Pyomo Documentation

Release 6.7.0.dev0

Pyomo

Aug 31, 2023

CONTENTS

Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities.

Getting Started	**User Guide**
Installation	*User guide index*
Developer Guide	**Reference Guide**
Index	*Library Reference*

CHAPTER

ONE

GETTING STARTED

TOOO

CHAPTER

TWO

USER GUIDE

Common Warnings/Errors External Pyomo Tutorials

CHAPTER

THREE

DEVELOPER GUIDE

This guide describes utilities and design philosophies useful for Pyomo developers or anyone interested in developing packages that use or interrogate Pyomo models.

3.1 The Pyomo Configuration System

The Pyomo config system provides a set of three classes (`ConfigDict`, `ConfigList`, and `ConfigValue`) for managing and documenting structured configuration information and user input. The system is based around the ConfigValue class, which provides storage for a single configuration entry. ConfigValue objects can be grouped using two containers (ConfigDict and ConfigList), which provide functionality analogous to Python's dict and list classes, respectively.

At its simplest, the Config system allows for developers to specify a dictionary of documented configuration entries, allow users to provide values for those entries, and retrieve the current values:

```
>>> from pyomo.common.config import (
...     ConfigDict, ConfigList, ConfigValue
... )
>>> config = ConfigDict()
>>> config.declare('filename', ConfigValue(
...     default=None,
...     domain=str,
...     description="Input file name",
... ))
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare("bound tolerance", ConfigValue(
...     default=1E-5,
...     domain=float,
...     description="Bound tolerance",
...     doc="Relative tolerance for bound feasibility checks"
... ))
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare("iteration limit", ConfigValue(
...     default=30,
...     domain=int,
...     description="Iteration limit",
...     doc="Number of maximum iterations in the decomposition methods"
... ))
<pyomo.common.config.ConfigValue object at ...>
>>> config['filename'] = 'tmp.txt'
>>> print(config['filename'])
```

(continues on next page)

(continued from previous page)

```
tmp.txt
>>> print(config['iteration limit'])
30
```

For convenience, ConfigDict objects support read/write access via attributes (with spaces in the declaration names replaced by underscores):

```
>>> print(config.filename)
tmp.txt
>>> print(config.iteration_limit)
30
>>> config.iteration_limit = 20
>>> print(config.iteration_limit)
20
```

3.1.1 Domain validation

All Config objects support a `domain` keyword that accepts a callable object (type, function, or callable instance). The domain callable should take data and map it onto the desired domain, optionally performing domain validation (see `ConfigValue`, `ConfigDict`, and `ConfigList` for more information). This allows client code to accept a very flexible set of inputs without "cluttering" the code with input validation:

```
>>> config.iteration_limit = 35.5
>>> print(config.iteration_limit)
35
>>> print(type(config.iteration_limit).__name__)
int
```

In addition to common types (like `int`, `float`, `bool`, and `str`), the config system profides a number of custom domain validators for common use cases:

Bool(val)	Domain validator for bool-like objects.
Integer(val)	Domain validation function admitting integers
PositiveInt(val)	Domain validation function admitting strictly positive integers
NegativeInt(val)	Domain validation function admitting strictly negative integers
NonNegativeInt(val)	Domain validation function admitting integers >= 0
NonPositiveInt(val)	Domain validation function admitting integers <= 0
PositiveFloat(val)	Domain validation function admitting strictly positive numbers
NegativeFloat(val)	Domain validation function admitting strictly negative numbers
NonPositiveFloat(val)	Domain validation function admitting numbers less than or equal to 0
NonNegativeFloat(val)	Domain validation function admitting numbers greater than or equal to 0
In(domain[, cast])	Domain validation class admitting a Container of possible values
InEnum(domain)	Domain validation class admitting an enum value/name.
ListOf(itemtype[, domain, string_lexer])	Domain validator for lists of a specified type
Module([basePath, expandPath])	Domain validator for modules.
Path([basePath, expandPath])	Domain validator for path-like options.
PathList([basePath, expandPath])	Domain validator for a list of path-like objects.
DynamicImplicitDomain(callback)	Implicit domain that can return a custom domain based on the key.

3.1.2 Configuring class hierarchies

A feature of the Config system is that the core classes all implement `__call__`, and can themselves be used as `domain` values. Beyond providing domain verification for complex hierarchical structures, this feature allows ConfigDicts to cleanly support the configuration of derived objects. Consider the following example:

```
>>> class Base(object):
...     CONFIG = ConfigDict()
...     CONFIG.declare('filename', ConfigValue(
...         default='input.txt',
...         domain=str,
...     ))
...     def __init__(self, **kwds):
...         c = self.CONFIG(kwds)
...         c.display()
...
>>> class Derived(Base):
...     CONFIG = Base.CONFIG()
...     CONFIG.declare('pattern', ConfigValue(
...         default=None,
...         domain=str,
...     ))
...
>>> tmp = Base(filename='foo.txt')
filename: foo.txt
```

(continues on next page)

(continued from previous page)

```
>>> tmp = Derived(pattern='.*warning')
filename: input.txt
pattern: .*warning
```

Here, the base class `Base` declares a class-level attribute CONFIG as a ConfigDict containing a single entry (`filename`). The derived class (`Derived`) then starts by making a copy of the base class' `CONFIG`, and then defines an additional entry (*pattern*). Instances of the base class will still create `c` instances that only have the single `filename` entry, whereas instances of the derived class will have `c` instances with two entries: the `pattern` entry declared by the derived class, and the `filename` entry "inherited" from the base class.

