asteval documentation

Release 1.0.2.post1

Matthew Newville

Aug 13, 2024

CONTENTS

The asteval package evaluates Python expressions and statements, providing a safer alternative to Python's builtin `eval()` and a richer, easier to use alternative to `ast.literal_eval()`. It does this by building an embedded interpreter for a subset of the Python language using Python's `ast` module. The emphasis and main area of application is the evaluation of mathematical expressions. Because of this emphasis, mathematical functions from Python's `math` module are built-in to asteval, and a large number of functions from numpy will be available if numpy is installed on your system. For backward compatibility, a few functions that were moved from numpy to numpy_financial will be imported, if that package is installed.

While the primary goal is evaluation of mathematical expressions, many features and constructs of the Python language are supported by default. These features include array slicing and subscripting, if-then-else conditionals, while loops, for loops, try-except blocks, list comprehension, and user-defined functions. All objects in the asteval interpreter are truly Python objects, and all of the basic built-in data structures (strings, dictionaries, tuple, lists, sets, numpy arrays) are supported, including the built-in methods for these objects.

However, asteval is by no means an attempt to reproduce Python with its own `ast` module. There are important differences and missing features compared to Python. Many of these absences are intentional, and part of the effort to try to make a safer version of `eval()`, while some are simply due to the reduced requirements for an embedded mini-language. These differences and absences include:

1. All variable and function symbol names are held in a single symbol table that can be accessed from the calling program. By default, this is a simple dictionary, giving a flat namespace. A more elaborate, still experimental, symbol table that allows both dictionary and attribute access can also be used.
2. creating classes is not allowed.
3. importing modules is not allowed, unless specifically enabled.
4. decorators, generators, type hints, and `lambda` are not supported.
5. `yield`, `await`, and async programming are not supported.
6. Many builtin functions (`eval()`, `getattr()`, `hasattr()`, `setattr()`, and `delattr()`) are not allowed.
7. Accessing many object attributes that can provide access to the python interpreter are not allowed.

The resulting "asteval language" acts very much like miniature version of Python, focused on mathematical calculations, and with noticeable limitations. It is the kind of toy programming language you might use to introduce simple scientific programming concepts, but also includes much of the standard Python features to be a reasonably complete language and not too restricted from what someone familiar with Python would expect.

Because asteval is designed for evaluating user-supplied input, safety against malicious or incompetent user input is an important concern. Asteval tries as hard as possible to prevent user-supplied input from crashing the Python interpreter or from returning exploitable parts of the Python interpreter. In this sense asteval is certainly safer than using `eval()`. However, asteval is an open source project written by volunteers, and we cannot guarantee that it is completely safe against malicious attacks.

CHAPTER

ONE

INSTALLING ASTEVAL

1.1 Requirements

Asteval is a pure Python module. The latest stable version is 1.0.2.post1, which supports Python 3.8 through 3.12.

Installing *asteval* requires *setuptools* and *setuptools_scm*. No other libraries outside of the standard library are required. If numpy and numpy_financial are available, *asteval* will make use of these libraries. Running the test suite requires the *pytest*, *coverage*, and *pytest-cov* modules, deployment uses *build* and *twine*, and building the documentation requires *sphinx*.

Python 3.8 through 3.12 are tested on Windows, MacOS, and Linux, with and without numpy installed. Older Python versions have generally been supported by *asteval* until they are well past the end of security fixes. That is, while *asteval* is no longer tested with Python 3.7, the latest release may continue to work with that version.

Support for new versions of the Python 3 series is not guaranteed until some time after the official release of that version, as we may not start testing until late in the "beta" period of development. Historically, the delay has not been too long, though *asteval* may not support newly introduced language features.

1.2 Installing with *pip*

The latest stable version of *asteval* is 1.0.2.post1 and is available at PyPI or as a conda package. You should be able to install *asteval* with:

```
pip install asteval
```

It may also be available on some conda channels, including *conda-forge*, but as it is a pure Python package with no dependencies or OS-specific extensions, using *pip* should be the preferred method on all platforms and environments.

