASTEVAL REFERENCE

The asteval module provides an `Interpreter` class, which creates an interpreter. There is also a convenience function `valid_symbol_name()`

class `asteval.`**`Interpreter`**(*symtable=None*[, *writer=None*[, *use_numpy=True*]])

create an asteval interpreter.

Parameters

- **`symtable`** (`None` or dict.) – a Symbol table (if `None`, one will be created).
- **`writer`** (*file-like.*) – callable file-like object where standard output will be sent.
- **`use_numpy`** (boolean (`True` / `False`)) – whether to use functions from numpy.

The symbol table will be loaded with several built in functions, several functions from the `math` module and, if available and requested, several functions from numpy. This will happen even for a symbol table explicitly provided.

The `writer` argument can be used to provide a place to send all output that would normally go to `sys.stdout`. The default is, of course, to send output to `sys.stdout`.

The `use_numpy` argument can be used to control whether functions from numpy are loaded into the symbol table.

`asteval.`**`eval`**(*expression*[, *lineno=0*[, *show_errors=True*]])

evaluate the expression, returning the result.

Parameters

- **`expression`** (*string*) – code to evaluate.
- **`lineno`** (*int*) – line number (for error messages).
- **`show_errors`** (*bool*) – whether to print error messages or leave them in the `errors` list.

`asteval.`**`__call__`**(*expression*[, *lineno=0*[, *show_errors=True*]])

same as `eval()`. That is one can do:

```
>>> from asteval import Interpreter
>>> a = Interpreter()
>>> a('x = 1')
```

instead of:

```
>>> a.eval('x = 1')
```

`asteval.`**`symtable`**

the symbol table. A dictionary with symbol names as keys, and object values (data and functions).

For full control of the symbol table, you can simply access the `symtable` object, inserting, replacing, or removing symbols to alter what symbols are known to your interpreter. You can also access the `symtable` to retrieve results.

asteval.**error**

a list of error information, filled on exceptions. You can test this after each call of the interpreter. It will be empty if the last execution was successful. If an error occurs, this will contain a liste of Exceptions raised.

asteval.**error_msg**

the most recent error message.

asteval.**valid_symbol_name**(*name*)

determines whether the input symbol name is a valid name

This checks for reserved words, and that the name matches the regular expression `[a-zA-Z_][a-zA-Z0-9_]`

PYTHON MODULE INDEX

a

asteval, 11

INDEX

Symbols

__call__() (in module asteval), 11

A

asteval (module), 11

E

error (in module asteval), 11
error_msg (in module asteval), 12
eval() (in module asteval), 11

I

Interpreter (class in asteval), 11

S

symtable (in module asteval), 11

V

valid_symbol_name() (in module asteval), 12