An extension of this design pattern provides a clean approach for handling "ephemeral" instance options. Consider an interface to an external "solver". Our class implements a `solve()` method that takes a problem and sends it to the solver along with some solver configuration options. We would like to be able to set those options "persistently" on instances of the interface class, but still override them "temporarily" for individual calls to `solve()`. We implement this by creating copies of the class's configuration for both specific instances and for use by each `solve()` call:

```
>>> class Solver(object):
...     CONFIG = ConfigDict()
...     CONFIG.declare('iterlim', ConfigValue(
...         default=10,
...         domain=int,
...     ))
...     def __init__(self, **kwds):
...         self.config = self.CONFIG(kwds)
...     def solve(self, model, **options):
...         config = self.config(options)
...         # Solve the model with the specified iterlim
...         config.display()
...
>>> solver = Solver()
>>> solver.solve(None)
iterlim: 10
>>> solver.config.iterlim = 20
>>> solver.solve(None)
iterlim: 20
>>> solver.solve(None, iterlim=50)
iterlim: 50
>>> solver.solve(None)
iterlim: 20
```

3.1.3 Interacting with argparse

In addition to basic storage and retrieval, the Config system provides hooks to the argparse command-line argument parsing system. Individual Config entries can be declared as argparse arguments using the `declare_as_argument()` method. To make declaration simpler, the `declare()` method returns the declared Config object so that the argument declaration can be done inline:

```
>>> import argparse
>>> config = ConfigDict()
>>> config.declare('iterlim', ConfigValue(
...     domain=int,
...     default=100,
```

(continues on next page)

(continued from previous page)

```
...     description="iteration limit",
... )).declare_as_argument()
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare('lbfgs', ConfigValue(
...     domain=bool,
...     description="use limited memory BFGS update",
... )).declare_as_argument()
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare('linesearch', ConfigValue(
...     domain=bool,
...     default=True,
...     description="use line search",
... )).declare_as_argument()
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare('relative tolerance', ConfigValue(
...     domain=float,
...     description="relative convergence tolerance",
... )).declare_as_argument('--reltol', '-r', group='Tolerances')
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare('absolute tolerance', ConfigValue(
...     domain=float,
...     description="absolute convergence tolerance",
... )).declare_as_argument('--abstol', '-a', group='Tolerances')
<pyomo.common.config.ConfigValue object at ...>
```

The ConfigDict can then be used to initialize (or augment) an argparse ArgumentParser object:

```
>>> parser = argparse.ArgumentParser("tester")
>>> config.initialize_argparse(parser)
```

Key information from the ConfigDict is automatically transferred over to the ArgumentParser object:

```
>>> print(parser.format_help())
usage: tester [-h] [--iterlim INT] [--lbfgs] [--disable-linesearch]
              [--reltol FLOAT] [--abstol FLOAT]
...
  -h, --help            show this help message and exit
  --iterlim INT         iteration limit
  --lbfgs               use limited memory BFGS update
  --disable-linesearch  [DON'T] use line search

Tolerances:
  --reltol FLOAT, -r FLOAT
                        relative convergence tolerance
  --abstol FLOAT, -a FLOAT
                        absolute convergence tolerance
```

Parsed arguments can then be imported back into the ConfigDict:

```
>>> args=parser.parse_args(['--lbfgs', '--reltol', '0.1', '-a', '0.2'])
>>> args = config.import_argparse(args)
>>> config.display()
iterlim: 100
```

(continues on next page)

(continued from previous page)

```
lbfgs: true
linesearch: true
relative tolerance: 0.1
absolute tolerance: 0.2
```

3.1.4 Accessing user-specified values

It is frequently useful to know which values a user explicitly set, and which values a user explicitly set but have never been retrieved. The configuration system provides two generator methods to return the items that a user explicitly set (`user_values()`) and the items that were set but never retrieved (`unused_user_values()`):

```
>>> print([val.name() for val in config.user_values()])
['lbfgs', 'relative tolerance', 'absolute tolerance']
>>> print(config.relative_tolerance)
0.1
>>> print([val.name() for val in config.unused_user_values()])
['lbfgs', 'absolute tolerance']
```

3.1.5 Generating output & documentation

Configuration objects support three methods for generating output and documentation: `display()`, `generate_yaml_template()`, and `generate_documentation()`. The simplest is `display()`, which prints out the current values of the configuration object (and if it is a container type, all of it's children). `generate_yaml_template()` is simular to `display()`, but also includes the description fields as formatted comments.

```
>>> solver_config = config
>>> config = ConfigDict()
>>> config.declare('output', ConfigValue(
...     default='results.yml',
...     domain=str,
...     description='output results filename'
... ))
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare('verbose', ConfigValue(
...     default=0,
...     domain=int,
...     description='output verbosity',
...     doc='This sets the system verbosity.  The default (0) only logs '
...     'warnings and errors.  Larger integer values will produce '
...     'additional log messages.',
... ))
<pyomo.common.config.ConfigValue object at ...>
>>> config.declare('solvers', ConfigList(
...     domain=solver_config,
...     description='list of solvers to apply',
... ))
<pyomo.common.config.ConfigList object at ...>
>>> config.display()
output: results.yml
```

(continues on next page)

(continued from previous page)

```
verbose: 0
solvers: []
>>> print(config.generate_yaml_template())
output: results.yml  # output results filename
verbose: 0           # output verbosity
solvers: []          # list of solvers to apply
```

It is important to note that both methods document the current state of the configuration object. So, in the example above, since the *solvers* list is empty, you will not get any information on the elements in the list. Of course, if you add a value to the list, then the data will be output:

```
>>> tmp = config()
>>> tmp.solvers.append({})
>>> tmp.display()
output: results.yml
verbose: 0
solvers:
  -
    iterlim: 100
    lbfgs: true
    linesearch: true
    relative tolerance: 0.1
    absolute tolerance: 0.2
>>> print(tmp.generate_yaml_template())
output: results.yml          # output results filename
verbose: 0                   # output verbosity
solvers:                     # list of solvers to apply
  -
    iterlim: 100             # iteration limit
    lbfgs: true              # use limited memory BFGS update
    linesearch: true         # use line search
    relative tolerance: 0.1  # relative convergence tolerance
    absolute tolerance: 0.2  # absolute convergence tolerance
```

The third method (`generate_documentation()`) behaves differently. This method is designed to generate reference documentation. For each configuration item, the *doc* field is output. If the item has no *doc*, then the *description* field is used.