1.3 Development Version

The latest development version can be found at the github repository, and cloned with:

```
git clone https://github.com/lmfit/asteval.git
```

Installation from the source tree on any platform is can then be done with:

```
pip install .
```

1.4 License

The *asteval* code and documentation is distribution under the following license:

```
The MIT License

Copyright (c) 2024 Matthew Newville, The University of Chicago

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

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 THE
SOFTWARE.
```

CHAPTER

TWO

USING ASTEVAL

This chapter gives a quick overview of asteval, showing basic usage and the most important features. Further details can be found in the next chapter (*Asteval Reference*).

2.1 Creating and using an asteval Interpreter

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 is very much like python:

```
>>> 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
```

There are lots of options when creating the Interpreter to controller what functionality is and isn't allowed and to pre-load data and functions. The default interpreter gives a limited but useful version of the Python language.

2.2 accessing the symbol table

The symbol table (that is, the mapping between variable and function names and the underlying objects) is a simple dictionary (by default, see *Symbol Tables used in asteval* for details of an optional alternative) held in the `symtable` attribute of the interpreter, and can be read or written to:

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

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

As with Python itself, valid symbol names must match the basic regular expression pattern:

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

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

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

2.3 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, 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, about 300 additional symbols are imported from numpy.

2.4 conditionals and loops

If-then-else blocks, for-loops (including the optional `else` block), `while` loops (also including optional `else` block), and `with` blocks 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
```

2.5 comprehensions

list, dict, and set comprehension are supported, acting just as they do in Python. Generators, yield, and async programming are not currently supported.

2.6 printing

For printing, asteval emulates Python's native `print()` function. You can change where output is sent with the `writer` argument when creating the interpreter, or suppress printing all together with the `no_print` option. By default, outputs are sent to `sys.stdout`.

2.7 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
```

2.8 exceptions

Asteval monitors and caches exceptions in the evaluated code. Brief error messages are printed (with Python's print 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

THREE

ASTEVAL REFERENCE

The asteval module has a pretty simple interface, providing an *Interpreter* class which creates an Interpreter of expressions and code. There are a few options available to control what language features to support, how to deal with writing to standard output and standard error, and specifying the symbol table. There are also a few convenience functions: *valid_symbol_name()* is useful for testing the validity of symbol names, and *make_symbol_table()* is useful for creating symbol tables that may be pre-loaded with custom symbols and functions.

3.1 The Interpreter class

class asteval.**Interpreter**(*symtable=None*, *nested_symtable=False*, *user_symbols=None*, *writer=None*, *err_writer=None*, *use_numpy=True*, *max_statement_length=50000*, *minimal=False*, *readonly_symbols=None*, *builtins_readonly=False*, *config=None*, ***kws*)

create an asteval Interpreter: a restricted, simplified interpreter of mathematical expressions using Python syntax.

Parameters

- **symtable** (dict or *None*) – dictionary or SymbolTable to use as symbol table (if *None*, one will be created).
- **nested_symtable** (*bool, optional*) – whether to use a new-style nested symbol table instead of a plain dict [False]
- **user_symbols** (dict or *None*) – dictionary of user-defined symbols to add to symbol table.
- **writer** (file-like or *None*) – callable file-like object where standard output will be sent.
- **err_writer** (file-like or *None*) – callable file-like object where standard error will be sent.
- **use_numpy** (*bool*) – whether to use functions from numpy.
- **max_statement_length** (*int*) – maximum length of expression allowed [50,000 characters]
- **readonly_symbols** (iterable or *None*) – symbols that the user can not assign to
- **builtins_readonly** (*bool*) – whether to blacklist all symbols that are in the initial symtable
- **minimal** (*bool*) – create a minimal interpreter: disable many nodes (see Note 1).
- **config** (*dict*) – dictionay listing which nodes to support (see note 2))

Notes

1. setting *minimal=True* is equivalent to setting a config with the following nodes disabled: ('import', 'importfrom', 'if', 'for', 'while', 'try', 'with', 'functiondef', 'ifexp', 'listcomp', 'dictcomp', 'setcomp', 'augassign', 'assert', 'delete', 'raise', 'print')
2. by default 'import' and 'importfrom' are disabled, though they can be enabled.

If not provided, a symbol table will be created with *make_symbol_table()* that will include several standard python builtin functions, several functions from the `math` module and (if available and not turned off) several functions from numpy.

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`. Similarly, `err_writer` will be used for output that will otherwise be sent to `sys.stderr`.

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

Whether the user-code is able to overwrite the entries in the symbol table can be controlled with the `readonly_symbols` and `builtins_readonly` keywords.

3.2 Configuring which features the Interpreter recognizes

The interpreter can be configured to enable or disable many language constructs, named according to the AST node in the Python language definition.

Table of optional Python AST nodes used asteval. The minimal configuration excludes all of the nodes listed, to give a bare-bones mathematical language but will full support for Python data types and array slicing.

node name	description	in default config	in minimal config
import	import statements	False	False
importfrom	from x import y	False	False
assert	assert statements	True	False
augassign	x += 1	True	False
delete	delete statements	True	False
if	if/then blocks	True	False
ifexp	a = b if c else d	True	False
for	for loops	True	False
formattedvalue	f-strings	True	False
functiondef	define functions	True	False
print	print function	True	False
raise	raise statements	True	False
listcomp	list comprehension	True	False
dictcomp	dict comprehension	True	False
setcomp	set comprehension	True	False
try	try/except blocks	True	False
while	while blocks	True	False
with	with blocks	True	False

The `minimal` configuration for the Interpreter will support many basic Python language constructs including all basic data types, operators, slicing. The `default` configuration adds many language constructs, including

- if-elif-else conditionals

- for loops, with `else`
- while loops, with `else`
- try-except-finally blocks
- with blocks
- augmented assignments: `x += 1`
- if-expressions: `x = a if TEST else b`
- list comprehension: `out = [sqrt(i) for i in values]`
- set and dict comprehension, too.
- print formatting with `%`, `str.format()`, or f-strings.
- function definitions

The nodes listed in Table *Table of optional Python AST nodes used asteval* can be enabled and disabled individually with the appropriate `no_NODE` or `with_NODE` argument when creating the interpreter, or specifying a `config` dictionary.

That is, you might construct an Interpreter as:

```
>>> from asteval import Interpreter
>>>
>>> aeval_nowhile = Interpreter(no_while=True)
>>>
>>> config = {'while': False, 'if': False, 'try': False,
              'for': False, 'with': False}
>>> aveal_noblocks = Interpreter(config=config)
```

Passing, `minimal=True` will turn off all the nodes listed in Table *Table of optional Python AST nodes used asteval*:

```
>>> from asteval import Interpreter
>>>
>>> aeval_min = Interpreter(minimal=True)
>>> aeval_min.config
{'import': False, 'importfrom': False, 'assert': False, 'augassign': False,
'delete': False, 'if': False, 'ifexp': False, 'for': False,
'formattedvalue': False, 'functiondef': False, 'print': False,
'raise': False, 'listcomp': False, 'dictcomp': False, 'setcomp': False,
'try': False, 'while': False, 'with': False}
```

As shown above, importing Python modules with `import module` or `from module import method` can be enabled, but is disabled by default. To enable these, use `with_import=True` and `with_importfrom=True`, as

```
>>> from asteval import Interpreter
>>> aeval_max = Interpreter(with_import=True, with_importfrom=True)
```

or by setting the config dictionary as described above:

3.3 Interpreter methods and attributes

An Interpreter instance has many methods, but most of them are implementation details for how to handle particular AST nodes, and should not be considered as part of the usable API. The methods described be low, and the examples elsewhere in this documentation should be used as the stable API.

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

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.
- **raise_errors** (*bool*) – whether to re-raise exceptions or leave them in the `errors` list.

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

same as *eval()*. That is:

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

instead of:

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

asteval.**symtable**

the symbol table where all data and functions for the Interpreter are stored and looked up. By default, this is a simple dictionary with symbol names as keys, and values of data and functions. If the `nested_symtable` option is used, the symbol tables will be a subclass of a dictionary with more features, as discussed in *Symbol Tables used in asteval*.

In either case, the symbol table can be accessed from the calling program using the *symtable* attribute of the Interpreter. This allows the calling program to read, insert, replace, or remove symbols to alter what symbols are known to your interpreter.

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.

3.4 Symbol Tables used in asteval

The symbol table holds all of the data used by the Interpreter. That is, when you execute `a = b * cos(pi/3)`, the Interpreter sees that it needs to lookup values for `b`, `cos`, and `pi` (it already knows `=`, `*`, `/`, `(`, and `)` mean), and then set the value for `a`. The place where it looks up and then sets those values for these assigned variables is the symbol table.