List containers have their *domain* documented and not their current values. The documentation can be configured through optional arguments. The defaults generate LaTeX documentation:

```
>>> print(config.generate_documentation())
\begin{description}[topsep=0pt,parsep=0.5em,itemsep=-0.4em]
  \item[{output}]\hfill
    \\output results filename
  \item[{verbose}]\hfill
    \\This sets the system verbosity.  The default (0) only logs warnings and
    errors.  Larger integer values will produce additional log messages.
  \item[{solvers}]\hfill
    \\list of solvers to apply
  \begin{description}[topsep=0pt,parsep=0.5em,itemsep=-0.4em]
    \item[{iterlim}]\hfill
      \\iteration limit
```

(continues on next page)

(continued from previous page)

```
    \item[{lbfgs}]\hfill
      \\use limited memory BFGS update
    \item[{linesearch}]\hfill
      \\use line search
    \item[{relative tolerance}]\hfill
      \\relative convergence tolerance
    \item[{absolute tolerance}]\hfill
      \\absolute convergence tolerance
  \end{description}
\end{description}
```

3.2 Deprecation and Removal of Functionality

During the course of development, there may be cases where it becomes necessary to deprecate or remove functionality from the standard Pyomo offering.

3.2.1 Deprecation

We offer a set of tools to help with deprecation in `pyomo.common.deprecation`.

By policy, when deprecating or moving an existing capability, one of the following utilities should be leveraged. Each has a required `version` argument that should be set to current development version (e.g., `"6.6.2.dev0"`). This version will be updated to the next actual release as part of the Pyomo release process. The current development version can be found by running `pyomo --version` on your local fork/branch.

`deprecated`([msg, logger, version, remove_in])	Decorator to indicate that a function, method, or class is deprecated.
`deprecation_warning`(msg[, logger, version, ...])	Standardized formatter for deprecation warnings
`relocated_module`(new_name[, msg, logger, ...])	Provide a deprecation path for moved / renamed modules
`relocated_module_attribute`(local, target, ...)	Provide a deprecation path for moved / renamed module attributes
`RenamedClass`(name, bases, classdict, *args, ...)	Metaclass to provide a deprecation path for renamed classes

`@pyomo.common.deprecation.`**`deprecated`**(*msg=None*, *logger=None*, *version=None*, *remove_in=None*)

Decorator to indicate that a function, method, or class is deprecated.

This decorator will cause a warning to be logged when the wrapped function or method is called, or when the deprecated class is constructed. This decorator also updates the target object's docstring to indicate that it is deprecated.

Parameters

- **`msg`** (*str*) – a custom deprecation message (default: "This {function|class} has been deprecated and may be removed in a future release.")
- **`logger`** (*str*) – the logger to use for emitting the warning (default: the calling pyomo package, or "pyomo")
- **`version`** (*str*) – [required] the version in which the decorated object was deprecated. General practice is to set version to the current development version (from *pyomo –version*) during development and update it to the actual release as part of the release process.

- **remove_in** (*str*) – the version in which the decorated object will be removed from the code.

Example

```
>>> from pyomo.common.deprecation import deprecated
>>> @deprecated(version='1.2.3')
... def sample_function(x):
...     return 2*x
>>> sample_function(5)
WARNING: DEPRECATED: This function (sample_function) has been deprecated and
    may be removed in a future release.  (deprecated in 1.2.3) ...
10
```

pyomo.common.deprecation.**deprecation_warning**(*msg*, *logger=None*, *version=None*, *remove_in=None*, *calling_frame=None*)

Standardized formatter for deprecation warnings

This is a standardized routine for formatting deprecation warnings so that things look consistent and "nice".

Parameters

- **msg** (*str*) – the deprecation message to format
- **logger** (*str*) – the logger to use for emitting the warning (default: the calling pyomo package, or "pyomo")
- **version** (*str*) – [required] the version in which the decorated object was deprecated. General practice is to set version to the current development version (from *pyomo –version*) during development and update it to the actual release as part of the release process.
- **remove_in** (*str*) – the version in which the decorated object will be removed from the code.
- **calling_frame** (*frame*) – the original frame context that triggered the deprecation warning.

Example

```
>>> from pyomo.common.deprecation import deprecation_warning
>>> deprecation_warning('This functionality is deprecated.', version='1.2.3')
WARNING: DEPRECATED: This functionality is deprecated.  (deprecated in 1.2.3) ...
```

pyomo.common.deprecation.**relocated_module**(*new_name*, *msg=None*, *logger=None*, *version=None*, *remove_in=None*)

Provide a deprecation path for moved / renamed modules

Upon import, the old module (that called *relocated_module()*) will be replaced in *sys.modules* by an alias that points directly to the new module. As a result, the old module should have only two lines of executable Python code (the import of *relocated_module* and the call to it).

Parameters

- **new_name** (*str*) – The new (fully-qualified) module name
- **msg** (*str*) – A custom deprecation message.
- **logger** (*str*) – The logger to use for emitting the warning (default: the calling pyomo package, or "pyomo")

- **version** (*str [required]*) – The version in which the module was renamed or moved. General practice is to set version to the current development version (from *pyomo –version*) during development and update it to the actual release as part of the release process.
- **remove_in** (*str*) – The version in which the module will be removed from the code.