Historically, and by default, the symbol table in Asteval is a simple dictionary with variable names as the keys, and their values as the corresponding values. This is slightly simpler than in Python or roughly equivalent to everything

being "global". This isn't exactly true, and what happens inside an Asteval Procedure (basically, a function) is a little different as a special local symbol table (or Frame) is created for that function, but it is mostly true.

Symbol names are limited to being valid Python object names, and must match `[a-zA-Z_][a-zA-Z0-9_]*` and not be a reserved word. The symbol table is held in the *symtable* attribute of the Interpreter, and can be accessed and manipulated from the containing Python program. This allows the calling program to read, insert, replace, or remove symbols to alter what symbols are known to your interpreter. That is, it is perfectly valid to do something like this:

```
>>> from asteval import Interpreter
>>> aeval = Interpreter()
>>> aeval.symtable['x'] = 10
>>> aeval('sqrt(x)')
3.1622776601683795
```

By default, the symbol table will be pre-loaded with many Python builtins, functions from the `math` module, and functions from `numpy` if available. You can control some of these settings or add symbols into the symbol table with the `use_numpy` and `user_symbols` arguments when creating an Interpreter. You can also build your own symbol table and pass that it, and use the `readonly_symbols` and `builtins_readonly` options to prevent some symbols to be writeable from within the Interpreter. You can also create your own symbol table, either as a plain dict, or with the *make_symbol_table()* function, and alter that to use as the `symtable` option when creating an Interpreter. That is, the calling program can fully control the symbol table, either pre-loading custom variables and functions or removing default functions.

Added in version 0.9.31.

3.4.1 New Style Symbol Table

Beginning with version 0.9.31, there is an option to use a more complex and nested symbol table. This symbol table uses a `"Group"` object which is a subclass of a Python dict that can also be used with `object.attribute` syntax:

```
>>> from asteval import Interpreter
>>> aeval = Interpreter(nested_symtable=True)
>>> aeval('x = 3')
>>> aeval.symtable['x']  # as with default dictionary
3
>>> aeval.symtable.x     # new
3
>>> aeval.symtable.y = 7  # new
>>> aeval('print(x+y)')
10
```

As with the plain-dictionary symbol table, all symbols must be valid Python identifiers, and cannot be reserved words.

In addition, this symbol table can be nested – not flat – and may have a special attribute called `_searchgroups` that give the name of sub-Groups to search for symbols. By default, when using this new-style symbol table, the mathematical functions imported from the `math` and `numpy` modules) are placed in a subgroup named `math` (with about 300 named functions and variables), and the `_searchgroups` variable is set to the tuple `('math',)`. When looking for the a symbol in an expression like `a = b * cos( pi /3)`, the Interpreter will have to find and use the symbols names for `b`, `cos` and `pi`. With the old-style symbol table, all of these must be in the flat dictionary, which makes it difficult to browse through the symbol table. With the new, nested symbol table, the names `b`, `cos` and `pi` are first looked for in the top-level Group. If not found there, they are looked for in the subgroups named in `_searchgroups`, in order and returned as soon as one is found. That is the expectation is that *b* would be found in the "top-level user Group", while `cos` and `pi` would be found in the `math` Group, and that:

```
>>> aeval('a = b * cos( pi /3)')
>>> aeval('a = b * math.cos(math.pi /3)')
```

would be equivalent, as if you had imported a module that would automatically be searched: something between `import math` and `from math import *`. Though different from how Python works, if using Asteval as a domain-specific language, this nesting and automated searching can be quite useful.

3.5 Utility Functions

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

Determine whether the input symbol name is a valid name.

Parameters
name (*str*) – name to check for validity.

Returns
valid – whether name is a a valid symbol name

Return type
bool

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

asteval.**make_symbol_table**(*use_numpy=True*, *nested=False*, *top=True*, ***kws*)

Create a default symboltable, taking dict of user-defined symbols.