Example

```
>>> from pyomo.common.deprecation import relocated_module
>>> relocated_module('pyomo.common.deprecation', version='1.2.3')
WARNING: DEPRECATED: The '...' module has been moved to
    'pyomo.common.deprecation'. Please update your import.
    (deprecated in 1.2.3) ...
```

pyomo.common.deprecation.**relocated_module_attribute**(*local*, *target*, *version*, *remove_in=None*, *msg=None*, *f_globals=None*)

Provide a deprecation path for moved / renamed module attributes

This function declares that a local module attribute has been moved to another location. For Python 3.7+, it leverages a module.__getattr__ method to manage the deferred import of the object from the new location (on request), as well as emitting the deprecation warning.

Parameters

- **local** (*str*) – The original (local) name of the relocated attribute
- **target** (*str*) – The new absolute import name of the relocated attribute
- **version** (*str*) – The Pyomo version when this move was released (passed to deprecation_warning)
- **remove_in** (*str*) – The Pyomo version when this deprecation path will be removed (passed to deprecation_warning)
- **msg** (*str*) – If not None, then this specifies a custom deprecation message to be emitted when the attribute is accessed from its original location.

class pyomo.common.deprecation.**RenamedClass**(*name*, *bases*, *classdict*, **args*, ***kwargs*)

Metaclass to provide a deprecation path for renamed classes

This metaclass provides a mechanism for renaming old classes while still preserving isinstance / issubclass relationships.

Examples

```
>>> from pyomo.common.deprecation import RenamedClass
>>> class NewClass(object):
...     pass
>>> class OldClass(metaclass=RenamedClass):
...     __renamed__new_class__ = NewClass
...     __renamed__version__ = '6.0'
```

Deriving from the old class generates a warning:

```
>>> class DerivedOldClass(OldClass):
...     pass
WARNING: DEPRECATED: Declaring class 'DerivedOldClass' derived from
    'OldClass'. The class 'OldClass' has been renamed to 'NewClass'.
    (deprecated in 6.0) ...
```

As does instantiating the old class:

```
>>> old = OldClass()
WARNING: DEPRECATED: Instantiating class 'OldClass'.  The class
    'OldClass' has been renamed to 'NewClass'.  (deprecated in 6.0) ...
```

Finally, *isinstance* and *issubclass* still work, for example:

```
>>> isinstance(old, NewClass)
True
>>> class NewSubclass(NewClass):
...     pass
>>> new = NewSubclass()
>>> isinstance(new, OldClass)
WARNING: DEPRECATED: Checking type relative to 'OldClass'.  The class
    'OldClass' has been renamed to 'NewClass'.  (deprecated in 6.0) ...
True
```

3.2.2 Removal

By policy, functionality should be deprecated with reasonable warning, pending extenuating circumstances. The functionality should be deprecated, following the information above.

If the functionality is documented in the most recent edition of [Pyomo - Optimization Modeling in Python], it may not be removed until the next major version release.

For other functionality, it is preferred that ample time is given before removing the functionality. At minimum, significant functionality removal will result in a minor version bump.

3.3 Pyomo Expressions

Warning: This documentation does not explicitly reference objects in pyomo.core.kernel. While the Pyomo5 expression system works with pyomo.core.kernel objects, the documentation of these documents was not sufficient to appropriately describe the use of kernel objects in expressions.

Pyomo supports the declaration of symbolic expressions that represent objectives, constraints and other optimization modeling components. Pyomo expressions are represented in an expression tree, where the leaves are operands, such as constants or variables, and the internal nodes contain operators. Pyomo relies on so-called magic methods to automate the construction of symbolic expressions. For example, consider an expression `e` declared as follows:

Python determines that the magic method `__mul__` is called on the `M.v` object, with the argument `2`. This method returns a Pyomo expression object `ProductExpression` that has arguments `M.v` and `2`. This represents the following symbolic expression tree:

Note: End-users will not likely need to know details related to how symbolic expressions are generated and managed in Pyomo. Thus, most of the following documentation of expressions in Pyomo is most useful for Pyomo developers. However, the discussion of runtime performance in the first section will help end-users write large-scale models.

3.3.1 Building Expressions Faster

Expression Generation

Pyomo expressions can be constructed using native binary operators in Python. For example, a sum can be created in a simple loop:

Additionally, Pyomo expressions can be constructed using functions that iteratively apply Python binary operators. For example, the Python `sum()` function can be used to replace the previous loop:

The `sum()` function is both more compact and more efficient. Using `sum()` avoids the creation of temporary variables, and the summation logic is executed in the Python interpreter while the loop is interpreted.

Linear, Quadratic and General Nonlinear Expressions

Pyomo can express a very wide range of algebraic expressions, and there are three general classes of expressions that are recognized by Pyomo:

- **linear polynomials**
- **quadratic polynomials**
- **nonlinear expressions**, including higher-order polynomials and expressions with intrinsic functions

These classes of expressions are leveraged to efficiently generate compact representations of expressions, and to transform expression trees into standard forms used to interface with solvers. Note that There not all quadratic polynomials are recognized by Pyomo; in other words, some quadratic expressions are treated as nonlinear expressions.

For example, consider the following quadratic polynomial:

This quadratic polynomial is treated as a nonlinear expression unless the expression is explicitly processed to identify quadratic terms. This *lazy* identification of of quadratic terms allows Pyomo to tailor the search for quadratic terms only when they are explicitly needed.

Pyomo Utility Functions

Pyomo includes several similar functions that can be used to create expressions:

`prod`
: A function to compute a product of Pyomo expressions.

`quicksum`
: A function to efficiently compute a sum of Pyomo expressions.

`sum_product`
: A function that computes a generalized dot product.

prod

The `prod` function is analogous to the builtin `sum()` function. Its main argument is a variable length argument list, `args`, which represents expressions that are multiplied together. For example:

quicksum

The behavior of the `quicksum` function is similar to the builtin `sum()` function, but this function often generates a more compact Pyomo expression. Its main argument is a variable length argument list, `args`, which represents expressions that are summed together. For example:

The summation is customized based on the `start` and `linear` arguments. The `start` defines the initial value for summation, which defaults to zero. If `start` is a numeric value, then the `linear` argument determines how the sum is processed:

- If `linear` is `False`, then the terms in `args` are assumed to be nonlinear.
- If `linear` is `True`, then the terms in `args` are assumed to be linear.
- If `linear` is `None`, the first term in `args` is analyze to determine whether the terms are linear or nonlinear.

This argument allows the `quicksum` function to customize the expression representation used, and specifically a more compact representation is used for linear polynomials. The `quicksum` function can be slower than the builtin `sum()` function, but this compact representation can generate problem representations more quickly.

Consider the following example:

The sum consists of linear terms because the exponents are one. The following output illustrates that quicksum can identify this linear structure to generate expressions more quickly:

If `start` is not a numeric value, then the `quicksum` sets the initial value to `start` and executes a simple loop to sum the terms. This allows the sum to be stored in an object that is passed into the function (e.g. the linear context manager `linear_expression`).

> **Warning:** By default, `linear` is `None`. While this allows for efficient expression generation in normal cases, there are circumstances where the inspection of the first term in `args` is misleading. Consider the following example:
>
> The first term created by the generator is linear, but the subsequent terms are nonlinear. Pyomo gracefully transitions to a nonlinear sum, but in this case `quicksum` is doing additional work that is not useful.

sum_product

The `sum_product` function supports a generalized dot product. The `args` argument contains one or more components that are used to create terms in the summation. If the `args` argument contains a single components, then its sequence of terms are summed together; the sum is equivalent to calling `quicksum`. If two or more components are provided, then the result is the summation of their terms multiplied together. For example:

The `denom` argument specifies components whose terms are in the denominator. For example:

The terms summed by this function are explicitly specified, so `sum_product` can identify whether the resulting expression is linear, quadratic or nonlinear. Consequently, this function is typically faster than simple loops, and it generates compact representations of expressions..

Finally, note that the `dot_product` function is an alias for `sum_product`.

3.3.2 Design Overview

Historical Comparison

This document describes the "Pyomo5" expressions, which were introduced in Pyomo 5.6. The main differences between "Pyomo5" expressions and the previous expression system, called "Coopr3", are:

- Pyomo5 supports both CPython and PyPy implementations of Python, while Coopr3 only supports CPython.

  The key difference in these implementations is that Coopr3 relies on CPython reference counting, which is not part of the Python language standard. Hence, this implementation is not guaranteed to run on other implementations of Python.

  Pyomo5 does not rely on reference counting, and it has been tested with PyPy. In the future, this should allow Pyomo to support other Python implementations (e.g. Jython).

- Pyomo5 expression objects are immutable, while Coopr3 expression objects are mutable.

  This difference relates to how expression objects are managed in Pyomo. Once created, Pyomo5 expression objects cannot be changed. Further, the user is guaranteed that no "side effects" occur when expressions change at a later point in time. By contrast, Coopr3 allows expressions to change in-place, and thus "side effects" make occur when expressions are changed at a later point in time. (See discussion of entanglement below.)

- Pyomo5 provides more consistent runtime performance than Coopr3.

  While this documentation does not provide a detailed comparison of runtime performance between Coopr3 and Pyomo5, the following performance considerations also motivated the creation of Pyomo5:

  - There were surprising performance inconsistencies in Coopr3. For example, the following two loops had dramatically different runtime:

  - Coopr3 eliminates side effects by automatically cloning sub-expressions. Unfortunately, this can easily lead to unexpected cloning in models, which can dramatically slow down Pyomo model generation. For example:

  - Coopr3 leverages recursion in many operations, including expression cloning. Even simple non-linear expressions can result in deep expression trees where these recursive operations fail because Python runs out of stack space.

  - The immutable representation used in Pyomo5 requires more memory allocations than Coopr3 in simple loops. Hence, a pure-Python execution of Pyomo5 can be 10% slower than Coopr3 for model construction. But when Cython is used to optimize the execution of Pyomo5 expression generation, the runtimes for Pyomo5 and Coopr3 are about the same. (In principle, Cython would improve the runtime of Coopr3 as well, but the limitations noted above motivated a new expression system in any case.)

Expression Entanglement and Mutability

Pyomo fundamentally relies on the use of magic methods in Python to generate expression trees, which means that Pyomo has very limited control for how expressions are managed in Python. For example:

- Python variables can point to the same expression tree

  This is illustrated as follows:

- A variable can point to a sub-tree that another variable points to

  This is illustrated as follows:

- Two expression trees can point to the same sub-tree

  This is illustrated as follows:

In each of these examples, it is almost impossible for a Pyomo user or developer to detect whether expressions are being shared. In CPython, the reference counting logic can support this to a limited degree. But no equivalent mechanisms are available in PyPy and other Python implementations.

Entangled Sub-Expressions

We say that expressions are *entangled* if they share one or more sub-expressions. The first example above does not represent entanglement, but rather the fact that multiple Python variables can point to the same expression tree. In the second and third examples, the expressions are entangled because the subtree represented by `e` is shared. However, if a leave node like `M.v` is shared between expressions, we do not consider those expressions entangled.

Expression entanglement is problematic because shared expressions complicate the expected behavior when sub-expressions are changed. Consider the following example:

What is the value of `e` after `M.w` is added to it? What is the value of `f`? The answers to these questions are not immediately obvious, and the fact that Coopr3 uses mutable expression objects makes them even less clear. However, Pyomo5 and Coopr3 enforce the following semantics:

> A change to an expression e that is a sub-expression of f does not change the expression tree for f.