Parameters

- **numpy** (*bool, optional*) – whether to include symbols from numpy [True]
- **nested** (*bool, optional*) – whether to make a "new-style" nested table instead of a plain dict [False]
- **top** (*bool, optional*) – whether this is the top-level table in a nested-table [True]
- **kws** (*optional*) – additional symbol name, value pairs to include in symbol table

Returns
symbol_table – a symbol table that can be used in *asteval.Interpereter*

Return type
dict or nested Group

To make and use a custom symbol table, one might do this:

```
from asteval import Interpreter, make_symbol_table
import numpy as np
def cosd(x):
    "cos with angle in degrees"
    return np.cos(np.radians(x))

def sind(x):
    "sin with angle in degrees"
    return np.sin(np.radians(x))

def tand(x):
```

(continues on next page)

(continued from previous page)

```
    "tan with angle in degrees"
    return np.tan(np.radians(x))

syms = make_symbol_table(use_numpy=True, cosd=cosd, sind=sind, tand=tand)

aeval = Interpreter(symtable=syms)
print(aeval("sind(30)"))
```

which will print `0.5`.

CHAPTER

FOUR

MOTIVATION FOR ASTEVAL

The asteval module allows you to evaluate a large subset of the Python language from within a python program, without using `eval()`. It is, in effect, a restricted version of Python's built-in `eval()`, forbidding several actions, and using (by default) a simple dictionary as a flat namespace. A completely fair question is: Why is this desirable? 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 or even scientific calculator inside a larger application. For this, `eval()` is pretty scary, as it exposes *all* of Python, which makes user input difficult to trust. Since asteval does not support the **import** statement (unless explicitly enabled) or many other constructs, user code cannot access the `os` and `sys` modules or any functions or classes outside those provided in the symbol table.

Many of the other missing features (modules, classes, lambda, yield, generators) are similarly motivated by a desire for a safer version of `eval()`. The idea for asteval is to make a simple procedural, mathematically-oriented language that can be embedded into larger applications.

In fact, the asteval module grew out the the need for a simple expression evaluator for scientific applications such as the lmfit and xraylarch modules. An early attempt using the *pyparsing* module worked but was error-prone and difficult to maintain. While the simplest of calculators or expressiona-evaluators is not hard with pyparsing, it turned out that using the Python `ast` module makes it much easier to implement a feature-rich scientific calculator, including slicing, complex numbers, keyword arguments to functions, etc. In fact, this approach meant that adding more complex programming constructs like conditionals, loops, exception handling, and even user-defined functions was fairly simple. An important benefit of using the `ast` module is that whole categories of implementation errors involving parsing, lexing, and defining a grammar disappear. Any 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 the evaluation process. 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. Evaluating the AST involves many function calls, which is going to be slower than Python - often 4x slower than Python. That said, for certain use cases (see https://stackoverflow.com/questions/34106484), use of asteval and numpy can approach the speed of *eval* and the *numexpr* modules.

4.1 How Safe is asteval?

Asteval avoids all of the exploits we know about that make `eval()` dangerous. For reference, see, Eval is really dangerous and the comments and links therein. From this discussion it is apparent that not only is `eval()` unsafe, but that it is a difficult prospect to make any program that takes user input perfectly safe. In particular, if a user can cause Python to crash with a segmentation fault, safety cannot be guaranteed. Asteval explicitly forbids the exploits described in the above link, and works hard to prevent malicious code from crashing Python or accessing the underlying operating system. That said, we cannot guarantee that asteval is completely safe from malicious code. We claim only that it is safer than the builtin `eval()`, and that you might find it useful.

Some of the things not allowed in the asteval interpreter for safety reasons include:

- importing modules. Neither `import` nor `__import__` are supported by default. If you do want to support `import` and `import from`, you have to explicitly enable these.
- create classes or modules.
- use `string.format()`, though f-string formatting and using the `%` operator for string formatting are supported.
- **access to Python's `eval()`, `getattr()`, `hasattr()`,** `setattr()`, and `delattr()`.
- accessing object attributes that begin and end with `__`, the so-called `dunder` attributes. This will include (but is not limited to `__globals__`, `__code__`, `__func__`, `__self__`, `__module__`, `__dict__`, `__class__`, `__call__`, and `__getattribute__`. None of these can be accessed for any object.

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

> `func_globals`, `func_code`, `func_closure`, `im_class`, `im_func`, `im_self`, `gi_code`, `gi_frame`, `f_locals`

While this approach of making a blacklist cannot be guaranteed to be complete, it does eliminate entire classes of attacks known to be able to seg-fault the Python interpreter.