This property ensures a change to an expression does not create side effects that change the values of other, previously defined expressions.

For instance, the previous example results in the following (in Pyomo5):

With Pyomo5 expressions, each sub-expression is immutable. Thus, the summation operation generates a new expression `e` without changing existing expression objects referenced in the expression tree for `f`. By contrast, Coopr3 imposes the same property by cloning the expression `e` before added `M.w`, resulting in the following:

This example also illustrates that leaves may be shared between expressions.

Mutable Expression Components

There is one important exception to the entanglement property described above. The `Expression` component is treated as a mutable expression when shared between expressions. For example:

Here, the expression `M.e` is a so-called *named expression* that the user has declared. Named expressions are explicitly intended for re-use within models, and they provide a convenient mechanism for changing sub-expressions in complex applications. In this example, the expression tree is as follows before `M.w` is added:

And the expression tree is as follows after `M.w` is added.

When considering named expressions, Pyomo5 and Coopr3 enforce the following semantics:

> A change to a named expression e that is a sub-expression of f changes the expression tree for f, because f continues to point to e after it is changed.

3.3.3 Design Details

Warning: Pyomo expression trees are not composed of Python objects from a single class hierarchy. Consequently, Pyomo relies on duck typing to ensure that valid expression trees are created.

Most Pyomo expression trees have the following form

1. Interior nodes are objects that inherit from the `ExpressionBase` class. These objects typically have one or more child nodes. Linear expression nodes do not have child nodes, but they are treated as interior nodes in the expression tree because they references other leaf nodes.
2. Leaf nodes are numeric values, parameter components and variable components, which represent the *inputs* to the expression.

Expression Classes

Expression classes typically represent unary and binary operations. The following table describes the standard operators in Python and their associated Pyomo expression class:

Operation	Python Syntax	Pyomo Class
sum	`x + y`	`SumExpression`
product	`x * y`	`ProductExpression`
negation	`- x`	`NegationExpression`
division	`x / y`	`DivisionExpression`
power	`x ** y`	`PowExpression`
inequality	`x <= y`	`InequalityExpression`
equality	`x == y`	`EqualityExpression`

Additionally, there are a variety of other Pyomo expression classes that capture more general logical relationships, which are summarized in the following table:

Operation	Example	Pyomo Class
external function	`myfunc(x,y,z)`	`ExternalFunctionExpression`
logical if-then-else	`Expr_if(IF=x, THEN=y, ELSE=z)`	`Expr_ifExpression`
intrinsic function	`sin(x)`	`UnaryFunctionExpression`
absolute function	`abs(x)`	`AbsExpression`

Expression objects are immutable. Specifically, the list of arguments to an expression object (a.k.a. the list of child nodes in the tree) cannot be changed after an expression class is constructed. To enforce this property, expression objects have a standard API for accessing expression arguments:

- `args` - a class property that returns a generator that yields the expression arguments
- `arg(i)` - a function that returns the `i`-th argument
- `nargs()` - a function that returns the number of expression arguments

Warning: Developers should never use the `_args_` property directly! The semantics for the use of this data has changed since earlier versions of Pyomo. For example, in some expression classes the the value `nargs()` may not equal `len(_args_)`!

Expression trees can be categorized in four different ways:

- constant expressions - expressions that do not contain numeric constants and immutable parameters.
- mutable expressions - expressions that contain mutable parameters but no variables.
- potentially variable expressions - expressions that contain variables, which may be fixed.
- fixed expressions - expressions that contain variables, all of which are fixed.

These three categories are illustrated with the following example:

The following table describes four different simple expressions that consist of a single model component, and it shows how they are categorized:

Category	m.p	m.q	m.x	m.y
constant	True	False	False	False
not potentially variable	True	True	False	False
potentially_variable	False	False	True	True
fixed	True	True	False	True

Expressions classes contain methods to test whether an expression tree is in each of these categories. Additionally, Pyomo includes custom expression classes for expression trees that are *not potentially variable*. These custom classes will not normally be used by developers, but they provide an optimization of the checks for potentially variability.

Special Expression Classes

The following classes are *exceptions* to the design principles describe above.

Named Expressions

Named expressions allow for changes to an expression after it has been constructed. For example, consider the expression `f` defined with the `Expression` component:

Although `f` is an immutable expression, whose definition is fixed, a sub-expressions is the named expression `M.e`. Named expressions have a mutable value. In other words, the expression that they point to can change. Thus, a change to the value of `M.e` changes the expression tree for any expression that includes the named expression.

Note: The named expression classes are not implemented as sub-classes of `NumericExpression`. This reflects design constraints related to the fact that these are modeling components that belong to class hierarchies other than the expression class hierarchy, and Pyomo's design prohibits the use of multiple inheritance for these classes.

Linear Expressions

Pyomo includes a special expression class for linear expressions. The class `LinearExpression` provides a compact description of linear polynomials. Specifically, it includes a constant value `constant` and two lists for coefficients and variables: `linear_coefs` and `linear_vars`.

This expression object does not have arguments, and thus it is treated as a leaf node by Pyomo visitor classes. Further, the expression API functions described above do not work with this class. Thus, developers need to treat this class differently when walking an expression tree (e.g. when developing a problem transformation).

Sum Expressions

Pyomo does not have a binary sum expression class. Instead, it has an `n`-ary summation class, `SumExpression`. This expression class treats sums as `n`-ary sums for efficiency reasons; many large optimization models contain large sums. But note that this class maintains the immutability property described above. This class shares an underlying list of arguments with other `SumExpression` objects. A particular object owns the first `n` arguments in the shared list, but different objects may have different values of `n`.