An important caveat is that asteval will typically expose numpy `ufuncs` from the numpy module. Several of these can seg-fault Python without too much trouble. If you are paranoid about safe user input that can never cause a segmentation fault, you may want to consider disabling the use of numpy, or take extra care to specify what can be used.

In 2024, an independent security audit of asteval done by Andrew Effenhauser, Ayman Hammad, and Daniel Crowley in the X-Force Security Research division of IBM showed insecurities with `string.format`, so that access to this and `string.format_map` method were removed. In addition, this audit showed that the `numpy` submodules `linalg`, `fft`, and `polynomial` expose many exploitable objects, so these submodules were removed by default. If needed, these modules can be added to any Interpreter either using the `user_symbols` argument when creating it, or adding the needed symbols to the symbol table after the Interpreter is created.

There are important categories of safety that asteval may attempt to address, but cannot guarantee success. The most important of these is resource hogging, which might be used for a denial-of-service attack. There is no guaranteed 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))   # using numpy.sqrt() and numpy.arange()
"""
aeval.eval(txt)
```

can take a noticeable amount of CPU time - if it does not, increasing that value of `nmax` almost certainly will, and can even crash the Python shell.

As another example, consider the expression `x**y**z`. For values `x=y=z=5`, the run time will be well under 0.001 seconds. For `x=y=z=8`, run time will still be under 1 sec. Changing to `x=8, y=9, z=9`, will cause the statement to take several seconds. With `x=y=z=9`, executing that statement may take more than 1 hour on some machines. It is not hard to come up with short program that would run for hundreds of years, which probably exceeds anyones threshold for an acceptable run-time. There simply is not a good way to predict how long any code will take to run from the text of the code itself: run time cannot be determined lexically.

To be clear, for the `x**y**z` exponentiation example, asteval will raise a runtime error, telling you that an exponent > 10,000 is not allowed. Several other attempts are made to prevent long-running operations or memory exhaustion. These checks will prevent:

- statements longer than 50,000 bytes.

- values of exponents (p in x**p) > 10,000.
- string operations with strings longer than 262144 bytes
- shift operations with shifts (p in x << p) > 1000.
- more than 262144 open buffers
- opening a file with a mode other than 'r', 'rb', or 'ru'.

These checks happen at runtime, not by analyzing the text of the code. As with the example above using numpy.arange, very large arrays and lists can be created that might approach memory limits. There are countless other "clever ways" to have very long run times that cannot be readily predicted from the text.

The exponential example also highlights the issue that there is not a good way to check for a long-running calculation within a single Python process. That calculation is not stuck within the Python interpreter, but in C code (no doubt the pow() function) called by the Python interpreter itself. That call will not return from the C library to the Python interpreter or allow other threads to run until that call is done. That means that from within a single process, there is not a reliable way to tell asteval (or really, even Python) when a calculation has taken too long: Denial of Service is hard to detect before it happens, and even challenging to detect while it is happening. The only reliable way to limit run time is at the level of the operating system, with a second process watching the execution time of the asteval process and either try to interrupt it or kill it.

For a limited range of problems, you can try to avoid asteval taking too long. For example, you may try to limit the *recursion limit* when executing expressions, with a code like this:

```
import contextlib

@contextlib.contextmanager
def limited_recursion(recursion_limit):
    old_limit = sys.getrecursionlimit()
    sys.setrecursionlimit(recursion_limit)
    try:
        yield
    finally:
        sys.setrecursionlimit(old_limit)

with limited_recursion(100):
    Interpreter().eval(...)
```

A secondary security concern is that the default list of supported functions does include Python's open() which will allow disk access to the untrusted user. If numpy is supported, its load() and loadtxt() functions will also normally be supported. Including these functions does not elevate permissions, but it does allow the user of the asteval interpreter to read files with the privileges of the calling program. In some cases, this may not be desirable, and you may want to remove some of these functions from the symbol table, re-implement them, or ensure that your program cannot access information on disk that should be kept private.

In summary, while asteval attempts to be safe and is definitely safer than using eval(), there may be ways that using asteval could lead to increased risk of malicious use. Recommendations for how to improve this situation would be greatly appreciated.

PYTHON MODULE INDEX

INDEX

Symbols

A

E

I

M

S

V