This class acts like a normal immutable expression class, and the API described above works normally. But direct access to the shared list could have unexpected results.

Mutable Expressions

Finally, Pyomo includes several **mutable** expression classes that are private. These are not intended to be used by users, but they might be useful for developers in contexts where the developer can appropriately control how the classes are used. Specifically, immutability eliminates side-effects where changes to a sub-expression unexpectedly create changes to the expression tree. But within the context of model transformations, developers may be able to limit the use of expressions to avoid these side-effects. The following mutable private classes are available in Pyomo:

`_MutableSumExpression`
: This class is used in the `nonlinear_expression` context manager to efficiently combine sums of nonlinear terms.

`_MutableLinearExpression`
: This class is used in the `linear_expression` context manager to efficiently combine sums of linear terms.

Expression Semantics

Pyomo clear semantics regarding what is considered a valid leaf and interior node.

The following classes are valid interior nodes:

- Subclasses of `ExpressionBase`
- Classes that that are *duck typed* to match the API of the `ExpressionBase` class. For example, the named expression class `Expression`.

The following classes are valid leaf nodes:

- Members of `nonpyomo_leaf_types`, which includes standard numeric data types like `int`, `float` and `long`, as well as numeric data types defined by *numpy* and other commonly used packages. This set also includes `NonNumericValue`, which is used to wrap non-numeric arguments to the `ExternalFunctionExpression` class.
- Parameter component classes like `ScalarParam` and `_ParamData`, which arise in expression trees when the parameters are declared as mutable. (Immutable parameters are identified when generating expressions, and they are replaced with their associated numeric value.)
- Variable component classes like `ScalarVar` and `_GeneralVarData`, which often arise in expression trees. <pyomo.core.expr.pyomo5_variable_types>`.

Note: In some contexts the `LinearExpression` class can be treated as an interior node, and sometimes it can be treated as a leaf. This expression object does not have any child arguments, so `nargs()` is zero. But this expression references variables and parameters in a linear expression, so in that sense it does not represent a leaf node in the tree.

Context Managers

Pyomo defines several context managers that can be used to declare the form of expressions, and to define a mutable expression object that efficiently manages sums.

The `linear_expression` object is a context manager that can be used to declare a linear sum. For example, consider the following two loops:

The first apparent difference in these loops is that the value of `s` is explicitly initialized while `e` is initialized when the context manager is entered. However, a more fundamental difference is that the expression representation for `s` differs from `e`. Each term added to `s` results in a new, immutable expression. By contrast, the context manager creates a mutable expression representation for `e`. This difference allows for both (a) a more efficient processing of each sum, and (b) a more compact representation for the expression.

The difference between `linear_expression` and `nonlinear_expression` is the underlying representation that each supports. Note that both of these are instances of context manager classes. In singled-threaded applications, these objects can be safely used to construct different expressions with different context declarations.

Finally, note that these context managers can be passed into the `start` method for the `quicksum` function. For example:

This sum contains terms for `M.x[i]` and `M.y[i]`. The syntax in this example is not intuitive because the sum is being stored in `e`.

Note: We do not generally expect users or developers to use these context managers. They are used by the `quicksum` and `sum_product` functions to accelerate expression generation, and there are few cases where the direct use of these context managers would provide additional utility to users and developers.

3.3.4 Managing Expressions

Creating a String Representation of an Expression

There are several ways that string representations can be created from an expression, but the `expression_to_string` function provides the most flexible mechanism for generating a string representation. The options to this function control distinct aspects of the string representation.

Algebraic vs. Nested Functional Form

The default string representation is an algebraic form, which closely mimics the Python operations used to construct an expression. The `verbose` flag can be set to `True` to generate a string representation that is a nested functional form. For example:

Labeler and Symbol Map

The string representation used for variables in expression can be customized to define different label formats. If the `labeler` option is specified, then this function (or class functor) is used to generate a string label used to represent the variable. Pyomo defines a variety of labelers in the *pyomo.core.base.label* module. For example, the `NumericLabeler` defines a functor that can be used to sequentially generate simple labels with a prefix followed by the variable count:

The `smap` option is used to specify a symbol map object (`SymbolMap`), which caches the variable label data. This option is normally specified in contexts where the string representations for many expressions are being generated. In that context, a symbol map ensures that variables in different expressions have a consistent label in their associated string representations.

Standardized String Representations

The `standardize` option can be used to re-order the string representation to print polynomial terms before nonlinear terms. By default, `standardize` is `False`, and the string representation reflects the order in which terms were combined to form the expression. Pyomo does not guarantee that the string representation exactly matches the Python expression order, since some simplification and re-ordering of terms is done automatically to improve the efficiency of expression generation. But in most cases the string representation will closely correspond to the Python expression order.

If `standardize` is `True`, then the pyomo expression is processed to identify polynomial terms, and the string representation consists of the constant and linear terms followed by an expression that contains other nonlinear terms. For example:

Other Ways to Generate String Representations

There are two other standard ways to generate string representations:

- Call the `__str__()` magic method (e.g. using the Python `str()` function. This calls `expression_to_string` with the option `standardize` equal to `True` (see below).
- Call the `to_string()` method on the `ExpressionBase` class. This defaults to calling `expression_to_string` with the option `standardize` equal to `False` (see below).

In practice, we expect at the `__str__()` magic method will be used by most users, and the standardization of the output provides a consistent ordering of terms that should make it easier to interpret expressions.

Cloning Expressions

Expressions are automatically cloned only during certain expression transformations. Since this can be an expensive operation, the `clone_counter` context manager object is provided to track the number of times the `clone_expression` function is executed.

For example:

Evaluating Expressions

Expressions can be evaluated when all variables and parameters in the expression have a value. The `value` function can be used to walk the expression tree and compute the value of an expression. For example:

Additionally, expressions define the `__call__()` method, so the following is another way to compute the value of an expression:

If a parameter or variable is undefined, then the `value` function and `__call__()` method will raise an exception. This exception can be suppressed using the `exception` option. For example:

This option is useful in contexts where adding a try block is inconvenient in your modeling script.

Note: Both the `value` function and `__call__()` method call the `evaluate_expression` function. In practice, this function will be slightly faster, but the difference is only meaningful when expressions are evaluated many times.

Identifying Components and Variables

Expression transformations sometimes need to find all nodes in an expression tree that are of a given type. Pyomo contains two utility functions that support this functionality. First, the `identify_components` function is a generator function that walks the expression tree and yields all nodes whose type is in a specified set of node types. For example:

The `identify_variables` function is a generator function that yields all nodes that are variables. Pyomo uses several different classes to represent variables, but this set of variable types does not need to be specified by the user. However, the `include_fixed` flag can be specified to omit fixed variables. For example:

Walking an Expression Tree with a Visitor Class

Many of the utility functions defined above are implemented by walking an expression tree and performing an operation at nodes in the tree. For example, evaluating an expression is performed using a post-order depth-first search process where the value of a node is computed using the values of its children.

Walking an expression tree can be tricky, and the code requires intimate knowledge of the design of the expression system. Pyomo includes several classes that define so-called visitor patterns for walking expression tree:

`SimpleExpressionVisitor`
: A `visitor()` method is called for each node in the tree, and the visitor class collects information about the tree.

`ExpressionValueVisitor`
: When the `visitor()` method is called on each node in the tree, the *values* of its children have been computed. The *value* of the node is returned from `visitor()`.

`ExpressionReplacementVisitor`
: When the `visitor()` method is called on each node in the tree, it may clone or otherwise replace the node using objects for its children (which themselves may be clones or replacements from the original child objects). The new node object is returned from `visitor()`.

These classes define a variety of suitable tree search methods:

- `SimpleExpressionVisitor`
  - **xbfs**: breadth-first search where leaf nodes are immediately visited
  - **xbfs_yield_leaves**: breadth-first search where leaf nodes are immediately visited, and the visit method yields a value
- `ExpressionValueVisitor`
  - **dfs_postorder_stack**: postorder depth-first search using a stack
- `ExpressionReplacementVisitor`
  - **dfs_postorder_stack**: postorder depth-first search using a stack

---

Note: The PyUtilib visitor classes define several other search methods that could be used with Pyomo expressions. But these are the only search methods currently used within Pyomo.

To implement a visitor object, a user creates a subclass of one of these classes. Only one of a few methods will need to be defined to implement the visitor:

`visitor()`
: Defines the operation that is performed when a node is visited. In the `ExpressionValueVisitor` and `ExpressionReplacementVisitor` visitor classes, this method returns a value that is used by its parent node.

`visiting_potential_leaf()`
: Checks if the search should terminate with this node. If no, then this method returns the tuple `(False, None)`. If yes, then this method returns `(False, value)`, where *value* is computed by this method. This method is not used in the `SimpleExpressionVisitor` visitor class.

`finalize()`
: This method defines the final value that is returned from the visitor. This is not normally redefined.

Detailed documentation of the APIs for these methods is provided with the class documentation for these visitors.

SimpleExpressionVisitor Example

In this example, we describe an visitor class that counts the number of nodes in an expression (including leaf nodes). Consider the following class:

The class constructor creates a counter, and the `visit()` method increments this counter for every node that is visited. The `finalize()` method returns the value of this counter after the tree has been walked. The following function illustrates this use of this visitor class:

ExpressionValueVisitor Example

In this example, we describe an visitor class that clones the expression tree (including leaf nodes). Consider the following class:

The `visit()` method creates a new expression node with children specified by `values`. The `visiting_potential_leaf()` method performs a `deepcopy()` on leaf nodes, which are native Python types or non-expression objects.

ExpressionReplacementVisitor Example

In this example, we describe an visitor class that replaces variables with scaled variables, using a mutable parameter that can be modified later. the following class:

No `visit()` method needs to be defined. The `visiting_potential_leaf()` function identifies variable nodes and returns a product expression that contains a mutable parameter. The `_LinearExpression` class has a different representation that embeds variables. Hence, this class must be handled in a separate condition that explicitly transforms this sub-expression.

The `scale_expression()` function is called with an expression and a dictionary, `scale`, that maps variable ID to model parameter. For example:

CHAPTER

FOUR

REFERENCE GUIDE

4.1 Bibliography

Bibliography

CHAPTER

FIVE

PYOMO RESOURCES

Pyomo development is hosted at GitHub:

- https://github.com/Pyomo/pyomo

See the Pyomo Forum for online discussions of Pyomo or to ask a question:

- http://groups.google.com/group/pyomo-forum/

Ask a question on StackOverflow using the *#pyomo* tag:

- https://stackoverflow.com/questions/ask?tags=pyomo

CHAPTER

SIX

CONTRIBUTING TO PYOMO

Interested in contributing code or documentation to the project? Check out our Contribution Guide

CHAPTER

SEVEN

RELATED PACKAGES

Pyomo is a key dependency for a number of other software packages for specific domains or customized solution strategies. A non-comprehensive list of Pyomo-related packages may be found here.

CHAPTER

EIGHT

CITING PYOMO

Bynum, Michael L., Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. Pyomo - Optimization Modeling in Python, 3rd Edition. Springer, 2021.

Hart, William E., Jean-Paul Watson, and David L. Woodruff. "Pyomo: modeling and solving mathematical programs in Python." Mathematical Programming Computation 3, no. 3 (2011): 219-260.